Programación con Java y Eclipse

Patrones de software básicos

Indice

1. Introducción a los patrones de diseño de Software

Un patrón de diseño es una solución a un problema de diseño no trivial que es efectiva (ya se resolvió el problema satisfactoriamente en ocasiones anteriores) y reusable (se puede aplicar a diferentes problemas de diseño en distintas circunstancias). El objetivo de los patrones es agrupar una colección de soluciones de diseño que son válidas en distintos contextos y que han sido aplicadas con éxito en otras ocasiones.

Los patrones son soluciones de sentido común que deberían formar parte del conocimiento de un diseñador experto. Las ventajas del uso de patrones son:

- facilitan la comunicación entre diseñadores, pues establecen un marco de referencia (terminología, justificación).
- facilitan el aprendizaje al programador inexperto, pudiendo establecer parejas problema-solución.
- Ayudan a reutilizar código, facilitando las decisiones de diseño (herencia, composición, delegación).
- Nos permiten hacer un diseño preparado para los cambios de mantenimiento.

Los patrones se clasifican según su propósito:

- Patrones de **creación**: para creación de instancias.
- Patrones **estructurales**: relaciones entre clases, combinación y formación de estructuras mayores.
- Patrones de **comportamiento**: interacción y cooperación entre clases.

En esta sesión estudiaremos algunos de los patrones más utilizados en las aplicaciones.

- Singleton
- Composite
- Strategy
- Template Method
- Fachada
- Cadena de Responsabilidad

Si se quiere profundizar en el estudio de patrones recomendamos visitar:

http://java.sun.com/blueprints/corej2eepatterns/Patterns/index.html
http://www.programacion.net/java/tutorial/patrones/
http://www.programacion.net/java/tutorial/patrones2/

2. Singleton

Este patrón de diseño está diseñado para restringir la creación de objetos pertenecientes a una clase. Su intención consiste en garantizar que una clase sólo tenga una instancia y proporcionar un punto de acceso global a ella.

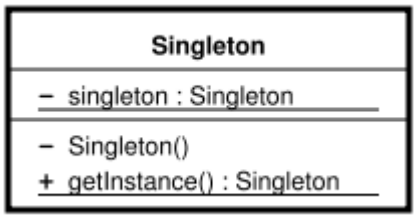
El patrón Singleton se implementa creando en nuestra clase un método que crea una instancia del objeto sólo si todavía no existe alguna. Para asegurar que la clase no puede ser instanciada nuevamente se regula el alcance del constructor haciendo privado.

Las situaciones más habituales de aplicación de este patrón son aquellas en las que dicha clase ofrece un conjunto de utilidades comunes para todas las capas (como puede ser el sistema de log, o el manejo de fechas) o cuando cierto tipo de datos debe estar disponible para todos los demás objetos de la aplicación.

El patrón Singleton provee una única instancia global gracias a que:

- La propia clase es responsable de crear la única instancia.
- Permite el acceso global a dicha instancia mediante un método de clase.
- Declara el constructor de clase como privado para que no sea instanciable directamente.

Esto es:



2.1. Ejemplo de Uso

Crearemos un proyecto llamado tema6, dentro del mismo crearemos un package llamado tema6 y una clase llamada LogSingleton cuyo código será:

```
package tema6;

public class LogSingleton {
int DEBUG=1;

/**objeto Singleton*/
private static LogSingleton miLogSingleton= new LogSingleton();

private LogSingleton(){
      log("Eventos de Usuario:") ;
      log("");
}

public static LogSingleton getInstance() {
 return miLogSingleton;
}
```

```
public void log(String contenido) {
      if (DEBUG==1)
      {
      System.out.println(contenido);
      }
}
}
```

Y para poder probarlo crearemos una nueva clase llamada PruebaSingleton cuyo código será:

```
package tema6;

public class PruebaSingleton {
      public static void main(String[] args) {
            LogSingleton.getInstance().log("Probando....");
            LogSingleton.getInstance().log("La clase singleton");
      }
}
```

