

Reutilización del Software

Patrones de Diseño

Introducción

- El diseño OO es difícil y el diseño de software orientado a objetos reutilizable lo es aún más.
- Los diseñadores expertos no resuelven los problemas desde sus principios; reutilizan soluciones que han funcionado en el pasado.
 - Se encuentran patrones de clases y objetos de comunicación recurrentes en muchos sistemas orientados a objetos.
 - Estos patrones resuelven problemas de diseño específicos y hacen el diseño flexible y reusable.

Definición de un patrón

Alexander(arquitecto/urbanista)

Cada patrón describe un problema que ocurre una y otra vez en nuestro entorno y describe también el núcleo de la solución al problema, de forma que puede utilizarse un millón de veces sin tener que hacer dos veces lo mismo.

Definición de un patrón de diseño

[Gamma]

Un patrón de diseño es una descripción de clases y objetos comunicándose entre sí adaptada para resolver un problema de diseño general en un contexto particular.

Introducción

- Es un tema importante en el desarrollo de software actual: permite capturar la experiencia
- Busca ayudar a la comunidad de desarrolladores de software a resolver problemas comunes, creando un cuerpo literario de base
 - Crea un lenguaje común para comunicar ideas y experiencia acerca de los problemas y sus soluciones
- El uso de patrones ayuda a obtener un software de calidad (reutilización y extensibilidad)

Elementos de un patrón

- **Nombre:** describe el problema de diseño.
- **El problema:** describe cuándo aplicar el patrón.
- **La solución:** describe los elementos que componen el diseño, sus relaciones, responsabilidades y colaboración.

Más información en...

- Desing Patterns. E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns*. Addison Wesley, 1995.
- Patterns Home Page: <http://hillside.net/patterns/>
- Thinking in patterns with java
<http://www.mindview.net/Books/TIPatterns/>

Clasificación de los patrones

- Según su propósito:
 - **De creación:** conciernen al proceso de creación de objetos.
 - **De estructura:** tratan la composición de clases y/o objetos.
 - **De comportamiento:** caracterizan las formas en las que interactúan y reparten responsabilidades las distintas clases u objetos.

Clasificación de los patrones

GoF (gang of Four) [Gamma]

Propósito Ámbito	Creación	Estructural	Comportamiento
Clase	✓ Factory Method	✓ Adapter	Interpreter Template Method
Objeto	Abstract Factory Builder Prototype ✓ Singleton	✓ Adapter Bridge ✓ Composite Decorator ✓ Facade Flyweight Proxy	Chain of Responsability Command ✓ Iterator Mediator Memento ✓ Observer State ✓ Strategy Visitor

Además: PATRONES DE DISEÑO FUNDAMENTALES

Patrones de diseño fundamentales

Son patrones que no aparecen la tabla definida por Gamma, pero se utilizan habitualmente:

- DELEGATION
- INTERFACE
- MARKER INTERFACE

Patrón DELEGATION

Utilidad:

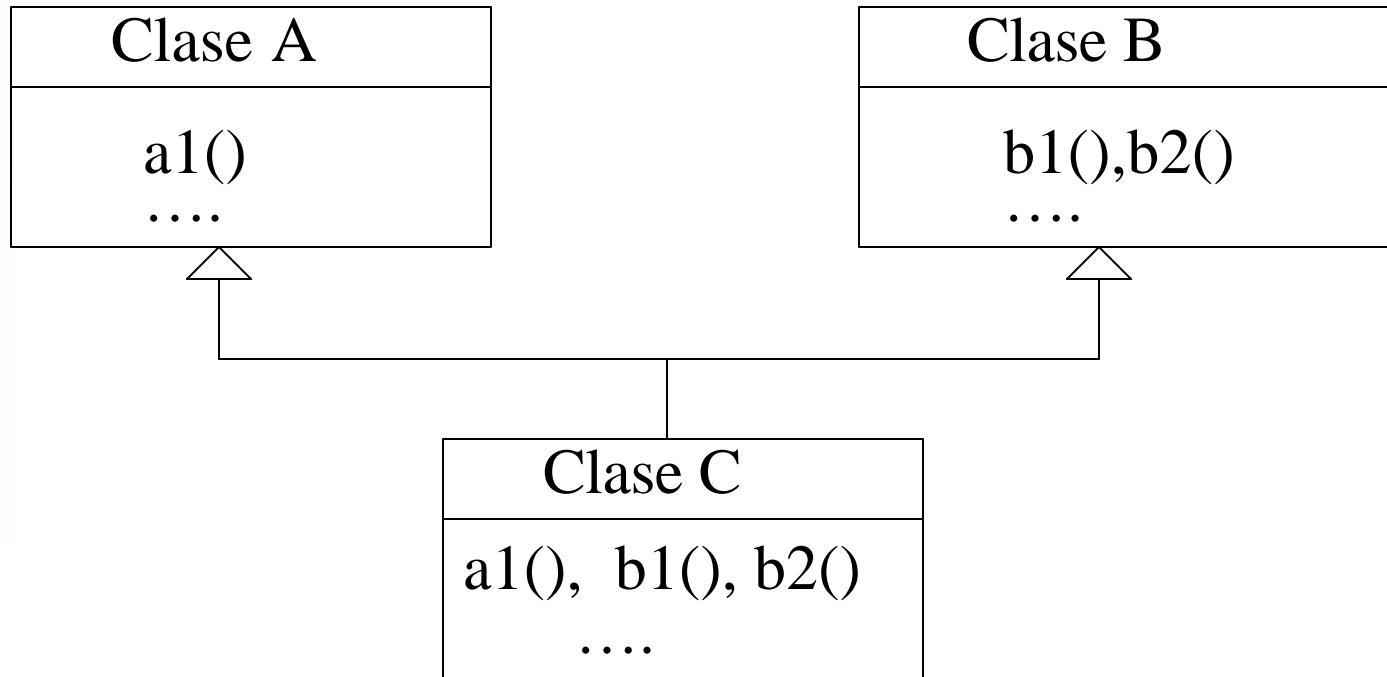
Cuando se quiere extender y reutilizar la funcionalidad de una clase SIN UTILIZAR LA HERENCIA

Ventajas:

- En vez de herencia múltiple
- Cuando una clase que hereda de otra quiere ocultar algunos de los métodos heredados
- Compartir código que NO se puede heredar

Patrón DELEGATION

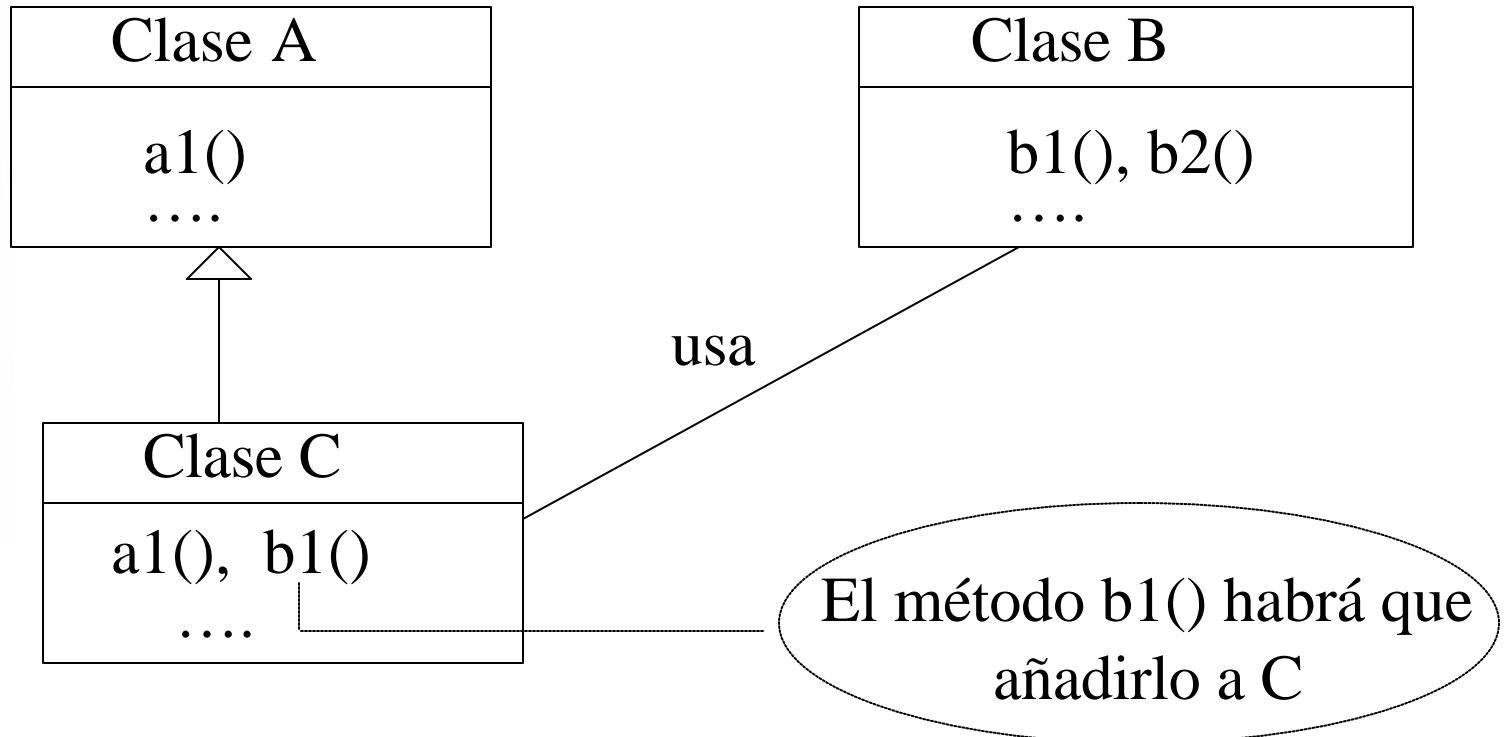
El problema



- El lenguaje utilizado NO PERMITE HERENCIA MÚLTIPLE
- La clase C no desea TODOS los métodos de B

Patrón DELEGATION

La solución



NO USAR HERENCIA

SINO LA RELACIÓN “USA”

Patrón DELEGATION

Implementación

```
class C extends A {  
    B objB;  
  
    C () { // En la constructora se puede crear obj. de B  
        objB=new B();  
    }  
  
    void b1( ) { objB.b1( );}  
  
    ....
```

Patrón INTERFACE

Utilidad y Ventajas

Utilidad

Definir un comportamiento independiente de donde vaya a ser utilizado

Ventajas

Desacople entre comportamiento y clase.
Realización de clases “Utilities”

Patrón INTERFACE

El problema

Utilities

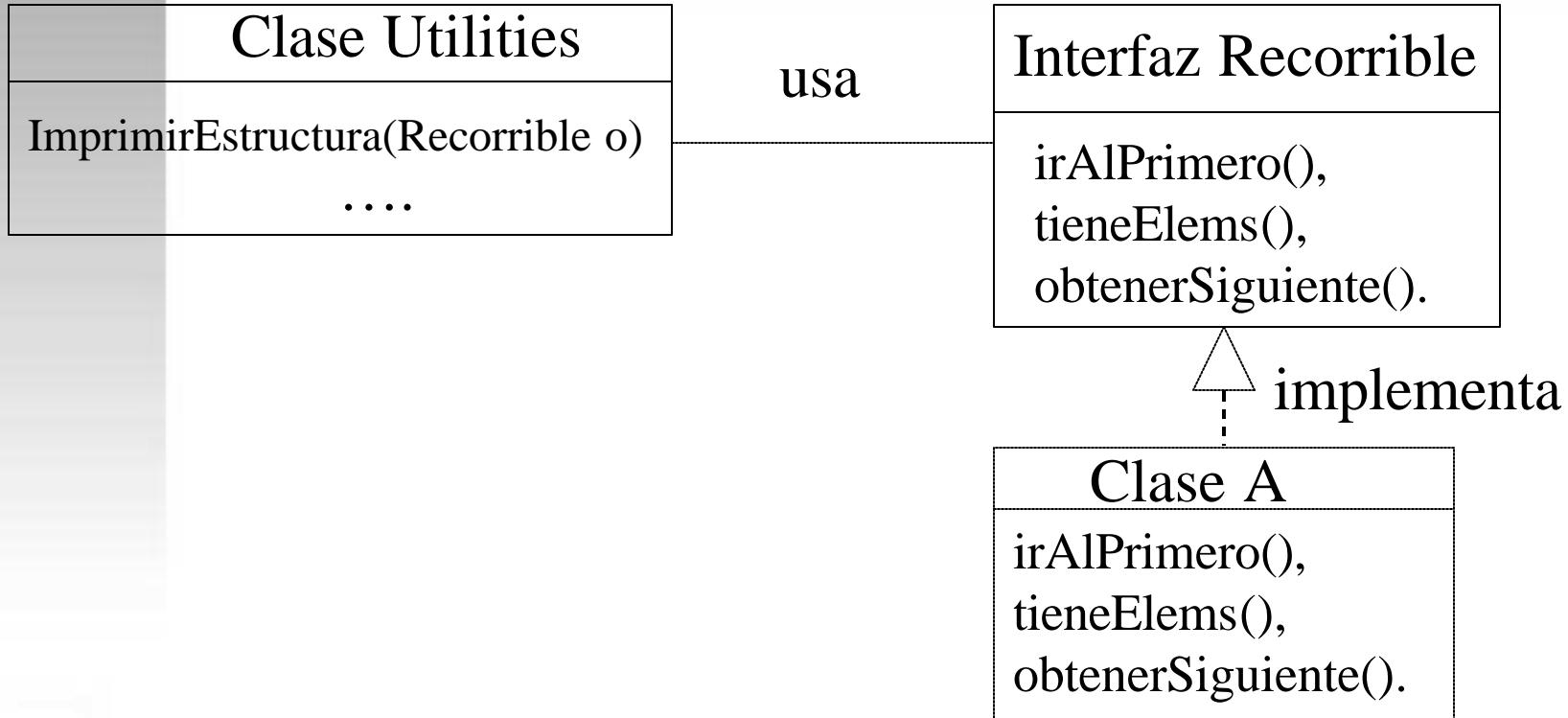
```
void imprimirEstructura (????? o) {  
    o.irAlPrimero();  
    while (o.tieneElems() );  
        imprimir (o.obtenerSiguiente() );  
}
```

Todos los objeto o, tienen que implementar:
irAlPrimero(), tieneElems(), obtenerSiguiente().

Problema: De qué tipo son los objetos o?

Patrón INTERFACE

La solución



Solución: Definir los parámetros de un tipo Interfaz.

