Ingeniería del Software II

Tema 3: Diseño Software

3.2. Patrones de diseño

A. Goñi, J. Iturrioz

OpenCourseWare

cc BY NC SA

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.

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, primando los aspectos de reutilización y extensibilidad.

Definición de un patrón de diseño software

"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." [1]

[1] Libro Patrones de diseño. Elementos de software orientado a objetos reutilizable. E. Gamma, R. Helm, R. Johnson, J.Vlissides. Addison Wesley, 2002

Elementos de un patrón

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

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 [2]

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: Patrón Delegation (Estructural)

[2] Libro Patrones de diseño. Elementos de software orientado a objetos reutilizable. E. Gamma, R. Helm, R. Johnson, J.Vlissides. Addison Wesley, 2002

Patrón DELEGATION

NO es un patrón que aparece en el libro de Gamma et al. pero es muy utilizado.

Utilidad:

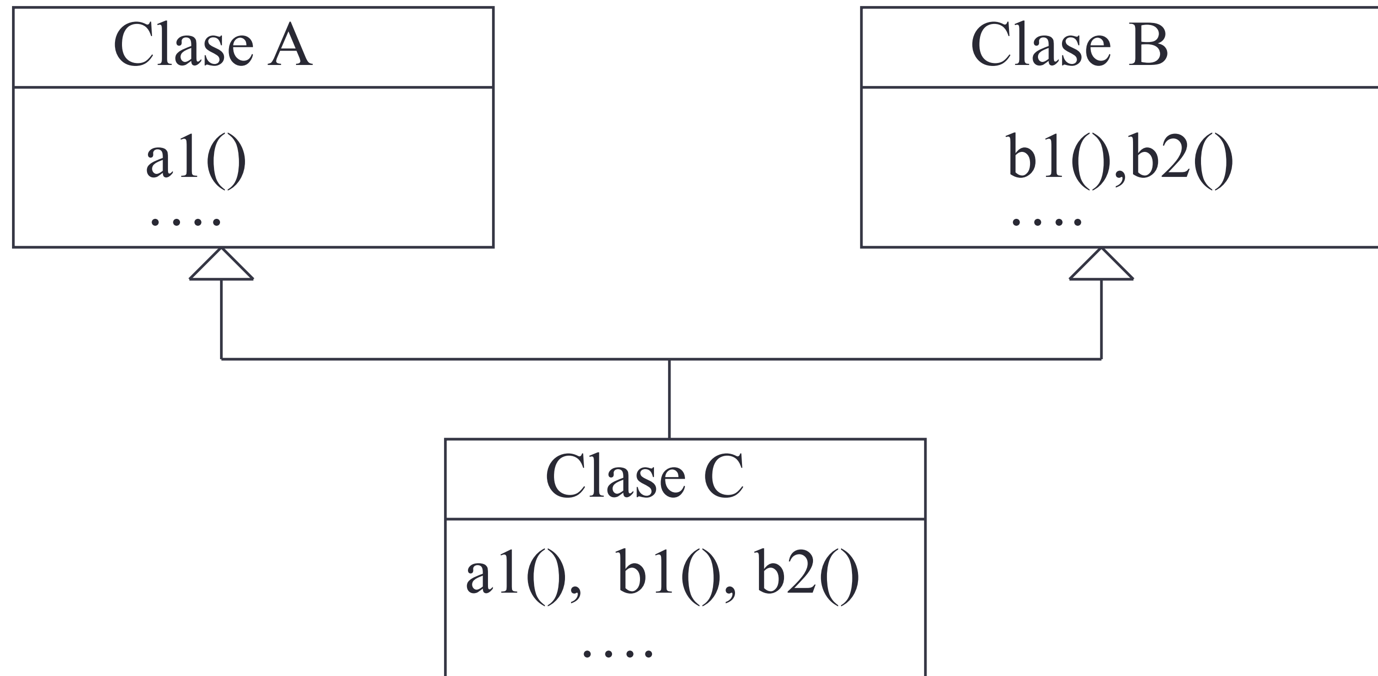
Cuando se quiere extender y reutilizar la funcionalidad de una clase SIN UTILIZAR LA HERENCIA.

