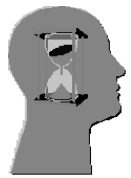
Programación Orientada a Objetos

Tema 6: Patrones de Diseño

Eduardo Mosqueira Rey



LIDIA
Laboratorio de Investigación y desarrollo en Inteligencia Artificial

Departamento de Computación
Universidade da Coruña, España

Objetivos

- **Introducir a los alumnos en los patrones de diseño de software.**
- **Explicar los distintos tipos de patrones y mostrar ejemplos de aquellos patrones más sencillos y representativos de cada tipo.**
- **Aprender a reconocer los patrones estudiados y aprender a aplicarlos a problemas concretos mediante su implementación en un lenguaje orientado a objetos como Java.**

Índice

1. Introducción
2. Patrones Creacionales
3. Patrones Estructurales
4. Patrones de Comportamiento

Índice

1. **Introducción**
   - **Historia**
   - **Tipos de patrones**

Introducción
Historia

- **Objetivo**
  - **Crear una colección de literatura que ayude a resolver problemas recurrentes encontrados durante el desarrollo del software.**
  - **Las soluciones a los problemas deben ser efectivas y reusables.**
    - **Efectivas en el sentido de que ya han resuelto el problema con anterioridad satisfactoriamente,**
    - **Reusable en el sentido de que pueden aplicarse a un abanico de problemas de diseño en distintas circunstancias.**
  - **Los patrones ayudan a la creación de un lenguaje común que permita comunicar la experiencia con estos problemas y sus soluciones.**
- **Definición**
  - **“Un patrón es una regla con tres partes que expresa la relación entre un determinado contexto, un problema y una solución.” Christopher Alexander (arquitecto)**

Introducción
Historia

- **1977 – Christopher Alexander**
  - **El término patrón deriva de los trabajos del arquitecto Christopher Alexander sobre planificación urbanística y arquitectura de edificaciones.**
  - **En su libro “A Pattern Language” defiende el uso de patrones en la arquitectura pero sus ideas son aplicables a otras disciplinas, como el desarrollo del software.**
  - **Alexander describía en sus trabajos un lenguaje de patrones y un método para aplicar estos patrones para conseguir diseños que garantizasen la calidad de vida de sus usuarios.**
- **1987 – Ward Cunningham y Kent Beck**
  - **Decidieron aplicar algunas de las ideas de Alexander para desarrollar un pequeño lenguaje compuesto por cinco patrones que sirviera de guía para los desarrolladores novatos de Smalltalk.**
  - **El resultado fue la publicación “Using Pattern Languages for Object-Oriented Programs” en el congreso OOPSLA’87.**

Introducción
Historia

- **1991 – Jim Coplien**
  - **Compiló un catálogo de lo que él llamó “Idiomas” para el lenguaje C++.**
  - **Este catálogo fue publicado en 1991 bajo el título “Advanced C++ Programming Styles and Idioms”.**
- **Conferencias PLoP (Pattern Languages of Program Design)**
  - **Se comienzan a desarrollar en 1994 y tratan de refinar y extender las definiciones de patrones con la ayuda de expertos en el área**
- **1995 – “Design Patterns: Elements of Reusable Object-Oriented Software”**
  - **En congresos posteriores se dedicaron reuniones especiales para debatir el tema de patrones y el asunto fue madurando hasta la publicación 1995 del libro de Erich Gamma, Richard Helm, Ralph Johnson y John Vlissides, conocidos habitualmente como la banda de los cuatro (Gang of Four) o simplemente GoF.**
  - **El éxito de este libro fue el espaldarazo definitivo que aupó al tema de los patrones como uno de los más candentes dentro de los desarrollos orientados a objetos.**

Introducción
Tipos de Patrones

- **Según el nivel conceptual:**
  - **Patrones arquitectónicos:**
    - **Un patrón arquitectónico expresa una organización estructural fundamental para sistemas de software. El patrón provee de un conjunto de subsistemas predefinidos, especifica sus responsabilidades e incluye reglas y guías para organizar las relaciones entre ellos.**
  - **Patrones de diseño:**
    - **Un patrón de diseño provee un esquema para refinar los subsistemas o componentes de un sistema software, o las relaciones existentes entre ellos. Describe una estructura común y recurrente de comunicación entre componentes que resuelve un problema general de diseño dentro de un contexto particular.**
  - **Idiomas:**
    - **Un idioma es un patrón de bajo nivel específico para un lenguaje de programación. Un idioma describe como implementar aspectos particulares de componentes o de relaciones entre ellos utilizando las características de un determinado lenguaje.**

Introducción
Tipos de Patrones

- **Antipatrones**
  - **Si los patrones representan prácticas correctas existen también los antipatrones, que representan lecciones aprendidas en el desarrollo del software.**
  - **Tipos: “malas prácticas” y “refactoring”**
  - **Antipatrón “malas prácticas”**
    - **Describen una mala solución a un determinado problema y que nos lleva a una mala situación.**
    - **Es útil ya que nos permite identificar las malas soluciones a tiempo y nos permite apartarnos de ese camino y seguir el camino marcado por los patrones apropiados para resolver el problema.**
  - **Antipatrón “refactoring”**
    - **Describen cómo salir de una mala situación y transitar hacia una buena solución.**
    - **Intentan ir un paso más allá recomendando un curso de acción para la corrección y recuperación de situaciones indeseables. Está técnica se conoce habitualmente como “refactoring” .**

Introducción
Tipos de Patrones

- **Tipos de patrones de diseño:**
  - **Patrones creacionales:**
    - **Tratan sobre cómo crear instancias de objetos y sobre cómo hacer los programas más flexibles y generales abstrayendo el proceso de creación de instancias.**
  - **Patrones estructurales:**
    - **Describen cómo las clases y los objetos pueden ser combinados para formar estructuras mayores.**
  - **Patrones de comportamiento:**
    - **Son patrones que tratan de forma más específica con aspectos relacionados con la comunicación entre objetos.**

Índice

2. Patrones creacionales

- Inmutable
- Instancia Única (Singleton)
- Método Factoría (Factory Method)

Patrones Creacionales

- **Aspectos que analizaremos de cada patrón:**
  - **Descripción:**
    - **En la que se detallará el nombre del patrón, el tipo de problema intenta resolver y a que grupo de patrones pertenece**
  - **Elementos que lo componen:**
    - **Qué clases componen el patrón, cuál es la relación entre ellas y cómo es su funcionamiento (se muestra el gráfico UML)**
  - **Ejemplo:**
    - **Código de ejemplo en el que se aplique el patrón de diseño**
  - **Ventajas e Inconvenientes:**
    - **Qué ventajas e inconvenientes surgen con la aplicación del patrón.**
  - **Ejemplo del API de Java:**
    - **Ejemplo del API de Java en el que se utiliza el patrón descrito.**

Patrones Creacionales
Inmutable

- **Descripción**
  - **El patrón singleton se ocupa de diseñar clases que sólo permitían la existencia de una instancia, el patron inmutable se encarga de diseñar clases que permiten crear cualquier número de instancias pero no permite la posterior modificación de estas instancias. De esta forma, se evita tener que sincronizar los objetos que comparten referencias al objeto inmutable.**
  - **Este patrón no aparece incluido en el libro GoF, aunque sí en otros libro de patrones como el de Grand**

- **Elementos que lo componen**
  - **El esquema de este patrón consiste en una clase que incorpora atributos privados, métodos de lectura y métodos que simulan alterar el estado del objeto pero que en realidad devuelven una nueva instancia de dicho objeto.**

Patrones Creacionales
Inmutable

- **Elementos que lo componen**

Inmutable
- atributos: tipo
+ getAtributos(): tipo + Actualiza (): Inmutable

- **Ejemplo**

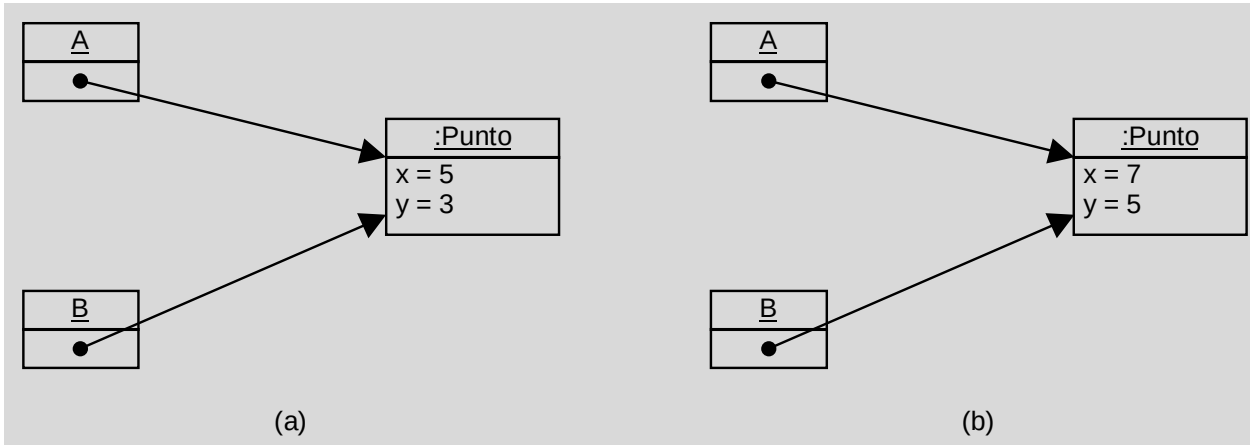
```
class Punto
{
  private int x;
  private int y;

  public Punto(int _x, int _y)
  {
    x=_x;
    y=_y;
  }

  public int getX()
  { return x; }

  public int getY()
  { return y; }

  public Punto offset (int xoff, int yoff)
  { return new Punto (x+xoff, y+yoff); }
}
```

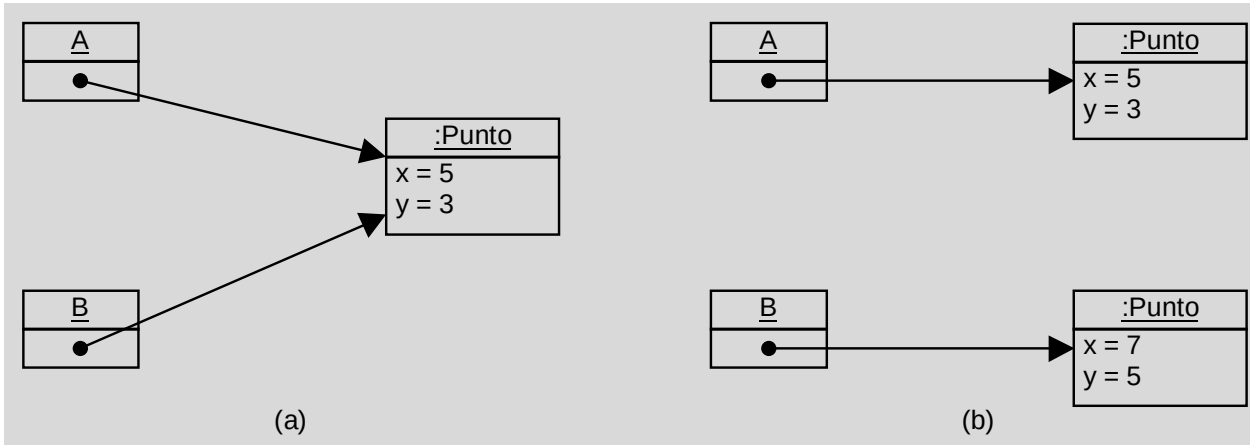
Patrones Creacionales
Inmutable

- **Ventajas**
  - **La principal ventaja del patrón inmutable es que nos asegura que, una vez que hemos obtenido una referencia a un objeto, éste no cambiará debido a efectos laterales de las modificaciones de otros elementos sobre dicho objeto.**
  - **Recordemos que en Java cuando una clase contiene a un objeto como parte de su estado, en realidad lo que contiene es una referencia a ese objeto. Un mismo objeto, por tanto, puede estar formando parte del estado de diversas clases por lo que un cambio en dicho objeto provocará el efecto lateral de cambios en el estado de todas las clases que lo referencian.**
  - **Sin embargo, si cada vez que se hace un cambio a un objeto éste en realidad realiza esos cambios sobre un clon, dejando al objeto original intacto, se habrán evitado los indeseados efectos laterales.**

Patrones Creacionales
Inmutable

- **Ventajas**





Patrones Creacionales
Inmutable

- **Inconvenientes**
  - **La modificación de su estado penaliza el rendimiento de la aplicación ya que en realidad implica la creación de un nuevo objeto**
  - **Esta penalización sólo será significativa cuando sean necesarias muchas modificaciones de los objetos inmutables o cuando la creación de un nuevo objeto inmutable sea un proceso costoso.**
  - **La solución a este problema consiste en la definición de clases mutables asociadas a clases inmutables y que puedan utilizarse para realizar las operaciones que resultan costosas en las clases inmutables.**
  - **La clase mutable es idéntica a la inmutable salvo en tres aspectos:**
    - **Las operaciones de modificación del estado no crean nuevas instancias sino que, efectivamente, varían el estado de esa clase.**
    - **Se incluye un nuevo constructor que permita crear instancias de la clase mutable a partir de instancias de la clase inmutable.**
    - **Se incluye un método que permite crear una instancia de la clase inmutable a partir de una instancia de la clase mutable.**
  - **Ahora la modificación sucesiva del estado de la clase inmutable puede hacerse utilizando sólo la clase inmutable o utilizando la clase mutable.**

