

# **RESUMENTDSGERMANSANCHEZ.pdf**



**germansnchz77**



**Tecnologías de Desarrollo de Software**



**3º Grado en Ingeniería Informática**



**Facultad de Informática  
Universidad de Murcia**



**Accede al documento original**

**70 años formando talento  
que transforma el futuro.**

La primera escuela de negocios de España,  
hoy líder en sostenibilidad y digitalización.



**EOI** Escuela de  
organización  
Industrial



**Descubre EOI**

# Google Gemini: Plan Pro a 0€ durante 1 año. **Tu ventaja por ser estudiante.**



Domina cualquier tema con el Aprendizaje Guiado.

Puedes explicarme como se crea un eclipse lunar completo y

¡Claro vamos paso a paso para que lo entiendas a la perfección! ☺



Aprendizaje Guiado

Oferta válida hasta el 9 de diciembre de 2025

[Consigue la oferta](#)

Después 21,99€/mes

## Tema 1: Algunos principios básicos de diseño orientado a objetos

## 1 Modelado del software con UML

"Un modelo es una **representación de la realidad** que es obtenida por medio de un proceso de **abstracción** con el propósito de ayudar a comprender y razonar sobre esa realidad". Un modelo es expresado en un **lenguaje**.

**Abstracción:** Se ignoran los detalles y nos centramos en las características esenciales que nos interesan.

Un Modelo del Software es una descripción abstracta de algún aspecto de los sistemas software, por ejemplo: requisitos, estructura, comportamiento, datos, y se construye el contenedor especificado

A mediados de los noventa existían muchos métodos de análisis y diseño OO: Mismos conceptos con distinta notación. **Mucha confusión.**

- En 1994, Grady Booch, James Rumbaugh e Ivar Jacobson deciden unificar las notaciones de sus métodos: [Unified Modeling Language \(UML\)](#).
  - Proceso de estandarización promovido por el [OMG](#).
    - Propone, elabora y mantiene especificaciones para aplicaciones empresariales distribuidas e interoperables.
    - Estándares OMG: Corba, UML, OCL, MDA, ..
  - Ventajas de UML:
    - Eliminar confusión y proporcionar estabilidad al mercado.
    - Reunir puntos fuertes de cada método y añadir mejoras.

**UML** es un lenguaje para **visualizar, especificar, construir y documentar** los **modelos** de un sistema software, desde una perspectiva orientada a objetos.

## Objetivos en su diseño

- Modelar desde los requisitos al despliegue
  - Modelar todo tipo de sistemas
  - Utilizable por personas y ejecutable
  - Equilibrio entre simplicidad y potencia expresiva

**Los modelos son útiles para comunicar, documentar y razonar. También se puede generar código directamente.**

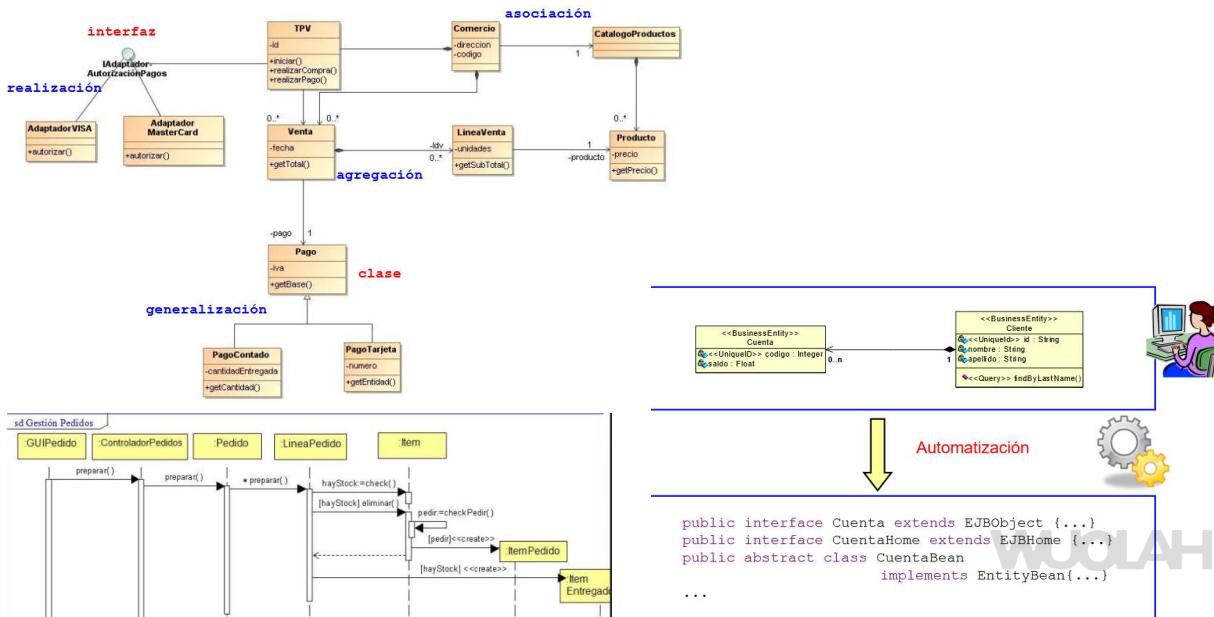
Un **modelo UML** es una **especificación** de un aspecto del sistema de interés para un grupo de usuarios o desarrolladores.

