**LOW LEVEL DESIGN**

DESIGN PATTERN

**✅ What is a Design Pattern?**

A **Design Pattern** is a **proven, reusable solution** to a **common problem** in software design. It’s not code itself, but a **template** on how to solve a design problem in various situations.

💡 Think of them like architectural blueprints – solutions you can customize to fit a particular need.

**🧱 Why Use Design Patterns in Low-Level Design?**

Low-Level Design (LLD) focuses on **class design, relationships, and implementation** details. Using patterns here helps:

* Promote **code reusability**
* Ensure **scalability**
* Support **readability and maintainability**
* Encourage **best practices** and standardization

**🧩 Types of Design Patterns (GoF Classification)**

Design Patterns are mainly classified into **3 categories**:

| **Type** | **Description** | **Example Patterns** |
| --- | --- | --- |
| **Creational** | Deal with object creation | Singleton, Factory, Builder, Prototype |
| **Structural** | Deal with object composition | Adapter, Decorator, Proxy, Facade |
| **Behavioral** | Deal with object interaction | Observer, Strategy, Command, State |

**✅ 1. Creational Patterns**

**🔹 Singleton Pattern**

**Intent:** Ensure a class has only one instance and provide a global access point.

**Real-world analogy:** Government has **one president**. Any action must go through that instance.

**Java Example:**

public class Singleton {

private static Singleton instance;

private Singleton() {}

public static Singleton getInstance() {

if (instance == null) {

instance = new Singleton(); // Lazy initialization

}

return instance;

}

}

**🔹 Factory Pattern**

**Intent:** Create objects **without exposing the creation logic** to the client.

**Real-world analogy:** A car factory makes SUV, Sedan, or Truck depending on the input, without the user knowing how it’s made.

**Java Example:**

interface Vehicle {

void drive();

}

class Car implements Vehicle {

public void drive() { System.out.println("Driving Car"); }

}

class Bike implements Vehicle {

public void drive() { System.out.println("Riding Bike"); }

}

class VehicleFactory {

public static Vehicle getVehicle(String type) {

return type.equals("car") ? new Car() : new Bike();

}

}

**🔹 Builder Pattern**

**Intent:** Construct a complex object step by step.

**Real-world analogy:** Building a burger with choices: bun, patty, sauce.

**Java Example:**

class Burger {

private String bun, patty, sauce;

public static class Builder {

private String bun, patty, sauce;

public Builder setBun(String bun) { this.bun = bun; return this; }

public Builder setPatty(String patty) { this.patty = patty; return this; }

public Builder setSauce(String sauce) { this.sauce = sauce; return this; }

public Burger build() {

return new Burger(bun, patty, sauce);

}

}

private Burger(String bun, String patty, String sauce) {

this.bun = bun; this.patty = patty; this.sauce = sauce;

}

}

**✅ 2. Structural Patterns**

**🔹 Adapter Pattern**

**Intent:** Convert the interface of a class into another interface the client expects.

**Real-world analogy:** Power adapter between 3-pin socket and 2-pin plug.

**Java Example:**

interface MediaPlayer {

void play(String fileType, String fileName);

}

class MP3Player implements MediaPlayer {

public void play(String fileType, String fileName) {

System.out.println("Playing MP3: " + fileName);

}

}

class MP4Player {

public void playMP4(String fileName) {

System.out.println("Playing MP4: " + fileName);

}

}

class MediaAdapter implements MediaPlayer {

private MP4Player mp4Player = new MP4Player();

public void play(String fileType, String fileName) {

if (fileType.equalsIgnoreCase("mp4"))

mp4Player.playMP4(fileName);

}

}

**🔹 Decorator Pattern**

**Intent:** Add behavior to objects dynamically.

**Real-world analogy:** Adding toppings to a pizza.

**Java Example:**

interface Coffee {

String getDescription();

int getCost();

}

class SimpleCoffee implements Coffee {

public String getDescription() { return "Simple Coffee"; }

public int getCost() { return 5; }

}

class MilkDecorator implements Coffee {

private Coffee coffee;

public MilkDecorator(Coffee c) { coffee = c; }

public String getDescription() { return coffee.getDescription() + ", Milk"; }

public int getCost() { return coffee.getCost() + 2; }

}

**✅ 3. Behavioral Patterns**

**🔹 Observer Pattern**

**Intent:** When one object changes, its dependents are notified automatically.

**Real-world analogy:** You subscribe to a YouTube channel; when a video is uploaded, you're notified.

**Java Example:**

interface Observer {

void update(String news);