Patrones Creacionales
Inmutable

- **Clase mutable asociada a una clase inmutable**

```
class PuntoMutable
{
  private int x;
  private int y;

  public PuntoMutable(int _x, int _y)
  { x=_x;
    y=_y;
  }

  public PuntoMutable(Punto p)
  { x=p.getX();
    y=p.getY();
  }

  public int getX()
  { return x; }

  public int getY()
  { return y; }

  public PuntoMutable offset (int xoff, int yoff)
  { x+=xoff;
    y+=yoff;
    return this;
  }

  public Punto PuntoInmutable()
  { return new Punto(x,y); }
}
```

Patrones Creacionales
Inmutable

- **Modificaciones en una clase inmutable**

```
public class PatronInmutable
{
  public static void main (String [] args)
  {
    int i;

    // Modificación realizada directamente sobre el objeto inmutable
    Punto P1 = new Punto(5,7);
    for (i=1; i<1000; i++)
      P1=P1.offset(1, 2);
    System.out.println(P1.getX() + " " + P1.getY());

    // Modificación realizada utilizando la clase mutable asociada
    Punto P2 = new Punto(5,7);
    PuntoMutable Aux = new PuntoMutable(P2);
    for (i=1; i<1000; i++)
      Aux=Aux.offset(1, 2);
    P2=Aux.mutableAInmutable();
    System.out.println(P2.getX() + " "  + P2.getY());
  }
}
```

Patrones Creacionales
Inmutable

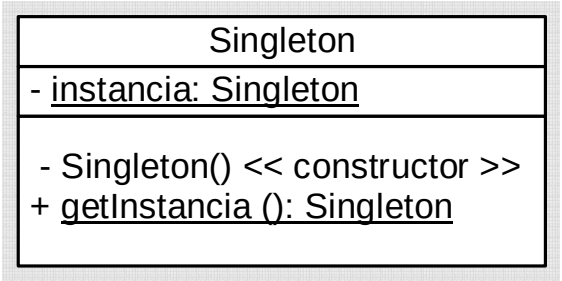
- **Ejemplo en el API de Java**
  - **En las clases `String` y `StringBuffer`**
    - **Un `String` en Java es inmutable ya que no puede ser modificado.**
    - **Existen una serie de métodos que aparentan modificar el `String` (como `toLowerCase` o `substring`) pero en realidad lo que hacen es devolver una nueva instancia de un objeto de tipo `String`.**
    - **La clase mutable asociada a `String` es `StringBuffer`. Esta clase incluye un constructor que permite crear un `StringBuffer` a partir de un `String`, incluye los mismos métodos que `String` pero sin implicar la creación de nuevos objetos e incluye un método `toString` que permite crear un `String` a partir de un `StringBuffer`.**
  - **En Java 5.0**
    - **Se incluye una nueva clase `StringBuilder` con un API compatible con `StringBuffer`, pero sin garantía de sincronización**
    - **Se recomienda utilizar `StringBuilder` en aplicaciones que utilizan un solo hilo de ejecución (lo más común) porque es más eficiente que `StringBuffer`**

Patrones Creacionales
Instancia Única (Singleton)

- **Descripción**
  - **Se utiliza para asegurarnos que una clase tiene sólo una instancia y proporcionar un punto global de acceso a ella.**
  - **Ejemplo: en un sistema pueden existir muchas impresoras pero sólo puede existir una instancia del objeto que representa al gestor de impresión.**
- **Elementos que lo componen**
  - **Una clase que define como privado su constructor (para evitar la creación de nuevos objetos), incluye un atributo de clase privado (que se utiliza para alojar la instancia única), y un método de clase que permite el acceso a dicha instancia.**

Patrones Creacionales
Instancia Única (Singleton)

- **Elementos que lo componen**



- **Ejemplo**

```
class Singleton // Inicialización temprana
{
  private static Singleton
    instancia=new Singleton();

  private Singleton ()  {}

  public static Singleton getInstancia()
  { return instancia; }
}
```

(a) Inicialización temprana o early

```
class Singleton  // Inicialización tardía
{
  private static Singleton instancia=null;

  private Singleton ()  {}

  public static Singleton getInstancia()
  {
    if (instancia == null )
      instancia = new Singleton();
    return instancia;
  }
}
```

(b) Inicialización tardía o lazy

Patrones Creacionales

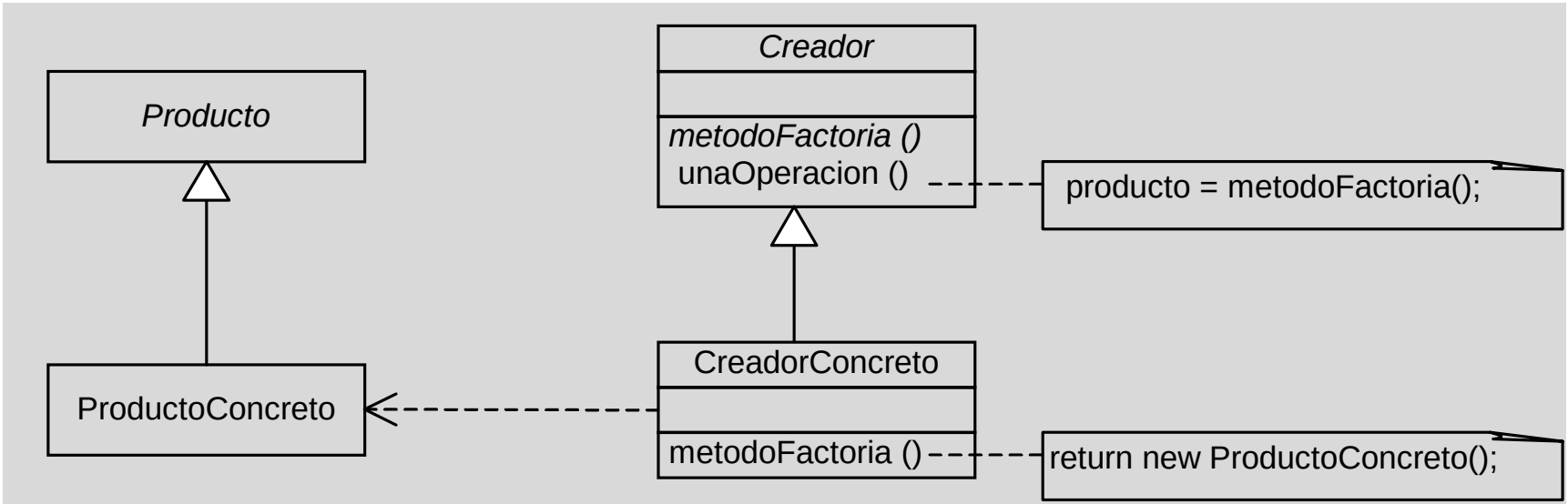
Método Factoría (Factory Method)

- **Descripción**
  - **Permite definir un interfaz para crear un objeto pero permitiendo que las subclases sean las que decidan qué objeto hay que crear.**
- **Elementos que lo componen**
  - ***Producto (Product)*:**
    - **Define el interfaz de los objetos creados por el método factoría.**
  - ***ProductoConcreto (ConcreteProduct)*:**
    - **Implementa el interfaz de los productos.**
  - ***Creador (Creator)*:**
    - **Declara el método factoría, que devuelve un objeto de tipo Producto. El método factoría puede ser abstracto o puede dar una implementación por defecto que devuelva un ProductoConcreto determinado. También puede definir otras operaciones que utilicen el método factoría.**
  - ***CreadorConcreto*:**
    - **Sobreescribe el método factoría para devolver una instancia de un ProductoConcreto.**

Patrones Creacionales

Método Factoría (Factory Method)

- **Elementos que lo componen**



Patrones Creacionales

Método Factoría (Factory Method)

- **Ejemplo 1 (factoría en clase abstracta):**

```
abstract class FondoInversion
{
  public int saldo;
  protected String nombre;

  public String toString()
  { return nombre; }
}

class FIAMM extends FondoInversion
{
  public FIAMM(int s)
  {
    saldo = s;
    nombre="FIAMM";
  }
}

class FIM extends FondoInversion
{
  public FIM(int s)
  {
    saldo = s;
    nombre="FIM";
  }
}
```

```
abstract class FactoriaFondo
{
  public abstract FondoInversion creaFondoInversion(int saldo);
}

class FactoriaFIAMM extends FactoriaFondo
{
  public FondoInversion creaFondoInversion(int saldo)
  { return new FIAMM(saldo); }
}

class FactoriaFIM extends FactoriaFondo
{
  public FondoInversion creaFondoInversion(int saldo)
  { return new FIM(saldo); }
}

class PatronMetodoFactoria
{
  public static void main (String [] args)
  {
    FactoriaFondo factoria = new FactoriaFIM();
    FondoInversion fondo = factoria.creaFondoInversion(1000);
    System.out.println("Mi fondo es = " + fondo);
  }
}
```

El método factoría se declara abstracto en la superclase y DEBE ser redefinido por las subclases (no hay implementación por defecto)

Patrones Creacionales

Método Factoría (Factory Method)

- **Ejemplo 2 (factoría en clase concreta):**

```
abstract class FondoInversion
{
  public int saldo;
  protected String nombre;

  public String toString()
  { return nombre; }
}

class FIAMM extends FondoInversion
{
  public FIAMM(int s)
  {
    saldo = s;
    nombre="FIAMM";
  }
}

class FIM extends FondoInversion
{
  public FIM(int s)
  {
    saldo = s;
    nombre="FIM";
  }
}
```

```
class FactoriaFondo
{
  public FondoInversion creaFondoInversion(int saldo)
  { return new FIAMM(saldo); }
}

class FactoriaFIAMM extends FactoriaFondo
{
  public FondoInversion creaFondoInversion(int saldo)
  { return new FIAMM(saldo); }
}

class FactoriaFIM extends FactoriaFondo
{
  public FondoInversion creaFondoInversion(int saldo)
  { return new FIM(saldo); }
}

class PatronMetodoFactoria2
{
  public static void main (String [] args)
  {
    FactoriaFondo factoria = new FactoriaFondo();
    FondoInversion fondo = factoria.creaFondoInversion(1000);
    System.out.println("Mi fondo es = " + fondo);
  }
}
```

El método factoría no es abstracto (incluye una implementación por defecto) y habitualmente la superclase tampoco es abstracta

No es necesario utilizar las subclases si no se quiere. Pero se abre la posibilidad a definir una subclase especializada

Patrones Creacionales

Método Factoría (Factory Method)

- **Ejemplo 3 (factoría en clase concreta parametrizada):**

```
abstract class FondoInversion
{
  public int saldo;
  protected String nombre;

  public String toString()
  { return nombre; }
}

class FIAMM extends FondoInversion
{
  public FIAMM(int s)
  {
    saldo = s;
    nombre="FIAMM";
  }
}

class FIM extends FondoInversion
{
  public FIM(int s)
  {
    saldo = s;
    nombre="FIM";
  }
}
```

```
class FactoriaFondo
{ public FondoInversion creaFondoInversion(int tipo, int saldo)
  { switch (tipo)
    {
      case 1 : return new FIAMM(saldo);
      case 2 : return new FIM(saldo);
      default: return new FIAMM(saldo);
    }
  }
}

class FactoriaExtendida extends FactoriaFondo
{ public FondoInversion creaFondoInversion(int tipo, int saldo)
  { switch (tipo)
    {
      case 1 : return new FIAMM(saldo);
      case 2 : return new FIM(saldo);
      default: return new FIM(saldo);
    }
  }
}

class PatronMetodoFactoria3
{
  public static void main (String [] args)
  {
    FactoriaFondo factoria = new FactoriaFondo();
    FondoInversion fondo = factoria.creaFondoInversion(1, 1000);
    System.out.println("Mi fondo es = " + fondo);
  }
}
```

En principio no hay subclases para cada producto sino una construcción switch en un método plantilla parametrizado

Se pueden definir subclases que cambien el funcionamiento del método plantilla parametrizado

Patrones Creacionales

Método Factoría (Factory Method)

- **Ejemplo 4 (factoría en clase concreta parametrizada usando reflexión):**

```
class FactoriaFondo
{
  public FondoInversion creaFondoInversion(String tipo)
  {
    FondoInversion f = null;
    try
    {
      f = (FondoInversion)Class.forName("patrones.metodofactoria."+tipo).newInstance();
    }
    catch (Exception e)
    {
      throw new IllegalArgumentException();
    }
    return f;
  }
}

class PatronMetodoFactoria4
{
  public static void main (String [] args)
  {
    FactoriaFondo factoria = new FactoriaFondo();
    FondoInversion fondo = factoria.creaFondoInversion("FIAMM");
    System.out.println("Mi fondo es = " + fondo);
  }
}
```

Usando reflexión podemos evitar el uso del switch. Las clases se construyen a partir de su nombre con el método Class.forName

Una vez obtenido el objeto Class que representa a la clase podemos crear una instancia usando el método newInstance

Por simplicidad se recogen todas las posibles excepciones y se manda una única IllegalArgumentException