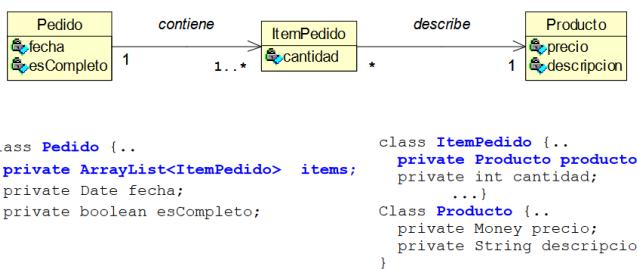
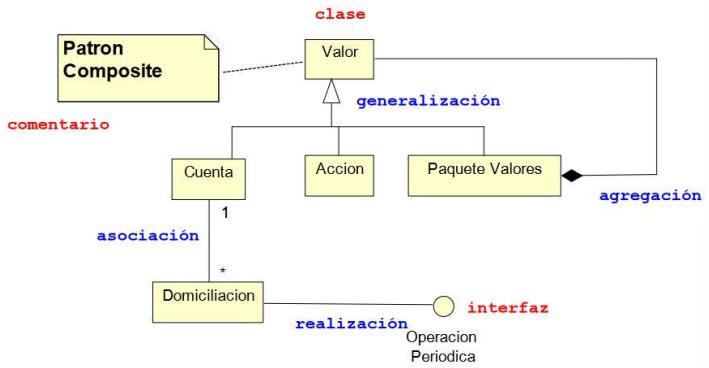
Un **diagrama UML** representa una parte de la información del modelo en forma gráfica o textual.

- En un diagrama de clases, por ejemplo, no aparecen todas las clases, atributos, métodos y asociaciones, sino aquellas que se consideran relevantes para el propósito

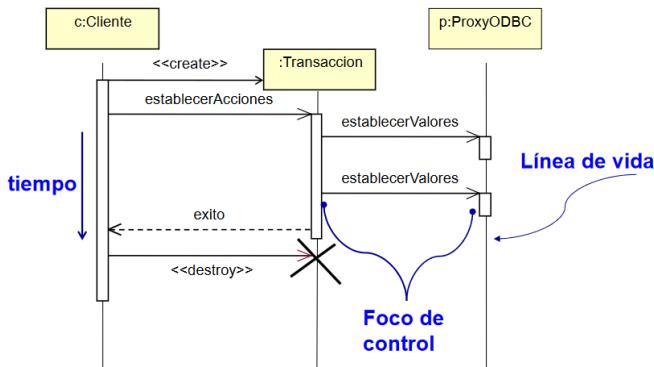
**“Sólo deberíamos usar diagramas para aquellas cosas sobre las que no se puede razonar sobre el código”**