Cuya sálida será:

```
Eventos de Usuario:

Probando....
La clase singleton
```

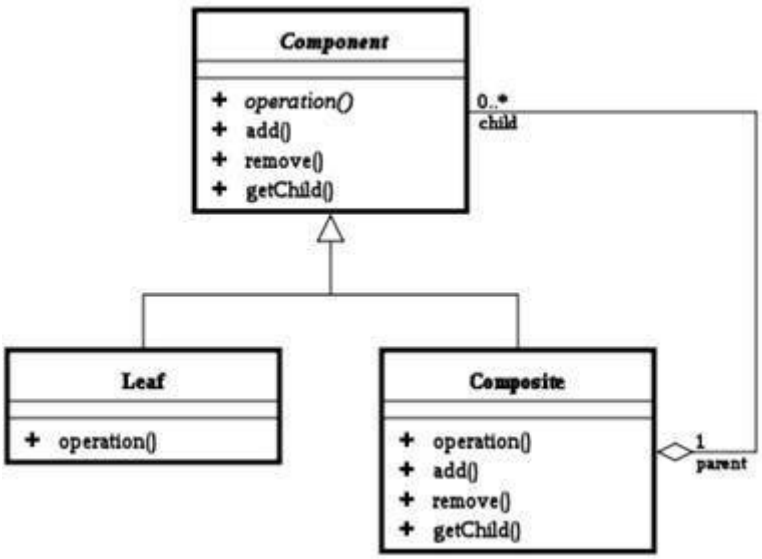
3. Composite

El patrón Composite sirve para construir objetos complejos a partir de otros más simples y similares entre sí, gracias a la composición recursiva y a una estructura en forma de árbol.

El patrón Composite:

- Nos permite crear un estructura que pueda tener cualquier nivel de complejidad y sea dinámica.
- Simplifica el programa al tratar a todos los objetos de igual forma. Para ello lo que hace es tratar de forma uniforme toda la estructura del componente, utilizando operaciones comunes para toda la jerarquía.
- Facilita agregar nuevas clases de componentes insertándolas en la jerarquía de clases como subclases de Compuesto o de Hoja.

Para lograrlo en este patrón se crea una jerarquía de objetos básicos y de composiciones de estos de forma recursiva.



Donde:
- **Component** es una clase abstracta con las operaciones base
- **Leaf** (Hoja) es una subclase de Component que no tiene más elementos (por eso solo redefine la operacion)
- **Composite** es una subclase de Component que permite agregar más elementos (bien de tipo Hoja o Composite)

3.1. Ejemplo de Uso

En este ejemplo mostraremos la relación laboral entre los Managers y Developers de una empresa. De este modo tendremos que Mark es el gerente general y de el dependen:

1. Michael que es developer
2. Daniel que es gerente. De Daniel,a su vez, dependen John y David que son developers.

Dentro de nuestro package tema6 crearemos una clase llamada EjemploComposite cuyo código será:

```
package tema6;
import java.util.ArrayList;
import java.util.Iterator;
import java.util.List;

abstract class Empleado {

        public void add(Empleado empleado) {
        }

        public Empleado getChild(int i) {
                return null;
        }

        public void remove(Empleado empleado) {
        }

        public abstract void print();
}
class Developer extends Empleado{
// NO SOBREESCRIBIREMOS LOS METODOS DE add,getChild y remove porque es un
// elemento HOJA
        private String nombre;
        private double salario;

        public Developer(String nombre,double salario){
          this.nombre = nombre;
          this.salario = salario;
        }

        public void print() {
          System.out.println("Nombre = "+this.nombre);
          System.out.println("Salario = "+this.salario);
          System.out.println("");
        }


}
class Manager extends Empleado{

       private String nombre;
       private double salario;
       List<Empleado> empleados = new ArrayList<Empleado>();

       public Manager(String nombre,double salario){
        this.nombre = nombre;
        this.salario = salario;
       }

       public void add(Empleado empleado) {
          empleados.add(empleado);
       }

       public Empleado getChild(int i) {
        return empleados.get(i);
       }
```

```
        public void print() {
               System.out.println("");
               System.out.println("Nombre = "+this.nombre);
               System.out.println("Salario = "+this.salario);
               System.out.println("Empleados a su cargo:");

               Iterator<Empleado> empleadosIterator = empleados.iterator();
                 while(empleadosIterator.hasNext()){
                  Empleado empleado = empleadosIterator.next();
                  empleado.print();
                 }


        }

        public void remove(Empleado empleado) {
         empleados.remove(empleado);
        }

}


public class EjemploComposite {

      public static void main(String[] args) {
               Empleado emp1=new Developer("John", 10000);
               Empleado emp2=new Developer("David", 15000);
               Empleado manager1=new Manager("Daniel",25000);
               manager1.add(emp1);
               manager1.add(emp2);

               Empleado emp3=new Developer("Michael", 20000);
               Manager generalManager=new Manager("Mark", 50000);
               generalManager.add(emp3);
               generalManager.add(manager1);

               generalManager.print();
      }

}
```