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

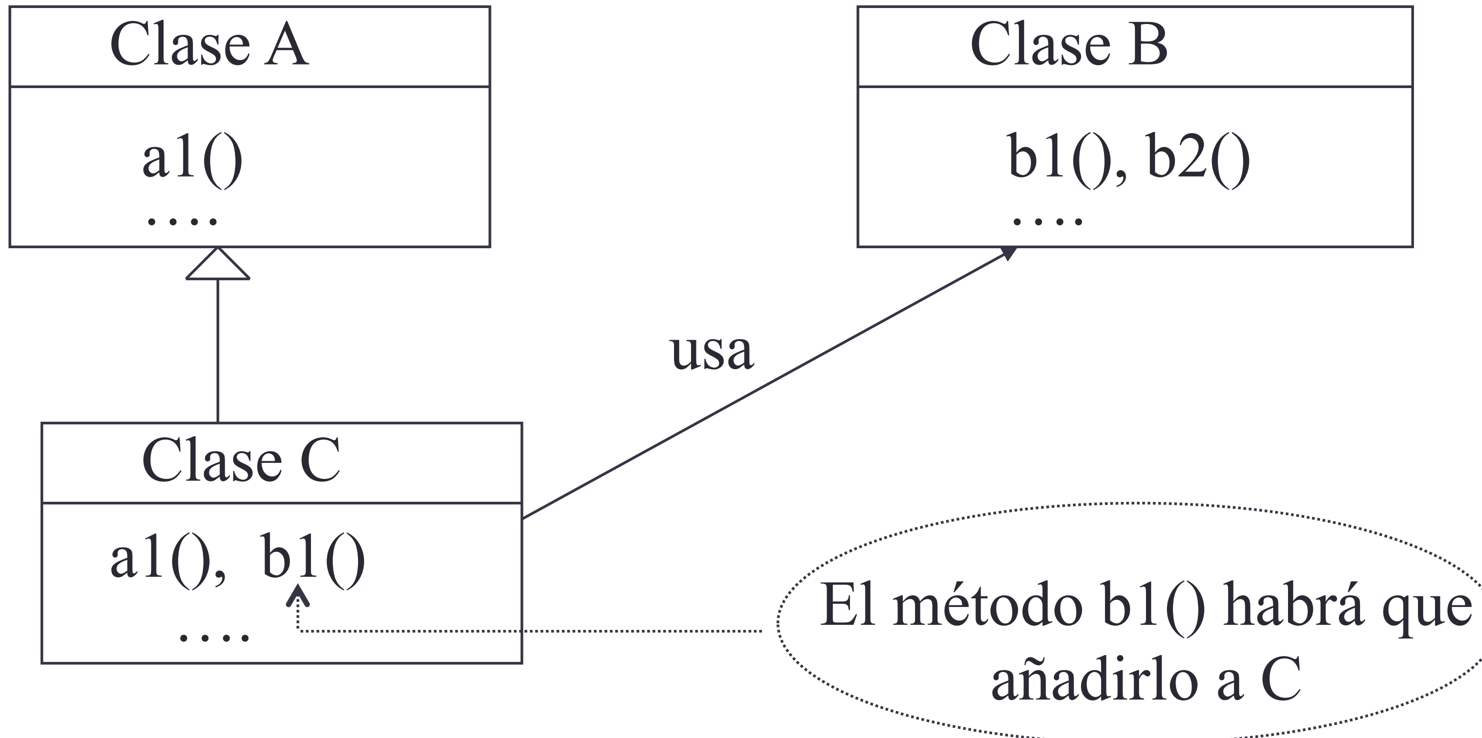
El problema



- 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( );}
        ….
```

Clasificación de los patrones

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

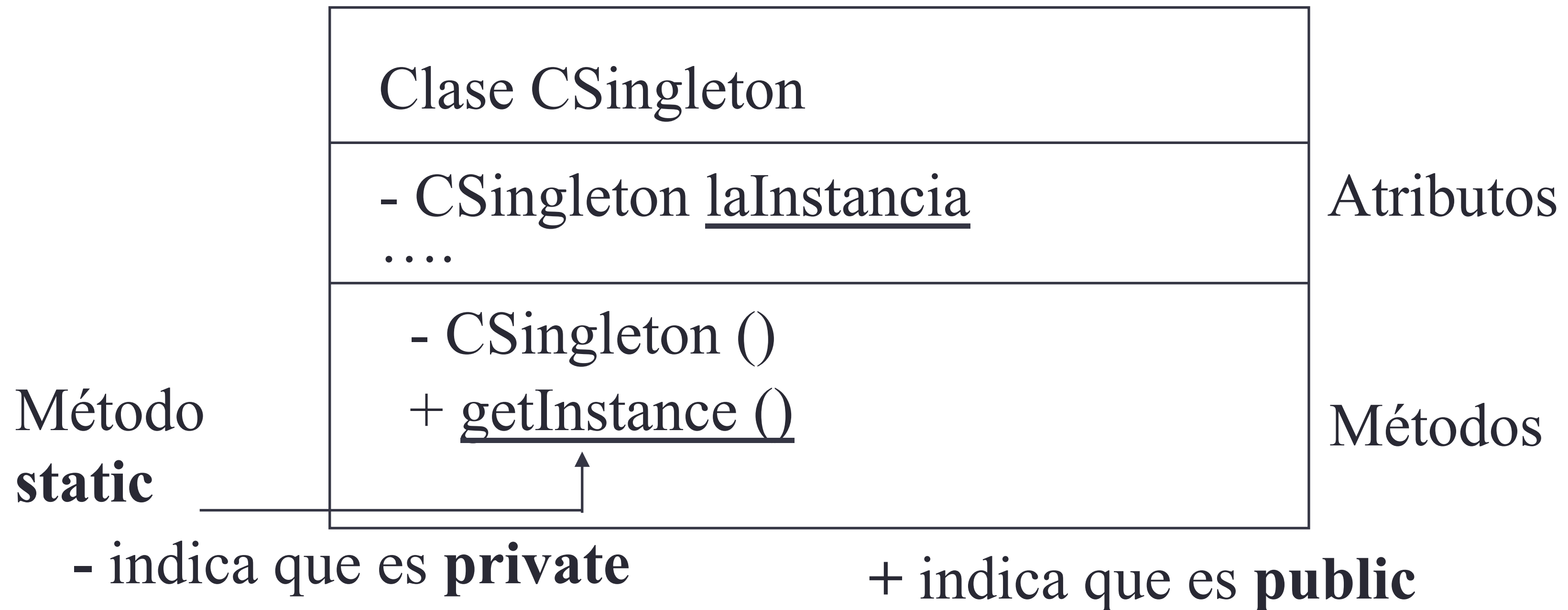
- **Utilidad:**
  - Asegurar que una clase tiene una sola instancia y proporcionar un punto de acceso global a ella.

- **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 {...}
```

Ejemplo SINGLETON

FacultadInfoDonosti

```
public final class FacultadInformaticaDonosti  {
        private Vector listaProfesores;
        private Vector listaEstudiantes;
        private Vector listaAsigs;
        private Vector listaMatrs;
        private static FacultadInfoDonosti laFacultad=new FacultadInfoDonosti();
        private FacultadInfoDonosti() {        // 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 FacultadInfoDonosti getInstance() {return laFacultad;}
        public Vector obtListaProfesores() {
                return listaProfesores;
        }
        public void añadirProfesor(Profesor p) {
                listaProfesores.addElement(p);
        }}
```

Clasificación de los patrones

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

Utilidad:

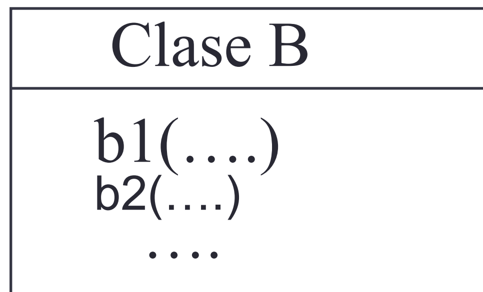
- Convertir la interfaz de una clase en otra interfaz esperada por los clientes.
- Permite que clases con interfaces incompatibles se comuniquen.

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 objetos con forma de clase.

Patrón ADAPTER

El problema



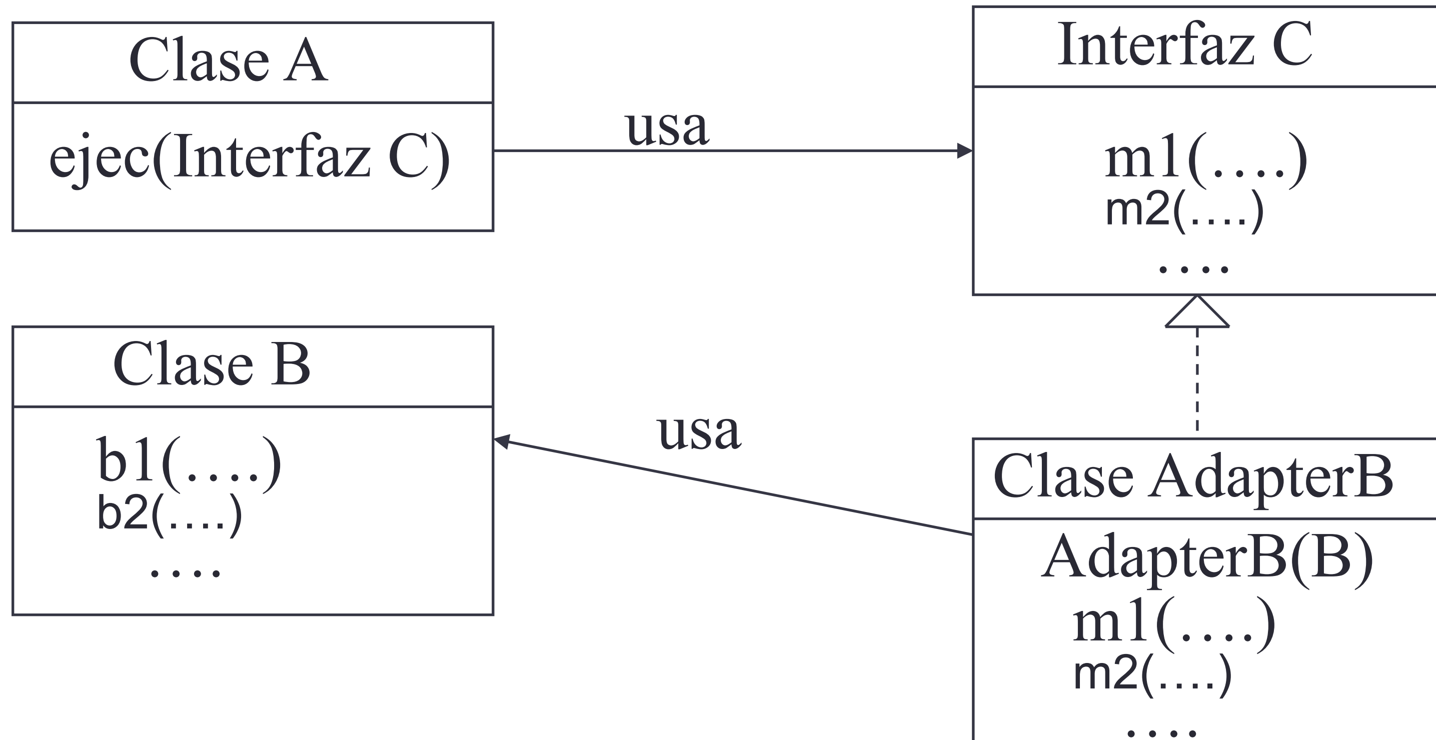
Se desea utilizar la clase A (el método ejec) utilizando como entrada un objeto de la clase B