}

class Subscriber implements Observer {

private String name;

public Subscriber(String name) { this.name = name; }

public void update(String news) {

System.out.println(name + " received: " + news);

}

}

class Channel {

private List<Observer> subs = new ArrayList<>();

public void subscribe(Observer o) { subs.add(o); }

public void notifySubscribers(String news) {

for (Observer o : subs) o.update(news);

}

}

**🔹 Strategy Pattern**

**Intent:** Define a family of algorithms and make them interchangeable at runtime.

**Real-world analogy:** Payment strategy (Card, UPI, Cash).

**Java Example:**

interface PaymentStrategy {

void pay(int amount);

}

class CardPayment implements PaymentStrategy {

public void pay(int amount) {

System.out.println("Paid ₹" + amount + " by Card");

}

}

class CashPayment implements PaymentStrategy {

public void pay(int amount) {

System.out.println("Paid ₹" + amount + " by Cash");

}

}

class ShoppingCart {

private PaymentStrategy strategy;

public ShoppingCart(PaymentStrategy strategy) { this.strategy = strategy; }

public void checkout(int amount) {

strategy.pay(amount);

}

}

**✅ When to Use Which Pattern? (Cheat Sheet)**

| **Situation** | **Pattern** |
| --- | --- |
| Need a single instance of a class | Singleton |
| Hide creation logic | Factory |
| Build object step by step | Builder |
| Want to adapt one interface to another | Adapter |
| Add responsibilities dynamically | Decorator |
| Notify many objects of changes | Observer |
| Switch algorithms at runtime | Strategy |

**✅ Real-World LLD Use Cases Using Patterns**

| **Use Case** | **Pattern** |
| --- | --- |
| Logger system | Singleton |
| UI Framework (e.g., adding scrollbars) | Decorator |
| Notification system (email/SMS) | Strategy + Observer |
| Payment gateways | Factory + Strategy |
| API response parser (JSON/XML) | Adapter |

**✅ Tips for Using Design Patterns**

* **Don’t overuse them.** Simplicity > cleverness.
* Understand the **problem** first. Then pick a pattern.
* Learn from **existing frameworks** like Spring, Hibernate – they use patterns heavily.
* Document your design pattern usage in LLD diagrams.

**✅ How to Practice?**

1. **Pick a real-world system** (e.g., Food Delivery App, Ride Booking).
2. Break into modules (User, Payment, Order).
3. Identify where patterns can help (Strategy in Payment, Observer in Notifications).
4. Implement in Java step-by-step.

Would you like a **real-world LLD example** of a complete module using 3-4 patterns? Or a **PDF/cheat sheet** version of this for offline reading?

Creational design patterns are a category of design patterns in **Low Level Design (LLD)** that **deal with object creation mechanisms**. These patterns provide various ways to create objects, increasing flexibility and reuse of existing code.

**✅ Definition of Creational Design Pattern**

Creational patterns abstract the instantiation process. They help **make a system independent of how its objects are created, composed, and represented.**

They are particularly useful when:

* The object creation logic is complex
* You want to decouple code from specific classes (to make the system extensible)
* You need to manage or restrict object creation (e.g., singleton)

**🔷 Types of Creational Design Patterns**

1. **Singleton Pattern**
2. **Factory Method Pattern**
3. **Abstract Factory Pattern**
4. **Builder Pattern**
5. **Prototype Pattern**

**1. 🟡 Singleton Pattern**

**👉 Intent:**

Ensure that a class has **only one instance** and provides a global access point to it.

