A design pattern in Java is a reusable solution to a common problem in software design. It provides a structured approach to solving problems in a way that is scalable, maintainable, and efficient.

GitHub : https://github.com/Prasannagk67/DesignPatternsPractice

Key Points:

- It is a best practice to solve recurring design problems.
- It improves code readability and reduces development time.
- It follows object-oriented principles like Encapsulation, Abstraction, and Polymorphism.
- Design patterns are not code but guidelines for writing better code.

Design patterns are classified into three main categories:

Category	Purpose	Most Used, Examples
Creational	Deals with object creation	Singleton*, Factory*, Builder, Prototype*, etc..
Structural	Deals with object relationships	Adapter*, Decorator, Composite, Proxy*, etc..
Behavioral	Deals with object communication	Observer*, Strategy*, Command, Iterator, etc..

Singleton : The Singleton Design Pattern ensures that a class has only one instance and provides a global access point to it. It is widely used in scenarios where only one object should control actions across the application.

Lazy Loading Singleton Design Pattern :

```
public class UserService {
  private static UserService userService;
  private UserService(){}
  public static UserService getUserservice(){
     if(userService == null){
        userService = new UserService();
     }
     return userService;
  }
  List<User> user = Arrays.asList(
   new User(1,"Prasanna","Prasanna123"),
        new User(2,"Prabhas","Prabhas123")
  );
  public List<User> getAllUsers(){
     return user;
  }
}
```

```
@RestController
@RequestMapping("/user")
public class UserController {
  UserService userService = UserService.getUserservice();
  @GetMapping
  public List<User> getAllUsers(){
     return userService.getAllUsers();
  }
}
```

Lazy Loading Singleton Design Pattern For Thread Safe

```
//Lazy Loading Design Pattern with Thread Safe Double-Checked Locking (Best Performance)
public class UserService {
  private static UserService userService;
  private UserService(){}
  public static UserService getUserService(){
     if(userService == null){
        synchronized (UserService.class){
           if (userService == null){
              userService = new UserService();
           }
        }
     }
     return userService;
```

```
//Lazy Loading Singleton Design Pattern with Thread safe but Slow
public class UserService {
  private static UserService userService;
  private UserService(){}
  public static synchronized UserService getUserService(){
     if(userService == null){
        userService = new UserService();
     }
     return userService;
  }
  List<User> user = Arrays.asList(
        new User(1,"Prasanna","Prasanna123"),
        new User(2,"Prabhas","Prabhas123")
```

```
  }
  List<User> user = Arrays.asList(
        new User(1,"Prasanna","Prasanna123"),
        new User(2,"Prabhas","Prabhas123")
  );
  public List<User> getAllUsers(){
     return user;
  }
}
```

```
  );
  public List<User> getAllUsers(){
     return user;
  }
}
```

Eager Loading Singleton Design Pattern :

```
public class UserService {
  private static UserService userService;
  private UserService(){}
  public static UserService getUserservice(){
     if(userService == null){
        userService = new UserService();
     }
     return userService;
  }
  List<User> user = Arrays.asList(
    new User(1,"Prasanna","Prasanna123"),
        new User(2,"Prabhas","Prabhas123")
  );
  public List<User> getAllUsers(){
     return user;
  }
}
```

```
@RestController
@RequestMapping("/user")
public class UserController {
  @GetMapping
  public List<User> getAllUsers(){
     return UserService.getUserService().getAllUsers();
  }
}
```

Default Singleton Design Pattern :

```
@Service
public class UserService {
  List<User> user = Arrays.asList(
    new User(1,"Prasanna","Prasanna123"),
        new User(2,"Prabhas","Prabhas123")
  );
  public List<User> getAllUsers(){
     return user;
  }
}
```

```
@RestController
@RequestMapping("/user")
public class UserController {
  @Autowired
  UserService userService;
  @GetMapping
  public List<User> getAllUsers(){
     return userService.getAllUsers();
  }
}
```

Type	Lazy	Eager	@Service (Default in Spring)
Instance Creation	Created only when needed (on first call)	Created at class loading time	Managed by the Spring IoC container
Thread Safety	Not thread-safe unless synchronized	Always thread-safe	Thread-safe by default
Performance	Faster startup but slower first access	Slower startup but faster access	Optimized by Spring
Memory Usage	Lower memory usage (only created when needed)	Higher memory usage (always created even if never used)	Optimized by Spring

Best Choice

- For Spring Applications → Spring @Service Singleton is the best
- For Non-Spring Applications → Eager Singleton is the best for simplicity & thread safety
- For Memory-Sensitive Applications → Lazy Singleton is better

If you're using Spring, always prefer the Spring-managed Singleton because it provides built-in optimization, dependency injection, and lifecycle management.