objetoDeA.ejec(objetoDeB)

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

Patrón ADAPTER

La solución



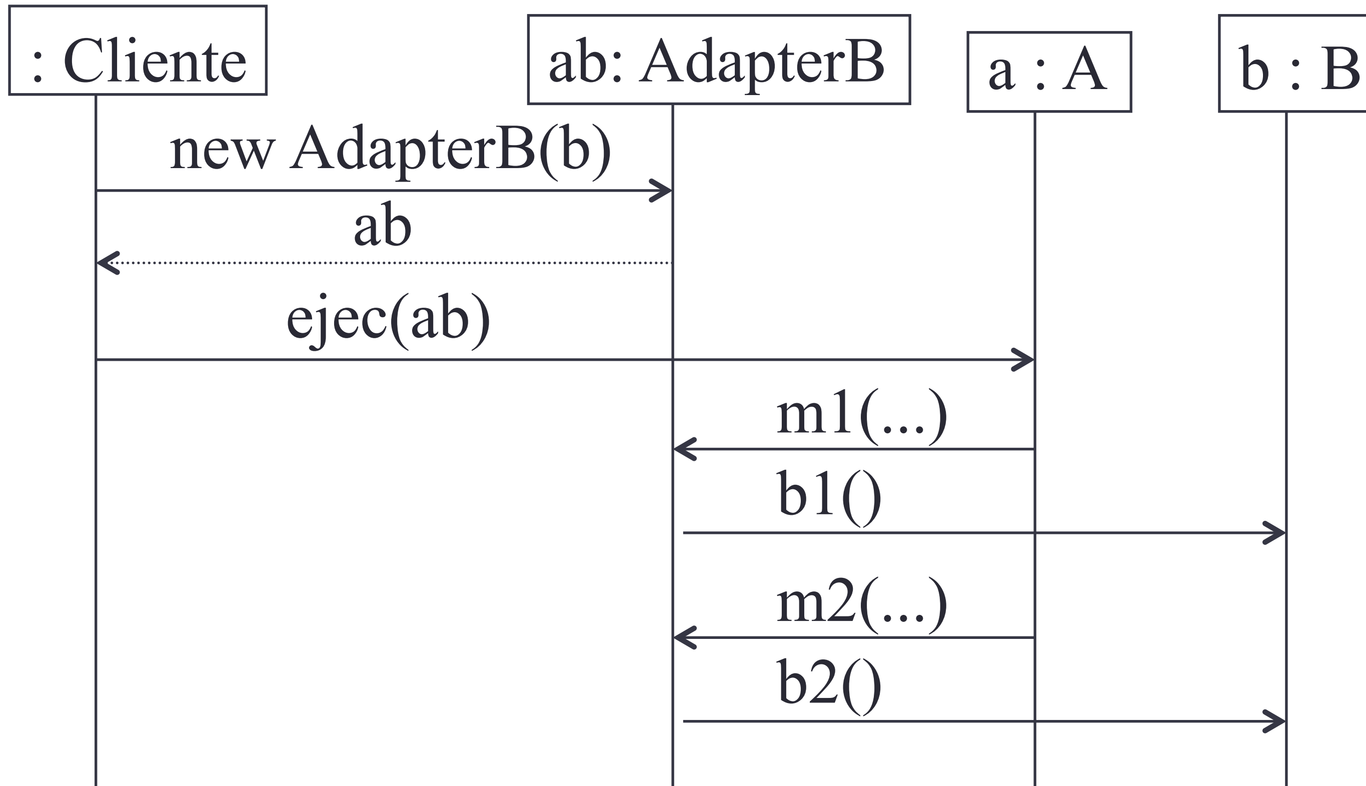
Solución: construir una clase Adaptadora de B que implemente la interfaz C. Al implementarla, usa un objeto de B y sus métodos

Para utilizar la clase A:

objetoDeAdapterB =NEW AdapterB(objetoDeB)

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

Ejemplo I. Lo que yo tengo y lo que yo quiero.

