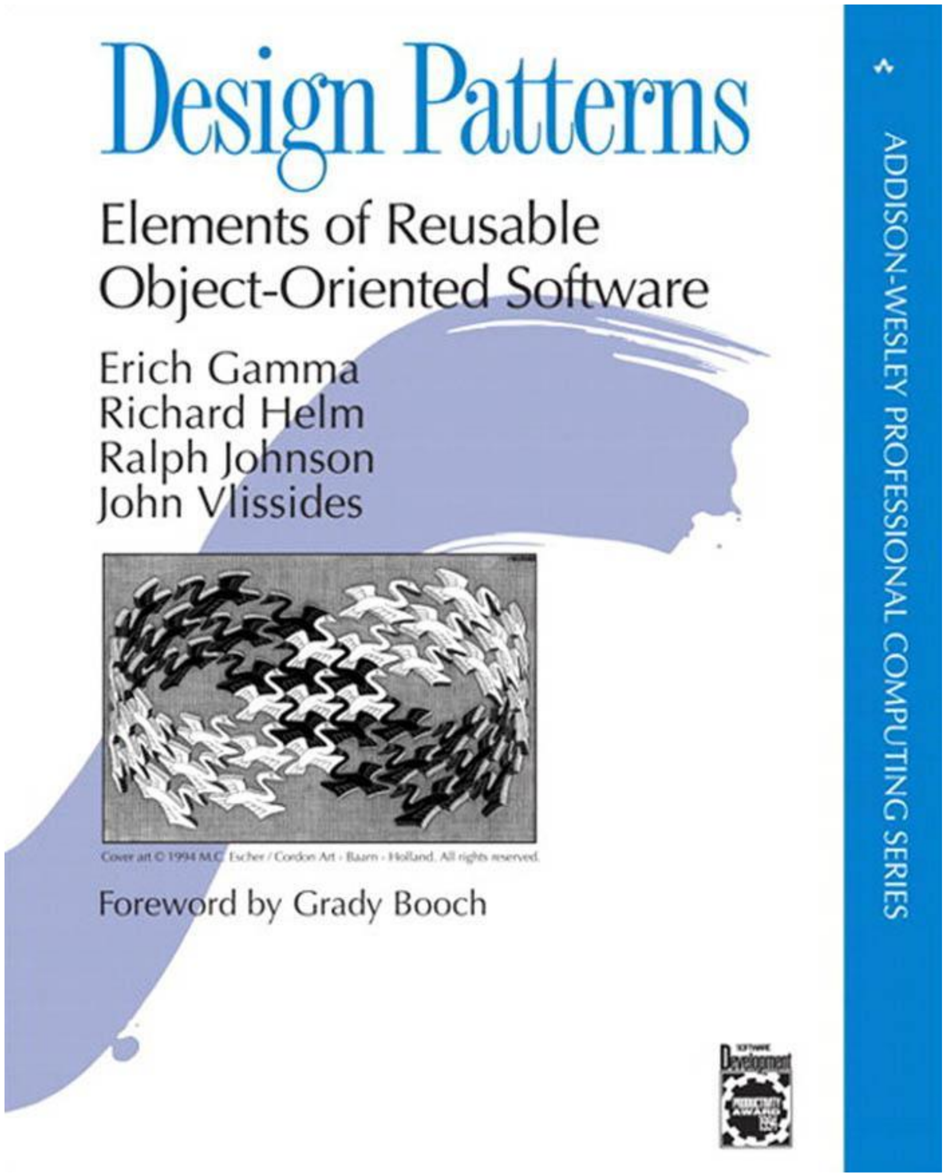
Patrones de diseño

José Galaviz



Design Patterns

Elements of Reusable Object-Oriented Software

Erich Gamma
Richard Helm
Ralph Johnson
John Vlissides

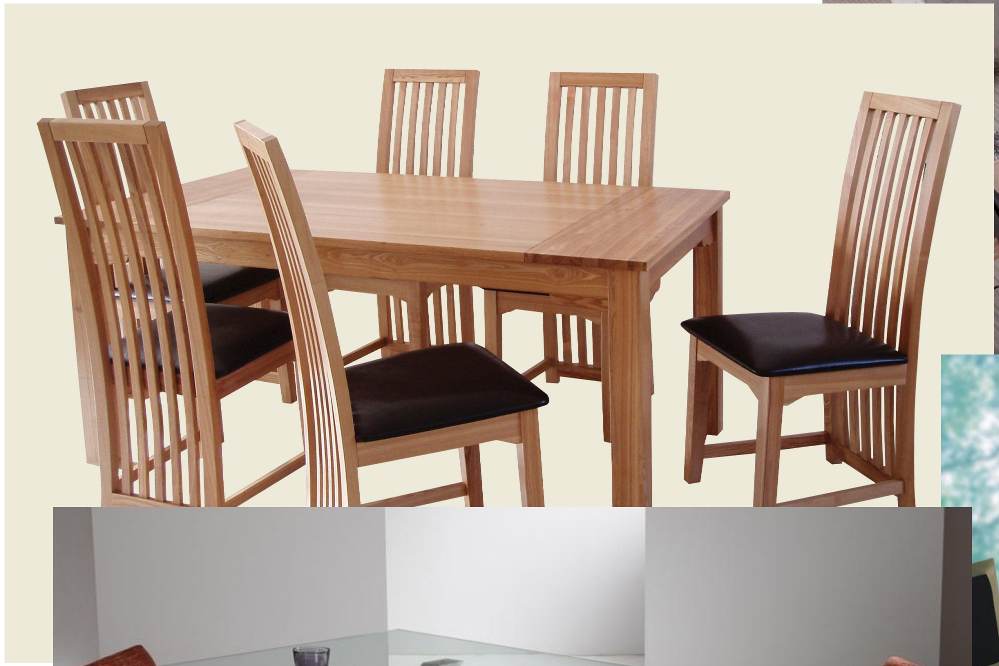
Cover art © 1994 M.C. Escher / Cordon Art - Baarn - Holland. All rights reserved.

Foreword by Grady Booch

ADDISON-WESLEY PROFESSIONAL COMPUTING SERIES

GoF

¿Qué son?



¿Qué son?

- Soluciones recurrentes a ciertos problemas de diseño.
- Abstracciones en un nivel superior al de las clases o componentes.
- Soluciones arquitectónicas usuales.

De GoF

- Cada patrón de diseño sistemáticamente nombra, explica y evalúa una solución de diseño usada con frecuencia en sistemas orientados a objetos.
- El objetivo es hacer diseños reutilizables en diferentes circunstancias

En síntesis

- Recetas para resolver problemas comunes de diseño en programación con objetos a un alto nivel de abstracción.
- Favorecen la reusabilidad del código porque son recetas generales.
- Proporcionan un marco conceptual común, un estándar para comunicarse entre programadores.

Lo ideal

- El software construido con un conjunto de módulos que son como cajas negras que cumplen su función.
- Bajo acoplamiento.
- Reutilizables.
- Extensibles.
- Con responsabilidades únicas.
- Encapsuladas.
- Utilizables a través de una interfaz sin tener que saber de su implementación.

Clasificación

- Patrones creacionales.
- Patrones estructurales.
- Patrones de comportamiento (conductuales).

Patrones creacionales

Definen los mecanismos usados para la creación de objetos.

- Abstract Factory.
- Builder.
- Factory method
- Prototype.
- Singleton

Abstract Factory

- Provee de una interfaz para la creación de objetos de ciertas clases relacionadas, sin especificar el tipo concreto.
- Es una fábrica que construye fábricas.
- Estas fábricas construyen objetos concretos de tipos relacionados.

