

# Patrones de diseño

NOTA: Para todos los problemas planteados en los ejercicios, proporciona un diseño UML del patrón que consideres adecuado (como el que se muestra en el tema de patrones de diseño) y completa el código proporcionado. Indica el patrón utilizado en cada caso.

## 1 Ejercicios de patrones de creación

### 1.1 Patrón Singleton

Se está monitorizando un sistema de ventas de vehículos y se quiere representar al vendedor con una clase que registre su información en vez de utilizar variables globales que contengan su nombre, dirección, email, etc. Esta clase debe poseer métodos para visualizar toda su información, establecer el nombre en su atributo nombre, obtener el nombre, y así con el resto de información (al menos, dirección e email).

Utiliza la clase TestComercial para crear la clase Comercial y visualizar su información.

Código a completar:

```
public class Comercial
{
    tipo String nombre;
    tipo String direccion;
    tipo String email;
```

código a completar

```
public void visualiza()
{
```

```

        System.out.println("Nombre: " + nombre);
        System.out.println("Dirección: " + dirección);
        System.out.println("Email: " + email);
    }

    public String getNombre()
    {
        return nombre;
    }

    public void setNombre(String nombre)
    {
        this.nombre = nombre;
    }

    public String getDirección()
    {
        return dirección;
    }

    public void setDirección(String dirección)
    {
        this.dirección = dirección;
    }

    public String getEmail()
    {
        return email;
    }

    public void setEmail(String email)
    {
        this.email = email;
    }
}

public class TestComercial {

    public static void main(String[] args)
    {
        código a completar
        elComercial.setNombre("Comercial Auto");
        elComercial.setDirección("Madrid");
        elComercial.setEmail("comercial@comerciales.com");
        // muestra el comercial del sistema
        visualiza();
    }

    public static void visualiza()
    {

```

```

        Comercial elComercial = Comercial.Instance();
        elComercial.visualiza();
    }

}

```

## 1.2 Patrón Abstract Factory

En el mismo sistema de ventas de vehículos se gestionan los que funcionan con gasolina y los eléctricos. Y esta gestión se delega en el objeto Catalogo encargado de crear esos objetos. Para cada producto, automóvil o scooter, se dispone de una clase abstracta y, además, de una subclase concreta para cada una de las anteriores que deriva una versión de producto que funciona con gasolina y otra subclase concreta que deriva una versión del producto que funciona con electricidad. Se prevé que evolucione el sistema y surjan nuevas clases de familias de vehículos (diésel o gasolina-eléctrico, etc.). Por lo tanto, no se desea que estas variaciones provoquen muchas y pesadas modificaciones en el objeto Catalogo.

Código a completar:

```

public abstract class Automovil
{
    protected String modelo;
    protected String color;
    protected int potencia;
    protected double espacio;

    public Automovil(String modelo, String color, int
        potencia, double espacio)
    {
        this.modelo = modelo;
        this.color = color;
        this.potencia = potencia;
        this.espacio = espacio;
    }

    public abstract void mostrarCaracteristicas();
}

public class AutomovilElectricidad extends Automovil
{
    public AutomovilElectricidad(String modelo, String
        color, int potencia, double espacio)
    {
        super(modelo, color, potencia, espacio);
    }

    public void mostrarCaracteristicas()
    {

