

# Métodos Avanzados de Programación Científica y Computación

M<sup>a</sup> Luisa Díez Platas

## Tema 3. Introducción a los patrones de diseño para problemas orientados a objetos

# ¿Cómo estudiar este tema?

» TEMA 3. INTRODUCCIÓN A LOS PATRONES DE DISEÑO PARA PROBLEMAS ORIENTADOS A OBJETOS

[ Esquema Tema ]

IDEAS CLAVE	LO + RECOMENDADO	+ INFORMACIÓN	TEST
<p>¿Cómo estudiar este tema?</p> <p>Qué son los patrones de diseño</p> <p>Patrón <code>Factory</code></p> <p>Patrón <code>Singleton</code></p> <p>Patrón <code>Observer</code></p> <p>Patrón <code>Composite</code></p> <p>Introducción a la POO distribuida</p>	<p><b>No dejes de leer...</b></p> <p>Importancia de los patrones de diseño</p> <p>Patrones</p> <p><b>No dejes de ver...</b></p> <p> Patrón <code>Factory</code></p>	<p><b>A fondo</b></p> <p>Patrones de fabricación</p> <p>Patrón <code>Singleton</code></p> <p>Patrón <code>Observer</code></p> <p>Diseño de patrones</p> <p>Implementar patrones en UML</p> <p>Patrones en Java</p> <p>Patrones de diseño en Java</p> <p><b>Recursos externos</b></p> <p>Eclipse</p>	

- ¿Qué es un patrón de diseño OO?
- Tipos y características
- Patrón `Singleton`
- Patrón `Factory` method
- Patrón `Composite`
- Patrón `Observer`

# Un poco de historia

- ▶ El arquitecto Christopher Alexander escribió varios libros sobre planificación urbana y arquitectura de edificios e introdujo el concepto de patrón.
- ▶ 1987 - Ward Cunningham y Kent Beck usaron varias de las ideas de Alexander para desarrollar un pequeño lenguaje de patrones para programadores novatos de Smalltalk.
- ▶ 1991 - Jim Coplien publica *Advanced C++ Programming Styles and Idioms* que compila un catálogo de *idioms* para C++.
- ▶ 1994 - Se publica el libro *Design Patterns : Elements of Reusable ObjectOriented Software* por GoF.

# ¿Qué es un Patrón?

- ▶ Un patrón es una información con nombre que captura la estructura esencial de soluciones comprobadas a problemas recurrentes en un cierto contexto y sistema de fuerzas.
- ▶ Un patrón es al mismo tiempo una cosa que ocurre en el mundo real y la regla que nos dice cómo crear esa cosa, la descripción del proceso que da lugar a esa cosa
- ▶ Un **¿anti-patrón?** nos dice lo que no hay que hacer; describen malas soluciones.

# Ventajas

- ▶ Ayudan a los desarrolladores de software a resolver problemas comunes encontrados durante todo el proceso de la Ingeniería del Software.
- ▶ Ayudan a crear un vocabulario y un lenguaje compartidos para comunicar comprensión profunda (*insight*) y experiencia sobre dichos problemas y sus soluciones.
- ▶ Comunican las razones de determinadas decisiones de diseño y no únicamente las soluciones.
- ▶ Lo hacen dando un nombre y codificando formalmente dichos problemas y sus soluciones.

# Tipos de Patrones

- ▶ El foco inicial, sobre todo a raíz del libro de GoF, fue sobre patrones de diseño.
- ▶ Hay otros tipos:
  - patrones de análisis
  - patrones organizativos
  - organización del desarrollo
  - proceso software
  - planificación de proyectos
  - ingeniería de requerimientos
  - . . .

# Patrones: Clasificación

- ▶ Según Frank Buschman los patrones se pueden clasificar en:
  - **ARQUITECTURA**: Expresan una estructura fundamental de organización de los sistemas SW. Proveen un conjunto de subsistemas predefinidos, especificando sus responsabilidades y relaciones.
  - **DISEÑO**: Proporcionan un esquema para refinar subsistemas o componentes. Resuelven problemas específicos de diseño.
  - **IDIOMS**: Son específicos de un lenguaje de programación. Describen cómo implementar ciertos aspectos de un problema utilizando las características de un lenguaje de programación.

# Patrones de Diseño: Clasificación (II)

- ▶ Teniendo en cuenta el nivel de detalle:
  - **ARQUITECTURA**: Afectan a la estructura global del sistema.
  - **DISEÑO**: Definen microestructuras de subsistemas y componentes.
  - **IDIOMS**: Se centran en detalles de la estructura y comportamiento de un componente.