NOTA: newInstance llama al constructor sin parámetros, por eso se ha omitido el saldo a la hora de construir el fondo. Para llamar al constructor con parámetros podría usarse Constructor.newInstance

Patrones Creacionales

Método Factoría (Factory Method)

- **Ventajas**
  - **Elimina la necesidad de incluir referencias a clases concretas en un código de carácter más genérico.**
  - **Lo que se hace es diferir a las subclases la decisión de qué objeto concreto crear.**
  - **Además este esquema es muy flexible ya que permite que las subclases creen versiones extendidas de los objetos para tratar problemas particulares.**
- **Inconvenientes**
  - **Puede obligar a extender la clase creadora sólo para crear un producto concreto.**
  - **Si no se quiere que esto ocurra es necesario buscar otras soluciones como el patrón prototipo.**
- **Ejemplo en el API de Java**
  - **La clase URLConnection tiene un método denominado getContent que devuelve el contenido de una URL empaquetado en el tipo de objeto apropiado.**
  - **El método getContent delega su funcionamiento a un objeto del tipo ContentHandler, que es una clase abstracta que juega el rol de producto en el patrón.**
  - **Existe una clase ContentHandlerFactory que juega el rol de creador en el patrón. La clase URLConnection tiene un método denominado setContentHandlerFactory que se utiliza para proveer de una factoría que pueda ser usada por los objetos URLConnection.**

Índice

3. Patrones estructurales

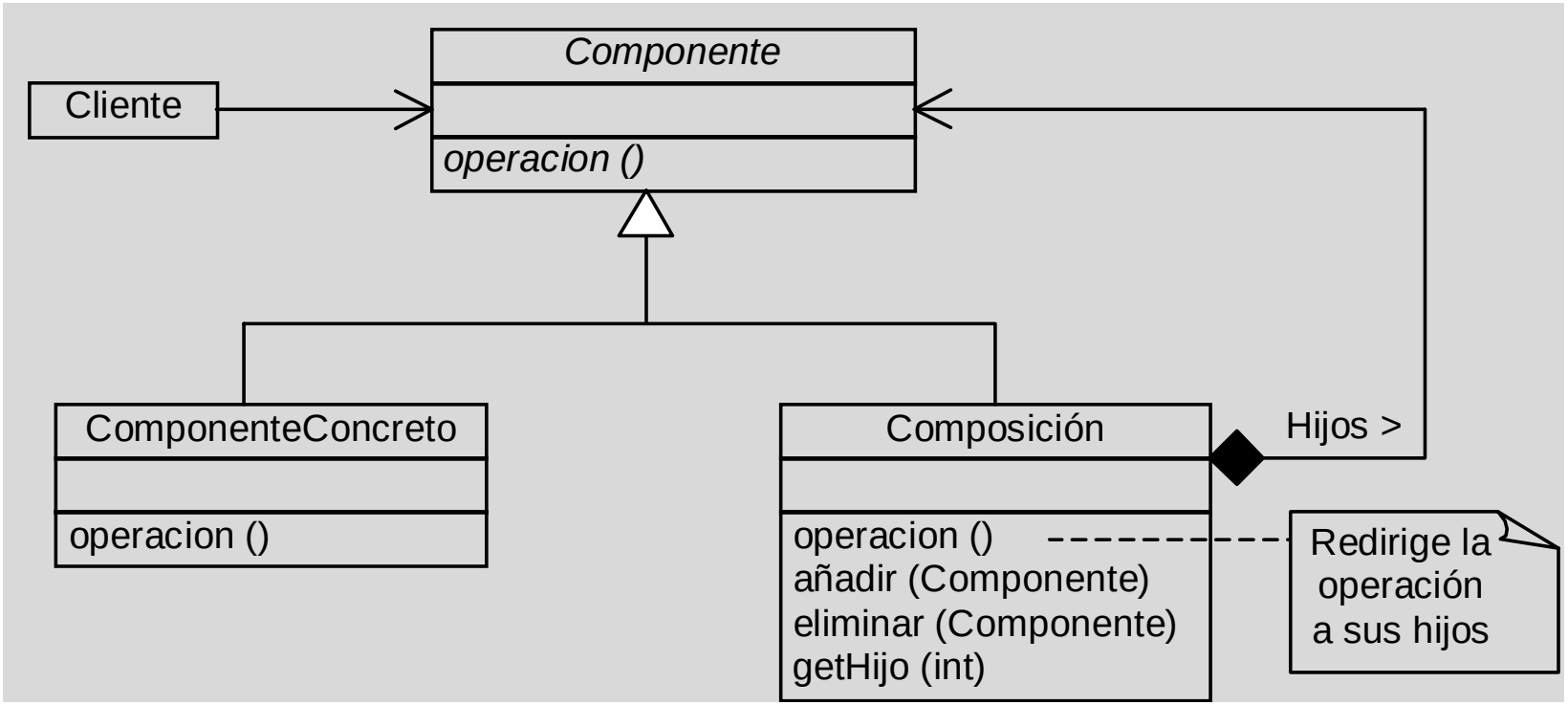
- Composición (Composite)
- Adaptador (Adapter)

Patrones Estructurales
Composición (Composite)

- **Descripción**
  - **Es un patrón estructural que se utiliza para componer objetos en estructuras de árbol que representan jerarquías todo-parte**
  - **Este patrón permite tratar uniformemente a los objetos y a las composiciones y es un buen ejemplo de utilización conjunta de herencia y composición.**
- **Elementos que lo componen**
  - ***Componente abstracto*: Declara la interfaz para los objetos en la composición implementando su comportamiento por defecto. Además declara la interfaz para acceder y manipular los componentes hijos.**
  - ***Componente concreto*: Define el comportamiento de los objetos elementales de la composición (aquellos que no pueden tener hijos).**
  - ***Composición (Composite)*: Define el comportamiento de los objetos compuestos de la composición.**
  - ***Cliente*: Maneja a los objetos a través del interfaz Componente.**

Patrones Estructurales
Composición (Composite)

- **Elementos que lo componen**



Patrones Estructurales
Composición (Composite)

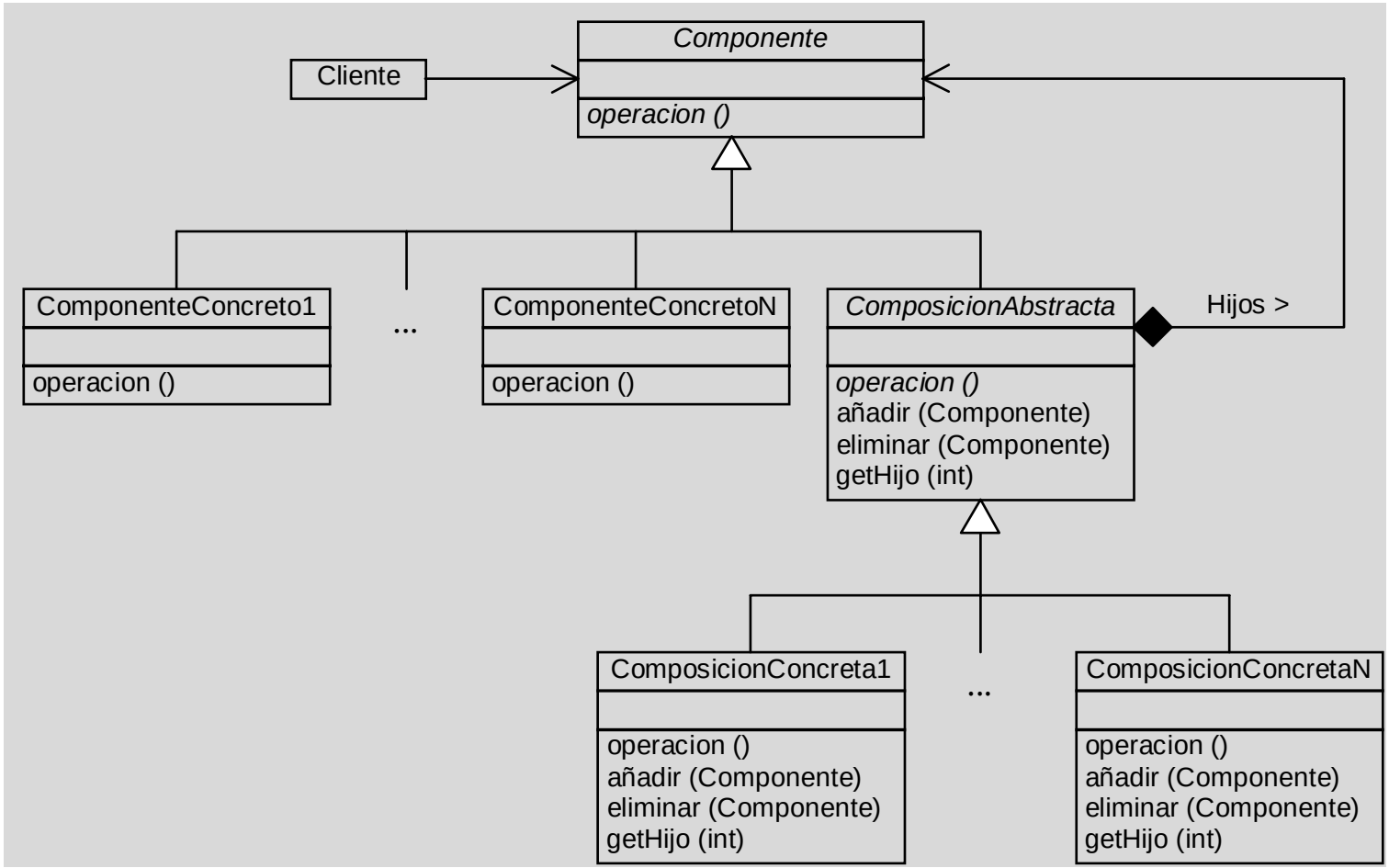
- **Ejemplo**

```
class ClienteArbol
{
  static public void main (String[] arg)
  {
    Nodo raiz = new Nodo();

    Hoja h1  = new Hoja(5);
    Nodo a1  = new Nodo();
    Nodo a2  = new Nodo();
    Hoja h21 = new Hoja(5);
    Hoja h22 = new Hoja(3);
    Hoja h31 = new Hoja(6);
    Hoja h32 = new Hoja(4);
    a2.añadir(h31);
    a2.añadir(h32);
    a1.añadir(a2);
    a1.añadir(h21);
    a1.añadir(h22);
    raiz.añadir(h1);
    raiz.añadir(a1);

   System.out.println(raiz.operacion());
  }
}
```

```
abstract class Arbol
{ public abstract int operacion(); }

class Hoja extends Arbol
{
  private int valor;

  Hoja (int v)
  { valor = v; }

  public int operacion()
  { return valor; }
}

class Nodo extends Arbol
{
  private java.util.Vector hijos;

  Nodo ()
  { hijos = new java.util.Vector();  }

  public int operacion()
  {
    int suma=0;
    for (int i=0; i < hijos.size(); i++)
      suma += getHijo(i).operacion();
    return suma;
  }

  public void añadir (Arbol a)
  { hijos.addElement(a); }

  public void eliminar (Arbol a)
  { hijos.removeElement(a); }

  public Arbol getHijo (int i)
  { return (Arbol) hijos.elementAt(i); }
}
```

Patrones Estructurales
Composición (Composite)

- **Elementos que lo componen (versión compleja)**

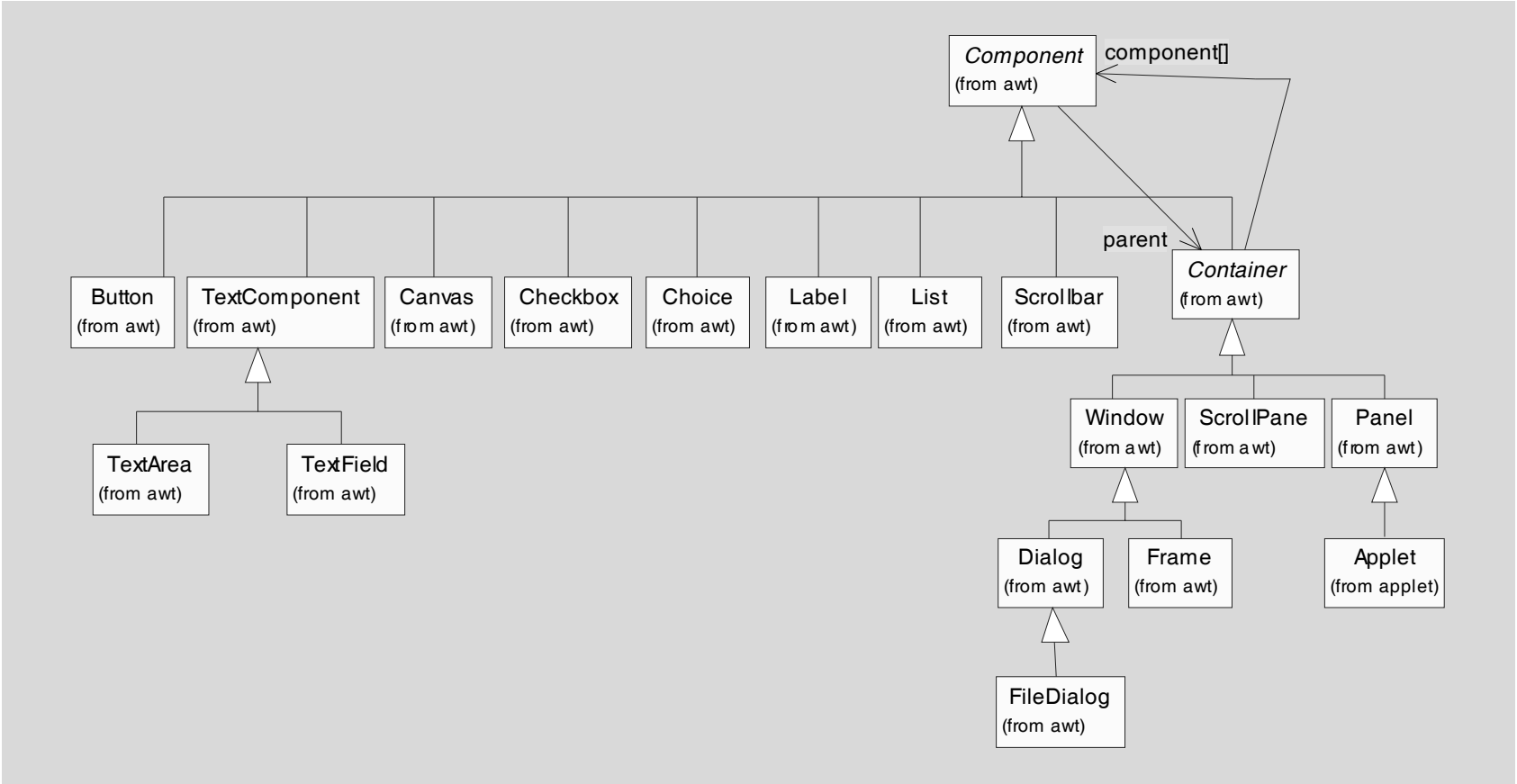


Patrones Estructurales
Composición (Composite)