Ejemplo

- Tienes una fábrica de mamíferos que puede construir objetos como perros y gatos.
- Tienes una fábrica de reptiles que puede construir objetos como serpientes y tiranosaurios.
- Podemos tener una fábrica abstracta que regrese estos tipos de fábricas a través de una interfaz común (polimorfismo).

```java
public class AbstractFactory {
     public SpeciesFactory getSpeciesFactory(String type) {
          if ("mammal".equals(type)) {
               return new MammalFactory();
          } else {
               return new ReptileFactory();
          }
     }
}
```

```java
public abstract class SpeciesFactory {
     public abstract Animal getAnimal(String type);
}
```

```java
public class MammalFactory extends SpeciesFactory {
     @Override
     public Animal getAnimal(String type) {
          if ("dog".equals(type)) {
               return new Dog();
          } else {
               return new Cat();
          }
     }
}
```

```java
public class ReptileFactory extends SpeciesFactory {
     @Override
     public Animal getAnimal(String type) {
          if ("snake".equals(type)) {
               return new Snake();
          } else {
               return new Tyrannosaurus();
          }
     }
}
```

```java
public abstract class Animal {
     public abstract String makeSound();
}


public class Cat extends Animal {
     @Override
     public String makeSound() {
          return "Meow";
     }
}


public class Dog extends Animal {
     @Override
     public String makeSound() {
          return "Woof";
     }
}
```

```java
public class Tyrannosaurus extends
Animal {
     @Override
     public String makeSound() {
          return "Roar";
     }
}


public class Snake extends Animal {
     @Override
     public String makeSound() {
          return "Hiss";
     }

}
```

```java
public class Demo {

    public static void main(String[] args) {
        AbstractFactory abstractFactory = new AbstractFactory();

        SpeciesFactory speciesFactory1 = abstractFactory.getSpeciesFactory("reptile");
        Animal a1 = speciesFactory1.getAnimal("tyrannosaurus");
        System.out.println("a1 sound: " + a1.makeSound());
        Animal a2 = speciesFactory1.getAnimal("snake");
        System.out.println("a2 sound: " + a2.makeSound());

        SpeciesFactory speciesFactory2 = abstractFactory.getSpeciesFactory("mammal");
        Animal a3 = speciesFactory2.getAnimal("dog");
        System.out.println("a3 sound: " + a3.makeSound());
        Animal a4 = speciesFactory2.getAnimal("cat");
        System.out.println("a4 sound: " + a4.makeSound());

    }

}
```

```
jose@gwen:~/TMP/DesPatt/AbstractFactory$ java Demo

a1 sound: Roar
a2 sound: Hiss
a3 sound: Woof
a4 sound: Meow
```

Builder

Abstrae los pasos necesarios para la construcción de diferentes objetos relacionados. La implementación concreta de estos pasos genera instancias de diferente clase.

Elementos

Este patrón siempre contiene cuatro elementos esenciales:

- Constructor. La interfaz que indica que se puede construir.
- Director. Que construye los diferentes tipos del producto.
- Constructores concretos. Implementan al constructor y construyen productos concretos.
- Producto. Define lo que se va a construir.

Ejemplo

- Creación de diferentes comidas de restaurante.
- Clase Meal que contiene lo que debe tener toda comida: bebida, plato principal y acompañamiento.
- Una interfaz MealBuilder para especificar que se pueden construir los diferentes elementos mencionados arriba.
- Constructores concretos de diferentes tipos de comidas que implementan la interfaz previa
- Un MealDirector que recibe como parámetro un MealBuilder de alguno de los tipos concretos y arma la comida.

```
public class Meal {
     private String drink;
     private String mainCourse;
     private String side;
     public String getDrink() {
          return drink;
     }
     public void setDrink(String drink) {
          this.drink = drink;
     }
     public String getMainCourse() {
          return mainCourse;
     }
     public void setMainCourse(String mainCourse) {
          this.mainCourse = mainCourse;
     }
     public String getSide() {
          return side;
     }
     public void setSide(String side) {
          this.side = side;
     }
     public String toString() {
          return "drink:" + drink + ", main course:" + mainCourse + ", side:" + side;
     }
}
```

```java
public interface MealBuilder {
     public void buildDrink();
     public void buildMainCourse();
     public void buildSide();
     public Meal getMeal();
}
```

```java
public class MealDirector {

     private MealBuilder mealBuilder = null;

     public MealDirector(MealBuilder mealBuilder) {
          this.mealBuilder = mealBuilder;
     }

     public void constructMeal() {
          mealBuilder.buildDrink();
          mealBuilder.buildMainCourse();
          mealBuilder.buildSide();
     }

     public Meal getMeal() {
          return mealBuilder.getMeal();
     }

}
```

```java
public class ItalianMealBuilder
implements MealBuilder {
	private Meal meal;
	public ItalianMealBuilder() {
		meal = new Meal();
	}
	@Override
	public void buildDrink() {
		meal.setDrink("red wine");
	}
	@Override
	public void buildMainCourse() {
		meal.setMainCourse("pizza");
	}
	@Override
	public void buildSide() {
		meal.setSide("bread");
	}
	@Override
	public Meal getMeal() {
		return meal;
	}
}
```

```java
public class JapaneseMealBuilder implements
MealBuilder {
	private Meal meal;
	public JapaneseMealBuilder() {
		meal = new Meal();
	}
	@Override
	public void buildDrink() {
		meal.setDrink("sake");
	}
	@Override
	public void buildMainCourse() {
		meal.setMainCourse("chicken teriyaki");
	}
	@Override
	public void buildSide() {
		meal.setSide("miso soup");
	}
	@Override
	public Meal getMeal() {
		return meal;
	}
}
```

```java
public class Demo {

     public static void main(String[] args) {

          MealBuilder mealBuilder = new ItalianMealBuilder();
          MealDirector mealDirector = new MealDirector(mealBuilder);
          mealDirector.constructMeal();
          Meal meal = mealDirector.getMeal();
          System.out.println("meal is: " + meal);

          mealBuilder = new JapaneseMealBuilder();
          mealDirector = new MealDirector(mealBuilder);
          mealDirector.constructMeal();
          meal = mealDirector.getMeal();
          System.out.println("meal is: " + meal);
     }

}
```

```
jose@gwen:~/TMP/DesPatt/Builder$ java Demo

meal is: drink:red wine, main course:pizza, side:bread
meal is: drink:sake, main course:chicken teriyaki, side:miso
soup
```

Factory Method

- Define una interfaz para la creación de objetos de ciertos tipos, pero difiere la implementación a las subclases.
- Encapsula el código de creación del objeto.
- Una clase que fabrica objetos regresa instancias construidas de diferentes tipos dependiendo de los parámetros.

Ejemplo

- Una clase que fabrica animales.
- Regresa un tipo de animal particular dependiendo de cierto dato de entrada.

```java
public abstract class Animal {
    public abstract String makeSound();
}


public class Dog extends Animal {

    @Override
    public String makeSound() {
        return "Woof";
    }

}


public class Cat extends Animal {

    @Override
    public String makeSound() {
        return "Meow";
    }

}
```

```java
public class AnimalFactory {
    public Animal getAnimal(String type) {
        if ("canine".equals(type)) {
            return new Dog();
        } else {
            return new Cat();
        }
    }
}


public class Demo {
    public static void main(String[] args) {
        AnimalFactory animalFactory =  new AnimalFactory();
        Animal a1 = animalFactory.getAnimal( "feline");
        System.out.println("a1 sound: " + a1.makeSound());
        Animal a2 = animalFactory.getAnimal( "canine");
        System.out.println("a2 sound: " + a2.makeSound());
    }
}
```

```
jose@gwen:~/TMP/DesPatt/FactoryMethod$ java Demo
a1 sound: Meow
a2 sound: Woof
```

Prototype

Crea un objeto a partir de una instancia de él previamente existente.