# Patrones de Diseño

- ▶ Un Patrón de Diseño identifica, abstrae y nombra los aspectos elementales de una estructura de diseño.
- ▶ Identifica:
  - participantes y sus instancias
  - colaboraciones
  - distribución de responsabilidades
- ▶ Cada patrón de diseño se centra en un problema de diseño particular y describe cuando es o no aplicable a la luz de otras restricciones de diseño.

# Patrones de Diseño (II)

- ▶ El libro que más ha contribuido a la popularización de los patrones:

Erich Gamma, Richard Helm, Ralph Johnson y John Vlissides  
*Design Patterns: Elements of Reusable Object Oriented Software* Addison Wesley 1995

- Compila un catálogo de 23 patrones de diseño

# Patrones de Diseño (III)

- ▶ La descripción de los patrones se realiza mediante una ‘ficha’ con la siguiente estructura:
  - **Nombre y Clasificación:** Un nombre que sucintamente transmita la esencia del patrón. Ha de escogerse bien ya que formará parte del vocabulario de diseño.
  - **Intención:** Una breve descripción de cuál es el problema que el patrón resuelve.
  - **Alias:** Otros nombres, si existen, del patrón.
  - **Motivación:** Un escenario que ilustra el problema de diseño y cómo el patrón lo resuelve. Ayuda a comprender la descripción más abstracta del patrón del resto de apartados.
  - **Aplicabilidad:** Situaciones en que el patrón es aplicable. Cómo reconocerlas.
  - **Estructura:** Representación gráfica de las clases del patrón utilizando algún tipo de notación, diagramas de interacción, ....

# Patrones de Diseño (IV)

- **Participantes:** Clases participantes y sus responsabilidades
- **Colaboraciones:** Cómo colaboran las clases para llevar a cabo sus responsabilidades.
- **Consecuencias:** Cómo el patrón cumple los objetivos. Compromisos y resultados al usar el patrón.
- **Implementación:** Ayudas, trucos, elementos específicos del lenguaje a la hora de implementar el patrón.
- **Código de ejemplo:** Fragmentos que ilustran la implementación en algún(os) lenguaje(s).
- **Usos conocidos:** Ejemplos de uso del patrón en sistemas conocidos.
- **Patrones relacionados:** Patrones relacionados, alternativos para el problema, etc.

# Patrones de Diseño . Esquema

**AMBITO**  
a qué es aplicable  
el patrón

**TIPO**



	<b>CREACIONAL</b> Creación de objetos	<b>ESTRUCTURAL</b> Composición de objetos	<b>COMPORTAMIENTO</b> Distribución de responsabilidades
<b>CLASES</b>	Delegan parte de la creación a las subclases	Usan la herencia para componer clases	Usan la herencia para describir algoritmos y flujos de control
<b>OBJETOS</b>	Delegan parte de la creación a otro objeto	Describen formas de ensamblar objetos	Cómo grupos de objetos colaboran para llevar a cabo una tarea

# Patrones de Diseño . Esquema

	CREACIONAL	ESTRUCTURAL	COMPORTAMIENTO
Clase	Factory Method	Adapter (class)	Interpreter Template Method
Objeto	Abstract Factory Builder Protoype Singleton	Adapter (object) Bridge Composite Decorator Facade Flyweight Proxy	Chain of Responsibility Command Iterator Mediator Memento Observer State Strategy Visitor

# Patrones creacionales

- ▶ Abstraen el proceso de la instanciación (creación)
- ▶ Ayuda a que el sistema sea independiente de cómo se crean, componen y representan los objetos.
- ▶ Un patrón creacional de clase usa la herencia para las distintas clases que instancia, mientras que uno de objeto delega la creación a otro objeto.

# Patrones creacionales: *Singleton*

## ► Propósito

Asegurar que una clase tiene una instancia única y proporcionar un punto de acceso global a ella

## ► Motivación

Algunas clases tienen una única instancia.

- Una solución sería utilizar una variable global pero ello no evita que podamos instanciar múltiples objetos.
- Una solución mejor es hacer que la propia clase gestione su única instancia.

