Análisis y Diseño de Software

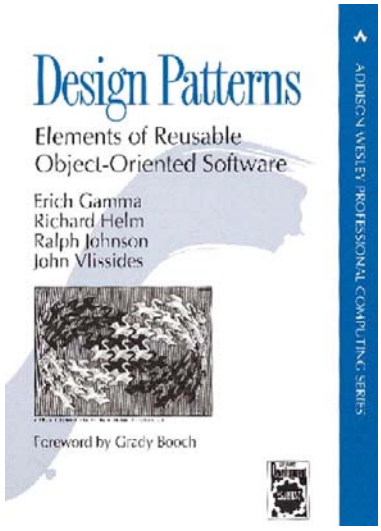
# Patrones de Diseño

Jesús García Molina
Departamento de Informática y Sistemas
Universidad de Murcia
http://dis.um.es/~jmolina

## Contenidos

- Introducción a los patrones de diseño GoF 
- Patrones de **creación**
  - Factoría Abstracta, Builder, Método Factoría, Prototipo, Singleton
- Patrones **estructurales**
  - Adapter, Bridge, Composite, Decorador, Fachada, Flyweight, Proxy
- Patrones de **comportamiento**
  - Cadena de Responsabilidad, Command, Iterator, Intérprete, Memento, Mediador, Observer, Estado, Estrategia, Método Plantilla, Visitor

## Texto básico

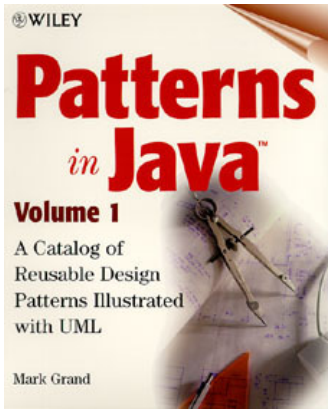



### Design Patterns

E. Gamma, R. Helm, T. Johnson, J. Vlissides
Addison-Wesley, 1995
(Versión española de 2003)

3

## Texto sobre patrones en Java



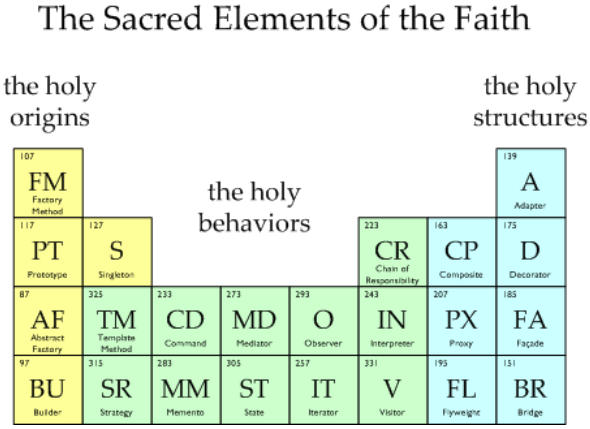

### Pattern in Java, vol. 1

Segunda edición

Mark Grand
John Wiley, 2002

4

# Sitio web sobre patrones en C++

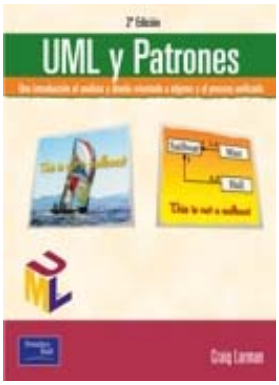



Houston Design Pattern

http://home.earthlink.net/~huston2/dp/patterns.html

5

# Patrones GoF básicos y Proceso UML




UML y Patrones
Introducción al análisis y diseño orientado a objetos
Craig Larman
Prentice-Hall, 2002

6

## Arquitectura software y patrones

"Una arquitectura orientada a objetos bien estructurada está llena de patrones. La calidad de un sistema orientado a objetos se mide por la atención que los diseñadores han prestado a las colaboraciones entre sus objetos."

"Los patrones conducen a arquitecturas más pequeñas, más simples y más comprensibles"

G. Booch

7

## Diseño orientado a objetos

- "Diseñar software orientado a objetos es difícil pero diseñar software orientado a objetos reutilizable es más difícil todavía. Diseños generales y flexibles son muy difíciles de encontrar la primera vez"
- ¿Qué conoce un programador experto que desconoce uno inexperto?

**Reutilizar soluciones que funcionaron en el pasado:**
**Aprovechar la experiencia**

8

## Patrones

- Describen un problema recurrente y una solución.
- Cada patrón nombra, explica, evalúa un diseño recurrente en sistemas OO.
- Elementos principales:
  - **Nombre**
  - **Problema**
  - **Solución:** Descripción abstracta
  - **Consecuencias**

9

## Patrones

- ***Patrones de código***
  - Nivel de lenguaje de programación
- ***Frameworks***
  - Diseños específicos de un dominio de aplicaciones.
- ***Patrones de diseño***
  - Descripciones de clases cuyas instancias colaboran entre sí que deben ser adaptados para resolver problemas de diseño general en un particular contexto.
  - Un patrón de diseño identifica: ***Clases***, ***Roles, Colaboraciones*** y la ***distribución de responsabilidades***.

10

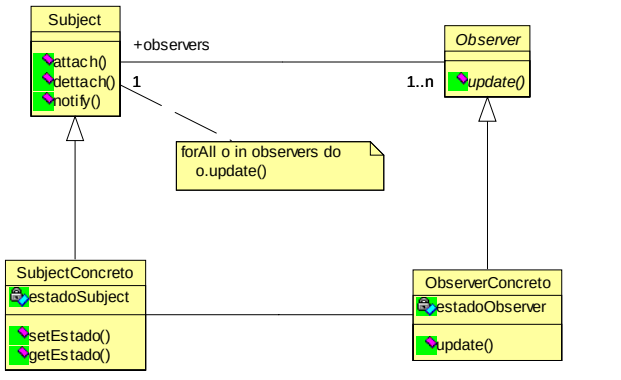
# Patrones y *frameworks*

- Un *framework* es una colección organizada de clases que constituyen un diseño reutilizable para una clase específica de software.
- Necesario adaptarlo a una aplicación particular.
- Establece la arquitectura de la aplicación
- Diferencias:
  - Patrones son más abstractos que los *frameworks*
  - Patrones son elementos arquitecturales más pequeños
  - Patrones son menos especializados



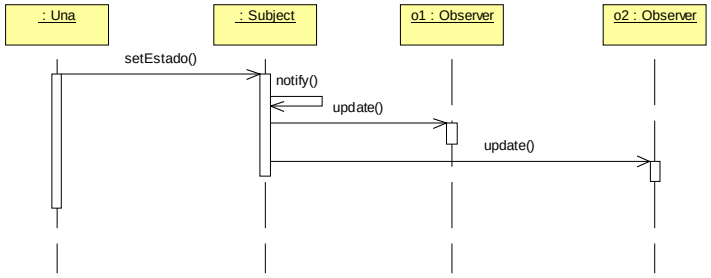
# Observer

**Estructura**



# Observer

**Colaboración**

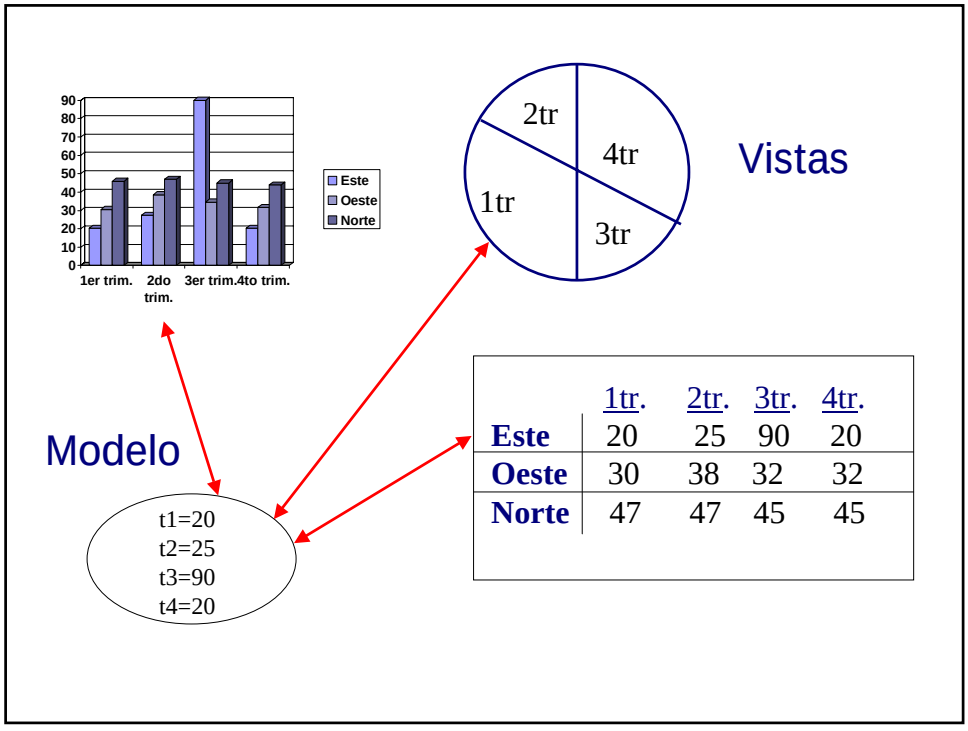



13

# Modelo-Vista-Control

- Utilizado para construir interfaces de usuario en Smalltalk-80.
- Basado en tres tipos de objetos:
  - **Modelo**: objetos del dominio
  - **Vista**: objetos presentación en pantalla (interfaz de usuario)
  - **Controlador**: define la forma en que la interfaz reacciona a la entrada del usuario.
- Desacopla el modelo de las vistas.

14




# Modelo-Vista-Control

- Utiliza los siguientes patrones:
  - ***Observer***
  - ***Composite***
  - ***Strategy***
  - ***Decorator***
  - ***Factory method***

16

## Plantilla de definición

- **Nombre**
- **Propósito**
  - ¿Qué hace?, ¿Cuál es su razón y propósito?, ¿Qué cuestión de diseño particular o problema aborda?
- **Sinónimos**
- **Motivación**
  - Escenario que ilustra un problema particular y cómo el patrón lo resuelve.

17

## Plantilla de definición

- **Aplicabilidad**
  - ¿En qué situaciones puede aplicarse?, ¿Cómo las reconoces?, ejemplos de diseños que pueden mejorarse.
- **Estructura**
  - Diagramas de clases que ilustran la estructura
- **Participantes**
  - Clases que participan y sus responsabilidades
- **Colaboraciones**
  - Diagramas de interacción que muestran cómo colaboran los participantes.

18

## Plantilla de definición

- **Consecuencias**
  - ¿Cómo alcanza el patrón sus objetivos?, ¿Cuáles son los compromisos y resultados de usar el patrón?, Alternativas, Costes y Beneficios
- **Implementación**
  - Técnicas, heurísticas y consejos para la implementación
  - ¿Hay cuestiones dependientes del lenguaje?
- **Ejemplo de Código**
- **Usos conocidos**
- **Patrones relacionados**

19

## Clasificación

| | | **Propósito** | | |
|---|---|---|---|---|
| | | *Creación* | *Estructural* | *Comportamiento* |
| **Ámbito** | *Herencia* | Factory Method | Adapter | Interpreter<br>Template Method |
| | *Compo-sición* | Abstract Factory<br>Builder<br>Prototype<br>Singleton | Adapter<br>Bridge<br>Composite<br>Decorator<br>Facade<br>Flyweight<br>Proxy | Chain of Responsability<br>Command<br>Iterator<br>Mediator<br>Memento<br>Observer<br>State<br>Strategy<br>Visitor |

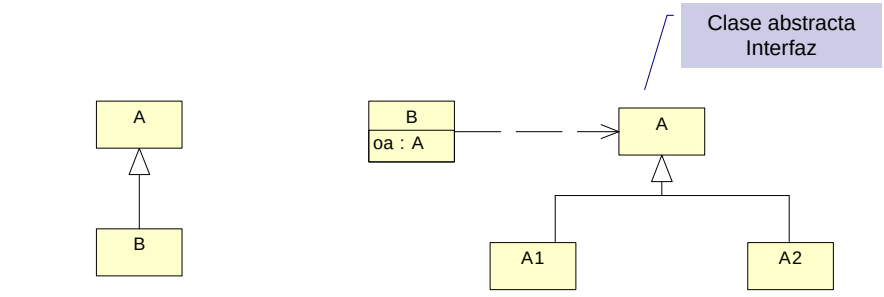
20

## ¿A qué ayudan los patrones?

- Encontrar clases de diseño
- Especificar interfaces
- *"Programar hacia la interfaz, no hacia la implementación"*
  - No declarar variables de clases concretas sino abstractas.
  - Patrones de creación permiten que un sistema esté basado en términos de interfaces y no en implementaciones.
- Favorecen la reutilización a través de la composición en vez de la herencia

21

## Herencia vs. Clientela




Relación fija

Reuso "caja blanca"

Relación variable

Reuso "caja negra"

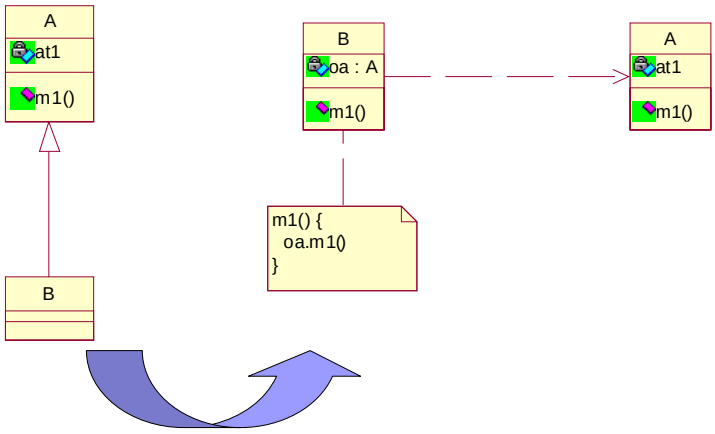
22

# ¿A qué ayudan los patrones?

- Utilizan bastante la *delegación*
  - Forma de hacer que la composición sea tan potente como la herencia.
  - Un objeto receptor delega operaciones en su delegado
  - Presente en muchos patrones: *State*, *Strategy*, *Visitor*,..
  - Caso extremo de composición, muestra que siempre puede sustituirse la herencia por composición.

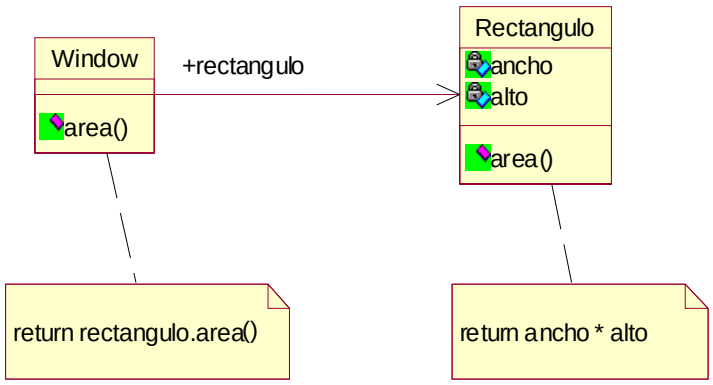
23

# Delegación




24

## Delegación




25

## ¿A qué ayudan los patrones?

- La clave para la reutilización es anticiparse a los nuevos requisitos y cambios, de modo que los sistemas evolucionen de forma adecuada.
- Cada patrón permite que algunos aspectos de la estructura del sistema puedan cambiar de forma independiente a otros aspectos.
- Facilitan reuso interno, extensibilidad y mantenimiento.

26

## Causas comunes de rediseño

i) Crear un objeto indicando la clase.
ii) Dependencia de operaciones específicas
iii) Dependencia de plataformas hardware o software
iv) Dependencia sobre representación de objetos.
v) Dependencias de algoritmos
vi) Acoplamiento fuerte entre clases
vii) Extender funcionalidad mediante subclases
viii) Incapacidad de cambiar clases convenientemente

27

## Patrones frente a esos peligros

i) *Abstract factory, Method factory, Prototype*
ii) *Chain of Responsability, Command*
iii) *Abstract factory, Bridge*
iv) *Abstract factory, Bridge, Memento, Proxy,*
v) *Builder, Iterator, Strategy, Template Method, Visitor*
vi) *Abstract factory, Bridge, Chain of Responsability*, *Command, Facade, Mediator, Observer*
vii) *Bridge, Chain of Responsability*, *Composite, Decorator, Observer, Strategy*
viii) *Adapter, Decorator, Visitor*

28

## ¿Cómo seleccionar un patrón?

- Considera de que forma los patrones resuelven problemas de diseño
- Lee la sección que describe el propósito de cada patrón
- Estudia las interrelaciones entre patrones
- Analiza patrones con el mismo propósito
- Examina las causas de rediseñar
- Considera que debería ser variable en tu diseño

29

## ¿Cómo usar un patrón?

- Lee el patrón, todos sus apartados, para coger una visión global.
- Estudia la *Estructura*, *Participantes* y *Colaboraciones*
- Mira el ejemplo de código
- Asocia a cada participante del patrón un elemento software de tu aplicación.
- Implementa las clases y métodos relacionados con el patrón.

30

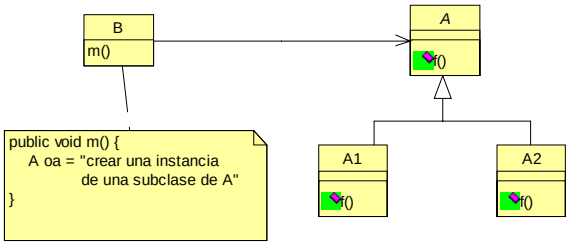
## Contenidos

- Introducción a los patrones de diseño GoF
- Patrones de **creación**
  - Factoría Abstracta, Builder, Método Factoría, Prototipo, Singleton
- Patrones **estructurales**
  - Adapter, Bridge, Composite, Decorador, Fachada, Flyweight, Proxy
- Patrones de **comportamiento**
  - Cadena de Responsabilidad, Command, Iterator, Intérprete, Memento, Mediador, Observer, Estado, Estrategia, Método Plantilla, Visitor

31

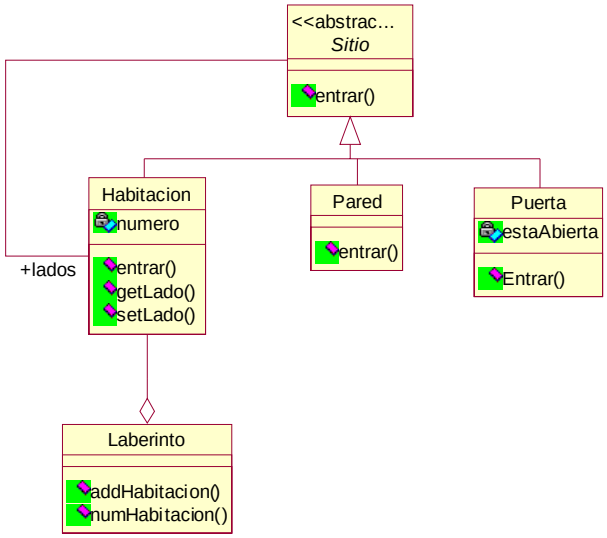
## Patrones de Creación

- Abstraen el proceso de creación de objetos.
- Ayudan a crear sistemas independientes de cómo los objetos son creados, compuestos y representados.
- El sistema conoce las clases abstractas
- Flexibilidad en **qué** se crea, **quién** lo crea, **cómo** se crea y **cuándo** se crea.