Factory : The Factory Design Pattern is a creational design pattern that provides a way to create objects without specifying their exact class. It allows the creation of objects based on input parameters or conditions while keeping the object creation logic centralized.It improves maintainability and scalability by hiding complex instantiation logic and keeping client code clean.

In simple terms when there is a superclass and multiple subclasses, we want to get the object of the subclass based on input and requirements. Then we create a factory class which takes the responsibility of creating objects of the class based on input.

Factory Design Pattern, the object creation logic is hidden from the client by encapsulating it inside a factory method. Instead of directly instantiating objects using new, the client calls the factory method, which decides which concrete class to instantiate based on input parameters.

```
public interface Payment {
  void payment(double amount);
}
```

```
public class CreditCard implements Payment{
  @Override
  public void payment(double amount) {
     System.out.println("Credit Card :"+amount);
  }}
```

```
public class DebitCard implements Payment{
  @Override
  public void payment(double amount) {
     System.out.println("Debit Card :"+amount);
  }}
```

```
public class UpiPayment implements Payment{
  @Override
  public void payment(double amount) {
     System.out.println("Upi Payment :"+amount);
  }}
```

```
public class PaymentFactory {
  public static Payment getPaymentType(String type){
     if (type.equalsIgnoreCase("CreditCard")){
        return new CreditCard();
     }else if (type.equalsIgnoreCase("UpiPayment")){
        return new UpiPayment();
     }else if (type.equalsIgnoreCase("DebitCard")){
        return new DebitCard();
     }
     throw new IllegalArgumentException("Invalid Payment Type");
  }}
```

```
public class FactoryMain {
  public static void main(String[] args){
     Payment paymentType =
PaymentFactory.getPaymentType("CreditCard");
     paymentType.payment(100);
  }
}
```

Prototype : The Prototype Pattern is a creational design pattern that allows cloning of objects instead of creating new instances. It improves performance when object creation is expensive.

When to Use?

- When object creation is costly or complex.
- When a class has multiple configurations and initializing them is resource-intensive.
- When you need to avoid subclassing by copying an existing object.

```
public interface Prototype extends Cloneable {
  Prototype clone();
}
```

```
@Data
public class Employee implements Prototype{
  private int id;
  private String name;
  private String company;
  public Employee(int id, String name, String company) {
     this.id = id;
```

```
public class PrototypeMain {
  public static void main(String[] args){
     Employee prototypeEmployee = new
Employee(1,"Prasanna","HCL");
     Employee employee = (Employee)
prototypeEmployee.clone();
     employee.setId(2);
```

```
    this.name = name;
    this.company = company;
  }

  @Override
  public Prototype clone() {
    return new Employee(this.id,this.name,this.company);
  }}
```

```
    employee.setName("Charan");
    employee.setCompany("IMSS");
    employee1.setId(3);
    employee1.setName("NTR");
    employee1.setCompany("Virtusa");
    System.out.println(employee1);
    System.out.println(employee);
  }}
```

Builder : The Builder Pattern is a creational design pattern used to create complex objects step by step and finally return the final object with desired values of attributes. It separates the object construction process from its representation, allowing us to create different variations of an object.

While creating objects when object contain may attributes there are many problems exists:

- We have to pass many arguments to create an object
- Some parameters might be optional.
- Factory class takes all responsibilities for creating objects. If the object is heavy then all complexity is the factory of class.

```
public class Loan {
  private String type;
  private double amount;
  private double interestRate;
  private int tenure;
  private double processingFee;
  private boolean insurance;
  private Loan (LoanBuilder builder){
    this.type=builder.type;
    this.amount=builder.amount;
    this.interestRate=builder.interestRate;
    this.tenure=builder.tenure;
    this.processingFee=builder.processingFee;
    this.insurance=builder.insurance;
  }
  @Override
  public String toString() {
    return "Loan{" +
          "type='" + type + '\'' +
          ", amount=" + amount +
          ", interestRate=" + interestRate +
          ", tenure=" + tenure +
          ", processingFee=" + processingFee +
          ", insurance=" + insurance +
          '}';
  }
  public static class LoanBuilder{
    private String type;
    private double amount;
    private double interestRate=10.0;
    private int tenure = 2;
    private double processingFee = 1000.0;
    private boolean insurance=false;
    public LoanBuilder(String type,double amount){
      this.type=type;
      this.amount=amount;
    }
    public LoanBuilder setInterestRate(double interestRate){
      this.interestRate=interestRate;
      return this;
    }
    public LoanBuilder setTenure(int tenure){
      this.tenure=tenure;
      return this;
    }
    public LoanBuilder setProcessingFee(double processingFee){
```

```
public class LoanMain {
  public static void main(String[] args){
      Loan homeLoan = new Loan.LoanBuilder("Home
Loan",500000).build();
      System.out.println(homeLoan);
      Loan carLoan = new Loan.LoanBuilder("Car
Loan",9000000)
            .setInterestRate(7.5)
            .setTenure(60)
            .build();
      System.out.println(carLoan);
      Loan personalLoan = new
Loan.LoanBuilder("Personal Loan",60000000)
            .setInterestRate(9.8)
            .setTenure(90)
            .setProcessingFee(800)
            .build();
      System.out.println(personalLoan);
  }
}
```

```
        this.processingFee=processingFee;
        return this;
    }
    public LoanBuilder includeInsurance(){
        this.insurance=true;
        return this;
    }
    public Loan build(){
        return new Loan(this);
    }}}
```

Why Use the Builder Pattern

- Solves Constructor Overloading Issue : Avoids multiple constructors with many parameters.
- Readable & Maintainable Code : Object creation is clear and step-by-step.
- Immutable Objects : Helps in creating immutable objects.
- Scalability : Easily extends object construction without breaking existing code.

Use the Builder Pattern when:

- A class has many optional fields.
- The constructor has too many parameters.
- Object creation should be step-by-step and flexible

Adapter: The Adapter Design Pattern is a structural design pattern that allows objects with incompatible interfaces to work together. It acts as a bridge between two incompatible interfaces.

Key Concepts

- The adapter class converts the interface of a class into another interface that a client expects.
- It allows existing classes to work with new interfaces without modifying their source code.
- It follows the principle of "program to an interface, not an implementation".

```
public interface Payment {
    void payment(String account, double amount);
}
```

```
public class CardPayment {
    public void makePayment(String email, double amount){
        System.out.println("Used Card Payment By "+email+" Of Amount :"+amount);
    }}
```

```
public class AdaptorImpl implements Payment {
    private CardPayment cardPayment;
    public AdaptorImpl (CardPayment cardPayment){
        this.cardPayment = cardPayment;
    }
    @Override
    public void payment(String account, double amount) {
        cardPayment.makePayment(account,amount);
    }}
```

```
public class AdapterMain {
    public static void main(String[] args) {
        Payment payment = new AdaptorImpl(new CardPayment());
        payment.payment("prasanna@gmail.com",500);
    }}
```

Conclusion : The Adapter Pattern is useful when two interfaces are incompatible but need to work together. It promotes code reusability and flexibility without modifying existing code. It is widely used in Java frameworks like Spring, Hibernate, and JDBC.

Simple Real-Life Example: Mobile Charger Adapter : Imagine you bought a laptop from the USA, and its charger has a 110V plug. But in India, the power supply is 220V. You need a power adapter to convert 220V to 110V so the laptop can charge.

Decorator : The Decorator Pattern is a structural design pattern that allows you to dynamically add new behaviors to objects without modifying their original code. It provides a flexible alternative to subclassing for extending functionality.

Key Concepts

- Wrapper-based approach : Instead of modifying a class, we "wrap" it with new functionality.
- Follows Open-Closed Principle : You can extend behaviors without modifying existing code.
- Multiple Decorators : You can apply multiple decorators dynamically at runtime.

Composite : The Composite Pattern is a structural design pattern used to treat individual objects and groups of objects (compositions) uniformly. It is useful when building a tree-like structure (e.g., file systems, menus, UI components, etc.).