Ejemplo

Un método clone para duplicar un objeto de tipo Person.

```java
public interface Prototype {
    public Prototype doClone();
}


public class Person implements Prototype {
    String name;
    public Person(String name) {
        this.name = name;
    }
    @Override
    public Prototype doClone() {
        return new Person(name);
    }
    public String toString() {
        return "This person is named " + name;
    }
}
```

Si ejecutamos:

```
Person person1 = new Person("Fred");
System.out.println("person 1:" + person1);
Person person2 = (Person) person1.doClone();
System.out.println("person 2:" + person2);
```

```
jose@gwen:~/TMP/DesPatt/Prototype$ java Demo
person 1:This person is named Fred
person 2:This person is named Fred
```

Singleton

- Restringe la instanciación de una clase para que sólo haya un objeto de ella.
- Se provee de acceso global a esa instancia.

Ejemplo

- Clase con un constructor privado y método de construcción estático.

```java
public class SingletonExample {
    private static SingletonExample singletonExample = null;
    private SingletonExample() {
    }
    public static SingletonExample getInstance() {
        if (singletonExample == null) {
            singletonExample = new SingletonExample();
        }
        return singletonExample;
    }
    public void sayHello() {
        System.out.println("Hello");
    }
}


public class Demo {
    public static void main(String[] args) {
        SingletonExample singletonExample = SingletonExample.getInstance();
        singletonExample.sayHello();
    }

}
```

Patrones estructurales

- Adapter.
- Composite.
- Bridge.
- Facade.
- Decorator.
- Proxy.
- Flyweight.

Adapter

- Es un wrapper. Envuelve un objeto con otro para adaptar la interfaz del primero.
- Adquiere una de dos formas:
  - El adaptador extiende al adaptado añadiendo la funcionalidad deseada.
  - El adaptador contiene una instancia de la clase adaptada ofreciendo una interfaz con la funcionalidad deseada.
- ¿Un objeto hace lo que quieres, pero no con la interfaz que quieres? Usa un wrapper.

Adaptado

```
public class CelsiusReporter {

    double temperatureInC;

    public CelsiusReporter() {
    }

    public double getTemperature() {
        return temperatureInC;
    }

    public void setTemperature(double temperatureInC) {
        this.temperatureInC = temperatureInC;
    }

}
```

Interfaz deseada

```
public interface TemperatureInfo {

    public double getTemperatureInF();

    public void setTemperatureInF(double temperatureInF);

    public double getTemperatureInC();

    public void setTemperatureInC(double temperatureInC);

}
```

```java
public class TemperatureClassReporter extends CelsiusReporter implements TemperatureInfo {
	@Override
	public double getTemperatureInC() {
		return temperatureInC;
	}
	@Override
	public double getTemperatureInF() {
		return cToF(temperatureInC);
	}
	@Override
	public void setTemperatureInC(double temperatureInC) {
		this.temperatureInC = temperatureInC;
	}
	@Override
	public void setTemperatureInF(double temperatureInF) {
		this.temperatureInC = fToC(temperatureInF);
	}
	private double fToC(double f) {
		return ((f - 32) * 5 / 9);
	}
	private double cToF(double c) {
		return ((c * 9 / 5) + 32);
	}
}
```

Adaptador (herencia)

```java
public class TemperatureObjectReporter implements TemperatureInfo {
	CelsiusReporter celsiusReporter;
	public TemperatureObjectReporter() {
		celsiusReporter = new CelsiusReporter();
	}
	@Override
	public double getTemperatureInC() {
		return celsiusReporter.getTemperature();
	}
	@Override
	public double getTemperatureInF() {
		return cToF(celsiusReporter.getTemperature());
	}
	@Override
	public void setTemperatureInC(double temperatureInC) {
		celsiusReporter.setTemperature(temperatureInC);
	}
	@Override
	public void setTemperatureInF(double temperatureInF) {
		celsiusReporter.setTemperature(fToC(temperatureInF));
	}
	private double fToC(double f) {
		return ((f - 32) * 5 / 9);
	}
	private double cToF(double c) {
		return ((c * 9 / 5) + 32);
	}
}
```

Adaptador (composición)

```java
public class AdapterDemo {
     public static void main(String[] args) {

          // class adapter
          System.out.println("class adapter test");
          TemperatureInfo tempInfo = new TemperatureClassReporter();
          testTempInfo(tempInfo);

          // object adapter
          System.out.println("\nobject adapter test");
          tempInfo = new TemperatureObjectReporter();
          testTempInfo(tempInfo);

     }
     public static void testTempInfo(TemperatureInfo tempInfo) {
          tempInfo.setTemperatureInC(0);
          System.out.println("temp in C:" + tempInfo.getTemperatureInC());
          System.out.println("temp in F:" + tempInfo.getTemperatureInF());

          tempInfo.setTemperatureInF(85);
          System.out.println("temp in C:" + tempInfo.getTemperatureInC());
          System.out.println("temp in F:" + tempInfo.getTemperatureInF());
     }
}
```

Programa

```
jose@gwen:~/Programacion/DesPatt/Adapter$ java AdapterDemo
class adapter test
temp in C:0.0
temp in F:32.0
temp in C:29.444444444444443
temp in F:85.0
object adapter test
temp in C:0.0
temp in F:32.0
temp in C:29.444444444444443
temp in F:85.0
```

Salida

Composite

Agrupa objetos de clases diferentes para proveerlos de una interfaz uniforme.

Cuando

¿Tienes varios objetos de clases diferentes pero que se usan más o menos igual casi siempre? Define una clase con el “común denominador”

El común denominador

```java
public interface Component {

    public void sayHello();

    public void sayGoodbye();

}
```

Hasta abajo en la jerarquía

```java
public class Leaf implements Component {
    String name;
    public Leaf(String name) {
        this.name = name;
    }
    @Override
    public void sayHello() {
        System.out.println(name + " leaf says hello");
    }
    @Override
    public void sayGoodbye() {
        System.out.println(name + " leaf says goodbye");
    }
}
```

```
import java.util.ArrayList;
import java.util.List;
public class Composite implements Component {
    List<Component> components = new ArrayList<Component>();
    @Override
    public void sayHello() {
        for (Component component : components) {
            component.sayHello();
        }
    }
    @Override
    public void sayGoodbye() {
        for (Component component : components) {
            component.sayGoodbye();
        }
    }
    public void add(Component component) {
        components.add(component);
    }
    public void remove(Component component) {
        components.remove(component);
    }
    public List<Component> getComponents() {
        return components;
    }
    public Component getComponent(int index) {
        return components.get(index);
    }
}
```

```java
public class CompositeDemo {
    public static void main(String[] args) {
        Leaf leaf1 = new Leaf("Bob");
        Leaf leaf2 = new Leaf("Fred");
        Leaf leaf3 = new Leaf("Sue");
        Leaf leaf4 = new Leaf("Ellen");
        Leaf leaf5 = new Leaf("Joe");
        Composite composite1 = new Composite();
        composite1.add(leaf1);
        composite1.add(leaf2);
        Composite composite2 = new Composite();
        composite2.add(leaf3);
        composite2.add(leaf4);
        Composite composite3 = new Composite();
        composite3.add(composite1);
        composite3.add(composite2);
        composite3.add(leaf5);
        System.out.println("Calling 'sayHello' on leaf1");
        leaf1.sayHello();
        System.out.println("\nCalling 'sayHello' on composite1");
        composite1.sayHello();
        System.out.println("\nCalling 'sayHello' on composite2");
        composite2.sayHello();
        System.out.println("\nCalling 'sayGoodbye' on composite3");
        composite3.sayGoodbye();

    }
}
```