- Modelo de clases del dominio: [Diagramas de clases](#)
  - Máquinas de estado: [Diagramas de estado](#)
  - [Diagramas de interacción](#) (principalmente para documentación)
  - [Diagramas de actividades](#) (workflows de procesos de negocio, herramientas BPMN) diagrama de clases
  - Se describen los **tipos de objetos** de un sistema y las **relaciones** que existen entre ellos.
    - Clases, Interfaces, Relaciones de dependencia, realización, generalización y asociación
  - Un [diagrama de clases](#) es una representación gráfica de un modelo estructural.
  - Diferentes usos: modelo **conceptual**, modelo de clases del **diseño** o de la **implementación**





## Diagrama de Secuencia



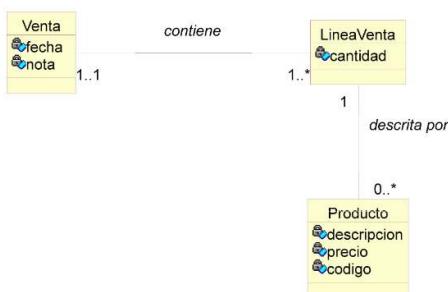
¿Cómo se asignan responsabilidades? “Asignar una responsabilidad a la clase que tiene la información necesaria para cumplimentarla”

### v Heurísticas relacionadas

- Distribuir responsabilidades de forma homogénea
- No crear clases “dios”
- v Beneficios
- Se conserva encapsulación: Bajo acoplamiento
- Alta Cohesión: clases más ligeras

## Ejemplo 1

¿Quién es el responsable de conocer el total de una venta?



### Ejemplo 1. ¿Escribimos esto?

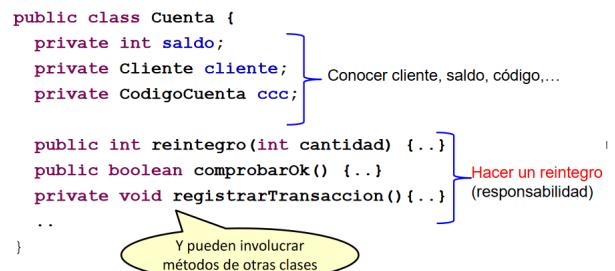
```

public double getTotal() {
    double total = 0.0;
    for (LineaVenta lv : lineasVenta) {
        double cant = lv.getCantidad();
        Producto pr = lv.getProducto();
        total = total + (cant*pr.getPrecio());
    }
    return total;
}

```

## 2 Patrones GRASP

- “Describen los principios básicos de asignación de responsabilidades a clases”.
- “Distribuir responsabilidades en la parte más difícil del diseño OO. Consume la mayor parte del tiempo”.
- Patrones GRASP:
  - Experto, Creador, Controlador**
  - Bajo Acoplamiento y Alta Cohesión
  - Polimorfismo y Proteger variación**
- Una **responsabilidad se implementa mediante uno o más métodos** y puede asignarse a varias clases según la granularidad (tamaño).
- “Contrato u obligación de una clase”. Dos categorías:
  - **Conocer** • datos encapsulados privados
  - existencia de objetos conectados • datos derivados o calculados
  - **Hacer** • Algo él mismo, como crear un objeto o realizar un cálculo • Iniciar una acción en otros objetos
  - Controlar y coordinar actividades en otros objetos
- **Diagramas de interacción** muestran elecciones en la asignación de responsabilidades.



## Ejemplo 1. Sin violar Experto

```

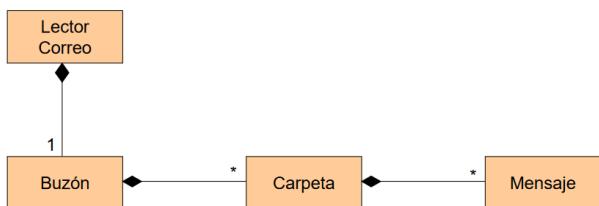
class Venta {
    public double getTotal() {
        double total = 0.0;
        for (LineaVenta lv : lineasVenta) {
            total = total + lv.getSubtotal();
        }
        return total;
    }
}

class LineaVenta {
    private int cantidad;
    private Producto producto;
    public double getSubtotal() {
        return cantidad*producto.getPrecio();
    }
}