**🔧 Use Case:**

* Database connection manager
* Configuration loader
* Logging service

**🔤 Java Code:**

public class Logger {

private static Logger instance;

private Logger() {

// private constructor

}

public static Logger getInstance() {

if (instance == null) {

instance = new Logger();

}

return instance;

}

public void log(String msg) {

System.out.println("Log: " + msg);

}

}

**✅ Real World Analogy:**

* **Electricity board (only one control room for a city)**

**2. 🟢 Factory Method Pattern**

**👉 Intent:**

Define an interface for creating an object, but **let subclasses decide which class to instantiate.**

**🔧 Use Case:**

* Shape creation (circle, square, triangle) based on input
* Notification system (SMS, Email, Push)

**🔤 Java Code:**

interface Notification {

void notifyUser();

}

class SMSNotification implements Notification {

public void notifyUser() {

System.out.println("Sending SMS Notification");

}

}

class EmailNotification implements Notification {

public void notifyUser() {

System.out.println("Sending Email Notification");

}

}

class NotificationFactory {

public Notification createNotification(String type) {

if (type.equalsIgnoreCase("SMS")) return new SMSNotification();

else if (type.equalsIgnoreCase("EMAIL")) return new EmailNotification();

return null;

}

}

**✅ Real World Analogy:**

* **Food Delivery App**: Select delivery partner (Zomato or Swiggy) at runtime

**3. 🔵 Abstract Factory Pattern**

**👉 Intent:**

Provide an interface for creating **families of related or dependent objects** without specifying their concrete classes.

**🔧 Use Case:**

* UI toolkit: create buttons, scrollbars for **different OS** (Windows, Mac, Linux)
* Theme factory (Dark vs Light themes)

**🔤 Java Code:**

interface Button {

void click();

}

class WindowsButton implements Button {

public void click() {

System.out.println("Windows Button clicked");

}

}

class MacButton implements Button {

public void click() {

System.out.println("Mac Button clicked");

}

}

interface GUIFactory {

Button createButton();

}

class WindowsFactory implements GUIFactory {

public Button createButton() {

return new WindowsButton();

}

}

class MacFactory implements GUIFactory {

public Button createButton() {

return new MacButton();

}

}

**✅ Real World Analogy:**

* **Car Manufacturer**: Factory for petrol or electric car components

**4. 🟠 Builder Pattern**

**👉 Intent:**

Separate the construction of a complex object from its representation so that the same construction process can create different representations.

**🔧 Use Case:**

* Creating complex objects like House, Pizza, Computer step-by-step
* Object with many optional parameters

**🔤 Java Code:**

class Pizza {

private String dough;

private String sauce;

private String topping;

private Pizza(Builder builder) {

this.dough = builder.dough;

this.sauce = builder.sauce;

this.topping = builder.topping;

}

public static class Builder {

private String dough;

private String sauce;

private String topping;

public Builder setDough(String dough) {

this.dough = dough;

return this;

}

public Builder setSauce(String sauce) {

this.sauce = sauce;

return this;

}

public Builder setTopping(String topping) {

this.topping = topping;

return this;

}

public Pizza build() {

return new Pizza(this);

}

}

public void display() {

System.out.println("Pizza with " + dough + ", " + sauce + ", " + topping);

}

}

**✅ Real World Analogy:**

* **Burger King**: You build your burger step-by-step

**5. 🔴 Prototype Pattern**

**👉 Intent:**

Create new objects by **copying an existing object** (clone), instead of creating from scratch.

**🔧 Use Case:**

* Object creation is expensive
* You want to avoid subclassing or complex instantiation

**🔤 Java Code:**

interface Prototype extends Cloneable {

Prototype clone();

}

class Document implements Prototype {

private String content;

public Document(String content) {

this.content = content;

}

public Prototype clone() {

return new Document(this.content);

}

public void print() {

System.out.println("Document: " + content);

}

}

**✅ Real World Analogy:**

* **Resume cloning**: Copying a resume and editing personal details

**🧠 Summary Table**

| **Pattern** | **Use Case** | **Key Benefit** |
| --- | --- | --- |
| Singleton | One and only one instance needed | Global access, controlled creation |
| Factory Method | Create objects based on input | Decouples client from instantiation |
| Abstract Factory | Family of related objects | Ensure consistency across families |
| Builder | Complex object with many steps | Step-by-step construction |
| Prototype | Copy existing object | Performance boost, avoids re-creation |

**🚀 Real World Use Case: Ride-Sharing App (e.g., Uber/Ola)**

| **Scenario** | **Pattern** | **Explanation** |
| --- | --- | --- |
| Single configuration for app-wide settings | Singleton | App settings loaded once |
| Creating different vehicle types (Car/Bike) | Factory Method | Client provides type, factory returns correct object |
| Platform theme across devices | Abstract Factory | Family of UI components per OS |
| User profile creation with optional fields | Builder | Email, photo, phone - selectively set |
| Cloning trip templates for repeat rides | Prototype | Copy ride settings without rebuilding |

If you'd like, I can also generate a **UML diagram** or **convert these into design notes/slides** for easy revision. Let me know!