Demo

```
jose@gwen:~/Programacion/DesPatt/Composite$ java CompositeDemo
Calling 'sayHello' on leaf1
Bob leaf says hello

Calling 'sayHello' on composite1
Bob leaf says hello
Fred leaf says hello

Calling 'sayHello' on composite2
Sue leaf says hello
Ellen leaf says hello

Calling 'sayGoodbye' on composite3
Bob leaf says goodbye
Fred leaf says goodbye
Sue leaf says goodbye
Ellen leaf says goodbye
Joe leaf says goodbye
```

Bridge

Provee de mecanismos para interactuar con otro objeto.

El objetivo es desacoplar la interfaz del objeto de su implementación para que ambas puedan cambiar independientemente.

Cuando

Ejemplo. Tienes una clase que hace lo que quieres pero no depende de ti y ya va en la versión 3, cambiando de interfaz. Haces tu interfaz que se monta en la otra.

¿La abstracción es invariante, pero la interfaz no? Entonces usa un bridge.

```java
public abstract class Vehicle {
    Engine engine;
    int weightInKilos;
    public abstract void drive();
    public void setEngine(Engine engine) {
        this.engine = engine;
    }
    public void reportOnSpeed(int horsepower) {
        int ratio = weightInKilos / horsepower;
        if (ratio < 3) {
            System.out.println("The vehicle is going at a fast speed.");
        } else if ((ratio >= 3) && (ratio < 8)) {
            System.out.println("The vehicle is going an average speed.");
        } else {
            System.out.println("The vehicle is going at a slow speed.");
        }
    }
}
```

```java
public class BigBus extends Vehicle {
     public BigBus(Engine engine) {
          this.weightInKilos = 3000;
          this.engine = engine;
     }
     @Override
     public void drive() {
          System.out.println("\nThe big bus is driving");
          int horsepower = engine.go();
          reportOnSpeed(horsepower);
     }
}

public class SmallCar extends Vehicle {
     public SmallCar(Engine engine) {
          this.weightInKilos = 600;
          this.engine = engine;
     }
     @Override
     public void drive() {
          System.out.println("\nThe small car is driving");
          int horsepower = engine.go();
          reportOnSpeed(horsepower);
     }
}
```

```java
public interface Engine {
     public int go();
}

public class BigEngine implements Engine {
     int horsepower;
     public BigEngine() {
          horsepower = 350;
     }
     @Override
     public int go() {
          System.out.println("The big engine is running");
          return horsepower;
     }
}

public class SmallEngine implements Engine {
     int horsepower;
     public SmallEngine() {
          horsepower = 100;
     }
     @Override
     public int go() {
          System.out.println("The small engine is running");
          return horsepower;
     }
}
```

```
public class BridgeDemo {

    public static void main(String[] args) {

        Vehicle vehicle = new BigBus(new SmallEngine());
        vehicle.drive();
        vehicle.setEngine(new BigEngine());
        vehicle.drive();

        vehicle = new SmallCar(new SmallEngine());
        vehicle.drive();
        vehicle.setEngine(new BigEngine());
        vehicle.drive();

    }
}
```

```
jose@gwen:~/Programacion/DesPatt/Bridge$ java BridgeDemo

The big bus is driving
The small engine is running
The vehicle is going at a slow speed.

The big bus is driving
The big engine is running
The vehicle is going at a slow speed.

The small car is driving
The small engine is running
The vehicle is going an average speed.

The small car is driving
The big engine is running
The vehicle is going at a fast speed.
```

Facade

Eleva el nivel de abstracción de la interfaz de un objeto(s).

Simplifica la interfaz del objeto (u objetos) al que da la fachada.

```java
public class Class1 {
    public int doSomethingComplicated( int x) {
        return x * x * x;
    }
}

public class Class2 {
    public int doAnotherThing(Class1 class1,  int x) {
        return 2 * class1.doSomethingComplicated(x);
    }
}

public class Class3 {
    public int doMoreStuff(Class1 class1, Class2 class2,  int x) {
        return class1.doSomethingComplicated(x) *
               class2.doAnotherThing(class1, x);
    }
}
```

```java
public class Facade {
    public int cubeX(int x) {
        Class1 class1 = new Class1();
        return class1.doSomethingComplicated(x);
    }
    public int cubeXTimes2(int x) {
        Class1 class1 = new Class1();
        Class2 class2 = new Class2();
        return class2.doAnotherThing(class1, x);
    }
    public int xToSixthPowerTimes2(int x) {
        Class1 class1 = new Class1();
        Class2 class2 = new Class2();
        Class3 class3 = new Class3();
        return class3.doMoreStuff(class1, class2, x);
    }
}
```

```java
public class FacadeDemo {
    public static void main(String[] args) {
        Facade facade = new Facade();
        int x = 3;
        System.out.println("Cube of " + x + ":" +
                                facade.cubeX(3));
        System.out.println("Cube of " + x + " times 2:" +
                                facade.cubeXTimes2(3));
        System.out.println(x + " to sixth power times 2:" +
                                facade.xToSixthPowerTimes2(3));
    }
}
```

```
jose@gwen:~/Programacion/DesPatt/Facade$ java FacadeDemo
Cube of 3:27
Cube of 3 times 2:54
3 to sixth power times 2:1458
```

Decorator

- Añade funcionalidad a un objeto de cierta clase en tiempo de ejecución.
- Es un tipo de objeto que define algunas responsabilidades que se sustentan esencialmente en las de uno previo.
- Suele ser la alternativa a hacer subclases. Hace lo mismo pero con instancias en tiempo de ejecución

```java
public interface Animal {
    public void describe();
}

public class LivingAnimal implements Animal {
    @Override
    public void describe() {
        System.out.println("\nI am an animal.");
    }
}

public abstract class AnimalDecorator implements Animal {
    Animal animal;
    public AnimalDecorator(Animal animal) {
        this.animal = animal;
    }
}
```

```java
public class LegDecorator extends AnimalDecorator {
    public LegDecorator(Animal animal) {
        super(animal);
    }
    @Override
    public void describe() {
        animal.describe();
        System.out.println("I have legs.");
        dance();
    }
    public void dance() {
        System.out.println("I can dance.");
    }
}
```

```java
public class WingDecorator extends AnimalDecorator {
    public WingDecorator(Animal animal) {
        super(animal);
    }
    @Override
    public void describe() {
        animal.describe();
        System.out.println("I have wings.");
        fly();
    }
    public void fly() {
        System.out.println("I can fly.");
    }
}
```

```java
public class GrowlDecorator extends AnimalDecorator {
    public GrowlDecorator(Animal animal) {
        super(animal);
    }
    @Override
    public void describe() {
        animal.describe();
        growl();
    }
    public void growl() {
        System.out.println("Grrrrr.");
    }
}
```

```java
public class DecoratorDemo {
    public static void main(String[] args) {
        Animal animal = new LivingAnimal();
        animal.describe();

        animal = new LegDecorator(animal);
        animal.describe();

        animal = new WingDecorator(animal);
        animal.describe();

        animal = new GrowlDecorator(animal);
        animal = new GrowlDecorator(animal);
        animal.describe();
    }
}
```

```
jose@gwen:~/Programacion/DesPatt/Decorator$ java DecoratorDemo
I am an animal.
I am an animal.
I have legs.
I can dance.
I am an animal.
I have legs.
I can dance.
I have wings.
I can fly.
I am an animal.
I have legs.
I can dance.
I have wings.
I can fly.
Grrrrr.
Grrrrr.
```