32

# Ejemplo: Laberinto




33

```
public class JuegoLaberinto {
    public Laberinto makeLaberinto () {
        Laberinto unLab = new Laberinto();
        Habitacion h1 = new Habitacion(1);
        Habitacion h2 = new Habitacion(2);
        Puerta unaPuerta = new Puerta(1,2)

        unLab .addHabitacion (h1);
        unLab .addHabitacion (h2);

        h1.setLado(Norte, new Pared() );
        h1.setLado(Sur, new Pared() );
        h1.setLado(Este, new Pared() );
        h1.setLado(Oeste,unaPuerta);

        h2.setLado(Norte, new Pared);
        …
        return unLab;}
}
```

## Ejemplo: Laberinto

- Poco flexible

  ¿Cómo crear laberintos con otros tipos de elementos como *habitacionesEncantadas* o *puertasQueEscuchan*?

- Patrones de creación permiten **eliminar referencias explícitas a clases concretas** desde el código que necesita crear instancias de esas clases.

35

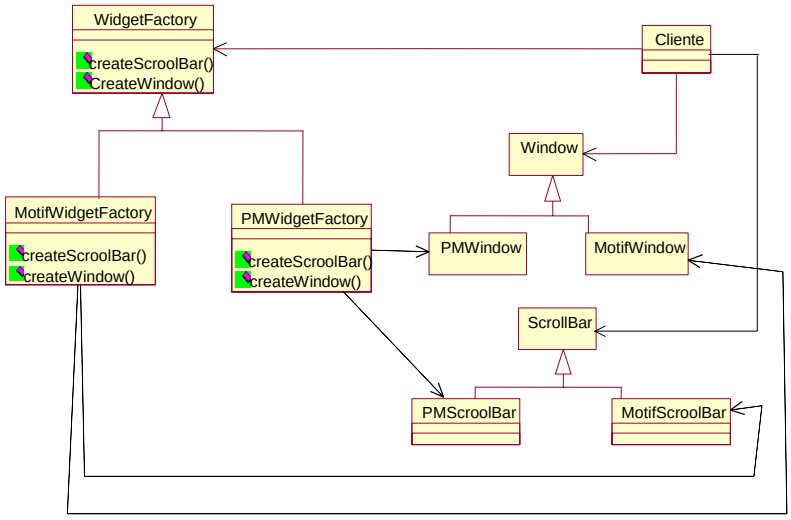
## Abstract Factory (Factoría Abstracta)

- **Propósito**
  - Proporcionar una interfaz para crear **familias de objetos** relacionados o dependientes sin especificar la clase concreta
- **Motivación**
  - Un *toolkit* interfaz de usuario que soporta diferentes formatos: Windows, Motif, X-Windows,..

36

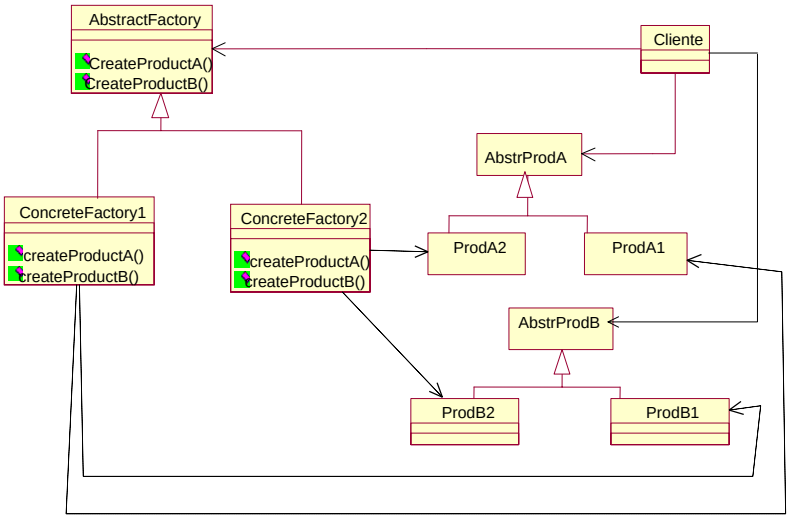
# Factoría Abstracta

**Motivación**




# Factoría Abstracta

**Estructura**




38

## Factoría Abstracta

- **Aplicabilidad**
  - Un sistema debería ser independiente de cómo sus productos son creados, compuestos y representados
  - Un sistema debería ser configurado para una familia de productos.
  - Una familia de objetos “productos” relacionados es diseñada para ser utilizado juntos y se necesita forzar la restricción.

39

## Factoría Abstracta

- **Consecuencias**
  - Aísla a los clientes de las clases concretas de implementación.
  - Facilita el intercambio de familias de productos.
  - Favorece la consistencia entre productos
  - Es difícil soportar nuevos productos.

40

# Factoría Abstracta

- **Implementación**
  - Factorías como *singleton.*
  - Se necesita una subclase de *AbstractFactory* por cada familia de productos que redefine un conjunto de *métodos factoría*.
    - Posibilidad de usar el patrón *Prototype*.
  - Definir factorías extensibles: *AbstractFactory* sólo necesita un método de creación.



# Ejemplo 1: Laberinto

```
public class FactoriaLaberinto {
    public Laberinto makeLaberinto { return new Laberinto();}
    public Pared  makePared {return new Pared();};
    public Habitacion  makeHabitacion(int n)
            { return new Habitacion(n); };
    public Puerta makePuerta (Habitacion h1, Habitacion h2)
           {return new Puerta(h1,h2);}
}
```

## Ejemplo 1: Laberinto

```
public class JuegoLaberinto {
    public Laberinto makeLaberinto (FactoriaLaberinto factoria) {
            Laberinto unLab = factoria.makeLaberinto();
            Habitacion h1 = factoria.makeHabitacion(1);
            Habitacion h2 = factoria.makeHabitacion(2);
            Puerta unaPuerta = factoria.makePuerta(h1,h2);
            unLab.addHabitacion(h1);
            unLab.addHabitacion(h2);
            h1.setLado(Norte, factoria.makePared() );
            h1.setLado(Este, unaPuerta)
            …
            h2.setLado(Oeste, unaPuerta);
            h2.setLado(Sur, factoria.makePared() )
            return unLab; }
}
```

43

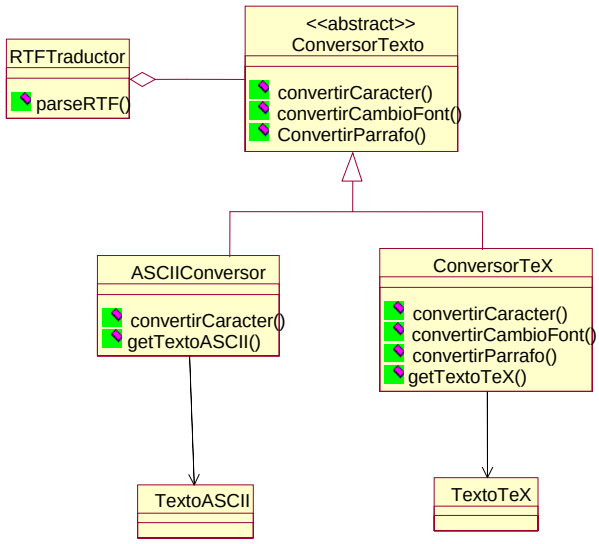
## Builder (Constructor)

- **Propósito**
  - Separa la **construcción de un objeto complejo** de su representación, así que el mismo proceso de construcción puede crear diferentes representaciones.
- **Motivación**
  - Un traductor de documentos RTF a otros formatos. ¿Es posible añadir una nueva conversión sin modificar el traductor?

44

# Builder

**Motivación**




45

# Builder

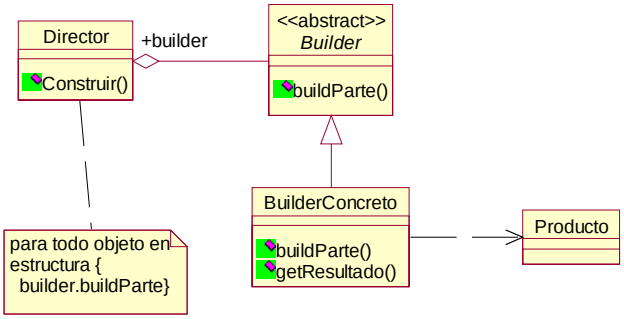
```
parseRTF {
   while (t = “obtener siguiente token”) {
      switch(t.tipo) {
         case Car: conversor.convertirCaracter(t.char);break;
         case Font:
                conversor.convertirCambioFont(t.font);break;
         case Par: conversor.convertirParrafo(); break
        }
   }
}
```

```
ConversorTeX ctex = new ConversorTeX()
RTFTraductor trad = new RTFTraductor (ctex, doc)
trad.parseRTF(doc)
TextoTex texto = conversorTex.getTextoTeX():
```

46

# Builder

**Estructura**




47

# Builder

- **Aplicabilidad**
  - Cuando el algoritmo para crear un objeto complejo debe ser independiente de las piezas que conforman el objeto y de cómo se ensamblan.
  - El proceso de construcción debe permitir diferentes representaciones para el objeto que se construye.

48

## Builder

- **Consecuencias**
  - Permite cambiar la representación interna del producto.
  - Separa el código para la representación y para la construcción.
  - Los clientes no necesitan conocer nada sobre la estructura interna.
  - Diferentes “directores” pueden reutilizar un mismo “builder”
  - Proporciona un control fino del proceso de construcción.



## Builder

- **Implementación**
  - La interfaz de *Builder* debe ser lo suficientemente general para permitir la construcción de productos para cualquier *Builder* concreto.
  - La construcción puede ser más complicada de añadir el nuevo *token* al producto en construcción.
  - Los métodos de *Builder* no son abstractos sino vacíos.
  - Las clases de los productos no siempre tienen una clase abstracta común.

# Ejemplo: Laberinto

```
public class BuilderLaberinto {
      public void buildLaberinto () { };
      public void   buildPuerta (int r1, int r2) { };
      public void buildHabitacion(int n) { };
      public Laberinto getLaberinto() { };
      protected BuilderLaberinto() { };
}
```

Las “paredes” forman parte de la representación interna

51

# Ejemplo: Laberinto

```
public class JuegoLaberinto {
      public Laberinto  makeLaberinto (BuilderLaberinto builder) {
            builder.buildLaberinto();

            builder.buildHabitacion(1);
            builder.buildHabitacion(2);
            builder.buildPuerta(1,2);

            return builder.getLaberinto(); }
}

class BuilderLabNormal extends BuilderLaberinto {
     private Laberinto labActual;
     // se define la construcción de un laberinto con determinado tipo de
   // puertas y habitaciones
     ...
}
```

**Oculta la estructura interna**

52

```
class BuilderLabNormal extends BuilderLaberinto {
    private Laberinto labActual;
    public BuilderLabNormal () {  labActual = null;   }
    public void buildLaberinto ()  {  labActual = new Laberinto(); }
    public Laberinto getLaberinto () {  return labActual;  }
    public void buildHabitacion (int n) {
            if ( ! labActual.tieneHabitacion(n) ) {
                Habitacion hab = new Habitacion (n);
                labActual.addHabitacion (hab);
                hab.setLado (Norte, new Pared());
                hab.setLado (Sur, new Pared());
                hab.setLado (Este, new Pared());
                hab.setLado (Oeste, new Pared());  }
            }
     public void buildPuerta (int n1, int n2)) {
            Habitacion h1 = labActual.getHabitacion(n1);
            Habitacion h2 = labActual.getHabitacion(n2);
            Puerta p = new Puerta(h1,h2);
            h1.setLado(paredComun(h1,h2), p);
            h2.setLado(paredComun(h2,h1), p);  }
}
```

```
Laberinto lab;
JuegoLaberinto juego;
BuilderLabNormal builder;

lab = juego.makeLaberinto(builder);
```

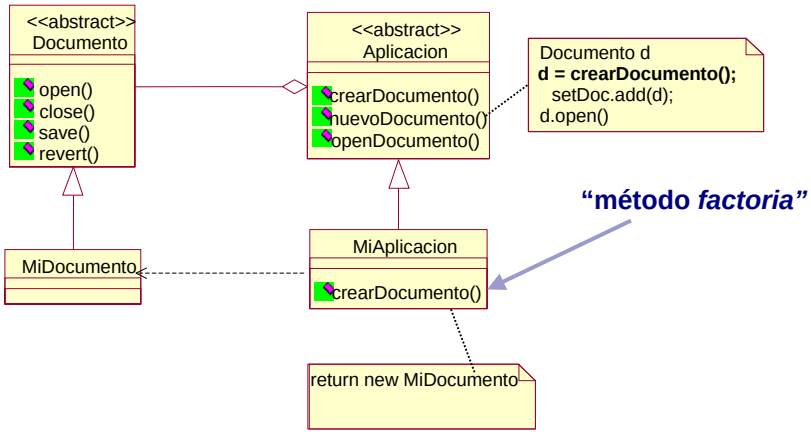
# Factory Method (Método Factoría)

- **Propósito**
  - Define un interfaz para crear un objeto, pero permite a las subclases decidir la clase a instanciar: ***instanciación diferida a las subclases.***
- **Motivación**
  - Una clase C cliente de una clase abstracta A necesita crear instancias de subclases de A que no conoce.
  - En un *framework* para aplicaciones que pueden presentar al usuario documentos de distinto tipo: clases Aplicación y Documento.
  - Se conoce **cuándo** crear un documento, no se conoce de **qué** tipo.

54

# Método Factoría

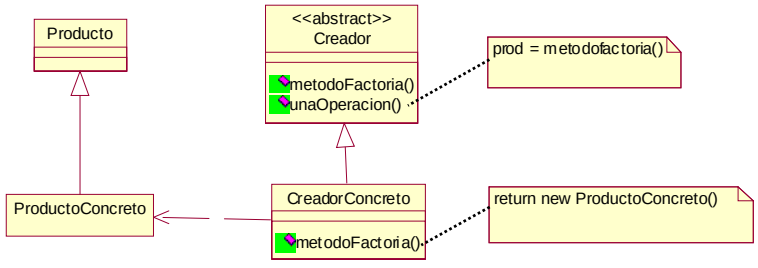
## Motivación






# Método Factoría

## Estructura



## Método Factoría

- **Aplicabilidad**
  - Una clase no puede anticipar la clase de objetos que debe crear.
  - Una clase desea que sus subclases especifiquen los objetos que debe crear.

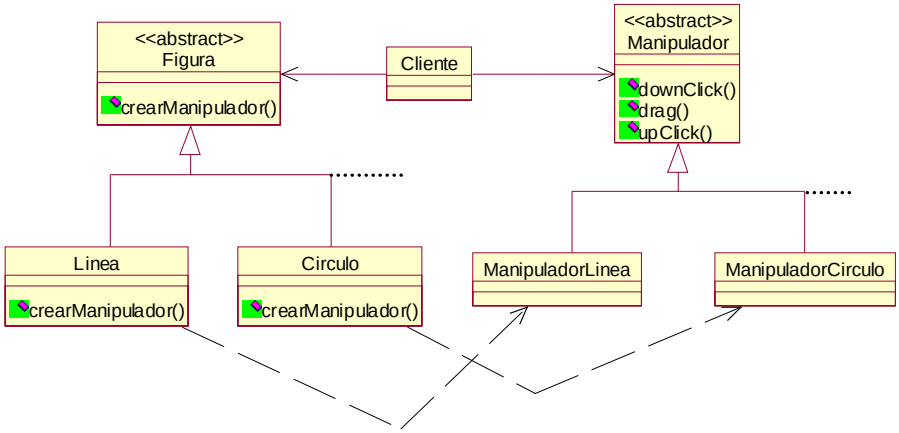
57

## Método Factoría

- **Consecuencias**
  - Evita ligar un código a clases específicas de la aplicación.
  - Puede suceder que las subclases de *Creador* sólo se crean con el fin de la creación de objetos.
  - Mayor flexibilidad en la creación: subclases ofreciendo versiones extendidas de un objeto.
  - El método factoría puede ser invocado por un cliente, no sólo por la clase *Creador*: jerarquías paralelas

58

# Método Factoría




59

# Método Factoría

- **Implementación**
  - Dos posibilidades
    - `Creador` es una clase abstracta con un método factoría abstracto.
    - `Creador` es una clase concreta que ofrece una implementación por defecto del método factoría.
  - El método factoría puede tener un parámetro que identifica a la clase del objeto a crear.
  - Se evita crear subclases de *Creador* con:
    - Metaclases (Smalltalk y Java)
    - Genericidad (C++)

60

## Método Factoría y Genericidad C++

```
template <class elProducto>
class Creador {
   public:
        virtual Producto* crearProducto {
                return new elProducto;
        }
}
class MiProducto : public Producto { ... }

Creador<MiProducto>  miCreador;
```

61

## Metaclases

- Una clase puede ser considerada un objeto:
  - *¿Cuál es su clase?*
- *Metaclase*
  - Clase que describe clases, sus instancias son clases.
  - Propiedades: lista de atributos, lista de variables de clase, lista de métodos, lista de métodos de clase.
- *Java, Ruby y Smalltalk* tienen metaclases
- Útil en programación avanzada, cuando se manejan entidades software, p.e. depuradores, inspectores, browsers,..

62

## Metaclases

- Metainformación
- Clases, atributos, métodos, etc., son representados por clases.
- Posibilidades
  - Crear o modificar clases en tiempo de ejecución.
  - Parametrizar métodos por clases o métodos.
  - Consultar estructura y comportamiento de una clase.
  - ...

63

## Metaclases

```
void metodo1 (Class c) {
    “crear instancia de la clase c”
}
void metodo2 (String c) {
    “obtener instancia de metaclase para clase c”
    “crear instancia de la clase c”
}
void metodo3 (Metodo m) {
    “invocar a m”
}
```

64

## Reflexión o Instropección en Java

- La clase **`Class`** es el punto de arranque de la reflexión.
- `Class` representa clases, incluye métodos que retornan información sobre la clase:
  - `getFields, getMethods, getSuperClass, getClasses, getConstructors, getModifiers, getInterfaces,..`
- *Field* representa atributos (hereda de *Member*):
  - *`getType, getName, get, set,..`*
- *Method* representa métodos (hereda de *Member*):
  - *`getName, getReturnType, getParameterTypes,..`*

65

## Reflexión o Instropección en Java

- Existe una instancia de `Class` por cada tipo (interface o clase).
- Cuatro formas de obtener un objeto *Class*

Cuenta oc; Class clase

1) clase = oc.getClass()
2) clase = Cuenta.class
3) clase = Class.forName("Cuenta")
4) Utilizando un método que retorna objetos *Class*

- **Ejercicio**: Escribe una clase que imprima los nombres de los campos y métodos de una clase dada.

66

## Reflexión o Instropección en Java

- Crear una instancia de una clase de la que se conoce su nombre, *nomClase*:

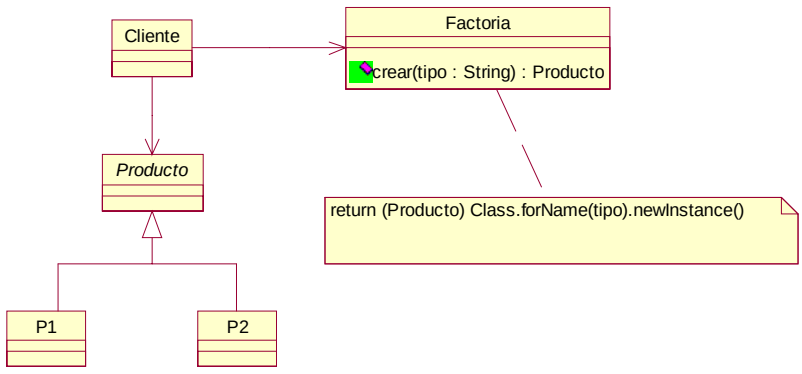
  Class.forName(nomClase).newInstance()
- Construir un mensaje (método *invoke* en *Method*)

  invoke(Object sobreEste, Object[] params)

  *getMethod(nom, parametros)* retorna el método con nombre *nom* de la clase.