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

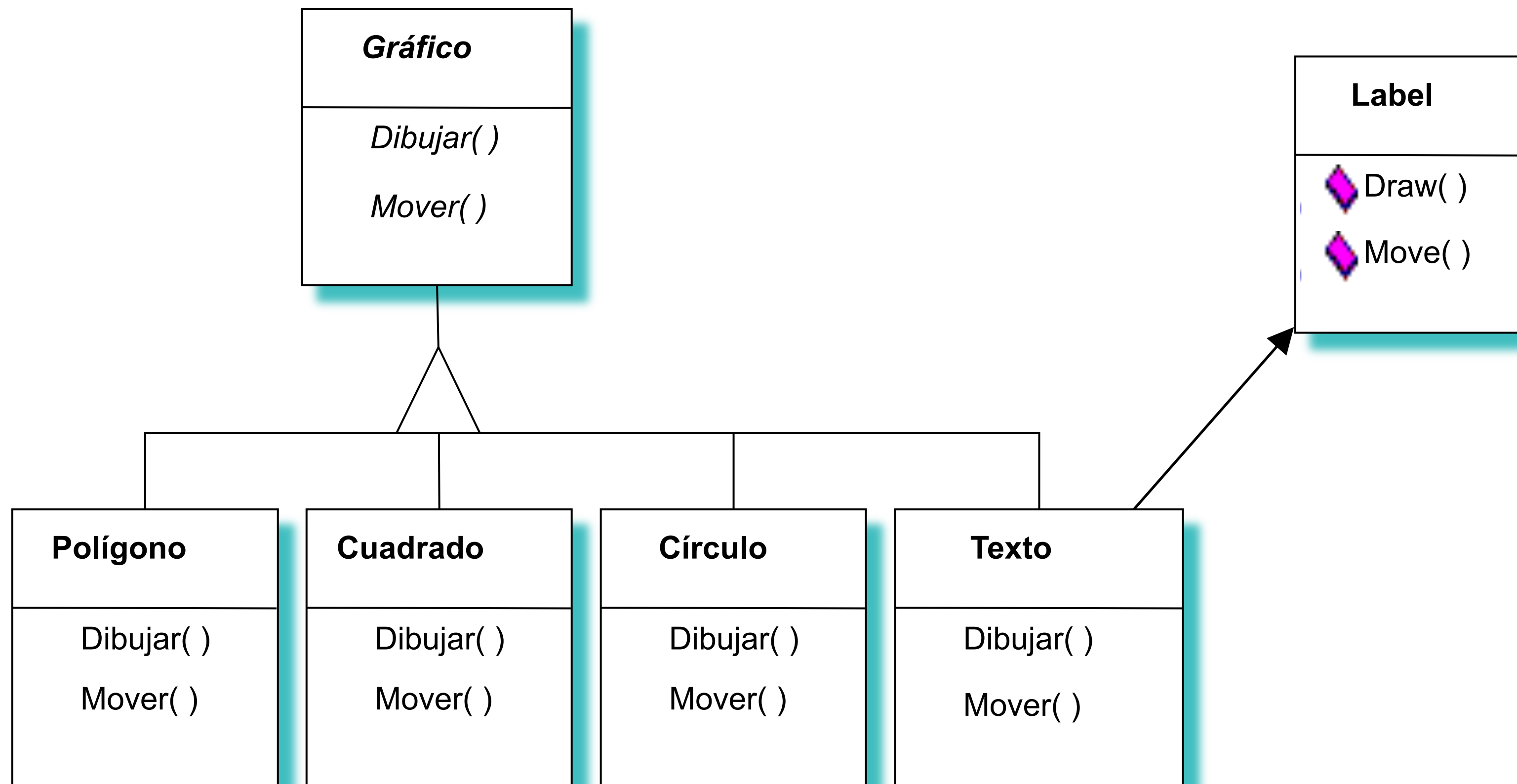
```
class WhatIHave {
  public void g() {}
  public void h() {} }
```

```
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.



Clasificación de los patrones

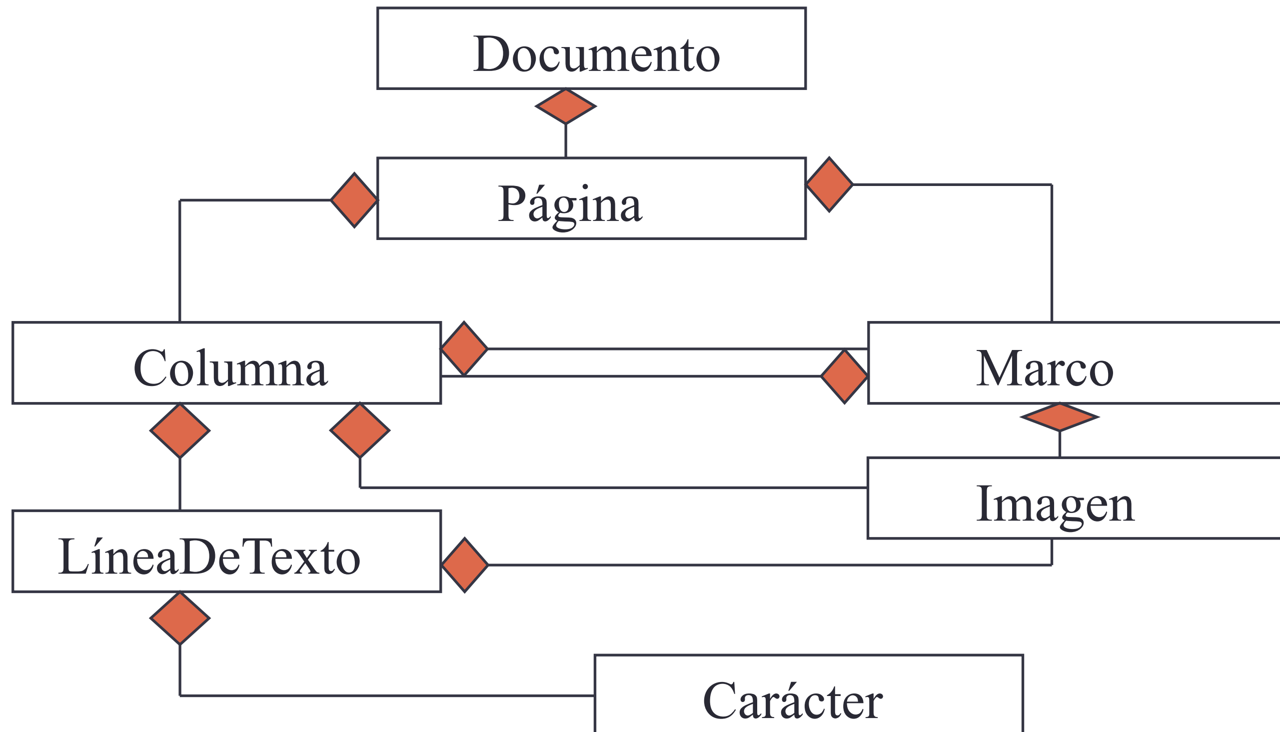
Propósito/ Ámbito	Creación	Estructural	Comportamiento
Clase	Factory Method	Adapter	Interpreter Template Method
Objeto	Abstract Factory Builder Prototype ✓ **Singleton**	✓ **Adapter** Bridge ✓ **Composite** Decorator Facade Flyweight Proxy	Chain of Responsability Command Iterator Mediator Memento ✓ **Observer**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

Patrón COMPOSITE

- **Utilidad:**
  - 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