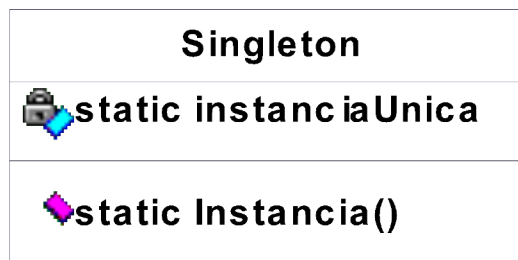


# Patrones creacionales: *Singleton* (II)

## ► Aplicabilidad

- Se necesita que una clase tenga una instancia única globalmente accesible.
- La instancia única podría ser extensible por una subclase, y los clientes podrían usar una instancia extendida sin modificar su código.

## ► Estructura



Si <no creada> crear instancia Unica  
return instanciaUnica

# Patrones creacionales: *Singleton* (III)

## ► Participantes

- **Singleton:** define una operación Instancia para acceder a su única instancia.
- Puede ser la responsable de crear su única instancia
- Se oculta la instancia y se oculta el constructor de la clase

## ► Colaboraciones

- Los clientes acceden a la instancia de Singleton sólo a través de la operación Instancia.

# Patrones creacionales: *Singleton* (IV)

## ► Consecuencias

- Controla el acceso a la única instancia
- Reduce el espacio de nombre
- Permite el refinamiento de operaciones y representaciones
- Permite un numero variable de instancias

## ► Implementación

- Para garantizar una única instancia se debe ocultar la operación que crea las instancias

# Patrones creacionales: *Singleton* (V)

```
Public class Singleton {  
  
    private static Singleton singleton = new Singleton( );  
  
    private Singleton(){ }  
    //private hace que el constructor no sea invocable.  
  
    //a través del método getInstance accedemos a la única instancia del  
objeto.  
    public static Singleton getInstance( ) {  
        return singleton;  
    }  
}
```

# Patrones creacionales: *Singleton* (VI)

- Uso del patrón para gestionar varias aperturas de un archivo

```
class Archivo{  
    //Parte estática, control de singleton:  
    static private Archivo instancia;  
    static int nAperturas = 0;  
    static private String nombre;  
    private Archivo(String Nombre)  
    {  
        this.nombre=new String(Nombre);  
    }  
}
```

```
//método para obtener las instancias de "Archivo"  
static public Archivo Abrir(String Nombre){  
    if (instancia == null){ //es la primera apertura  
        instancia = new Archivo(Nombre);  
    }  
    if (instancia.nombre.equals(Nombre)){  
        nAperturas++;  
        return instancia;  
    }  
    else  
        return null;  
}
```

# Patrones creacionales: *Singleton* (VII)

```
static public boolean Cerrar(String Nombre)
{
    if (Nombre.equals(instancia.nombre)){
        nAperturas--;
        if(nAperturas == 0){
            instancia = null;
        }
        return true;
    }
    else
        return false;
}
```

```
public String obtenerNombre()
{
    //se devuelve una "copia" del nombre
    return new String(nombre);
}
```

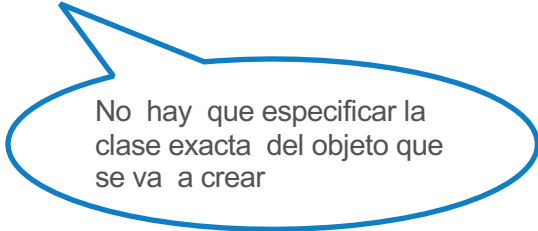
# Patrones creacionales: *Factory* ó *Factory Method*

## ► Propósito

- Define una interfaz para la creación de objetos, permitiendo que sean las subclasses las que decidan qué clase necesitan instanciar.

## ► Aplicabilidad

- Si en una clase no se puede prever la clase de objetos que debe crear.
- “Una clase quiere que sean sus subclasses quienes especifiquen los objetos que esta crea”.
- Las clases delegan la responsabilidad en una de entre varias clases auxiliares y se pretende localizar concretamente en qué subclase de auxiliar se delega.



No hay que especificar la clase exacta del objeto que se va a crear

# Patrones creacionales: *Factory* ó *Factory Method*

## ► Uso

- Cuando una clase no sabe qué subclases serán necesarias para crear objetos
- Cuando una clase quiere que sus subclases especifiquen los objetos a crear.
- Cuando las clases superiores eligen la creación de objetos para sus subclases.