Todas las clases(instancias) que quieran utilizar ese comportamiento deberán implementar dicho interfaz

Patrón INTERFACE

Implementación

```
class MiClase<T> implements Recorrible<T> {  
    Vector<T> v=new Vector<T>();  
  
    int pos;  
  
    void irAlPrimero() { pos=0; }  
  
    boolean tieneElems() { v.length < pos; }  
  
    T obtenerSiguiente() {  
  
        T o=v.elementAt(pos); pos++;  
  
        return o; }  
  
    }  
}
```

Patrón INTERFACE

Ejemplo

```
// Estructuras de datos, algoritmos y predicados JGL  
Array queue = new Array( );  
queue.add( ...);
```

```
Counting.countIf(queue, new MayorEdad());
```

Uso de
countIf

```
class MayorEdad implements UnaryPredicate {  
    public boolean execute (Object o1)  
    { ..... } }
```

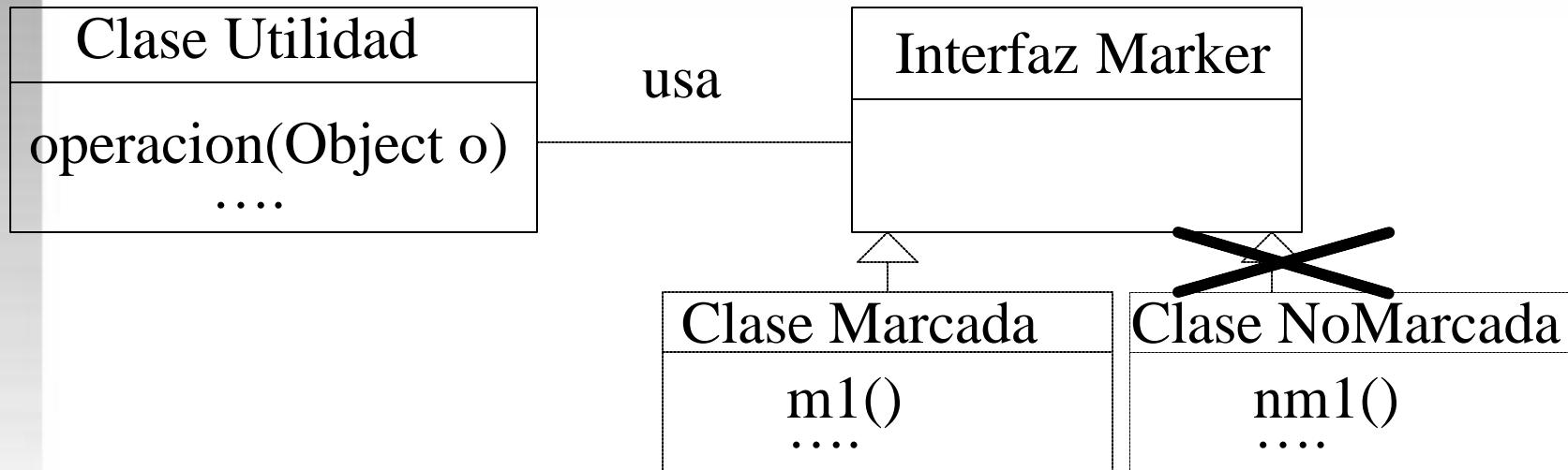
Definición: Counting.countIf(Container,UnaryPredicate);

Patrón MARKER INTERFACE

Utilidad y Ventajas

- Utilidad
 - Sirve para indicar atributos semánticos de una clase.
 - Ventajas:
 - Se puede preguntar si un objeto pertenece a una clase de un determinado tipo o no.
 - Habitualmente se utiliza en clases de utilidades que tienen que determinar algo sobre objetos sin asumir que son instancias de una determinada clase o no.
- 20

Patrón MARKER INTERFACE



Dentro del código del método `operacion(Object o)` de la clase Utilidad podemos preguntar si el objeto o es de una clase Marker:

```
if (o instanceof Marker) {...}
else {...}
```

Patrón MARKER INTERFACE

Ejemplo

En Java hay clases “serializables”, “cloneables”, etc. Para ello, basta con que implementen las interfaces Serializable, Cloneable, etc.

Clasificación de los patrones

GoF (gang of Four) [Gamma]

Propósito Ámbito	Creación	Estructural	Comportamiento
Clase	✓ Factory Method	✓ Adapter	Interpreter Template Method
Objeto	Abstract Factory Builder Prototype ✓ Singleton	✓ Adapter Bridge ✓ Composite Decorator ✓ Facade Flyweight Proxy	Chain of Responsability Command ✓ Iterator Mediator Memento ✓ Observer State ✓ Strategy Visitor

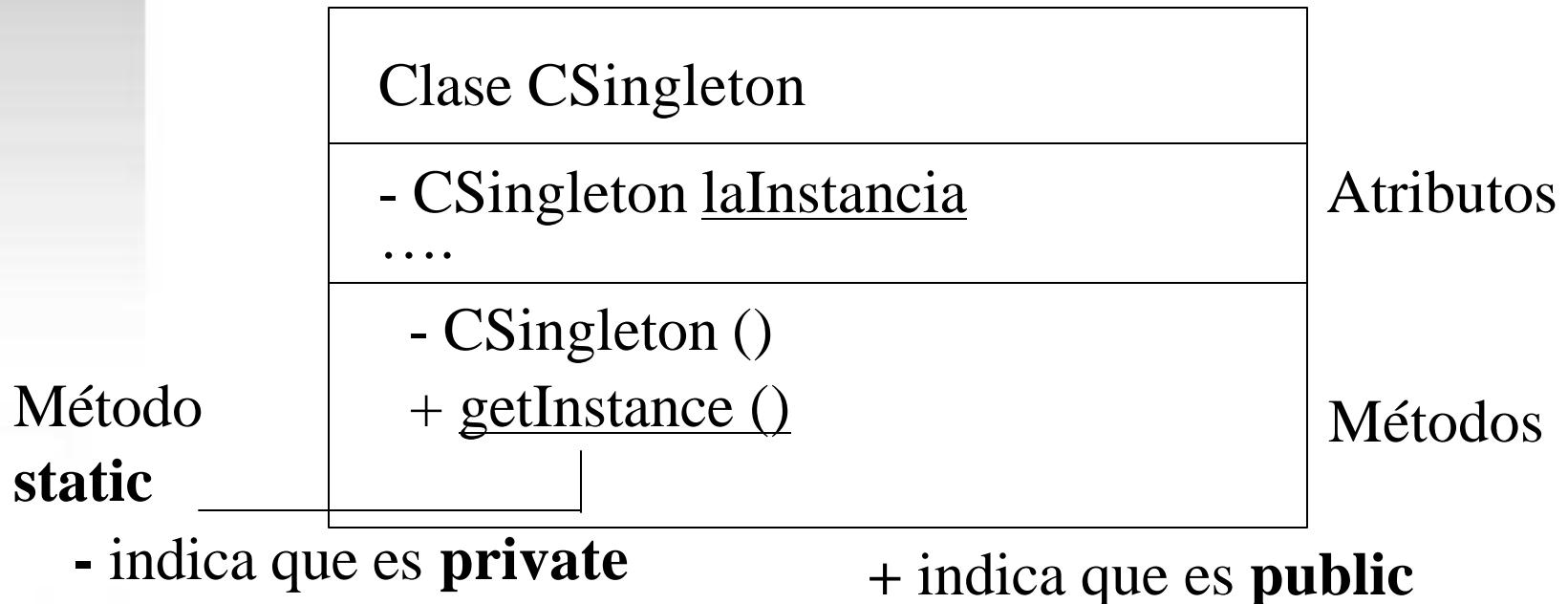
Patrón SINGLETON

- **Utilidad**
 - Asegurar que una clase tiene una sola instancia y proporcionar un punto de acceso global a ella
- **Ventajas**
 - Es necesario cuando hay clases que tienen que gestionar de manera centralizada un recurso
 - Una variable global no garantiza que sólo se instancie una vez

Patrón SINGLETON

La solución

- El constructor de la clase DEBE SER PRIVADO
- Se proporciona un método ESTÁTICO en la clase que devuelve LA ÚNICA INSTANCIA DE LA CLASE: `getInstance()`



Patrón SINGLETON

Implementación

```
public class CSingleton {  
    private static CSingleton laInstancia = new CSingleton();  
    private CSingleton() {}  
    public static CSingleton getInstance() {  
        return laInstancia;  
    }  
    .....  
}
```

Patrón SINGLETON

Inconvenientes

Se podría obtener una nueva instancia usando el esquema anterior:

```
public class MiCSingleton extends CSingleton  
    implements Cloneable {}  
  
MiCSingleton mcs = new MiCSingleton();  
  
MiCSingleton mcsCopia = mcs.clone();
```

Para solucionarlo: definir CSingleton como final

```
public final class CSingleton {...}
```

```
public final class Facultad implements Serializable {  
    private Vector listaProfesores;  
    private Vector listaEstudiantes;  
    private Vector listaAsigs;  
    private Vector listaMatrs;  
private static Facultad laFacultad=new Facultad();  
private Facultad() // EL CONSTRUCTOR ES PRIVADO  
{ listaProfesores = new Vector(); // Sólo hay UNA instancia  
listaEstudiantes= new Vector(); // y se guarda en laFacultad  
listaAsigs = new Vector();  
listaMatrs = new Vector(); }  
public static Facultad getInstance() {return laFacultad;}  
public Vector obtListaProfesores()  
    {return listaProfesores; }  
public void anadirProfesor(Profesor p)  
    {listaProfesores.addElement(p); }
```

EJEMPLO DE
PATRÓN
SINGLETON

Patrón FACTORY METHOD

Utilidad

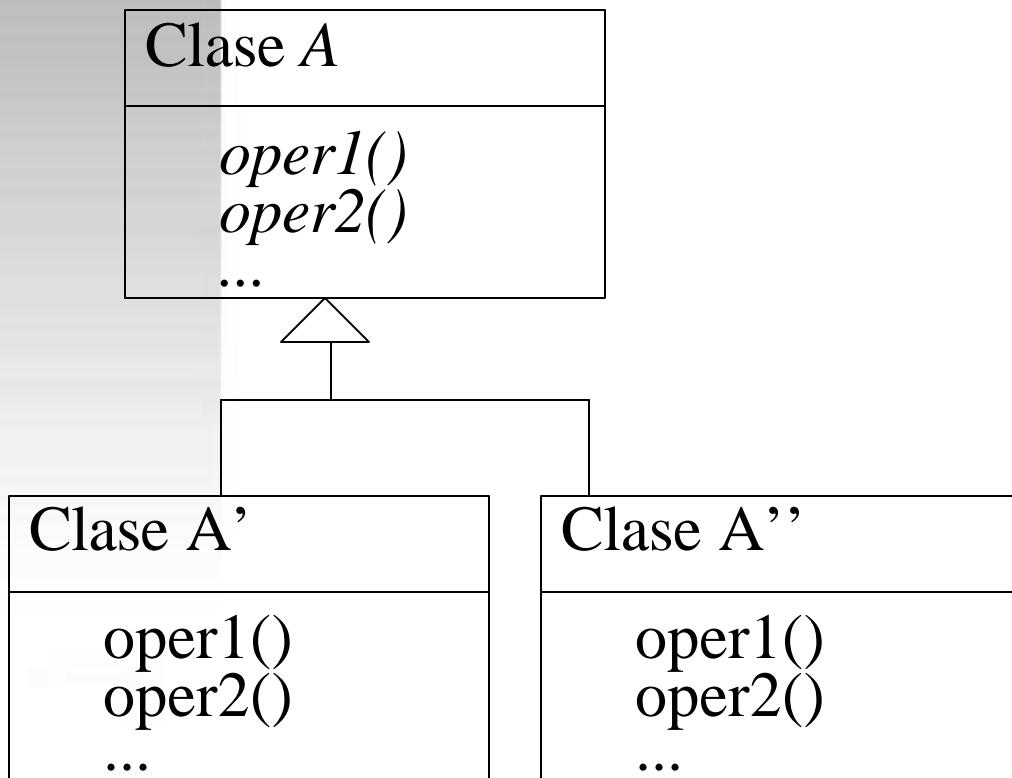
Separar la clase que crea los objetos, de la jerarquía de objetos a instanciar

Ventajas

- Centralización de la creación de objetos
- Facilita la escalabilidad del sistema
- El usuario se abstrae de la instancia a crear

Patrón FACTORY METHOD

El problema



Programa1

```
If (tipo==1)
    create A'
else
    create A''
```

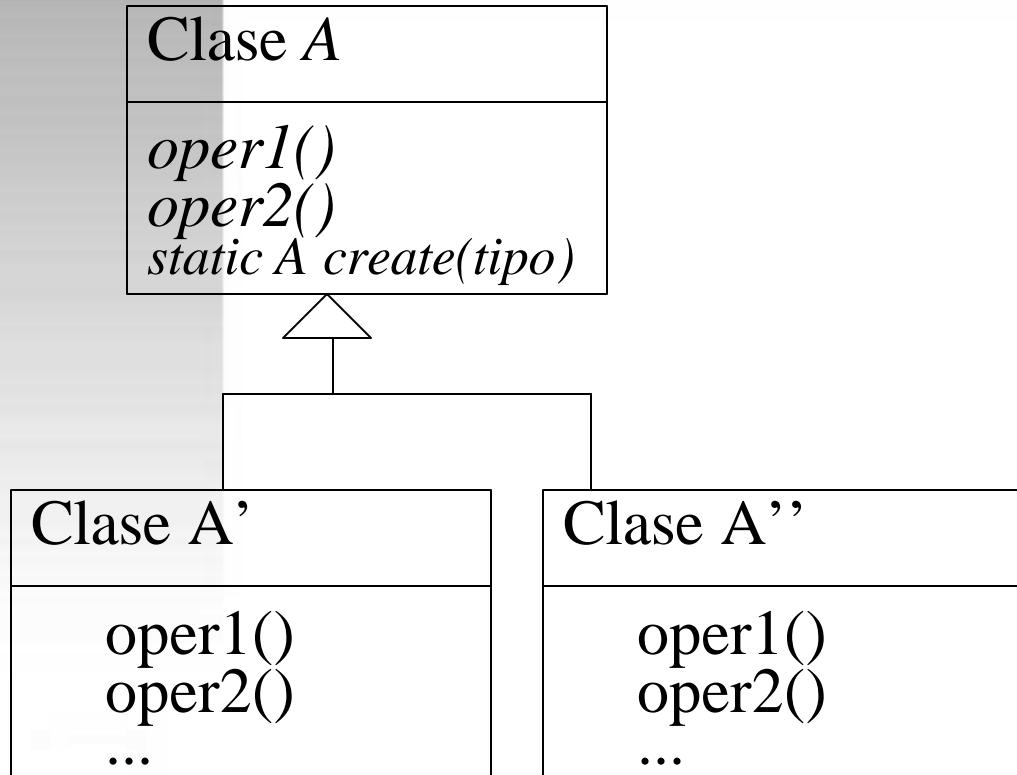
Programa2

```
If (tipo==1)
    create A'
else
    create A''
```

Problema: Qué sucede si queremos añadir A'''?