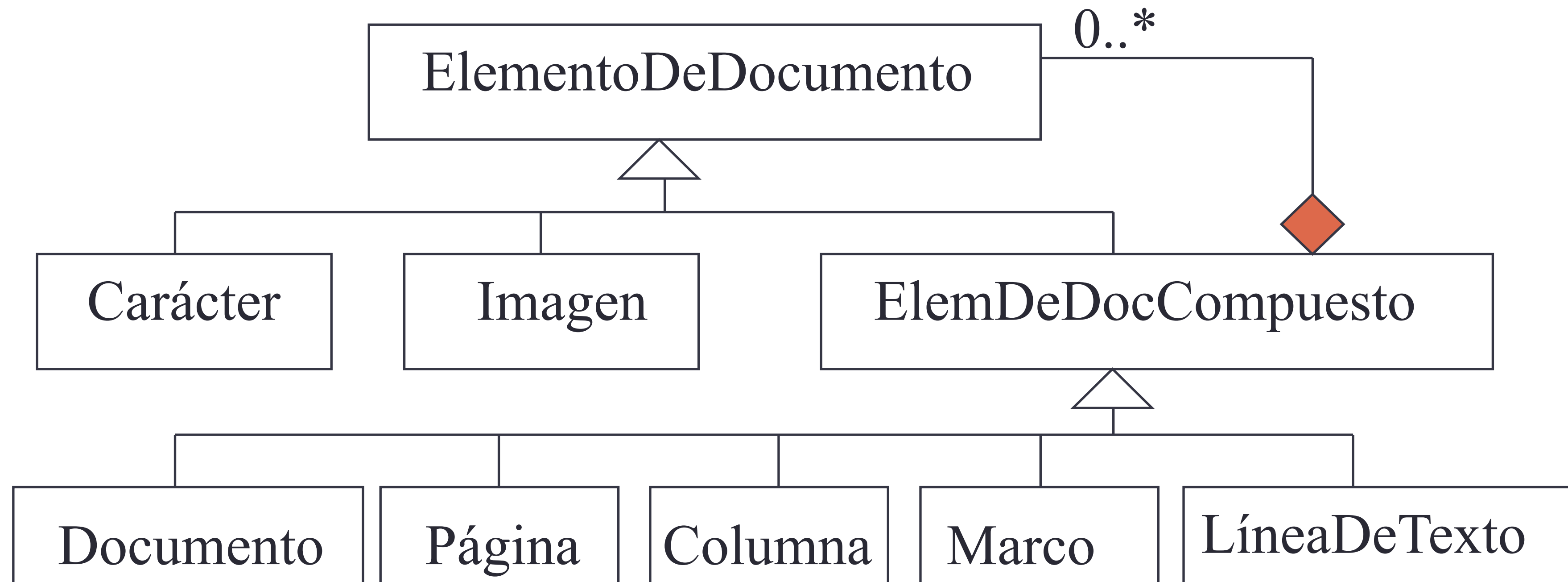
El problema: La escalabilidad



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

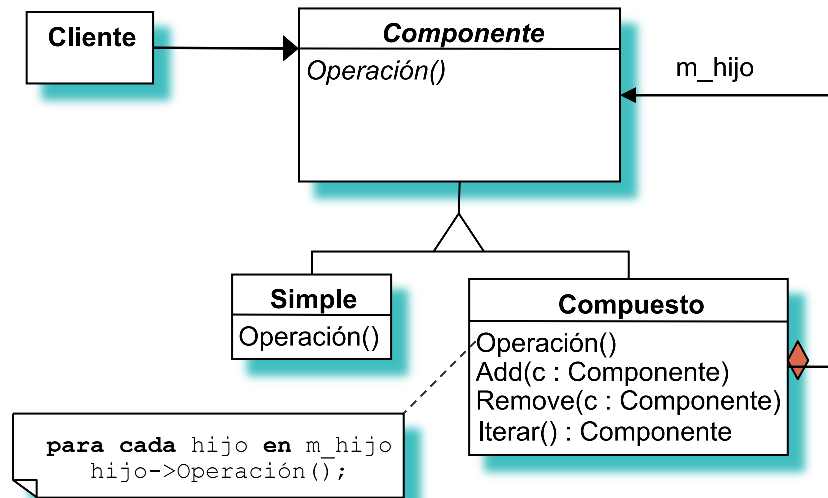


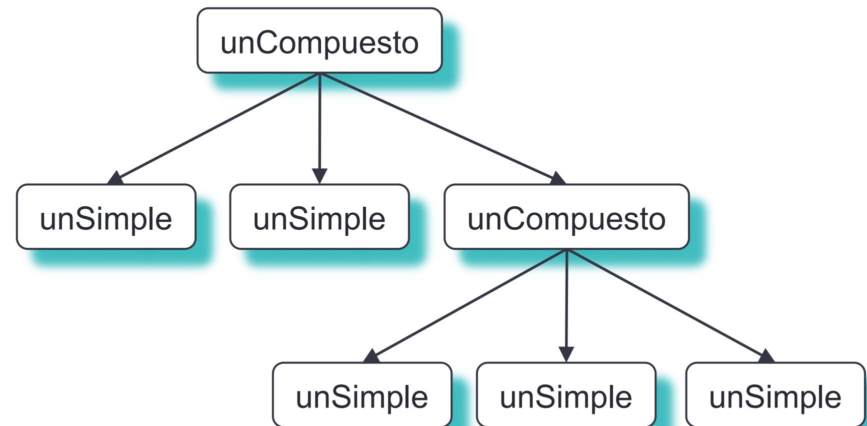
Un **documento** está formado por varias **páginas**, las cuales están formadas por **columnas** que contienen **líneas de texto**, formadas por **caracteres.**

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

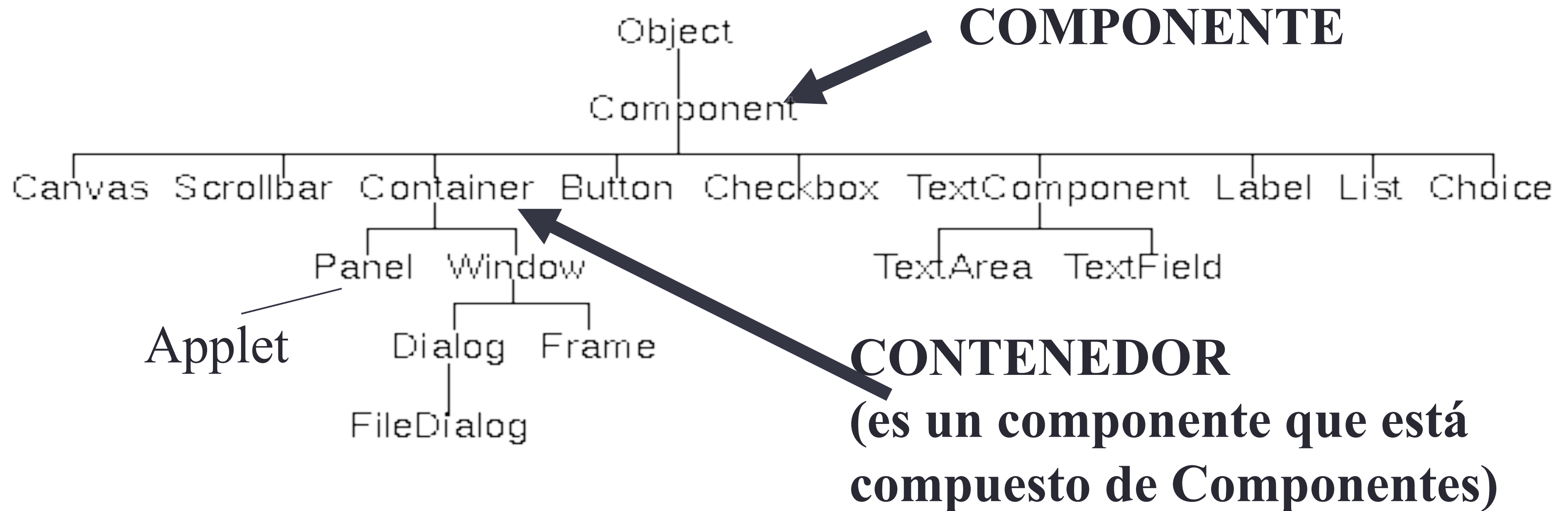
Ejemplo: La jerarquía de clases de AWT (Abstract Window Toolkit)

- 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

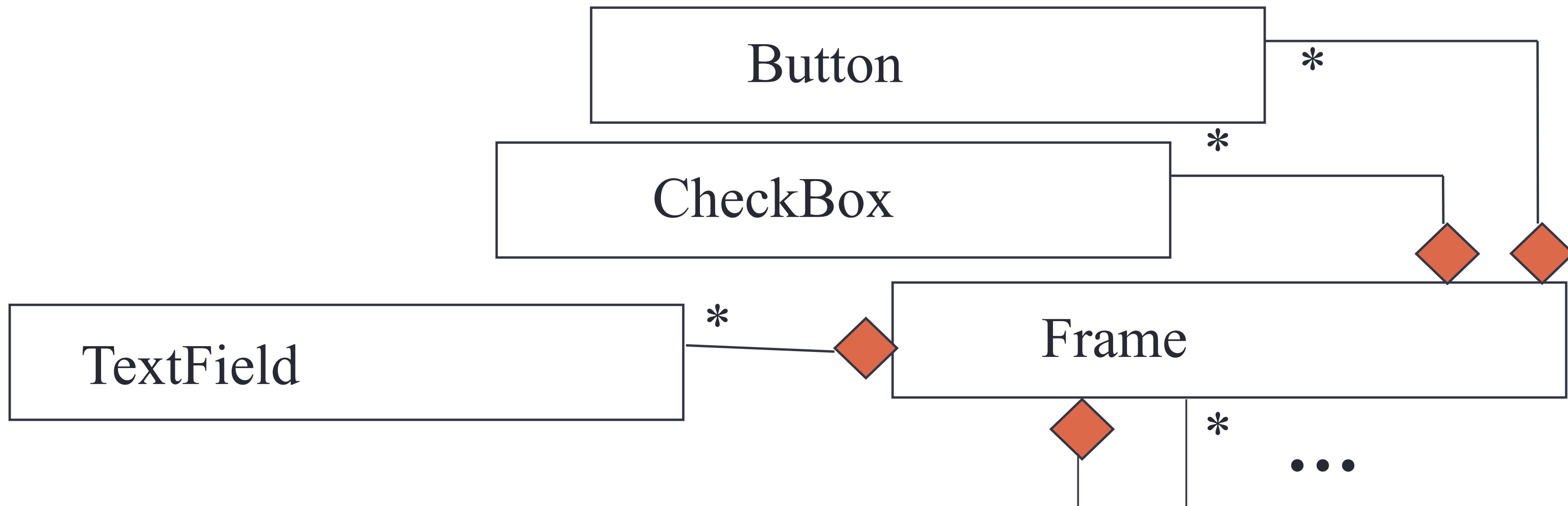


Las posibilidades son infinitas…

- 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)

