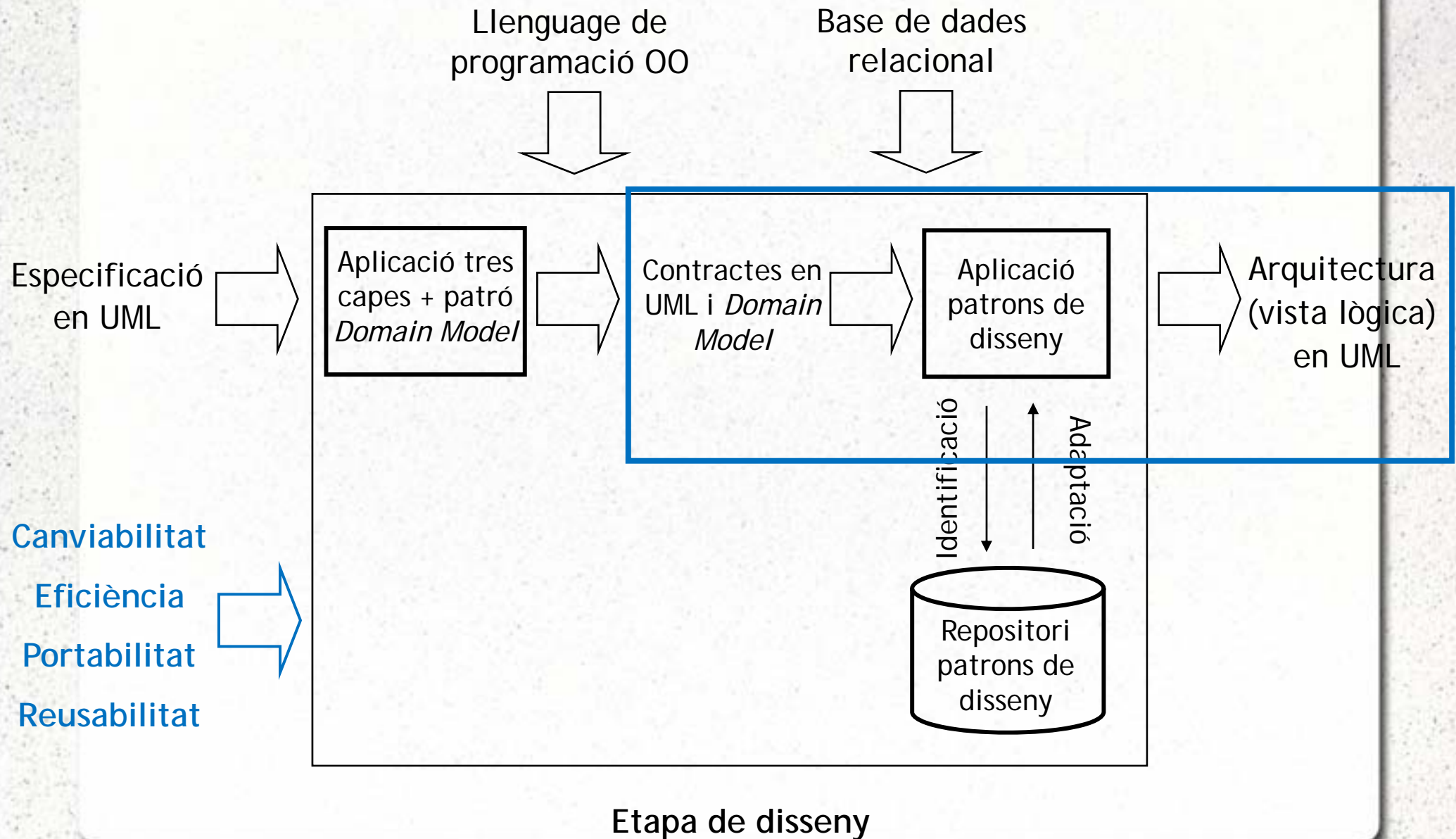


# Projecte de Programació

## **Patrones de Diseño en Java**

# Etapa de disseny a l'assignatura IES



## Concepto de patrón:

"Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice"

Christopher Alexander, arquitecto (1977)

- Es difícil encontrar un sistema orientado a objetos que no use al menos un par de patrones; en particular, los grandes sistemas usan muchos de ellos
- Los diseñadores expertos en sistemas orientados a objetos no resuelven cada problema desde cero: reúsan soluciones que han funcionado bien anteriormente -> un diseñador familiarizado con estos patrones puede aplicarlos directamente a problemas de diseño sin necesidad de redescubrirlos
- El uso de patrones vuelve los diseños flexibles, elegantes y reusables



# Patrones de Diseño

El catálogo de patrones es enorme y varían en granularidad y nivel de abstracción. Según su **propósito**, se pueden clasificar en:

- Patrones creadores (*creational*): conciernen al proceso de creación de objetos
- Patrones estructurales (*structural*): tratan con la composición de clases u objetos
- Patrones de comportamiento (*behavioral*): caracterizan la forma en que objetos o clases interactúan y distribuyen la responsabilidad

# Patrón Singleton

## *Creational Pattern*

**Propósito:** Asegurarse de que una clase tiene sólo una instancia, y proporcionar un punto de acceso global a ésta

**Solución:** Hacer a la propia clase responsable de registrar su instancia única. La clase intercepta peticiones para crear nuevas instancias y proporciona una forma de acceder a la instancia en cuestión

# Patrón Singleton

## Implementación 1

```
package singleton.demo;

public class Singleton {

    // la inicialización podría suprimirse (default)
    private static Singleton instance = null;

    private Singleton() {...}

    public static Singleton getInstance() {

        //inicialización "lazy"
        if (instance == null) {
            instance = new Singleton();
        }
        return instance;
    }
}
```



# Patrón Singleton

## Implementación 2

```
package singleton.demo;

public class Singleton {

    // inicialización "eager"
    private static Singleton instance = new Singleton();

    // la constructora es privada para prohibir la libre
    // instanciación
    private Singleton() {...}

    public static Singleton getInstance() {

        return instance;
    }
}
```

# Patrón Singleton

## Implementación 3

```
package singleton.demo;

public class Singleton {

    // la inicialización podría suprimirse (default)
    private static Singleton instance = null;

    private Singleton() {...}

    private static class SingletonHelper {

        private static final Singleton instance = new Singleton();
    }

    public static Singleton getInstance() {

        return SingletonHelper.instance;
    }
}
```



# Patrón Decorador

## *Structural Pattern*

**Propósito:** Asociar responsabilidades adicionales a un objeto de forma dinámica.

Una forma de añadir responsabilidades es la herencia, pero no es flexible, ya que obliga a añadirlas de forma estática (se añaden a la clase entera). Los decoradores ofrecen una alternativa flexible a las subclases para extender la funcionalidad.