Patrón FACTORY METHOD

Una primera solución



Programa1

```
A.create(1);
```

Programa2

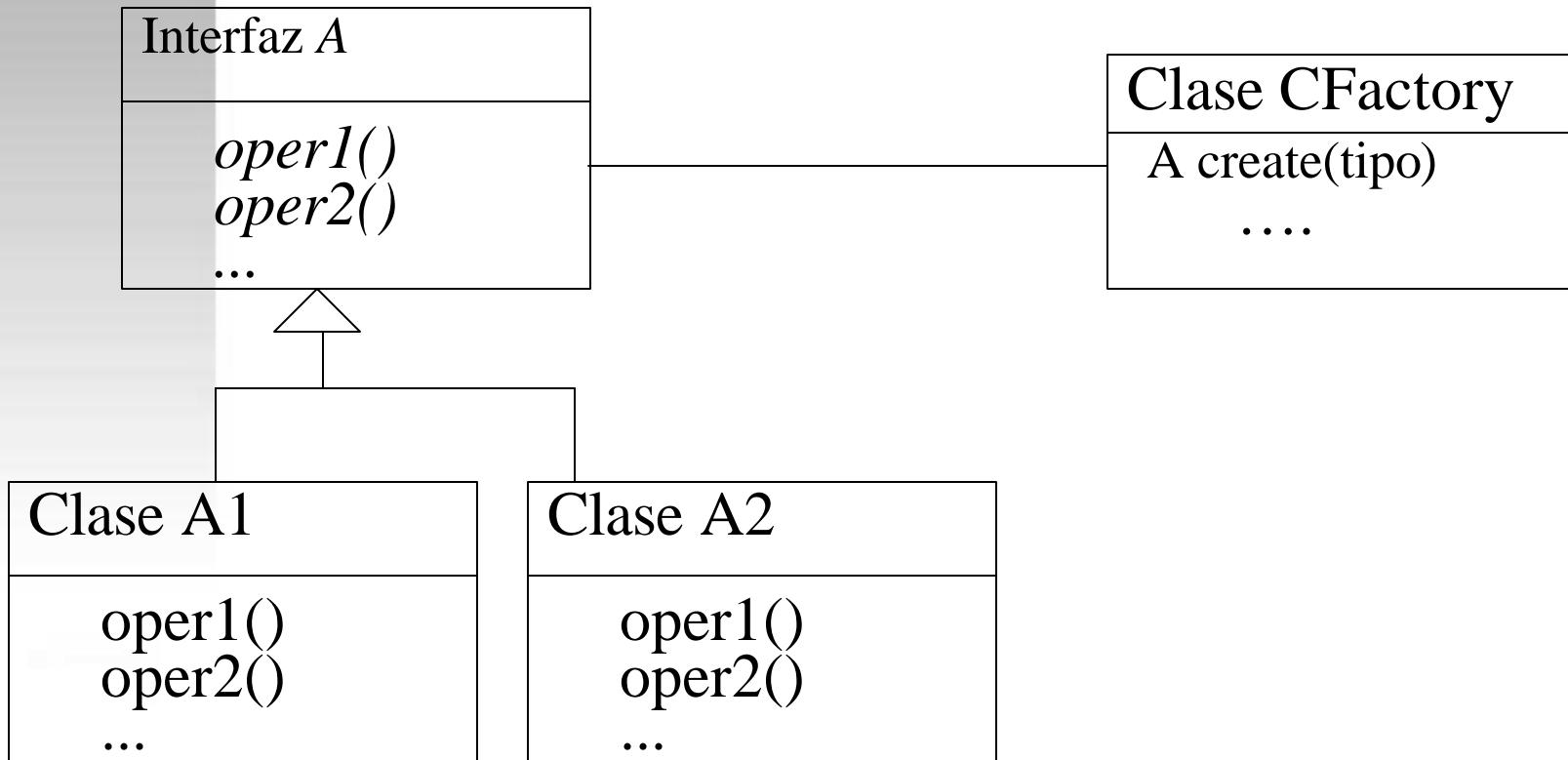
```
A.create(2)
```

Solución: Hay que recomilar todas las clases que heredan de A

Puede que no tengamos acceso al código de A

Patrón FACTORY METHOD

La solución final



Solución: Separar el creador de las instancias de la propia clase
las instancias se crean en una clase CFactory

Patrón FACTORY METHOD

```
class Aplicación {  
    ...  
    A newA(...) {  
        A miA;  
        CFactory fact= new CFactory();  
        ...  
        miA = fact.create("TIPO 2");  
        ...  
        return miA;    }  
}
```

- No se crean las instancias directamente en Aplicación
- Si se quisiera añadir un nuevo A A''', NO NECESARIAMENTE habría que modificar la clase Aplicación. Los tipos que CFactory puede crear los puede devolver un método y Aplicación trabajar con él.

Patrón FACTORY METHOD

```
class CFactory {  
    ...  
    A create(String tipo) {  
        if ("TIPO 1".equals(tipo))  
            return new A1();  
        else if ("TIPO 2".equals(tipo))  
            return new A2(); .... }  
    }
```

- Se crean las instancias de los Aes en CFactory, no en la clase Aplicación
- Se pueden añadir nuevos Aes sin modificar Aplicación. Basta con añadir la clase A3 y modificar CFactory (añadir el tipo 34 correspondiente a dicho A: "TIPOX")

Clasificación de los patrones

GoF (gang of Four) [Gamma]

Propósito Ámbito	Creación	Estructural	Comportamiento
Clase	✓ Factory Method	✓ Adapter	Interpreter Template Method
Objeto	Abstract Factory Builder Prototype ✓ Singleton	✓ Adapter Bridge ✓ Composite Decorator ✓ Facade Flyweight Proxy	Chain of Responsability Command ✓ Iterator Mediator Memento ✓ Observer State ✓ Strategy Visitor

Patrón ADAPTER

Intención

Convertir la interfaz de una clase en otra interfaz esperada por los clientes.

Permite que clases con interfaces incompatibles se comuniquen

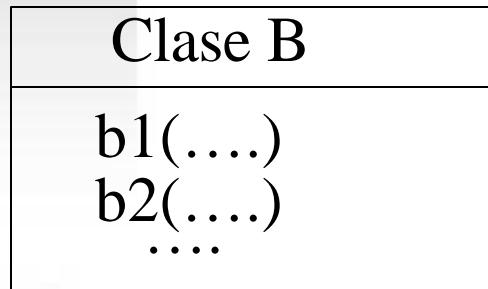
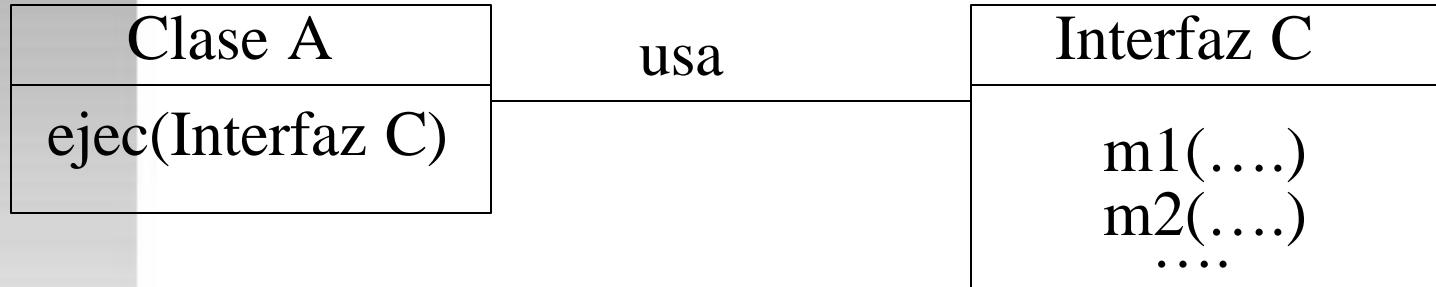
Ventajas

Se quiere utilizar una clase ya existente y su interfaz no se corresponde con la interfaz que se necesita

Se quiere envolver código no orientado a objeto con forma de clase

Patrón ADAPTER

El problema



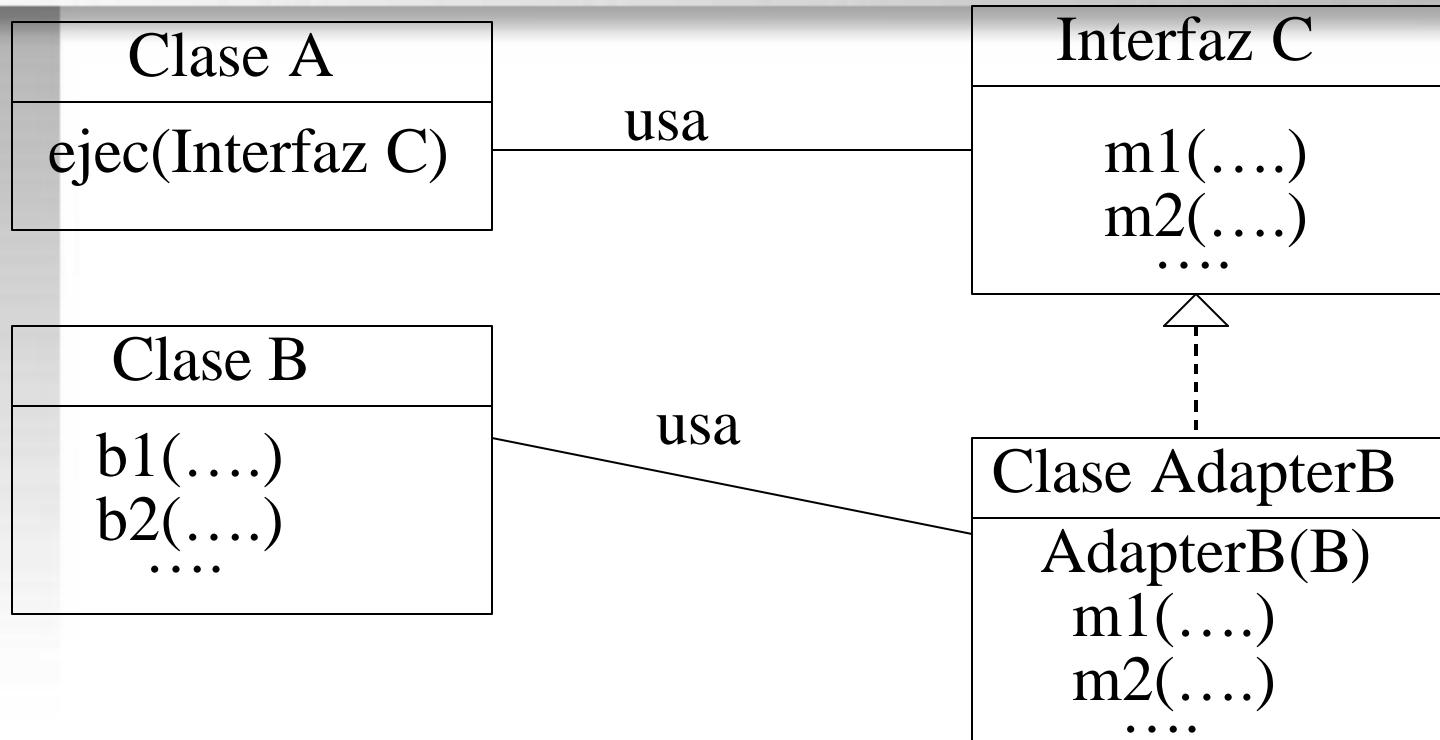
Se desea utilizar la clase A (el método ejec) utilizando como entrada un objeto de la clase B

`objetoDeA.ejec(objetoDeB)`

Pero no se puede, ya que la clase B no implementa la interfaz C

Patrón ADAPTER

La solución



Solución: construir una clase Adaptadora de B que implemente la interfaz C. Al implementarla, usa un objeto de B y sus métodos

Para utilizar la clase A:

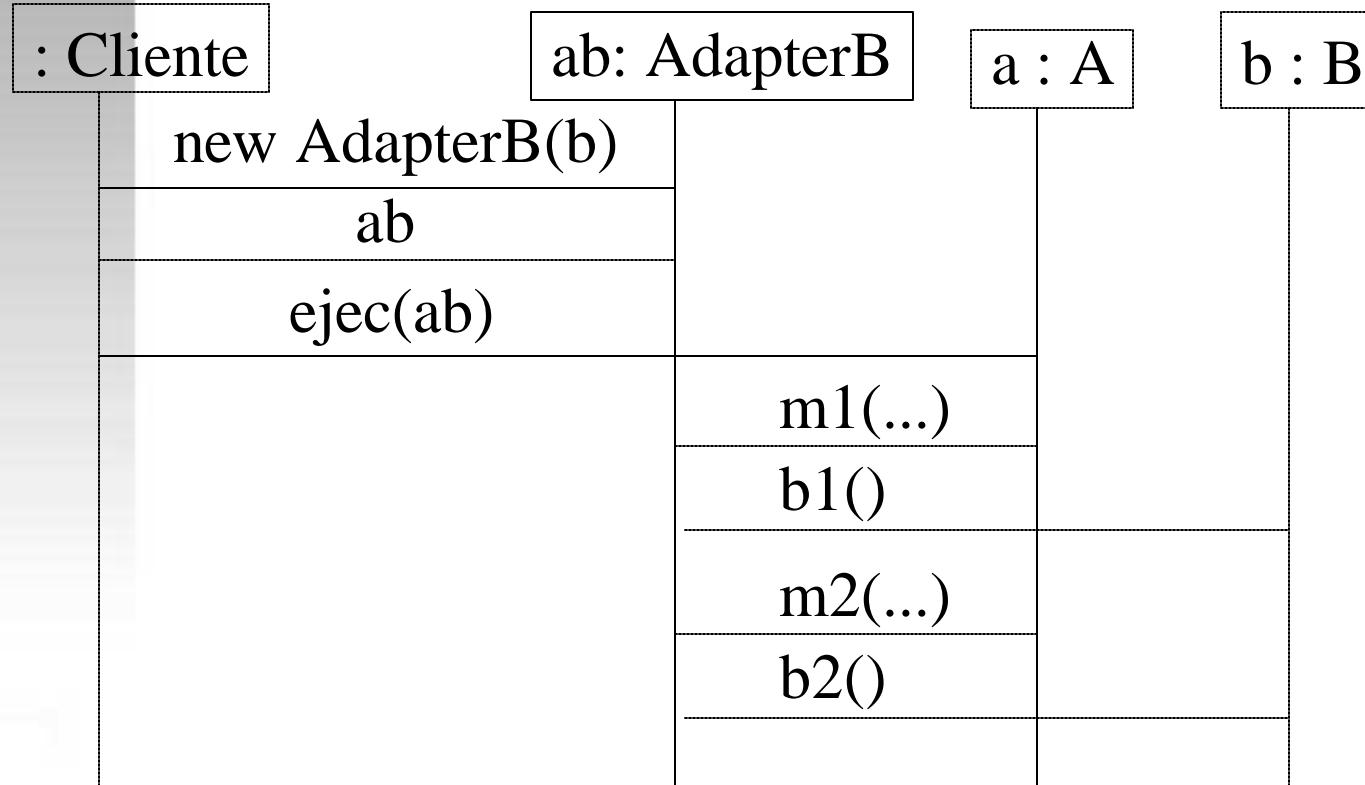
```
objetoDeAdapterB = NEW AdapterB(objetoDeB)
```

38

```
objetoDeA.ejec(objetoDeAdapterB)
```

Patrón ADAPTER

Diagrama de interacción



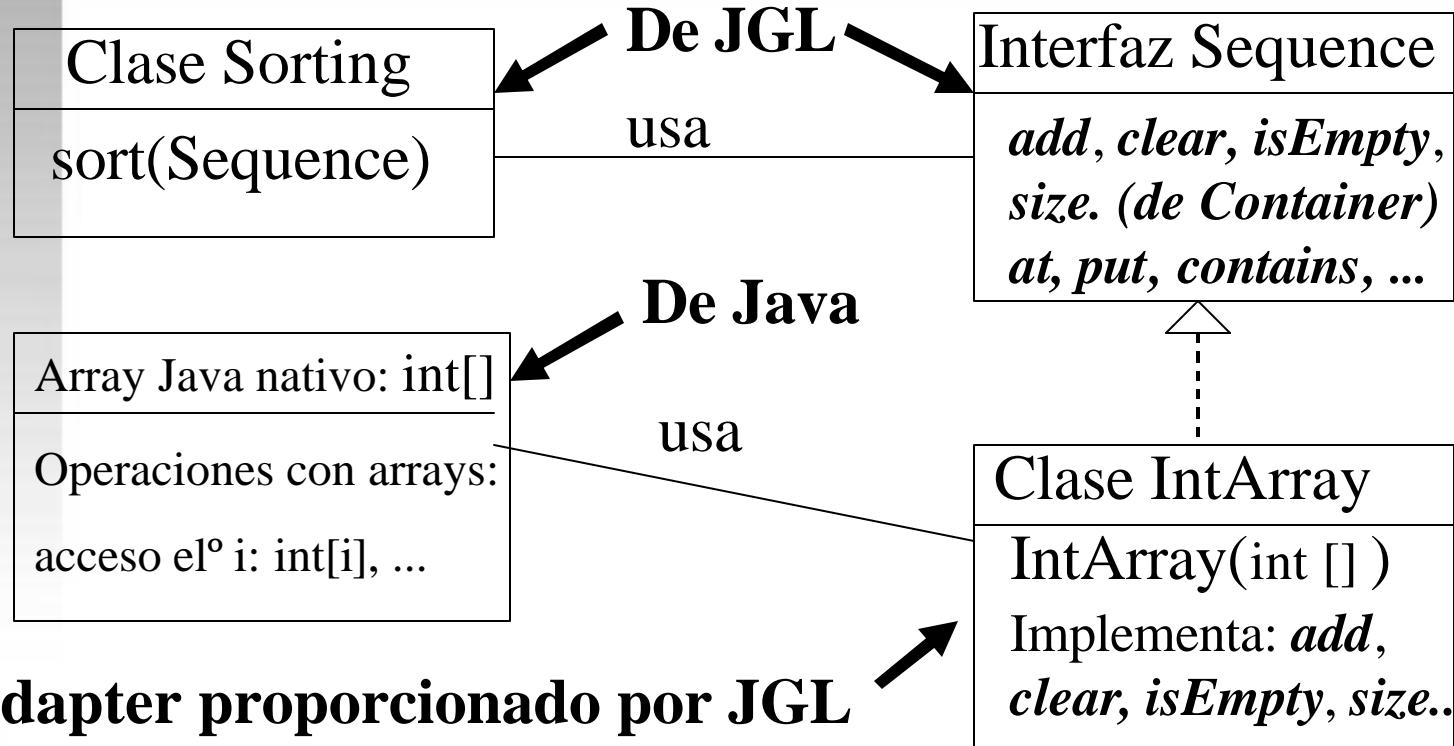
El cliente quiere usar el objeto a (para que ejecute el método ejec) con un objeto de b. Para ello necesita crear un objeto adaptador que encapsule b

Patrón ADAPTER

- Convierte la interfaz de una clase en otra interfaz que los clientes esperan. Permite que clases con interfaces incompatibles puedan ser utilizadas conjuntamente.
 - La interfaz de la clase B (los métodos b1(),b2()...) los convierte/adapta a los métodos esperados por la clase A (métodos m1(),m2(),...)