```

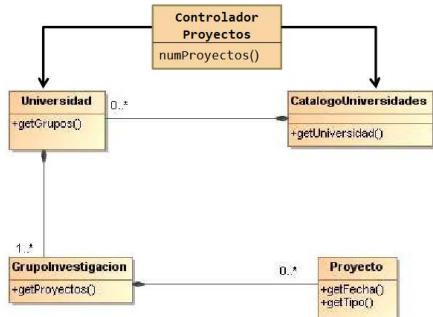
## Ejemplo 2: Lector Emails

¿Quién es el responsable de conocer todos los mensajes recibidos entre dos fechas en un buzón de emails?



## Ejemplo 3: Indicadores universidades

¿Quién es el responsable de conocer el **número de proyectos de cierto tipo** realizados en una universidad en un rango de fechas ?



## Ejemplo 3. Sin violar Experto

```

public class ControladorProyectos {
    public int numProyectos(Universidad uni, int tipo, Date f1, Date f2) {
        return uni.numProyectos(tipo, f1, f2);
    }
}

public class Universidad {
    private Collection<GrupoInvestigacion> grupos;
    public int numProyectos(int tipo, Date f1, Date f2) {
        int total = 0;
        for (Grupo grupo: grupos)
            total = total + grupo.numProyectos(tipo, f1, f2);
        return total;
    }
}
  
```

## Creador

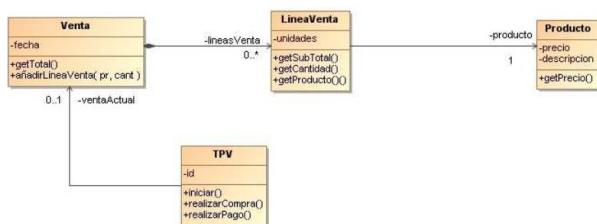
¿Quién es responsable de crear una nueva instancia de una cierta clase?

“Asignar a la clase B la responsabilidad de crear instancias de una clase A si:

- B es una agregación de objetos de A – B contiene objetos de A
- B registra instancias de A – B hace un uso específico de los objetos de A
- B proporciona los datos de inicialización necesarios para crear un objeto de A”
- Beneficios:
- Bajo acoplamiento

## Ejemplo 1

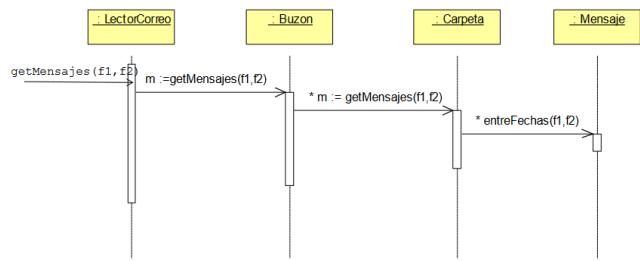
- Venta debería crear las instancias de LineaVenta



- ¿Quién debería crear instancias de Venta?

## Ejemplo 2. Sin violar Experto

¿Quién es el responsable de conocer todos los mensajes recibidos entre dos fechas?