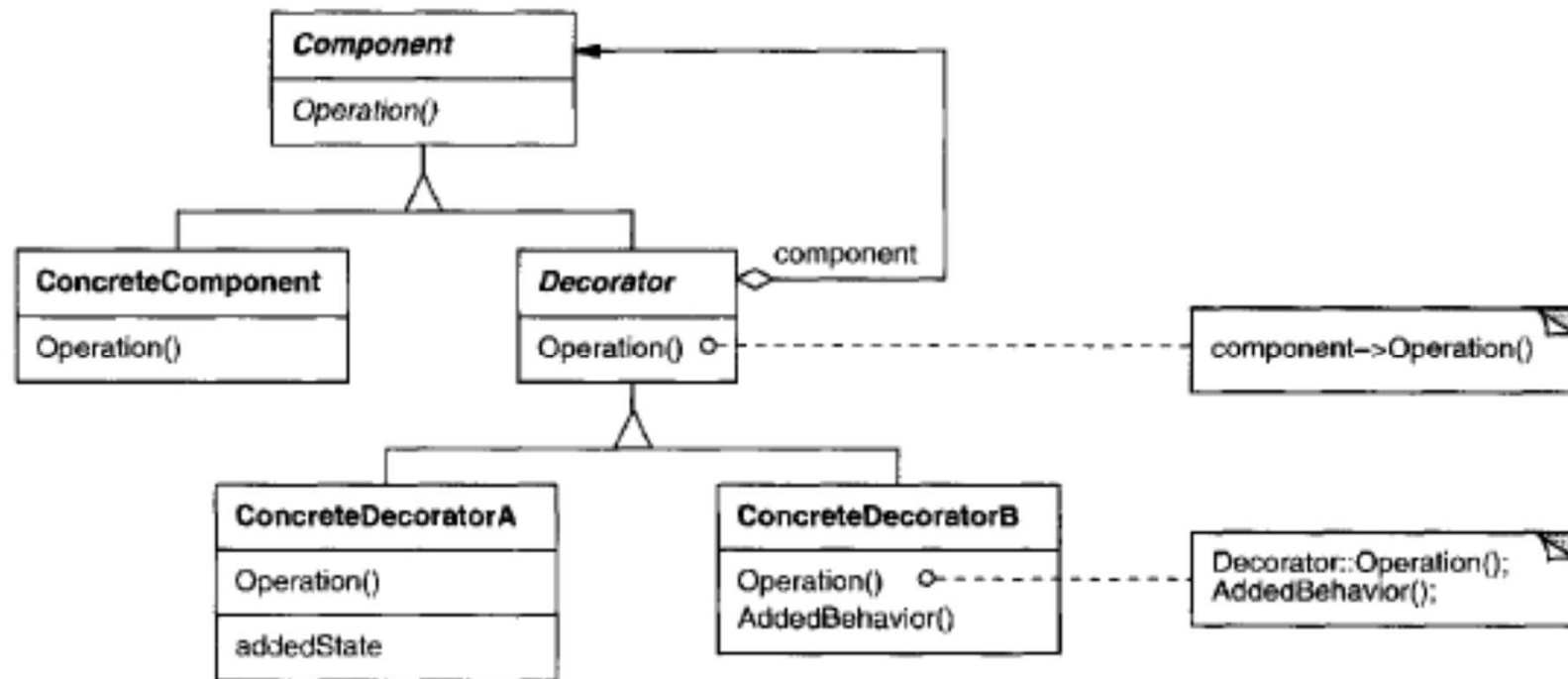
# Patrón Decorador

**Ejemplo:** Una interfaz gráfica debería permitir añadir propiedades (p.ej. bordes) y/o comportamientos (p.ej. *scrolling*) a cada uno de sus componentes de forma flexible.

**Solución:** Envolver el componente en otro objeto que añada la propiedad/comportamiento, el Decorador.

- El decorador se ajusta a la interficie del componente -> la presencia del decorador es transparente a los clientes del componente
- El decorador reenvía las peticiones al componente (eventualmente ejecutando acciones adicionales antes o después)
- La transparencia permite anidar decoradores recursivamente -> número ilimitado de responsabilidades añadidas