**Structural Design Patterns in Low-Level Design (LLD)**

**🔷 Definition:**

**Structural Design Patterns** are design patterns that **simplify the structure of complex systems by identifying relationships** among components. They **focus on how classes and objects are composed** to form larger structures. The intent is to **create flexible and efficient structures** by controlling how objects interact.

These patterns help ensure that **if one part of a system changes**, the rest of the system **does not break** due to strong coupling.

**📦 Types of Structural Design Patterns:**

| **Pattern** | **Purpose** |
| --- | --- |
| Adapter | Bridge interface incompatibility |
| Bridge | Decouple abstraction from implementation |
| Composite | Treat individual and group objects uniformly |
| Decorator | Add responsibilities dynamically |
| Facade | Provide a simplified interface |
| Flyweight | Share objects to support large numbers efficiently |
| Proxy | Control access to another object |

**✅ 1. Adapter Pattern**

**👉 Intent:**

Convert the interface of a class into another interface clients expect. It lets classes work together that couldn't otherwise due to incompatible interfaces.

**👨‍💻 Java Code:**

// Old interface

class OldCharger {

void chargeWithRoundPin() {

System.out.println("Charging using round pin charger");

}

}

// New interface

interface NewCharger {

void chargeWithFlatPin();

}

// Adapter

class ChargerAdapter implements NewCharger {

OldCharger oldCharger;

ChargerAdapter(OldCharger oldCharger) {

this.oldCharger = oldCharger;

}

public void chargeWithFlatPin() {

oldCharger.chargeWithRoundPin(); // Adapting old to new

}

}

// Usage

public class Main {

public static void main(String[] args) {

OldCharger old = new OldCharger();

NewCharger adapter = new ChargerAdapter(old);

adapter.chargeWithFlatPin();

}

}

**🌍 Real-world Use Case:**

Using an adapter plug to connect a European charger (round pin) to an Indian socket (flat pin).

**✅ 2. Bridge Pattern**

**👉 Intent:**

Separate abstraction from implementation so they can vary independently.

**👨‍💻 Java Code:**

// Implementor

interface DrawingAPI {

void drawCircle(int radius, int x, int y);

}

// ConcreteImplementors

class DrawingAPI1 implements DrawingAPI {

public void drawCircle(int radius, int x, int y) {

System.out.println("API1.circle at (" + x + "," + y + ") with radius " + radius);

}

}

class DrawingAPI2 implements DrawingAPI {

public void drawCircle(int radius, int x, int y) {

System.out.println("API2.circle at (" + x + "," + y + ") with radius " + radius);

}

}

// Abstraction

abstract class Shape {

protected DrawingAPI drawingAPI;

Shape(DrawingAPI api) {

this.drawingAPI = api;

}

public abstract void draw();

}

// Refined Abstraction

class Circle extends Shape {

private int x, y, radius;

Circle(int x, int y, int radius, DrawingAPI api) {

super(api);

this.x = x;

this.y = y;

this.radius = radius;

}

public void draw() {

drawingAPI.drawCircle(radius, x, y);

}

}

// Usage

public class Main {

public static void main(String[] args) {

Shape circle1 = new Circle(1, 2, 3, new DrawingAPI1());

Shape circle2 = new Circle(5, 7, 11, new DrawingAPI2());

circle1.draw();

circle2.draw();

}

}

**🌍 Real-world Use Case:**

Separating UI abstraction from platform-specific rendering APIs.

**✅ 3. Composite Pattern**

**👉 Intent:**

Compose objects into tree structures to represent part-whole hierarchies.