Clasificación de los patrones

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

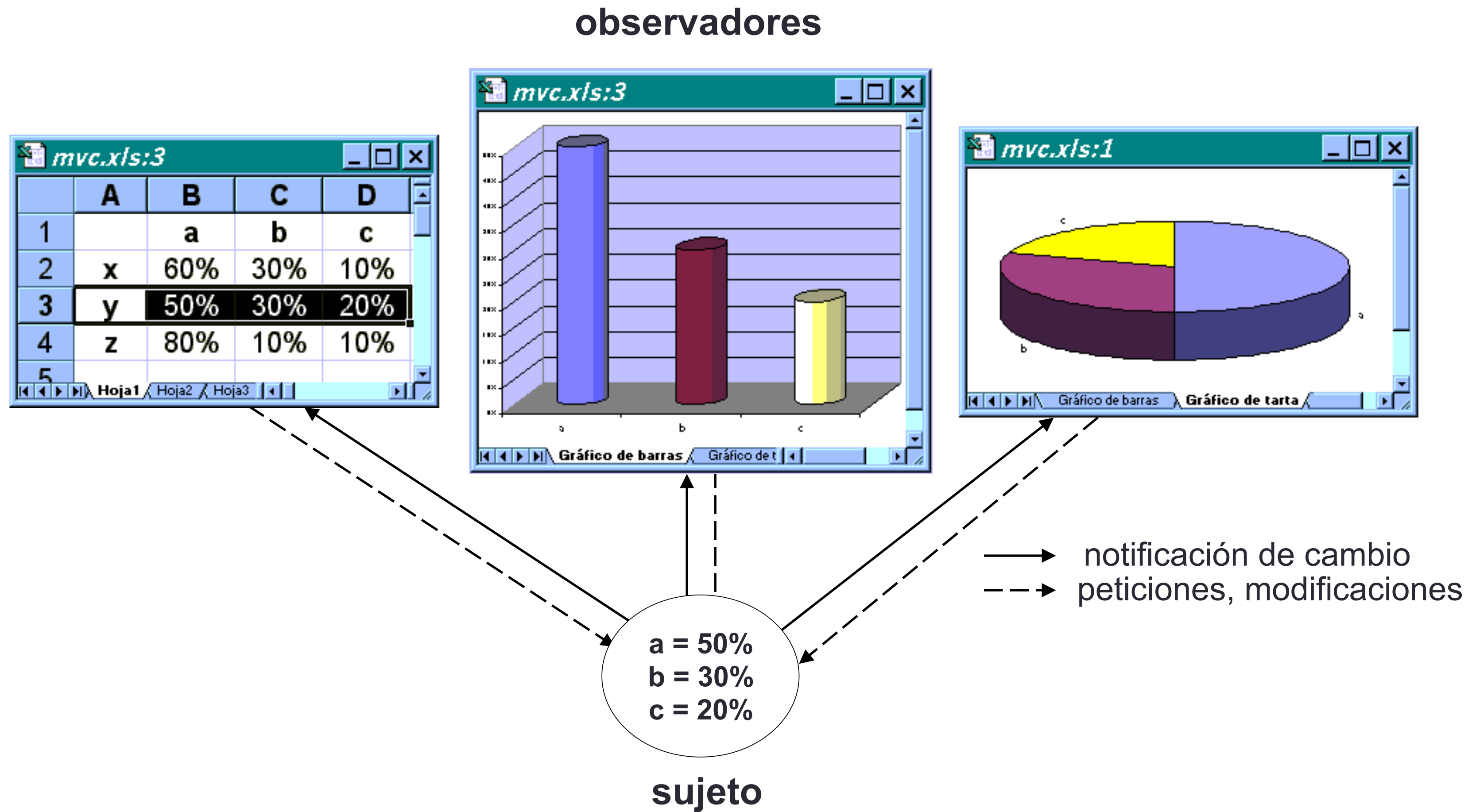
Patrón OBSERVER

- **Utilidad:**
  - 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



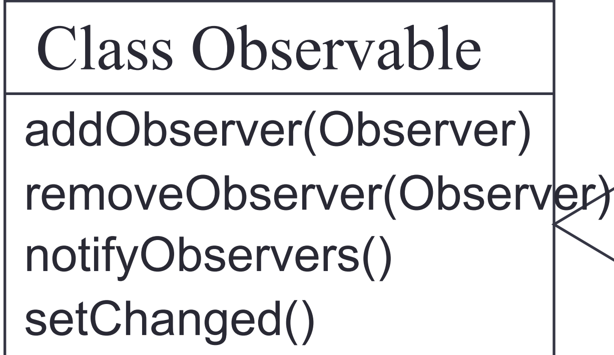
Patrón OBSERVER

La solución

Clase OBSERVADA (Modelo)

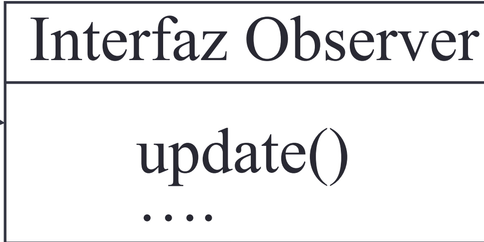
Se implementa añadiendo el atributo **Vector susObservers** en **Subject**

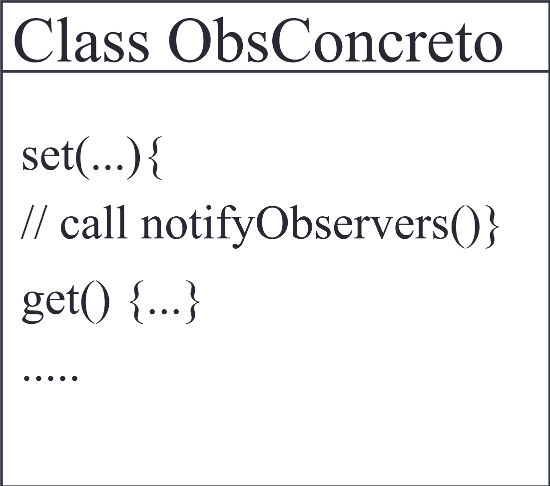
Clases OBSERVADORAS (Vistas)



susObservers

0..*

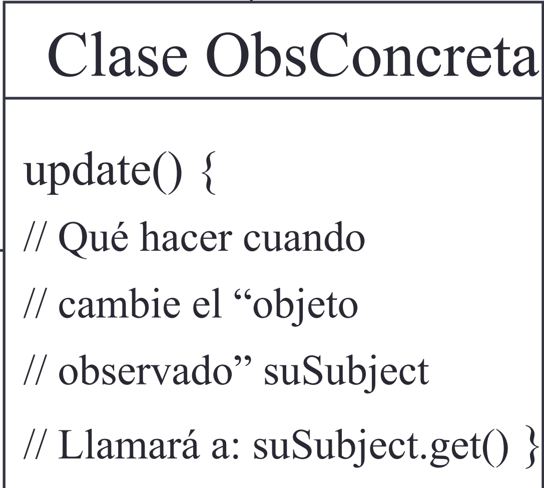




suSubject

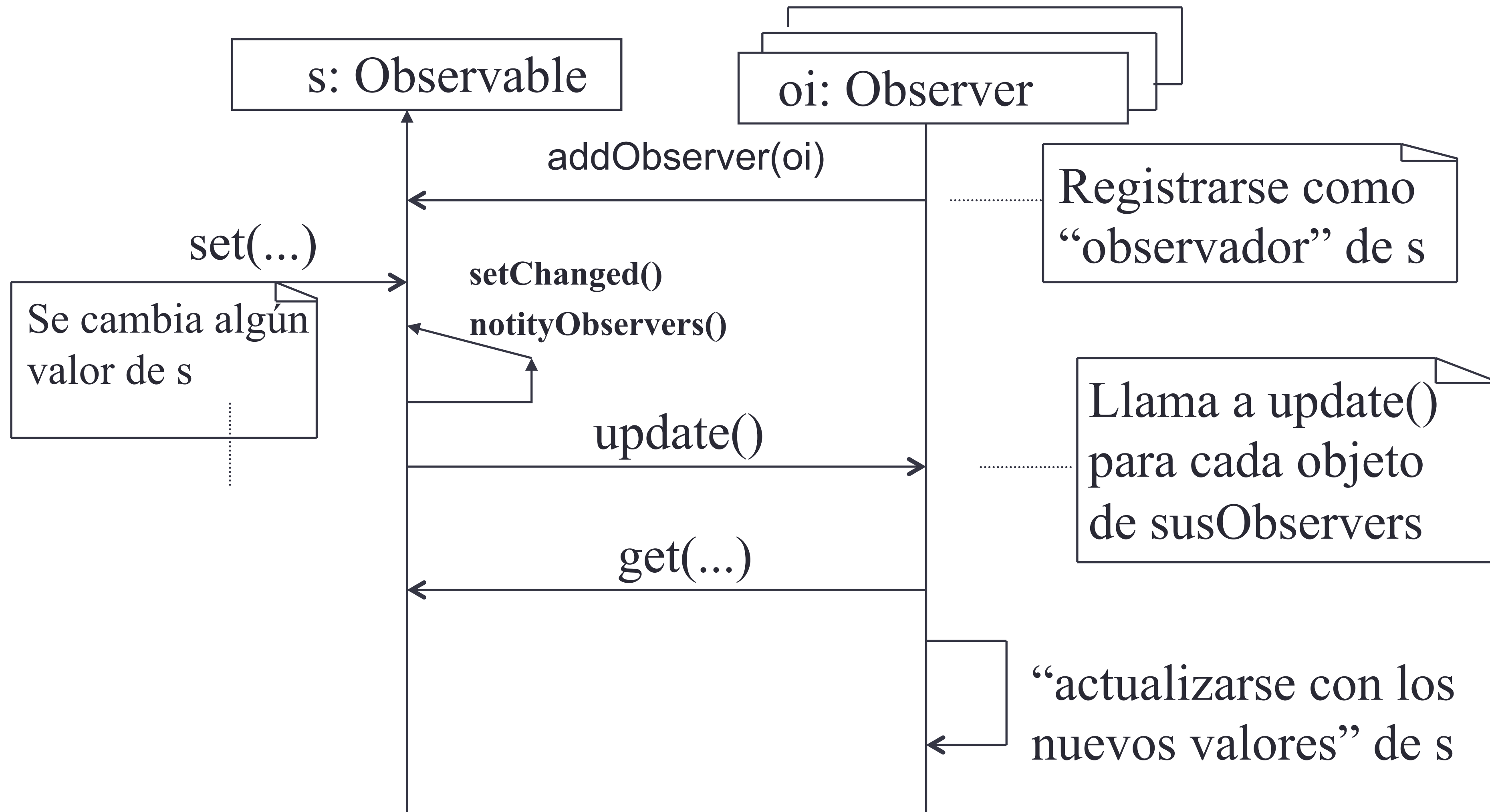
1

Se implementa añadiendo el atributo **Observable suSubject** en **ObsConcreta**



...

Patrón OBSERVER
Diagrama de interacción