Key Concepts

- Component (Interface/Abstract Class) : Defines a common structure for both individual and composite objects.
- Leaf (Concrete Class) : Represents individual objects (no children).
- Composite (Concrete Class) : Contains multiple child components, which can be either Leaf or another Composite.

Proxy : The Proxy Pattern is a structural design pattern that provides a placeholder (proxy) object to control access to another object. It acts as an intermediary between the client and the real object.

Why Use Proxy Pattern?

- Access Control : Restrict access to certain operations (e.g., security checks).
- Lazy Initialization : Load heavy objects only when needed.
- Logging & Monitoring : Track method calls without modifying the real object.
- Remote Access : Act as a local representation of a remote object.

Types of Proxy

1. Virtual Proxy : Creates expensive objects only when needed (lazy loading).
2. Protection Proxy : Controls access based on user roles (security).
3. Remote Proxy : Acts as a local representation of a remote object (e.g., RMI, APIs).
4. Smart Proxy : Adds extra functionalities like logging, caching, etc.

```java
public interface Internet {
  void connectTo(String serverHost) throws Exception;
}
```

```java
public class RealInternet implements Internet{
  @Override
  public void connectTo(String serverHost) throws Exception {
    System.out.println("Connected To:"+serverHost);
  }
}
```

```java
public class ProxyInternet implements Internet{
  private final RealInternet realInternet = new RealInternet();
  private final static List<String> bannedSites;
  static {
    bannedSites = new ArrayList<>();
    bannedSites.add("youtube.com");
  }
  @Override
  public void connectTo(String serverHost) throws Exception {
    if (bannedSites.contains(serverHost.toLowerCase())){
      throw new Exception("Access Denied to :"+serverHost);
    }else{
      realInternet.connectTo(serverHost);
    }
  }
}
```

```java
public class ProxyMain {
  public static void main(String[] args){
    Internet internet = new ProxyInternet();
    try {
      internet.connectTo("google.com");
      internet.connectTo("youtube.com");
    }catch (Exception e){
      System.out.println(e.getMessage());
    }
  }
}
```

Observer : The Observer Pattern is a behavioral design pattern where an object (Subject) maintains a list of dependent objects (Observers) and notifies them automatically of any state changes. It is useful for implementing event-driven programming.

Key Components

- Subject (Observable) : Maintains a list of observers and notifies them of changes.
- Observer : Gets notified when the subject's state changes.
- Concrete Subject : The actual object being observed.
- Concrete Observer : The object that reacts to changes in the subject.

```
public interface Stock {
  void registerObserver(Investor investor);
  void removeObserver(Investor investor);
  void notifyObserver();
}
```

```
public interface Investor {
  void update(double stockPrice);
}
```

```
public class CompanyInvestor implements Investor{
  private String companyName;
  public CompanyInvestor(String companyName) {
    this.companyName = companyName;
  }
  @Override
  public void update(double stockPrice) {
    System.out.println("Company " + companyName + " 
notified. New stock price: $" + stockPrice);
  }}
```

```
public class IndividualInvestor implements Investor{
  private String name;
  public IndividualInvestor(String name) {
    this.name = name;
  }
  @Override
  public void update(double stockPrice) {
    System.out.println("Investor " + name + " notified. New 
stock price: $" + stockPrice);
  }}
```

```
public class StockMarket implements Stock{
  private List<Investor> investors = new ArrayList<>();
  private double stockPrice;
  public void setStockPrice(double stockPrice){
    this.stockPrice=stockPrice;
    notifyObserver();
  }
  public double getStockPrice(){
    return stockPrice;
  }
  @Override
  public void registerObserver(Investor investor) {
    investors.add(investor);
  }
  @Override
  public void removeObserver(Investor investor) {
    investors.remove(investor);
  }
  @Override
  public void notifyObserver() {
    for (Investor investor : investors){
      investor.update(stockPrice);
    }}}
```

```
public class ObserverMain {
  public static void main(String[] args) {
    StockMarket stockMarket = new StockMarket();
    IndividualInvestor investor1 = new 
IndividualInvestor("Alice");
    IndividualInvestor investor2 = new 
IndividualInvestor("Bob");
    CompanyInvestor company1 = new 
CompanyInvestor("XYZ Corp");
    stockMarket.registerObserver(investor1);
    stockMarket.registerObserver(investor2);
    stockMarket.registerObserver(company1);
    stockMarket.setStockPrice(150.75);
    System.out.println("-----");
    stockMarket.setStockPrice(155.50);
    System.out.println("-----");
    stockMarket.removeObserver(investor1);
    stockMarket.setStockPrice(160.00);
  }
}
```

Strategy : The Strategy Pattern is a behavioral design pattern that allows you to define a family of algorithms, encapsulate them, and make them interchangeable at runtime. This pattern is useful when you have multiple ways to perform a task and want to switch between them dynamically.