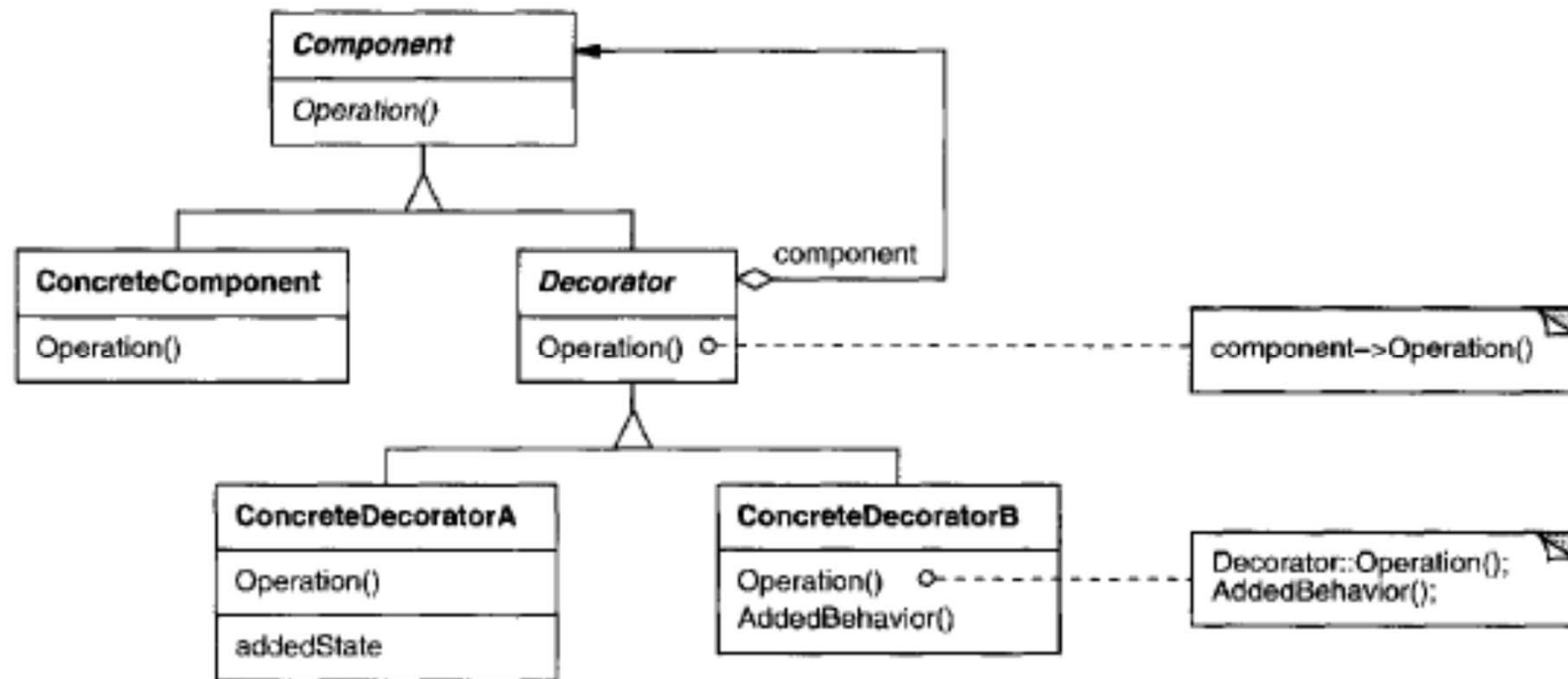
# Patrón Decorador



- **Componente:** define la interficie de aquellos objetos a los que se puede añadir responsabilidades de forma dinámica
- **ComponenteConcreto:** define un objeto al que se le pueden asociar responsabilidades adicionales



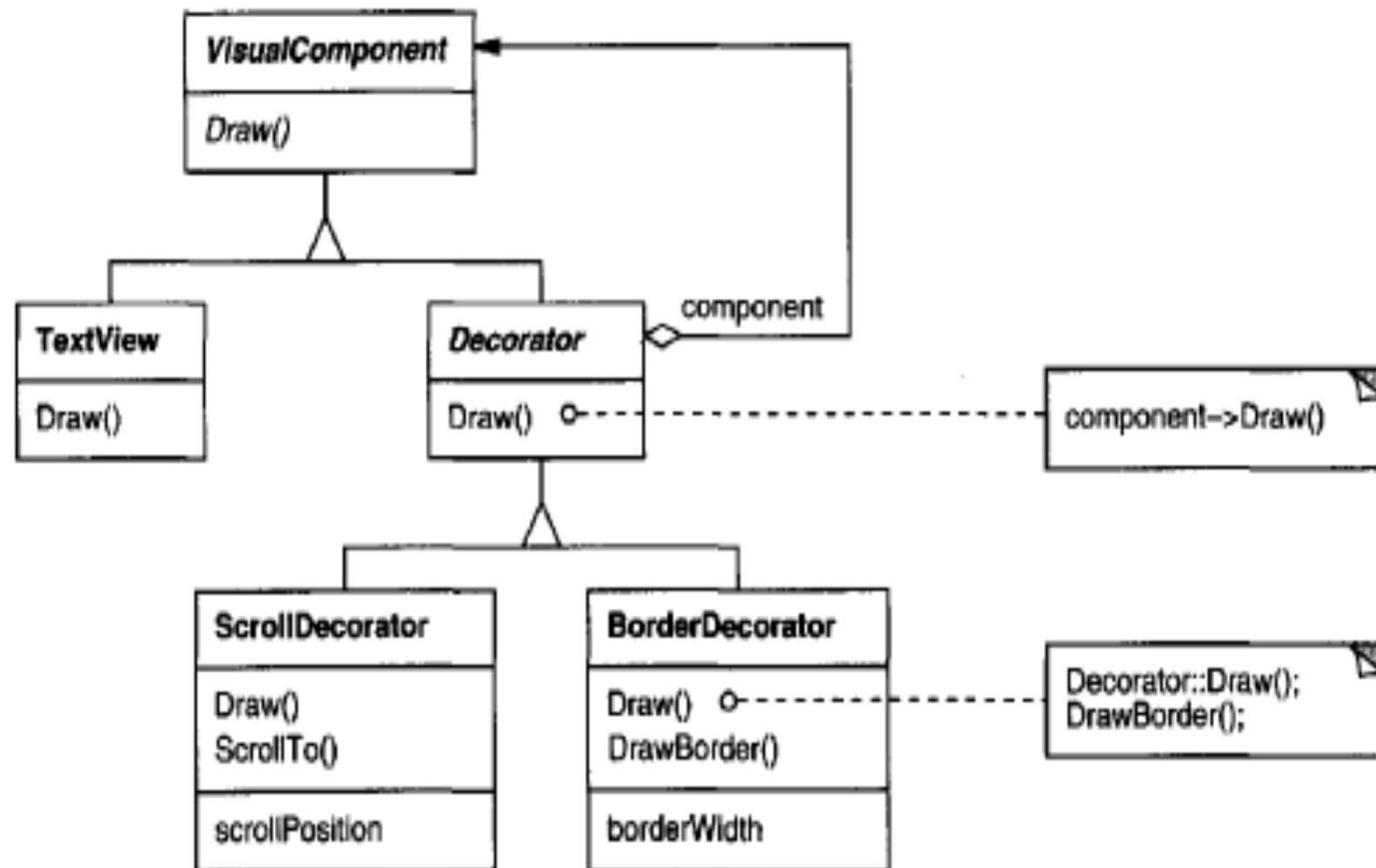
# Patrón Decorador



- **Decorator**: mantiene una referencia un objeto Componente y define una interficie que se ajusta a la del Componente
- **DecoratorConcreto**: añade responsabilidades al Componente

# Patrón Decorador

**Ejemplo:** Una interfaz gráfica quiere permitir añadir propiedades (bordes) y comportamientos (*scrolling*) a sus ventanas de texto de forma flexible



# Patrón Decorador

## Implementación

```
// Si sólo tuviera operaciones diferidas, podría ser una
interface
abstract class VisualComponent {

    public abstract void Draw();
    public abstract void Resize();
    ...
}

// TextComponent: closed for modification, open for
extension
class TextComponent extends VisualComponent {

    public void Draw() {...}

    public void Resize() {...}

    ...
}
```



# Patrón Decorador

## Implementación

```
abstract class Decorator extends VisualComponent {  
  
    // alternativas: private, protected final  
    protected VisualComponent component;  
  
    public Decorator(VisualComponent c) {  
        component = c;  
    }  
  
    //implementación por defecto  
    public void Draw() {  
        component.Draw();  
    }  
  
    //implementación por defecto  
    public void Resize() {  
        component.Resize();  
    }  
    ...  
}
```

# Patrón Decorador

## Implementación

```
class BorderDecorator extends Decorator {  
  
    private int width;  
  
    public BorderDecorator(VisualComponent c, int borderWidth) {  
        super(c);  
        width = borderWidth;  
    }  
  
    public void Draw() {  
        super.Draw();  
        DrawBorder(width);  
    }  
  
    private void DrawBorder (int w) {...}  
  
    ...  
}
```

Idem clase ScrollDecorator...