- **Ventajas**
  - **El patrón composición permite tratar a los objetos básicos y a los objetos compuestos de manera uniforme, facilitando la introducción de nuevos componentes sin afectar al cliente**
- **Inconvenientes**
  - **Para restringir qué componentes concretos pueden ir en una determinada composición es necesario añadir comprobaciones en tiempo de ejecución**
- **Seguridad y transparencia**
  - **La decisión de dónde situar las operaciones propias de la Composición (añadir, eliminar, getHijo) es una decisión entre transparencia y seguridad**
  - **Transparencia: Si se definen en Componente se maximiza su interfaz haciendo que las hojas y las composiciones puedan ser tratadas de forma genérica (aunque puede dar lugar a comportamientos no deseados cuando se intenta añadir o borrar hijos de una hoja)**
  - **Seguridad: Si se definen en Composición estamos seguros de que las operaciones se llaman sobre la clase correcta, pero será necesario trabajar con variables del tipo Composición para utilizar estos métodos**

Patrones Estructurales
Composición (Composite)

- **Ejemplo en el API de Java**



Patrones Estructurales
Adaptador (Adapter)

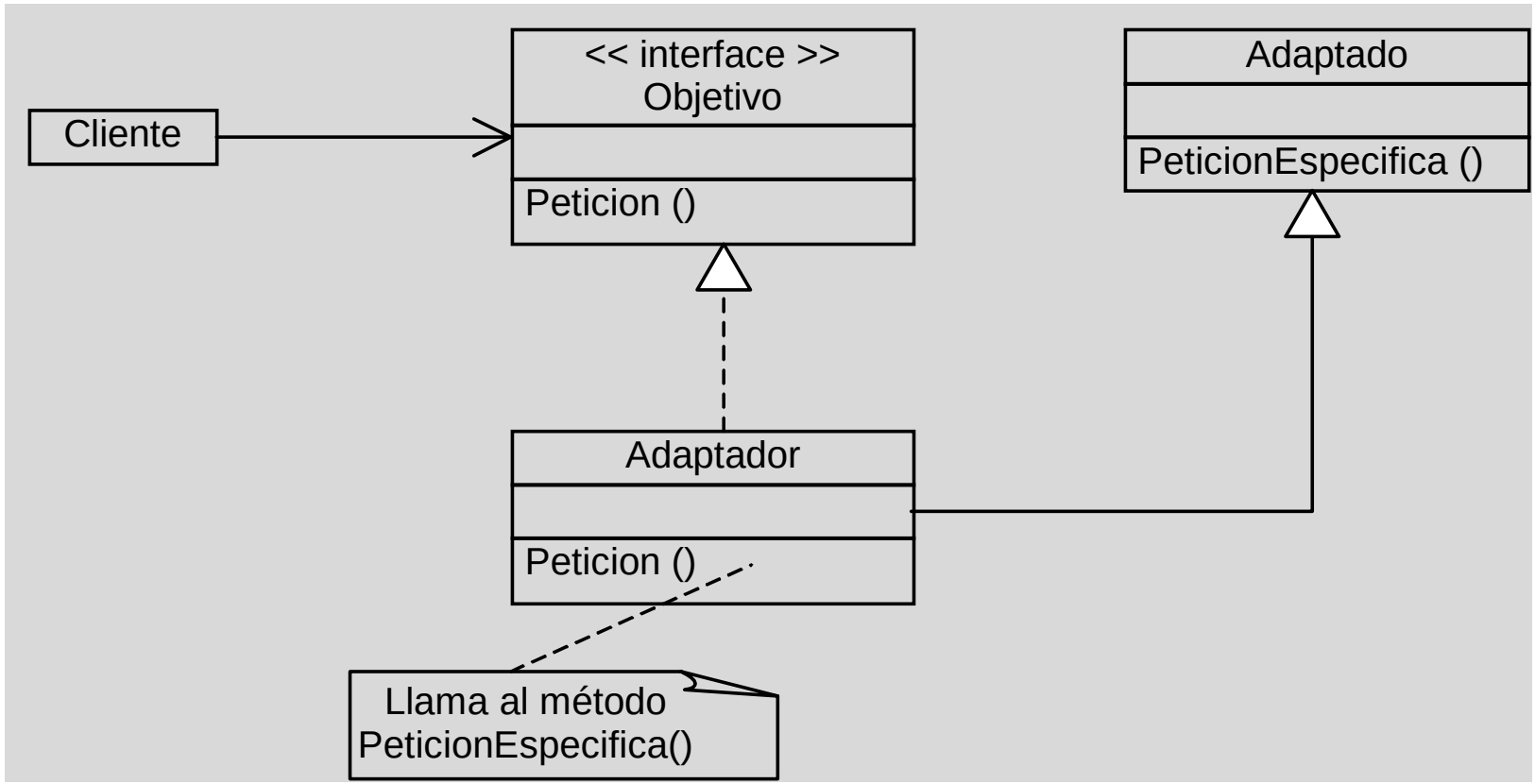
- **Descripción**
  - **Los interfaces se utilizan para desacoplar las clases clientes de las clases que ofrecían ciertos servicios.**
  - **De esta forma se hace independiente la clase cliente de la clase servidora, sin embargo se fuerza a que la clase servidora implementara un determinado interfaz.**
  - **En diversas ocasiones puede no interesar modificar la clase servidora, bien porque no se puede (porque no se dispone del código fuente), o porque no se quiere modificar la clase para un determinado uso concreto y especializado.**
  - **Para convertir el interfaz de una clase en otro interfaz que es el esperado por el cliente que usa dicha clase se utiliza el patrón adaptador**

Patrones Estructurales
Adaptador (Adapter)

- **Elementos que lo componen**
  - ***Objetivo (Target)*: Interfaz que utiliza el cliente.**
  - ***Cliente (Client)*: Colabora con los objetos usando la interfaz Objetivo**
  - ***Adaptado (Adaptee)*: Define una interfaz existente que necesita ser adaptada.**
  - ***Adaptador (Adapter)*: Adapta la interfaz del adaptado a la interfaz Objetivo.**
- **Esquema herencia múltiple**
  - **El objeto adaptador hereda el interfaz de Objetivo al mismo tiempo que hereda de la clase Adaptado el método PeticiónEspecífica**
  - **En este caso la clase Objetivo sólo puede desarrollarse mediante un interface ya que Java no admite herencia múltiple en clases convencionales.**
- **Esquema composición**
  - **El objeto adaptador hereda el interfaz de Objetivo y está compuesto de un objeto del tipo Adaptado a cuyo método PeticiónEspecífica delega las peticiones que le llegan del cliente**
  - **Objetivo puede ser un interface o una clase abstracta..**

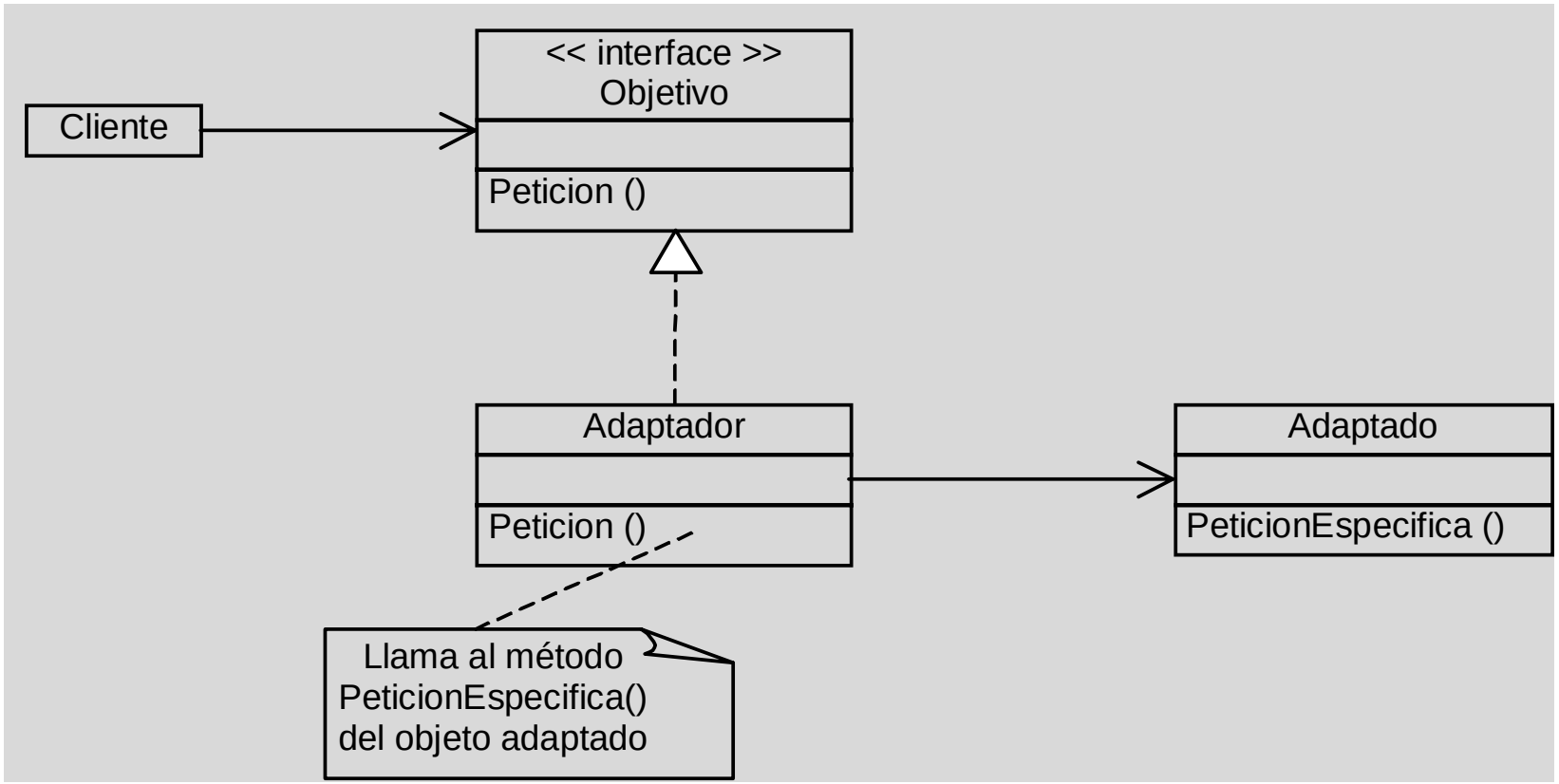
Patrones Estructurales
Adaptador (Adapter)

- **Elementos que lo componen (herencia múltiple)**



Patrones Estructurales
Adaptador (Adapter)

- **Elementos que lo componen (composición)**



Patrones Estructurales
Adaptador (Adapter)

- **Ejemplo (herencia múltiple)**

```
class PatronAdaptador
{
  public static void main (String[] arg)
  {
    Objetivo Adr = new Adaptador();
    Adr.Peticion();
  }
}

interface Objetivo
{
  public void Peticion();
}

class Adaptador extends Adaptado implements Objetivo
{
  public void Peticion()
  { PeticionEspecifica() ; }
}

class Adaptado
{
  public void PeticionEspecifica()
  { System.out.println ("Peticion especifica..."); }
}
```

Patrones Estructurales
Adaptador (Adapter)