**👨‍💻 Java Code:**

import java.util.\*;

interface Employee {

void showDetails();

}

class Developer implements Employee {

private String name;

private int id;

Developer(String name, int id) {

this.name = name;

this.id = id;

}

public void showDetails() {

System.out.println("Developer: " + name + ", ID: " + id);

}

}

class Manager implements Employee {

private String name;

private int id;

Manager(String name, int id) {

this.name = name;

this.id = id;

}

public void showDetails() {

System.out.println("Manager: " + name + ", ID: " + id);

}

}

class CompanyDirectory implements Employee {

private List<Employee> employeeList = new ArrayList<>();

public void addEmployee(Employee emp) {

employeeList.add(emp);

}

public void showDetails() {

for (Employee emp : employeeList) {

emp.showDetails();

}

}

}

**🌍 Real-world Use Case:**

Representing an organization hierarchy (CEO → Managers → Developers).

**✅ 4. Decorator Pattern**

**👉 Intent:**

Attach new responsibilities to an object dynamically.

**👨‍💻 Java Code:**

interface Food {

String prepare();

double price();

}

class VegFood implements Food {

public String prepare() {

return "Veg Food";

}

public double price() {

return 50.0;

}

}

// Decorator

class FoodDecorator implements Food {

protected Food food;

FoodDecorator(Food food) {

this.food = food;

}

public String prepare() {

return food.prepare();

}

public double price() {

return food.price();

}

}

class NonVegFood extends FoodDecorator {

NonVegFood(Food food) {

super(food);

}

public String prepare() {

return food.prepare() + " + Chicken Curry";

}

public double price() {

return food.price() + 100.0;

}

}

**🌍 Real-world Use Case:**

Adding toppings to a pizza dynamically (Cheese, Olives, etc.).

**✅ 5. Facade Pattern**

**👉 Intent:**

Provide a unified interface to a set of interfaces in a subsystem.

**👨‍💻 Java Code:**

class CPU {

void start() {

System.out.println("CPU started");

}

}

class Memory {

void load() {

System.out.println("Memory loaded");

}

}

class Disk {

void boot() {

System.out.println("Disk booted");

}

}

// Facade

class ComputerFacade {

private CPU cpu;

private Memory memory;

private Disk disk;

ComputerFacade() {

cpu = new CPU();

memory = new Memory();

disk = new Disk();

}

public void startComputer() {

cpu.start();

memory.load();

disk.boot();

}

}

**🌍 Real-world Use Case:**

Simplified startup of a computer (you press one button, it runs many subsystems).

**✅ 6. Flyweight Pattern**

**👉 Intent:**

Use sharing to support large numbers of fine-grained objects efficiently.

**👨‍💻 Java Code:**

import java.util.\*;

class Robot {

private String type;

Robot(String type) {

this.type = type;

}

public void display(int x, int y) {

System.out.println("Robot of type: " + type + " at (" + x + "," + y + ")");

}

}

// Flyweight Factory

class RobotFactory {

private static final Map<String, Robot> robots = new HashMap<>();

public static Robot getRobot(String type) {

robots.putIfAbsent(type, new Robot(type));

return robots.get(type);

}

}

**🌍 Real-world Use Case:**

Used in games for rendering multiple similar objects like trees, soldiers, bullets, etc.

**✅ 7. Proxy Pattern**

**👉 Intent:**

Provide a surrogate or placeholder to control access to another object.

**👨‍💻 Java Code:**

interface Internet {

void connectTo(String site) throws Exception;

}

class RealInternet implements Internet {

public void connectTo(String site) {

System.out.println("Connecting to " + site);

}

}

class ProxyInternet implements Internet {

private Internet internet = new RealInternet();

private static List<String> bannedSites = Arrays.asList("abc.com", "xyz.com");

public void connectTo(String site) throws Exception {

if (bannedSites.contains(site)) {

throw new Exception("Access Denied to " + site);

}

internet.connectTo(site);

}

}

**🌍 Real-world Use Case:**

Firewall proxy controlling access to restricted websites.