```

```
        System.out.println(
            "Automovil electrico de modelo: " + modelo +
            " de color: " + color + " de potencia: " +
            potencia + " de espacio: " + espacio);
    }
}

public class AutomovilGasolina extends Automovil
{
    public AutomovilGasolina(String modelo, String
        color, int potencia, double espacio)
    {
        super(modelo, color, potencia, espacio);
    }

    public void mostrarCaracteristicas()
    {
        System.out.println(
            "Automovil de gasolina de modelo: " + modelo +
            " de color: " + color + " de potencia: " +
            potencia + " de espacio: " + espacio);
    }
}

public abstract class Scooter
{
    protected String modelo;
    protected String color;
    protected int potencia;

    public Scooter(String modelo, String color, int
        potencia)
    {
        this.modelo = modelo;
        this.color = color;
        this.potencia = potencia;
    }
    public abstract void mostrarCaracteristicas();
}

public class ScooterElectricidad extends Scooter
{
    public ScooterElectricidad(String modelo, String color,
        int potencia)
    {
        super(modelo, color, potencia);
    }

    public void mostrarCaracteristicas()
    {
        System.out.println("Scooter electrica de modelo: " +
            modelo + " de color: " + color +
```

```
        " de potencia: " + potencia);
    }

}

public class ScooterGasolina extends Scooter
{
    public ScooterGasolina(String modelo, String color,
                           int potencia)
    {
        super(modelo, color, potencia);
    }

    public void mostrarCaracteristicas()
    {
        System.out.println("Scooter de gasolina de modelo: " +
                           modelo + " de color: " + color +
                           " de potencia: " + potencia);
    }

}

public interface incluye nombre
{
    Automovil creaAutomovil(String modelo, String color,
                            int potencia, double espacio);

    Scooter creaScooter(String modelo, String color, int
                        potencia);
}
```

incluye código

```

import java.util.*;
public class Catalogo
{
    public static int nAutos = 3;
    public static int nScooters = 2;

    public static void main(String[] args)
    {
        Scanner reader = new Scanner(System.in);
        FabricaVehiculo fabrica;
        Automovil[] autos = new Automovil[nAutos];
        Scooter[] scooters = new Scooter[nScooters];
        System.out.print("Desea utilizar " +
            "vehiculos electricos (1) o a gasolina (2):");
        String eleccion = reader.next();
        if (eleccion.equals("1"))
        {
            [ ] incluye código
        }
        else
        {
            [ ] incluye código
        }

        for (int index = 0; index < nAutos; index++)
            completa esta línea].creaAutomovil("estandar",
                "amarillo", 6+index, 3.2);
        for (int index = 0; index < nScooters; index++)
            completa esta línea].creaScooter("clasico",
                "rojo", 2+index);
        for (Automovil auto: autos)
            auto.mostrarCaracteristicas();
        for (Scooter scooter: scooters)
            scooter.mostrarCaracteristicas();
    }
}

```

### 1.3 Patrón Factory Method

En el mismo sistema de ventas de vehículos, la clase Cliente implementa el método creaPedido que debe crear el pedido del cliente. Sin embargo, hay ciertos clientes que solicitan un vehículo pagando al contado y otros utilizan un crédito. Así, en función de lo que prefiera el Cliente (ClienteContado o ClienteCredito), el método creaPedido debe crear una instancia de la clase PedidoContado o una instancia de la clase PedidoCredito. Por simplificar, se considerará que cuando la validación de un pedido al contenido es continua o sistemática, se podrá escoger aceptar únicamente los pedidos provistos de un crédito cuyo valor esté entre 1.000 y 5.000 euros.

Código a completar:

```
public abstract class Pedido
{
    protected double importe;

    public Pedido(double importe)
    {
        this.importe = importe;
    }

    public abstract boolean valida();

    public abstract void paga();
}

public class PedidoContado extends Pedido
{
    public PedidoContado(double importe)
    {
        super(importe);
    }

    public void paga()
    {
        System.out.println(
            "El pago del pedido por importe de: " +
            importe + " se ha realizado.");
    }

    public boolean valida()
    {
        return true;
    }
}

public class PedidoCredito extends Pedido
{
    public PedidoCredito(double importe)
    {
        super(importe);
    }
}
```

```

}

public void paga()
{
    System.out.println(
        "El pago del pedido a credito de: " +
        importe + " se ha realizado.");
}

public boolean valida()
{
    return (importe >= 1000.0) && (importe <= 5000.0);
}
}

import java.util.*;
public abstract class Cliente
{
    protected List<Pedido> pedidos =
        new ArrayList<Pedido>();

completa aquí la declaración creaPedido(double importe);

public void nuevoPedido(double importe)
{
    Pedido pedido = this.creaPedido(importe);
    if (pedido.valida())
    {
        pedido.paga();
        pedidos.add(pedido);
    }
}
}

public class ClienteContado extends Cliente
{
    protected Pedido creaPedido(double importe)
    {
        return new [complete linea];
    }
}

public class ClienteCredito extends Cliente
{
    protected Pedido creaPedido(double importe)
    {
        return new [complete linea];
    }
}