- **Ejemplo (composición)**

```
class PatronAdaptador
{
  public static void main (String[] arg)
  {
    Adaptado  Ado = new Adaptado();
    Objetivo Adr = new Adaptador(Ado);
    Adr.Peticion();
  }
}

interface Objetivo
{
  public void Peticion();
}

class Adaptador implements Objetivo
{
  private Adaptado adaptado;

  public Adaptador (Adaptado a)
  {  adaptado = a; }

  public void Peticion()
  { adaptado.PeticionEspecifica() ; }
}

class Adaptado
{
  public void PeticionEspecifica()
  { System.out.println ("Peticion especifica..."); }
}
```

Patrones Estructurales
Adaptador (Adapter)

- **Ventajas e inconvenientes**
  - **Herencia múltiple**
    - **Sólo permite adaptar una clase, no sus subclases**
    - **Permite que el adaptador sobreescriba parte de la conducta del adaptado (ya que es una subclase)**
    - **Además con la herencia nos evitamos la indirección que se produce en la composición.**
  - **Composición**
    - **Permite que un único adaptador trabaje sobre muchos adaptados (una clase y sus subclases).**
    - **Introduce un nivel más de indirección, lo cual implica una penalización en rendimiento**
    - **Complica la sobreescritura de la conducta del adaptado (se necesita crear una subclase del adaptado y que el adaptador se refiera a esta subclase).**

Patrones Estructurales
Adaptador (Adapter)

- **Ejemplo en el API de Java (Eventos)**

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;

public class PatronAdaptadorSwing
{ public static void main (String [] args)
  { Ventana f = new Ventana();
    f.show();
  }
}

class Ventana extends JFrame
{ private JButton b1 = new JButton ("Boton 1"),
                  b2 = new JButton ("Boton 2");
  private JTextField t = new JTextField(20);
  private AdaptadoSwing Ado = new AdaptadoSwing();
  Container contentPane = getContentPane();

  public Ventana()
  { init();
    pack();
  }

  ...
```

Método Peticion que llama a PeticiónEspecifica

```
  public void init ()
  { setTitle ("Patrón adaptador");
    contentPane.add(b1, BorderLayout.WEST);
    contentPane.add(b2, BorderLayout.EAST);
    contentPane.add(t,  BorderLayout.NORTH);
    b1.addActionListener(new B1());
    b2.addActionListener(new B2());
    addWindowListener (new WindowAdapter()
    {
      public void windowClosing (WindowEvent e)
      { System.exit(0); }
    });
  }

  class B1 implements ActionListener
  { public void actionPerformed (ActionEvent e)
    { t.setText(Ado.PeticionEspecifica() + "Composicion"); }
  }

  class B2 extends AdaptadoSwing implements ActionListener
  { public void actionPerformed (ActionEvent e)
    { t.setText(PeticionEspecifica() + "Herencia múltiple"); }
  }
}

class AdaptadoSwing
{
  public String PeticionEspecifica()
  { return "Peticion especifica..."; }
}
```

AddActionListener requiere un objeto que implemente el interface Objetivo (ActionListener)

Clase Adaptador

Interface Objetivo

Clase Adaptado

Patrones Estructurales
Adaptador (Adapter)

- **Ejemplo en el API de Java (Wrapper classes)**

```
public final class Integer extends Number
{
  public static final int   MIN_VALUE = 0x80000000;
  public static final int   MAX_VALUE = 0x7fffffff;
  private int value;

  public Integer(int value)
  { this.value = value; }

  public int intValue()
  { return value; }

  public boolean equals(Object obj)
  {
    if ((obj != null) && (obj instanceof Integer))
    { return value == ((Integer)obj).intValue(); }
    return false;
  }

  public String toString()
  { return String.valueOf(value); }
}
```

Clase Adaptador

Método Peticion que llama a PeticiónEspecifica (se llama a intValue para obtener el valor de un int)

```
class VectorInteger
{ public static void main (String [] args)
  { Integer suma;
    int i1, i2;

    java.util.Vector VectorEnteros = new java.util.Vector ();
    VectorEnteros.add(new Integer(1));
    VectorEnteros.add(new Integer(2));
    i1=((Integer)VectorEnteros.elementAt(0)).intValue();
    i2=((Integer)VectorEnteros.elementAt(1)).intValue();
    suma=new Integer(i1+i2);
    VectorEnteros.add(suma);
    for (int i=0; i<3; i++)
      System.out.println(((Integer)VectorEnteros.elementAt(i)).intValue());
  }
}
```

Adaptado

Un vector almacena objetos que cumplan el interfaz de Object (nuestro interfaz Objetivo)

Índice

4. Patrones estructurales

- Estado (State)
- Observador (Observer)
- Estrategia (Strategy)
- Metodo plantilla (Template Method)
- Comando (Command)

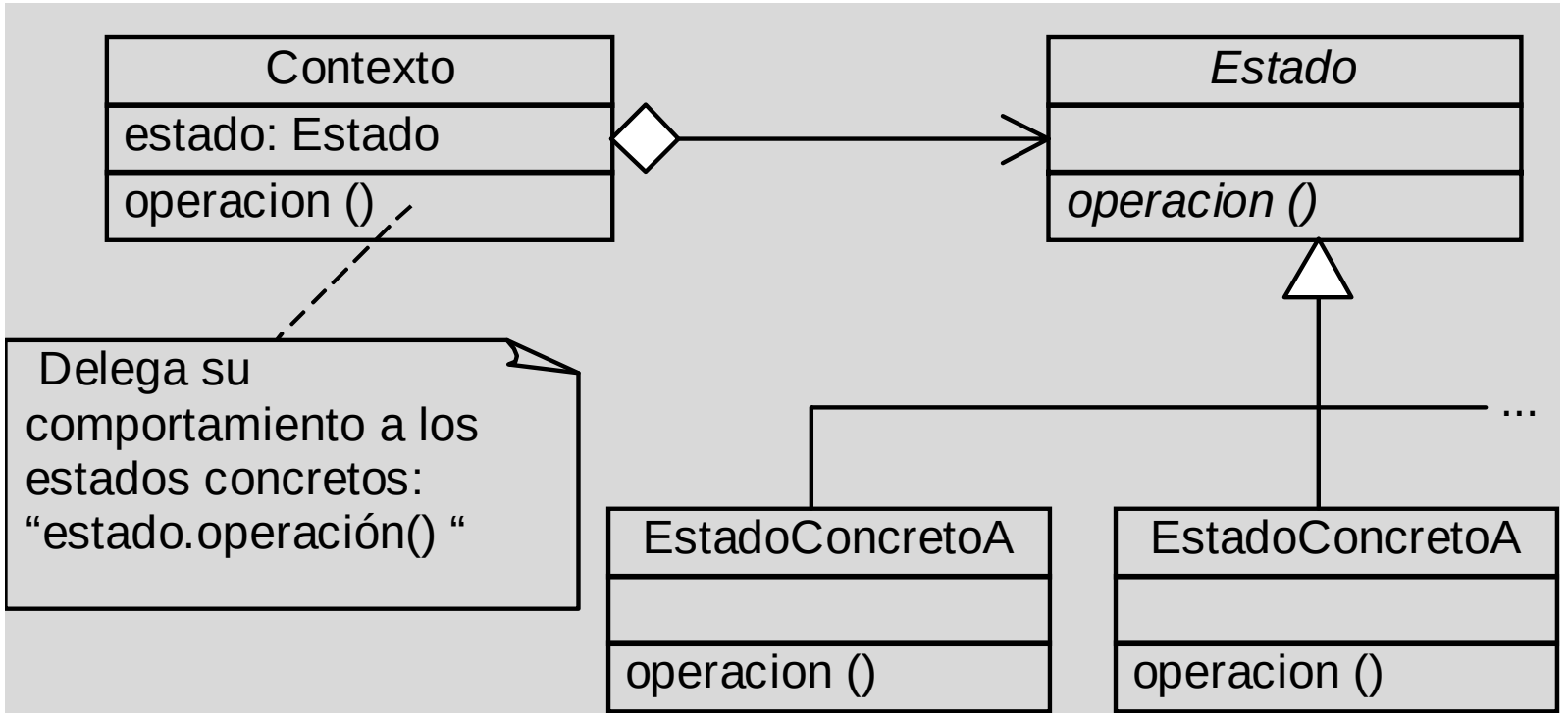
Patrones Comportamiento

Estado (State)

- **Descripción**
  - **El patrón estado es un patrón de comportamiento que permite a un objeto modificar su conducta al cambiar su estado interno.**
- **Elementos que lo componen**
  - ***Contexto*:**
    - **Mantiene una instancia de un estado concreto y puede incluir los criterios para la transición entre estados, aunque generalmente es más flexible que sean los propios estados los que definan sus transiciones.**
  - ***Estado*:**
    - **Clase abstracta que encapsula el comportamiento de los estados del contexto.**
  - ***Estados concretos*:**
    - **Cada subclase implementa el comportamiento asociado con un estado del contexto**

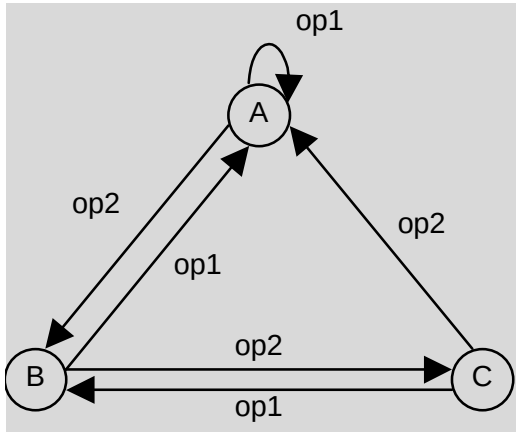
Patrones Comportamiento
Estado (State)

- **Elementos que lo componen**



Patrones Comportamiento
Estado (State)

- **Ejemplo**




```
class Contexto
{
  private Estado estado;
  public Contexto ()
  { estado = EstadoA.getInstancia(); }

  public void op1()
  { estado.op1(this); }

  public void op2()
  { estado.op2(this); }

  void cambiaEstado (Estado e)
  { estado = e; }
}
```

Patrones Comportamiento

Estado (State)

- **Ejemplo**

```
abstract class Estado
{
  abstract void op1(Contexto c);
  abstract void op2(Contexto c);
}

class EstadoA extends Estado
{
  private static EstadoA instancia;
  private EstadoA() {}
  public static EstadoA getInstancia()
  { if (instancia == null)
      instancia = new EstadoA();
    return instancia;
  }
  void op1(Contexto c)
  { ... }
  void op2(Contexto c)
  { c.cambiaEstado(EstadoB.getInstancia()); }
}
```

```
class EstadoB extends Estado
{
  // Código del Singleton
  ...
  void op1(Contexto c)
  { c.cambiaEstado(EstadoA.getInstancia()); }
  void op2(Contexto c)
  { c.cambiaEstado(EstadoC.getInstancia()); }
}