Patrón ADAPTER

Ejemplo de uso en JGL



OBJETIVO: PODER USAR LOS ALGORITMOS DE JGL CON `int []`

```
int ints[] = { 3, -1, 2, -3, 4 };
```

```
IntArray intArray = new IntArray( ints );
```

```
Sorting.sort( intArray );
```

Patrón ADAPTER

Ejemplo I

```
class WhatIUse {  
    public void op(WhatIWant wiw)  
    { wiw.f(); } }
```

```
class WhatIHave {  
    public void g() {}  
    public void h() {} }
```

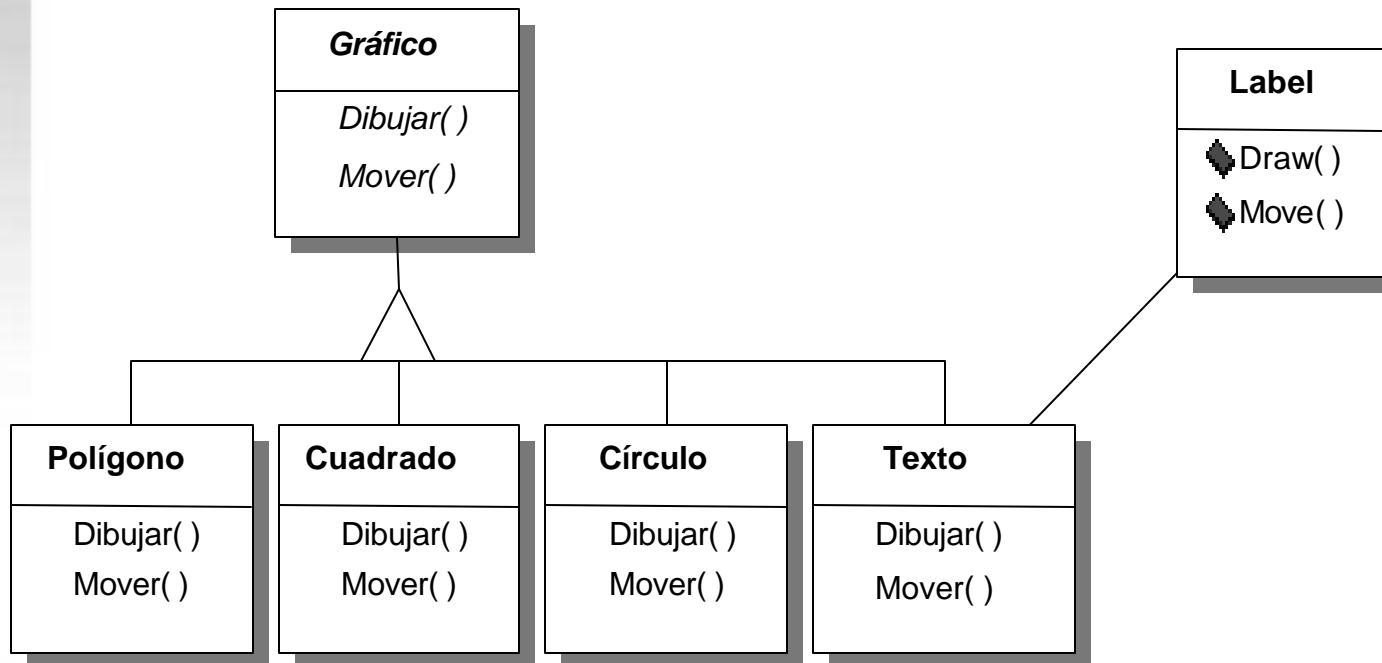
```
interface WhatIWant  
{ void f(); }
```

```
class Adapter implements WhatIWant {  
    WhatIHave whatIHave;  
    public Adapter(WhatIHave wiH) {  
        whatIHave = wiH; }  
    public void f() {  
        // Implementa usando los  
        // métodos de WhatIHave:  
        whatIHave.g();  
        whatIHave.h(); } }
```

Patrón ADAPTER

Ejemplo II

- ◆ Extender un editor gráfico con clases de otro toolkit



Patrón COMPOSITE

Intención

Componer objetos en jerarquías *todo-parte* y permitir a los clientes tratar objetos simples y compuestos de manera uniforme

Ventajas

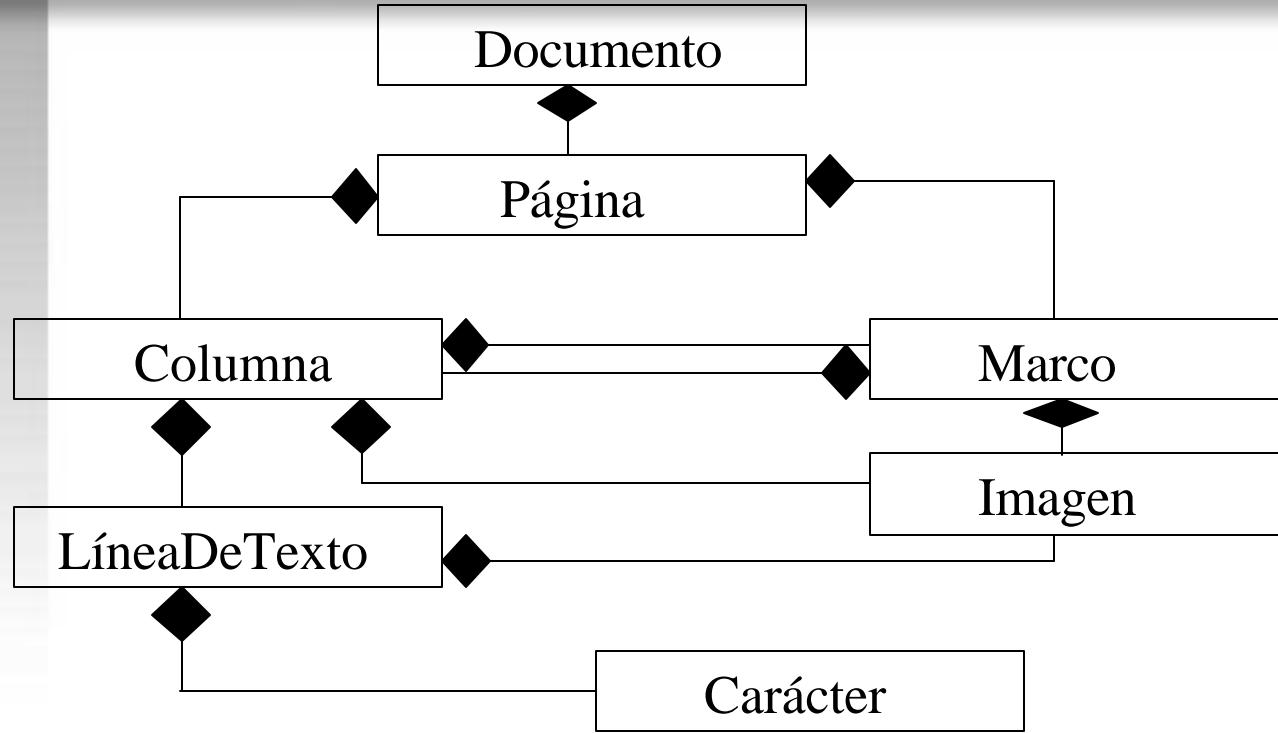
- ◆ Permite tratamiento uniforme de objetos simples y complejos así como composiciones recursivas
- ◆ Simplifica el código de los clientes, que sólo usan una interfaz
- ◆ Facilita añadir nuevos componentes sin afectar a los clientes

Inconvenientes

- ◆ Es difícil restringir los tipos de los hijos
- ◆ Las operaciones de gestión de hijos en los objetos simples pueden presentar problemas: seguridad frente a flexibilidad

Patrón COMPOSITE

El problema: La escalabilidad



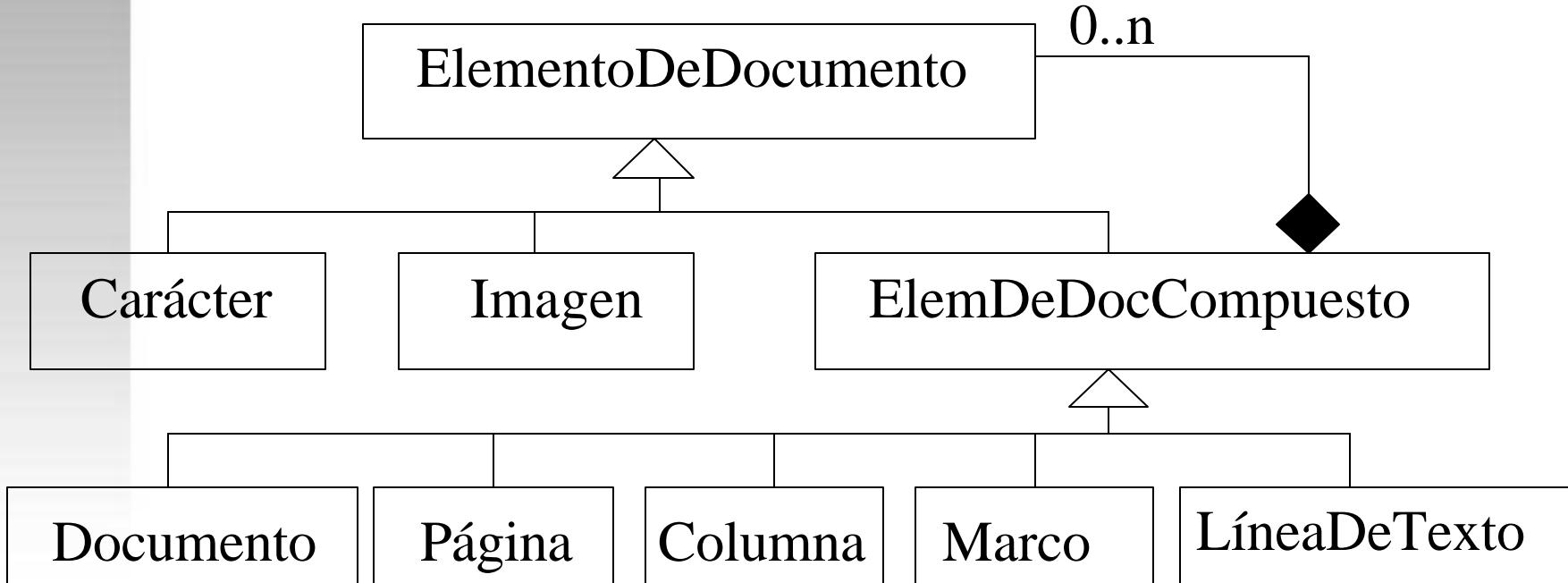
Un documento está formado por varias páginas, las cuales están formadas por columnas que contienen líneas de texto, formadas por caracteres.

Las columnas y páginas pueden contener marcos. Los marcos pueden contener columnas.

Las columnas, marcos y líneas de texto pueden contener imágenes.

Patrón COMPOSITE

La solución



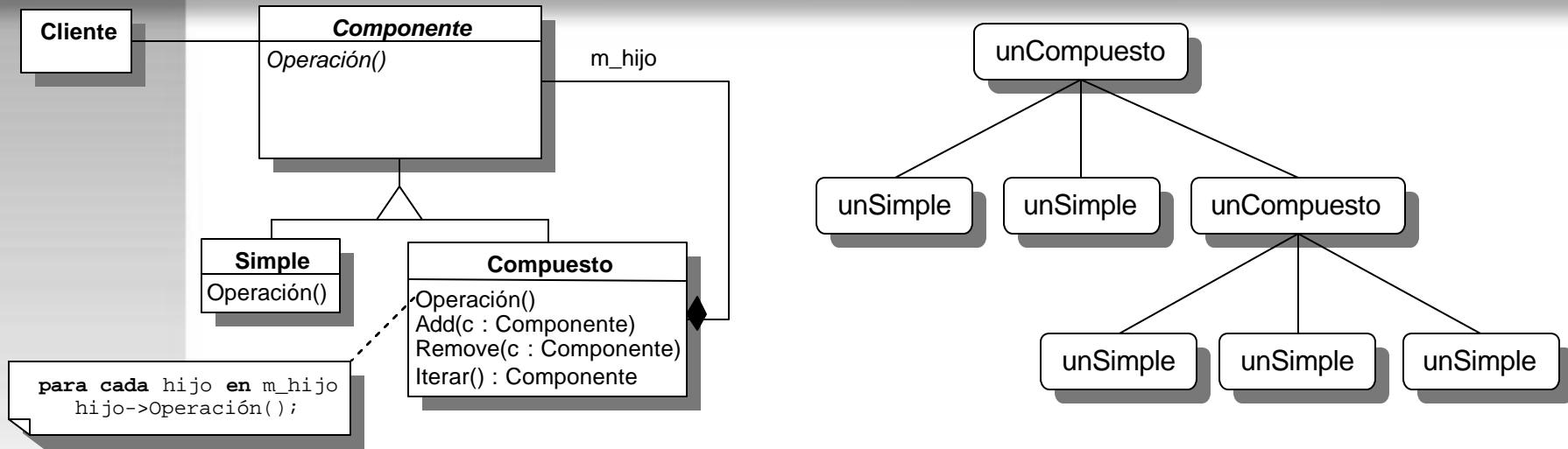
Un documento está formado por varias páginas, las cuales están formadas por columnas que contienen líneas de texto, formadas por caracteres.