# Patrones creacionales: *Factory* ó *Factory Method*

## ► Implementación

- Los objetos se crean llamando a un método de fábrica

El método se especifica en una interfaz o clase abstracta y se implementa en las hijas

ó

Se implementa en una clase base y opcionalmente se anula en clases derivadas, en lugar de llamar a un constructor.

el código interactúa únicamente con la interfaz resultante o con la clase abstracta, ➔funcionará con cualquier clase que implemente esa interfaz o que extienda esa clase abstracta.

# Patrones creacionales: *Factory* ó *Factory Method*(II)

## ► Estructura

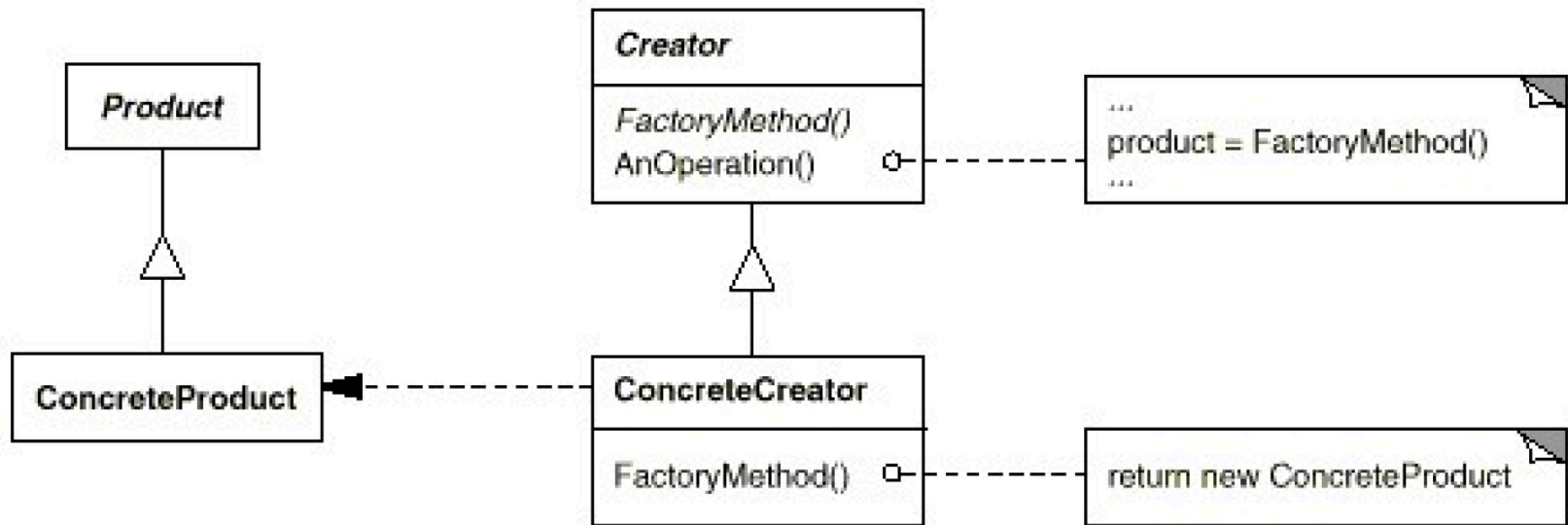


Diagrama OMT de *Factory Method*. Fuente: Gamma, E., Helm, R., Johnson, R. & Vlissides, J. M. (1995). *Design Patterns. Elements of Reusable Object-Oriented Software*. Indiana: Addison Wesley.

# Patrones creacionales: *Factory* ó *Factory Method*(III)

- ▶ Participantes
  - **Creator**: abstracta
  - **ConcreteCreator**: implementa el método de creación
  - **ProductoAbstracto**: declara una interfaz para un tipo de producto
  - **ProductoConcreto**: define un producto para ser creado por la fabrica concreta correspondiente. Implementa la interfaz del producto abstracto

# Patrones creacionales: *Factory* ó *Factory Method*(IV)

## ► Consecuencias

- Crear objetos dentro de una clase con un método de fabricación es siempre más flexible que hacerlo directamente.
- Conecta jerarquías de clases paralelas.