```

```

public class Usuario
{
    public static void main(String[] args)
    {
        // crean un Cliente que paga al contado con dos pedidos de 2000 y
        // 10000, respectivamente, y otro cliente que utiliza crédito con dos
        // pedidos: de 2000 y 10000 también //
        Cliente cliente;
        completa código
    }
}

```

## 1.4 Patrón Prototype

En el mismo sistema de ventas de vehículos, un cliente debe recibir una documentación compuesta de un número concreto de documentos (certificado de cesión, solicitud de matriculación y orden de pedido). Existen otros tipos de documentos que pueden incluirse o excluirse a esta documentación en función de los requerimientos de gestión o de cambios en la reglamentación asociada.

Dado que este ejemplo es más complejo y contiene el uso de varios patrones, se da ya detalle de la estructura final del patrón a usar; no sólo el problema en sí para averiguar qué patrón es el más adecuado.

Hay una clase Documentación cuyas instancias son documentaciones compuestas por diversos documentos obligatorios. Para cada tipo de documento se incluye su clase correspondiente. Se quiere crear una instancia de Documentación que contenga los distintos documentos necesarios “en blanco”. Así se define a nivel de instancias, y no de clases, el contenido preciso de la documentación que debe recibir un cliente. Incluir o excluir un documento en la documentación en blanco no supondrá ninguna modificación en su clase. Cada nueva documentación se creará duplicando todos los documentos de la documentación en blanco.

La clase Documento es una clase abstracta conocida por la clase Documentación. Sus subclases corresponden a los distintos tipos de documentos e incluye un método duplique que permite clonar una instancia existente para obtener una nueva.

DocumentacionEnBlanco posee **una única instancia** que contiene todos los documentos necesarios (documentos en blanco). Esta instancia se manipula mediante los métodos incluye y excluye (documentos).

DocumentacionCliente crea su conjunto de documentos solicitando a la única instancia de la clase DocumentacionEnBlanco la lista de documentos en blanco y agregándolos uno a uno tras haberlos clonado.

Código a completar:

```
public abstract class Documento2
implements Cloneable
{
    protected String contenido = new String();

    public Documento2 duplica()
    {
        Documento2 resultado;

        try
        {
            resultado = (Documento2)this.clone();
        }
        catch (CloneNotSupportedException exception)
        {
            return null;
        }
        return resultado;
    }

    public void rellena(String informacion)
    {
        contenido = informacion;
    }

    public abstract void imprime();
    public abstract void visualiza();
}

public class SolicitudMatriculacion extends Documento2
{
    public void visualiza()
    {
        System.out.println(
            "Muestra la solicitud de matriculacion: " + contenido);
    }

    public void imprime()
    {
        System.out.println(
            "Imprime la solicitud de matriculacion: " + contenido);
    }
}

public class CertificadoCesion extends Documento2
```

```

{
    public void visualiza()
    {
        System.out.println(
            "Muestra el certificado de cesion: " + contenido);
    }

    public void imprime()
    {
        System.out.println(
            "Imprime el certificado de cesion: " + contenido);
    }
}

import java.util.*;
public abstract class Documentacion
{
    protected List<Documento2> documentos;

    public List<Documento2> getDocumentos()
    {
        return completar linea
    }
}

import java.util.*;
public class DocumentacionEnBlanco extends Documentacion
{
    private static DocumentacionEnBlanco _instance = null;

    private DocumentacionEnBlanco()
    {
        documentos = new ArrayList<completa linea>();
    }

    public static DocumentacionEnBlanco Instance()
    {
        if completa código
        {
            return completar linea
        }
    }

    public void incluye(Documento2 doc)
    {
        completa linea.add(doc);
    }

    public void excluye(Documento2 doc)
    {
        completa linea.remove(doc);
    }
}