Proxy

- Es un tipo de objeto que controla la interacción con otro tipo de objeto.
- No añade funcionalidad.

```java
import java.util.Date;
public abstract class Thing {
    public void sayHello() {
        System.out.println(this.getClass().getSimpleName() +
                                  " says howdy at " + new Date());
    }
}
```

```java
public class FastThing extends Thing {
    public FastThing() {
    }
}

public class SlowThing extends Thing {
    public SlowThing() {
        try {
            Thread.sleep(5000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}
```

```java
import java.util.Date;
public class Proxy {
    SlowThing slowThing;
    public Proxy() {
        System.out.println("Creating proxy at " + new Date());
    }
    public void sayHello() {
        if (slowThing == null) {
            slowThing = new SlowThing();
        }
        slowThing.sayHello();
    }
}
```

```java
public class ProxyDemo {
    public static void main(String[] args) {
        Proxy proxy = new Proxy();

        FastThing fastThing = new FastThing();
        fastThing.sayHello();

        proxy.sayHello();
    }
}
```

```
jose@gwen:~/Programacion/DesPatt/Proxy$ java ProxyDemo

Creating proxy at Sun Sep 20 23:24:15 CDT 2015

FastThing says howdy at Sun Sep 20 23:24:15 CDT 2015

SlowThing says howdy at Sun Sep 20 23:24:20 CDT 2015
```

Diferencias

Adapter: Cambia la interfaz de un objeto para hacerla como la espera el cliente. Se hace pensando en adaptar la interfaz de un tipo concreto.

Decorator: añade funcionalidad a un objeto sin alterarlo.

Bridge: A diferencia del adapter establece la interfaz para una abstracción, podría cambiar en runtime el objeto que conecta. Se hace pensando en adaptar la interfaz necesaria para operar con cierto tipo de abstracciones.

Facade: simplifica la interfaz

Patrones de comportamiento

Chain of responsibility.

Command

Iterator

Visitor

Mediator

Memento

Observer

State

Strategy

Template method

Chain of responsibility

- Las peticiones de servicio se pasan a través de los objetos hasta llegar a alguno que la puede satisfacer (o puede responder).
- Desacopla a los clientes de los servidores.
- Hace posible una misma interfaz para diversos servicios.

```
public abstract class PlanetHandler {
    PlanetHandler successor
    public void setSuccessor(PlanetHandler successor) {
        this.successor = successor;
    }
    public abstract void handleRequest(PlanetEnum request);
}

public enum PlanetEnum {
    MERCURY, VENUS, EARTH, MARS, JUPITER, SATURN, URANUS,
NEPTUNE;
}
```

```java
public class MercuryHandler extends PlanetHandler {
    public void handleRequest(PlanetEnum request) {
        if (request == PlanetEnum.MERCURY) {
            System.out.println("MercuryHandler handles " + request);
            System.out.println("Mercury is hot.\n");
        } else {
            System.out.println("MercuryHandler doesn't handle " + request);
            if (successor != null) {
                successor.handleRequest(request);
            }
        }
    }
}
```

```java
public class VenusHandler extends PlanetHandler {
    public void handleRequest(PlanetEnum request) {
        if (request == PlanetEnum.VENUS) {
            System.out.println("VenusHandler handles " + request);
            System.out.println("Venus is poisonous.\n");
        } else {
            System.out.println("VenusHandler doesn't handle " + request);
            if (successor != null) {
                successor.handleRequest(request);
            }
        }
    }
}
```

```java
public class EarthHandler extends PlanetHandler {
    public void handleRequest(PlanetEnum request) {
        if (request == PlanetEnum.EARTH) {
            System.out.println("EarthHandler handles " + request);
            System.out.println("Earth is comfortable.\n");
        } else {
            System.out.println("EarthHandler doesn't handle " + request);
            if (successor != null) {
                successor.handleRequest(request);
            }
        }
    }
}
```

```java
public class Demo {
    public static void main(String[] args) {
        PlanetHandler chain = setUpChain();
        chain.handleRequest(PlanetEnum.VENUS);
        chain.handleRequest(PlanetEnum.MERCURY);
        chain.handleRequest(PlanetEnum.EARTH);
        chain.handleRequest(PlanetEnum.JUPITER);
    }
    public static PlanetHandler setUpChain() {
        PlanetHandler mercuryHandler =  new MercuryHandler();
        PlanetHandler venusHandler =  new VenusHandler();
        PlanetHandler earthHandler =  new EarthHandler();
        mercuryHandler.setSuccessor(venusHandler);
        venusHandler.setSuccessor(earthHandler);
        return mercuryHandler;
    }
}
```

```
jose@gwen:~/Programacion/DesPatt/Chain$ java Demo
MercuryHandler doesn't handle VENUS
VenusHandler handles VENUS
Venus is poisonous.

MercuryHandler handles MERCURY
Mercury is hot.

MercuryHandler doesn't handle EARTH
VenusHandler doesn't handle EARTH
EarthHandler handles EARTH
Earth is comfortable.

MercuryHandler doesn't handle JUPITER
VenusHandler doesn't handle JUPITER
EarthHandler doesn't handle JUPITER
```

Command

Encapsula en un objeto lo que realmente es una tarea a ejecutar.

Facilita parametrizar y manipular las tareas. Sobre todo cuando quien las recibe no sabe cómo hacerlas.

```
public interface Command {

    public void execute();

}
public class LunchCommand implements Command {

    Lunch lunch;

    public LunchCommand(Lunch lunch) {
        this.lunch = lunch;
    }

    @Override
    public void execute() {
        lunch.makeLunch();
    }

}
```

```java
public class DinnerCommand implements Command {

    Dinner dinner;

    public DinnerCommand(Dinner dinner) {
        this.dinner = dinner;
    }

    @Override
    public void execute() {
        dinner.makeDinner();
    }

}
```

```java
public class Lunch {

    public void makeLunch() {
        System.out.println("Lunch is being made");
    }

}

public class Dinner {

    public void makeDinner() {
        System.out.println("Dinner is being made");
    }

}
```

```java
public class MealInvoker {
     Command command;
     public MealInvoker(Command command) {
          this.command = command;
     }
     public void setCommand(Command command) {
          this.command = command;
     }
     public void invoke() {
          command.execute();
     }
}
```

```java
public class Demo {
    public static void main(String[] args) {
        Lunch lunch = new Lunch(); // receiver
        Command lunchCommand = new LunchCommand(lunch);  // concrete command
        Dinner dinner = new Dinner(); // receiver
        Command dinnerCommand = new DinnerCommand(dinner);  // concrete command
        MealInvoker mealInvoker = new MealInvoker(lunchCommand);  // invoker
        mealInvoker.invoke();
        mealInvoker.setCommand(dinnerCommand);
        mealInvoker.invoke();
    }
}
```

```
jose@gwen:~/Programacion/DesPatt/Command$ java Demo
Lunch is being made
Dinner is being made
```