# Patrones creacionales: *Factory* ó *Factory Method(V)*

## ► Implementación

```
abstract class Creator{  
    // Definimos método abstracto  
    public abstract Product factoryMethod();  
}  
  
public class ConcreteCreator extends Creator{  
    public ConcreteProduct factoryMethod() {  
        return new ConcreteProduct();  
    }  
}  
  
public interface Product{  
    public void operacion();  
}
```

# Patrones creacionales: *Factory* ó *Factory Method*(VI)

## ► Implementación

```
public class ConcreteProduct implements Product{
    public void operacion(){
        System.out.println("Una operación de este producto");
    }
}

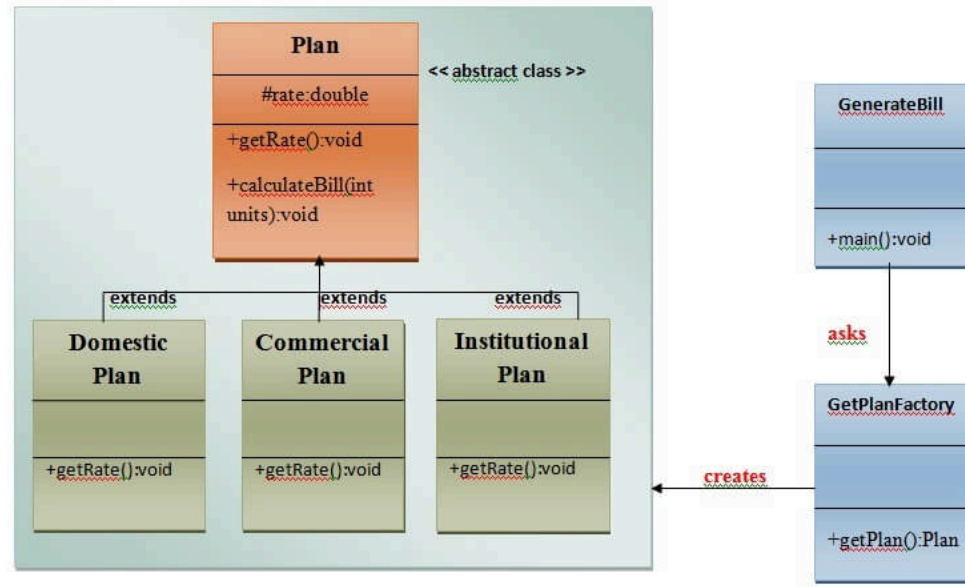
public class Factory{
    public static void main(String args[]){
        Creator aCreator;
        aCreator = new ConcreteCreator();
        Product producto = aCreator.factoryMethod();
        producto.operacion();
    }
}
```

# Patrones creacionales: *Factory* ó *Factory Method*(VII)

- ▶ Secuencia de uso
  - El cliente le solicita al *creador concreto* la creación del *Producto Concreto* .
  - El *Creador concreto* crea el *Producto Concreto* y lo facilita
  - El *ConcreteFactory* regresa el *ConcreteProduct* creado.

# Patrones creacionales: *Factory* ó *Factory Method*. Ejemplo

- ▶ Plan de electricidad



Fuente: <https://www.javatpoint.com/factory-method-design-pattern>



# Patrones creacionales: *Factory* ó *Factory Method*. Ejemplo

```
import java.io.*;

abstract class Plan{

    protected double rate;

    abstract void getRate();

    public void calculateBill(int units){
        System.out.println(units*rate);
    }
}
```

# Patrones creacionales: *Factory* ó *Factory Method*. Ejemplo

```
class DomesticPlan extends Plan{
    //@override
    public void getRate(){
        rate=3.50;
    }
}

class CommercialPlan extends Plan{
    //@override
    public void getRate(){
        rate=7.50;
    }
}
```

```
class InstitutionalPlan extends Plan{
    //@override
    public void getRate(){
        rate=5.50;
    }
}
/end of InstitutionalPlan class.
```

## Patrones creacionales: *Factory* ó *Factory Method*. Ejemplo

```
class GetPlanFactory{

    //use getPlan method to get object of type Plan
    public Plan getPlan(String planType){
        if(planType == null){
            return null;
        }
        if(planType.equalsIgnoreCase("DOMESTICPLAN")) {
            return new DomesticPlan();
        }
        else if(planType.equalsIgnoreCase("COMMERCIALPLAN")){
            return new CommercialPlan();
        }
        else if(planType.equalsIgnoreCase("INSTITUTIONALPLAN")) {
            return new InstitutionalPlan();
        }
        return null;
    }
}

//end of GetPlanFactory class.
```

