

# Design Patterns Index



---

Welcome!

This document provides quick access to summaries of all 22 classic design patterns from [Refactoring Guru](#).

---

## Versions Available

-  **English version:** [Jump to English Summary](#)
  -  **Spanish version:** [Ir al Resumen en Español](#)
- 

## Contents

### Creational Patterns (English)

- [Factory Method — Refactoring Guru](#)
- [Abstract Factory — Refactoring Guru](#)
- [Builder — Refactoring Guru](#)
- [Prototype — Refactoring Guru](#)
- [Singleton — Refactoring Guru](#)

### Structural Patterns (English)

- [Adapter — Refactoring Guru](#)
- [Bridge — Refactoring Guru](#)
- [Composite — Refactoring Guru](#)
- [Decorator — Refactoring Guru](#)
- [Facade — Refactoring Guru](#)
- [Flyweight — Refactoring Guru](#)
- [Proxy — Refactoring Guru](#)

### Behavioral Patterns (English)

- [Chain of Responsibility — Refactoring Guru](#)
  - [Command — Refactoring Guru](#)
  - [Iterator — Refactoring Guru](#)
  - [Mediator — Refactoring Guru](#)
  - [Memento — Refactoring Guru](#)
  - [Observer — Refactoring Guru](#)
  - [State — Refactoring Guru](#)
  - [Strategy — Refactoring Guru](#)
  - [Template Method — Refactoring Guru](#)
  - [Visitor — Refactoring Guru](#)
- 

### Patrones Creacionales (Español)

- [Factory Method — Refactoring Guru](#)
- [Abstract Factory — Refactoring Guru](#)
- [Builder — Refactoring Guru](#)
- [Prototype — Refactoring Guru](#)
- [Singleton — Refactoring Guru](#)



## Patrones Estructurales (Español)

- [Adapter — Refactoring Guru](#)
- [Bridge — Refactoring Guru](#)
- [Composite — Refactoring Guru](#)
- [Decorator — Refactoring Guru](#)
- [Facade — Refactoring Guru](#)
- [Flyweight — Refactoring Guru](#)
- [Proxy — Refactoring Guru](#)



## Patrones de Comportamiento (Español)

- [Chain of Responsibility — Refactoring Guru](#)
- [Command — Refactoring Guru](#)
- [Iterator — Refactoring Guru](#)
- [Mediator — Refactoring Guru](#)
- [Memento — Refactoring Guru](#)
- [Observer — Refactoring Guru](#)
- [State — Refactoring Guru](#)
- [Strategy — Refactoring Guru](#)
- [Template Method — Refactoring Guru](#)
- [Visitor — Refactoring Guru](#)



*Tip:*

Click on any section to jump directly to its detailed description and Java example below, or visit the Refactoring Guru page for more details.



# Design Patterns — Refactoring Guru (Summary + Java Examples)

Design patterns are **reusable solutions** to common problems in software design. They are grouped into three main categories:

- **Creational:** How objects are created.
- **Structural:** How classes and objects are composed.
- **Behavioral:** How objects interact and communicate.



## CREATIONAL PATTERNS [Creational Patterns]

### [Factory Method](#)

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

**Java Example:**

```
abstract class Creator {
    public abstract Product createProduct();
}

class ConcreteCreator extends Creator {
    public Product createProduct() {
        return new ConcreteProduct();
    }
}
```

---

## Abstract Factory

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

**Java Example:**

```
interface GUIFactory {
    Button createButton();
    Checkbox createCheckbox();
}

class WinFactory implements GUIFactory {
    public Button createButton() { return new WinButton(); }
    public Checkbox createCheckbox() { return new WinCheckbox(); }
}
```

---

## Builder

**Intent:** Constructs complex objects step by step.

**Java Example:**

```
class HouseBuilder {
    private House house = new House();

    public HouseBuilder buildWalls() { house.setWalls(true); return this; }

    public HouseBuilder buildRoof() { house.setRoof(true); return this; }
    public House build() { return house; }
}
```

---

## Prototype

**Intent:** Lets you copy existing objects without making your code dependent on their classes.

**Java Example:**

```
class Shape implements Cloneable {
    public Object clone() throws CloneNotSupportedException {
        return super.clone();
    }
}
```

---

## Singleton

**Intent:** Ensures that a class has only one instance and provides a global point of access to it.

**Java Example:**

```
class Singleton {
    private static Singleton instance;
    private Singleton() {}
    public static Singleton getInstance() {
        if (instance == null) instance = new Singleton();
        return instance;
    }
}
```