## Factoría y Metaclases



# Ejemplo: Laberinto

```
public class JuegoLaberinto {
    public Laberinto  nuevoLaberinto () {...};

    public Laberinto crearLaberinto() { return new Laberinto(); };
    public Habitacion crearHabitacion(int n)
          { return new Habitacion(n); };
    public Pared crearPared() { return new Pared(); };
    public Puerta crearPuerta(Habitacion h1, Habitacion h2)
                { return new Puerta(h1,h2); };

}
```

69

# Ejemplo: Laberinto

```
public Laberinto nuevoLaberinto () {
        Laberinto unLab = crearLaberinto();
        Habitacion h1 = crearHabitacion(1);
        Habitacion h2 = crearHabitacion(2);
        Puerta unaPuerta = crearPuerta(h1,h2);
        unLab.addHabitacion(h1);
        unLab.addHabitacion(h2);
        h1.setLado(Norte, crearPared() );
        h1.setLado(Este, unaPuerta)
        ..
        h2.setLado(Oeste, unaPuerta);
        h2.setLado(Sur, crearPared() );
        return unLab; }
}
```

70

## Ejemplo: Laberinto

```
public class JuegoLaberintoEncantado  extends JuegoLaberinto {
        public Habitacion crearHabitacion(int n) {
            return new HabitacionEncantada(n); }
        public Pared crearPared() {
            return new ParedEncantada(); }
        public Puerta crearPuerta( Habitacion h1, Habitacion h2) {
            return new PuertaEncantada(h1, h2); }
}
```

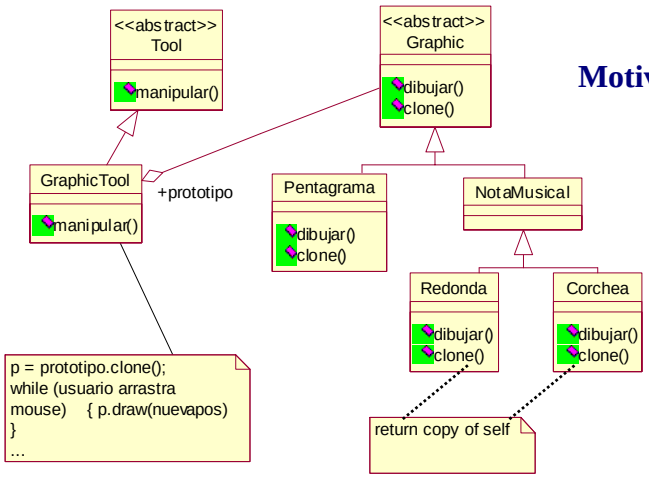
71

## Prototype (Prototipo)

- **Propósito**
  - Especificar los tipos de objetos a crear utilizando una **instancia prototípica**, y crear nuevas instancias mediante **copias del prototipo**.
- **Motivación**
  - `GraphicTool` es una clase dentro de un *framework* de creación de editores gráficos, que crea objetos gráficos, (instancias de subclases de `Graphic`) y los inserta en un documento, **¿cómo puede hacerlo si no sabe qué objetos gráficos concretos debe crear?**
  - Una instancia de `GraphicTool` es una herramienta de una paleta de herramientas y es inicializada con una instancia de `Graphic`: **instancia prototípica.**

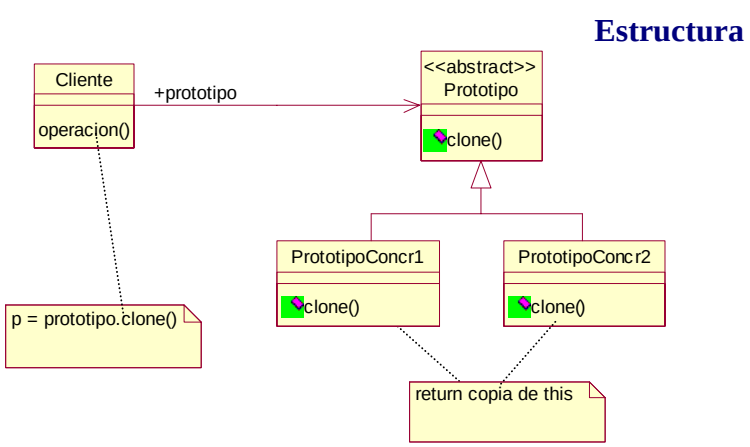
72

# Prototipo




73

# Prototipo




74

# Prototipo

- **Aplicabilidad**
  - Cuando un sistema deba ser independiente de cómo sus productos son creados, compuestos y representados y:
    - las clases a instanciar son especificadas en tiempo de ejecución, **o**
    - para evitar crear una jerarquía de factorías paralela a la de productos, **o**
    - cuando las instancias de una clase sólo pueden encontrarse en uno de un pequeño grupo de estados, **o**
    - inicializar un objeto es costoso



# Prototipo

- **Consecuencias**
  - Al igual que con *Builder* y *Factory Abstract* oculta al cliente las clases producto concretas.
  - Es posible añadir y eliminar productos en tiempo de ejecución: mayor flexibilidad que los anteriores.
  - Reduce la necesidad de crear subclases.

# Prototipo

- **Consecuencias**
  - La composición reduce el número de clases que es necesario crear: *"Definir nuevas clases sin programar"*
  - ***Definir nuevas clases mediante la creación de instancias de clases existentes.***
  - ***Definir nuevas clases cambiando la estructura.***
    - *objetos compuestos actúan como clases*

77

# Prototipo

- **Implementación**
  - Importante en lenguajes que no soportan el concepto de metaclase, así en Smalltalk o Java es menos importante.
  - Usar un manejador de prototipos que se encarga del mantenimiento de una tabla de prototipos.
  - Implementar la operación *clone* es lo más complicado.
  - A veces se requiere inicializar el objeto creado mediante copia: prototipo con operaciones "*set*"

78

## Ejemplo: Laberinto

```
public class FactoriaPrototipoLaberinto {
    factoriaPrototipoLaberinto (Laberinto lab, Puerta p, Habitacion h,
                Pared m);
      { protoLab = lab; protoPuerta = p; protoHab = h; protoPared = m; }

    public Laberinto makeLaberinto() {return (Laberinto)protoLab.clone(); };
    public Pared makePared() { return (Pared) protoPared.clone(); };
    public Habitacion  makeHabitacion (int n) {
          Habitacion h = (Habitacion) protoHab.clone();
          h.initialize(n); return h;}
    public Puerta makePuerta () (Habitacion h1, Habitacion h2)
          { Puerta p = (Puerta) protoPuerta.clone();
          p.initialize(h1,h2); return p; }
}
```

79

## Ejemplo: Laberinto

```
JuegoLaberinto juego;
FactoriaPrototipoLaberinto laberintoSimple =
    new FactoriaPrototipoLaberinto (new Laberinto, new Puerta,
                                    new Habitacion, new Pared);
Laberinto unlab = juego. makeLaberinto(laberintoSimple)

JuegoLaberinto juego;
FactoriaPrototipoLaberinto otroLaberinto =
    new FactoriaPrototipoLaberinto (new Laberinto,
                new PuertaQueOye, new HabitacionEncantada,
                new Pared);
Laberinto unlab = juego. makeLaberinto (otroLaberinto)
```

80

## Ejemplo: Laberinto

```
public class JuegoLaberinto {
    public Laberinto makeLaberinto (FactoriaPrototipoLaberinto factoria) {
            Laberinto unLab = factoria.makeLaberinto();
            Habitacion h1 = factoria.makeHabitacion(1);
            Habitacion h2 = factoria.makeHabitacion(2);
            Puerta unaPuerta = factoria.makePuerta(h1,h2);
            unLab.addHabitacion(h1);
            unLab.addHabitacion(h2);
            h1.setLado(Norte, factoria.makePared() );
            h1.setLado(Este, unaPuerta)
            ..
            h2.setLado(Oeste, unaPuerta);
            h2.setLado(Sur, factoria.makePared() )
            return unLab;
      }
}
```

81

## Factoría Abstracta: Alternativa 1

- Con ***métodos factoría***

```
public abstract class WidgetFactory {
    public abstract Window createWindow();
    public abstract Menu createScrollBar();
    public abstract Button createButton();
}

public class MotifWidgetFactory extends WidgetFactory {
    public Window createWindow() {return new MotifWidow();}
    public ScrollBar createScrollBar() {return new MotifScrollBar();}
    public Button createButton() {return new MotifButton();}
}
```

82

# Factoría Abstracta: Alternativa 1

```
// Código cliente

WidgetFactory wf = new MotifWidgetFactory();


Button b = wf.createButton();

Window w = wf.createWindow();
...
```



# Factoría Abstracta: Alternativa 2

- Con ***prototipos***

```
public class WidgetFactory {
     private Window protoWindow;
     private ScrollBar protoScrollBar;
     private Button protoButton;
     public WidgetFactory(Window wp, ScrollBar sbp, Button bf) {
              protoWindow = wp;  protoScrollBar = sbp; protoButton = bp; }

     public Window createWindow() {return (Window) protoWindow.clone(); }
     public ScrollBar createScrollBar() {return (ScrollBar)
                                                  protoScrollBar.clone();}
     public Button createButton() {return (Button) protoButton.clone();}
}
```

## Factoría Abstracta: Alternativa 2

```
// crea objetos de una jerarquía de productos
class FactoriaProductos {
   private Hashtable catalogo;
   public Producto getProducto(String s) {
         return (Producto) ((Producto) catalogo.get(s)).clone(); }
   public FactoriaProductos (String familia) {
                                        "inicializa el catalogo"}
   ...
}
```

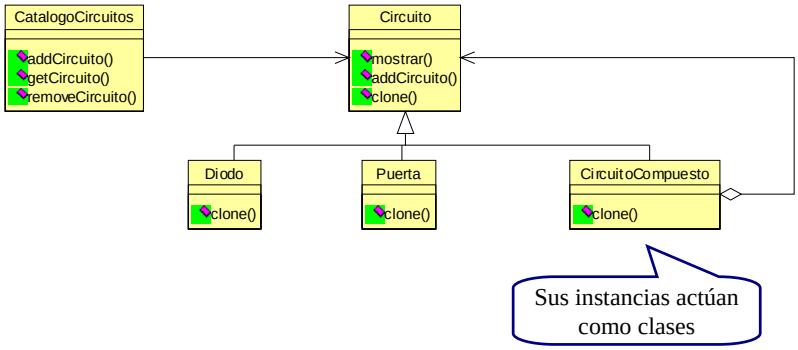


## Ejercicio

Sea una herramienta para diseño y simulación de circuitos digitales que proporciona una paleta con elementos tales como puertas, diodos, interruptores, flip-flop,.., y que permite construir circuitos a partir de ellos. Los circuitos construidos por el usuario pueden ser añadidos a la paleta para ser utilizados directamente en la construcción de nuevos circuitos. Diseña una solución basada en patrones para el mecanismo de crear instancias de un nuevo circuito añadido a la paleta, describiendo la estructura de clases propuesta.

# Solución

Composite + Prototype + Singleton




87

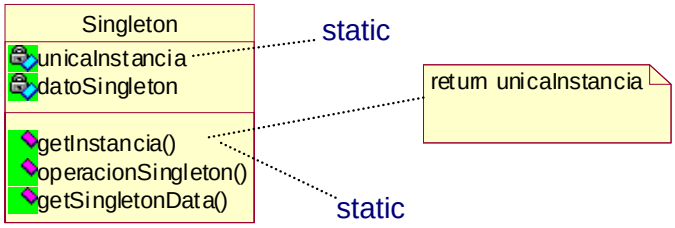
# Singleton

- **Propósito**
  - Asegurar que una clase tiene una única instancia y asegurar un punto de acceso global.
- **Motivación**
  - Un manejador de ventanas, o un *spooler* de impresoras o de un sistema de ficheros o una factoría abstracta.
  - La clase se encarga de asegurar que exista una única instancia y de su acceso.

88

# Singleton

## Estructura




Singleton.getInstancia().operacionSingleton()

89

# Singleton

- **Aplicabilidad**
  - Debe existir una única instancia de una clase, accesible globalmente.
- **Consecuencias**
  - Acceso controlado a la única instancia
  - Evita usar variables globales
  - Generalizar a un número variable de instancias
  - La clase *Singleton* puede tener subclases.

90

# Singleton (Sin subclases)

```
public class Singleton {
      public static Singleton getInstancia() {
            if (unicaInstancia == null)  unicaInstancia = new Singleton();
            return unicaInstancia;
      }
       private Singleton() { }

      private static Singleton unicaInstancia = null;
      private int dato = 0;
      public int getDato() {return dato;}
       public void setDato(int i) {dato = i;}
}
```



# Ejemplo: Catálogo instancias prototípicas

```
public class CatalogoCircuitos {
    public static CatalogoCircuitos getInstancia() {
            if (unicaInstancia == null)
                  unicaInstancia = new CatalogoCircuitos();
            return unicaInstancia;
    }
    private CatalogoCircuitos() { elems = new HashTable();  }

    private static CatalogoCircuitos unicaInstancia = null;
    private Hashtable elems;
    public Circuito getCircuito(String id) {
            Circuito c = (Circuito) elems.get(id);
            return (Circuito) c.clone(); }
  public void addCircuito(String id,Circuito c) {
                                    elems.put(id,c); }
}

Circuito sum = CatalogoCircuitos.getInstancia().getCircuito("sumador");
```

## Singleton (Con subclases)

```
public class FactoriaLaberinto {
   public static FactoriaLaberinto instancia(String s) {
          if (unicaInstancia == null)
               unicaInstancia =
                    Class.forName(s).newInstance();
          return unicaInstancia ;
}

      protected FactoriaLaberinto () { }
      ...
}
```

93

## Singleton (Con subclases)

```
public abstract class FactoriaLaberinto {
      public static FactoriaLaberinto instancia() { return unicaInstancia;}
      protected static FactoriaLaberinto unicaInstancia = null;
      protected FactoriaLaberinto () { }
      ...
}
public class FactoriaLaberintoEncantado extends FactoriaLaberinto {
     public static FactoriaLaberinto instancia() {
          if (unicaInstancia == null)
                    unicaInstancia = new FactoriaLaberintoEncantado();
          return unicaInstancia;
     }
     private FactoriaLaberintoEncantado() { }
}
```

94

## Contenidos

- Introducción a los patrones de diseño GoF
- Patrones de **creación**
  - Factoría Abstracta, Builder, Método Factoría, Prototipo, Singleton
- Patrones **estructurales**
  - Adapter, Bridge, Composite, Decorador, Fachada, Flyweight, Proxy
- Patrones de **comportamiento**
  - Cadena de Responsabilidad, Command, Iterator, Intérprete, Memento, Mediador, Observer, Estado, Estrategia, Método Plantilla, Visitor



## Patrones estructurales

- Cómo clases y objetos se combinan para formar estructuras más complejas.
- Patrones basados en herencia
  - Sólo una forma de *Adapter*
- Patrones basados en composición
  - Describen formas de combinar objetos para obtener nueva funcionalidad
  - Posibilidad de cambiar la composición en tiempo de ejecución.

# Adapter / Wrapper (Adaptador)

- **Propósito**
  - **Convertir la interfaz de una clase en otra que el cliente espera**. Permite la colaboración de ciertas clases a pesar de tener interfaces incompatibles
- **Motivación**
  - Un editor gráfico incluye una jerarquía que modela elementos gráficos (líneas, polígonos, texto,..) y se desea reutilizar una clase existente *TextView* para implementar la clase que representa elementos de texto, pero que tiene una interfaz incompatible.

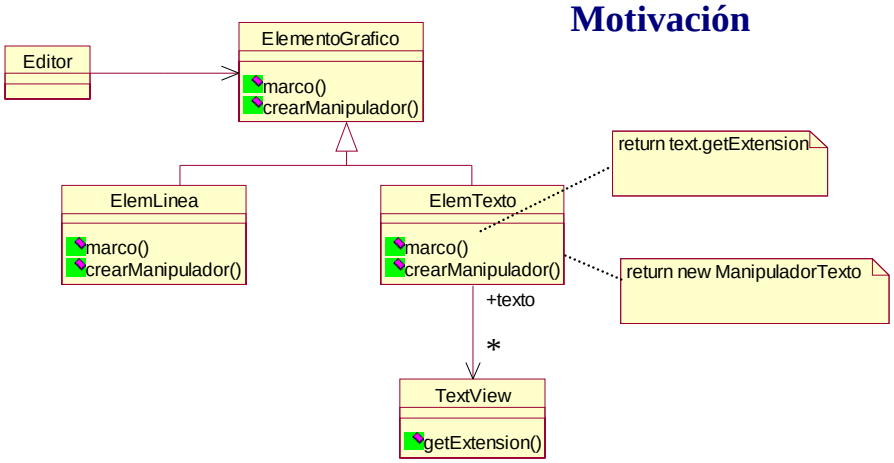


# Adaptador

- **Motivación**
  - A menudo un *toolkit* o librería de clases no se puede utilizar debido a que su interfaz es incompatible con la interfaz requerida por la aplicación.
  - No debemos o podemos cambiar la interfaz de la librería de clases

# Adaptador

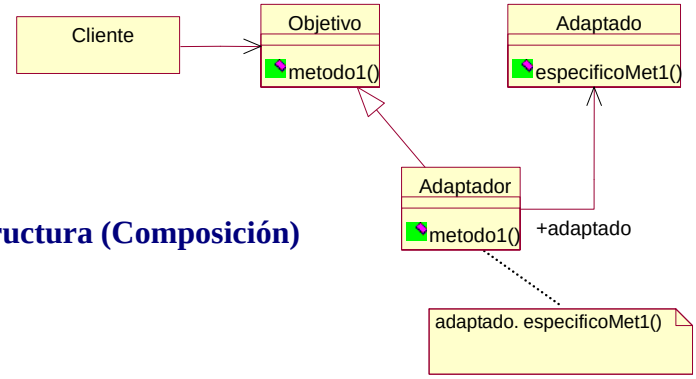
**Motivación**



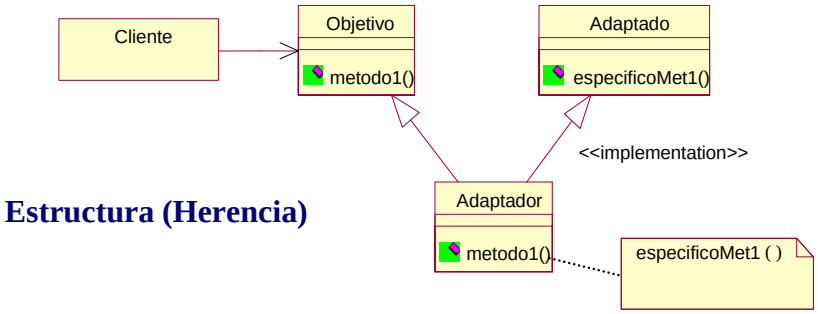



# Adaptador

**Estructura (Composición)**



# Adaptador




**Estructura (Herencia)**



# Adaptador

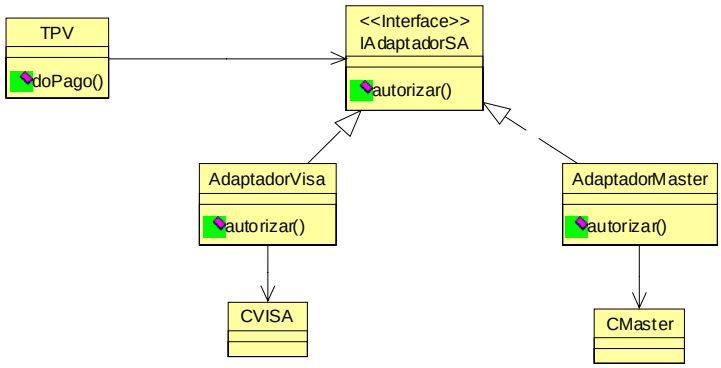
- **Aplicabilidad**
  - Se desea usar una clase existente y su interfaz no coincide con la que se necesita.
  - Se desea crear una clase reutilizable que debe colaborar con clases no relacionadas o imprevistas.

# Problema

- ¿Cómo resolver interfaces incompatibles o proporcionar una interfaz estable para componentes que ofrecen una misma funcionalidad pero su interfaz es diferente?
- En una aplicación TPV soportar diferentes tipos de servicios externos de autorización de pagos.