**✅ Summary Table**

| **Pattern** | **Key Concept** | **Real-World Analogy** |
| --- | --- | --- |
| Adapter | Convert one interface to another | Plug adapter |
| Bridge | Separate abstraction & implementation | Remote control & TV |
| Composite | Hierarchical part-whole objects | Organization chart |
| Decorator | Add behavior dynamically | Toppings on a pizza |
| Facade | Simplified interface | Start button on a computer |
| Flyweight | Shared object reuse | Chess pieces (shared white/black) |
| Proxy | Access control | Bank ATM acting on behalf of the bank |

Would you like all these patterns implemented as a Java project structure (e.g., per package per pattern)?

**🧠 What Are Behavioral Design Patterns?**

**✅ Definition:**

**Behavioral Design Patterns** are concerned with **how objects interact and communicate with each other** while keeping the system loosely coupled. They help define the **responsibility of objects**, **flow of control**, and **patterns of communication**.

These patterns increase flexibility in carrying out communication between objects.

**📂 Types of Behavioral Design Patterns (GoF)**

There are **11 types**:

| **Pattern** | **Intent** |
| --- | --- |
| 1. Chain of Responsibility | Pass requests along a chain of handlers |
| 2. Command | Encapsulate a request as an object |
| 3. Interpreter | Implement a grammar interpreter |
| 4. Iterator | Access elements of a collection sequentially |
| 5. Mediator | Reduce chaotic dependencies between objects |
| 6. Memento | Capture and restore object state |
| 7. Observer | One-to-many dependency between objects |
| 8. State | Allow an object to alter its behavior when its state changes |
| 9. Strategy | Define a family of algorithms, encapsulate each one |
| 10. Template Method | Define the skeleton of an algorithm in a superclass |
| 11. Visitor | Separate algorithm from the object structure it operates on |

**🧪 Let’s go through each one briefly with Java code and real-world analogy:**

**1. 🧵 Chain of Responsibility**

**✅ Definition:**

Pass a request along a chain of handlers; each handler decides to process or forward.

**🧠 Real World:**

Customer support ticket escalates from Level 1 to Level 2, and so on.

**💻 Java Code:**

abstract class Handler {

protected Handler next;

public void setNext(Handler next) {

this.next = next;

}

public abstract void handleRequest(String request);

}

class Manager extends Handler {

public void handleRequest(String request) {

if (request.equals("low")) {

System.out.println("Manager handled: " + request);

} else if (next != null) {

next.handleRequest(request);

}

}

}

class Director extends Handler {

public void handleRequest(String request) {

System.out.println("Director handled: " + request);

}

}

public class ChainDemo {

public static void main(String[] args) {

Handler manager = new Manager();

Handler director = new Director();

manager.setNext(director);

manager.handleRequest("low");

manager.handleRequest("high");

}

}

**2. 🧾 Command**

**✅ Definition:**

Encapsulates a request as an object.

**🧠 Real World:**

Remote control buttons execute commands like TurnOn, TurnOff.

**💻 Java Code:**

interface Command {

void execute();

}

class Light {

void on() { System.out.println("Light On"); }

void off() { System.out.println("Light Off"); }

}

class LightOnCommand implements Command {

Light light;

LightOnCommand(Light l) { light = l; }

public void execute() { light.on(); }

}

class RemoteControl {

Command command;

void setCommand(Command c) { command = c; }

void pressButton() { command.execute(); }

}

public class CommandPattern {

public static void main(String[] args) {

Light light = new Light();

Command onCommand = new LightOnCommand(light);

RemoteControl remote = new RemoteControl();

remote.setCommand(onCommand);

remote.pressButton();

}

}

**3. 🧮 Interpreter**

**✅ Definition:**

Defines a grammar for language and interprets expressions.

**🧠 Real World:**

SQL interpreters, mathematical expression parsers.

**💻 Java Code:**

interface Expression {

boolean interpret(String context);

}

class TerminalExpression implements Expression {

private String data;

TerminalExpression(String data) { this.data = data; }

public boolean interpret(String context) {

return context.contains(data);

}

}

class OrExpression implements Expression {

Expression expr1, expr2;

OrExpression(Expression e1, Expression e2) { expr1 = e1; expr2 = e2; }

public boolean interpret(String context) {

return expr1.interpret(context) || expr2.interpret(context);

}

}

public class InterpreterExample {

public static void main(String[] args) {

Expression isJava = new TerminalExpression("Java");

Expression isPython = new TerminalExpression("Python");

Expression isProgramming = new OrExpression(isJava, isPython);

System.out.println(isProgramming.interpret("I love Python"));

}

}

**4. 🔁 Iterator**

**✅ Definition:**

Sequentially access elements without exposing internal representation.