Las columnas y páginas pueden contener marcos. Los marcos pueden contener columnas.

Las columnas, marcos y líneas de texto pueden contener imágenes.

Patrón COMPOSITE

La solución



◆ Participantes

- ◆ **Componente**: declara una clase abstracta para la composición de objetos,
- ◆ **Simple**: representa los objetos de la composición que no tienen hijos e implementa sus operaciones
- ◆ **Compuesto**: implementa las operaciones para los componentes con hijos y almacena a los hijos
- ◆ **Cliente**: utiliza objetos de la composición mediante la interfaz de Componente

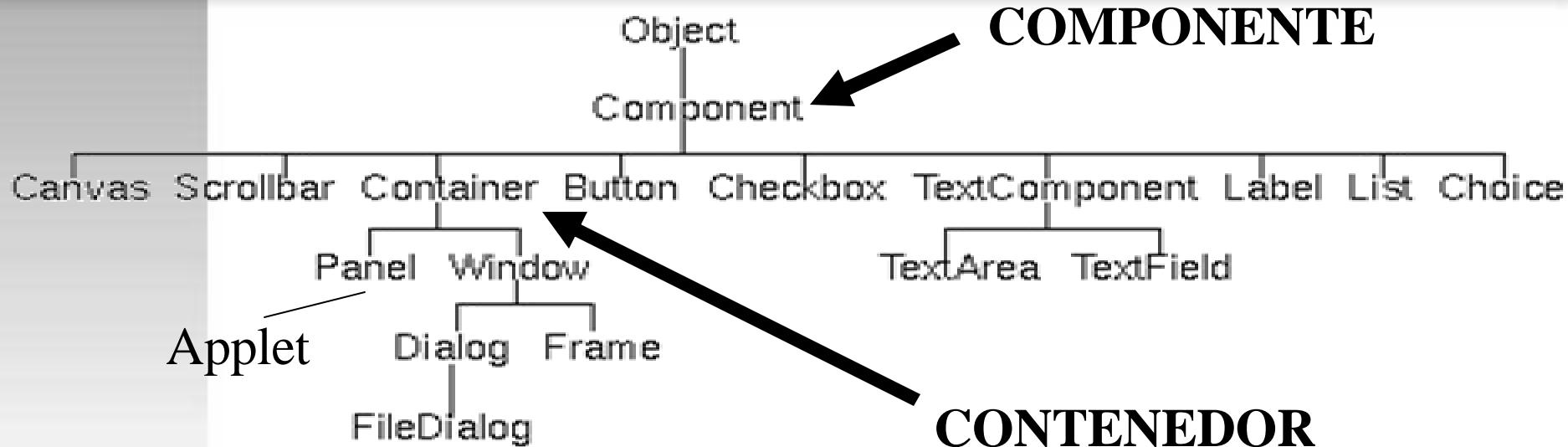
Patrón COMPOSITE

Ejemplo: La jerarquía de clases de AWT

- Sirve para diseñar clases que agrupen a objetos complejos, los cuales a su vez están formados por objetos complejos y/o simples
- La jerarquía de clases AWT se ha diseñado según el patrón COMPOSITE

Patrón COMPOSITE

Ejemplo: La jerarquía de clases de AWT



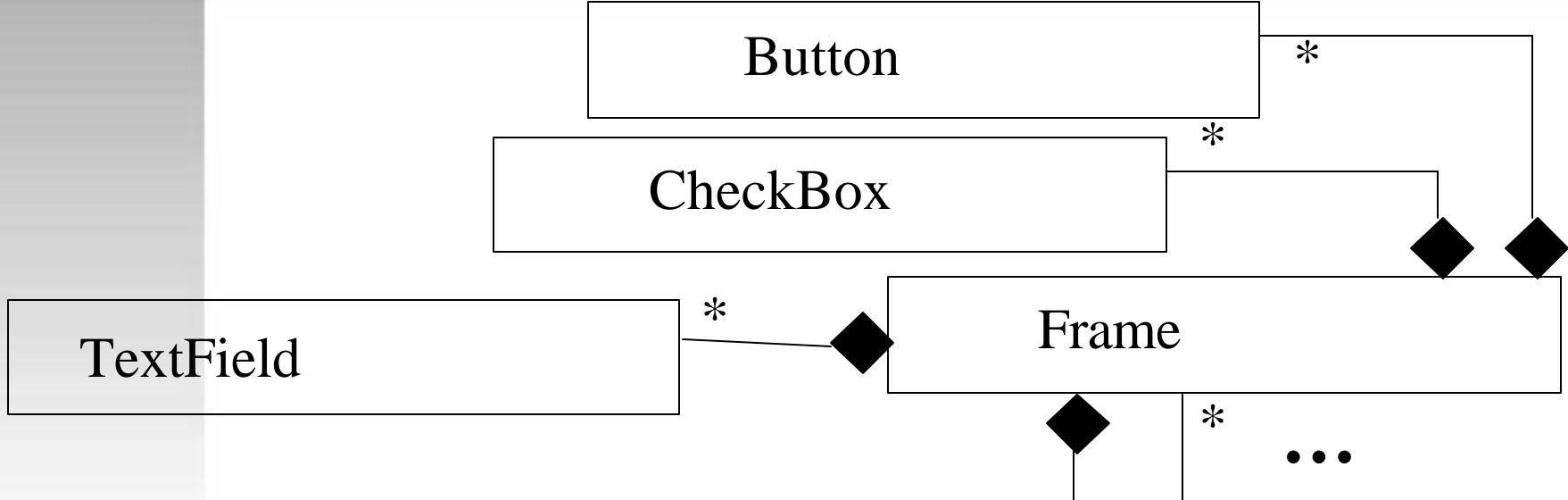
Las posibilidades son infinitas...

- Una ventana formada por 2 cajas de texto, 2 campos de texto, 3 botones, 1 panel que contenga 5 casillas de validación y una lista desplegable.
- Una ventana formada por 2 etiquetas, 2 campos de texto y un botón
- ...

Además, añadir nuevos tipos de contenedores y de componentes no sería muy costoso (estaríamos ante una solución extensible)

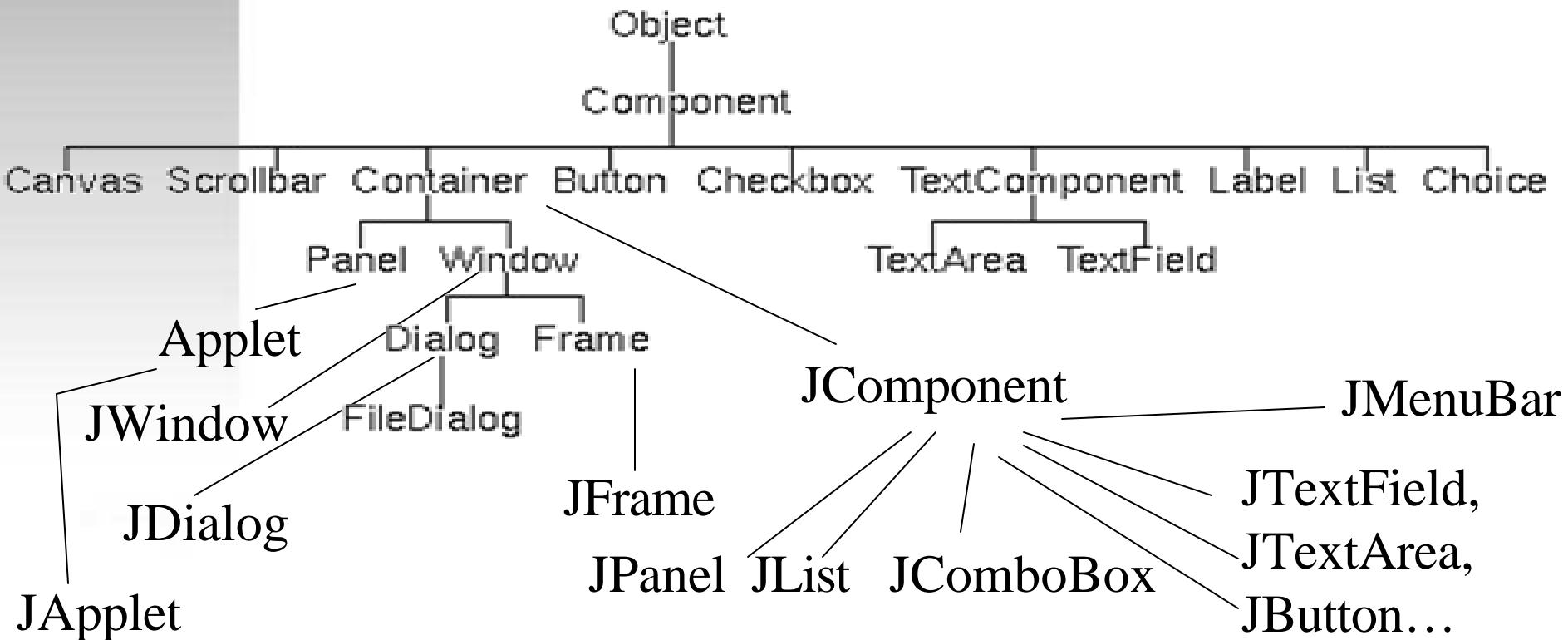
Patrón COMPOSITE

Un diseño francamente malo ...



- Se necesitarían métodos addButton, addCheckBox, addTextField, addFrame en la clase Frame (y los correspondientes a todos los que faltan)
- Además el diagrama está muy incompleto, ya que un Button puede ser componente de un Panel, Dialog, ...
- Si se quisiera añadir un nuevo componente XXX, habría que cambiar la clase Frame y añadir el método addXXX (nada extensible)

Componentes principales de Swing



Patrón FACADE

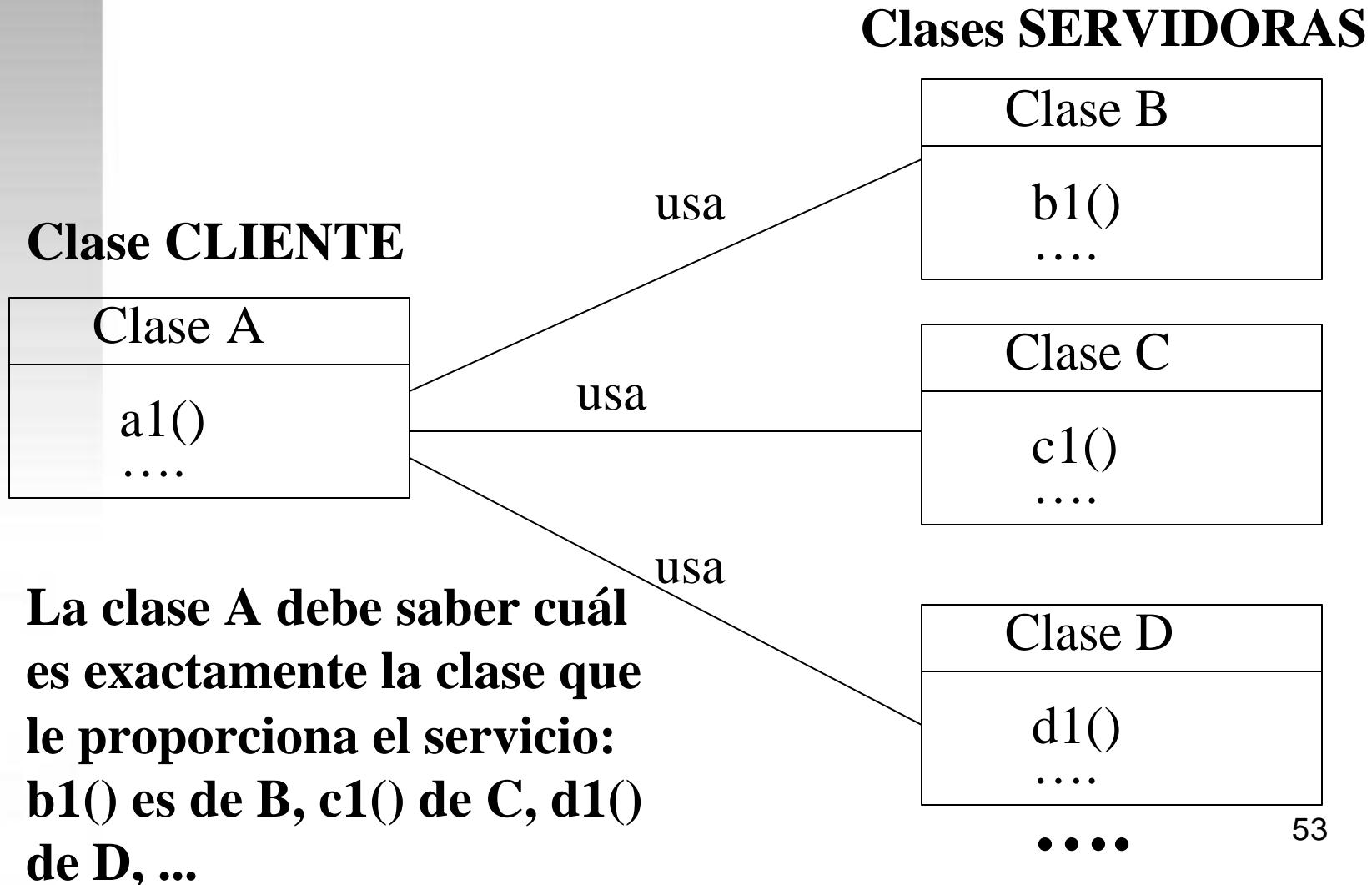
Intención: El patrón FACADE simplifica el acceso a un conjunto de clases proporcionando una única clase que todos utilizan para comunicarse con dicho conjunto de clases.