Patrón OBSERVER

La clase *Observable* y la interfaz *Observer de java.Util*

La clase Observable

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

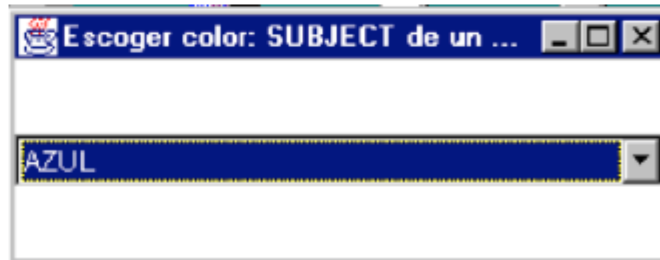
La interfaz Observer

update()	Este método es invocado cuando el objeto observado ha cambiado

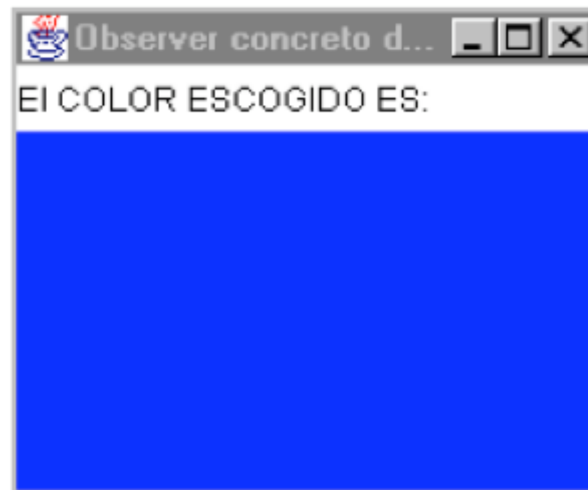
Patrón OBSERVER

Ejemplo: Enunciado

- Se desea realizar una aplicación que cuando se seleccione un color en el desplegable (es decir, AZUL)

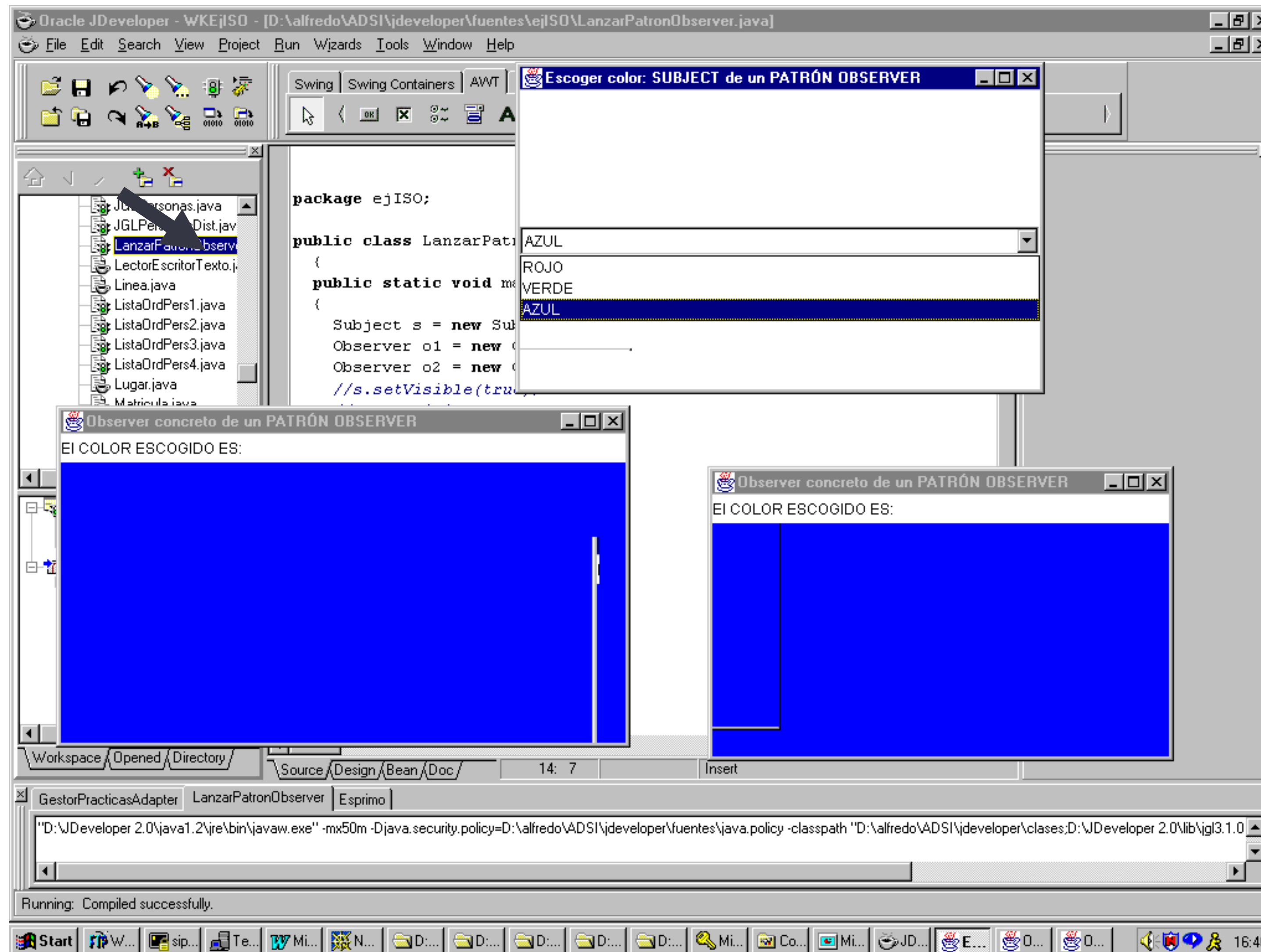


- Cambie el color de sus 2 ventanas



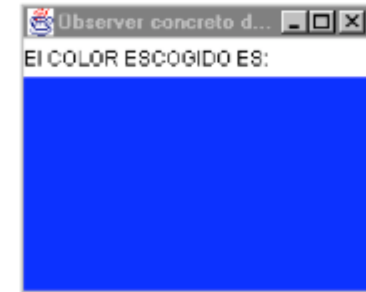
Patrón OBSERVER

Ejemplo en ejecución



Patrón OBSERVER

La vista. Implementa interfaz *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);
  }
 }
```