Iterator

Provee acceso a los elementos de una estructura abstrayendo la implementación de esta.

```java
public class Item {

    String name;
    float price;

    public Item(String name, float price) {
        this.name = name;
        this.price = price;
    }

    public String toString() {
        return name + ": $" + price;
    }
}
```

```java
import java.util.ArrayList;
import java.util.Iterator;
import java.util.List;
public class Menu {
     List<Item> menuItems;
     public Menu() {
          menuItems = new ArrayList<Item>();
     }
     public void addItem(Item item) {
          menuItems.add(item);
     }
     public Iterator<Item> iterator() {
          return new MenuIterator();
     }
//..
}
```

```java
class MenuIterator implements Iterator<Item> {
    int currentIndex = 0;
    @Override
    public boolean hasNext() {
        if (currentIndex >= menuItems.size()) {
            return false;
        } else {
            return true;
        }
    }
    @Override
    public Item next() {
        return menuItems.get(currentIndex++);
    }
    @Override
    public void remove() {
        menuItems.remove(--currentIndex);
    }
}
```

```java
import java.util.Iterator;
public class Demo {
     public static void main(String[] args) {
          Item i1 = new Item("spaghetti", 7.50f);
          Item i2 = new Item("hamburger", 6.00f);
          Item i3 = new Item("chicken sandwich", 6.50f);
          Menu menu = new Menu();
          menu.addItem(i1);
          menu.addItem(i2);
          menu.addItem(i3);
          System.out.println("Displaying Menu:");
          Iterator<Item> iterator = menu.iterator();
          while (iterator.hasNext()) {
               Item item = iterator.next();
               System.out.println(item);
          }
          System.out.println("\nRemoving last item returned");
          iterator.remove();
          System.out.println("\nDisplaying Menu:");
          iterator = menu.iterator();
          while (iterator.hasNext()) {
               Item item = iterator.next();
               System.out.println(item);
          }
     }
}
```

```
jose@gwen:~/Programacion/DesPatt/Iterator$ java Demo
Displaying Menu:
spaghetti: $7.5
hamburger: $6.0
chicken sandwich: $6.5

Removing last item returned

Displaying Menu:
spaghetti: $7.5
hamburger: $6.0
```

Visitor

Provee de los mecanismos para representar una operación a ser realizada sobre cada uno de los elementos de una estructura.

Permite abstraer lo que se hace cuando se “visita”.

```java
import java.util.List;
public interface NumberVisitor {
    public void visit(TwoElement twoElement);
    public void visit(ThreeElement threeElement);
    public void visit(List<NumberElement> elementList);
}

public interface NumberElement {
    public void accept(NumberVisitor visitor);
}
```

```java
public class TwoElement implements NumberElement {
    int a;
    int b;
    public TwoElement(int a, int b) {
        this.a = a;
        this.b = b;
    }
    @Override
    public void accept(NumberVisitor visitor) {
        visitor.visit(this);
    }
}
```

```java
public class ThreeElement implements NumberElement {
    int a;
    int b;
    int c;
    public ThreeElement(int a, int b, int c) {
        this.a = a;
        this.b = b;
        this.c = c;
    }
    @Override
    public void accept(NumberVisitor visitor) {
        visitor.visit(this);
    }
}
```

```java
import java.util.List;
public class SumVisitor implements NumberVisitor {
    @Override
    public void visit(TwoElement twoElement) {
        int sum = twoElement.a + twoElement.b;
        System.out.println(twoElement.a + "+" + twoElement.b + "=" + sum);
    }
    @Override
    public void visit(ThreeElement threeElement) {
        int sum = threeElement.a + threeElement.b + threeElement.c;
        System.out.println(threeElement.a + "+" + threeElement.b + "+"
                            + threeElement.c + "=" + sum);
    }
    @Override
    public void visit(List<NumberElement> elementList) {
        for (NumberElement ne : elementList) {
            ne.accept(this);
        }
    }
}
```

```java
import java.util.List;
public class TotalSumVisitor implements NumberVisitor {
     int totalSum = 0;
     @Override
     public void visit(TwoElement twoElement) {
          int sum = twoElement.a + twoElement.b;
          System.out.println("Adding " + twoElement.a + "+" + twoElement.b + "="
                                   + sum + " to total");
          totalSum += sum;
     }
     @Override
     public void visit(ThreeElement threeElement) {
          int sum = threeElement.a + threeElement.b + threeElement.c;
          System.out.println("Adding " + threeElement.a + "+" + threeElement.b + "+"
                                   + threeElement.c + "=" + sum + " to total");
          totalSum += sum;
     }
     @Override
     public void visit(List<NumberElement> elementList) {
          for (NumberElement ne : elementList) {
               ne.accept(this);
          }
     }
     public int getTotalSum() {
          return totalSum;
     }
}
```

```java
import java.util.ArrayList;
import java.util.List;
public class Demo {
     public static void main(String[] args) {
          TwoElement two1 = new TwoElement(3, 3);
          TwoElement two2 = new TwoElement(2, 7);
          ThreeElement three1 = new ThreeElement(3, 4, 5);
          List<NumberElement> numberElements = new ArrayList<NumberElement>();
          numberElements.add(two1);
          numberElements.add(two2);
          numberElements.add(three1);
          System.out.println("Visiting element list with SumVisitor");
          NumberVisitor sumVisitor = new SumVisitor();
          sumVisitor.visit(numberElements);
          System.out.println("\nVisiting element list with TotalSumVisitor");
          TotalSumVisitor totalSumVisitor = new TotalSumVisitor();
          totalSumVisitor.visit(numberElements);
          System.out.println("Total sum:" + totalSumVisitor.getTotalSum());

     }
}
```

```
jose@gwen:~/Programacion/DesPatt/Visitor$ java Demo
Visiting element list with SumVisitor
3+3=6
2+7=9
3+4+5=12

Visiting element list with TotalSumVisitor
Adding 3+3=6 to total
Adding 2+7=9 to total
Adding 3+4+5=12 to total
Total sum:27
```

Mediator

Es un tipo de objeto que encapsula la manera en la que interactúan los objetos de otro tipo.