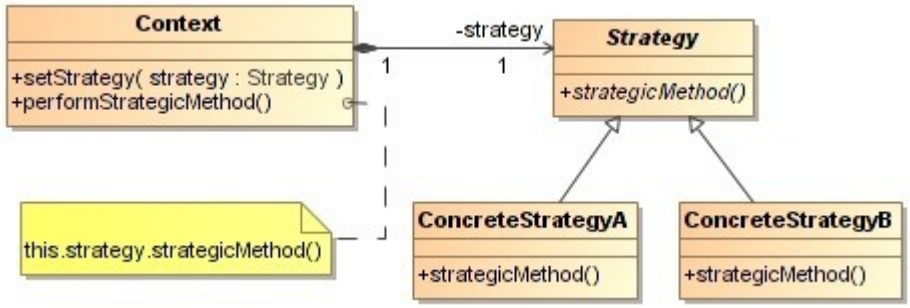
Cuya salida será:

```
Nombre = Mark
Salario = 50000.0
Empleados a su cargo:
Nombre = Michael
Salario = 20000.0


Nombre = Daniel
Salario = 25000.0
Empleados a su cargo:
Nombre = John
Salario = 10000.0

Nombre = David
Salario = 15000.0
```

4. Strategy

El patrón estrategia permite mantener un conjunto de algoritmos de entre los cuales el objeto cliente puede elegir aquel que le conviene e intercambiarlo dinámicamente según sus necesidades.

Aplicando el patrón a una clase que defina múltiples comportamientos mediante instrucciones condicionales, se evita emplear estas instrucciones, moviendo el código a clases independientes donde se almacenará cada estrategia.



Contexto (Context) : Es el elemento que usa los algoritmos.

Estrategia (Strategy): Declara una interfaz común para todos los algoritmos soportados. Esta interfaz será usada por el contexto para invocar a la estrategia concreta.

EstrategiaConcreta (ConcreteStrategy): Implementa el algoritmo utilizando la interfaz definida por la estrategia.

4.1. Ejemplo de Uso

Crearemos un coche que pueda elegir una estrategia de frenado con ABS o sin ABS. Para ello dentro de nuestro package tema6 crearemos una clase llamada EjemploEstrategia cuyo código será:

```
package tema6;

interface Strategy{
    public void comofrena();
}

class Coche {
      Strategy c;

      public void setEstrategia(Strategy c) {
            this.c = c;
      }

      public void frenar()
      {
            c.comofrena();
      }
}
```

```
class EstrategiaABS implements Strategy{
	@Override
	public void comofrena() {
		System.out.println("Realizamos un frenado en 5 metros con ABS");
	}
}

class EstrategiaSinABS implements Strategy{
	@Override
	public void comofrena() {
		System.out.println("Realizamos un frenado en 10 metros sin ABS");
	}
}

class EjemploEstrategia {
	public static void main(String args[])
	{
		Coche context = new Coche();

		//Usamos la estrategia A
		Strategy estrategia_inicial = new EstrategiaABS();
		context.setEstrategia(estrategia_inicial);
		context.frenar();

		//Decidimos usar la estrategia B
		Strategy estrategia2 = new EstrategiaSinABS();
		context.setEstrategia(estrategia2);
		context.frenar();

		//Finalmente,usamos de nuevo la estrategia A
		context.setEstrategia(estrategia_inicial);
		context.frenar();

	}
}
```

Y cuya salida será:

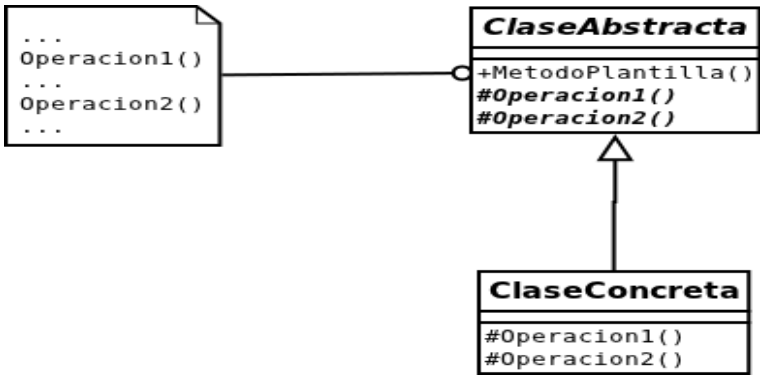
```
Realizamos un frenado en 5 metros con ABS
Realizamos un frenado en 10 metros sin ABS
Realizamos un frenado en 5 metros con ABS
```

5. Template Method

Este patrón se caracteriza por la definición, dentro de una operación de una superclase, de los pasos de un algoritmo, de forma que todos o parte de estos pasos son redefinidos en las subclases herederas de la citada superclase.

La utilización del patrón Método Plantilla es adecuada en los siguientes casos:

- Cuando contamos con un algoritmo con varios pasos que no cambian, de modo que dichos pasos invariantes serían implementados en una superclase, dejando la implementación de los pasos que cambian para las subclases.
- Para evitar la replicación de código mediante generalización: se factoriza el comportamiento común de varias subclases en una única superclase.
- Para controlar las extensiones de las subclases. El Método Plantilla utiliza métodos especiales (métodos de enganche o hooks) en ciertos puntos, siendo los únicos puntos que pueden ser redefinidos y, por tanto, los únicos puntos donde es posible la extensión.