## Ejemplo 3. Con violación del “experto”

```

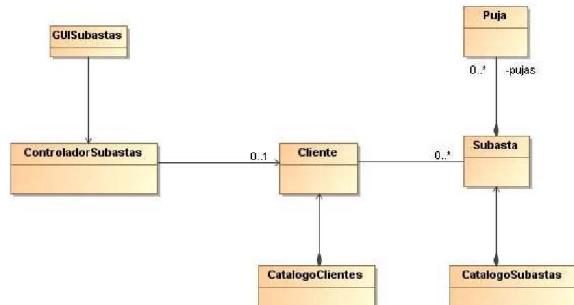
public class ControladorProyectos {
    public int numProyectos(Universidad uni, int tipo, Date f1, Date f2) {
        int totalProyectos = 0;
        Collection<Grupo> grupos = uni.getGrupos();
        for (Grupo grupo: grupos) {
            Collection<Proyecto> proyectos = grupo.getProyectos();
            for (Proyecto proyecto: proyectos) {
                fechaProyecto = proyecto.getFecha();
                if((fechaProyecto.before(f2))
                    &&(fechaProyecto.after(f1))
                    &&(tipo == proyecto.getTipo()))
                    totalProyectos= totalProyectos + 1;
            }
        }
        return totalProyectos;
    }
}

public class GrupoInvestigacion {
    private List<Proyecto> proyectos;
    public int numProyectos(int tipo, Date f1, Date f2) {
        int total = 0;
        for (Proyecto proyecto: proyectos){
            if ((proyecto.esTipo(tipo)) &&
                (proyecto.estaEntre(f1, f2)))
                total = total + 1;
        }
        return total;
    }
}

public class Proyecto {
    private Date fecha;
    private int tipo;
    public boolean esTipo(int tipo) {
        return (this.tipo == tipo);
    }
    public boolean estaEntre(Date f1, Date f2) {
        return ((fecha.after(f1)) && (fecha.before(f2)));
    }
}
  
```

## Ejemplo 2

- ¿Quién es responsable de crear una instancia de Puja ?





# Organiza tu futuro: estudia hoy para destacar mañana.

En Carpe Diem te esperan cursos adaptados a ti.  
Una forma fácil y real de avanzar profesionalmente.



## **Controlador**

**¿Quién se encarga de atender los eventos del sistema (una GUI, normalmente)?**

Asignar responsabilidad de manejar eventos externos a uno o más objetos **controlador** que puede representar, por ejemplo, el sistema o una funcionalidad concreto (caso de uso o historia de uso).

θ **Fachada** entre clases de la capa presentación (GUI) y capa del dominio.

θ Incluye, al menos, un método por cada operación atendida.

#### θ **Beneficios** de un controlador:

- **Separación modelo-vista**
  - Posibilidad de capturar información sobre el **estado de una sesión**
  - Cada **capa** reúne clases relacionadas con un mismo , por ejemplo: interfaz de usuario, dominio o almacenamiento.
  - **Utilidad**
    - Separación de aspectos favorece el cambio, la reutilización, y el desarrollo en paralelo.
    - Evita acoplamiento

## Separación Modelo-Vista

- Las clases del **modelo** (dominio) **NO deben conocer/acceder** a los objetos de la vista (presentación o interfaz).
  - Las clases del modelo encapsulan la información y el comportamiento relacionado con la lógica del negocio o de la aplicación.
  - Las clases de la **vista** son responsables de la entrada y salida, capturando y manejando los eventos, pero no delegan en un objeto controlador.
  - ¡OJO! Una clase de la vista puede invocar peticiones "get" sobre un objeto del dominio que le ha pasado el Controlador o un catálogo.

La **capa presentación** no debería tener ninguna responsabilidad de la lógica del negocio/aplicación. Sólo debería ser responsable de las tareas propias de la interfaz de usuario: mostrar ventanas y capturar eventos.

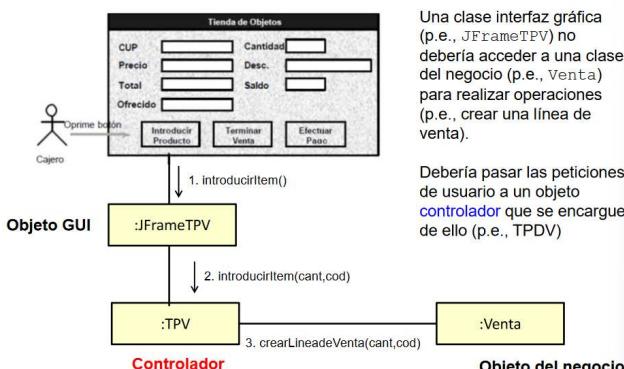
La **capa presentación** debería remitir las responsabilidades propias de la capa del dominio a esa capa, a través de un controlador.

- Favorece clases cohesivas
  - Permite desarrollar al mismo tiempo las clases de la vista y del dominio
  - Minimiza el impacto de los cambios en la interfaz sobre las clases del modelo.
  - Facilita conectar otras vistas a una capa del dominio existente (reutilización).
  - Permite varias vistas simultáneas sobre un mismo modelo.
  - Permite que la capa del modelo se ejecute como un proceso independiente a la capa de presentación.