103

# Solución




104

## Adaptador

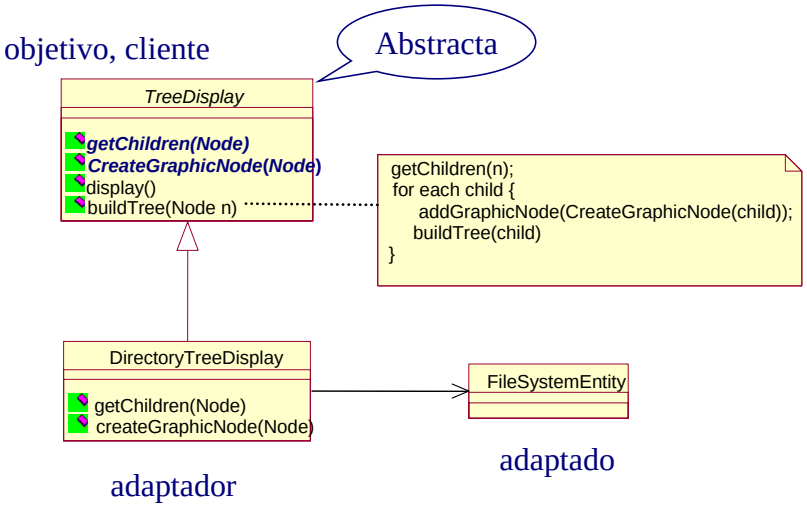
- **Consecuencias**
  - Un adaptador basado en herencia no funciona cuando queramos adaptar una clase y sus subclases.
  - Un adaptador basado en herencia puede redefinir comportamiento heredado de *Adaptado*.
  - Un adaptador basado en herencia no implica ninguna indirección, sólo introduce un objeto.
  - Un adaptador basado en composición permite adaptar una clase y sus subclases, pudiendo añadir funcionalidad a todos los adaptados a la vez.
  - Para un adaptador basado en composición es complicado redefinir comportamiento de *Adaptado*.



## Adaptador

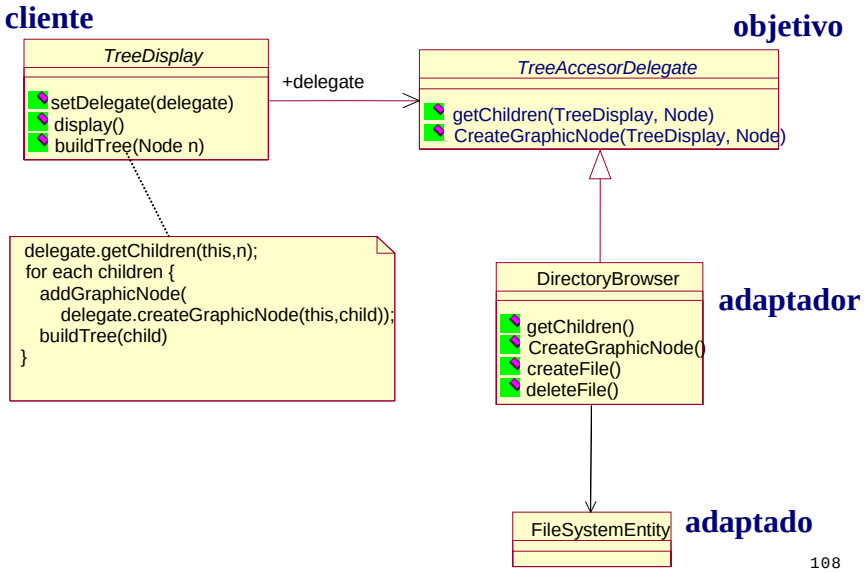
- **Consecuencias**
  - La funcionalidad de Adaptador depende de la similitud entre la interfaz de las clases *Objetivo* y *Adaptado*.
  - **Adaptadores *"pluggable":***
    - ¿Cómo creamos una clase reutilizable preparada para colaborar con clases cuya interfaz se desconoce?
    - **Ejemplo**: Clase `TreeDisplay` para visualizar estructuras jerárquicas de cualquier naturaleza.

# Adaptador




107

# Adaptador




108

# Ejemplo en Java 1.3 Swing

- La librería Swing de Java incluye clases gráficas que muestran información organizada en cierto formato, como son `JTree` (información jerárquica), `JTable` (tabla bidimensional) o JList (lista de datos). Estas clases son clientes respectivamente de TreeModel, `TableModel` y `ListModel` que le aíslan de los detalles de un modelo de datos concreto y que incluyen métodos que le permiten obtener los datos a representar según el formato que corresponda.

109

# Ejemplo en Java 1.3 Swing

```
public interface ListModel {
        int getSize();
        Object getElementAt(int index);
}
public interface TableModel {
        public int getRowCount();
            // obtener número de filas
        public int getColumnCount();
            // obtener número de columnas
        public Object getValueAt(int fil, int col);
            // retorna valor del elemento fila fil y columna col
        public String getColumnName(int col);
            // retorna nombre de la columna col
}
```

110

# Ejemplo en Java 1.3 Swing

```
public interface TreeModel {
        public Object getRoot();
            // retorna nodo raíz
        public Object getChild(Object parent, int index);
            // retorna nodo hijo del padre en posición index
        public int getChildCount(Object parent);
            // retorna número de hijos del nodo padre
        public boolean isLeaf(Object node);
            // retorna true si es un nodo hoja
}
```



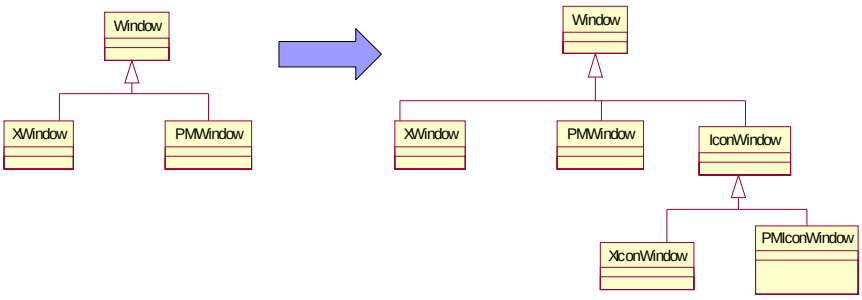
# Bridge / Handle (Puente)

- **Propósito**
  - **Desacoplar una abstracción de su implementación**, de modo que los dos puedan cambiar independientemente.
- **Motivación**
  - Clase que modela la abstracción con subclases que la implementan de distintos modos.
  - Herencia hace difícil reutilizar abstracciones e implementaciones de forma independiente
    - Si refinamos la abstracción en una nueva subclase, está tendrá tantas subclases como tenía su superclase.
    - El código cliente es dependiente de la implementación.

# Bridge

- **Motivación**

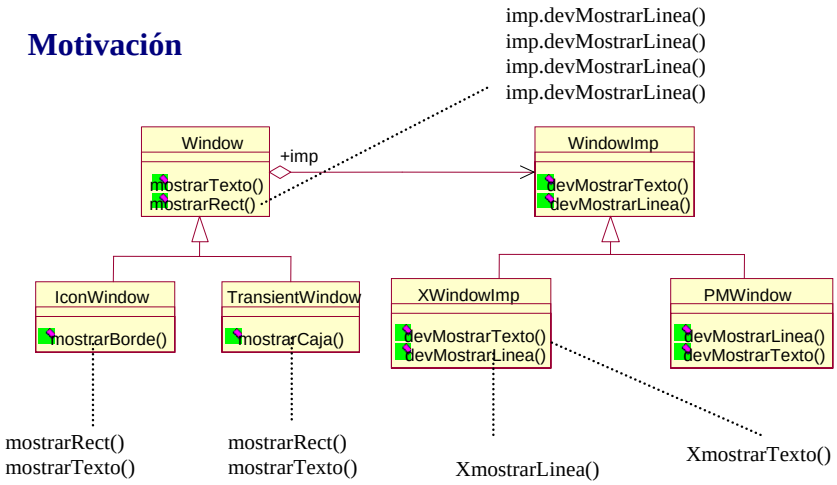
  Implementación de una abstracción *"window"* portable en una librería GUI.





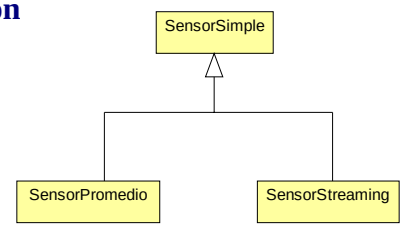

# Bridge

**Motivación**



# Bridge

**Motivación**

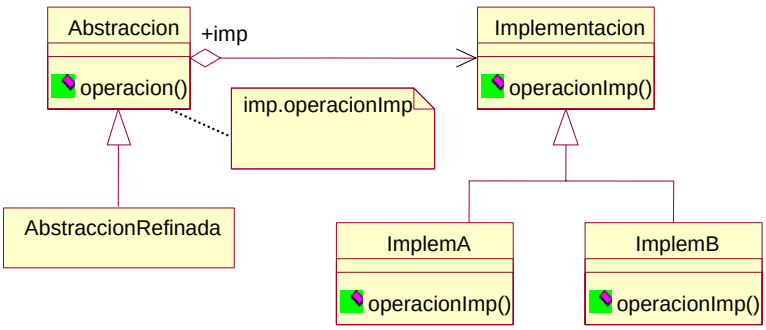



¿Qué hacemos si hay varios fabricantes de cada tipo de sensor?

¡Explosión de clases!



# Bridge / Handle




**Estructura**

## Bridge

- **Aplicabilidad**
  - Se quiere evitar una ligadura permanente entre una abstracción y su implementación, p.e. porque se quiere elegir en tiempo de ejecución.
  - Abstracciones e implementaciones son extensibles.
  - Cambios en la implementación de una abstracción no deben afectar a los clientes (no recompilación).
  - Ocultar a los clientes la implementación de la interfaz.
  - Se tiene una proliferación de clases, como sucedía en la *Motivación*.



## Bridge

- **Consecuencias**
  - Un objeto puede cambiar su implementación en tiempo de ejecución.
  - Cambios en la implementación no requerirán compilar de nuevo la clase `Abstracción` y sus clientes.
  - Se mejora la extensibilidad.
  - Se ocultan detalles de implementación a los clientes.

# Bridge

- **Implementación**
  - Aunque exista una única implementación puede usarse el patrón para evitar que un cliente se vea afectado si cambia.
  - ¿Cómo, cuándo y dónde se decide que implementación usar?
    - Constructor de la clase `Abstraccion`
    - Elegir una implementación por defecto
    - Delegar a otro objeto, por ejemplo un objeto factoría

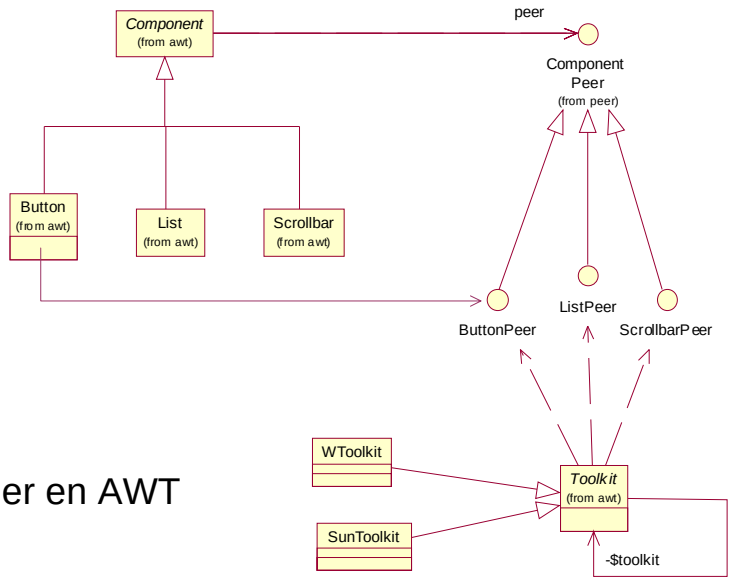


# Ejemplo en Java 1.1 AWT

La clase `Component` del paquete `java.awt` es una clase abstracta que encapsula la lógica común a todos los componentes GUI y tiene subclases tales como `Button`, `List` y `TextField` que encapsulan la lógica para estos componentes GUI de forma independiente de una plataforma concreta (Motif, W32, GTK,...). Por cada una de estas clases existe una interfaz (por ejemplo `ComponentPeer`, `ButtonPeer`, `ListPeer`,..) que declara métodos cuya implementación proporcionará la funcionalidad específica de una plataforma concreta. Existe una clase abstracta denominada `Toolkit` que contiene métodos abstractos para crear componentes para una plataforma concreta, tales como el método que crea un `ButtonPeer` o el que crea un `ListPeer`. `Toolkit` tiene una subclase para cada tipo de plataforma que implementan dichos métodos, también tiene una variable estática que almacena la única instancia que existirá de un *toolkit* concreto y un método estático `getToolkit` que retorna esa instancia. Cada clase componente (por ejemplo `Button`) tiene un atributo peer que se inicializa del siguiente modo (en este caso con un objeto `ButtonPeer`),

```
peer = getToolkit().createButton(this);
```

## Builder en AWT






# Ejemplo Java 1.1 AWT

- Java 1.1 AWT (Abstract Window Toolkit) fue diseñado para proporcionar una interfaz GUI en un entorno heterogéneo.
- AWT utiliza una factoría abstracta para crear todos los componentes *peer* requeridos para la plataforma específica que sea usada.
- Ejemplo:

```
public class List extends Component implements ItemSelectable
{
    ...
    peer = getToolkit().createList(this);
    ...
}
```

- El método `getToolkit()` es heredado de `Component` y devuelve una referencia al objeto factoría usado para crear todos los *widgets*

# Ejemplo Java 1.1 AWT

```
// método getToolkit en Component

public Toolkit getToolkit() {
        return getToolkitImpl();
}
final Toolkit getToolkitImpl() {
        ComponentPeer peer = this.peer;
        if ((peer != null) && ! (peer instanceof LightweightPeer)){
                return peer.getToolkit();
        }
        Container parent = this.parent;
        if (parent != null) {
                return parent.getToolkitImpl();
        }
        return Toolkit.getDefaultToolkit();
}
```

123

# Ejemplo en Java 1.1 AWT

```
// método getDefaultToolkit en la clase Toolkit

public static Toolkit getDefaultToolkit() {
    String nm;
    Class cls;
    if (toolkit == null) {
        try { nm = System.getProperty("awt.toolkit",
                                        "sun.awt.motif.MToolkit");
            try { cls = Class.forName(nm);
            } catch (ClassNotFoundException e){..}
        if (cls != null) toolkit = (Toolkit) cls.newInstance();
    ...
    return toolkit;
}
```

124

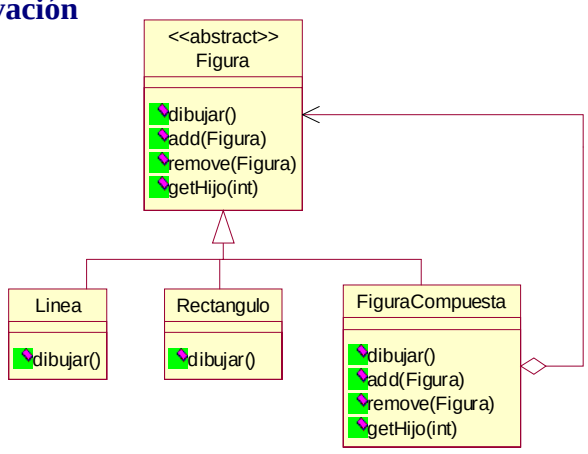
# Composite (Compuesto)

- **Propósito**
  - **Componer objetos en estructuras jerárquicas para representar jerarquías parte/todo**. Permite a los clientes manejar a los objetos primitivos y compuestos de forma uniforme.
- **Motivación**
  - Modelar figuras compuestas
  - Modelar documentos
  - Modelar paquetes de valores (acciones, bonos,..)



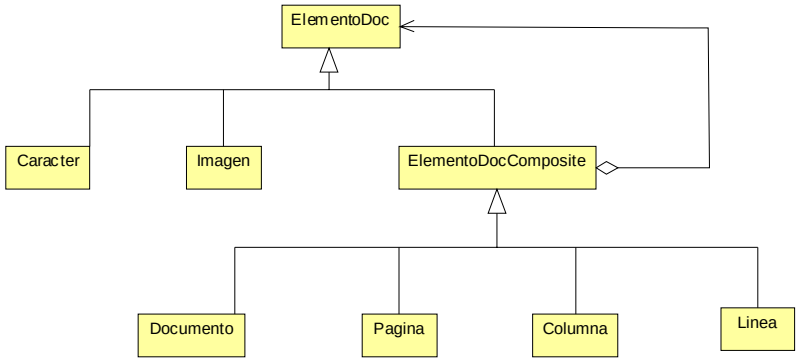
# Composite

**Motivación**



# Ejemplo “Documento”

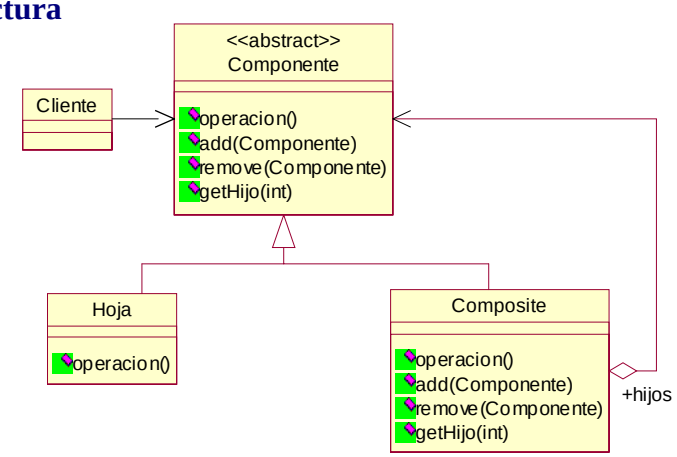
## Motivación




127

# Composite

## Estructura




128

## Composite

- **Aplicabilidad**
  - Se quiere representar jerarquías parte/todo
  - Se quiere que los clientes ignoren la diferencia entre objetos compuestos y los objetos individuales que los forman.



## Composite

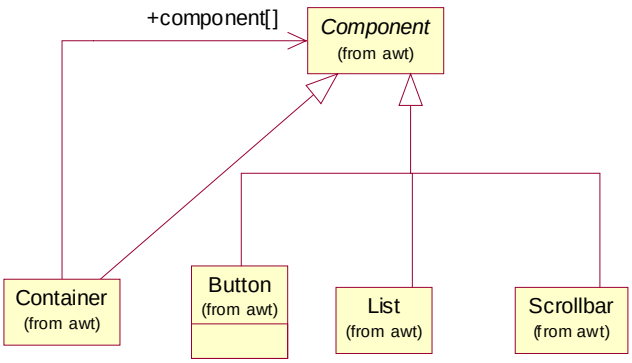
- **Consecuencias**
  - Jerarquía con clases que modelan objetos primitivos y objetos compuestos, que permite composición recursiva.
  - Clientes pueden tratar objetos primitivos y compuestos de modo uniforme.
  - Es fácil añadir nuevos tipos de componentes.
  - No se puede confiar al sistema de tipos que asegure que un objeto compuesto sólo contendrá objetos de ciertas clases, necesidad de comprobaciones en tiempo de ejecución.

# Composite

- **Implementación**
  - Referencias de componentes hijos a su padre puede ayudar al recorrido y manejo de la estructura compuesta.
  - ¿Debe contener *Componente* operaciones que no tienen significado para sus subclases?
  - **¿Dónde colocamos las operaciones de añadir y eliminar hijos, y obtener hijos?**
    - Compromiso entre seguridad y transparencia.
    - **Transparencia**: en clase Componente
    - **Seguridad**: en clase Composite
  - Puede ser necesario especificar un orden en los hijos.
  - ¿Qué estructura de datos es la adecuada?