---

## STRUCTURAL PATTERNS

### Adapter

**Intent:** Allows objects with incompatible interfaces to collaborate.

**Java Example:**

```
class EuropeanPlug {
    public void connect() { System.out.println("Connected to 220V"); }
}

class AmericanAdapter {
    private EuropeanPlug plug;
    public AmericanAdapter(EuropeanPlug plug) { this.plug = plug; }
    public void connect110V() { plug.connect(); }
}
```

---

## Bridge

**Intent:** Separates abstraction from its implementation so that both can vary independently.

### Java Example:

```
interface Device { void turnOn(); }
class TV implements Device { public void turnOn() { System.out.println("TV on"); } }

abstract class Remote {
    protected Device device;
    protected Remote(Device device) { this.device = device; }
    abstract void turnOn();
}

class BasicRemote extends Remote {
    public BasicRemote(Device d) { super(d); }
    public void turnOn() { device.turnOn(); }
}
```

---

## Composite

**Intent:** Lets clients treat individual objects and compositions of objects uniformly.

### Java Example:

```
interface Component {
    void show();
}

class Leaf implements Component {
    public void show() { System.out.println("Leaf"); }
}

class Composite implements Component {
    private List<Component> children = new ArrayList<>();
    public void add(Component c) { children.add(c); }
    public void show() { children.forEach(Component::show); }
}
```

---

## Decorator

**Intent:** Adds responsibilities to objects dynamically without changing their class.

### Java Example:

```
interface Notifier {
    void send(String message);
}

class EmailNotifier implements Notifier {
    public void send(String message) { System.out.println("Email: " +
message); }
}

class SMSNotifier implements Notifier {
    private Notifier wrappee;
    public SMSNotifier(Notifier n) { this.wrappee = n; }
    public void send(String message) {
        wrappee.send(message);
        System.out.println("SMS: " + message);
    }
}
```

---

## Facade

**Intent:** Provides a simplified interface to a complex subsystem.

### Java Example:

```
class AudioSystem { void on(){} }
class LightSystem { void off(){} }

class HomeTheaterFacade {
    private AudioSystem audio = new AudioSystem();
    private LightSystem lights = new LightSystem();

    public void watchMovie() {
        audio.on();
        lights.off();
        System.out.println("Ready to watch a movie");
    }
}
```

---

## Flyweight

**Intent:** Reduces memory usage by sharing common parts of objects instead of duplicating them.

### Java Example:

```
class Flyweight {
    private final String color;
```

```
    public Flyweight(String color) { this.color = color; }
}

class FlyweightFactory {
    private static final Map<String, Flyweight> cache = new HashMap<>();
    public static Flyweight get(String color) {
        return cache.computeIfAbsent(color, Flyweight::new);
    }
}
```

---

## Proxy

**Intent:** Provides a substitute or placeholder for another object to control access to it.

### Java Example:

```
interface Service {
    void run();
}

class RealService implements Service {
    public void run() { System.out.println("Real service running"); }
}

class ProxyService implements Service {
    private RealService service;
    public void run() {
        if (service == null) service = new RealService();
        service.run();
    }
}
```

---

## BEHAVIORAL PATTERNS

### Chain of Responsibility

**Intent:** Passes requests along a chain of handlers until one of them handles it.

### Java Example:

```
abstract class Handler {
    protected Handler next;
    public void setNext(Handler next) { this.next = next; }
    public abstract void handle(String req);
}

class ConcreteHandler extends Handler {
```

```
    public void handle(String req) {  
        if (req.equals("ok")) System.out.println("Handled");  
        else if (next != null) next.handle(req);  
    }  
}
```

---

## Command

**Intent:** Turns a request into a stand-alone object that can be stored, queued, or undone.

### Java Example:

```
interface Command { void execute(); }  
  
class Light {  
    void on() { System.out.println("Light on"); }  
}  
  
class TurnOnLight implements Command {  
    private Light light;  
    public TurnOnLight(Light l) { this.light = l; }  
    public void execute() { light.on(); }  
}
```

---

## Iterator

**Intent:** Lets you traverse elements of a collection without exposing its internal structure.

### Java Example:

```
Iterator<String> it = List.of("a", "b", "c").iterator();  
while (it.hasNext()) System.out.println(it.next());
```