# Patrón Decorador

## Implementación

Un ejemplo de uso:

```
class DecoratorPatternEx {  
  
    public static void main (String[] args) {  
  
        TextComponent c = new TextComponent();  
  
        // "decoro" la ventana con un scroll  
        ScrollDecorator sd = new ScrollDecorator(c);  
        sd.Draw();  
  
        // "decoro" la ventana con scroll con un borde  
        BorderDecorator bd = new BorderDecorator(sd, 5);  
        bd.Draw();  
  
        // asumiendo que la clase Window tiene una operación SetContents  
        // para poner un VisualComponent en un objeto Window  
        Window w = new Window();  
        w.SetContents(bd);  
  
        ...  
    }  
}
```



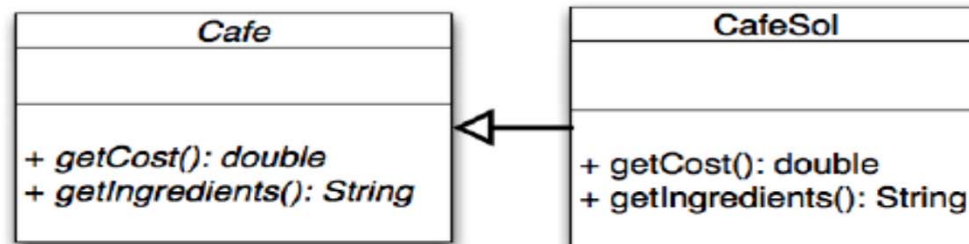
# Patrón Decorador

## **Cuándo aplicarlo:**

- Para asociar responsabilidades a objetos individuales de forma dinámica y transparente (sin afectar a otros objetos)
- Para responsabilidades que puedan ser retiradas
- Para casos en que usar subclasses no es posible:
  - Gran número de posibles extensiones independientes-> explosión de subclasses para soportar cada combinación
  - No es posible extender la clase (por definición de ésta, por herencia múltiple, ...)

# Patrón Decorador - Ejercicio

Tenim les següents classes:



on **Cafe** és una classe abstracta amb un parell de mètodes abstractes (`getCost()` i `getIngredients()`). Volem ampliar aquest domini amb més classes (per exemple, les classes **Llet**, **Crema**, **Xocolata** i d'altres) de manera que sigui senzill afegir ingredients al cafè. La intenció és fer servir el sistema de manera que un programa com aquest:

```
public class Main {
    public static void main(String[] args) {
        Cafe c = new CafeSol();
        System.out.println("Cost: " + c.getCost() +
                           "; Ingredients: " + c.getIngredients());

        c = new Llet(c);
        System.out.println("Cost: " + c.getCost() +
                           "; Ingredients: " + c.getIngredients());

        c = new Xocolata(c);
        System.out.println("Cost: " + c.getCost() +
                           "; Ingredients: " + c.getIngredients());

        c = new Crema(c);
        System.out.println("Cost: " + c.getCost() +
                           "; Ingredients: " + c.getIngredients());

        c = new Xocolata(c);
        System.out.println("Cost: " + c.getCost() +
                           "; Ingredients: " + c.getIngredients());
    }
}
```



# Patrón Decorador - Ejercicio

hauria de generar la següent sortida:

```
Cost: 1.0; Ingredients: Cafe
```

```
Cost: 1.5; Ingredients: Cafe, Llet
```

```
Cost: 1.7; Ingredients: Cafe, Llet, Xocolata
```

```
Cost: 2.4; Ingredients: Cafe, Llet, Xocolata, Crema
```

```
Cost: 2.6; Ingredients: Cafe, Llet, Xocolata, Crema, Xocolata
```

Es valorarà molt positivament que el codi es reutilitzi al màxim.

1.1 (1 punt).- Implementa les classes `Cafe` i `CafeSol`.

1.2 (3 punts).- Afegeix al diagrama les classes `Llet`, `Crema` i `Xocolata`. Fes-ho pensant que la redundància en el codi ha de ser mínima. Si penses que fan falta, fes servir classes auxiliars. Cal ser precís en l'especificació de les relacions entre les classes (poseu-les totes).

1.3 (3 punts).- Implementa les classes afegides, és a dir, les classes `Llet`, `Crema` i `Xocolata` i les classes auxiliars que hagi utilitzat.



# Patrón Estado

## *Behavioral Pattern*

**Propósito:** Permitir a un objeto alterar su comportamiento cuando su estado interno cambie. Parecerá que el objeto cambie de clase (sin hacerlo).

La tecnología actual no permite que un objeto cambie dinámicamente de subclase, y las estructuras condicionales para tratar el comportamiento en función del estado no son deseables pues añaden complejidad y/o duplican el código

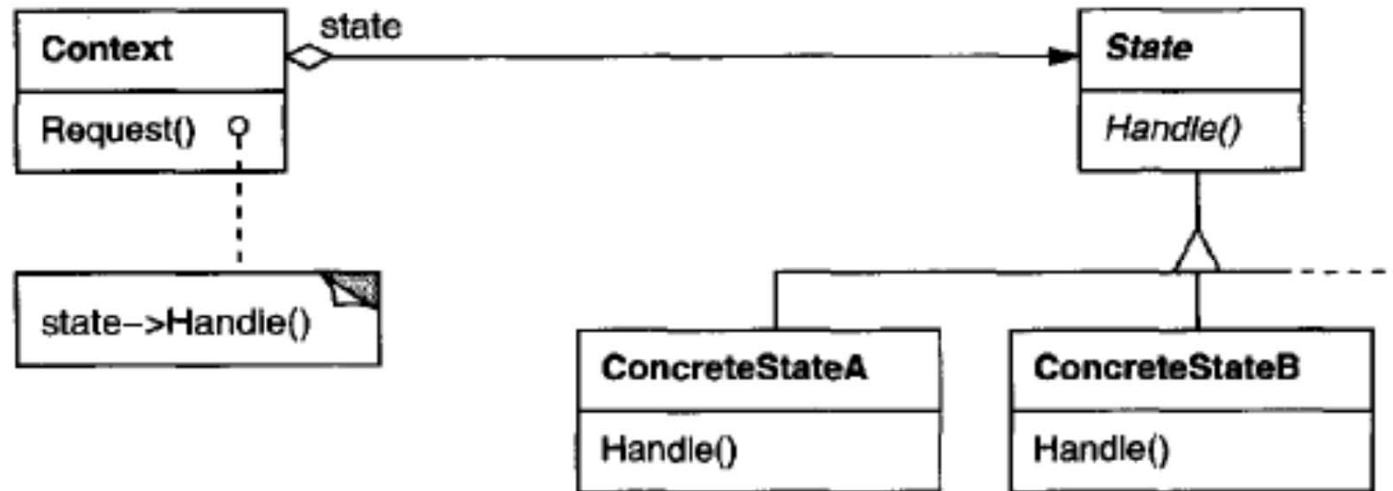
# Patrón Estado

**Ejemplo:** Una conexión de red puede estar en 3 estados (establecida, escuchando o cerrada). Cuando recibe una petición, debería responder de forma diferente según su estado actual.