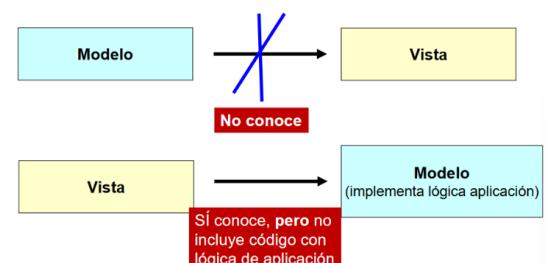
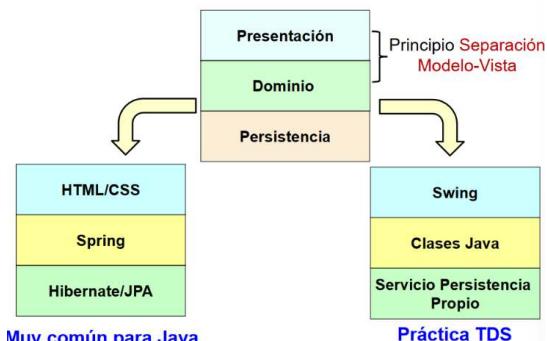
Las clases de la interfaz delegan en un controlador la realización de las tareas ligadas a cada evento del sistema que procede, normalmente de la GUI.

- El controlador es una “fachada” entre las clases de la vista y del dominio: Separar interfaz de usuario de la lógica del dominio o de la aplicación. No pertenece a la capa presentación sino del dominio.

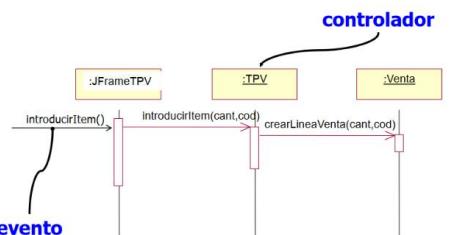
## Ejemplo de Separación modelo-vista



## Arquitectura de 3 capas: Muy extendida



## Ejemplo de Separación modelo-vista



Acoplamiento adecuado de la capa presentación con la capa del dominio

## Implementación en Swing

```

public class JFrameProcesarVenta extends JFrame {
    private TPV tpv; //objeto controlador
    public JFrameProcesarVenta (TPV _tpv) {
        this._tpv = _tpv;
    }
    private JButton botonAddItem;
    private JButton getBotonAddItem() {
        botonAddItem.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent ev) {
                ...
                tpv.introducirItem(cant, producto);
            });
        return botonAddItem;
    }
}

CatalogoClientes
    -clientes 1..*
    Cliente
        +getCuentas()
        -titular 1..*
        -cuentas
    Cuenta
        -bloqueada : boolean
        +isBloqueada()
        +setBloqueada()
        +getTransacciones()
    Transaccion
        -transacciones 0..*
        -fecha
    Bloqueo
        +fecha

```

```

public class ViewGestionCuentas {
    botonBloquear.addActionListener(ev ->
        Cliente cliente = CatalogoClientes.getInstancia().getCliente(campoDNI.getText());
        if (cliente != null)
            AppCuentas.getInstancia().bloquearCuentas(cliente);
        else JOptionPane.showMessageDialog(ViewGestionCuentas.this,
            "No existe cliente con ese dni");
    }
}

public class AppCuentas {
    //controlador de la aplicación
    public int bloquearCuentas(Cliente cliente) {
        cliente.bloquearCuentas();
    }
}

public class Cliente {
    private ArrayList<Cuenta> cuentas;
    ...
    public int bloquearCuentas() {
        for (Cuenta cuenta: cuentas)
            cuenta.bloquear();
    }
}

public class Cuenta {
    private boolean bloqueada;
    private ArrayList<Transaccion> transacciones;
    ...
    public void bloquear(){
        this.bloqueada = true;
        transacciones.add(new Bloqueo(LocalDate.now()));
    }
}

```

## Ejercicio. Modificar código para que no se viole ningún patrón GRASP

El siguiente código es parte de una aplicación bancaria que dispone de la funcionalidad de **bloquear todas las cuentas de un cliente** a petición de Hacienda.