---

## Mediator

**Intent:** Reduces chaotic dependencies between objects by introducing a mediator object.

### Java Example:

```
interface Mediator { void send(String message, Colleague sender); }  
  
class ConcreteMediator implements Mediator {  
    private List<Colleague> colleagues = new ArrayList<>();  
    public void register(Colleague c) { colleagues.add(c); }  
}
```



```
    public void send(String message, Colleague sender) {
        for (Colleague c : colleagues) if (c != sender)
            c.receive(message);
    }
}

class Colleague {
    private Mediator mediator;
    public Colleague(Mediator m) { this.mediator = m; }
    public void send(String m) { mediator.send(m, this); }
    public void receive(String m) { System.out.println("Received: " + m); }
}
```

---

## Memento

**Intent:** Allows saving and restoring an object's state without violating encapsulation.

### Java Example:

```
class Memento {
    private final String state;
    public Memento(String s) { state = s; }
    public String getState() { return state; }
}

class Originator {
    private String state;
    public Memento save() { return new Memento(state); }
    public void restore(Memento m) { state = m.getState(); }
}
```

---

## Observer

**Intent:** Defines a one-to-many dependency so that when one object changes state, all dependents are notified.

### Java Example:

```
interface Observer { void update(String msg); }

class Subject {
    private List<Observer> observers = new ArrayList<>();
    public void add(Observer o) { observers.add(o); }
    public void notify(String msg) { observers.forEach(o ->
o.update(msg)); }
}
```

---

## State

**Intent:** Allows an object to alter its behavior when its internal state changes.

### Java Example:

```
interface State { void handle(); }

class StateA implements State {
    public void handle() { System.out.println("State A"); }
}

class Context {
    private State state;
    public void setState(State s) { state = s; }
    public void execute() { state.handle(); }
}
```

---

## Strategy

**Intent:** Defines a family of algorithms, encapsulates each one, and makes them interchangeable.

### Java Example:

```
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;
    public void setStrategy(Strategy s) { strategy = s; }
    public int run(int a, int b) { return strategy.execute(a, b); }
}
```

---

## Template Method

**Intent:** Defines the skeleton of an algorithm, letting subclasses redefine specific steps.

### Java Example:

```
abstract class Game {
    abstract void initialize();
    abstract void play();
}
```

```
    abstract void end();

    public final void playGame() {
        initialize();
        play();
        end();
    }
}
```

---

## Visitor

**Intent:** Lets you define new operations on an object structure without changing the classes of the elements.

### Java Example:

```
interface Visitor { void visit(ConcreteElement e); }

interface Element {
    void accept(Visitor v);
}

class ConcreteElement implements Element {
    public void accept(Visitor v) { v.visit(this); }
}
```

# Patrones de Diseño — Refactoring Guru (Resumen + Ejemplos en Java)

---

Los **patrones de diseño** son soluciones reutilizables a problemas comunes en el diseño de software. Se dividen en tres categorías principales:

- **Creacionales:** Cómo se crean los objetos.
- **Estructurales:** Cómo se componen las clases y los objetos.
- **De Comportamiento:** Cómo interactúan los objetos entre sí.

---

## PATRONES CREACIONALES

### 1. Factory Method

**Propósito:** Define una interfaz para crear un objeto, pero deja que las subclases decidan qué clase instanciar. Útil cuando una clase no puede anticipar qué tipo de objeto debe crear.

### Ejemplo en Java:

```
abstract class Creador {
    public abstract Producto crearProducto();
}

class CreadorConcreto extends Creador {
    public Producto crearProducto() {
        return new ProductoConcreto();
    }
}
```

---

## 2. Abstract Factory

**Propósito:** Crea familias de objetos relacionados sin especificar sus clases concretas.

**Ejemplo en Java:**

```
interface GUIFactory {
    Boton crearBoton();
    Checkbox crearCheckbox();
}

class WinFactory implements GUIFactory {
    public Boton crearBoton() { return new WinBoton(); }
    public Checkbox crearCheckbox() { return new WinCheckbox(); }
}
```

---

## 3. Builder

**Propósito:** Permite construir objetos complejos paso a paso.

**Ejemplo en Java:**

```
class CasaBuilder {
    private Casa casa = new Casa();

    public CasaBuilder construirParedes() { casa.setParedes(true); return this; }
    public CasaBuilder construirTecho() { casa.setTecho(true); return this; }
    public Casa build() { return casa; }
}
```

---

## 4. Prototype

**Propósito:** Permite copiar objetos existentes sin acoplar el código a sus clases.

**Ejemplo en Java:**

```
class Forma implements Cloneable {  
    public Object clone() throws CloneNotSupportedException {  
        return super.clone();  
    }  
}
```