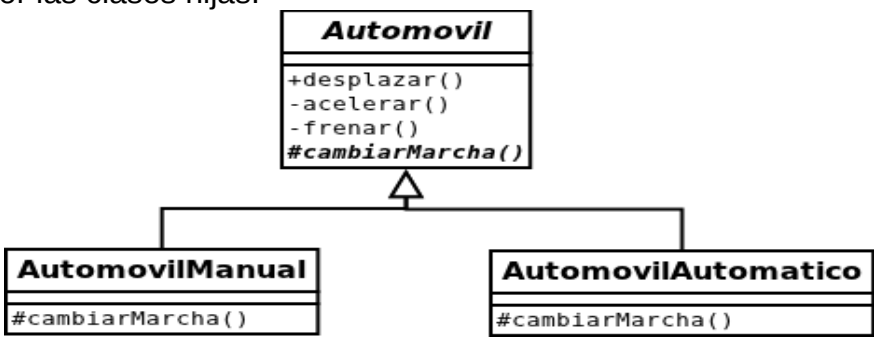


Clase Abstracta: proporciona la definición de una serie de operaciones (normalmente abstractas) que implementan los pasos de un algoritmo y que serán definidas en las subclases.
Se encarga también de la implementación de un método desde el cual son invocadas las operaciones. Dicho método actúa a modo de plantilla, de ahí el nombre de este patŕon, definiendo la secuencia de operaciones de un algoritmo.

Clase Concreta: implementa las operaciones definidas en la clase abstracta de la cual hereda, quedando así determinado el comportamiento específico del algoritmo definido en el método plantilla, para cada subclase.

5.1. Ejemplo de Uso

Vamos a implementar la siguiente relación de clases y con método cambiarMarcha a implementar por las clases hijas:



Para ello dentro de nuestro package tema6 crearemos una clase llamada EjemploTemplate cuyo código será:

```
package tema6;

abstract class AutoMovil
{
    public final void desplazar()
    {
        acelerar();
        cambiarMarcha();
        frenar();
    }
    private void acelerar()
    {
        System.out.println("Acelerando...");
    }

    private void frenar()
    {
        System.out.println("Frenando...");
    }

    protected abstract void cambiarMarcha();
}

class AutomovilManual extends AutoMovil
{
    protected void cambiarMarcha()
    {
        System.out.println("Cambiando de marcha de forma manual");
    }
}

class AutomovilAutomatico extends AutoMovil
{
    protected void cambiarMarcha()
    {
        System.out.println("Cambiando de marcha de forma automática");
    }
}

public class EjemploTemplate {
      public static void main(String[] args) {
            AutoMovil manual=new AutomovilManual();
            manual.desplazar();
            AutoMovil automatico=new AutomovilAutomatico();
            automatico.desplazar();
      }
}
```

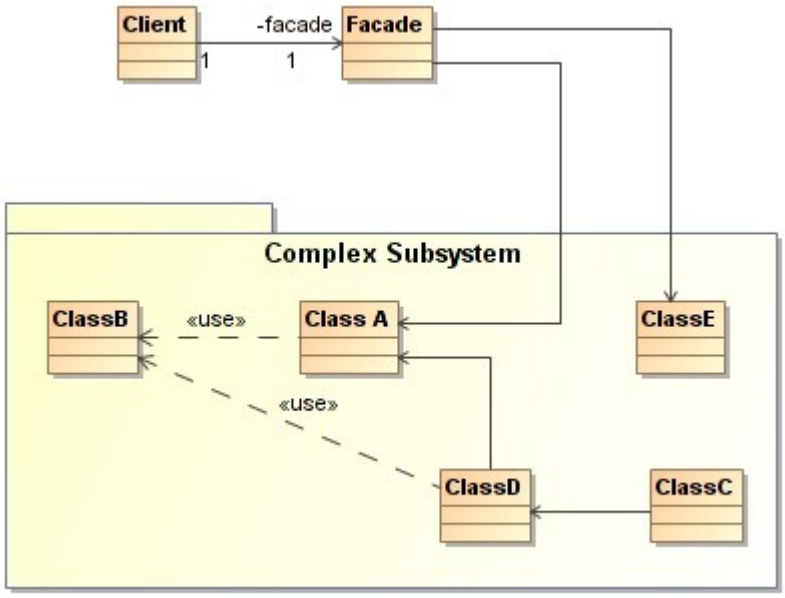
Mostrará por salida:

```
Acelerando...
Cambiando de marcha de forma manual
Frenando...
Acelerando...
Cambiando de marcha de forma automática
Frenando...
```

6. Patrón Fachada

El patrón fachada viene motivado por la necesidad de estructurar un entorno de programación y reducir su complejidad con la división en subsistemas, minimizando las comunicaciones y dependencias entre éstos.