# Patrones creacionales: *Factory* ó *Factory Method*. Ejemplo

```
import java.io.*;

class GenerateBill{

    public static void main(String args[])throws IOException{

        GetPlanFactory planFactory = new GetPlanFactory();

        System.out.print("Enter the name of plan for which the bill will be generated: ");
        BufferedReader br=new BufferedReader(new InputStreamReader(System.in));

        String planName=br.readLine();
        System.out.print("Enter the number of units for bill will be calculated: ");
        int units=Integer.parseInt(br.readLine());

        Plan p = planFactory.getPlan(planName);
        //call getRate() method and calculateBill()method of DomesticPaln.

        System.out.print("Bill amount for "+planName+" of "+units+" units is: ");
        p.getRate();
        p.calculateBill(units);
    }

} //end of GenerateBill class.
```

# Patrones estructurales: *Composite*

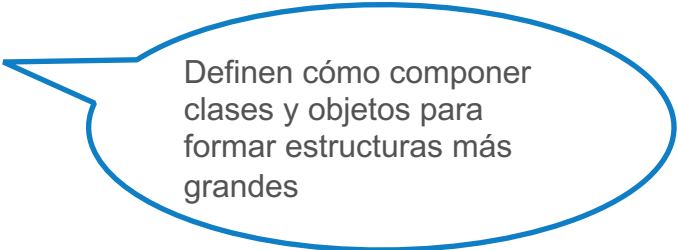
## ► Propósito

Componer objetos en una estructura de árbol para representar relaciones todo-parte y que los clientes traten uniformemente objetos simples y compuestos

## ► Motivación

En muchos casos es necesario

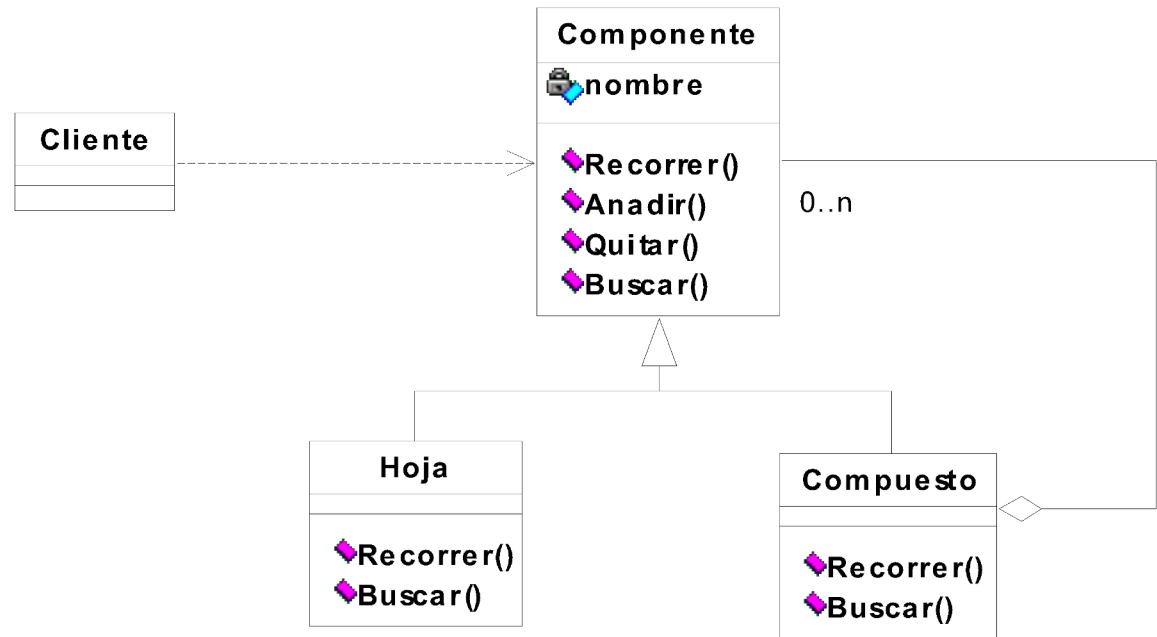
- definir clases y componentes que almacene sus objetos.
- debe tratar a los objetos contenedores y a los objetos simples de forma distinta.
- permite representar mediante una clase abstracta tanto a los objetos simples como a los contenedores



Definen cómo componer clases y objetos para formar estructuras más grandes

# Patrones estructurales: *Composite*

- ▶ Aplicabilidad
  - Se quiere representar jerarquías todo-parte
  - Los clientes deben ignorar las diferencias entre objetos individuales y objetos compuestos.
- ▶ Estructura



# Patrones estructurales: *Composite*

## ► Participantes

### – **Componente:**

- Declara la interfaz para los objetos de la composición
- implementa comportamiento por defecto para dicho interfaz
- declara interfaz para acceder y manipular objetos hijos