Ventajas

- Los clientes no necesitan conocer las clases que hay tras la clase FACADE
- Se pueden cambiar las clases “ocultadas” sin necesidad de cambiar los clientes. Sólo hay que realizar los cambios necesarios en FACADE

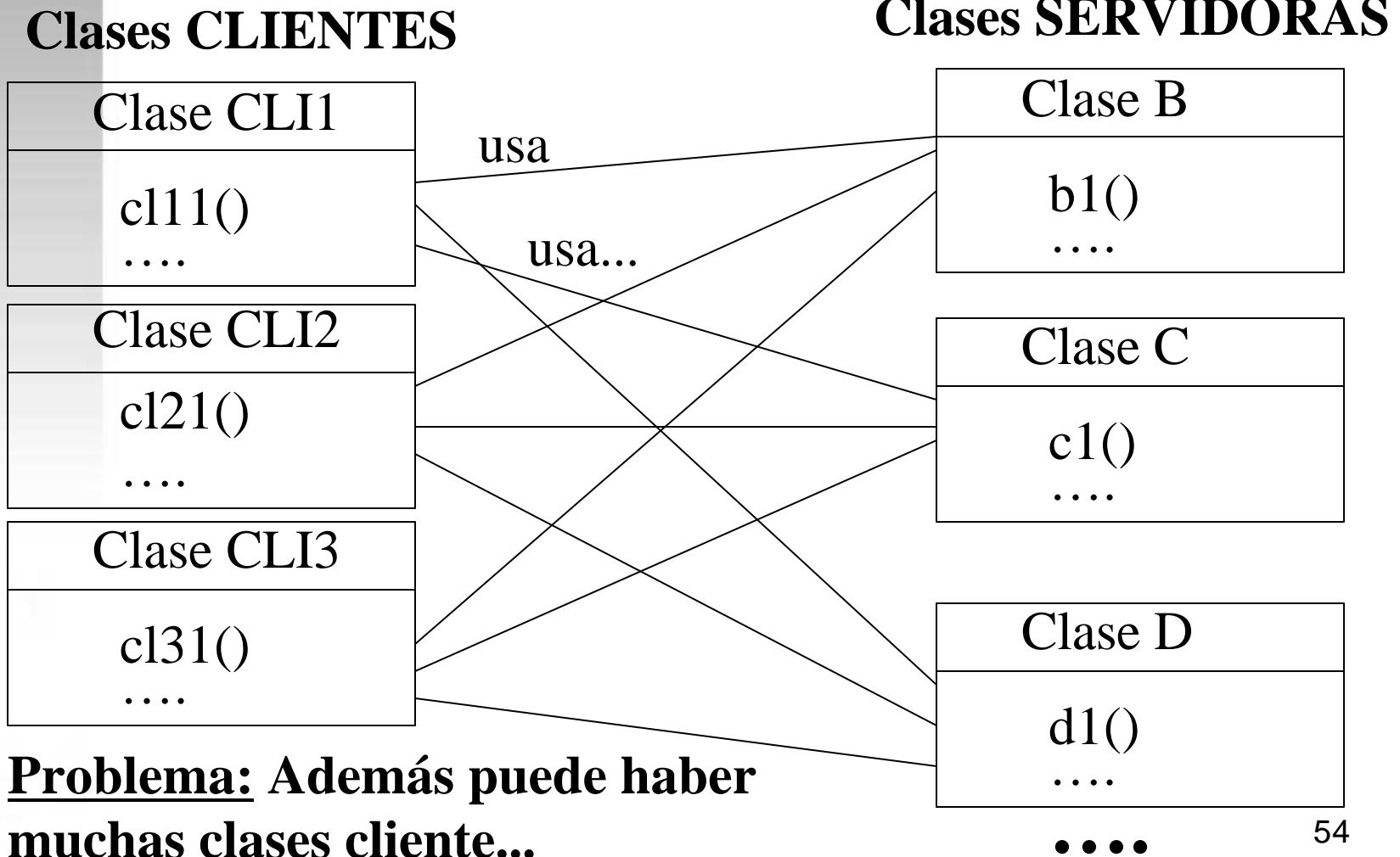
Patrón FACADE

El problema



Patrón FACADE

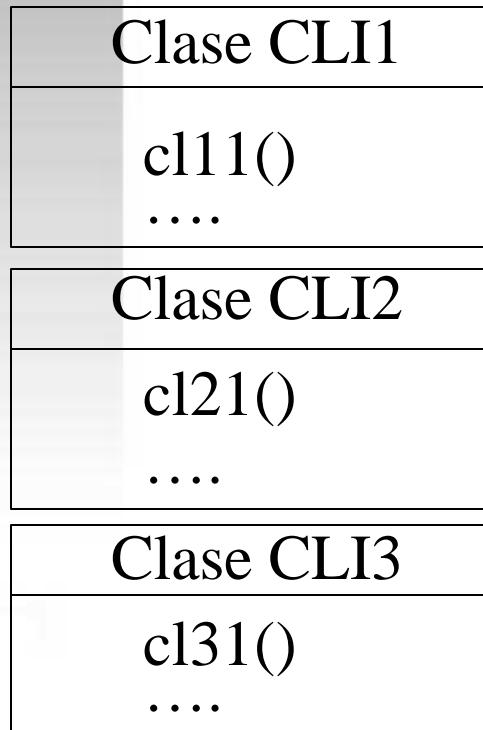
El problema



Patrón FACADE

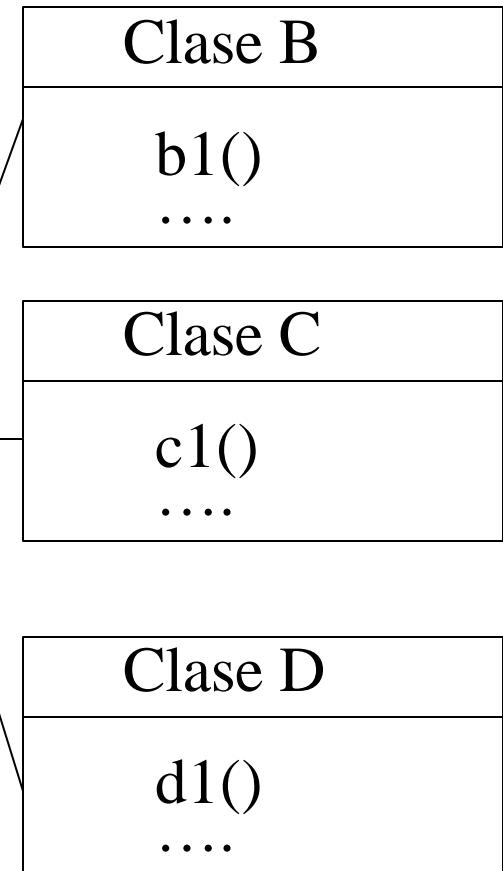
La solución

Clases CLIENTES



```
public class Facade {  
    B objB = new B();  
    C objC = new C();...  
    void b1() { objB.b1();}..}
```

Clases SERVIDORAS

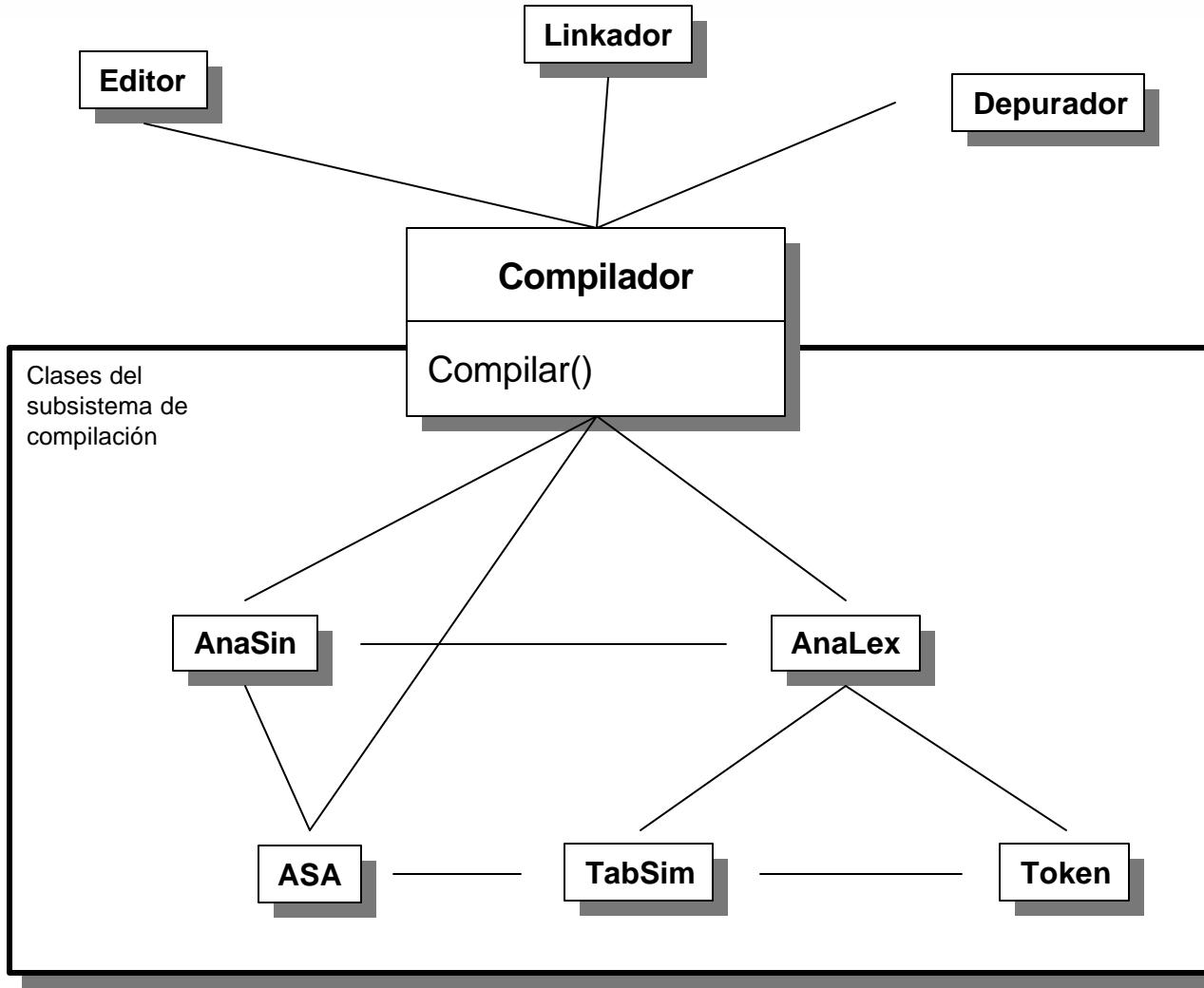


Solución: Proporcionar una clase que implemente todos los servicios (b1(),...). Los clientes sólo usarán dicha clase.

....

Patrón FACADE

Ejemplo: Estructurar un entorno de programación



Clasificación de los patrones

GoF (gang of Four) [Gamma]

Propósito Ámbito	Creación	Estructural	Comportamiento
Clase	✓ Factory Method	✓ Adapter	Interpreter Template Method
Objeto	Abstract Factory Builder Prototype ✓ Singleton	✓ Adapter Bridge ✓ Composite Decorator ✓ Facade Flyweight Proxy	Chain of Responsability Command ✓ Iterator Mediator Memento ✓ Observer State ✓ Strategy Visitor

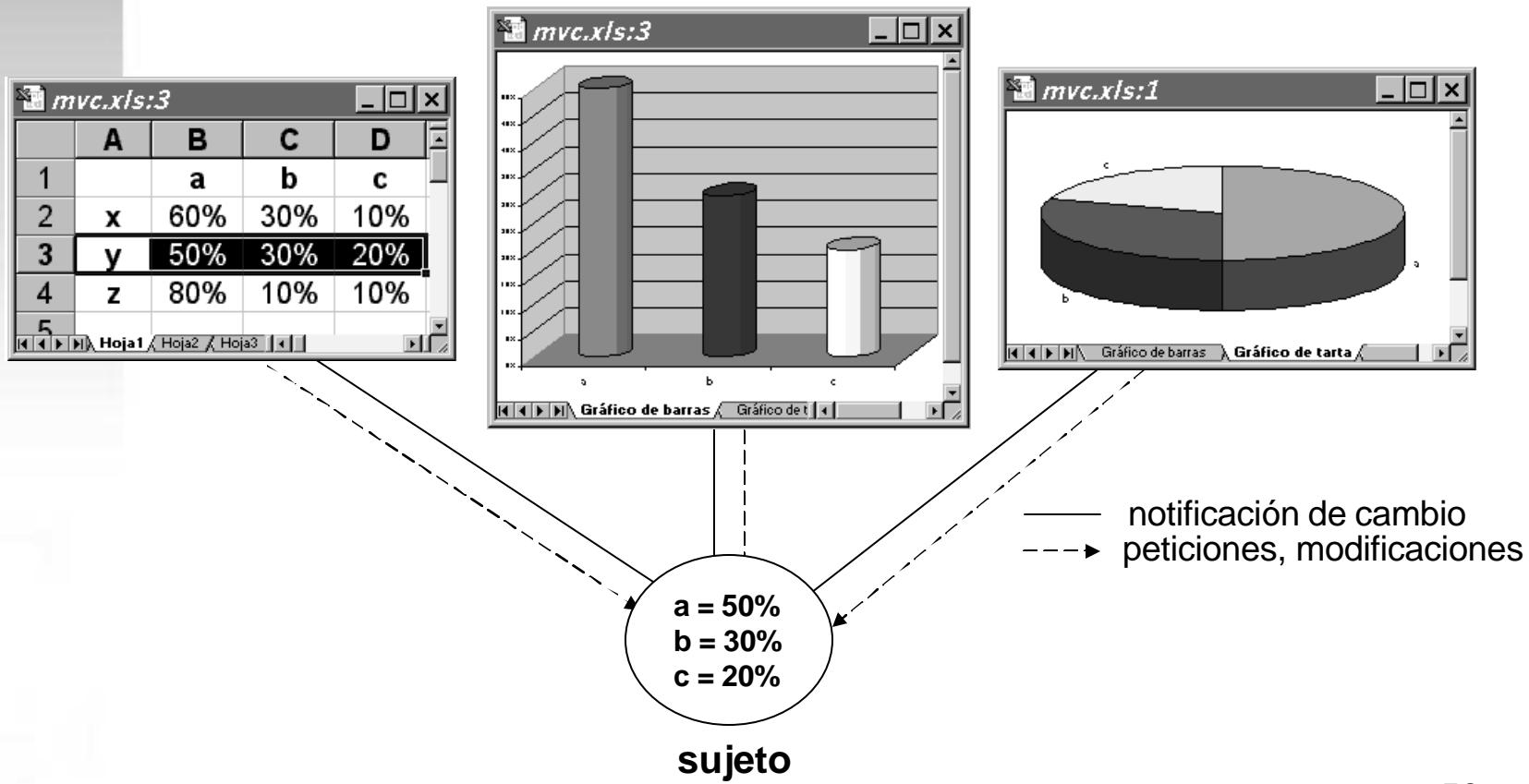
Patrón OBSERVER

- Intención
 - Definir una dependencia $1:n$ de forma que cuando el objeto 1 cambie su estado, los n objetos sean notificados y se actualicen automáticamente
- Motivación
 - En un toolkit de GUI, separar los objetos de presentación (**vistas**) de los objetos de datos, de forma que se puedan tener varias vistas sincronizadas de los mismos datos (**editor-subscriptor**)