# Ejemplo en Java AWT

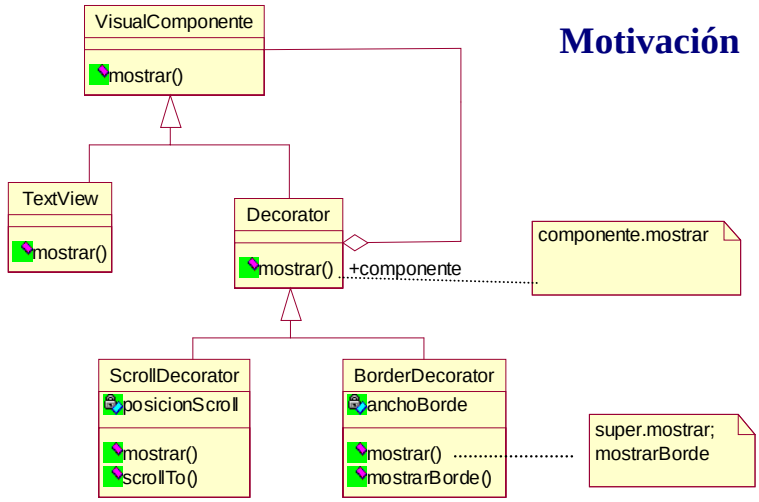


# Decorator (Decorador)

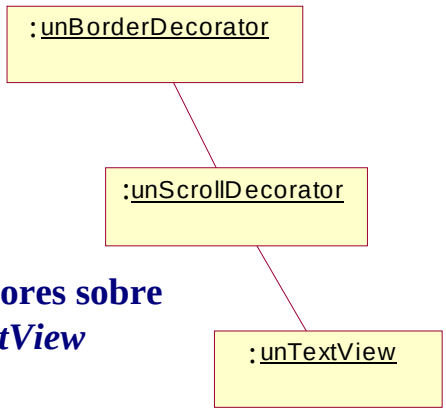
- **Propósito**
  - **Asignar dinámicamente nuevas responsabilidades a un objeto**. Alternativa más flexible a crear subclases para extender la funcionalidad de una clase.
- **Motivación**
  - Algunas veces se desea añadir atributos o comportamiento adicional a un objeto concreto no a una clase.
  - Ejemplo: bordes o *scrolling* a una ventana.
  - Herencia no lo permite.



# Decorador



# Decorador

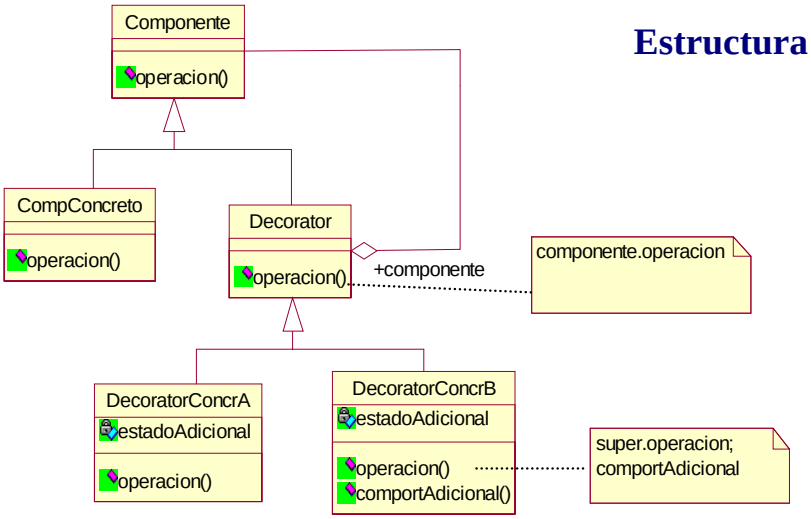



**Dos decoradores sobre un *TextView***

135

# Decorador

**Estructura**




136

## Decorador

- **Aplicabilidad**
  - Añadir dinámicamente responsabilidades a objetos individuales de forma transparente, sin afectar a otros objetos.
  - Responsabilidades de un objeto pueden ser eliminadas.
  - Para evitar una explosión de clases que produce una jerarquía inmanejable.



## Decorador

- **Consecuencias**
  - Más flexible que la herencia: responsabilidades pueden añadirse y eliminarse en tiempo de ejecución.
  - Diferentes decoradores pueden ser conectados a un mismo objeto.
  - Reduce el número de propiedades en las clases de la parte alta de la jerarquía.
  - Es simple añadir nuevos decoradores de forma independiente a las clases que extienden.
  - Un objeto decorador tiene diferente OID al del objeto que decora.
  - Sistemas con muchos y pequeños objetos.

## Decorador

- **Implementación**
  - La interfaz de un objeto *decorador* debe conformar con la interfaz del objeto que decora. Clases *decorador* deben heredar de una clase común.
  - *Componentes* y *decoradores* deben heredar de una clase común que debe ser “ligera” en funcionalidad.
  - Si la clase Componente no es ligera, es mejor usar el patrón *Estrategia* que permite alterar o extender el comportamiento de un objeto.
  - Un componente no sabe nada acerca de sus decoradores, con *Estrategia* sucede lo contrario.



## Decorador

- **Implementación**
  - El decorador envía los mensajes al componente que decora, pudiendo extender la operación con nuevo comportamiento.
  - Las clases que modelan los decoradores concretos pueden añadir responsabilidades a las que heredan de *Decorator*.
- **Ejercicio.** Muestra la diferencia entre aplicar el patrón *Decorador* y el patrón *Estrategia* para añadir funcionalidad a instancias de una clase TextView, tal como un borde o una barra de desplazamiento.

# Ejemplo Java

Java posee una librería de clases e interfaces para manejar streams de caracteres y bytes. Por ejemplo, la clase abstracta `Reader` que proporciona un stream de caracteres tiene subclases tales como `InputStreamReader` o `StringReader` que implementan diferentes formas de proporcionar un stream de entrada de caracteres. También se dispone de una clase `Writer` para streams de salida de caracteres. Para cada uno de los tipos de stream la librería incluye unas clases que representan "filtros de stream" (filter) que permiten encadenar filtros sobre un stream y aplicarle varias transformaciones, como son las clases abstractas `FilterReader` y `FilterWriter`. A continuación se muestra parte del código de la clase `FilterReader` que implementa métodos de la clase `Reader` como `read()` y que es raíz de clases que implementan filtros de lectura.

```
public abstract class FilterReader extends Reader {
        protected Reader in;
        protected FilterReader(Reader in) {
                super(in);
                this.in = in; }
        public int read() throws IOException {
                return in.read(); }
… }
```



# Ejemplo Java

Podemos definir diferentes subclases de `FilterReader` con filtros tales como convertir a mayúsculas o eliminar espacios redundantes entre palabras. La siguiente clase sería un ejemplo:

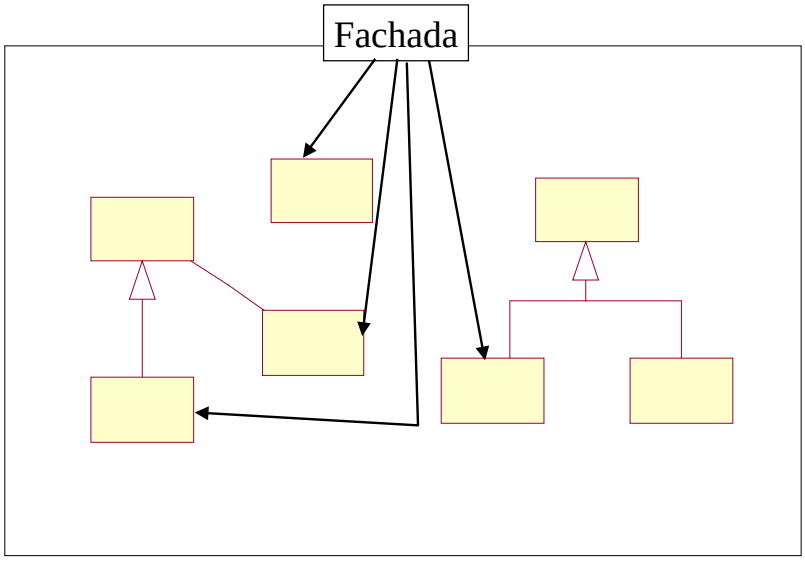
```
public class ConversorMayúsculas extends FilterReader {
    public ConversorMayúsculas (Reader in) {
        super(in);
    }
    public int read() throws IOException {
        int c = super.read();
        if (c != -1) return Character.toUpperCase((char)c);
        else return c;
    }
}
```

# Facade (Fachada)

- **Propósito**
  - **Proporciona una única interfaz a un conjunto de clases de un subsistema**. Define una interfaz de más alto nivel que facilita el uso de un subsistema.
- **Motivación**
  - Reducir las dependencias entre subsistemas.
  - Un entorno de programación que ofrece una librería de clases que proporcionan acceso a su subsistema compilador: `Scanner`, `Parser`, `ProgramNode`, `ByteCodeStream` y `ProgramNodeBuilder`. Clase `Compiler` actúa como fachada.

143

## Estructura



## Fachada

- **Aplicabilidad**
  - Proporcionar una interfaz simple a un subsistema.
  - Hay muchas dependencias entre clientes y las clases que implementan una abstracción.
  - Se desea una arquitectura de varios niveles: una fachada define el punto de entrada para cada nivel-subsistema.

145

## Fachada

- **Consecuencias**
  - Una fachada ofrece los siguientes beneficios:

1. Facilita a los clientes el uso de un subsistema, al ocultar sus componentes.
2. Proporciona un acoplamiento débil entre un subsistema y los clientes: cambios en los componentes no afectan a los clientes.
3. No se impide a los clientes el uso de las clases del subsistema si lo necesitan.

146

## Fachada

- **Implementación**
  - Es posible reducir el acoplamiento entre clientes y subsistema, definiendo la fachada como una clase abstracta con una subclase por cada implementación del subsistema.
  - La fachada no es la única parte pública de un subsistema, sino que es posible declarar clases individuales del subsistema como públicas (paquetes en Java, *name space* en C++).



## Flyweight (Peso Ligero)

- **Propósito**
  - Uso de **objetos compartidos para soportar eficientemente un gran número de objetos de poco tamaño**.
- **Motivación**
  - En una aplicación editor de documentos, ¿modelamos los caracteres mediante una clase?
  - Un *flyweight* es un objeto compartido que puede ser utilizado en diferentes contextos simultáneamente.
    - No hace asunciones sobre el contexto
    - Estado intrínseco vs. Estado extrínseco

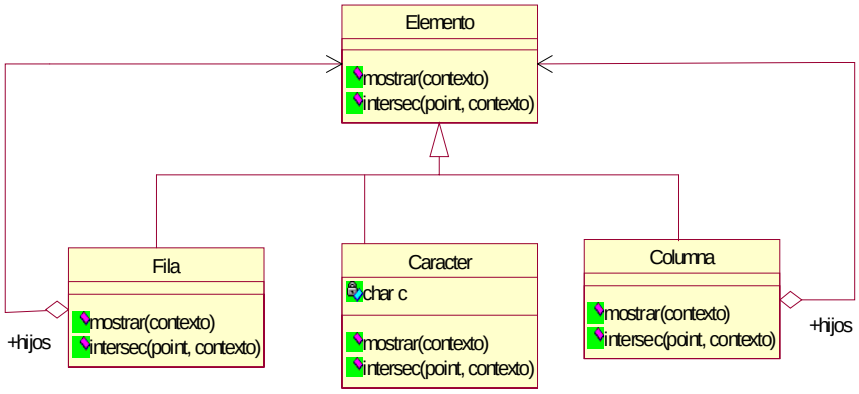
# Flyweight

- Estado intrínseco se almacena en el *flyweight* y consiste de información que es independiente del contexto y se puede compartir.
- Estado extrínseco depende del contexto y por tanto no puede ser compartido.
- Objetos clientes son responsables de pasar el estado extrínseco al *flyweight* cuando lo necesita.
- Objetos *flyweight* se usan para modelar conceptos o entidades de los que se necesita una gran cantidad en una aplicación, p.e. caracteres de un texto.

149

# Flyweight

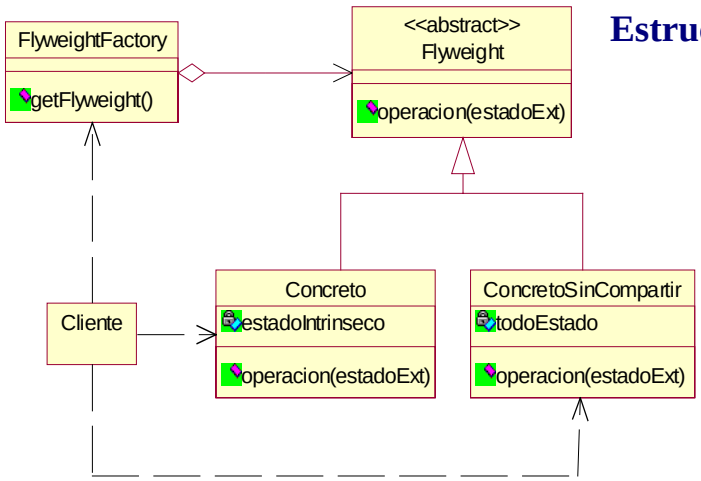
- *Flyweight* carácter de un texto:
  - estado intrínseco: código del carácter
  - estado extrínseco: información sobre posición y estilo




150

# Flyweight

**Estructura**






# Flyweight

```
Flyweight  getFlyweight (key) {
    if “existe (flyweight[key])” { return “flyweight existente”}
   else { “crear nuevo flyweight”;
            “añadirlo al conjunto de flyweights”;
            return  “nuevo flyweight”}
}
```

## Flyweight

- **Aplicabilidad**
  - Aplicarlo siempre que se cumplan las siguientes condiciones:

1. Una aplicación utiliza un gran número de cierto tipo de objetos.
2. El coste de almacenamiento es alto debido al excesivo número de objetos.
3. La mayor parte del estado de esos objetos puede hacerse extrínseco.
4. Al separar el estado extrínseco, muchos grupos de objetos pueden reemplazarse por unos pocos objetos compartidos.
5. La aplicación no depende de la identidad de los objetos**.**



## Flyweight

- **Consecuencias**
  - Puede introducir costes *run-time* debido a la necesidad de calcular y transferir el estado extrínseco.
  - La ganancia en espacio depende de varios factores:
    - la reducción en el número de instancias
    - el tamaño del estado intrínseco por objeto
    - si el estado extrínseco es calculado o almacenado
  - Interesa un estado extrínseco calculado
  - Se suele combinar con el *Composite*.

# Flyweight

- **Implementación**
  - La aplicabilidad depende de la facilidad de obtener el estado extrínseco. Idealmente debe poder calcularse a partir de una estructura de objetos que necesite poco espacio de memoria.
  - Debido a que los *flyweight* son compartidos, no deberían ser instanciados directamente por los clientes: clase *FlyweightFactory*.



# Código del Ejemplo

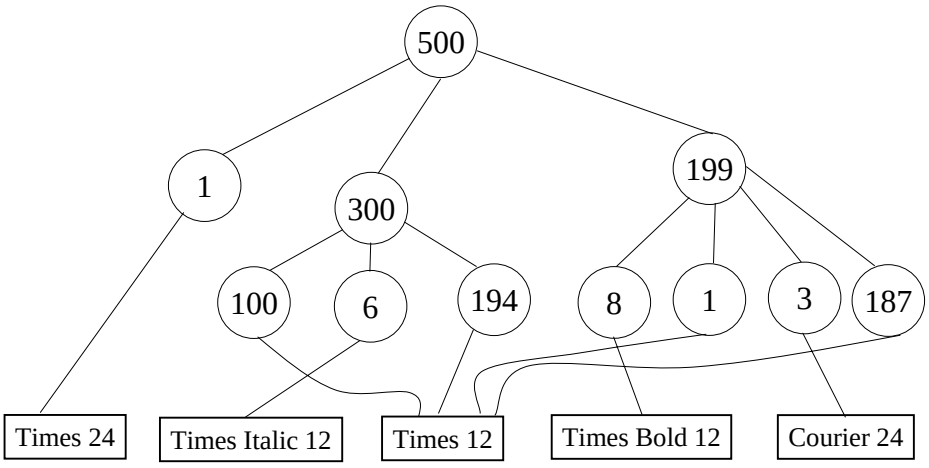
```
public class Elemento {
    public Elemento() {..}
    public void mostrar (Window w, ElemContexto ec) {..}
    public void setFont (Font f, ElemContexto ec) {..}
    public Font getFont (ElemContexto ec) {..}
    public void insert (ElemContexto ec, Elemento e) {..}
    ...
}
public class Caracter extends Elemento {
   char code;
   public Caracter(char c) {..}
   public void mostrar (Window w, ElemContexto ec) {..}
}
```

# Código del Ejemplo

```
public class ElemContexto {
        int index;
        Btree fonts;

        public ElemContexto() {...}
        public void setFont (Font f, int span) {..}
        public Font getFont () {..}
        public void insert (int cant) {..}
        public void next (){..}
}
```

**Repositorio de estado extrínseco**

157




158

## Ejemplos en Java

- *Strings*
  - Pool de objetos compartidos para literales de tipo String
    - "hola" = "hola"
  - Método intern() en clase String: dado un string lo añade al pool
    String c = (a+b).intern
- *Bordes* de Swing

```
class BorderFactory {
 public static Border createLineBorder();
 public static Border createCompoundBorder()
 ...
}

JPanel pane = new Jpanel
pane.setBorder(Borderfactory.createBorderLine(Color.black))
```

159

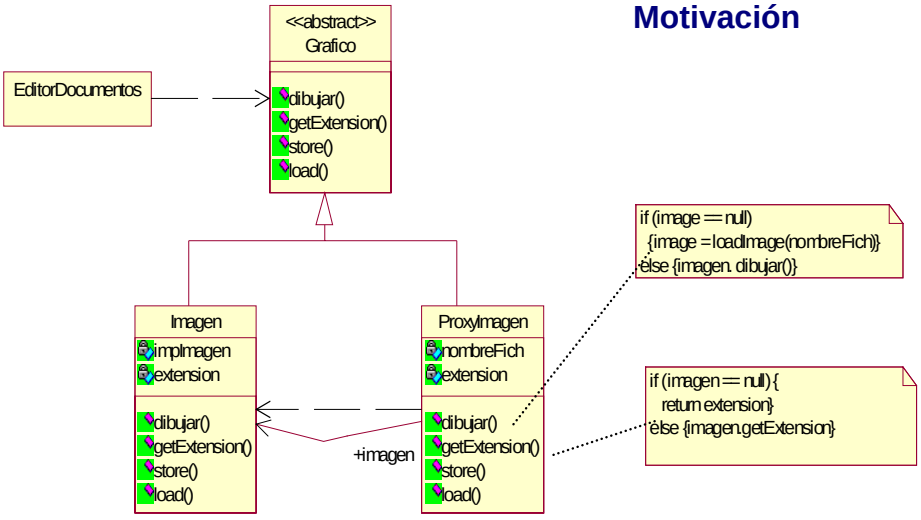
## Proxy (Sustituto)

- **Propósito**
  - **Proporcionar un sustituto** (*surrogate*) **de un objeto para controlar el acceso a dicho objeto.**
- **Motivación**
  - Diferir el coste de crear un objeto hasta que sea necesario usarlo: creación bajo demanda.
  - Un editor de documentos que incluyen objetos gráficos.
  - ¿Cómo ocultamos que una imagen se creará cuando se necesite?: manejar el documento requiere conocer información sobre la imagen.