### – **Compuesto:** Define el comportamiento de los objetos que pueden contener a otros y almacena componentes hijos

### – **Hoja:** Define el comportamiento de los objetos primitivos

### – **Cliente:** Interacciona los objetos de la composición a través del interfaz Componente.

# Patrones estructurales: *Composite*

## ► Consecuencias

- Cliente más sencillo al otorgar uniformidad de acceso a los compuestos y a las hojas
- Facilita la tarea de añadir nuevos tipos de componentes
- Dificulta la tarea de restringir los componentes que forman el compuesto

## ► Implementación

- Maximizar la interfaz de *Componente*
- Ordenación de los hijos
- Borrado de los componentes
- Estructura de almacenamiento



# Patrones estructurales: *Composite*

```
//interfaz Componente
public interface Componente
{
    public void getNombre();
    public void agregar(Componente c);
    public void eliminar(Componente c);
    public Componente getComponente();
}
```

```
//Componente1
public class Componente1 implements Componente{
    public Componente getComponente (){return null;}
    public void getNombre{
        System.out.println("Componente1");
    }
    public void agregar(Componente c){}
    public void eliminar(){}
```

```
//Componente2
public class Componente2 implements Componente{
    public Componente getComponente (){return null;}
    public void getNombre{
        System.out.println("Componente2");
    }
    public void agregar(Componente c){}

    public void eliminar(Componente c){}
}
```

# Patrones estructurales: *Composite*

```
public class Composite implements Componente{
    private ArrayList listaComponente= new ArrayList();
    public Composite (){
    }
    public Composite (Componente c){
        this.agregar(c);
    }

    public Componente getComponente(){
        return null;
    }
    public getNombre(){
    for (int i=0; i<listaComp.size();i++){
        ((Component)listaComponente.get(i)).getNombre();
    }
    }
    public agregar (Component c){
        listaComponente.add(c);
    }
    public eliminar (Component c){
        listaComponente.remove(c);
    }
}
```

```
//Cliente
public class Cliente {
    public static void main(String[]args){
        Componente [] c={new Componente1(), new Componente2()};
        Componente c2=new Composite();
        c2.agregar(c[0]);
        c2.agregar(c[1]);
        Componente c3=new Composite(c2);
        c3.agregar(c[0]);
        c3.getNombre();
        c3.agregar(c[1]);
        c3.eliminar(c2);
        System.out.println();
        c3.getNombre();
    }
}
```

# Patrones de comportamiento: *Observer*

## ► Propósito

Avisar a los objetos “observadores” de los cambios que se producen en el objeto observado

## ► Motivación

En muchos casos es necesario que

- Los cambios que se producen en ciertos objetos afecten a otros que los controlan

# Patrones de comportamiento: *Observer*

## ► Aplicabilidad

- Los cambios de estado en un objeto (observado) requieren cambios en otros objetos
- (observadores).
- No se conoce el número de objetos que deben cambiar cuando cambia el objeto
- observado.
- Un objeto sea capaz de notificar algo a otros sin saber quiénes son esos objetos.
- Una abstracción tiene dos aspectos, que dependen uno del otro.