```

```

}

import java.util.*;
public class DocumentacionCliente extends completa linea
{
    public DocumentacionCliente(String informacion)
    {
        documentos = new ArrayList<completa linea>();
        DocumentacionEnBlanco documentacionEnBlanco =
            completa linea
        List<completa linea> documentosEnBlanco =
            documentacionEnBlanco.getDocumentos();
        for (Documento2 documento: documentosEnBlanco)
        {
            Documento2 copiaDocumento = documento.completa linea
            copiaDocumento.rellena(informacion);
            documentos.add(completa linea);
        }
    }

    public void visualiza()
    {
        for (Documento2 documento: documentos)
            documento.visualiza();
    }

    public void imprime()
    {
        for (Documento2 documento: documentos)
            documento.imprime();
    }
}

public class Usuario3
{
    public static void main(String[] args)
    {
        //inicializacion de la documentacion en blanco
        DocumentacionEnBlanco documentacionEnBlanco =
            DocumentacionEnBlanco.Instance();
        documentacionEnBlanco.incluye(new completa linea);
        documentacionEnBlanco.incluye(new completa linea);
        documentacionEnBlanco.incluye(new completa linea);
        // creacion de documentacion nueva para dos clientes
        completa código
    }
}

documentacionCliente1.visualiza();

```

```

        documentacionCliente2.visualiza();
    }
}

```

## 2 Ejercicios de patrones estructurales

### 2.1 Patrón Composite

Se está monitorizando un sistema de ventas de vehículos y se quiere representar las empresas cliente, en especial, para conocer el número de vehículos de los que disponen y así proporcionarles ofertas de mantenimiento para su parque de vehículos.

Las empresas con filiales solicitan ofertas de mantenimiento que tengan en cuenta el parque de vehículos de sus filiales.

Una forma inmediata consiste en procesar de forma distinta las empresas sin filiales y las que poseen filiales. Este procesado diferenciado para ambos tipos de empresas hace que la aplicación sea más compleja y dependa de la composición interna de las empresas clientes.

Suponiendo que no hay filiales comunes entre dos empresas y que una empresa puede tener filiales que posean, a su vez, otras filiales, utilice un patrón que unifique el tratamiento de ambos tipos de empresas. Considere las clases Empresa, EmpresaSinFilial y EmpresaMadre (empresa que consta de empresas filiales). Empresa tiene un método concreto (agregaVehiculo) que no depende de la composición en filiales de la empresa y dos métodos abstractos (agregaFilial) y calculaCosteMantenimiento, respectivamente.

Código a completar:

```

public abstract class Empresa
{
    protected static double costeUnitarioVehiculo = 5.0;
    protected int nVehiculos;

    public void agregaVehiculo()
    {
        nVehiculos = nVehiculos + 1;
    }

    public abstract double calculaCosteMantenimiento();

    public abstract boolean agregaFilial(Empresa filial);
}

public class EmpresaSinFilial extends Empresa
{
    public boolean agregaFilial(Empresa filial)

```

```

    {
        return completa linea
    }

    public double calculaCosteMantenimiento()
    {
        return complete linea
    }
}

import java.util.*;
public class EmpresaMadre extends Empresa
{
    protected List<Empresa> filiales =
        new ArrayList<Empresa>();

    public boolean agregaFilial(Empresa filial)
    {
        return completa linea
    }

    public double calculaCosteMantenimiento()
    {
        double coste = 0.0;
        for (Empresa filial: filiales)
            coste = coste + complete linea
        return coste + complete linea
    }
}

public class Usuario2
{
    public static void main(String[] args)
    {
        //el cliente crea una empresa madre que posee un vehículo y dos
        //filiales. La primera filial posee un vehículo, y la segunda posee dos
        //filiarias a su vez //

        Empresa empresa1 = new EmpresaSinFilial();
        empresa1.agregaVehiculo();

        Empresa empresa2 = new complete linea
        Complete código
        Empresa grupo = new EmpresaMadre();
        grupo.completa linea
        grupo.completa linea
        grupo.completa linea
        System.out.println(
            "Coste de mantenimiento total del grupo: " +
            grupo.calculaCosteMantenimiento());
    }
}

```

## 2.2 Patrón Adapter

Se está monitorizando un sistema de ventas de vehículos. El servidor web de este sistema crea y administra los documentos destinados a los clientes. La interfaz Documento se ha definido para realizar esta gestión, que posee tres métodos: setContenido, dibuja e imprime. Se ha realizado ya una primera clase de implementación de esta interfaz: la clase DocumentoHtml que implementa adecuadamente estos tres métodos. Los objetos clientes de esta interfaz y esta clase cliente ya se han diseñado.