Key Concepts

- Strategy (Interface) : Defines a common interface for all strategies (algorithms).
- Concrete Strategies (Classes) : Implement different variations of an algorithm.
- Context (Class) : Uses a strategy and allows clients to change it dynamically.

```
public interface PaymentStrategy
{
  void pay(double amount);
}
```

```
public class CreditCard implements
PaymentStrategy{
  private String cardNumber;
  public CreditCard(String cardNumber){
    this.cardNumber=cardNumber;
  }
  @Override
```

```
public class UpiPayment implements
PaymentStrategy{
  private String upiId;
  public UpiPayment(String upiId){
    this.upiId=upiId;
  }
  @Override
```

	public void pay(double amount) { System.*out*.println("CreditCard :"+amount); }}	public void pay(double amount) { System.*out*.println("UpiId :"+amount); }}

```
public class PaymentContext {
  private PaymentStrategy paymentStrategy;
  public void setPaymentStrategy(PaymentStrategy
paymentStrategy){this.paymentStrategy = paymentStrategy;}
  public void payAmount(double amount){
    if(paymentStrategy == null){
      System.out.println("Please select a Payment Method");
    }else{
      paymentStrategy.pay(amount);
    }}}
```

```
public class StrategyMain {
  public static void main(String[] args){
    PaymentContext paymentContext = new PaymentContext();
    paymentContext.setPaymentStrategy(new
UpiPayment("1234567@ybl"));
    paymentContext.payAmount(800);
  }
}
```

Command : The Command Pattern is a behavioral design pattern that turns a request (command) into an object, allowing users to parameterize methods, delay execution, or queue requests. It is commonly used in undo/redo functionality, task scheduling, and event-driven programming.

Key Components

- Command (Interface) : Declares the execute() method for all commands.
- Concrete Commands (Classes) : Implement the execute() method by calling specific actions on the receiver.
- Receiver (Class) : Performs the actual operation when a command is executed.
- Invoker (Class) : Stores and executes commands.
- Client (Main Class) : Creates command objects and assigns them to an invoker.

```
public interface Command {
  void execute();
}
```

```
public class Light {
  public void turnOn(){
    System.out.println("Light On");
  }
  public void turnOff(){
    System.out.println("Light Off");
  }}
```

```
public class LightOffCommand implements Command{
  private Light light;
  public LightOffCommand(Light light){
    this.light=light;
  }
  @Override
  public void execute() {
    light.turnOff();
  }}
```

```
public class LightOnCommand implements Command {
  private Light light;
  public LightOnCommand (Light light){
    this.light=light;
  }
  @Override
  public void execute() {
    light.turnOn();
  }}
```

```
public class CommandControl {
  private Command command;
  public void setCommand(Command command){
    this.command=command;
  }
  public void pressButton(){
    if (command != null){
      command.execute();
    }else{
      System.out.println("No Command Assigned");
    }}}
```

```
public class CommandMain {
  public static void main(String[] args){
    Light light = new Light();
    Command lightOn = new LightOnCommand(light);
    Command lightOff = new LightOffCommand(light);
    CommandControl control = new CommandControl();
    control.setCommand(lightOn);
    control.pressButton();
    control.setCommand(lightOff);
    control.pressButton();
  }}
```

Iterator : The Iterator Pattern is a behavioral design pattern that provides a way to access elements of a collection sequentially without exposing its internal structure. This pattern is commonly used in Java's Collection Framework (e.g., List, Set, Map, etc.).

Key Concepts

- Iterator (Interface) : Defines methods for traversing elements (e.g., hasNext(), next()).
- Concrete Iterator (Class) : Implements the iterator interface for a specific collection.
- Aggregate (Collection Interface) : Declares a method to return an iterator.
- Concrete Aggregate (Collection Class) : Implements the aggregate interface and provides an iterator.