Centraliza la comunicación con otro objeto.

```java
public class Mediator {
    Buyer swedishBuyer;
    Buyer frenchBuyer;
    AmericanSeller americanSeller;
    DollarConverter dollarConverter;
    public Mediator() {
    }
    public void registerSwedishBuyer(SwedishBuyer swedishBuyer) {
        this.swedishBuyer = swedishBuyer;
    }
    public void registerFrenchBuyer(FrenchBuyer frenchBuyer) {
        this.frenchBuyer = frenchBuyer;
    }
    public void registerAmericanSeller(AmericanSeller americanSeller) {
        this.americanSeller = americanSeller;
    }
    public void registerDollarConverter(DollarConverter dollarConverter) {
        this.dollarConverter = dollarConverter;
    }
    public boolean placeBid(float bid, String unitOfCurrency) {
        float dollarAmount = dollarConverter.convertCurrencyToDollars(bid,
                                                      unitOfCurrency);
        return americanSeller.isBidAccepted(dollarAmount);
    }
}
```

```java
public class Buyer {
    Mediator mediator;
    String unitOfCurrency;
    public Buyer(Mediator mediator, String unitOfCurrency) {
        this.mediator = mediator;
        this.unitOfCurrency = unitOfCurrency;
    }
    public boolean attemptToPurchase(float bid) {
        System.out.println("Buyer attempting a bid of "
                                   + bid + " " + unitOfCurrency);
        return mediator.placeBid(bid, unitOfCurrency);
    }
}
```

```java
public class SwedishBuyer extends Buyer {
    public SwedishBuyer(Mediator mediator) {
        super(mediator, "krona");
        this.mediator.registerSwedishBuyer(this);
    }
}

public class FrenchBuyer extends Buyer {
    public FrenchBuyer(Mediator mediator) {
        super(mediator, "euro");
        this.mediator.registerFrenchBuyer(this);
    }
}
```

```java
public class AmericanSeller {
    Mediator mediator;
    float priceInDollars;
    public AmericanSeller(Mediator mediator,  float priceInDollars) {
        this.mediator = mediator;
        this.priceInDollars = priceInDollars;
        this.mediator.registerAmericanSeller( this);
    }
    public boolean isBidAccepted(float bidInDollars) {
        if (bidInDollars >= priceInDollars) {
            System.out.println("Seller accepts the bid of "
                                    + bidInDollars +  " dollars\n");
            return true;
        } else {
            System.out.println("Seller rejects the bid of "
                                    + bidInDollars +  " dollars\n");
            return false;
        }
    }
}
```

```java
public class DollarConverter {
     Mediator mediator;
     public static final float DOLLAR_UNIT = 1.0f;
     public static final float EURO_UNIT = 0.7f;
     public static final float KRONA_UNIT = 8.0f;
     public DollarConverter(Mediator mediator) {
          this.mediator = mediator;
          mediator.registerDollarConverter(this);
     }
     private float convertEurosToDollars(float euros) {
          float dollars = euros * (DOLLAR_UNIT / EURO_UNIT);
          System.out.println("Converting " + euros + " euros to "
                                   + dollars + " dollars");
          return dollars;
     }
     private float convertKronorToDollars(float kronor) {
          float dollars = kronor * (DOLLAR_UNIT / KRONA_UNIT);
          System.out.println("Converting " + kronor + " kronor to "
                                   + dollars + " dollars");
          return dollars;
     }
     public float convertCurrencyToDollars(float amount, String unitOfCurrency) {
          if ("krona".equalsIgnoreCase(unitOfCurrency)) {
               return convertKronorToDollars(amount);
          } else {
               return convertEurosToDollars(amount);
          }
     }
}
```

```
public class Demo {
    public static void main(String[] args) {
        Mediator mediator = new Mediator();
        Buyer swedishBuyer = new SwedishBuyer(mediator);
        Buyer frenchBuyer = new FrenchBuyer(mediator);
        float sellingPriceInDollars = 10.0f;
        AmericanSeller americanSeller = new AmericanSeller(mediator,
                                                 sellingPriceInDollars);
        DollarConverter dollarConverter = new DollarConverter(mediator);
        float swedishBidInKronor = 55.0f;
        while (!swedishBuyer.attemptToPurchase(swedishBidInKronor)) {
            swedishBidInKronor += 15.0f;
        }
        float frenchBidInEuros = 3.0f;
        while (!frenchBuyer.attemptToPurchase(frenchBidInEuros)) {
            frenchBidInEuros += 1.5f;
        }
    }
}
```

```
Buyer attempting a bid of 55.0 krona
Converting 55.0 kronor to 6.875 dollars
Seller rejects the bid of 6.875 dollars

Buyer attempting a bid of 70.0 krona
Converting 70.0 kronor to 8.75 dollars
Seller rejects the bid of 8.75 dollars

Buyer attempting a bid of 85.0 krona
Converting 85.0 kronor to 10.625 dollars
Seller accepts the bid of 10.625 dollars

Buyer attempting a bid of 3.0 euro
Converting 3.0 euros to 4.285714 dollars
Seller rejects the bid of 4.285714 dollars

Buyer attempting a bid of 4.5 euro
Converting 4.5 euros to 6.4285717 dollars
Seller rejects the bid of 6.4285717 dollars
```

Memento

Provee de un mecanismo por el que un objeto puede entregar al exterior una fotografía de su estado actual para poder recuperarlo después.

Sin violar la encapsulación

```java
// originator - object whose state we want to save
public class DietInfo {
	String personName;
	int dayNumber;
	int weight;
	public DietInfo(String personName, int dayNumber, int weight) {
		this.personName = personName;
		this.dayNumber = dayNumber;
		this.weight = weight;
	}
	public String toString() {
		return "Name: " + personName + ", day number: "
					+ dayNumber + ", weight: " + weight;
	}
	public void setDayNumberAndWeight(int dayNumber, int weight) {
		this.dayNumber = dayNumber;
		this.weight = weight;
	}
	public Memento save() {
		return new Memento(personName, dayNumber, weight);
	}
	public void restore(Object objMemento) {
		Memento memento = (Memento) objMemento;
		personName = memento.mementoPersonName;
		dayNumber = memento.mementoDayNumber;
		weight = memento.mementoWeight;
	}
}
```

```java
// memento - object that stores the saved state of the originator
private class Memento {
    String mementoPersonName;
    int mementoDayNumber;
    int mementoWeight;
    public Memento(String personName,  int dayNumber,  int weight) {
        mementoPersonName = personName;
        mementoDayNumber = dayNumber;
        mementoWeight = weight;
    }
}
```

```
// caretaker - saves and restores a DietInfo object's state via a memento
// note that DietInfo.Memento isn't visible to the caretaker so we need to cast
the memento to Object
public class DietInfoCaretaker {

    Object objMemento;

    public void saveState(DietInfo dietInfo) {
        objMemento = dietInfo.save();
    }

    public void restoreState(DietInfo dietInfo) {
        dietInfo.restore(objMemento);
    }

}
```

```java
public class MementoDemo {

    public static void main(String[] args) {

        // caretaker
        DietInfoCaretaker dietInfoCaretaker = new DietInfoCaretaker();

        // originator
        DietInfo dietInfo = new DietInfo("Fred", 1, 100);
        System.out.println(dietInfo);
        dietInfo.setDayNumberAndWeight(2, 99);
        System.out.println(dietInfo);
        System.out.println("Saving state.");

        dietInfoCaretaker.saveState(dietInfo);
        dietInfo.setDayNumberAndWeight(3, 98);
        System.out.println(dietInfo);
        dietInfo.setDayNumberAndWeight(4, 97);
        System.out.println(dietInfo);
        System.out.println("Restoring saved state.");
        dietInfoCaretaker.restoreState(dietInfo);
        System.out.println(dietInfo);
    }
}
```

```
jose@gwen:~/Programacion/DesPatt/Behavioral/Memento$ java MementoDemo
Name: Fred, day number: 1, weight: 100
Name: Fred, day number: 2, weight: 99
Saving state.
Name: Fred, day number: 3, weight: 98
Name: Fred, day number: 4, weight: 97
Restoring saved state.
Name: Fred, day number: 2, weight: 99
```