Patrón OBSERVER

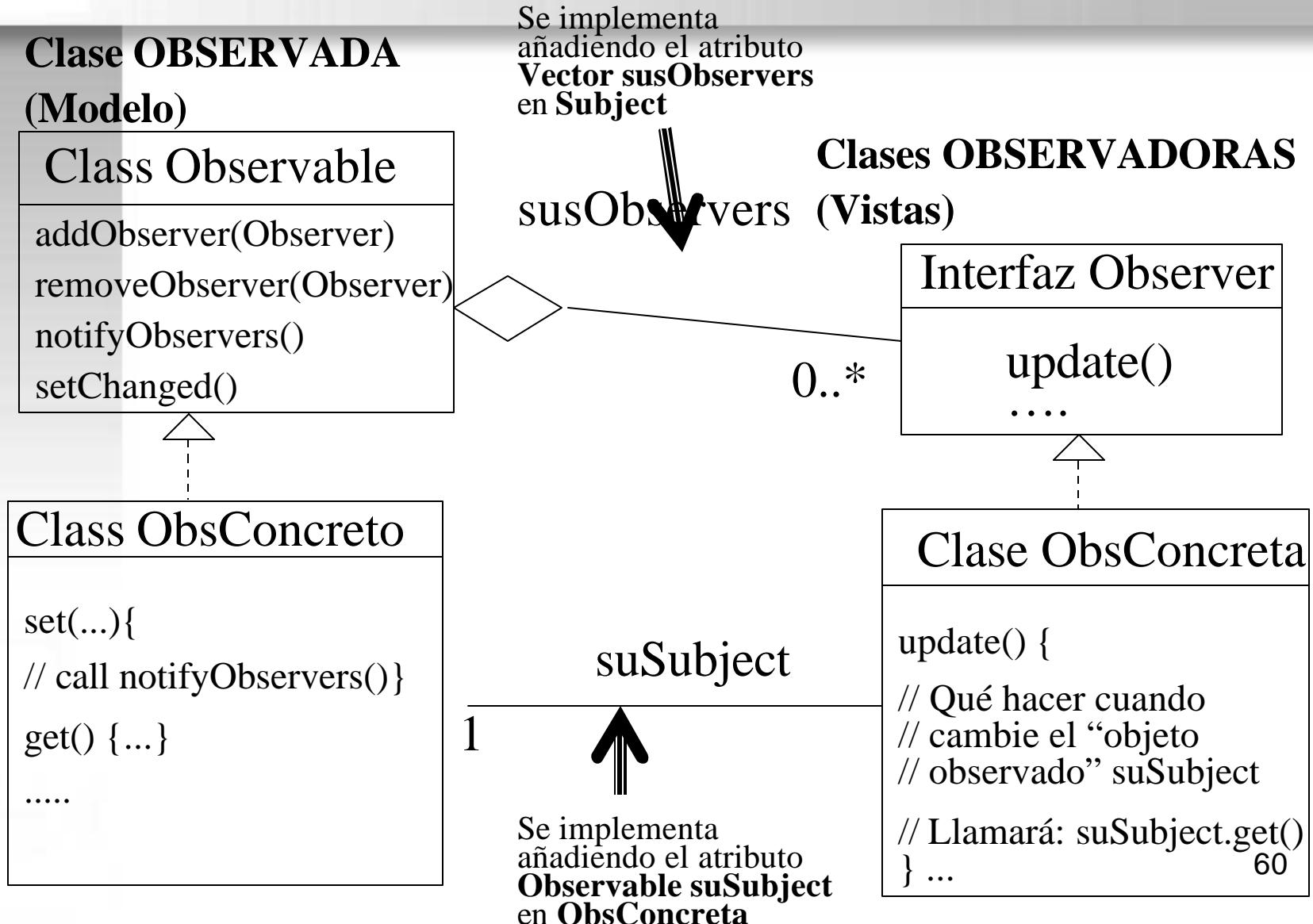
Ejemplo

observadores



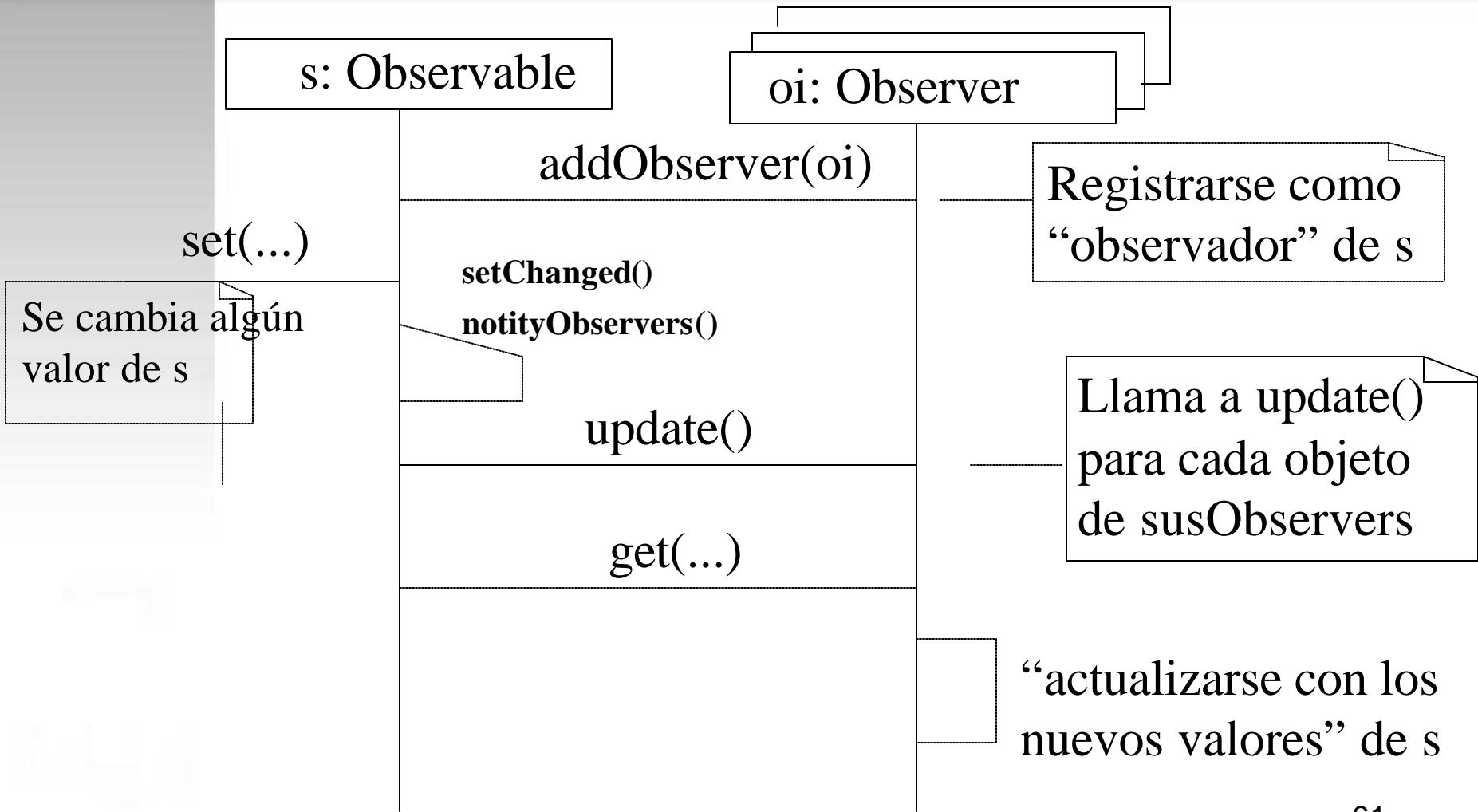
Patrón OBSERVER

La solución



Patrón OBSERVER

Diagrama de interacción



Patrón OBSERVER

La clase Observable

Métodos	Significado
<i>addObserver (o)</i>	Añade un nuevo observador o
<i>deleteObserver(o)</i>	Elimina el observador o
<i>deleteObservers</i>	Elimina todos los observadores
<i>countObservers</i>	Devuelve el número de observadores
<i>setChanged()</i>	marca el objeto como “modificado”
<i>hasChanged</i>	Devuelve cierto si el objeto esta “modificado”
<i>notifyObservers</i>	Notifica a todos los objetos suscritos si el objeto esta en Estado “modificado”

La interfaz Observer

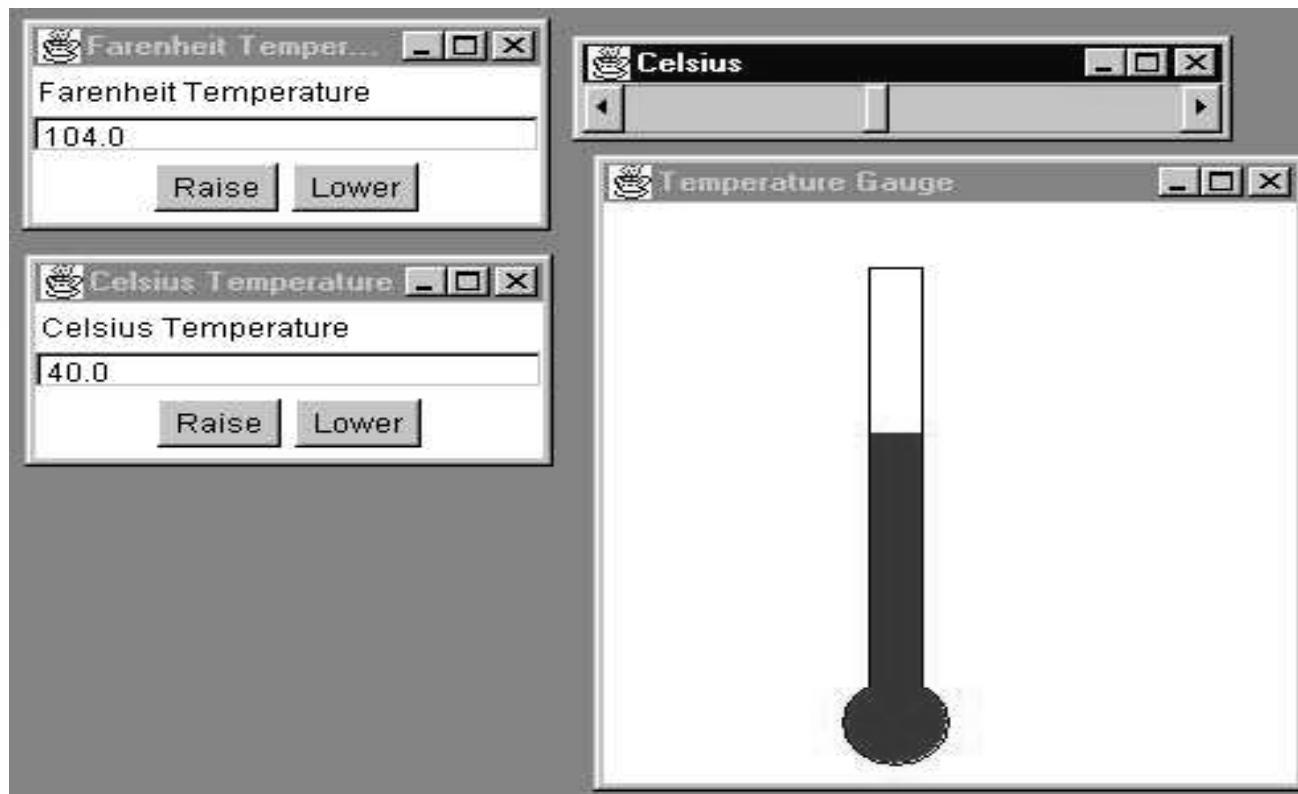
<i>update()</i>	Este método es invocado cuando el objeto observado ha cambiado
-----------------	--

Patrón OBSERVER

Otro Ejemplo

<http://csis.pace.edu/~bergin/mvc/mvcgui.html>

Building Graphical User Interfaces with the MVC Pattern



Patrón OBSERVER

Ejemplo: La clase Observable (el modelo)

```
package mvc;
import java.util.*;
public class Observado extends java.util.Observable{
    public double getF(){return temperatureF; }
    public double getC(){return (temperatureF - 32.0) * 5.0 / 9.0; }
    public void setF(double tempF){
        temperatureF = tempF;
        setChanged();
        notifyObservers();
    }
    public void setC(double tempC){
        temperatureF = tempC*9.0/5.0 + 32.0;
        setChanged();
        notifyObservers();
    }
    private double temperatureF = 32.0;
}
```

Patrón OBSERVER

Ejemplo: La clase Observer (la vista)

```
public class Observador implements java.util.Observer
{
    private Observado modelo;
    public Observador(Observado model)
    {
        modelo=model;
        model.addObserver(this); // Conecta la Vista con el Modelo
    }
    public void update(Observable t, Object o) // Invocada desde el modelo
    {
        System.out.println("La temperatura ha cambiado" + modelo.getF());
    }
}
```

Patrón OBSERVER

Ejemplo: El programa principal

```
public static void main(String[] args) {  
    Observado o=new Observado();  
    Observador ob=new Observador(o);  
    Observador2 ob2=new Observador2(o);  
  
    o.setF(45.5);  
}
```

The temperature has changed 45.5

La temperatura ha cambiado 45.5

Process exited with exit code 0.

OBSERVABLE

OBSERVER

\ejISO\LanzarPatronObserver.java]

Escoger color: SUBJECT de un PATRÓN OBSERVER

AZUL

ROJO

VERDE

AZUL

```
Observer o2 = new Observer() {  
    public void update(Subject s) {  
        //s.setVisible(true);  
    }  
};
```

Observer concreto de un PATRÓN OBSERVER

El COLOR ESCOGIDO ES:

Observer concreto de un PATRÓN OBSERVER

El COLOR ESCOGIDO ES:

Ejemplo de patrón OBSERVER

```
...  
public class ObserverQuePinta extends Frame  
    implements Observer {  
    Label label1 = new Label();  
    Panel panel1 = new Panel();  
    Observable suSubject;  
  
    public ObserverQuePinta(Observable s)  
        suSubject = s;  
        s.addObserver(this);  
    }  
  
    public void update() {  
        String c = suSubject.getColor();  
        if (c.equals("ROJO")) panel1.setBackground(Color.red);  
        else if (c.equals("VERDE")) panel1.setBackground(Color.green);  
        else if (c.equals("AZUL")) panel1.setBackground(Color.blue);  
    } }
```

VISTA

```
public interface Observer {  
    void update();  
}
```

```
public class Subject extends Observable {  
    Choice choice1 = new Choice();  
    public Subject() {  
        choice1.addItem("ROJO"); choice1.addItem("VERDE");  
        choice1.addItem("AZUL");  
        choice1.addItemListener(new java.awt.event.ItemListener() {  
            public void itemStateChanged(ItemEvent e) {  
                setChanged();  
                notifyObservers(); } }  
    }  
    public String getColor() {return choice1.getSelectedItem();}  
}
```

Patrón OBSERVER

Inicialización

```
Subject s = new Subject();  
Observer o1 = new ObserverQuePinta(s);  
Observer o2 = new ObserverQuePinta(s);
```

CREACIÓN DE OBJETOS SUBJECT
Y OBSERVER

Patrón OBSERVER

Adaptabilidad de la solución

- Si se quisiera añadir un nuevo OBSERVER que en vez de pintar escribiera un número para cada color (1 si ROJO, 2 si VERDE, 3 si AZUL), ¿qué habría que hacer?
 - Escribir nueva clase OBSERVER que implemente “notificar”
- Si se quisiera cambiar el SUBJECT para escoger el color pinchando un Checkbox en vez de escogerlo de un Choice
 - Hay que sustituir el Subject y llamar a “notificar” de los observers cuando se seleccione un nuevo Checkbox

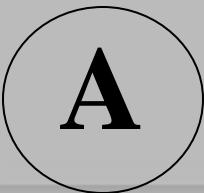
A

B

```

...
public class ObserverQueEscribeNums extends Frame
    implements Observer {
    Label label1 = new Label();
    TextArea textArea1 = new TextArea();
    Subject suSubject;
    public ObserverQueEscribeNums(Subject s) {
        suSubject = s;
        s.addObserver(this); ...
    }
    public void update() {
        String c = suSubject.getColor();
        if (c.equals("ROJO"))
            textArea1.append("NUEVO COLOR: "+1+"\n");
        else if (c.equals("VERDE"))
            textArea1.append("NUEVO COLOR: "+2+"\n");
        else if (c.equals("AZUL"))
            textArea1.append("NUEVO COLOR: "+3+"\n");
    }
}