160

# Proxy

**Motivación**

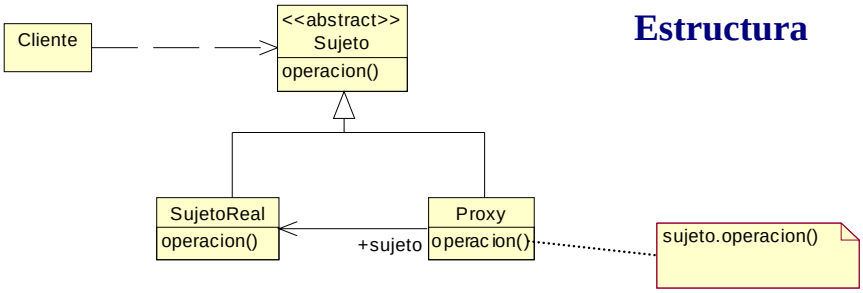





# Proxy

- **Motivación**
  - Hay situaciones en las que un cliente no referencia o no puede referenciar a un objeto directamente, pero necesita interactuar con él.
  - Un objeto *proxy* puede actuar como intermediario entre el objeto cliente y el objeto destino.
  - El objeto *proxy* **tiene la misma interfaz** como el objeto destino.
  - El objeto *proxy* **mantiene una referencia** al objeto destino y puede pasarle a él los mensajes recibidos (delegación).

# Proxy




**Estructura**



# Proxy

- **Motivación**
  - Retrasar la operación de una clonación de una tabla hasta conocer que es realmente necesaria. Se desea clonar la tabla para evitar mantener un bloqueo un largo período de tiempo: operación costosa. Se puede crear una clase que encapsule la tabla y sólo clone cuando sea necesario.
  - Mantenimiento de los servicios ante los fallos.
  - Materialización perezosa de tuplas en objetos.

(Ver fotocopias entregadas)

## Proxy

- **Aplicabilidad**
  - Siempre que hay necesidad de referenciar a un objeto mediante una referencia más rica que un puntero o una referencia normal.
  - Situaciones comunes;

1. Proxy **acceso remoto** (acceso a un objeto en otro espacio de direcciones)
2. Proxy **virtual** (crea objetos grandes bajo demanda)
3. Proxy para **protección** (controlar acceso a un objeto)
4. **Referencia inteligente** (*smart reference,* proporciona operaciones adicionales)

165

## Proxy

- **Consecuencias**
  - Introduce un nivel de indirección para:

1. Un *proxy* remoto oculta el hecho que objetos residen en diferentes espacios de direcciones.
2. Un *proxy* virtual tales como crear o copiar un objeto bajo demanda.
3. Un *proxy* para protección o las referencias inteligentes permiten realizar tareas de control sobre los objetos accedidos.

166

## Proxy

Ejercicio 1. Se tiene una gran colección de objetos, tal como una tabla hash, y debe permitirse el acceso concurrente de diferentes clientes. Uno de los clientes desea realizar varias operaciones de búsqueda consecutivas, no permitiéndose el acceso a otros clientes para añadir o eliminar elementos de la colección.

Ejercicio 2. Un servidor tiene varias utilidades que se ejecutan como *daemons* sobre algunos puertos. ¿Cómo puede conseguirse acceder a estos servicios como si fuesen objetos locales desde varias máquinas cliente?



## Contenidos

- Introducción a los patrones de diseño GoF
- Patrones de **creación**
  - Factoría Abstracta, Builder, Método Factoría, Prototipo, Singleton
- Patrones **estructurales**
  - Adapter, Bridge, Composite, Decorador, Fachada, Flyweight, Proxy
- Patrones de **comportamiento**
  - Cadena de Responsabilidad, Command, Iterator, Intérprete, Memento, Mediador, Observer, Estado, Estrategia, Método Plantilla, Visitor

## Patrones de comportamiento

- Relacionados con la asignación de responsabilidades entre clases.
- Enfatizan la colaboración entre objetos.
- Caracterizan un flujo de control más o menos complejo que será transparente al que utilice el patrón.
- Basados en herencia: *Template Method e Interpreter*
- Basados en composición: *Mediator, Observer,..*

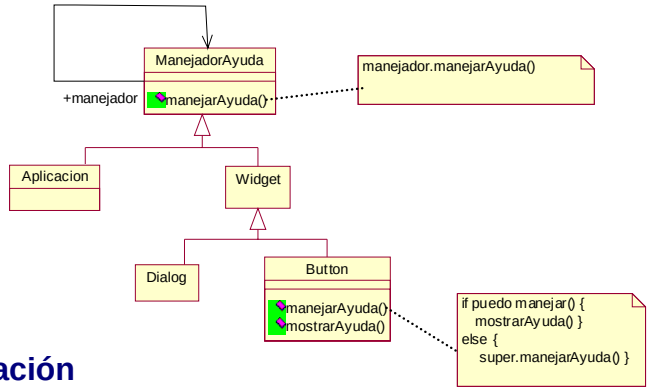
169

## Chain of Responsibility (Cadena de Responsabilidad)

- **Propósito**
  - **Evita acoplar el emisor de un mensaje a su receptor dándole a más de un objeto la posibilidad de manejar la solicitud**. Se define una cadena de objetos, de modo que un objeto pasa la solicitud al siguiente en la cadena hasta que uno la maneja.
- **Motivación**
  - Facilidad de ayuda sensible al contexto.
  - El objeto que proporciona la ayuda no es conocido al objeto (p.e. un *Button*) que inicia la solicitud de ayuda.

170

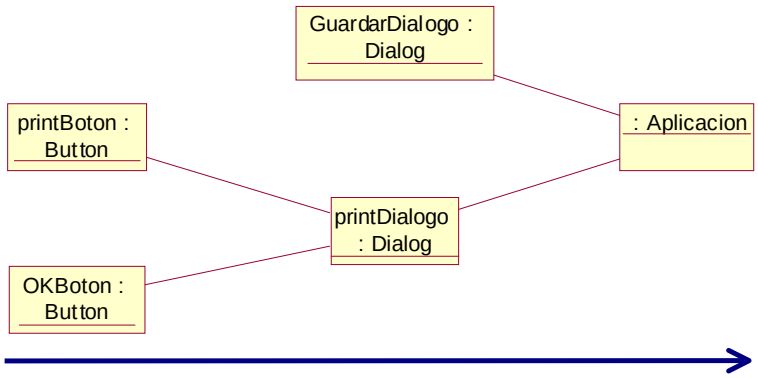
# Cadena de Responsabilidad




**Motivación**

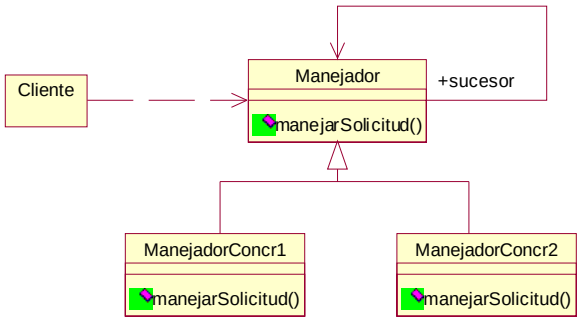


# Cadena de Responsabilidad



# Cadena de Responsabilidad

## Estructura




173

# Cadena de Responsabilidad

- **Aplicabilidad**
  - Más de un objeto puede manejar una solicitud, y el manejador no se conoce a priori.
  - Se desea enviar una solicitud a uno entre varios objetos sin especificar explícitamente el receptor.
  - El conjunto de objetos que puede manejar una solicitud puede ser especificado dinámicamente.

174

# Cadena de Responsabilidad

- **Consecuencias**
  - Reduce acoplamiento
  - Proporciona flexibilidad al asignar responsabilidades
  - No se garantiza el manejo de la solicitud



# Cadena de Responsabilidad

- **Implementación**
  - Dos posibles formas de implementar la cadena
    - Definir nuevos enlaces (en Manejador)
    - Usar enlaces existentes (p.e. un objeto *composite*)

```
class ManejadorAyuda {
        public ManejadorAyuda (ManejadorAyuda s) {sucesor = s;};
        public void manejarAyuda () { if (sucesor != null)
                        sucesor.manejarAyuda();};
        private ManejadorAyuda sucesor;
}
```

# Cadena de Responsabilidad

- **Implementación**
  - ¿Qué sucede si tenemos diferentes tipos de solicitudes?
    - En Manejador un método para cada solicitud
    - En Manejador un único método con un parámetro que representa el tipo de solicitud, por ejemplo un String.
    - En Manejador un único método que tiene un parámetro de una clase Solicitud que representa la solicitud.



# Cadena de Responsabilidad

- Ejemplo 1: Java 1.0 AWT
  - Mecanismo de delegación de eventos: Un evento es pasado al componente donde ocurre que puede manejarlo o lo pasa a su contenedor.

```
public boolean action(Event event, Object obj) {
    if (event.target == test_button)  doTestButtonAction();
   else if (event.target == exit_button) doExitButtonAction();
        else return super.action(event,obj);
   return true;}
```

## Cadena de Responsabilidad

- Ejemplo 2: Sistema de control de seguridad
  - Existen varios sensores que transmiten el estado a un ordenador central. La acción de un sensor depende del contexto que lo incluye (jerarquía de áreas o zonas de seguridad). Un sensor pasa el evento al agregado que lo incluye.

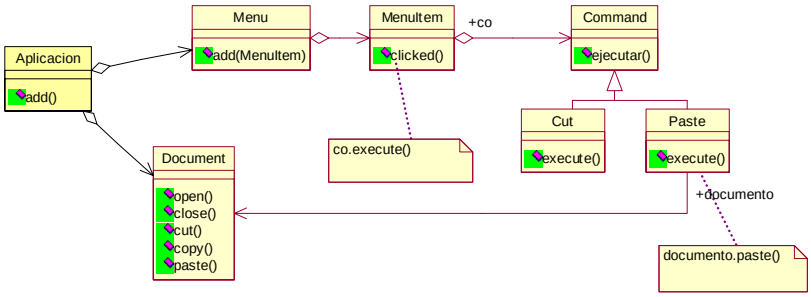
179

## Command (Orden)

- **Propósito**
  - **Encapsula un mensaje como un objeto**, permitiendo parametrizar los clientes con diferentes solicitudes, añadir a una cola las solicitudes y soportar funcionalidad *deshacer/rehacer* (*undo/redo*)
- **Motivación**
  - Algunas veces es necesario enviar mensaje a un objeto sin conocer el selector del mensaje ni el objeto receptor.
  - Por ejemplo *widgets* (botones, menús,..) realizan una acción como respuesta a la interacción del usuario, pero no se puede explicitar en su implementación.

180

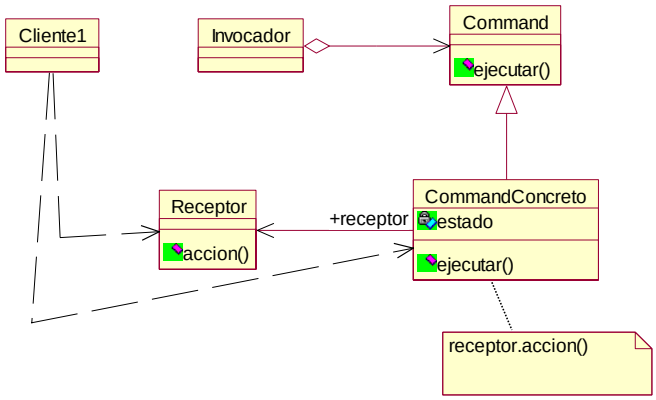
# Command




**Motivación**

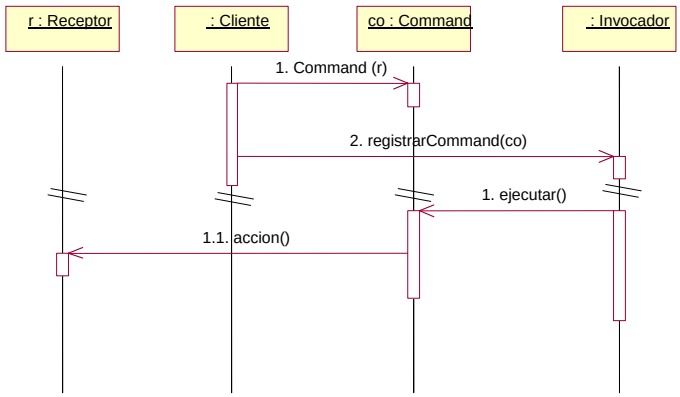


# Command




**Estructura**

# Command




**Colaboración**



# Command

- **Aplicabilidad**
  - Parametrizar objetos por la acción a realizar (alternativa a funciones ***Callback:*** *función que es registrada en el sistema para ser llamada más tarde;* en C++ se puede usar punteros a funciones)
  - Especificar, añadir a una cola y ejecutar mensajes en diferentes instantes: un objeto *Command* tiene un tiempo de vida independiente de la solicitud original.
  - Soportar facilidad *undo/redo*.
  - Recuperación de fallos.

## Command

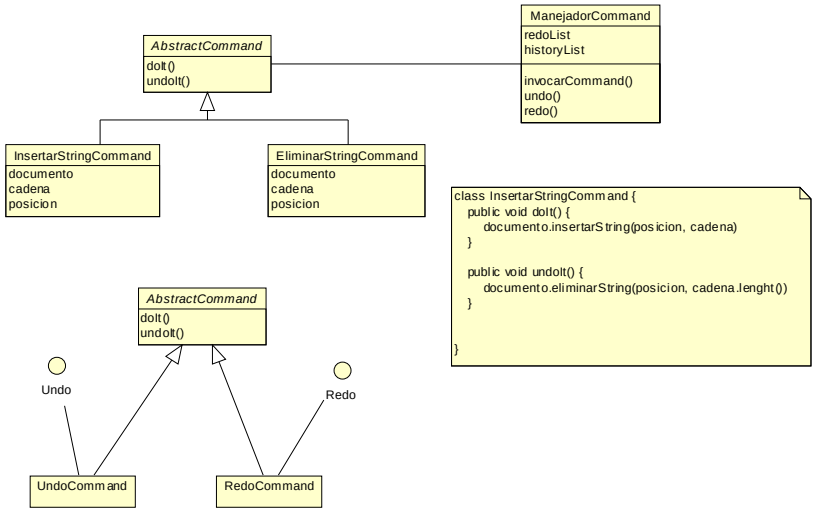
- **Consecuencias**
  - Desacopla el objeto que invoca la operación del objeto que sabe cómo realizarla.
  - Cada subclase *CommandConcreto* especifica un par receptor/acción, almacenando el receptor como un atributo e implementando el método *ejecutar*.
  - Objetos *command* pueden ser manipulados como cualquier otro objeto.
  - Se pueden crear *command* compuestos (aplicando el patrón *Composite*).
  - Es fácil añadir nuevos *commands*.
  - Un objeto *Command* es realmente un *Adapter*.



## Command

- **Implementación**
  - ¿Cuál debe ser la “inteligencia” de un *command*?
    - No delegar en nadie
    - Encontrar dinámicamente el objeto receptor y delegar en él
  - Soportar *undo/redo*
    - Atributos para almacenar estado y argumentos de la operación.
    - Lista de objetos *commands*
    - En la lista se colocan copias.

# Mecanismo undo/redo






# Mecanismo undo/redo

```
public void invocarCommand (AbstractCommand command){
   if (command instanceof Undo) {
            undo(); return; }
   if (command instanceof Redo) {
            redo(); return; }
   if (command.doIt()) {
            addToHistory(command);
   } else { historiaList.clear();}
   if (redoList.size() > 0) redoList.clear();
}
private void undo() {
   if (historyList.size() > 0) {
      AbstractCommand ac;
      undoCmd = (AbstractCommand) historyList.removeFirst();
      undoCmd.undoIt();
      redoList.addFirst(undoCmd);
}
```

## Command

- En lenguajes como Eiffel o Java no es posible que un método pueda actuar como argumento de otro método. Por ello es necesario definir una clase (***functor*** ) que encapsule al método y sus instancias se pasarán como argumento.
- Ejemplos en Java: Comparator, Observer, ActionListener, ...

189

## Ejemplos *Functor*

- **Ejemplo 1:** Comparar dos objetos utilizando cualquier criterio de ordenación: interfaz Comparator en Java

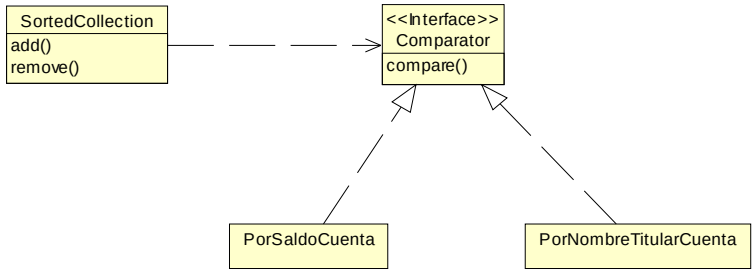
```
public interface Comparator {
    int compare (Object o1, Object o2); }
```

  Un objeto Comparator puede ser pasado a una rutina de ordenación o ser utilizado por una colección para mantener el orden de sus elementos.

- **Ejemplo 2:** Interfaz Observer en el patrón Observer.
- **Ejemplo 3:** Interfaces *listener* en el modelo de delegación de eventos.

190

# Ejemplo




191

# Command

- **Implementación**
  - Podemos evitar la jerarquía de subclases de *Command* si no necesitamos la facilidad *undo/redo* con metaclases. En Smalltalk,

    ```
    self receptor perform: unSelector with: unArg
    ```

  - Un objeto *functor* normalmente implementa el comportamiento deseado sin delegar. Un objeto *command* normalmente delega en otro objeto.
  - Un objeto *command* actúa como un *functor* si tan sólo implementa el comportamiento deseado.

192

## Command

- **Ejercicio**: Escribir una clase con la funcionalidad de ejecutar periódicamente uno o más métodos sobre diferentes objetos. Por ejemplo, ejecutar una operación de copia de seguridad cada hora o una operación de comprobación de estado de un disco cada 10 minutos o imprimir la secuencia de Fibonacci. Se desea que la clase no conozca qué operaciones concretas debe ejecutar ni los objetos sobre los que debe aplicarlas. Esto es, se debe desacoplar la clase que planifica la ejecución de esos métodos de las clases que realmente proporcionan las operaciones a ejecutar.



## Interpreter

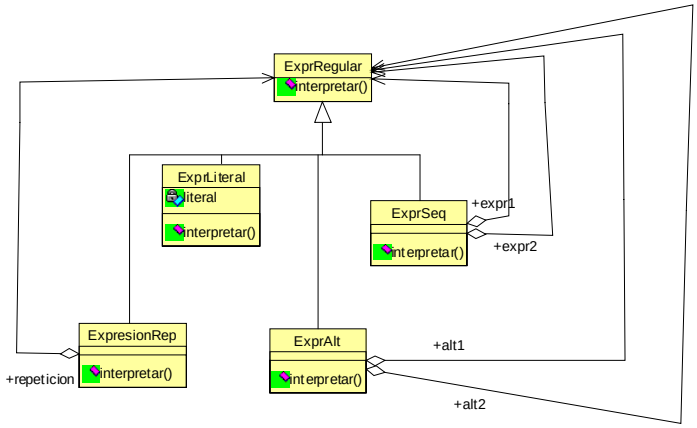
- **Propósito**
  - **Dado un lenguaje, definir una representación para su gramática junto con un intérprete** que utiliza la representación para interpretar sentencias en dicho lenguaje.
- **Motivación**
  - Interpretar una expresión regular.
  - Usa una clase para representar cada regla, los símbolos en la parte derecha son atributos.
- **Aplicabilidad**
  - Usar si la gramática es simple y la eficiencia no es importante.