**Solución:** Introducir una clase abstracta que represente los estados de la conexión, Estado.

- Estado declara una interficie común a todas las clases que representan diferentes estados concretos
- Las subclases de Estado implementan el comportamiento dependiente de cada estado concreto
- La clase contexto (la conexión) mantiene un objeto Estado, que será una instancia de una de sus subclases, y delega todas las peticiones dependientes del estado a este objeto
- Cuando el contexto cambia de estado, reemplaza el contenido del objeto Estado por la subclase correspondiente al nuevo

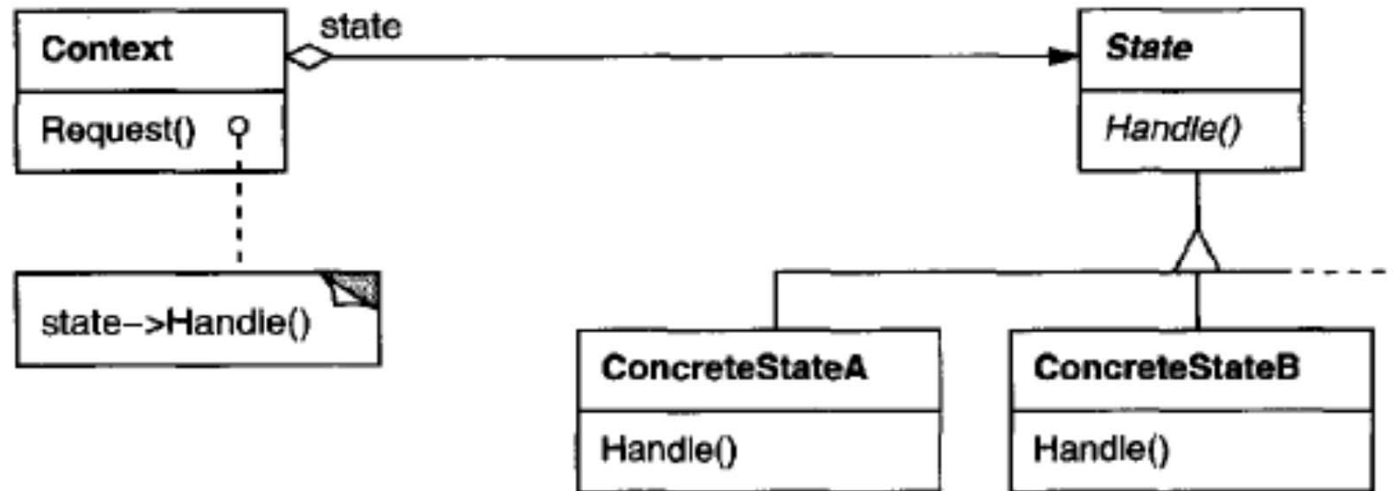
# Patrón Estado



- **Contexto**: define la interficie que usarán los clientes y mantiene una instancia de un Estado Concreto que define el estado actual. Podría pasarse a sí mismo como parámetro al Estado para que éste acceda al Contexto si se necesita.
- **Estado**: define una interficie que encapsula el comportamiento asociado con un estado particular del Contexto