Observer

Es un mecanismo por el que es posible que muchos objetos (observadores) puedan percibir los cambios de estado de otro (el sujeto o *subject*)

```java
public interface WeatherSubject {
    public void addObserver(WeatherObserver
                              weatherObserver);
    public void removeObserver(WeatherObserver
                                 weatherObserver);
    public void doNotify();
}

public interface WeatherObserver {
    public void doUpdate(int temperature);
}
```

```
import java.util.HashSet;
import java.util.Iterator;
import java.util.Set;
public class WeatherStation implements WeatherSubject {
     Set<WeatherObserver> weatherObservers;
     int temperature;
     public WeatherStation(int temperature) {
          weatherObservers = new HashSet<WeatherObserver>();
          this.temperature = temperature;
     }
     @Override
     public void addObserver(WeatherObserver weatherObserver) {
          weatherObservers.add(weatherObserver);
     }
     @Override
     public void removeObserver(WeatherObserver weatherObserver) {
          weatherObservers.remove(weatherObserver);
     }
```

```
    @Override
    public void doNotify() {
        Iterator<WeatherObserver> it = weatherObservers.iterator();
        while (it.hasNext()) {
            WeatherObserver weatherObserver = it.next();
            weatherObserver.doUpdate(temperature);
        }
    }
    public void setTemperature(int newTemperature) {
        System.out.println("\nWeather station setting temperature to " + newTemperature);
        temperature = newTemperature;
        doNotify();
    }
}
```

```java
public class WeatherCustomer1 implements WeatherObserver {
    @Override
    public void doUpdate(int temperature) {
        System.out.println(
            "Weather customer 1 just found out the temperature is:"
            + temperature);
    }

}

public class WeatherCustomer2 implements WeatherObserver {
    @Override
    public void doUpdate(int temperature) {
        System.out.println(
            "Weather customer 2 just found out the temperature is:"
            + temperature);
    }

}
```

```java
public class Demo {

    public static void main(String[] args) {

        WeatherStation weatherStation =  new WeatherStation(33);

        WeatherCustomer1 wc1 = new WeatherCustomer1();
        WeatherCustomer2 wc2 = new WeatherCustomer2();
        weatherStation.addObserver(wc1);
        weatherStation.addObserver(wc2);

        weatherStation.setTemperature(34);

        weatherStation.removeObserver(wc1);

        weatherStation.setTemperature(35);
    }

}
```

```
jose@gwen:~/Programacion/DesPatt/Behavioral/Observer$ java Demo

Weather station setting temperature to 34
Weather customer 1 just found out the temperature is:34
Weather customer 2 just found out the temperature is:34

Weather station setting temperature to 35
Weather customer 2 just found out the temperature is:35
```

State

Hace posible que un objeto de cierta clase cambie su comportamiento cuando alcanza cierto estado.

Como si cambiara de clase.

```java
// State
public interface EmotionalState {

    public String sayHello();

    public String sayGoodbye();

}
```

```java
// Concrete State
public class HappyState implements EmotionalState {
    @Override
    public String sayGoodbye() {
        return "Bye, friend!";
    }
    @Override
    public String sayHello() {
        return "Hello, friend!";
    }
}

//Concrete State
public class SadState implements EmotionalState {
    @Override
    public String sayGoodbye() {
        return "Bye. Sniff, sniff.";
    }
    @Override
    public String sayHello() {
        return "Hello. Sniff, sniff.";
    }
}
```

```java
// Context
public class Person implements EmotionalState {

    EmotionalState emotionalState;

    public Person(EmotionalState emotionalState) {
        this.emotionalState = emotionalState;
    }

    public void setEmotionalState(EmotionalState emotionalState) {
        this.emotionalState = emotionalState;
    }

    @Override
    public String sayGoodbye() {
        return emotionalState.sayGoodbye();
    }

    @Override
    public String sayHello() {
        return emotionalState.sayHello();
    }

}
```

```
public class Demo {

    public static void main(String[] args) {

        Person person = new Person(new HappyState());
        System.out.println("Hello in happy state: "
                           + person.sayHello());
        System.out.println("Goodbye in happy state: "
                           + person.sayGoodbye());

        person.setEmotionalState(new SadState());
        System.out.println("Hello in sad state: "
                           + person.sayHello());
        System.out.println("Goodbye in sad state: "
                           + person.sayGoodbye());

    }

}
```

```
java jose@gwen:~/Programacion/DesPatt/Behavioral/State$ java
Demo
Hello in happy state: Hello, friend!
Goodbye in happy state: Bye, friend!
Hello in sad state: Hello. Sniff, sniff.
Goodbye in sad state: Bye. Sniff, sniff.
```

Strategy

Define una familia de algoritmos que pueden intercambiarse. El algoritmo cambia con el cliente que lo usa.

```java
public interface Strategy {
    boolean checkTemperature(int temperatureInF);
}
```

```java
public class HikeStrategy implements Strategy {
    @Override
    public boolean checkTemperature(int temperatureInF) {
        if ((temperatureInF >= 50) && (temperatureInF <= 90)) {
            return true;
        } else {
            return false;
        }
    }
}

public class SkiStrategy implements Strategy {
    @Override
    public boolean checkTemperature(int temperatureInF) {
        if (temperatureInF <= 32) {
            return true;
        } else {
            return false;
        }
    }
}
```

```
public class Context {

     int temperatureInF;
     Strategy strategy;

     public Context(int temperatureInF, Strategy strategy) {
          this.temperatureInF = temperatureInF;
          this.strategy = strategy;
     }

     public void setStrategy(Strategy strategy) {
          this.strategy = strategy;
     }

     public int getTemperatureInF() {
          return temperatureInF;
     }

     public boolean getResult() {
          return strategy.checkTemperature(temperatureInF);
     }

}
```

```java
public class Demo {

     public static void main(String[] args) {

          int temperatureInF = 60;

          Strategy skiStrategy = new SkiStrategy();
          Context context = new Context(temperatureInF, skiStrategy);

          System.out.println("Is the temperature ("
                                  + context.getTemperatureInF()
                                  + "F) good for skiing? " + context.getResult());

          Strategy hikeStrategy = new HikeStrategy();
          context.setStrategy(hikeStrategy);

          System.out.println("Is the temperature ("
                                   + context.getTemperatureInF()
                                   + "F) good for hiking? " + context.getResult());

     }

}
```

```
jose@gwen:~/Programacion/DesPatt/Behavioral/Strategy$ java Demo
Is the temperature (60F) good for skiing? false
Is the temperature (60F) good for hiking? true
```

Template method

Define el esqueleto de la operación general de un algoritmo, las subclases lo particularizan.

```java
public abstract class Meal {

    // template method
    public final void doMeal() {
        prepareIngredients();
        cook();
        eat();
        cleanUp();
    }

    public abstract void prepareIngredients();

    public abstract void cook();

    public void eat() {
        System.out.println("Mmm, that's good");
    }

    public abstract void cleanUp();

}
```

```java
public class HamburgerMeal extends Meal {

    @Override
    public void prepareIngredients() {
        System.out.println("Getting burgers, buns, and french fries");
    }

    @Override
    public void cook() {
        System.out.println("Cooking burgers on grill and fries in oven");
    }

    @Override
    public void cleanUp() {
        System.out.println("Throwing away paper plates");
    }
}
```

```java
public class TacoMeal extends Meal {

    @Override
    public void prepareIngredients() {
        System.out.println("Getting ground beef and shells");
    }

    @Override
    public void cook() {
        System.out.println("Cooking ground beef in pan");
    }

    @Override
    public void eat() {
        System.out.println("The tacos are tasty");
    }

    @Override
    public void cleanUp() {
        System.out.println("Doing the dishes");
    }

}
```

```java
public class Demo {

    public static void main(String[] args) {

        Meal meal1 = new HamburgerMeal();
        meal1.doMeal();

        System.out.println();

        Meal meal2 = new TacoMeal();
        meal2.doMeal();

    }

}
```

```
jose@gwen:~/Programacion/DesPatt/Behavioral/TemplateMethod$ java Demo
Getting burgers, buns, and french fries
Cooking burgers on grill and fries in oven
Mmm, that's good
Throwing away paper plates

Getting ground beef and shells
Cooking ground beef in pan
The tacos are tasty
Doing the dishes
```