class EstadoC extends Estado
{
  // Código del Singleton
  ...
  void op1(Contexto c)
  { c.cambiaEstado(EstadoB.getInstancia()); }
  void op2(Contexto c)
  { c.cambiaEstado(EstadoA.getInstancia()); }
}
```

Patrones Comportamiento

Estado (State)

- **Aspectos a tener en cuenta**
  - **Los propios estados son los que definen la transición entre ellos. Podría haberse utilizado el contexto si, por ejemplo, los estados son utilizados por varias máquinas con distintas transiciones.**
  - **El estado suele implementarse como una variable privada del contexto. Si queremos que los estados sean capaces de modificar esta variable privada debe ser accesible a través de un método de escritura. La visibilidad de este método puede limitarse a package para evitar que otras clases lo utilicen pero en este caso es necesario que tanto los estados como el contexto se sitúen en el mismo paquete.**
  - **Los estados son definidos a través del patrón de instancia única (singleton), también se podría haber optado por crearlos y destruirlos cada vez que se necesiten.**

Patrones Comportamiento

Estado (State)

- **Cuando el comportamiento de un objeto depende de su estado interno tenemos dos posibilidades:**
  - **Incluir en los métodos del objeto sentencias condicionales que permitan realizar unas operaciones u otras dependiendo del estado interno**
  - **Utilizar el patrón estado creando una clase abstracta que representa al estado genérico y que se extiende con clases que representan estados concretos**
- **Ventajas e inconvenientes**
  - **Primera solución**
    - **Es más sencilla de implementar y más compacta**
    - **Se recomienda para un numero de estado pequeño, estable (no van a aparecer nuevos estados en el futuro) y que afecta a pocos métodos del contexto**
  - **Segunda solución (patrón estado)**
    - **Es más complicado y menos compacto que la solución anterior**
    - **Localiza y separa el comportamiento específico de cada estado.**
    - **Permite añadir nuevos estados de forma sencilla, simplemente añadiendo nuevas clases.**
    - **Hace más explícitas las transiciones entre estados.**
    - **Los objetos que representan a los estados pueden compartirse, siempre y cuando no incluyan información en variables de instancia.**

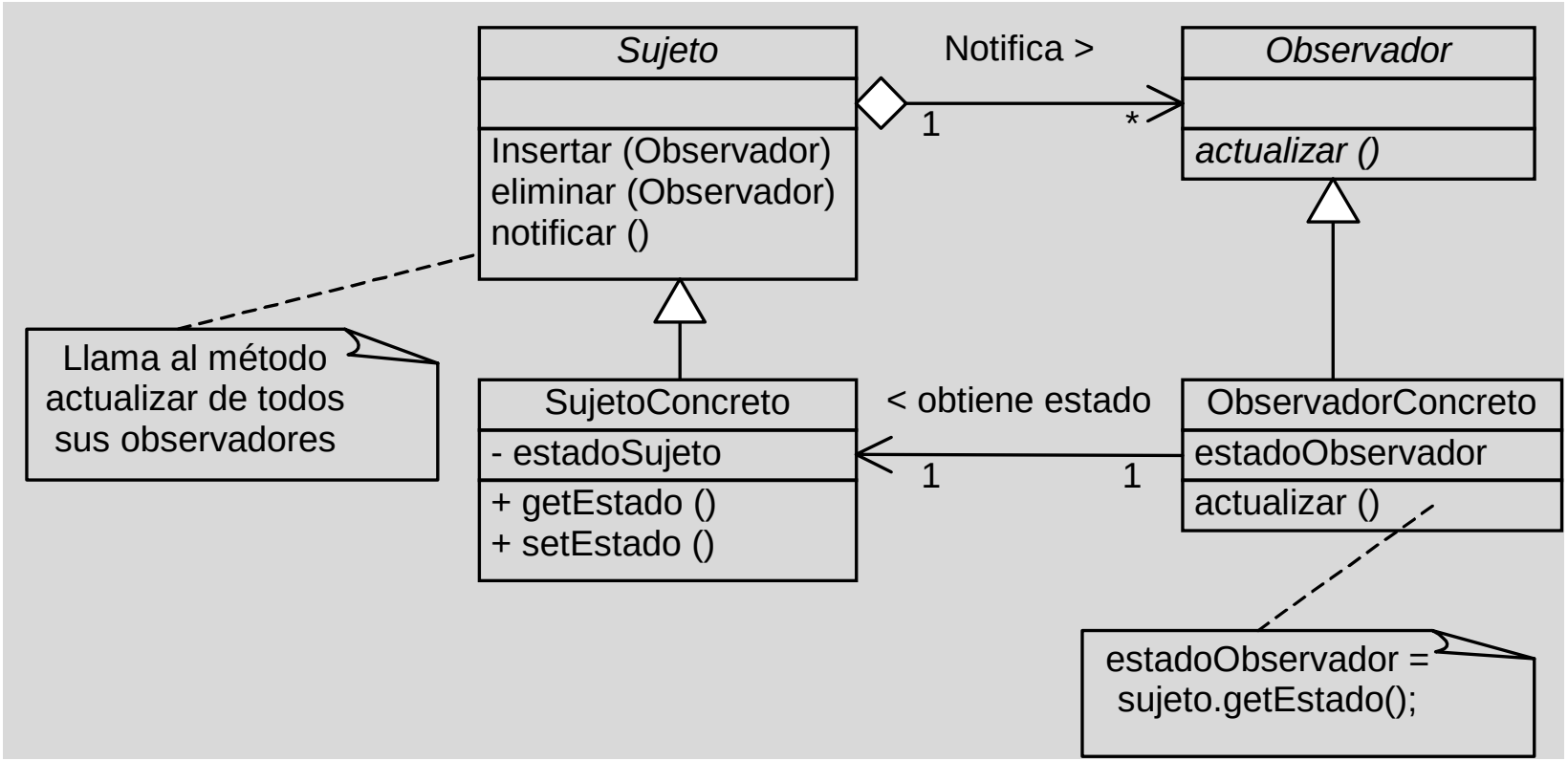
Patrones Comportamiento

Observador (Observer)

- **Descripción**
  - **Permite definir una dependencia “uno a muchos” entre objetos de tal forma que, cuando el objeto cambie de estado, todos sus objetos dependientes sean notificados y actualizados automáticamente.**
  - **En este patrón es importante la suposición de que el número de objetos observadores no esté limitado y que el objeto independiente no tenga que hacer ninguna asunción acerca de la clase a la que pertenecen estos objetos.**
- **Elementos que lo componen**
  - ***Sujeto (Subject)*: Clase abstracta que conoce a sus observadores y que proporciona la interfaz para agregar y eliminar observadores.**
  - ***Observador (Observer)*: Clase abstracta que define un método actualizar utilizado por el sujeto para notificar cambios en su estado.**
  - ***Sujeto concreto (ConcreteSubject)*: Subclase de sujeto que mantiene el estado del mismo y que se encarga de notificar a los observadores al cambiar su estado.**
  - ***Observador concreto (ConcreteObserver)*: Subclase de observador que mantiene una referencia al sujeto concreto e implementa la interfaz de actualización.**

Patrones Comportamiento
Observador (Observer)

- **Elementos que lo componen**



Patrones Comportamiento
Observador (Observer)

- **Elementos que lo componen (clases y métodos en Java)**

Método	**Significado**
void update (Observable o, Object arg)	Método llamado cuando cambia el estado del objeto observado

Método	**Significado**
void addObserver (Observer o)	Añade un observador al conjunto de observadores para este objeto (siempre y cuando el observado no haya sido añadido con anterioridad).
void deleteObserver (Observer o)	Elimina un observador del conjunto de obsevadores para este objeto.
void deleteObservers()	Elimina completamente la lista de observadores para este objeto.
int countObservers ()	Devuelve el número de observadores para este objeto.
void notifyObservers()	Si el objeto cambia, como es indicado por el método hasChanged, entonces notifica a todos sus observadores y llama al método clearChanged para indicar que el objeto ya ha notificado su cambio.
void notifyObservers(Object arg)	Igual que el anterior pero pasando el argumento arg.
void setChanged()	Marca al objeto observable como que ha cambiado, de forma que ahora el método hasChanged devolverá true.
void clearChanged ()	Indica que este objeto no ha cambiado últimamente o que ya ha notificado a todos sus observadores de un cambio reciente. De esta forma la llamada al método hasChange devolverá false.
boolean hasChanged ()	Comprueba si el estado del objeto observado ha cambiado.

Patrones Comportamiento
Observador (Observer)

- **Elementos que lo componen (código en Java)**

```
package java.util;

public class Observable
{
  private boolean changed = false;
  private Vector obs;

  public Observable()
  { obs = new Vector(); }

  public synchronized void addObserver(Observer o)
  {
    if (o == null)
      throw new NullPointerException();
    if (!obs.contains(o))
    { obs.addElement(o);  }
  }

  public synchronized void deleteObserver(Observer o)
  { obs.removeElement(o); }

  public synchronized void deleteObservers()
  { obs.removeAllElements(); }

  public synchronized int countObservers()
  { return obs.size(); }
```

```
  public void notifyObservers()
  { notifyObservers(null); }


  public void notifyObservers(Object arg)
  {
    Object[] arrLocal;

    if (!changed) return;

    arrLocal = obs.toArray();
    clearChanged();
    for (int i = arrLocal.length-1; i>=0; i--)
      ((Observer)arrLocal[i]).update(this, arg);
  }


  protected synchronized void setChanged()
  { changed = true;  }

  protected synchronized void clearChanged()
  { changed = false; }

  public synchronized boolean hasChanged()
  { return changed; }
}
```

```
package java.util;

public interface Observer
{ void update(Observable o, Object arg); }
```

Patrones Comportamiento
Observador (Observer)

- **Ejemplo**

```
import java.util.*;

class patronObservador
{
  static public void main (String[] args)
  {
    Piscina losCastros = new Piscina (75);
    AlarmaPiscina alarmaSocorrista = new AlarmaPiscina();
    AlarmaPiscina alarmaBomberos = new AlarmaPiscina();

    losCastros.addObserver (alarmaSocorrista);
    losCastros.addObserver (alarmaBomberos);

    System.out.println ("Nivel de la piscina: " + losCastros.getNivel());
    losCastros.abreGrifo(10);
    System.out.println ("Nivel de la piscina: " + losCastros.getNivel());
    losCastros.abreGrifo(30);
    System.out.println ("Nivel de la piscina: " + losCastros.getNivel());
    losCastros.quitaTapon(100);
  }
}
```

Patrones Comportamiento
Observador (Observer)

- **Ejemplo**

```
import java.util.*;

class Piscina extends Observable
{
  public static final int MAX=100;
  public static final int MIN=50;
  public static final String NIVEL_MAX="NIVEL_MAX";
  public static final String NIVEL_MIN="NIVEL_MIN";
  private int nivel=75;

  public Piscina (int valor)
  { nivel = valor; }

  public int getNivel()
  { return nivel; }

  public void abreGrifo (int incremento)
  { nivel += incremento;
    if (nivel > MAX)
    { setChanged();
      notifyObservers(NIVEL_MAX);
    }
  }

  public void quitaTapon (int decremento)
  { nivel -= decremento;
    if (nivel < MIN)
    { setChanged();
      notifyObservers(NIVEL_MIN);
    }
  }
}
```

Patrones Comportamiento
Observador (Observer)

- **Ejemplo**

```
class AlarmaPiscina implements Observer
{
  public void update (Observable o, Object arg)
  {
    System.out.println ("El nivel de la piscina es: "+((Piscina)o).getNivel());
    if (arg.equals (Piscina.NIVEL_MAX))
      System.out.println ("Hay que VACIAR urgentemente la piscina");
    else
      if (arg.equals (Piscina.NIVEL_MIN))
        System.out.println ("Hay que LLENAR urgentemente la piscina");
  }
}
```

Patrones Comportamiento

Observador (Observer)

- **Implementación del API de Java**
  - **Facilita la utilización del patrón pero obliga a que la clase observable herede de la clase Observable**
  - **El interfaz Observer trabaja con elementos Observable genéricos, es necesario hacer un typecast para acceder a las características propias del objeto observable**
- **Ventajas**
  - **Al desacoplar los observadores de la clase observable, ésta última no tiene por qué conocer la clase concreta a la que pertenecen sus observadores, simplemente se limita a notificar los cambios a través de un interfaz conocido.**
  - **La responsabilidad del objeto observado se acaba con la notificación, son los observadores los que tienen que decidir qué acción llevar a cabo después de la notificación.**
- **Inconvenientes**
  - **Los observadores no tienen contacto entre si, esto puede dar lugar a situaciones difíciles de controlar cuando varios observadores responden simultáneamente ante un cambio. En el peor de los casos puede dar lugar a una cascada de actualizaciones**
  - **La comunicación entre observado y observadores es simple, lo que fuerza a los observadores a deducir qué ha cambiado en el observado**
  - **Objetivo: buscar un equilibrio entre el *pull model* y el *push model***

Patrones Comportamiento
Observador (Observer)

- ***Pull model***
  - **Los observadores "tiran" del observado**
  - **El observado manda a los observadores una mínima notificación indicando que su estado ha cambiado**
  - **Es responsabilidad de los observadores descubrir qué ha cambiado y actuar en consecuencia**
  - **Este modelo enfatiza la ignorancia que tiene el objeto observado de sus observadores pero exige un mayor trabajo por parte de estos últimos (que puede dar lugar a problemas de eficiencia)**

- ***Push model***
  - **El observado "empuja" a los observadores**
  - **El observado manda a los observadores información detallada acerca de lo que ha cambiado**
  - **En este modelo el observado tiene un cierto conocimiento sobre las necesidades de los observadores. De esta forma no son tan independientes como en el modelo *pull***
  - **Sin embargo la comunicación puede ser más eficiente ya que no se fuerza a descubrir a los observadores qué es lo que ha cambiado**

Patrones Comportamiento
Observador (Observer)

- **Ejemplo en el API de Java**
  - **El patrón observador se utiliza en el modelo de eventos de Java (el utilizado a partir del JDK 1.1).**
  - **Por ejemplo el esquema para crear una clase que "observe" a un botón y que "reaccione" cuando este se pulsa sigue el siguiente proceso:**
    - **Creamos una clase que implemente el interfaz ActionListener, lo que nos obliga a dar implementación a un método actionPerformed que acepta como parámetro a un objeto de la clase ActionEvent.**
    - **Registramos un objeto de la clase que implementa ActionListener como escuchador del generador del evento (el botón). Este registro se realiza mediante el método addActionListener de la clase JButton. Este método acepta como parámetro una instancia de una clase que implementa el interfaz ActionListener.**
    - **Cuando el evento de pulsar el botón sucede el botón avisa a todos los escuchadores registrados de este hecho llamando a su método actionPerformed e incluyendo como parámetro un instancia de ActionEvent.**

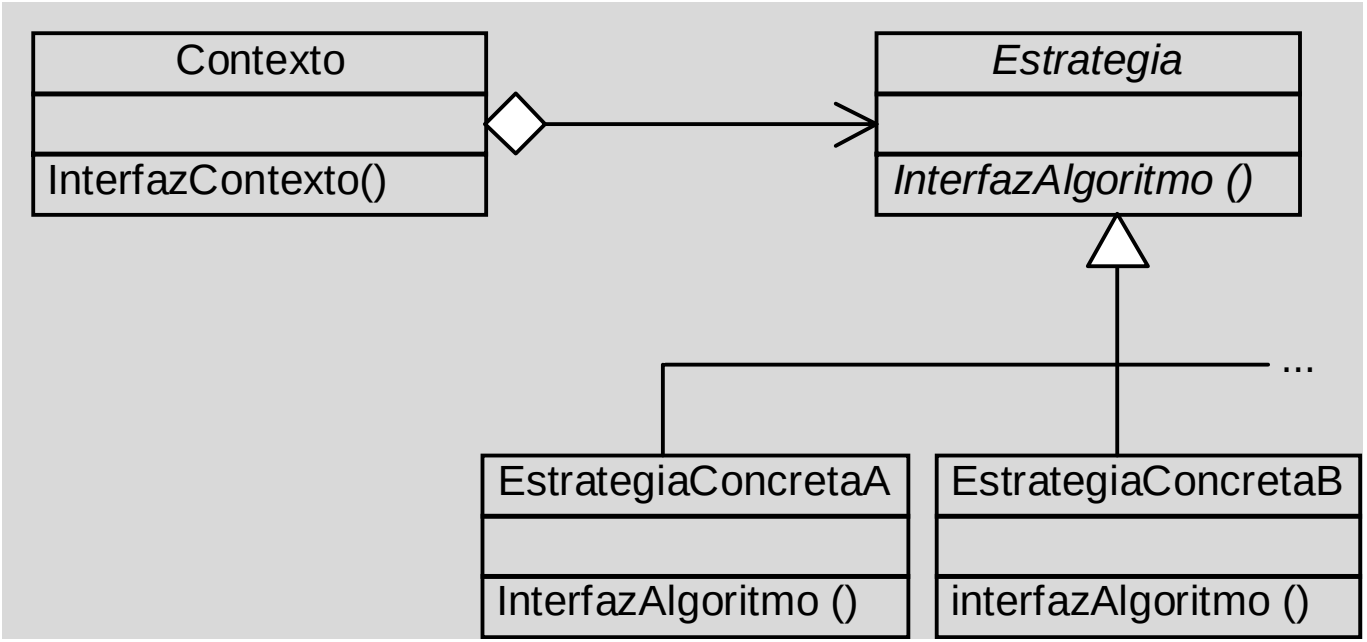
Patrones Comportamiento

Estrategia (Strategy)

- **Descripción**
  - **El patrón estrategia es un patrón de comportamiento que se utiliza para definir una familia de algoritmos, encapsularlos y hacerlos intercambiables.**
- **Elementos que lo componen**
  - ***Estrategia (Strategy)*:**
    - **Declara una interfaz común para todos los algoritmos soportados.**
  - ***Estrategia concreta (ConcreteStrategy)*:**
    - **Implementa el algoritmo utilizando el interfaz definido por la clase Estrategia.**
  - ***Contexto (Context)*:**
    - **Mantiene una referencia a un objeto del tipo Estrategia instanciado con una estrategia concreta**
    - **Delega en el objeto Estrategia el cálculo del algoritmo**
    - **Puede definir una interfaz que permita a la clase Estrategia el acceso a sus datos.**

Patrones Comportamiento
Estrategia (Strategy)

- **Elementos que lo componen**



Patrones Comportamiento

Estrategia (Strategy)

- **Ejemplo**