# Intérprete

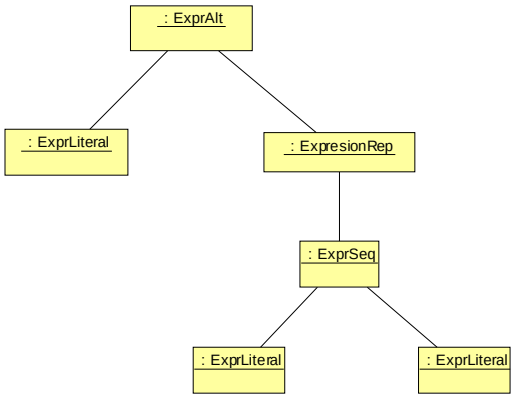
expre := literal | cond | seq | rep
alt := expre ‘|’ expre
seq := expre ‘&’ expre
rep := expre ‘*’
literal := ‘a’|’b’|’c’|…{‘a’|’b’|’c’|…| } *



# Intérprete



# Intérprete

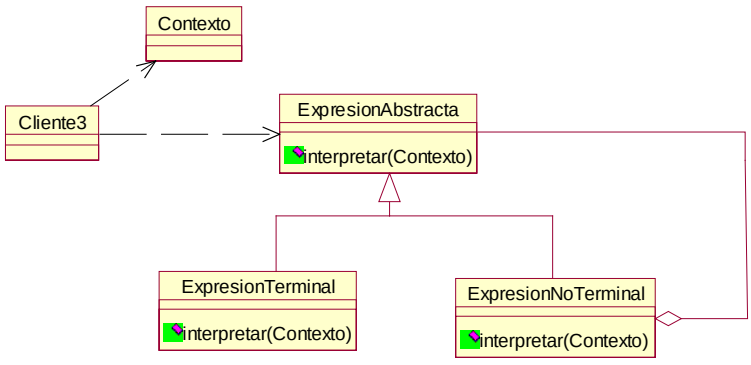



lluvias | (trasvase & desaladoras) *

197

# Interpreter

**Estructura**




198

## Iterator (Iterador)

- **Propósito**
  - Proporciona una forma para acceder a los elementos de una estructura de datos sin exponer los detalles de la representación.
- **Motivación**
  - Un objeto contenedor (agregado o colección) tal como una lista debe permitir una forma de recorrer sus elementos sin exponer su estructura interna.
  - Debería permitir diferentes métodos de recorrido
  - Debería permitir recorridos concurrentes
  - No queremos añadir esa funcionalidad a la interfaz de la colección

199

## Iterator (Iterador)

- **Motivación**
  - Una clase *Iterator* define la interfaz para acceder a una estructura de datos (p.e. una lista).
  - Iteradores ***externos*** vs. Iteradores ***internos***.
  - Iteradores externos: recorrido controlado por el cliente
  - Iteradores internos: recorrido controlado por el iterador

200

# Iterador Externo




# Iterador Externo

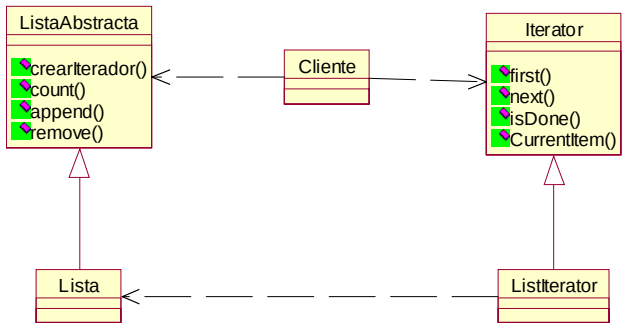
```
List lista = new List();
...
ListIterator iterator = new ListIterator(lista);

iterator.first();
while (!iterator.isDone()) {
    Object item = iterator.item();
    // código para procesar item
    iterator.next();
}
...
```

# Iteradores Externos en Java

```
interface Iterator {
    boolean hasNext();
    Object next();
    void remove();
}
Iterator it = lista.iterator();
while (it.hasNext()) {
    Object item = it.next();
        // código para procesar item
}
```



# Iterador Externo Polimórfico



## Iterador Interno

- La clase que modela la iteración ofrece métodos que controlan la ejecución: ejecutar una acción sobre todos los elementos, ejecutar una acción sobre aquellos elementos que cumplan una condición,etc.
  - Ejemplo: iteradores en Smalltalk y Ruby
- Dados como parámetros una acción y/o condición, el método de iteración se encarga de recorrer la colección.
- En Eiffel y Java no podemos pasar rutinas o código como argumento, en lenguajes OO tipados dinámicamente como Smalltalk o Ruby se dispone de *bloques* de código.

205

## Iterador externo

- En Smalltalk, sea `colCuentas` una colección de instancias de `Cuenta`
  - `colCuentas select:[:cuenta| cuenta.saldo > 1000]`
  - `colCuentas reject:[:cuenta| cuenta.saldo > 1000]`
  - `colCuentas do: [:cuenta| cuenta.reintegro:1000`

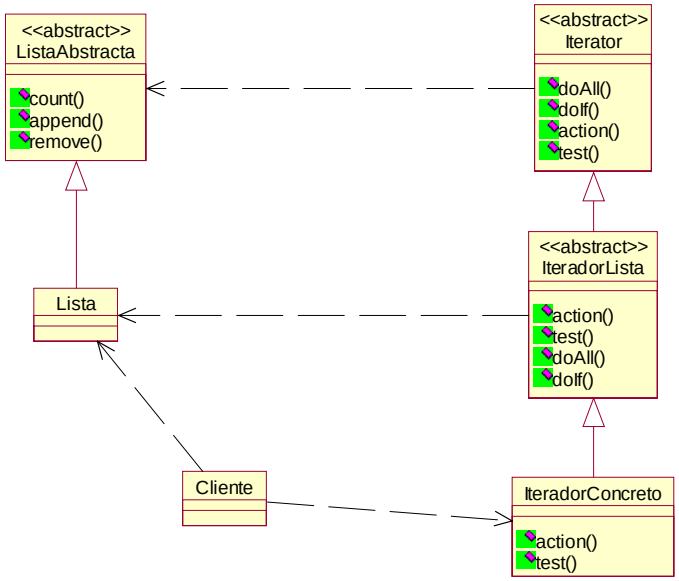
206

# Iterador Interno

```
public void doIf() {
    Iterator it = col.iterator();
    while (it.hasNext()) {
            Object o = iterator.next();
            if (test(o)) action(o)
    }
}
```



# Iterador Interno



# Iterador

- **Consecuencias**
  - Simplifica la interfaz de un contenedor al extraer los métodos de recorrido
  - Permite varios recorridos concurrentes
  - Soporta variantes en las técnicas de recorrido



# Iterador

- **Implementación**
  - ¿Quién controla la iteración?
    - Externos vs. Internos
  - ¿Quién define el algoritmo de recorrido?
    - Agregado: iterador sólo almacena el estado de la iteración (*Cursor*). Ejemplo de uso del patrón Memento
    - Iterador: es posible reutilizar el mismo algoritmo sobre diferentes colecciones o aplicar diferentes algoritmos sobre una misma colección

## Iterador

- **Implementación**
  - ¿Es posible modificar la colección durante la iteración?
  - Colección e iterador son clases íntimamente relacionadas: clases amigas en C++ y clases internas en Java
  - Suele ser usado junto con el patrón Composite.

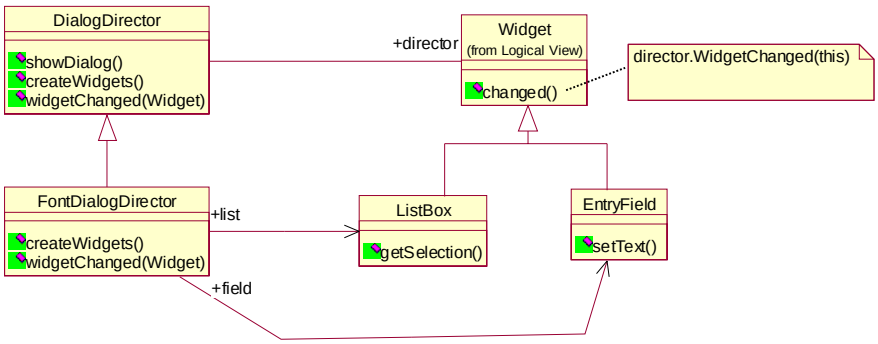


## Mediator (Mediador)

- **Propósito**
  - Define un **objeto que encapsula cómo interaccionan un conjunto de objetos**. Favorece un bajo acoplamiento, liberando a los objetos de referenciarse unos a otros explícitamente, y permite variar la interacción de manera independiente.
- **Motivación**
  - Muchas interconexiones entre objetos dificulta la reutilización y la especialización del comportamiento.
  - Ventana que incluye un conjunto de elementos gráficos con dependencias entre ellos.

# Mediador

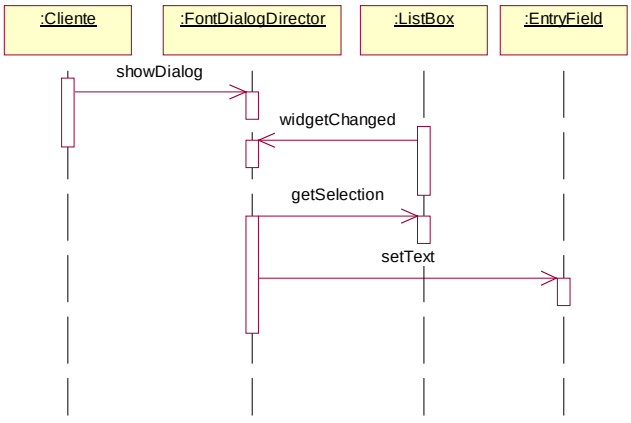
## Motivación




213

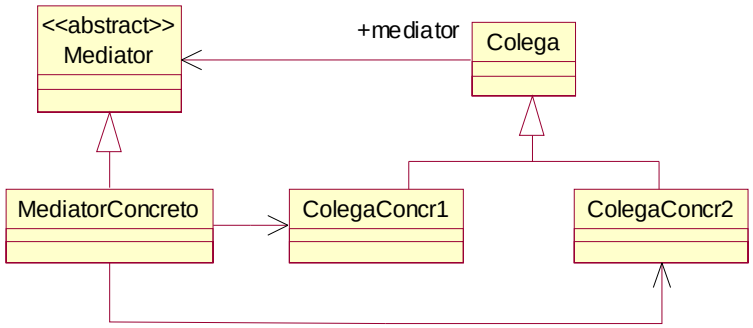
# Mediador

## Motivación




214

# Mediador




**Estructura**



# Mediador

- **Aplicabilidad**
  - Un conjunto de objetos se comunica entre sí de una forma bien definida, pero compleja. Las interdependencias son poco estructuradas y difíciles de comprender.
  - Reutilizar una clase es difícil porque tiene dependencias con muchas otras clases.
  - Un comportamiento que es distribuido entre varias clases debería ser adaptable sin crear muchas subclases.

## Mediador

- **Consecuencias**
  - Evita crear subclases de los colegas, sólo se crean subclases del *mediador*.
  - Desacopla a los *colegas*.
  - Simplifica los protocolos entre las clases
  - Abstrae el cómo cooperan los objetos
  - Centraliza el control en el mediador: clase difícil de mantener



## Mediador

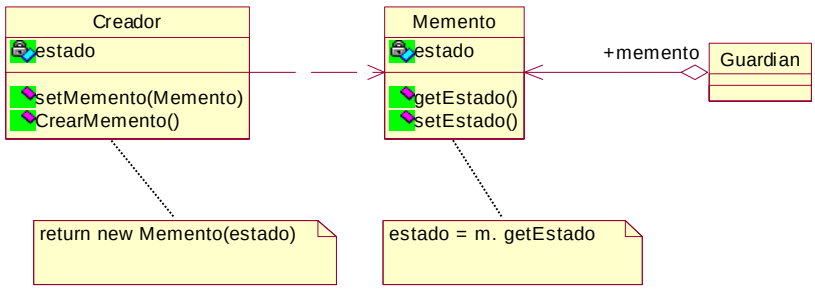
- **Implementación**
  - No hay necesidad de definir una clase abstracta Mediator si los colegas trabajan con un único mediador.
  - Los colegas deben comunicarse con el mediador cuando un evento de interés ocurre, varias posibilidades:
    - patrón Observer
    - interfaz de notificación especializada en Mediator (por ejemplo en Smalltalk-V, la clase ViewManager)

# Memento

- **Propósito**
  - **Captura y externaliza el estado interno de un objeto, sin violar la encapsulación, de modo que el objeto puede ser restaurado a este estado más tarde.**
- **Motivación**
  - Algunas veces es necesario registrar el estado interno de un objeto: mecanismos *checkpoints* y deshacer cambios que permiten probar operaciones o recuperación de errores.

219

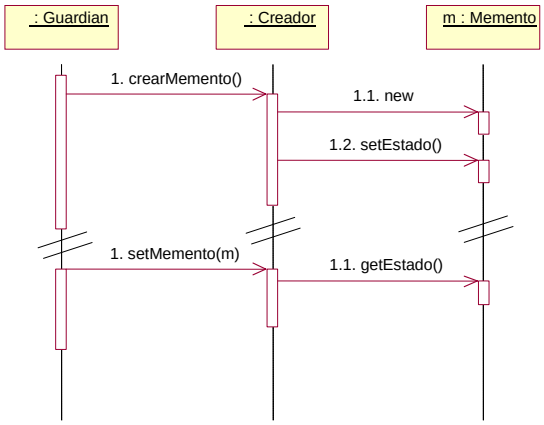
# Memento




**Estructura**

220

# Memento




**Colaboración**



# Memento

- **Aplicabilidad**
  - Una parte del estado de un objeto debe ser guardado para que pueda ser restaurado más tarde y una interfaz para obtener el estado de un objeto podría romper la encapsulación exponiendo detalles de implementación.

## Memento

- **Consecuencias**
  - Mantiene la encapsulación
  - Simplifica la clase *Creador* ya que no debe preocuparse de mantener las versiones del estado interno.
  - Podría incurrir en un considerable gasto de memoria: encapsular y restaurar el estado no debe ser costoso.
  - Puede ser difícil en algunos lenguajes asegurar que sólo el *Creador* tenga acceso al estado del *Memento*.



## Memento

- **Implementación**
  - *Memento* tiene dos interfaces, una para los creadores y otra para el resto de clases: clases *amigas* en C++, *exportación selectiva* (Eiffel), acceso *paquete* en Java.
  - Un memento puede registrar cambio incremental sobre el estado del creador: cuando se usa el patrón *Command* para el mecanismo *undo/redo*.

# Memento (Ejemplo)

```
package memento;

class Memento {
        String estado;
        Memento(String estadoAGuardar) { estado = estadoAGuardar; }
}

package memento;

class Creador {
        private String estado;
        public Memento crearMemento() {
                return new Memento(estado);
        }
        public void setMemento(Memento m) {
                estado = m.estado; }
}
```



# Memento (Ejemplo)

```
package otro;

class Guardian {
    private ArrayList<Memento>  estadosGuardados = new ArrayList<Memento>();
    public void addMemento (Memento m) { estadosGuardados.add(m); }
    public Memento getMemento (int index) {
                        return estadosGuardados.get(index); }
}
package otro;

class EjemploMemento {
    public static void main(String[] args) {
        Guardian g = new Guardian();
        Creador c = new Creador();
        ...
        g. addMemento(c. crearMemento());
        ...
        g. addMemento(c. crearMemento() );
        ...
        c. setMemento( g. getMemento(1) );
    }
}
```

## Memento: Ejemplo *Cursor*

```
template <class G>
class Collection {
     IterationState* createInitialState();
     void next(IterationState*);
     bool isDone(IterationState*);
     G item() (IterationState*);
    void append(G)
     void remove(G)
     //...
}
```

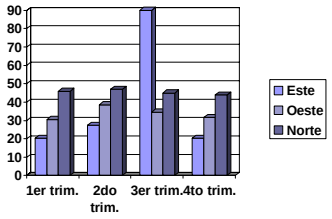
```
Collection<Cuenta*> cc;
IterationState* estado;

estado = cc.createInitialState();

while (!cc.isDone(estado)) {
    cc.item(estado)->print();
     cc.next(estado)
```

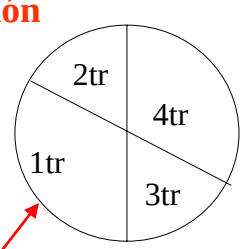


## Observer / Publish-Subscribe (Observador)

- **Propósito**
  - Define una **dependencia uno-a-muchos entre objetos**, de modo que **cuando cambia el estado de un objeto**, **todos sus dependientes automáticamente son notificados** y actualizados.
- **Motivación**
  - ¿Cómo mantener la consistencia entre objetos relacionados, sin establecer un acoplamiento fuerte entre sus clases?
  - Ejemplo: Separación *Modelo-Vista*

## Motivación




Vistas

Modelo

t1=20
t2=25
t3=90
t4=20

| | 1tr. | 2tr. | 3tr. | 4tr. |
|---|---|---|---|---|
| Este | 20 | 25 | 90 | 20 |
| Oeste | 30 | 38 | 32 | 32 |
| Norte | 47 | 47 | 45 | 45 |

## Separación Modelo-Vista

- Los objetos del modelo (dominio) no deben conocer directamente a los objetos de la vista (presentación).
- Las clases del dominio encapsulan la información y el comportamiento relacionado con la lógica de la aplicación.
- Las clases de la interfaz (ventanas) son responsables de la entrada y salida, capturando los eventos, pero no encapsulan funcionalidad de la aplicación.

230

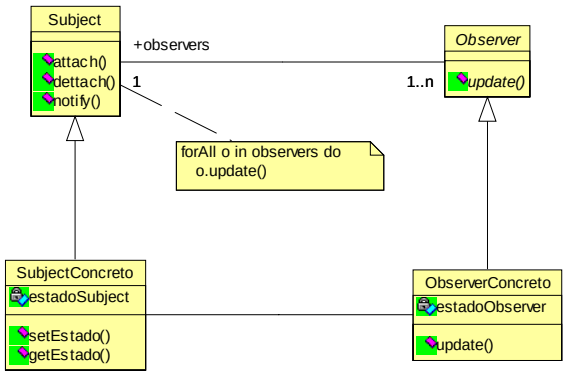
# Separación Modelo-Vista

- Justificación
  - Clases cohesivas
  - Permitir separar el desarrollo de las clases de la vista y del dominio
  - Minimizar el impacto de los cambios en la interfaz sobre las clases del modelo.
  - Facilitar conectar otras vistas a una capa del dominio existente.
  - Permitir varias vistas simultáneas sobre un mismo modelo.
  - Permitir que la capa del modelo se ejecute de manera independiente a la capa de presentación.



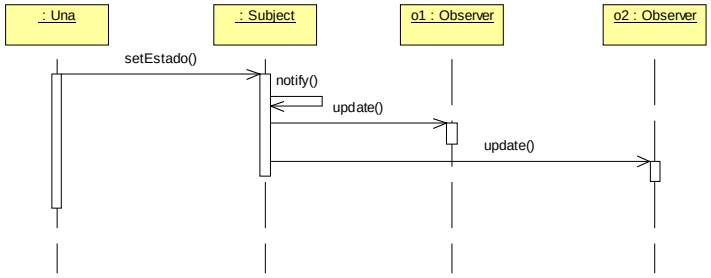
# Observer

Estructura



# Observer

**Colaboración**






# Observer

- **Aplicabilidad**
  - Cuando un cambio de estado en un objeto requiere cambios en otros objetos, y no sabe sobre qué objetos debe aplicarse el cambio.
  - Cuando un objeto debe ser capaz de notificar algo a otros objetos, sin hacer asunciones sobre quiénes son estos objetos.

# Observer

- **Consecuencias**
  - Acoplamiento abstracto y mínimo entre *Subject* y *Observer:*
    - `Subject` no necesita conocer las clases concretas de *observers*
    - permite modificar independientemente *subjects* y *observers*
    - pueden reutilizarse por separado
    - pueden añadirse *observers* sin modificar el *subject*



# Observer

- **Consecuencias**
  - Soporte para comunicación de tipo *broadcast*
    - el *subject* no especifica al receptor concreto de la notificación
    - es posible añadir y eliminar *observers* en cualquier instante
  - Actualizaciones inesperadas.
  - Una interfaz simple de `Observer` requiere que los *observers* tengan que deducir el ítem cambiado.