En esta situación, para la agregación de documentos PDF, se da un problema porque son documentos más complejos de construir y gestionar que los documentos HTML. Para ello, se ha escogido un producto del mercado, ComponentePdf, aunque su interfaz no se corresponde con la interfaz Documento ya que incluye más métodos y su nomenclatura es muy distinta (posee prefijos pdf). ¿Cómo resolvería este problema con patrones?

Código a completar:

```
public interface Documento
{
    void setContenido(String contenido);
    void dibuja();
    void imprime();
}

public class DocumentoHtml implements Documento
{
    protected String contenido;

    public void setContenido(String contenido)
    {
        this.contenido = contenido;
    }

    public void dibuja()
    {
        System.out.println("Dibuja el documento HTML: " +
            contenido);
    }

    public void imprime()
    {
        System.out.println("Imprime el documento HTML: " +
            contenido);
    }
}
```

```

}

public class ComponentePdf
{
    protected String contenido;

    public void pdfFijaContenido(String contenido)
    {
        this.contenido = contenido;
    }

    public void pdfPreparaVisualizacion()
    {
        System.out.println("Visualiza PDF: Comienzo");
    }

    public void pdfRefresca()
    {
        System.out.println("Visualiza contenido PDF: " +
            contenido);
    }

    public void pdfFinalizaVisualizacion()
    {
        System.out.println("Visualiza PDF: Fin");
    }

    public void pdfEnviaImpresora()
    {
        System.out.println("Impresi n PDF: " + contenido);
    }
}

public class DocumentoPdf implements Documento
{
    protected ComponentePdf herramientaPdf = new completa linea

    public void setContenido(String contenido)
    {
        complete linea de  digo
    }

    public void dibuja()
    {
        complete  digo
    }
}

```

```

public void imprime()
{
    complete línea de código
}
}

public class ServidorWeb
{
    public static void main(String[] args)
    {
        Documento documento1, documento2;
        documento1 = new DocumentoHtml();
        documento1.setContenido("Hello");
        documento1.dibuja();
        System.out.println();
        documento2 = new completa línea
        documento2.setContenido("Hola");
        documento2.dibuja();
    }
}

```

### 2.3 Patrón Façade

Se está monitorizando un sistema de ventas de vehículos. El servidor web de este sistema crea y administra los documentos destinados a los clientes. La interfaz Documento se ha definido para esta gestión con tres métodos: setContenido, dibuja e imprime. Se ha realizado una primera clase de implementación de esta interfaz: la clase DocumentoHtml que implementa estos tres métodos. Los objetos clientes de esta interfaz y esta clase cliente ya se han diseñado.

En esta situación, la agregación de documentos PDF es conflictiva al ser documentos más complejos de construir y de administrar que los documentos HTML. Se ha escogido un producto del mercado para ello, pero su interfaz no se corresponde con la de Documento. Se trata del componente ComponentePDF cuya interfaz incluye más métodos y la nomenclatura es completamente diferente (con prefijos pdf). ¿Cómo solucionaría este problema? señoy se quiere proporcionar la posibilidad de acceder al sistema de venta de vehículos como servicio web. La arquitectura del sistema se ha estructurado bajo la forma de un conjunto de componentes que poseen su propia interfaz (los componente Catalogo, GestionDocumento, RecogidaVehiculo, etc). Se desea ahora dar acceso al conjunto de la interfaz de estos componentes a los clientes del servicio web teniendo en cuenta dos inconvenientes fundamentales:

- Ciertas funcionalidades no se utilizan por los clientes del servicio web (por ejemplo, las funcionalidades de visualización del catálogo, etc.).
- La arquitectura interna del sistema responde a las exigencias de modularidad y evolución que no forman parte de las necesidades de los clientes del servicio web, para los que estos requerimientos suponen una complejidad inútil.