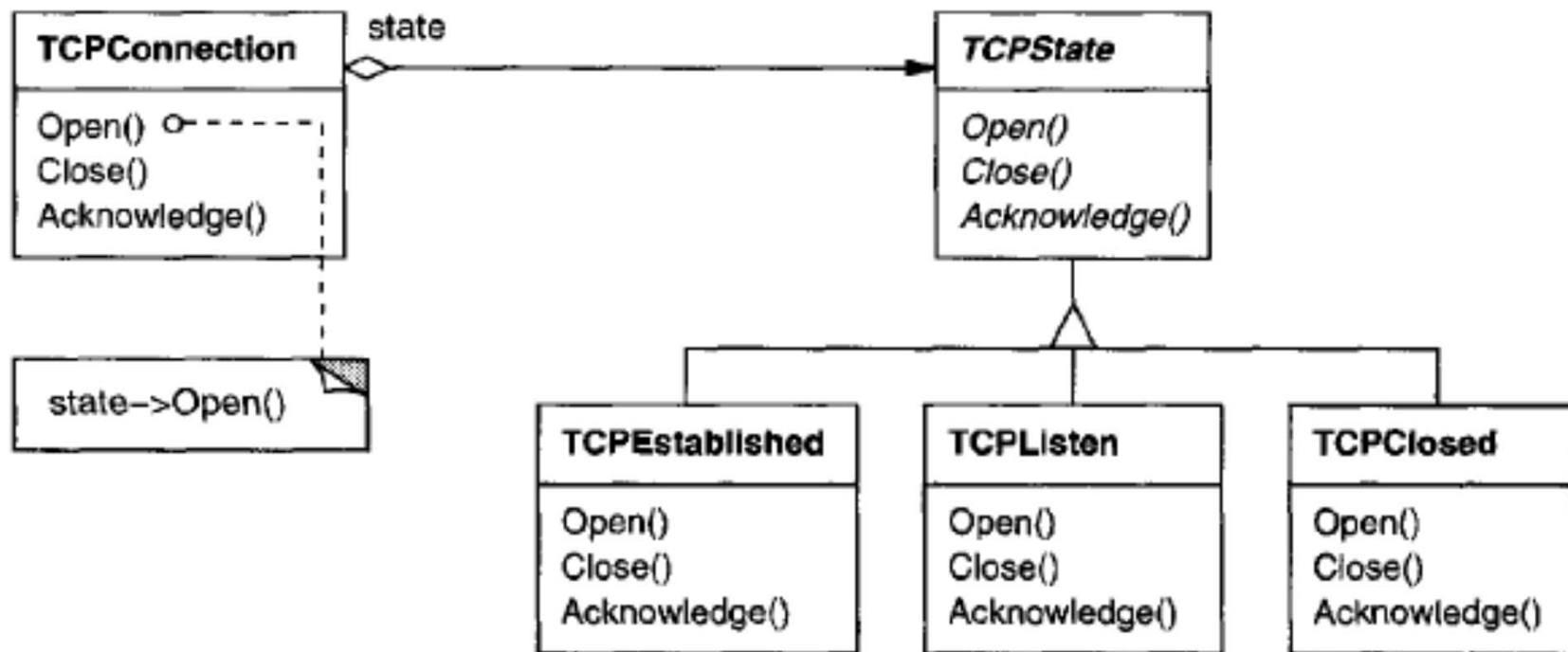
# Patrón Estado



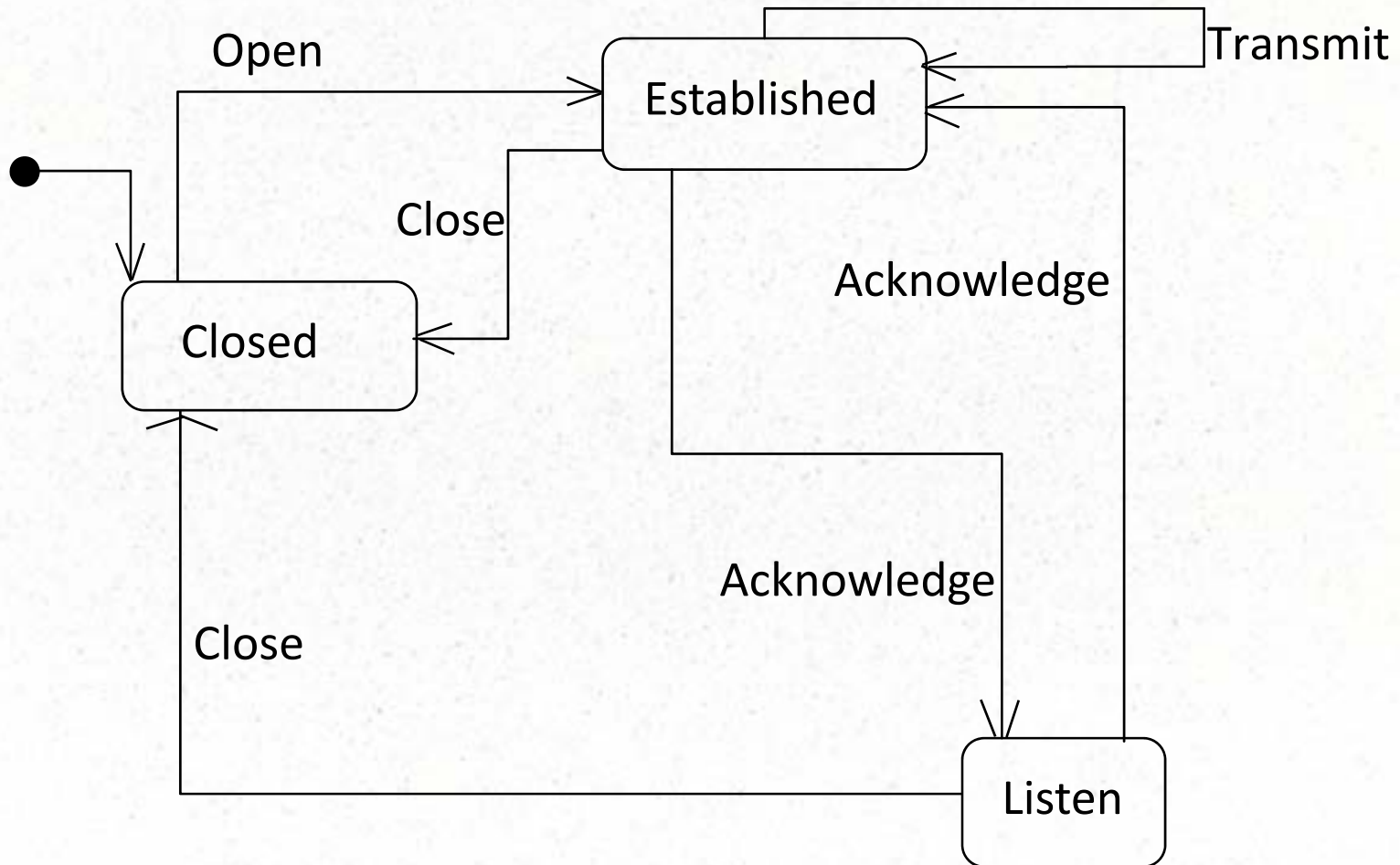
- **Estados Concretos:** cada subclase implementa el comportamiento asociado a un estado del Contexto. Tanto el Contexto como los Estados Concretos pueden decidir qué estado sucede a otro y bajo qué circunstancias -> para añadir nuevos estados y transiciones, basta con definir nuevas subclases de Estados Concretos

# Patrón Estado

**Ejemplo:** Una conexión de red puede estar en 3 estados (establecida, escuchando o cerrada). Cuando recibe una petición, debería responder de forma diferente según su estado actual.



# Patrón Estado





# Patrón Estado

## Implementación

```
class TCPConnection {  
  
    private TCPState state;  
  
    public TCPConnection() {  
        state = new TCPClosed();  
    }  
  
    public void Open() {  
        state.Open(this);  
    }  
  
    public void Close() {  
        state.Close(this);  
    }  
  
    public void Acknowledge() {  
        state.Acknowledge(this);  
    }  
  
    public void Transmit() {  
        state.Transmit(this);  
    }  
  
    public void ChangeState(TCPState s) {  
        this.state = s;  
    }  
}
```

# Patrón Estado

## Implementación

```
abstract class TCPState {  
  
    public void Open(TCPConnection c) {  
        System.out.println("Transition Error");  
    }  
  
    public void Close(TCPConnection c) {  
        System.out.println("Transition Error");  
    }  
  
    public void Acknowledge(TCPConnection c) {  
        System.out.println("Transition Error");  
    }  
  
    public void Transmit(TCPConnection c) {  
        System.out.println("Transition Error");  
    }  
  
    protected void ChangeState(TCPConnection c, TCPState s) {  
        c.ChangeState(s);  
    }  
}
```

# Patrón Estado

## Implementación

```
class TCPClosed extends TCPState {  
  
    public void Open(TCPConnection c) {  
        // acciones que se hayan de ejecutar antes de establecer la  
        conexión  
        ...  
        ChangeState(c, new TCPEstablished());  
    }  
}
```

ALTERNATIVA: eliminar la operación ChangeState de TCPState:

```
class TCPClosed extends TCPState {  
  
    public void Open(TCPConnection c) {  
        ...  
        c.ChangeState(new TCPEstablished());  
    }  
}
```



# Patrón Estado

## Implementación

```
class TCPEstablished extends TCPState {  
  
    public void Close(TCPConnection c) {  
        // acciones que se han de ejecutar antes de cerrar la  
        conexión  
        ...  
        ChangeState(c,new TCPClosed());  
    }  
  
    public void Acknowledge(TCPConnection c) {  
        // acciones que se han de ejecutar antes de realizar la  
        confirmación  
        ...  
        ChangeState(c,new TCPListen());  
    }  
  
    public void Transmit(TCPConnection c) {  
        // acciones que se han de ejecutar para transmitir  
        ...  
        ChangeState(c,this);  
    }  
}
```