```
class PatronEstrategia
{
  public static void main (String[] arg)
  {
    Contexto c = new Contexto(5);
    c.calculaX();
    c.cambiaAlgoritmo (new AlgoritmoXPreciso());
    c.calculaX();
  }
}
```

```
class Contexto
{
  private AlgoritmoX algoritmoX;
  private int dato;

  public Contexto (int d)
  { dato = d;
    algoritmoX = new AlgoritmoXRapido();
  }

  public void calculaX()
  { algoritmoX.CalculaValor(dato); }

  public void cambiaAlgoritmo(AlgoritmoX alg)
  { algoritmoX = alg; }
}

abstract class AlgoritmoX
{ abstract void CalculaValor(int dato); }

class AlgoritmoXRapido extends AlgoritmoX
{
  void CalculaValor (int dato)
  { System.out.println ("Calculamos X de forma rápida"); }
}

class AlgoritmoXPreciso extends AlgoritmoX
{
  void CalculaValor (int dato)
  { System.out.println ("Calculamos X de forma precisa"); }
}
```

Patrones Comportamiento

Estrategia (Strategy)

- **Ventajas**
  - **Permite representar de forma sencilla familias de algoritmos factorizando sus partes comunes en una misma clase padre.**
  - **Podría hacerse realizando subclases del contexto, pero mezclaríamos algoritmo y contexto complicando su comprensión o modificación**
  - **De forma similar al patrón estado, la utilización del patrón estrategia evita tener que utilizar múltiples sentencias condicionales en el contexto para elegir el algoritmo adecuado.**
- **Inconvenientes**
  - **Al compartir todos los posibles algoritmos un mismo interfaz, puede pasar que las versiones más sencillas del algoritmo no utilicen todos los parámetros de inicialización que le ofrece el contexto, lo que significa una sobrecarga de comunicación entre contexto y algoritmo.**
  - **También puede ser un problema el número de objetos que hay que crear si tenemos muchas alternativas aunque puede solucionarse implementando las estrategias como objetos sin estado que puedan compartirse entre distintos contextos.**
- **Ejemplo en el API de Java**
  - **Gestores de composición (Layout Managers)**

Patrones Comportamiento

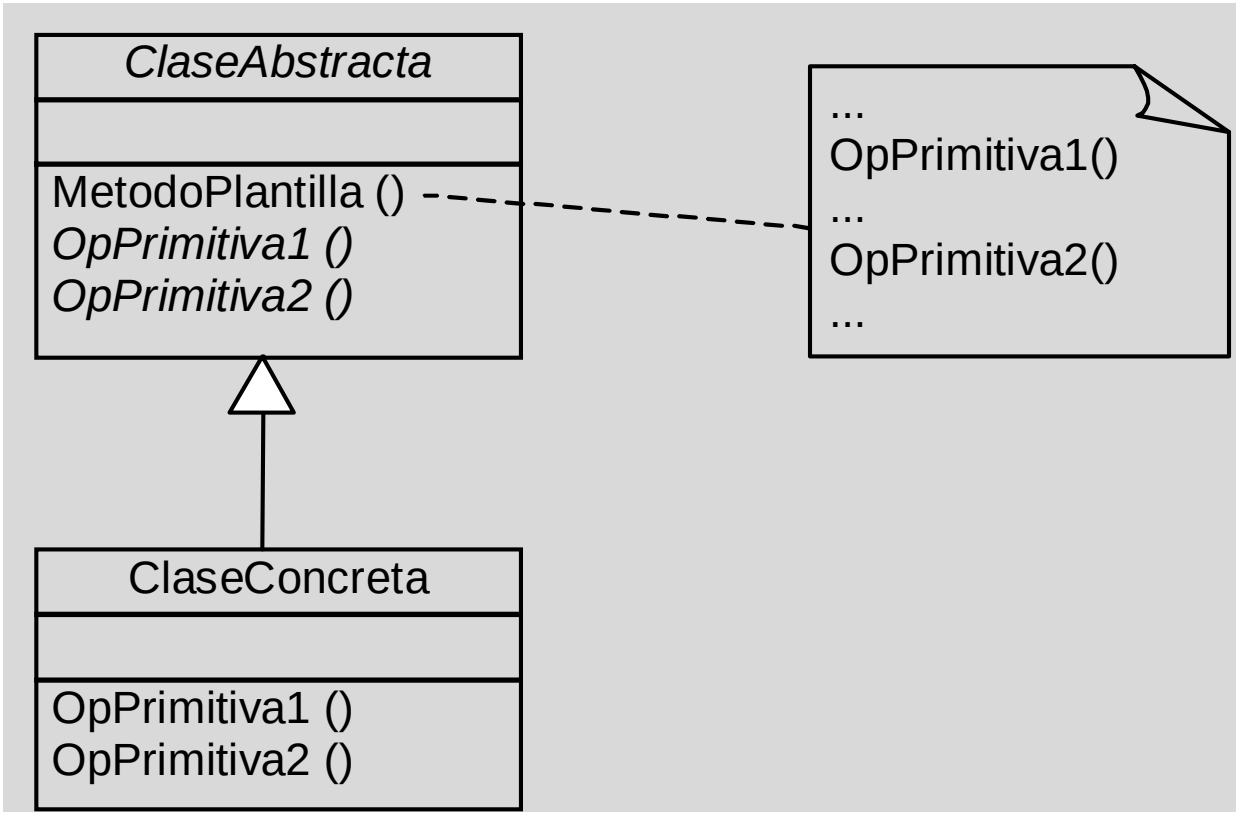
Método Plantilla (Template Method)

- **Descripción**
  - **Define el esqueleto de un algoritmo en una operación pero difiriendo alguno de los pasos a las subclases, de esta forma las subclases pueden cambiar ciertos aspectos del algoritmo, sin cambiar su estructura general.**
  - **El patrón método plantilla debe su nombre a que utilizarlo es como cubrir una plantilla o un formulario:**
    - **Una clase abstracta incluye un método concreto que hace de plantilla**
    - **El método plantilla llama a una serie de métodos abstractos que son como los huecos de la plantilla que deben cubrir las subclases.**
- **Elementos que lo componen**
  - ***Clase Abstracta (Abstract Class)*:**
    - **Define operaciones primitivas y abstractas que forman los pasos de un algoritmo y que las subclases deben implementar. También implementa el método plantilla que define el esqueleto del algoritmo. El método plantilla llama a las operaciones primitivas pero también puede utilizar otros elementos de la clase abstracta o de otras clases.**
  - ***Clase Concreta (Concrete Class)*:**
    - **Implementa las operaciones primitivas que definen el comportamiento específico del algoritmo para esta subclase.**

Patrones Comportamiento

Método Plantilla (Template Method)

- **Elementos que lo componen**



Patrones Comportamiento

Método Plantilla (Template Method)

- **Ejemplo**

```
class PatronPlantilla
{
  static void main(String[] args)
  {
    ClaseAbstracta ccA = new ClaseConcretaA();
    ClaseAbstracta ccB = new ClaseConcretaB();
    ccA.algoritmo();
    ccB.algoritmo();
  }
}
```

```
abstract class ClaseAbstracta
{
  final void algoritmo ()
  {
    paso1();
    paso2();
    paso3();
  }

  protected abstract void paso1();
  protected abstract void paso2();
  protected void paso3()
  { System.out.println ("Version común del paso 3"); }
}

class ClaseConcretaA extends ClaseAbstracta
{
  protected void paso1()
  { System.out.println ("Version A del paso 1"); }
  protected void paso2()
  { System.out.println ("Version A del paso 2"); }
}