Teniendo en cuenta estos supuestos, qué patrón aplicaría de tal forma que proporcione una interfaz única más sencilla y con un nivel de abstracción más elevado. Utilizad una clase denominada WebServiceAuto que debe estar constituida por el método buscaVehiculos(precioMedio, desviacionMax) cuya definición consiste en invocar al método buscaVehiculos(precioMin, precioMax) del catálogo, adaptando el valor de los argumentos de este método en función del precio medio y la desviación máxima.

Código a completar:

```
// componentes del sistema

import java.util.*;
public interface Catalogo2
{
    List<String> buscaVehiculos(int precioMin, int
        precioMax);
}

import java.util.*;
public class ComponenteCatalogo implements Catalogo2
{
    protected Object[] descripcionVehiculo =
    {
        "Berlina 5 puertas", 6000, "Compacto 3 puertas", 4000,
        "Espace 5 puertas", 8000, "Break 5 puertas", 7000,
        "CoupÈ 2 puertas", 9000, "Utilitario 3 puertas", 5000
    };

    public List<String> buscaVehiculos(int precioMin,
        int precioMax)
    {
        int indice, tamaño;
        List<String> resultado = new ArrayList<String>();
        tamaño = descripcionVehiculo.length / 2;
        for (indice = 0; indice < tamaño; indice++)
        {
            int precio = (Integer)descripcionVehiculo[2 * indice + 1];
            if ((precio >= precioMin) && (precio <= precioMax))
                resultado.add((String)descripcionVehiculo[2 *
                    indice]);
        }
        return resultado;
    }
}

public interface GestionDocumento
{
    String documento(int indice);
}
```

```

public class ComponenteGestionDocumento implements GestionDocumento
{
    public String documento(int indice)
    {
        return "Documento número " + indice;
    }
}

import java.util.List;
completa línea WebServiceAuto
{
    String documento(int indice);
    List<String> buscaVehiculos(int precioMedio, int
        desviacionMax);
}

import java.util.List;
public class WebServiceAutoImpl implements WebServiceAuto
{
    completa línea catalogo = new ComponenteCatalogo();
    completa línea gestionDocumento = new
        ComponenteGestionDocumento();

    public String documento(int indice)
    {
        return gestionDocumento.documento(completa línea);
    }

    public List<String> buscaVehiculos(int precioMedio,
        int desviacionMax)
    {
        return catalogo.buscaVehiculos(completa línea);
    }
}

```

### 3 Ejercicios de patrones de comportamiento

#### 3.1 Patrón Observer

Se está monitorizando un sistema de ventas de vehículos. Se desea que, cada vez que la información relativa a un vehículo se modifica, se actualice su visualización. Puede, además, existir varias visualizaciones simultáneas.

Se van a definir cuatro clases:

- **Sujeto**: clase abstracta que incluye todo objeto y que se encarga de notificar al resto de objetos las modificaciones del estado interno de ese objeto.
- **Vehículo**: subclase concreta de **Sujeto** que describe los vehículos. Gestiona dos atributos: descripción y precio.
- **Observador**: interfaz que un objeto puede necesitar para recibir las notificaciones del cambio de estado de los objetos a los que se ha inscrito previamente.
- **VistaVehículo**: subclase concreta correspondiente a la implementación de **Observador** cuyas instancias muestran la información de un vehículo.

Código a completar:

```

import java.util.*;
completa linea Sujeto
{
    protected List<Observador> observadores =
        new ArrayList<Observador>();

    public void agrega(Observador observador)
    {
        observadores.add(completa linea);
    }

    public void suprime(Observador observador)
    {
        observadores.remove(completa linea);
    }

    public void notifica()
    {
        for (Observador observador: observadores)
            observador.completa linea;
    }
}

public interface Observador
{
    void actualiza();
}

public class Vehiculo extends Completa definición
{
    protected String descripcion;
    protected Double precio;

    public String getDescripcion()
    {
        return descripcion;
    }

    public void setDescripcion(String descripcion)
}