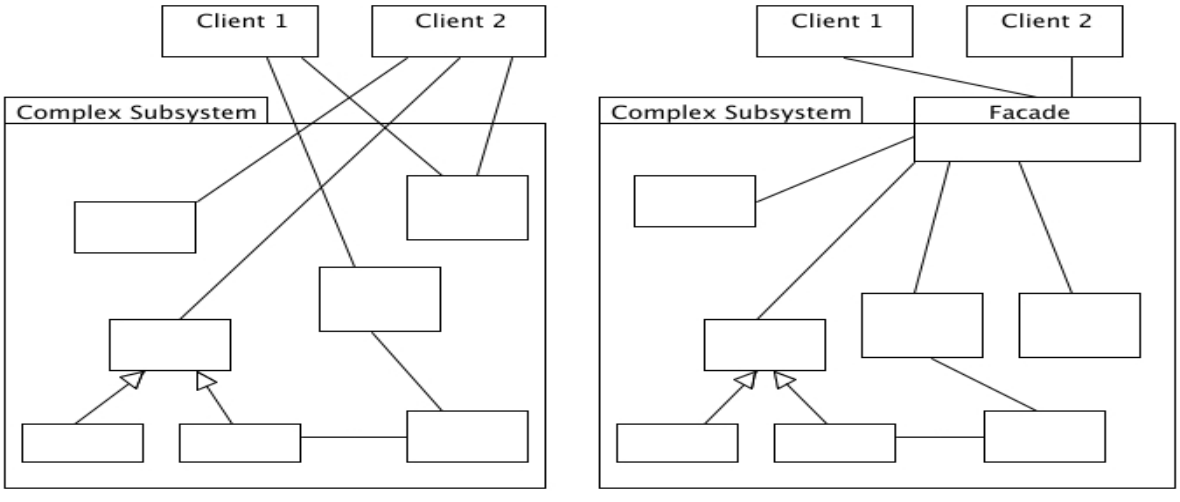
Se aplicará el patrón fachada cuando se necesite proporcionar una interfaz simple para un subsistema complejo



Fachada (Facade): conoce qué clases del subsistema son responsables de una determinada petición, y delega esas peticiones de los clientes a los objetos apropiados del subsistema.

Subclases (ClassA, ClassB,ClassC...): implementan la funcionalidad del subsistema. Realizan el trabajo solicitado por la fachada. No conocen la existencia de la fachada.

La principal ventaja del patrón fachada consiste en que para modificar las clases de los subsistemas, sólo hay que realizar cambios en la interfaz/fachada, y los clientes pueden permanecer ajenos a ello. Además, y como se mencionó anteriormente, los clientes no necesitan conocer las clases que hay tras dicha interfaz.



Como se aprecia en la figura anterior, al lado izquierdo tenemos un diagrama sin aplicación del patrón fachada, esto hace compleja la relación entre clases y produce un alto grado de acoplamiento, sin embargo, en la figura 2 vemos que se ha aplicado el patrón Fachada usando una clase como puerta de acceso al subsistema.

6.1. Ejemplo de Uso

Crearemos dos clases para sumar y restar, pero las ocultaremos al sistema mediante una fachada que las usa. Dentro de nuestro package tema6 crearemos una clase llamada EjemploFachada cuyo código será:

```
package tema6;

//clase que suma
class SumaNumeros {

public int suma(int pA,int pB){
    return pA + pB;
}
}

//clase que resta
class RestaNumeros{

public int resta(int pA,int pB){
    return pA-pB;
}
}

//clase Fachada que oculta la complejidad
class FachadaCalculadora {

      int numeroA = 0;
      int numeroB = 0;
      String operacion = null;

      //constructor que recibe los numeros a sumar o restar
      public FachadaCalculadora( int pA, int pB, String pOperacion){

      this.numeroA =pA;
      this.numeroB =pB;
      this.operacion=pOperacion;
      }

      public int operacion(){
         int resultado=0;
         if (operacion.equals("+")){
               SumaNumeros s = new SumaNumeros();
                resultado = s.suma(numeroA,numeroB);
         }

         if (operacion.equals("-")){
               RestaNumeros s = new RestaNumeros();
                resultado = s.resta(numeroA,numeroB);
         }
         return resultado;
      }
}
```

```java
//clase fachada

public class EjemploFachada{

  public static void main(String[] args) {
      int a=7;
      int b=4;
      String operacion="-";
    System.out.println("Se usa calculadora "
            + "para hacer: "+a+" "+ operacion +" "+b);
    FachadaCalculadora calc = new FachadaCalculadora(a,b,operacion);
    System.out.println("Resultado de la operacion:" + calc.operacion());
  }

}
```

Muestra por salida:

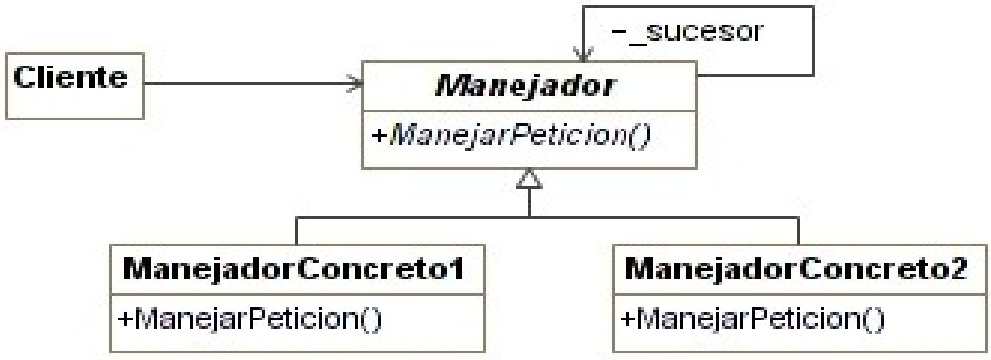
```
Se usa calculadora para hacer: 7 - 4
Resultado de la operacion:3
```

7. Cadena de Responsabilidad

Este es un patrón de comportamiento que evita acoplar el emisor de una petición a su receptor dando a más de un objeto la posibilidad de responder a una petición. Para ello, se encadenan los receptores y pasa la petición a través de la cadena hasta que es procesada por algún objeto.

Las ventajas de este patrón son:

- **Reduce el acoplamiento.** El patrón libera a un objeto de tener que saber qué otro objeto maneja una petición. Ni el receptor ni el emisor se conocen explícitamente entre ellos, y un objeto de la cadena tampoco tiene que conocer la estructura de ésta. Por lo tanto, simplifica las interconexiones entre objetos. En vez de que los objetos mantengan referencias a todos los posibles receptores, sólo tienen una única referencia a su sucesor.
- **Añade flexibilidad** para asignar responsabilidades a objetos. Se pueden añadir o cambiar responsabilidades entre objetos para tratar una petición modificando la cadena de ejecución en tiempo de ejecución.



Manejador: define una interfaz para tratar las peticiones. Opcionalmente, implementa el enlace al sucesor.
ManejadorConcreto: trata las peticiones de las que es responsable; si el ManejadorConcreto puede manejar la petición, lo hace; en caso contrario la reenvía a su sucesor.
Cliente: inicializa la petición a un Manejador Concreto de la cadena.

7.1. Ejemplo de Uso

Crearemos un cajero que sea capaz de ante una petición de sacar dinero comprobar que la petición es correcta (valor mayor que 0), que hay saldo (partimos de 100€) y si todo es correcto entregará el dinero. Para ello dentro de nuestro package tema6 crearemos una clase llamada EjemploCadenaResponsabilidad cuyo código será:

```
package tema6;

abstract class Manejador {
	protected Manejador sucesor;
```

```
	public void setSucesor(Manejador sucesor) {
		this.sucesor = sucesor;
	}

	public abstract void manejarPeticion(Peticion peticion);
}

class ComprobarPeticion extends Manejador{

	@Override
	public void manejarPeticion(Peticion peticion) {
		// TODO Auto-generated method stub
		if (peticion.getValue() == 0)
		{
			System.out.println("Su peticion no puede "
					+ "procesarse por solicitar al banco 0 €");

		}else
		if (peticion.getValue() < 0)
		{
			System.out.println("Su peticion no puede "
					+ "procesarse por solicitar "
					+ "al banco "+peticion.getValue()+" euros");
		}
		else
		{
			if (sucesor!=null)  sucesor.manejarPeticion(peticion);
		}
	}

}

class ComprobarDineroEnCuenta extends Manejador{

	@Override
	public void manejarPeticion(Peticion peticion) {
		int saldoExistente=100;//Tenemos 100 euros disponibles
		if (peticion.getValue() > saldoExistente) {
			System.out.println("Su peticion no puede procesarse "
					+ "por solicitar "
					+ "al banco "+peticion.getValue()+" euros "
					+ "cuando su saldo actual"
					+ " es de "+ saldoExistente +" euros");
		} else {
			if (sucesor!=null)  sucesor.manejarPeticion(peticion);
		}


	}

}

class DarDinero extends Manejador{

	@Override
	public void manejarPeticion(Peticion peticion) {

		System.out.println("Pasamos a darle "
				+ "sus "+ peticion.getValue() +" euros");
		if (sucesor!=null) sucesor.manejarPeticion(peticion);
```

```
	}

}

class Peticion {

	private int value;
	public Peticion (int value){
		this.value = value;
	}

	public int getValue() {
		return value;
	}

}


public class EjemploCadenaResponsabilidad {
	public static void main(String[] args)
	{


		Manejador m1 = new ComprobarPeticion();
		Manejador m2 = new ComprobarDineroEnCuenta();
		Manejador m3 = new DarDinero();

		m1.setSucesor(m2);
		m2.setSucesor(m3);

		m1.manejarPeticion(new Peticion(-20));
		m1.manejarPeticion(new Peticion(0));
		m1.manejarPeticion(new Peticion(105));
		m1.manejarPeticion(new Peticion(25));

	}
}
```