**🧠 Real World:**

TV remote goes channel by channel.

**💻 Java Code:**

List<String> names = Arrays.asList("Ajay", "Meera", "John");

Iterator<String> itr = names.iterator();

while (itr.hasNext()) {

System.out.println(itr.next());

}

**5. 🤝 Mediator**

**✅ Definition:**

Central object coordinates communication between components.

**🧠 Real World:**

Air traffic controller communicates with multiple flights.

**💻 Java Code:**

interface Mediator {

void send(String message, Colleague colleague);

}

abstract class Colleague {

protected Mediator mediator;

Colleague(Mediator m) { mediator = m; }

}

class ConcreteColleague1 extends Colleague {

ConcreteColleague1(Mediator m) { super(m); }

void send(String msg) { mediator.send(msg, this); }

void receive(String msg) { System.out.println("Colleague1 received: " + msg); }

}

class ConcreteMediator implements Mediator {

ConcreteColleague1 c1;

public void setColleague(ConcreteColleague1 c1) { this.c1 = c1; }

public void send(String msg, Colleague sender) {

if (sender != c1) c1.receive(msg);

}

}

**6. 🧳 Memento**

**✅ Definition:**

Capture and restore an object’s internal state.

**🧠 Real World:**

Undo/Redo in editors like Word.

**7. 👀 Observer**

**✅ Definition:**

One-to-many dependency. If one object changes, all dependents get notified.

**🧠 Real World:**

YouTube subscribers get updates on new videos.

**💻 Java Code:**

interface Observer {

void update(String msg);

}

class Subscriber implements Observer {

private String name;

Subscriber(String name) { this.name = name; }

public void update(String msg) {

System.out.println(name + " received update: " + msg);

}

}

class Channel {

List<Observer> subs = new ArrayList<>();

void subscribe(Observer o) { subs.add(o); }

void notifyObservers(String msg) {

for (Observer o : subs) o.update(msg);

}

}

**8. 🔄 State**

**✅ Definition:**

Allow object behavior to change when internal state changes.

**🧠 Real World:**

Traffic light changes color and behavior.

**9. 🎯 Strategy**

**✅ Definition:**

Encapsulate interchangeable algorithms and make them interchangeable.

**🧠 Real World:**

Google Maps choosing shortest or fastest path.

**💻 Java Code:**

interface Strategy {

int execute(int a, int b);

}

class Add implements Strategy {

public int execute(int a, int b) { return a + b; }

}

class Context {

private Strategy strategy;

Context(Strategy strategy) { this.strategy = strategy; }

public int executeStrategy(int a, int b) {

return strategy.execute(a, b);

}

}

**10. 📜 Template Method**

**✅ Definition:**

Define algorithm skeleton, defer steps to subclasses.

**🧠 Real World:**

Making tea or coffee - boil water → brew → pour.

**11. 🚶‍♂️ Visitor**

**✅ Definition:**

Separate algorithm from object structure.

**🧠 Real World:**

Tax calculator visiting different items.

**🧑‍💼 Real-World Use Cases Summary**

| **Pattern** | **Real-World Example** |
| --- | --- |
| Chain of Responsibility | Tech support escalation |
| Command | Remote controls |
| Interpreter | SQL parser |
| Iterator | For loop over playlists |
| Mediator | Chat room or air traffic control |
| Memento | Undo/redo in text editors |
| Observer | Newsletter subscription |
| State | Vending machine state transitions |
| Strategy | Sorting algorithm choices |
| Template Method | Document processing steps |
| Visitor | Tax calculation on invoices |

**✅ Conclusion**

* **Behavioral patterns** make object communication more **organized and maintainable**.
* They promote **loose coupling** and **flexibility**.
* Understanding these is **critical** for building robust LLD systems like messaging apps, editors, UIs, etc.

Would you like me to prepare a PDF or a visual chart summarizing these for revision or presentation?