```

```
{  
    this.descripcion = descripcion;  
    this.completa linea;  
}  
  
public Double getPrecio()  
{  
    return precio;  
}  
  
public void setPrecio(Double precio)  
{  
    this.precio = precio;  
    this.completa linea;  
}  
  
}  
  
public class VistaVehiculo implements Observador  
{  
    protected Vehiculo vehiculo;  
    protected String texto = "";  
  
    public VistaVehiculo(Vehiculo vehiculo)  
{  
        this.vehiculo = vehiculo;  
        vehiculo.agrega(this);  
        actualizaTexto();  
    }  
  
    protected void actualizaTexto()  
{  
        texto = "Descripcion " + vehiculo.descripcion +  
            " Precio: " + vehiculo.precio;  
    }  
  
    public void actualiza()  
{  
        Completa linea de código  
        this.redibuja();  
    }  
  
    public void redibuja()  
{  
        System.out.println(texto);  
    }  
}  
  
public class Usuario4  
{
```

```

public static void main(String[] args)
{
    Vehiculo vehiculo = new Vehiculo();
    vehiculo.setDescripcion("Vehiculo economico");
    vehiculo.setPrecio(5000.0);
    VistaVehiculo vistaVehiculo = new VistaVehiculo(vehiculo);
    vistaVehiculo.redibuja();
    vehiculo.setPrecio(4500.0);
    VistaVehiculo vistaVehiculo2 = new VistaVehiculo(vehiculo);
    vehiculo.setPrecio(5500.0);
    //¿hace falta más para refrescar las vistas creadas?
}

```

### 3.2 Patrón Strategy

Se está monitorizando un sistema de ventas de vehículos. La clase VistaCatalogo dibuja la lista de vehículos que van destinados a la venta. Para ello, se usa un algoritmo de diseño gráfico que se encarga de calcular la representación gráfica en función del navegador. Existen así dos versiones del algoritmo:

- Versión 1. Sólo muestra un vehículo por línea (el vehículo ocupa todo el ancho disponible). Muestra toda la información posible así como cuatro fotos.
- Versión 2. Muestra tres vehículos por línea. Muestra menos información y una única foto.

La interfaz de la clase VistaCatalogo no depende de la elección del algoritmo de representación gráfica. Hay varias soluciones: 1) transformar la clase VistaCatalogo en una interfaz o en una clase abstracta e incluir dos subclases de implementación distintas según la selección del algoritmo, 2) implementar ambos algoritmos en la clase VistaCatalogo y apoyarse en instrucciones condicionales para realizar la elección del algoritmo. Piense los inconvenientes de estas soluciones y por qué otra solución más adecuada puede ser incluir una clase por algoritmo (DibujaUnVehiculoPorLinea y DibujaTresVehiculosPorLinea) con una interfaz común (DibuajaCatalogo) que se utilice para dialogar con la clase VistaCatalogo.

Código a completar:

```

import java.util.*;
public interface DibujaCatalogo
{
    void dibuja(List<VistaVehiculo> contenido);
}

import java.util.*;
public class DibujaUnVehiculoPorLinea implements completa
definición
{

```

```

public void dibuja(List<VistaVehiculo1> contenido)
{
    System.out.println(
        "Dibuja los vehiculos mostrando un vehiculo por linea");
    for (VistaVehiculo1 vistaVehiculo: contenido)
    {
        vistaVehiculo.completa linea;
        System.out.println();
    }
    System.out.println();
}

import java.util.*;
public class DibujaTresVehiculosPorLinea implements completa
definición
{
    public void dibuja(List<VistaVehiculo1> contenido)
    {
        int contador;
        System.out.println(
            "Dibuja los vehiculos mostrando tres vehiculos por linea");
        contador = 0;
        for (VistaVehiculo1 vistaVehiculo: contenido)
        {
            vistaVehiculo.completa linea;
            contador++;
            if (contador == 3)
            {
                System.out.println();
                contador = 0;
            }
            else
                System.out.print(" ");
        }
        if (contador != 0)
            System.out.println();
        System.out.println();
    }
}

public class VistaVehiculo1
{
    protected String descripcion;

    public VistaVehiculo1(String descripcion)
    {
        this.descripcion = completa linea;
    }
}