y cuya salida será:

```
Su peticion no puede procesarse por solicitar al banco -20 euros
Su peticion no puede procesarse por solicitar al banco 0 €
Su peticion no puede procesarse por solicitar al banco 105 euros cuando su saldo
actual es de 100 euros
Pasamos a darle sus 25 euros
```

8. Ejercicios

(2,5 puntos cada uno)

1- Crea una aplicación que implemente la base de una calculadora (sin la parte gráfica).

Usa los siguiente patrones para llevar a cabo la aplicación:

- Singleton->para mostrar mensajes por pantalla
- Template Method->para completar la operación especifica : Suma / Resta

Aconsejamos usar la siguiente clase para implementarlo::

```
abstract class Operacion
{
      double resultado=0;
      String operacion="";
    public final void realizaOperacion(double a,double b)
    {
        muestraOperacion(a,b);
        calcula(a,b);
        muestraResultado();
    }

    private void muestraOperacion(double a,double b)
    {
      LoggingSingleton.getInstance().log("Operacion :"+a+operacion+b);
    }

    private void muestraResultado()
    {
      LoggingSingleton.getInstance().log("El resultado es "+resultado);
    }

    protected abstract void calcula(double a,double b);
}
```

- Patrón Fachada-> para ocultar la complejidad del sistema

Se probará mediante la clase siguiente:

```
public class EjercicioCalculadora {

      public static void main(String[] args) {
            double a=7;
            double b=4;
            String operacion="-";
            LoggingSingleton.getInstance().log("Se usa calculadora "
                  + "para hacer: "+a+" "+ operacion +" "+b);
          FachadaCalculadoraOps calc = new FachadaCalculadoraOps(a,b,operacion);
          calc.operacion();
      }
}
```

2- Crea una aplicación llamada Gramática que usando el patrón cadena de responsabilidad compruebe para una cadena de texto:

1- si mide mas de 5 letras. Esto se comprueba mediante:

```
if (cadena.length()> 5)
System.out.println("La cadena "+cadena+"  tiene más 5 caracteres");
```

2- si empieza por mayúscula. Esto se comprueba mediante:

```
String _first=cadena.substring(0, 1);
if (_first.toUpperCase().equals(_first))
System.out.println("La cadena "+cadena+"  empieza por mayúscula");
```

3- si es un numero o no. Esto se comprueba mediante:

```
try
{
	Integer.parseInt(cadena);
	System.out.println("La cadena "+cadena+" es un numero");
}catch(Exception e)
{
	System.out.println("La cadena "+cadena+" NO es un numero");
}
```

Lo probaremos a través de la siguiente clase:

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


		Comprobador m1 = new ComprobarLongitud();
		Comprobador m2 = new ComprobarInicial();
		Comprobador m3 = new ComprobarNumero();

		m1.setSucesor(m2);
		m2.setSucesor(m3);

		m1.comprobar("prueba");
		m1.comprobar("EjemploEjercicio");
		m1.comprobar("1234");

	}
}
```

3- Crea una aplicación que será una máquina expendedora que mediante usar el patrón Estrategia diga la expresión Buenos días en inglés, francés, español o alemán.

Lo probaremos a través de la siguiente clase:

```
public class MaquinaExpendora {
	public static void main(String args[])
	{
		Maquina maquina = new Maquina();