---

## 5. Singleton

**Propósito:** Garantiza que una clase tenga solo una instancia y proporciona un punto de acceso global.

**Ejemplo en Java:**

```
class Singleton {  
    private static Singleton instancia;  
    private Singleton() {}  
    public static Singleton getInstancia() {  
        if (instancia == null) instancia = new Singleton();  
        return instancia;  
    }  
}
```

---

## PATRONES ESTRUCTURALES

### 6. Adapter

**Propósito:** Permite que objetos con interfaces incompatibles trabajen juntos.

**Ejemplo en Java:**

```
class EnchufeEuropeo {  
    public void conectar() { System.out.println("Conectado a 220V"); }  
}  
  
class AdaptadorAmericano {  
    private EnchufeEuropeo enchufe;  
    public AdaptadorAmericano(EnchufeEuropeo enchufe) { this.enchufe =  
enchufe; }  
    public void conectar110V() { enchufe.conectar(); }  
}
```

---

## 7. Bridge

**Propósito:** Separa la abstracción de su implementación para que ambas puedan variar independientemente.

**Ejemplo en Java:**

```
interface Dispositivo { void encender(); }
class TV implements Dispositivo { public void encender() {
System.out.println("TV encendida"); } }

abstract class ControlRemoto {
    protected Dispositivo dispositivo;
    protected ControlRemoto(Dispositivo dispositivo) { this.dispositivo =
dispositivo; }
    abstract void encender();
}

class ControlBasico extends ControlRemoto {
    public ControlBasico(Dispositivo d) { super(d); }
    public void encender() { dispositivo.encender(); }
}
```

---

## 8. Composite

**Propósito:** Permite tratar objetos individuales y composiciones de objetos de la misma forma.

**Ejemplo en Java:**

```
interface Componente {
    void mostrar();
}

class Hoja implements Componente {
    public void mostrar() { System.out.println("Hoja"); }
}

class Compuesto implements Componente {
    private List<Componente> hijos = new ArrayList<>();
    public void agregar(Componente c) { hijos.add(c); }
    public void mostrar() { hijos.forEach(Componente::mostrar); }
}
```

---

## 9. Decorator

**Propósito:** Agrega responsabilidades a un objeto dinámicamente sin modificar su clase.

### Ejemplo en Java:

```
interface Notificador {
    void enviar(String mensaje);
}

class NotificadorEmail implements Notificador {
    public void enviar(String mensaje) { System.out.println("Email: " +
mensaje); }
}

class NotificadorSMS implements Notificador {
    private Notificador wrappee;
    public NotificadorSMS(Notificador n) { this.wrappee = n; }
    public void enviar(String mensaje) {
        wrappee.enviar(mensaje);
        System.out.println("SMS: " + mensaje);
    }
}
```

---

## 10. Facade

**Propósito:** Proporciona una interfaz unificada para un conjunto complejo de subsistemas.

### Ejemplo en Java:

```
class SistemaAudio { void encender(){} }
class SistemaLuces { void apagar(){} }

class HomeTheaterFacade {
    private SistemaAudio audio = new SistemaAudio();
    private SistemaLuces luces = new SistemaLuces();

    public void verPelicula() {
        audio.encender();
        luces.apagar();
        System.out.println("Listo para la película");
    }
}
```

---

## 11. Flyweight

**Propósito:** Reduce el uso de memoria compartiendo datos comunes entre múltiples objetos.

### Ejemplo en Java:

```
class Flyweight {
    private final String color;
    public Flyweight(String color) { this.color = color; }
}

class FlyweightFactory {
    private static final Map<String, Flyweight> cache = new HashMap<>();
    public static Flyweight get(String color) {
        return cache.computeIfAbsent(color, Flyweight::new);
    }
}
```

---

## 12. Proxy

**Propósito:** Proporciona un sustituto para controlar el acceso a un objeto.

**Ejemplo en Java:**

```
interface Servicio {
    void operar();
}

class ServicioReal implements Servicio {
    public void operar() { System.out.println("Ejecutando operación real"); }
}

class ProxyServicio implements Servicio {
    private ServicioReal servicio;
    public void operar() {
        if (servicio == null) servicio = new ServicioReal();
        servicio.operar();
    }
}
```