```



```

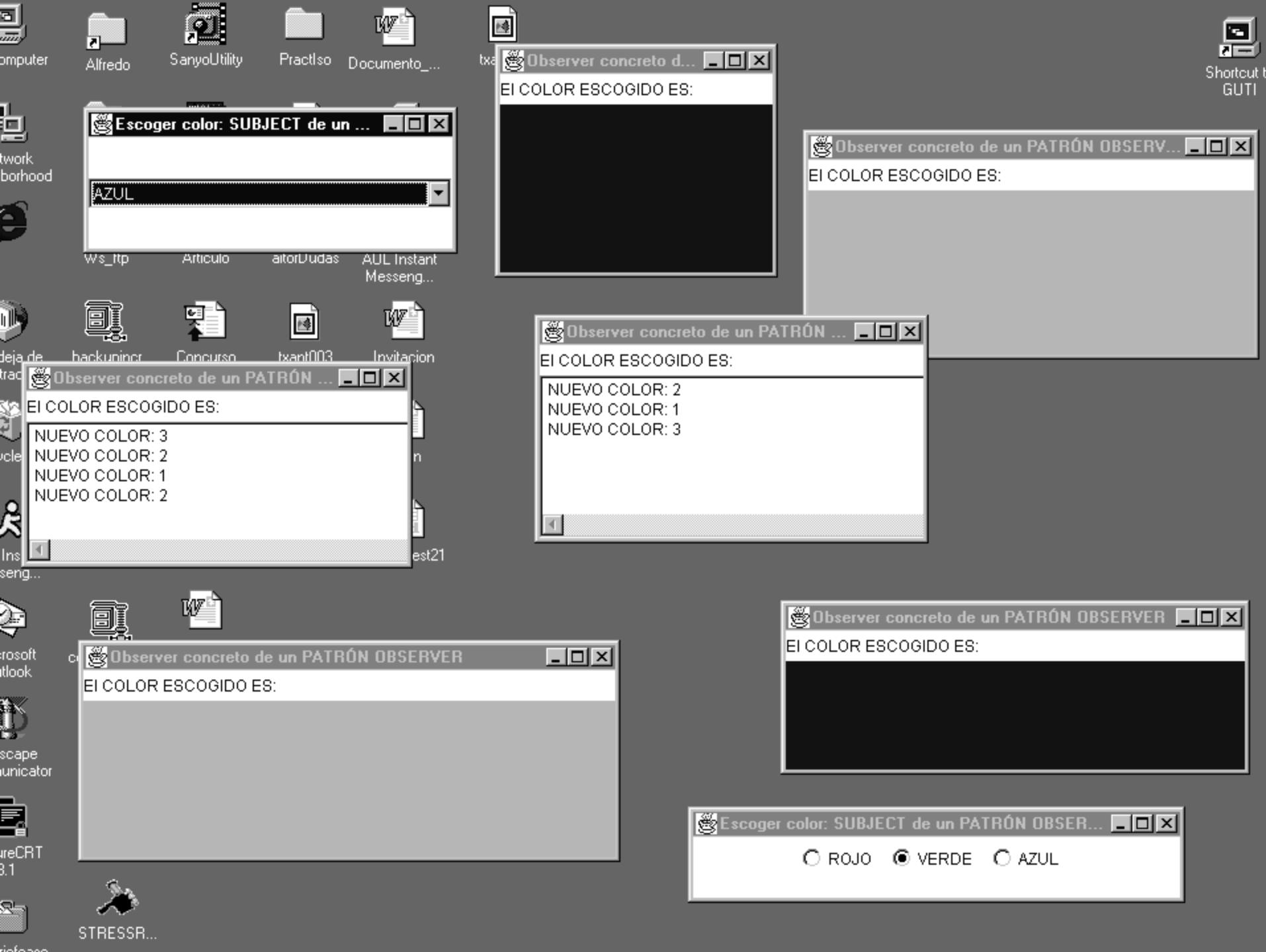
public interface Observer {
    void notificar();
}

```

EXTENDER CON UN NUEVO OBSERVER

...

```
public class Subject extends Frame {  
    Checkbox checkbox1 = new Checkbox();  
    Checkbox checkbox2 = new Checkbox() // ... y checkbox3  
    CheckboxGroup checkboxGroup1 = new CheckboxGroup();  
    public Subject() {  
        checkbox1.setLabel("ROJO");  
        checkbox1.setCheckboxGroup(checkboxGroup1);  
        checkbox1.addItemListener(new java.awt.event.ItemListener() {  
            public void itemStateChanged(ItemEvent e) {  
                notificar(); } });  
        checkbox2.setLabel("VERDE"); // y checkbox3.setLabel("AZUL");...  
    }  
    public void addObserver(Observer o) { susObservers.addElement(o); }  
    public void notificar() {  
        setChanged();  
        notifyObservers(); } }  
    public String getColor() {  
        return checkboxGroup1.getSelectedCheckbox().getLabel(); } }
```



Patrón STRATEGY

Intención:

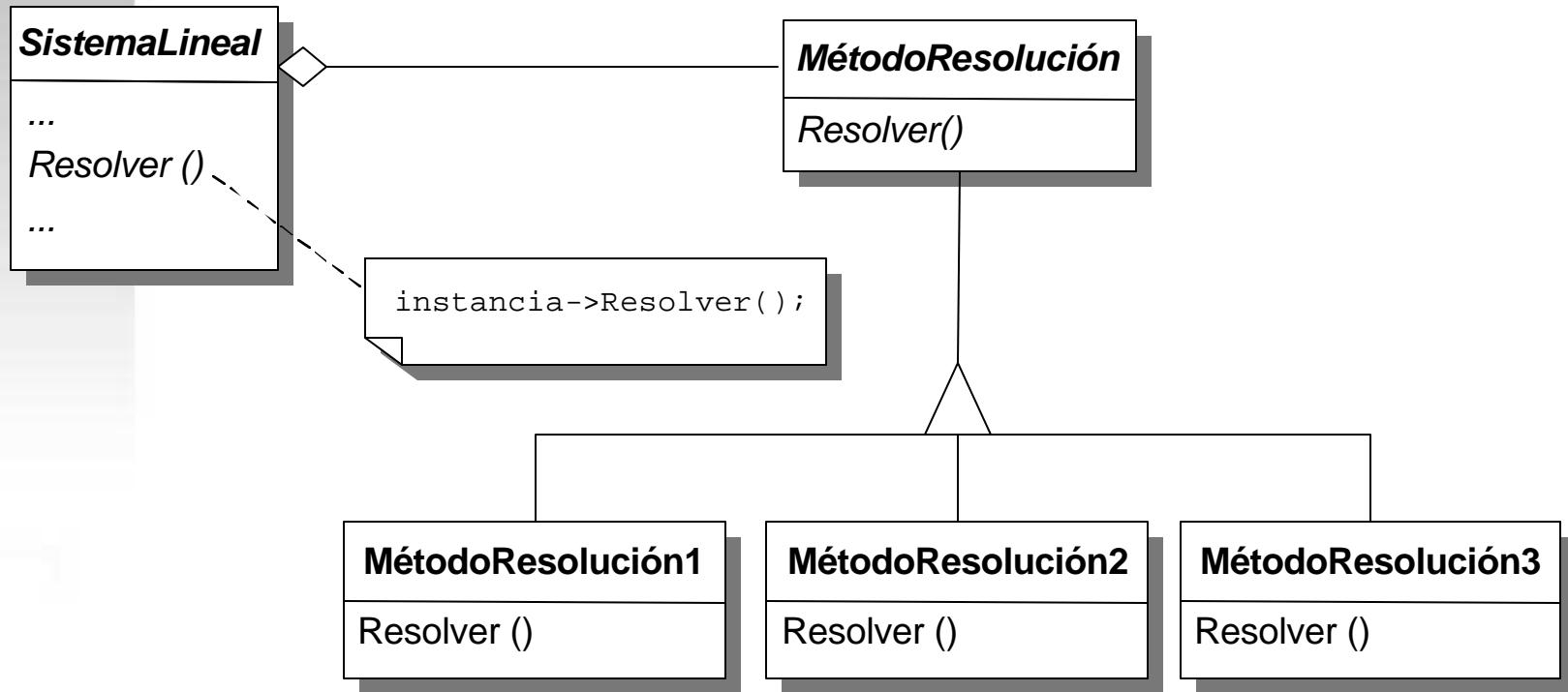
- Encapsular algoritmos relacionados en clases y hacerlos intercambiables.
- Se permite que la selección del algoritmo se haga según el objeto que se trate.

Ventajas

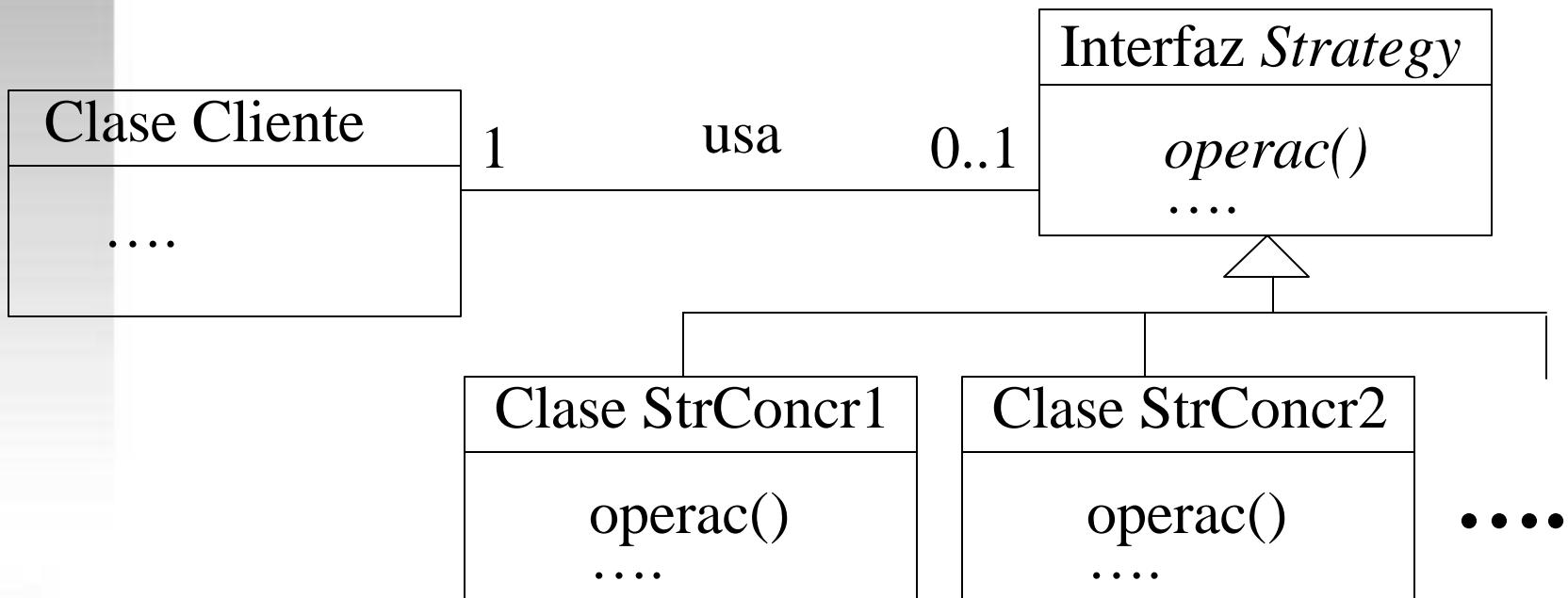
- Se permite cambiar el algoritmo dinámicamente
- Se eliminan sentencias condicionales para seleccionar el algoritmo deseado

Patrón STRATEGY

La solución



Patrón STRATEGY



La clase Cliente necesita ejecutar un método: operac(), que puede ser implementado siguiendo distintos algoritmos. Se pueden implementar todos ellos (en las clases StrConcrX) y ser transparente a Cliente.

Patrón STRATEGY

Implementación

```
class Cliente {  
    Strategy s;  
    // PARA SELECCIONAR EL ALGORITMO  
    // SE ASIGNA A s LA INSTANCIA  
    // DE LA SUBCLASE QUE LO IMPLEMENTA  
    Cliente(Strategy str) { s = str; }  
    // EL Cliente SIEMPRE SE EJECUTARÁ ASÍ:  
    s.operac();      }
```

Cliente c = new Cliente(new StrConcr1());

Se indica cuál es el algoritmo que se ejecutará

Patrón STRATEGY

Implementación

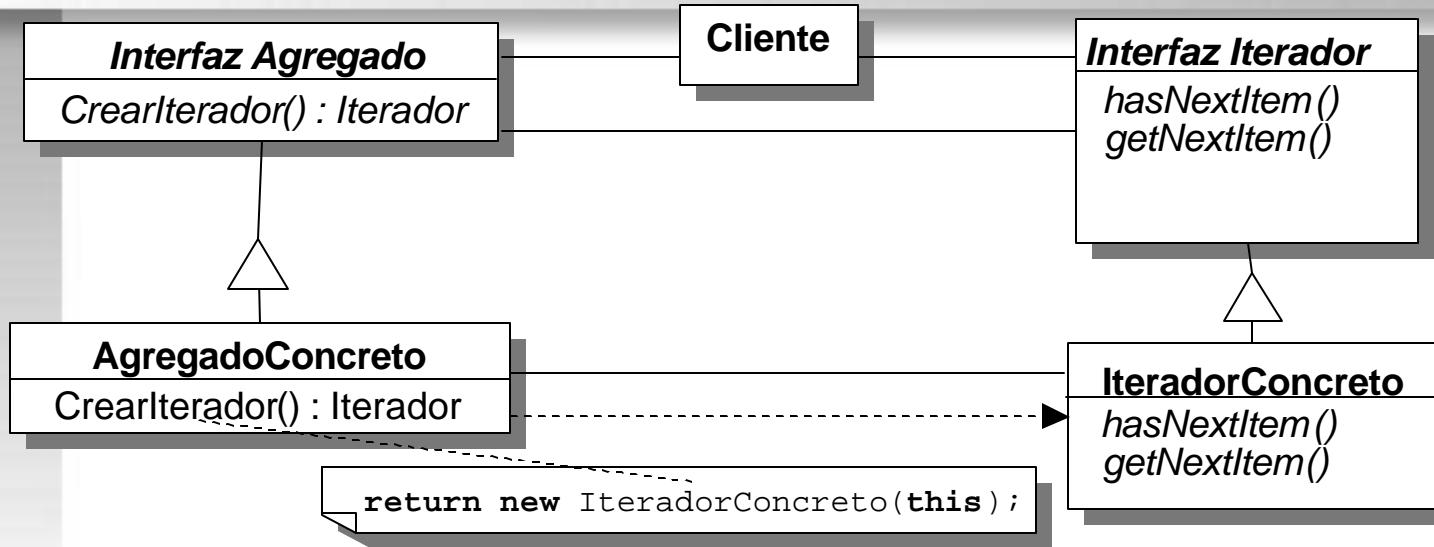
```
class Cliente {  
    // EN VEZ DE:  
    if (cond1) operac1();  
    else if (cond2) operac2();  
    else ...  
}
```

- Sentencias condicionales para seleccionar el algoritmo
- Si se quisiera añadir una nueva forma de ejecutar operac() entonces HABRÍA QUE CAMBIAR EL CÓDIGO DE LA CLASE Cliente

Patrón ITERATOR

- **Intención:**
 - Proporcionar una forma de acceder a los elementos de una colección de objetos de manera secuencial sin revelar su representación interna. Define una interfaz que declara métodos para acceder secuencialmente a la colección.
- **Ventajas:**
 - La clase que accede a la colección solamente a través de dicho interfaz permanece independiente de la clase que implementa la interfaz.

Patrón ITERATOR



◆ Participantes

- ◆ **Iterador**: define una interfaz para acceder a los elementos del agregado y recorrerlos
- ◆ **IteradorConcreto**: implementa la interfaz de **Iterador** y mantiene la posición actual del recorrido
- ◆ **Agregado**: define una interfaz para crear un objeto iterador
- ◆ **AgregadoConcreto**: implementa la interfaz de creación del iterador para devolver una instancia apropiada de **IteradorConcreto**

Patrón ITERATOR

- Ejemplos en Java:

```
Vector listOfStudents = new Vector();
// PARA RECORRER EL VECTOR:
```

```
Enumeration list = listOfStudents.elements();
while ( list.hasMoreElements() )
    System.out.println( list.nextElement() );
```

```
Hashtable anIndex = new Hashtable();
// PARA RECORRER LA TABLA HASH:
```

```
Enumeration list = anIndex.keys();
while ( list.hasMoreElements() )
    System.out.println( list.nextElement() );
```