		maquina.setEstrategia(new HablaFrances());
```

```
            maquina.saludar();
            maquina.setEstrategia(new HablaIngles());
            maquina.saludar();
            maquina.setEstrategia(new HablaEspanyol());
            maquina.saludar();
            maquina.setEstrategia(new HablaAleman());
            maquina.saludar();
      }
}
```

4- Crea una aplicación llamada JerarquiaEmpresa que usando el patrón Composite sea capaz de implementar las relaciones laborales en una empresa entre :

1 director es responsable de 2 jefes de zona.

Cada jefe de zona es responsable de 2 comerciales.

Lo probaremos a través de la siguiente clase:

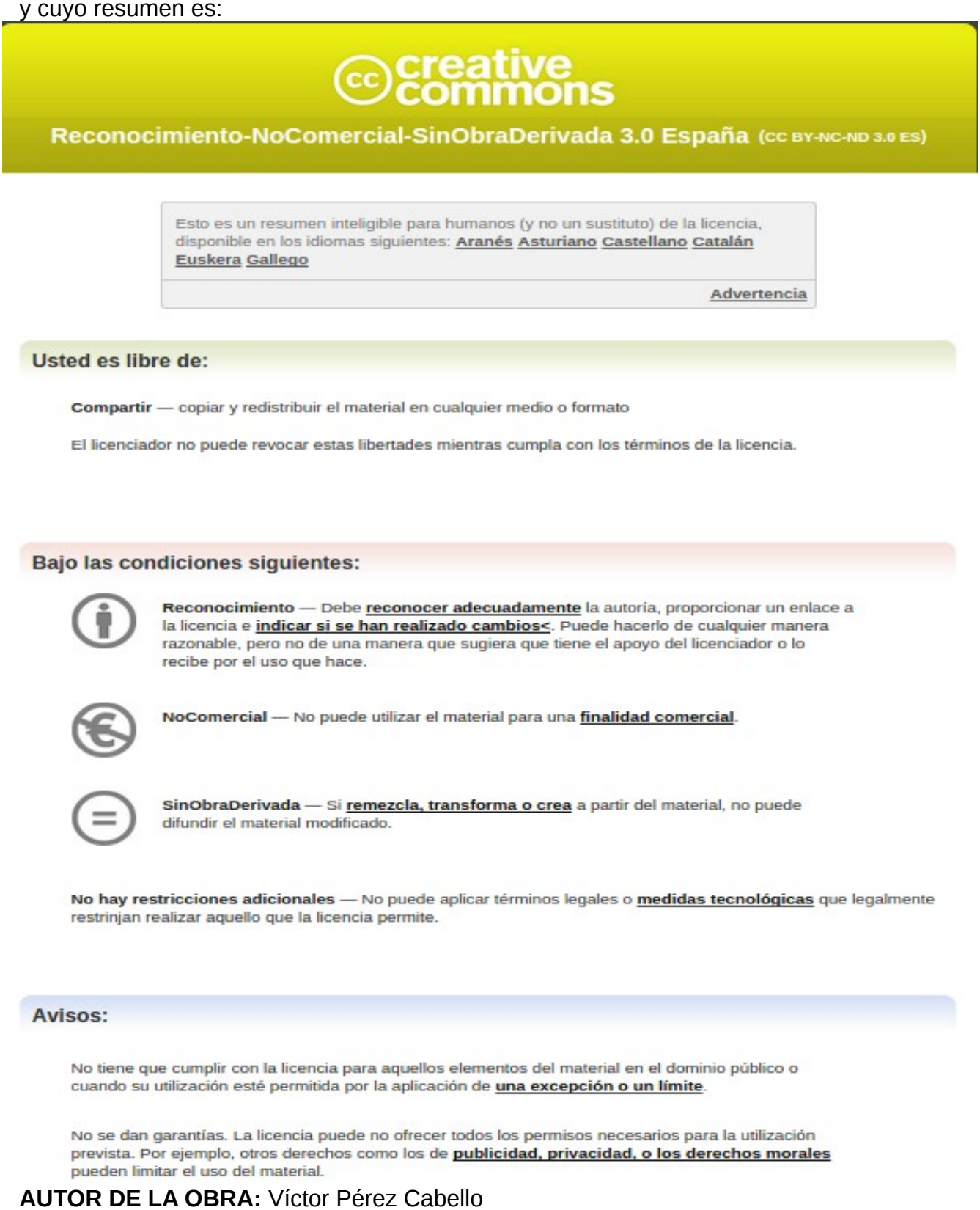
```
public class JerarquiaEmpresa {

      public static void main(String[] args) {
            EmpleadoBase emp1=new Comercial("Juan");
            EmpleadoBase emp2=new Comercial("Pepe");
            EmpleadoBase emp3=new Comercial("Carlos");
            EmpleadoBase emp4=new Comercial("Josep");
            EmpleadoBase emp5=new Directivo("Josep","Jefe de Zona");
            emp5.add(emp1);
            emp5.add(emp2);
            EmpleadoBase emp6=new Directivo("Enrique","Jefe de Zona");
            emp6.add(emp3);
            emp6.add(emp4);
            EmpleadoBase emp7=new Directivo("Fracisco","Director General");
            emp7.add(emp5);
            emp7.add(emp6);


            emp7.print();
      }

}
```

9. Ampliar conocimientos

http://best-practice-software-engineering.ifs.tuwien.ac.at/patternmap.html

10. Licencia del documento

Esta obra se rige bajo la licencia cuyo texto completo está en:
http://creativecommons.org/licenses/by-nc-nd/3.0/es/legalcode.es
y cuyo resumen es:



AUTOR DE LA OBRA: Víctor Pérez Cabello