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

Patrón OBSERVER

Ejemplo 2. El modelo. Extiende la clase Observable

```
public class Modelo extends Observable {
  String elColor; //Este es el modelo. Guarda el color seleccionado
  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) {
                            //se guarda el color seleccionado
                            elColor=choice1.getSelectedItem();
                           // Cambia el estado del objeto
                            setChanged();
                           // Notifica a sus observadores
                            notifyObservers(); }}
  }
          public String getColor() {
            // Se devuelve el color seleccionado
            return elColor;
          }
}
```

Patrón OBSERVER

El programa que crea y relaciona las vistas y el modelo

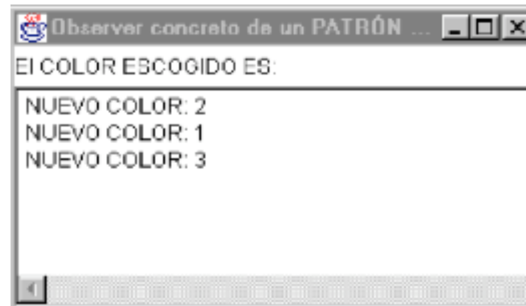
```
Observable m = new Modelo();
Observer o1 = new ObserverQuePinta(m);
Observer o2 = new ObserverQuePinta(m);
```

CREACIÓN DE OBJETOS
OBSERVABLE Y OBSERVER

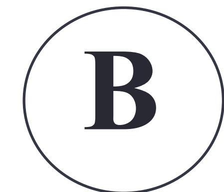
Patrón OBSERVER

Adaptabilidad de la solución. Posibles cambios

- Se quiere añadir un nuevo OBSERVER que en vez de pintar escriba un número para cada color (1 si ROJO, 2 si VERDE, 3 si AZUL).



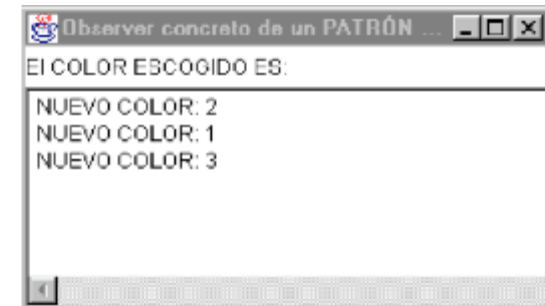
- Se quiere cambiar el SUBJECT para escoger el color pinchando en un Checkbox en vez de escogerlo de un Choice.



Patrón OBSERVER

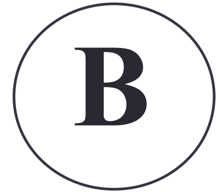
Extensión A. Añadir un nuevo *Observer*

```
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");
  }
}
```



Patrón OBSERVER

Extensión B. Cambiar el *Observable*




```
public class Modelo extends Observable {
  String elColor //Este sigue siendo el Modelo
  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) {
                elColor=checkboxGroup1.getSelectedCheckbox().getLabel();
                setChanged();
                notifyObservers(); }    });
        checkbox2.setLabel("VERDE"); // y checkbox3.setLabel("AZUL");...
  public String getColor() {
    return elColor;
  }
}
```

Patrón OBSERVER

Versión final con varios *Observers* y *Observables*