class ClaseConcretaB extends ClaseAbstracta
{
  protected void paso1()
  { System.out.println ("Version B del paso 1"); }
  protected void paso2()
  { System.out.println ("Version B del paso 2"); }
  protected void paso3()
  { System.out.println ("Version B del paso 3"); }
}
```

Operaciones primitivas que DEBEN ser redefinidas en las subclases (huecos de la plantilla)

Hook methods. Métodos de enganche que PUEDEN ser definidos en las subclases

Patrones Comportamiento

Método Plantilla (Template Method)

- **Ventajas e inconvenientes**
  - **La principal ventaja del patrón método plantilla es que permite definir la estructura general de un algoritmo pero permitiendo que las subclases puedan cambiar ciertos aspectos, sin cambiar su estructura general.**
  - **Esta manera de actuar, en la que la clase padre es la encargada de llamar a las operaciones de las subclases y no al contrario, se conoce como el "principio de Hollywood", esto es, "no nos llame, ya le llamaremos".**
  - **Para que el desarrollo de este patrón sea efectivo es necesario indicar de forma clara qué métodos son de enganche y cuáles son operaciones abstractas.**

- **Ejemplo en el API de Java**
  - **La clase AbstractBorder del paquete java.swing.border implementa un borde vacío y sin tamaño que rodea a un componente gráfico (de la clase JComponent). Esta clase funciona como una clase base de la cual pueden derivar otras clases que implementen bordes. Las subclases deben dar una nueva implementación específica a los métodos definidos en AbstractBorder**

Patrones Comportamiento

Método Plantilla (Template Method)

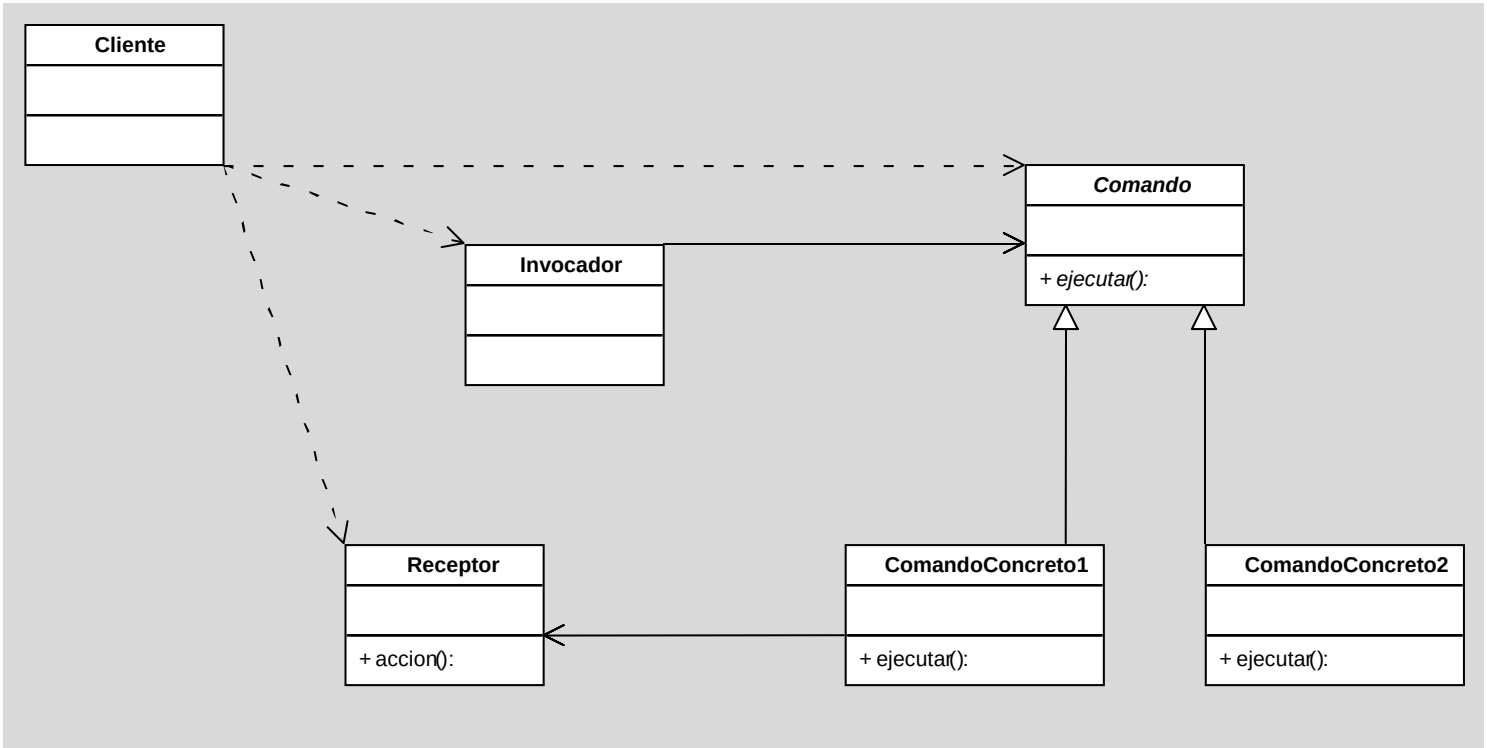
- **Patrón estrategia y patrón método plantilla**
  - **Similitudes**
    - **Se utilizan para implementar distintos tipos de algoritmos similares**
  - **Diferencias**
    - **El patrón estrategia utiliza la delegación (el contexto delega en la estrategia) para escoger entre las distintas variantes de un algoritmo (los algoritmos persiguen el mismo objetivo pero pueden ser completamente diferentes)**
    - **El patrón método plantilla utiliza la herencia para variar determinadas partes de un algoritmo dado (no se puede variar el esquema general del algoritmo)**

Patrones Comportamiento
Comando (Command)

- **Descripción**
  - **Permite encapsular una acción o requerimiento como un objeto de forma que pueda pasarse como parámetro al cliente que la utilice**
  - **Permite crear acciones complejas a partir de acciones elementales y realizar operaciones de "undo"**
- **Elementos que lo componen**
  - ***Comando (Command)*:**
    - **Declara un interfaz para ejecutar una operación**
  - ***ComandoConcreto (ConcreteCommand)*:**
    - **Implementa el interfaz Comando y define el enlace entre la acción y el objeto que la recibe**
  - ***Invocador (Invoker)*:**
    - **Solicita al comando que lleve a cabo la acción**
  - ***Receptor (Receiver)*:**
    - **Recibe el encargo de realizar las operaciones asociadas con una acción**
  - ***Cliente (Client)*:**
    - **Crea el comando concreto y fija su receptor**

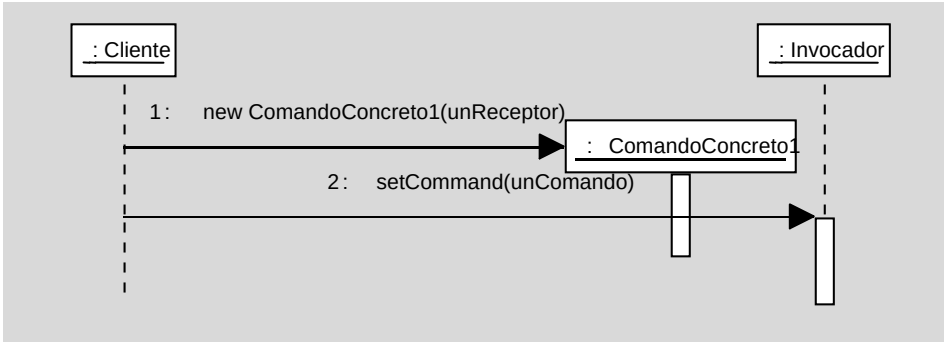
Patrones Comportamiento
Comando (Command)

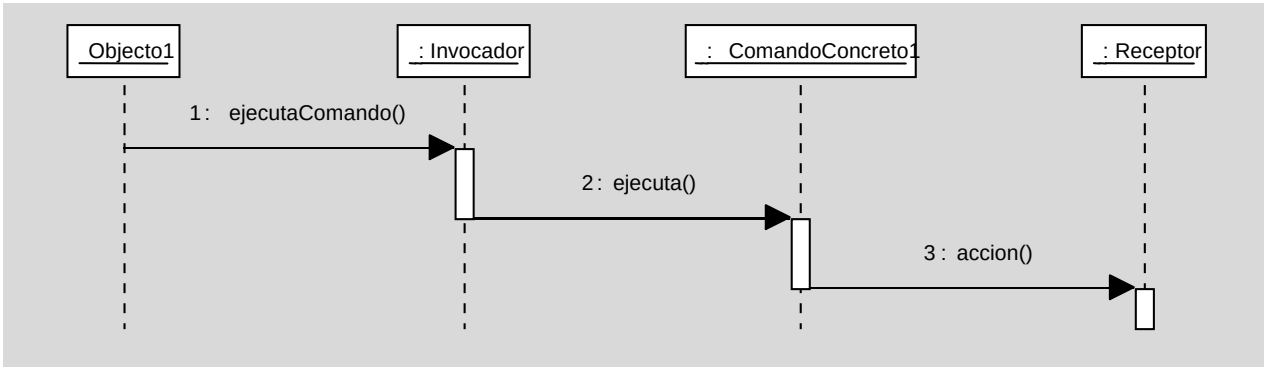
- **Elementos que lo componen**



Patrones Comportamiento
Comando (Command)

- **Elementos que lo componen**





Patrones Comportamiento
Comando (Command)

- **Ejemplo**

```
interface Comando
{
   void ejecuta();
}

class ComandoBorrarVocales implements Comando
{
  Receptor r;
  public ComandoBorrarVocales(Receptor r)
  { this.r = r; }

  public void ejecuta()
  { System.out.println("Ejecutamos la accion de borrar vocales");
    r.borrarVocales();
  }
}

class ComandoBorrarConsonantes implements Comando
{
  Receptor r;
  public ComandoBorrarConsonantes(Receptor r)
  { this.r = r; }

  public void ejecuta()
  { System.out.println("Ejecutamos la accion de borrar consonantes");
    r.borrarConsonantes();
  }
}

class Receptor
{
  StringBuffer s;

  Receptor(String s)
  { this.s = new StringBuffer(s); }

  public void borrarVocales()
  { System.out.println("Vocales borradas"); }

  public void borrarConsonantes()
  { System.out.println("Consonantes borradas"); }
}
```

```
class Invocador
{
  Comando c;

  public Invocador (Comando c)
  { this.c = c; }

  public void ejecutaComando()
  { c.ejecuta(); }

  public void setComando(Comando c)
  { this.c = c; }
}


class PatronComando
{
  public static void main (String[] args)
  {
    Receptor r = new Receptor("Hola Mundo 123");
    ComandoBorrarVocales c1 = new ComandoBorrarVocales(r);
    ComandoBorrarConsonantes c2 = new ComandoBorrarConsonantes(r);

    Invocador i1 = new Invocador(c1);
    Invocador i2 = new Invocador(c2);

    try
    {
      if (args[0].equals("vocales"))
        i1.ejecutaComando();
      else if (args[0].equals("consonantes"))
        i2.ejecutaComando();
      else System.out.println("Comando no definido");
    }
    catch (Exception e)
    {  System.out.println("No has indicado ningun comando"); }
  }
}
```

Patrones Comportamiento
Comando (Command)

- **Ventajas**
  - **El patrón desacopla el objeto que invoca la acción del objeto que finalmente la ejecuta**
  - **Al encapsular acciones en objetos estas pueden ser manipuladas, intercambiadas y extendidas como otros objetos**
  - **Resulta sencillo añadir nuevos comandos ya que sólo habría que crear su clase correspondiente, sin afectar a la otras clases**
  - **Es sencillo realizar comandos que se compongan de otros comandos utilizando patrones como el Composición**
  - **Se pueden implementar acciones de deshacer (undo) si el comando, además de tener un método ejecutar tiene un método deshacer**
- **Inconvenientes**
  - **Como todos los patrones aumenta el número de clases del sistema aumentando su complejidad, además el interfaz de todos los comandos tiene que ser igual.**
- **¿Qué acciones realiza un comando?**
  - **Comando simple**
    - **Actúa simplemente como un enlace entre las llamadas del invocador y las acciones del receptor permitiendo desacoplar invocador y receptor**
  - **Comando inteligente**
    - **El comando es el encargado de realizar las acciones sin necesidad de delegar en un receptor (que es posible que no exista)**

Patrones Comportamiento
Comando (Command)

- **Soporte en Java**
  - **Java define un interfaz Action en el paquete javax.swing**
  - **Este interfaz hereda del interfaz ActionListener del paquete java.awt.event por lo que si se implementa el interfaz hay que implementar el método actionPerformed**
  - **Además el interfaz define una serie de constantes de clase que nos permiten definir las características de la acción que encapsulan**

Field Summary	
static String	**ACCELERATOR_KEY** The key used for storing a KeyStroke to be used as the accelerator for the action.
static String	**ACTION_COMMAND_KEY** The key used to determine the command String for the ActionEvent that will be created when an Action is going to be notified as the result of residing in a Keymap associated with a JComponent.
static String	**DEFAULT** Not currently used.
static String	**LONG_DESCRIPTION** The key used for storing a longer String description for the action, could be used for context-sensitive help.
static String	**MNEMONIC_KEY** The key used for storing a KeyEvent to be used as the mnemonic for the action.
static String	**NAME** The key used for storing the String name for the action, used for a menu or button.
static String	**SHORT_DESCRIPTION** The key used for storing a short String description for the action, used for tooltip text.
static String	**SMALL_ICON** The key used for storing a small Icon, such as ImageIcon, for the action, used for toolbar buttons.

Patrones Comportamiento
Comando (Command)

- **Soporte en Java (cont.)**
  - **Los componentes gráficos de Swing permiten que se les asocien acciones**
    - **A traves del constructor: JButton (Action a)**
    - **A través de métodos de escritura: public void setAction(Action a)**
  - **Funcionamiento**
    - **Creamos la clase que representa la acción:**

```
class OpenAction extends AbstractAction
  { public OpenAction()
    { putValue(Action.NAME,"Open");
      putValue(Action.SMALL_ICON,new ImageIcon(getClass().getResource("/icons/Open.gif")));
      putValue(Action.SHORT_DESCRIPTION, "Open VAL file");
    }

    public void actionPerformed(ActionEvent e)
    { myData = obtenerValidationData();
      setTitle("Application (default data)");
    }
  }
```

    - **Creamos una instancia**
      - **private Action openAction = new OpenAction();**
    - **La asociamos a componentes gráficos de nuestra interfaz**
      - **jButton2.setAction(openAction);**
      - **jMenuItem5.setAction(openAction);**

Bibliografía

- **Bibliografía fundamental**
  - **Gamma, E., Helm, R., Johnson, R., Vlissides, J. Desing Patterns: Elements of Reusable Object-Oriented Software, Addison-Wesley, Reading, MA, 1995.**
  - **Grand, M., Patterns in Java: A catalog of reusable design patterns illustrated with UML, John Wiley and Sons, New York, 1998.**
  - **Cooper, J.W. Java Design Patterns: A Tutorial, Addison-Wesley, Reading, MA, 2000.**
  - **Brown, W.J., Malveau, R.C., McCormick, H.W., Mowbray, T.J., Antipatterns: Refactoring Software, Architectures and Projects in Crisis, John Wiley and Sons, New York, 1998.**
- **Bibliografía complementaria**
  - **Larman, C. "Applying UML and Patterns", Prentice-Hall PTR, Upper Saddle River, NJ, 2005**
  - **Rising, Linda, The Pattern Almanac 2000, Addison-Wesley, Boston, 2000.**
  - **Coplien, J. "Advanced C++ programming styles and idioms", 1991.**
  - **Cunningham, W., Beck, K., "Using pattern languages for object-oriented programs", OOPSLA'87, Orlando, 1987.**
- **Bibliografía en internet**
  - **Patterns Home Page, URL: http://hillside.net/patterns/**
  - **Design Patterns, Pattern Languages, and Frameworks. URL: http://www.cs.wustl.edu/~schmidt/patterns.html**
  - **Object-Oriented Language: Patterns. URL: http://www.cetus-links.org/oo_patterns.html**
  - **Eckel, B. "Thinking in Patterns with Java". URL: http://www.mindview.net/Books/TIPatterns/**