# Observer

- **Implementación**
  - Es posible que un *observer* esté ligado a más de un *subject*: la operación update tendrá como argumento el *subject*
  - ¿Quién dispara la notificación?
    - Métodos *set* en la clase `Subject`
    - Clientes de la clase `Subject`
  - Asegurarse de notificar con el estado de *Subject* consistente
  - ¿Cuánta información sobre el cambio se le envía a los *observers* con la notificación?
    - Modelo *Push* (Mucha) y Modelo *Pull* (Muy Poca)



# Observer

- **Implementación**
  - Al registrar un *observer* es posible asociarle el evento sobre el que quiere ser notificado.

    attach(Observer obs, Evento interes);

  - Cuando las relaciones de dependencia entre *subjects* y *observers* son complicadas encapsular la semántica de actualización en una clase *ManejadorCambio (Mediador).*
  - En Smalltalk, las interfaces de las clases *Subject* y *Observer* se definen en la clase *Object*.

# Observer en Java

```
public class Observable {
   private boolean changed = false;
   private Vector obs;
    public Observable() {
            obs = new Vector();
   }
    public synchronized void addObserver (Observer o) {..}
    public synchronized void deleteObserver (Observer o) {..}
    public void notifyObservers (Object arg) {..}
    protected synchronized void setChanged() {..}
     protected synchronized void clearChanged() {..}
    public synchronized boolean hasChanged() {.. }
```

239

# Observer en Java

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

Argumento pasado al método notifyObservers, puede indicar el tipo de cambio

240

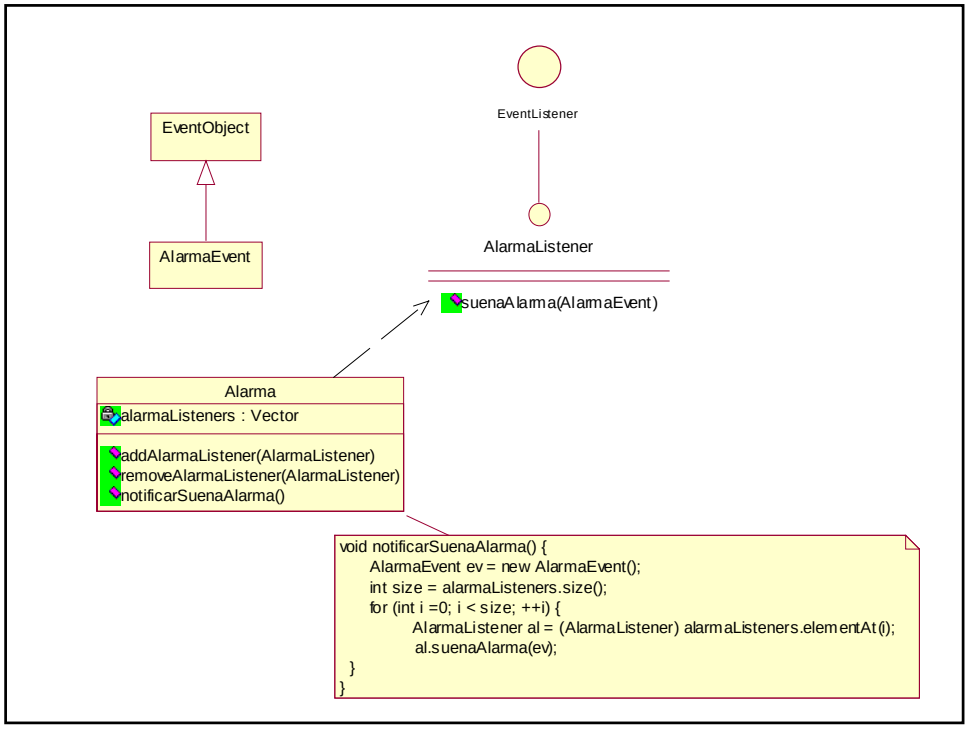
## Observer en Java

- **Problema**
  - Que la clase que deseamos que actúe como *Observable* ya herede de otra clase: ¡En Java no hay herencia múltiple!
    - Usar delegación
    - Una clase ConcreteSubject contendrá a un objeto de una clase que heredará de Observable, y tendrá implementaciones "*wrapper*" de addObserver *y* deleteObserver.
    - Se delega el comportamiento que necesita ConcreteSubject al objeto Observable que contiene.
    - ¿Por qué una subclase de *Observable*?



## Modelo de Eventos de Java 1.1

- Java 1.1 introdujo un nuevo modelo de eventos basado en el patrón `Observer`.
- Objetos que pueden generar eventos son llamados *fuentes de eventos (event sources)*.
- Objetos que desean ser notificados de eventos son llamados *oyentes de eventos* (*event listeners*).
- Una *fuente* juega el papel de *ConcreteSubject* y un *listener* juega el papel de *ConcreteObserver*.
- Los *listeners* deben registrarse en las *fuentes.*
- Un *listener* debe implementar una interfaz que proporciona los métodos que deben ser llamados por la *fuente* para notificar un evento.




# Modelo de Eventos de Java 1.1

- Es un modelo no orientado a eventos de componentes GUI.
- Hay 11 interfaces listener cada una apropiada para un diferente tipo de evento GUI:
  - `ActionListener`, `ItemListener`, `KeyListener`, `MouseListener`, `WindowListener`, ...
- En algunos casos la interfaz *listener* incluye más de un método: Java proporciona adaptadores

244

# Modelo de Eventos de Java 1.1

```
public class CounterView extends Frame {
    private TextField tf = new TextField(10);
    private Counter counter;                    // referencia al modelo
    public CounterView(String title, Counter c) {
        super(title); counter = c;
        Panel tfPanel = new Panel();
        ...
        Button incButton = new Button("Increment");
        incButton.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                counter.incCount();
                tf.setText(counter.getCount() + ""); } } );
        buttonPanel.add(incButton);
        ...
```



# Modelo de Eventos de Java 1.1

- Si creamos dos instancias de `CounterView` para el mismo objeto contador, ¿Qué debemos hacer para que un cambio en el valor de contador en una de ellas afecte a la otra?

```
public class ObservableCounter extends Observable {
    private int count;
    public ObservableCounter(int count) { this.count = count; }
    public int getCount() { return count; }
    public void incCount() {
        count++;
        setChanged();
        notifyObservers(); }
    ... }
```

# Modelo de Eventos de Java 1.1

```
public class ObservableCounterView extends Frame {
    private ObservableCounter counter;
    private TextField tf = new TextField(10);
    public ObservableCounterView(String title,
    ObservableCounter c) {
        super(title);  counter = c;
        counter.addObserver(new Observer() {
            public void update(Observable src, Object obj) {
                if (src == counter) {
                    tf.setText(((ObservableCounter)src).getCount() + "");}
        } } );
        ...
```

247

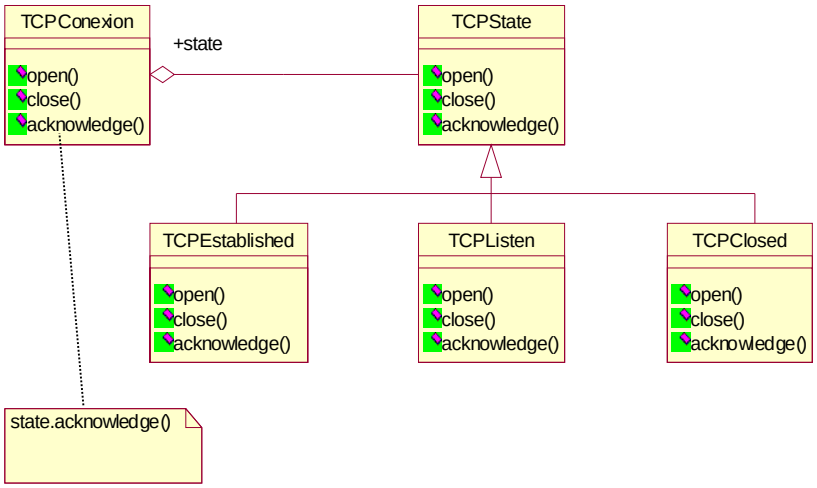
# State (Estado)

- **Propósito**
  - Permite a un objeto cambiar su comportamiento cuando cambia su estado. El objeto parece cambiar de clase.
- **Motivación**
  - Una conexión TCP puede encontrarse en uno de varios estados, y dependiendo del estado responderá de un modo diferente a los mensajes de otros objetos para solicitudes tales como abrir, cerrar o establecer conexión.

248

# Estado

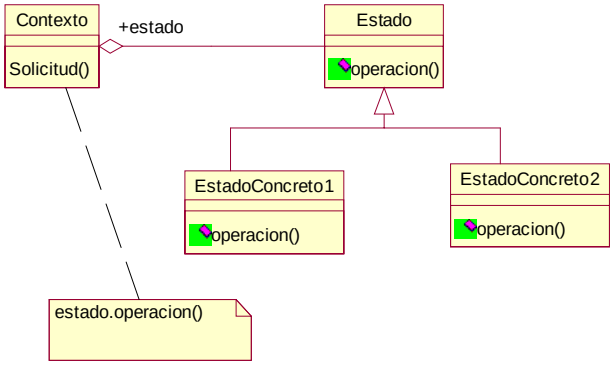
## Motivación




249

# Estado

## Estructura




250

## Estado

- **Aplicabilidad**
  - El comportamiento del objeto depende de su estado, y debe cambiar su comportamiento en tiempo de ejecución dependiendo de su estado.
  - Las operaciones tienen grandes estructuras CASE que dependen del estado del objeto, que es representado por uno o más constantes de tipo enumerado.



## Estado

- **Consecuencias**
  - Coloca todo el comportamiento asociado a un particular estado en una clase.
  - Subclases vs. Sentencias CASE.
  - Ayuda a evitar estados inconsistentes
  - Transiciones de estado son más explícitas.
  - Incrementa el número de objetos
  - Los objetos *estado* pueden ser *Singleton*.

## Estado

- **Implementación**
  - ¿Quién define las transiciones entre estados? *Contexto* o subclases estado.
  - Una alternativa es definir tablas que representan la matriz de transiciones de estado: no es posible asociar acciones a los cambios de estado y más difícil de comprender.
  - ¿Cuándo son creados los objetos *estado*? Pueden crearse cuando se necesiten o con antelación y que *Contexto* tenga una referencia a ellos.

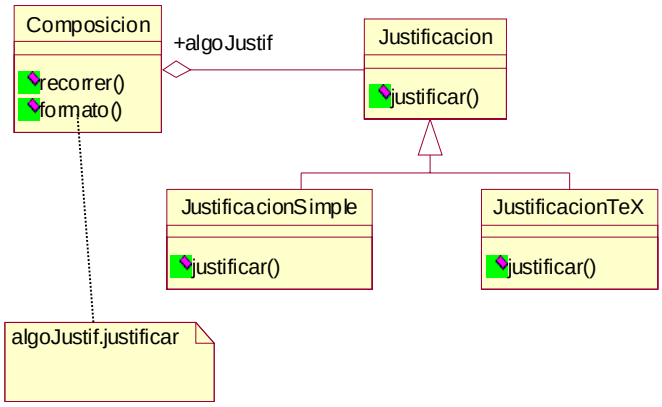


## Strategy/Policy (Estrategia)

- **Propósito**
  - Define una familia de algoritmos, encapsula cada uno, y permite intercambiarlos. Permite variar los algoritmos de forma independiente a los clientes que los usan
- **Motivación**
  - Existen muchos algoritmos para justificación de texto, ¿debe implementarlo el cliente que lo necesita?

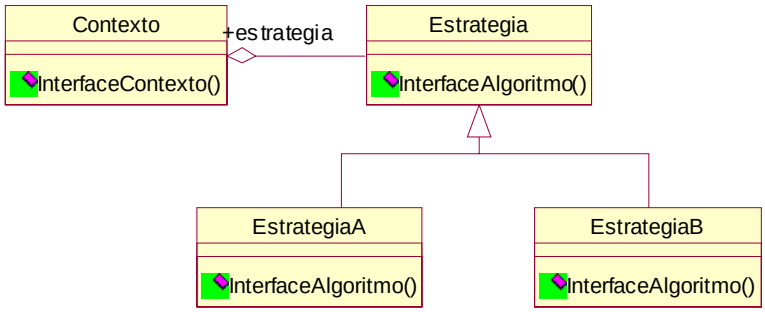
# Estrategia

**Motivación**




255

# Estrategia




**Estructura**

256

## Estrategia

- **Aplicabilidad**
  - Configurar una clase con uno de varios comportamientos posibles.
  - Se necesitan diferentes variantes de un algoritmo.
  - Una clase define muchos comportamientos que aparecen como sentencias CASE en sus métodos.



## Estrategia

- **Consecuencias**
  - Define una familia de algoritmos relacionados.
  - Una alternativa a crear subclases de la clase *Contexto*.
  - Elimina sentencias CASE
  - El cliente puede elegir entre diferentes estrategias o implementaciones: debe conocer detalles
  - Se incrementa el número de objetos: usar *Flyweight*
  - *State* y *Strategy* son similares, cambia el *Propósito*: ejemplos de composición con delegación

## Estrategia

- **Implementación**
  - ¿Cómo una estrategia concreta accede a los datos del contexto?
    - Pasar datos como argumentos
    - Pasar el contexto como argumento
    - *Estrategia* almacena una referencia al contexto



## Template Method (Método Plantilla)

- **Propósito**
  - **Define el esqueleto (esquema, patrón) de un algoritmo en una operación, difiriendo algunos pasos a las subclases**. Permite a las subclases redefinir ciertos pasos de un algoritmo sin cambiar la estructura del algoritmo.
- **Motivación**
  - Fundamental para escribir código en un framework.
  - Clase *Aplicación* que maneja objetos de la clase *Documento*: método *OpenDocument*

## Método Plantilla

```
void openDocument (String nombre) {
    if (! canOpenDocument (nombre)) { return; }
    Document doc = createDocument();
    if (doc != null) {
                docs. addDocument(doc);
                aboutToOpenDocument (doc);
                doc.open();
                doc.read();
    }
}
```

261

## Método Plantilla

```
has(v:G): Boolean is do
    from start until after or else equal (v,item)
    loop forth  end
    Result := not after
end
```

262

## Método Plantilla

- **Aplicabilidad**
  - Implementar las partes fijas de un algoritmo y dejar que las subclases implementen el comportamiento que puede variar.
  - Cuando el comportamiento común entre varias subclases debe ser factorizado y localizado en una superclase común.
  - Controlar extensiones de las subclases: algoritmos con *puntos de extensión*



## Método Plantilla

- **Consecuencias**
  - Técnica fundamental para la reutilización: factorizar comportamiento común en librerías de clases
  - Estructura de control invertida conocida como “Principio de *Hollywood*”: “No nos llames, nosotros te llamaremos”.
  - Un método plantilla invoca a los siguientes tipos de métodos:
    - operaciones abstractas
    - operaciones concretas en la clase abstracta
    - operaciones concretas en clientes
    - métodos factoría
    - métodos *hook* que proporcionan comportamiento por defecto

## Método Plantilla

- **Implementación**
  - Minimizar el número de métodos que es necesario redefinir en las subclases.
  - Nombrar a los métodos que se deben redefinir añadiéndole cierto prefijo, por ejemplo “do”.
  - En C++, métodos a redefinir son protegidos y virtuales, el método plantilla no será virtual.

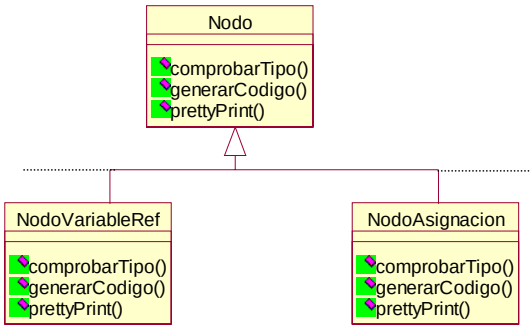


## Visitor (Visitante)

- **Propósito**
  - **Representar una operación que debe ser aplicada sobre los elementos de una estructura de objetos**. Permite definir una nueva operación sin cambiar las clases de los elementos sobre los que opera.
- **Motivación**
  - Un compilador que representa los programas como árboles de derivación de la sintaxis abstracta necesita aplicar diferentes operaciones sobre ellos: comprobación de tipos, generación de código, .. y además listados de código fuente, reestructuración de programas,...

# Visitor

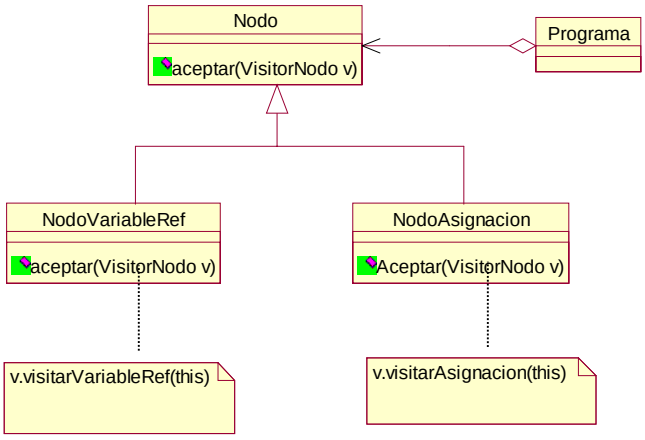
## Motivación




267

# Visitor

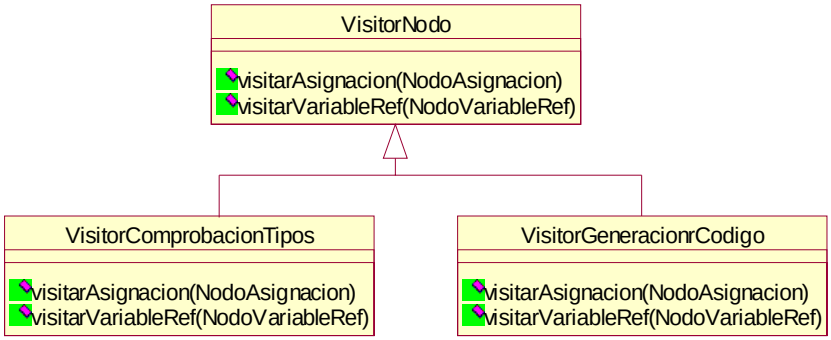
## Motivación




268

# Visitor

**Motivación**






# Visitor

- **Aplicabilidad**
  - Tenemos una jerarquía de clases que representan objetos de propósito general (p.e. nodos de un árbol de derivación de sintaxis) y podemos utilizarlo en diferentes aplicaciones, lo que implicaría añadir métodos en las clases de la jerarquía.
  - Una estructura de objetos contiene muchas clases de objetos con diferentes interfaces, y se quiere realizar operaciones sobre los objetos que dependen de las clases concretas.
  - Las clases definiendo la estructura de objetos cambian con poca frecuencia, pero a menudo se definen nuevas operaciones sobre la estructura. Mejor definir las operaciones en clases aparte.

## Visitor

- **Consecuencias**
  - Facilidad para añadir nuevas operaciones: en vez de distribuir la funcionalidad, se añade un nuevo *visitor*.
  - Un objeto *visitor* recoge comportamiento relacionado, lo que simplifica las clases de los elementos.
  - Es difícil añadir nuevas subclases de elementos concretos, ya que implica cambiar la jerarquía de *Visitor*.
  - A diferencia de un iterador, *Visitor* puede visitar objetos de clases que no tienen una superclase común.
  - Compromete la encapsulación: los elementos concretos deben permitir a los *visitors* hacer su trabajo.



## Visitor

- **Implementación**
  - El patrón permite añadir operaciones a clases sin cambiarlas. Esto se consigue aplicando la técnica ***double-dispatch***.
  - ¿Quién es responsable del recorrido de la estructura de objetos?
    - Estructura de objetos
    - Un iterador
    - El *visitor*

# *Double-Dispatch (Smalltalk)*

```
Point>> + delta
    ^delta isPoint
            ifTrue:[(x+delta x) @ (y + delta y)]
            ifFalse:[(x+delta) @ (y + delta)]

Point>> + delta
    ^delta addPoint: self
Number>> addPoint: aPoint
    ^(self + aPoint x) @ (self + aPoint y)
Point>> addPoint: aPoint
    ^(self  x + aPoint x) @ (self  y + aPoint y)
```