---

## PATRONES DE COMPORTAMIENTO

### 13. Chain of Responsibility

**Propósito:** Pasa una solicitud a lo largo de una cadena de manejadores hasta que uno la procesa.

**Ejemplo en Java:**

```
abstract class Manejador {
    protected Manejador siguiente;
    public void setSiguiente(Manejador s) { siguiente = s; }
```



```
    public abstract void manejar(String req);
}

class ManejadorConcreto extends Manejador {
    public void manejar(String req) {
        if (req.equals("ok")) System.out.println("Procesado");
        else if (siguiente != null) siguiente.manejar(req);
    }
}
```

---

## 14. Command

**Propósito:** Encapsula una solicitud como un objeto, permitiendo deshacer o encolar comandos.

**Ejemplo en Java:**

```
interface Comando { void ejecutar(); }

class Luz {
    void encender() { System.out.println("Luz encendida"); }
}

class EncenderLuz implements Comando {
    private Luz luz;
    public EncenderLuz(Luz l) { this.luz = l; }
    public void ejecutar() { luz.encender(); }
}
```

---

## 15. Iterator

**Propósito:** Permite recorrer elementos de una colección sin exponer su estructura.

**Ejemplo en Java:**

```
Iterator<String> it = List.of("a", "b", "c").iterator();
while (it.hasNext()) System.out.println(it.next());
```

---

## 16. Mediator

**Propósito:** Centraliza la comunicación entre objetos para reducir dependencias directas.

**Ejemplo en Java:**

```
interface Mediator { void enviar(String mensaje, Colega emisor); }

class MediatorConcreto implements Mediator {
    private List<Colega> colegas = new ArrayList<>();
    public void registrar(Colega c) { colegas.add(c); }
    public void enviar(String mensaje, Colega emisor) {
        for (Colega c : colegas) if (c != emisor) c.recibir(mensaje);
    }
}

class Colega {
    private Mediator mediador;
    public Colega(Mediador m) { this.mediador = m; }
    public void enviar(String m) { mediador.enviar(m, this); }
    public void recibir(String m) { System.out.println("Recibido: " + m); }
}
```

---

## 17. Memento

**Propósito:** Guarda y restaura el estado interno de un objeto sin violar su encapsulación.

**Ejemplo en Java:**

```
class Memento {
    private final String estado;
    public Memento(String e) { estado = e; }
    public String getEstado() { return estado; }
}

class Originador {
    private String estado;
    public Memento guardar() { return new Memento(estado); }
    public void restaurar(Memento m) { estado = m.getEstado(); }
}
```

---

## 18. Observer

**Propósito:** Define una dependencia uno-a-muchos: cuando un objeto cambia, notifica a sus observadores.

**Ejemplo en Java:**

```
interface Observador { void actualizar(String msg); }

class Sujeto {
    private List<Observador> obs = new ArrayList<>();
    public void agregar(Observador o) { obs.add(o); }
```

```
public void notificar(String msg) { obs.forEach(o ->
o.actualizar(msg)); }
}
```

---

## 19. State

**Propósito:** Permite cambiar el comportamiento de un objeto cuando cambia su estado interno.

**Ejemplo en Java:**

```
interface Estado { void manejar(); }

class EstadoA implements Estado {
    public void manejar() { System.out.println("Estado A"); }
}

class Contexto {
    private Estado estado;
    public void setEstado(Estado e) { estado = e; }
    public void ejecutar() { estado.manejar(); }
}
```

---

## 20. Strategy

**Propósito:** Define una familia de algoritmos intercambiables en tiempo de ejecución.

**Ejemplo en Java:**

```
interface Estrategia { int operar(int a, int b); }

class Sumar implements Estrategia { public int operar(int a, int b) {
return a + b; } }

class Contexto {
    private Estrategia estrategia;
    public void setEstrategia(Estrategia e) { estrategia = e; }
    public int ejecutar(int a, int b) { return estrategia.operar(a, b); }
}
```

---

## 21. Template Method

**Propósito:** Define la estructura de un algoritmo dejando que las subclases redefinan algunos pasos.

**Ejemplo en Java:**

```
abstract class Juego {
    abstract void inicializar();
    abstract void jugar();
    abstract void terminar();

    public final void jugarPartida() {
        inicializar();
        jugar();
        terminar();
    }
}
```

---

## 22. Visitor

**Propósito:** Permite definir nuevas operaciones sobre una jerarquía de clases sin modificarlas.

### Ejemplo en Java:

```
interface Visitante { void visitar(ElementoConcreto e); }

interface Elemento {
    void aceptar(Visitante v);
}

class ElementoConcreto implements Elemento {
    public void aceptar(Visitante v) { v.visitar(this); }
}
```