# Patrón Estado

## Implementación

```
class TCPListen extends TCPState {  
  
    public void Close(TCPConnection c) {  
        // acciones que se han de ejecutar antes de cerrar la  
        conexión  
        ...  
        ChangeState(c,new TCPClosed());  
    }  
  
    public void Acknowledge(TCPConnection c) {  
        // acciones que se han de ejecutar antes de efectuar la  
        confirmación  
        ...  
        ChangeState(c,new TCPEstablished());  
    }  
}
```

# Patrón Estado

## Implementación

Un ejemplo de uso:

```
class StatePatternEx {  
  
    public static void main (String[] args) {  
  
        TCPConnection c = new TCPConnection();  
  
        // El estado inicial de la conexión c es Closed  
        c.Open();  
        // El estado de c cambia a Established  
        c.Transmit();  
        // El estado de c sigue Established  
        c.Transmit();  
        // El estado de c sigue Established  
        c.Acknowledge();  
        //El estado de c cambia a Listen  
        c.Close();  
        // El estado de c cambia a Closed  
    }  
}
```



# Patrón Estado

## **Cuándo aplicarlo:**

- Cuando el comportamiento de un objeto depende de su estado, y éste ha de cambiar en tiempo de ejecución
- Cuando las operaciones de la clase tienen múltiples instrucciones condicionales que dependen del estado del objeto. Frecuentemente, la misma estructura condicional se repetirá en varias operaciones -> el patrón estado pone cada rama del condicional en una clase individual

# Patrón Estado - Ejercicio

Volem programar el funcionament d'una màquina de vending que admet monedes de 5 cèntims i 10 cèntims i proporciona llaunes, que valen 15 cèntims. Aquest programa simula el funcionament de la màquina:

```
public static void main (String[] args) {
    Scanner in = new Scanner(System.in);
    MaquinaVending m = new MaquinaVending();

    while (true) {
        if (m.get_llauna()) {
            System.out.println("UNA LLAUNA");
        }
        System.out.println("Saldo: " + m.get_saldo());
        System.out.println("Introdueix moneda: ");
        int num = in.nextInt();
        if (num == 5) {
            m.add_5_saldo();
        } else if (num == 10) {
            m.add_10_saldo();
        }
    }
}
```



# Patrón Estado - Ejercicio

la interacció amb el programa té una sortida similar a:

```
Saldo: 0
Introdueix moneda: 5
Saldo: 5
Introdueix moneda: 10
UNA LLAUNA
Saldo: 0
Introdueix moneda: 10
Saldo: 10
Introdueix moneda: 10
UNA LLAUNA
Saldo: 5
Introdueix moneda: ...(interrumpim manualment el bucle amb ^C)
```

Si us fixeu, la màquina deixa anar una llauna *de seguida* que té 15 cèntims (o més), no s'espera a que se li reclami la llauna. Cal que dissenyeu i implementeu la classe **MaquinaVending**, més totes les classes que penseu que us poden fer falta. La resposta hauria de ser *un diagrama de classes i la implementació d'aquest diagrama*. Hi ha, però una restricció MOLT important: **No podeu fer servir cap mena d'estructura condicional en el vostre codi** (no `if`'s, no `switch`, etc)



# Patrón Observador

## *Behavioral Pattern*

**Propósito:** Definir una dependencia uno-a-muchos entre objetos de forma que cuando un objeto cambie su estado interno, todos sus dependientes sean notificados y actualizados automáticamente.

Un típico efecto lateral de particionar un sistema en conjuntos de clases que cooperan entre sí es la necesidad de mantener la consistencia entre los objetos relacionados. Si lo hacemos vía acoplamiento, se reducirá la reusabilidad de las clases.

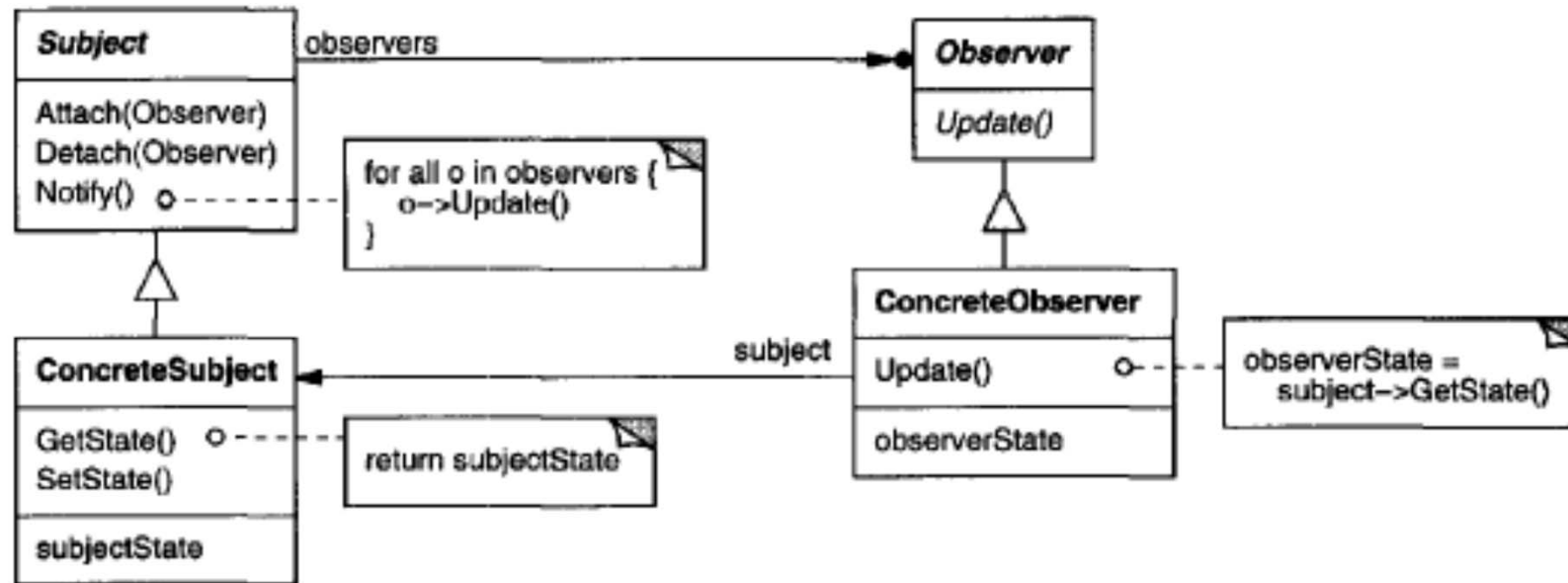
# Patrón Observador

**Ejemplo:** Una interfaz de usuario puede estar conectada a la lógica de la aplicación, de forma que cuando un usuario hace una consulta, el resultado se refleja en la interfaz. Si los datos de la aplicación sufrieran un cambio, se tendría que notificar a la interfaz para que actualice su pantalla de acuerdo con el cambio.