```

```

    }

    public void dibuja()
    {
        System.out.print(descripcion);
    }
}

import java.util.*;
public class VistaCatalogo
{
    protected List<VistaVehiculo1> contenido =
        new ArrayList<VistaVehiculo1>();
    protected DibujaCatalogo dibujo;

    public VistaCatalogo(DibujaCatalogo dibujo)
    {
        contenido.add(new completa linea("vehiculo economico"));
        contenido.add(new completa linea("vehiculo especial"));
        contenido.add(new completa linea("vehiculo rapido"));
        contenido.add(new completa linea("vehiculo confortable"));
        contenido.add(new completa linea("vehiculo deportivo"));
        this.dibujo = completa linea;
    }

    public void dibuja()
    {
        dibujo.dibuja(contenido);
    }
}

public class Usuario5
{
    public static void main(String[] args)
    {
        // crear una representación gráfica de cada tipo
        VistaCatalogo vistaCatalogo1 = new VistaCatalogo(new
            completa linea);
        vistaCatalogo1.dibuja();
        VistaCatalogo vistaCatalogo2 = new VistaCatalogo(new
            completa linea);
        vistaCatalogo2.dibuja();
    }
}

```

### 3.3 Patrón Template Method

Se está monitorizando un sistema de ventas online de vehículos. Se quiere gestionar pedidos de clientes de España y de Bélgica. La diferencia entre ambas peticiones se refiere esencialmente al cálculo del IVA. En España, la tasa de IVA es fija, del 21%, y en el caso de Bélgica, es variable, del 12% para servicios, y del 15% para material –por simplicidad, consideraremos una tasa única del 15% sólo–). El cálculo del IVA requiere dos operaciones de cálculo distintas en función del país.

Una posible solución consistiría en implementar dos clases distintas sin superclase común; PedidoEspaña y PedidoBelgica. El inconveniente es que la solución tiene código idéntico que no ha sido factorizado, como por ejemplo, la visualización de la información del pedido (método visualiza). Por lo tanto, podría incluirse una clase abstracta Pedido para factorizar los métodos comunes, como el anterior. Yendo más lejos, el método Template Method (denominado también método modelo) permite factorizar el código común en el interior de los métodos (incluyendo una parte común de un cierto algoritmo más o menos sencillo que se complementa con partes específicas). El código específico se puede desplazar a un método aparte, cuya implementación, en este caso concreto, es específica de cada país. El método en sí se puede incluir en la clase Pedido como método abstracto.

Código a completar:

```
public abstract class Pedido1
{
    protected double importeSinIVA;
    protected double importeIVA;
    protected double importeConIVA;

    protected abstract void calculaIVA();

    public void calculaPrecioConIVA()
    {
        this.completa línea;
        importeConIVA = importeSinIVA + importeIVA;
    }

    public void setImporteSinIVA(double importeSinIVA)
    {
        this.importeSinIVA = importeSinIVA;
    }

    public void visualiza()
    {
        System.out.println("Pedido");
        System.out.println("Importe sin IVA " + importeSinIVA);
        System.out.println("Importe con IVA " + importeConIVA);
    }
}

public class PedidoEspanna extends completa definición
{
    protected void calculaIVA()
```

```
{  
    importeIVA = importeSinIVA * 0.21;  
}  
}  
  
public class PedidoBelgica extends completa definición  
{  
    protected void calculaIVA()  
    {  
        importeIVA = importeSinIVA * 0.15;  
    }  
}  
  
public class Usuario6  
{  
    public static void main(String[] args)  
    {  
        // crear un pedido de cada uno de los dos países considerados  
  
        Pedido1 pedidoEspanna = new completa línea;  
        pedidoEspanna.completa línea(10000);  
        pedidoEspanna.completa línea();  
        pedidoEspanna.completa línea();  
  
        Pedido1 pedidoBelgica = new completa línea;  
        pedidoBelgica.completa línea(10000);  
        pedidoBelgica.completa línea();  
        pedidoBelgica.completa línea();  
    }  
}
```