# Patrones de comportamiento: *Observer*

## ► Estructura

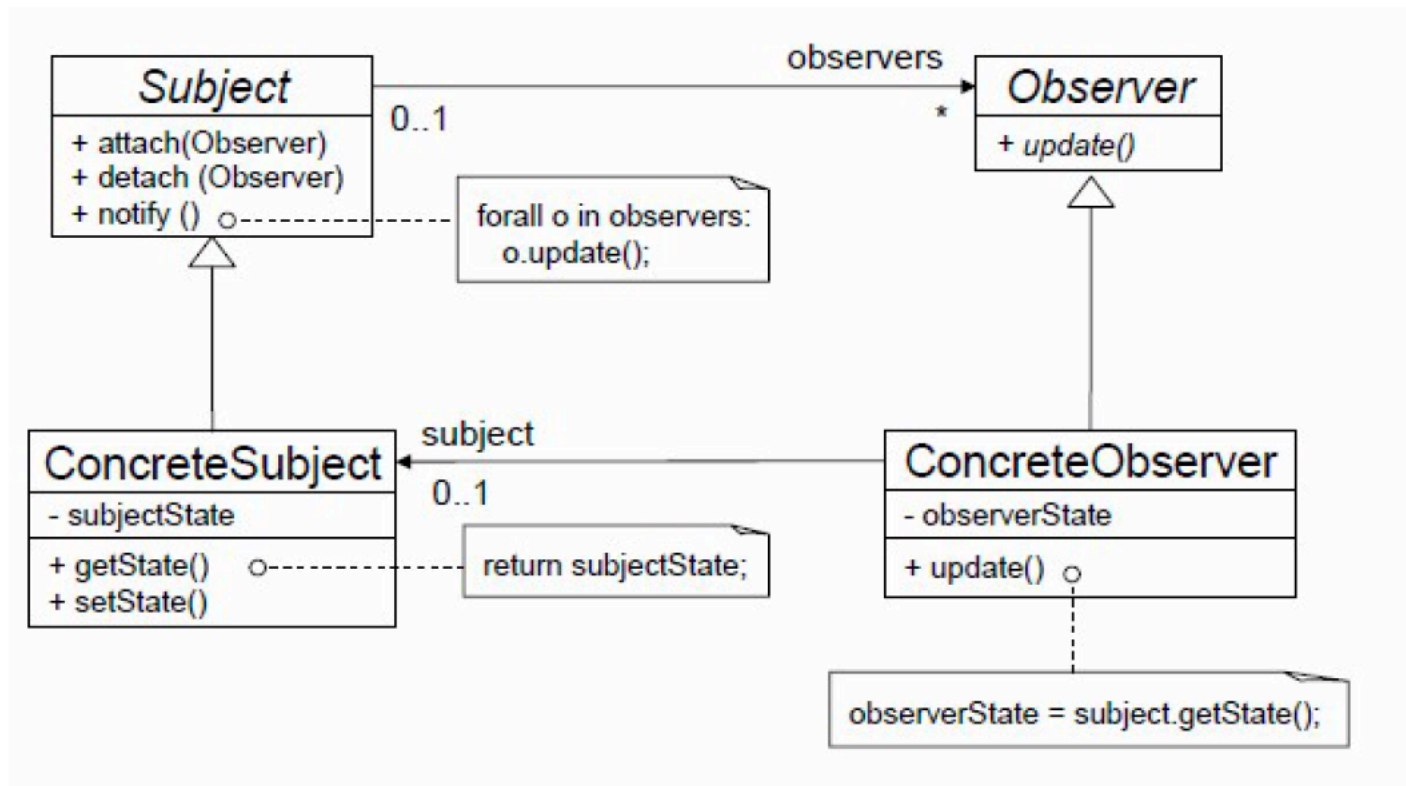


Diagrama patrón observer. Fuente: Gamma, E., Helm, R., Johnson, R. & Vlissides, J. M. (1995). *Design Patterns. Elements of Reusable Object-Oriented Software*. Indiana: Addison Wesley.

# Patrones de comportamiento: *Observer*

## ► Participantes

- **Observado abstracto**
  - Declara la interfaz para los objetos de la observación
- **Observado concreto:** Define el comportamiento de los objetos de las clases que implementan el abstracto
- **Observador abstracto:** define la relación con los observados
- **Observador concreto:** Usa los objetos concretos observados

# Patrones de comportamiento: *Observer*

## ► Implementación

- En Java se cuenta con bibliotecas predefinidas para su implementación: `java.util.Observable` y la interface `java.util.Observer`

```
import java.util.Observable;
import java.util.Observer;

class Foro extends Observable { //el objeto que queremos observar
    private String msg;

    public String getMessage() {
        return msg;
    }

    public void ActualizaForo(String msg) {
        this.msg = msg;
        //cambiamos el estado
        setChanged();
        //notificamos a quien nos esté observando
        notifyObservers(msg);
    }
}
```

<http://examples.oreilly.com/learnjava2/CD-ROM/examples/ch10/MessageBoard.java>

# Patrones de comportamiento: *Observer*

```
class Detector implements Observer { //quien lo va a observer
    public void update(Observable o, Object arg) {
        //sobreescribimos el método, es la acción que se generará cuando se
        //detecte un cambio

        System.out.println("Nuevo mensaje: " + arg);
    }
}

public class PatronObserver{

    public static void main(String[] args) {
        Foro foro = new Foro();
        Detector d1 = new Detector();
        Detector d2 = new Detector();
        foro.addObserver(d1);
        foro.addObserver(d2);
        foro.ActualizaForo("Examen el viernes!!!");
    }
}
```



UNIVERSIDAD  
INTERNACIONAL  
DE LA RIOJA

**unir**

[www.unir.net](http://www.unir.net)