**Solución:** Definir una clase Sujeto que puede tener cualquier número de clases dependientes, los Observadores.

- Los observadores se registran como interesados en "seguir" al sujeto
- Los observadores se desasocian del sujeto cuando pierden interés en él
- Cuando se produce un cambio de estado en el sujeto, éste notifica a todos sus observadores cuál es el cambio que se ha producido

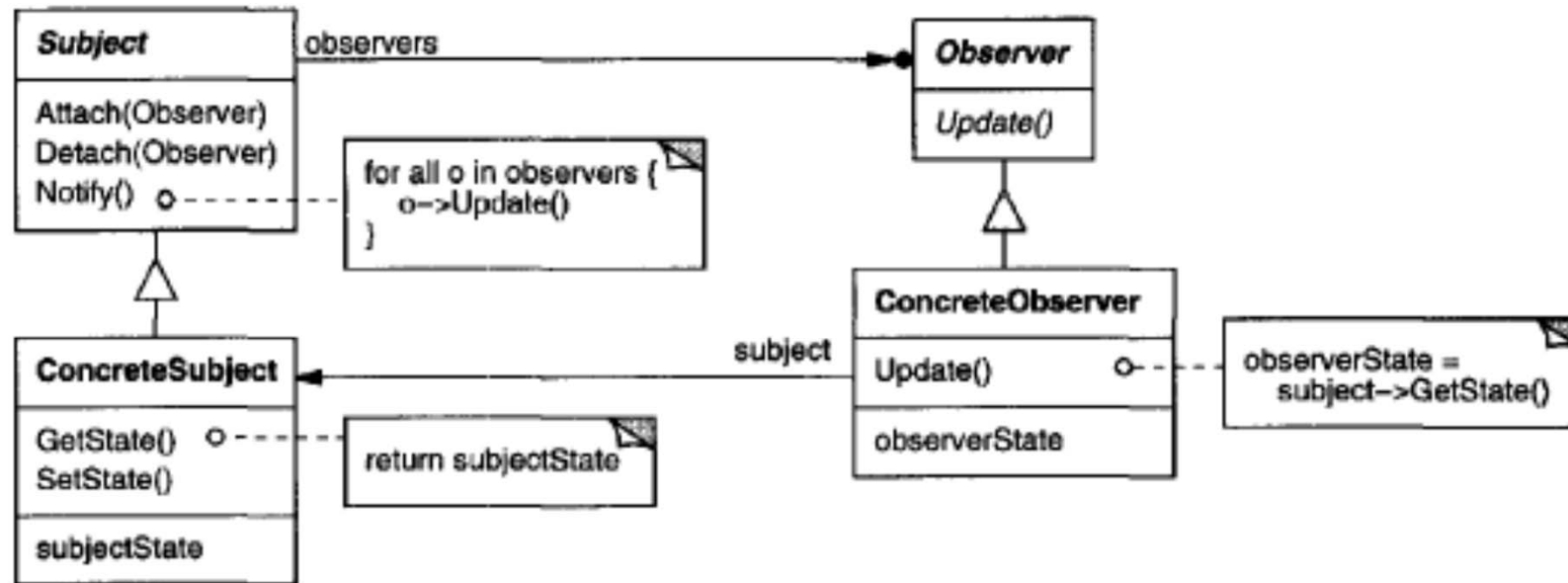
# Patrón Observador



- **Sujeto**: conoce a sus observadores. Define una interficie que permite asociar y desasociar objetos Observador (puede haber cualquier número de éstos).
- **Observador**: define una interficie para actualizar los objetos que deberían ser notificados de cambios en el Sujeto.



# Patrón Observador



- **Sujeto Concreto**: guarda el estado de interés para sus Observadores Concretos y envía una notificación a éstos cuando este estado cambia.
- **Observador Concreto**: guarda un estado consistente con el del Sujeto Concreto (podría mantener una referencia a éste) e implementa la interficie actualizadora del Observador para mantener su estado consistente con el del Sujeto Concreto.

# Patrón Observador

## Implementación

```
abstract class Subject {  
  
    private List<Observer> observersList = new ArrayList<Observer>();  
  
    void Attach(Observer o) {  
        observersList.add(o);  
    }  
  
    void Detach(Observer o) {  
        observersList.remove(o);  
    }  
  
    void Notify() {  
        for (Observer obs : observersList) {  
            obs.Update(this);  
        }  
    }  
}
```

# Patrón Observador

## Implementación

```
// Ejemplo de ConcreteSubject: clase que mantiene y actualiza la
hora actual
class ClockTimer extends Subject {

    //estado del sujeto concreto:
    private int hour;
    private int minute;
    private int second;

    int GetHour() {return hour;}
    int GetMinute() {return minute;};
    int GetSecond() {return second};

    // operación llamada por un timer interno del sistema
    void Tick() {
        // se actualiza el estado interno del timer
        ...
        Notify();
    }

    ...
}
```



# Patrón Observador

## Implementación

```
// podría haber sido una clase abstracta
interface Observer {

    //actualiza el estado del observador concreto de acuerdo con el del sujeto
    void Update(Subject s);
}

// Ejemplo de ConcreteObserver: reloj digital que muestra la hora
// eventualmente heredaría de algún componente gráfico de una GUI
class DigitalClock implements Observer {

    DigitalClock(ClockTimer ct) {
        ct.attach(this);
    }

    // muestra el reloj
    private void Draw(int h, int m, int s) {... }

    public void Update(Subject ct) {
        int hour = ((ClockTimer)ct).getHour();
        int minute = ((ClockTimer)ct).getMinute();
        int second = ((ClockTimer)ct).getSecond();
        Draw(hour, minute, second);
    }
    ...
}
```

# Patrón Observador

## **Cuándo aplicarlo:**

- Cuando un cambio en un objeto requiere cambiar otros, y no sabemos cuántos objetos se necesita cambiar
- Cuando un objeto debería ser capaz de notificar a otros sin hacer suposiciones sobre quiénes son estos otros objetos (no queremos crear acoplamientos)

El patrón observador permite cambiar sujetos y objetos independientemente. Se puede reusar sujetos sin reusar observadores, y viceversa. Permite añadir observadores sin modificar el sujeto ni otros observadores. Permite que sujetos y observadores estén en capas diferentes, pues su acoplamiento es abstracto y mínimo

# Bibliografia:

Design Patterns: Elements of Reusable Object-Oriented Software  
Gamma, E.; Helm, R.; Johnson, R.; Vlissides, J.  
Addison-Wesley 2009

Java Design Patterns  
Vaskaran Sarcar  
Apress 2016