- La clase ViewGestionCuentas implementa una ventana a través de la cual se pueden realizar diversas operaciones sobre las cuentas de un cliente.
- La clase Cliente tiene un atributo cuentas de tipo `ArrayList<Cuenta>` que registra todas las cuentas de ese cliente y el método `getCuentas()` retorna esa colección.
- La clase Cuenta tiene un atributo bloqueada de tipo booleano que determina si una cuenta está bloqueada y el método `setBloqueada()` que establece su valor.
- Al bloquear la cuenta se crea una instancia de la clase Bloqueo que es una subclase de la clase abstracta Transaccion. Esta clase tiene un atributo para registrar la fecha actual y sus subclases añaden atributos según el tipo de transacción.
- La clase Cuenta también tiene un atributo transacciones de tipo `ArrayList<Transaccion>` que registra todas las transacciones realizadas sobre una cuenta y el método `getTransacciones()` que retorna dicha colección.
- La clase CatalogoClientes es una clase que registra la colección de clientes.

```

public class ViewGestionCuentas extends JFrame {
    private Cliente cliente;
    private JTextField campoDNI;

    public ViewGestionCuentas() {
        ...
        JPanel panel = new JPanel();
        JLabel rotuloDNI = new JLabel("DNI Cliente ");
        CampoDNI = new JTextField("0");
        CampoDNI.setText("0");
        panel.add(rotuloDNI);
        panel.add(campoDNI);

        JButton botonBloquear = new JButton("Bloquear");
        botonBloquear.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent ev) {
                Cliente cliente = CatalogoClientes.getInstancia().getCliente(campoDNI.getText());
                if (cliente != null) {
                    AppCuentas.getInstancia().bloquearCuentas(cliente);
                } else JOptionPane.showMessageDialog(ViewGestionCuentas.this,
                    "No existe cliente con ese dni");
            }
        });
        ...
    }
}

```

### Bajo Acoplamiento

¿Cómo reducir las dependencias entre clases?

“Asignar responsabilidad de modo que se consiga un bajo acoplamiento”

✓ No considerarlo de forma independiente, sino junto a los patrones Experto y Alta Cohesióñ.

✓ Beneficios: Facilita i) reutilización, ii) comprensión de las clases y iii) mantenimiento

✓ Es un patrón evaluativo

✓ ¿Cuándo existe acoplamiento entre una clase A y otra B?

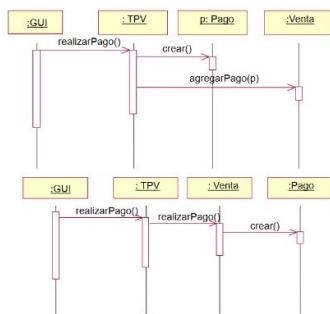
-A posee un atributo o campo de tipo B

-A tiene un método con un parámetro, una variable local o valor de retorno de tipo B.

-A es subclase directa o indirecta de B

-A implementa la interfaz B (Java)

### Ejemplo simple de evitar acoplamiento



**OFERTAS  
BLACK FRIDAY**

**msi®**

# **Esta oferta**

**es como una doble victoria  
tuya: disfrútala ahora porque  
no pasa dos veces.**

**Ver ofertas**



Tu viejo portátil ya dio lo que tenía que dar. Pásate a MSI: rápido, potente y sin dramas. Lo enciendes y estás listo para todo. Aprovecha las ofertas y despídete del modo “se cuelga cada dos por tres”.

## Alta Cohesión

¿Cuánto están de relacionadas las responsabilidades de una clase?

“ Asignar una responsabilidad de modo que la cohesión siga siendo alta”

“ Una clase debería representar una única abstracción bien definida” que tiene un conjunto de métodos muy relacionados funcionalmente.

v Baja Cohesión:

-Clases con responsabilidades que deberían haber delegado en otras.

-Son “clases dios” con gran cantidad de métodos y se pueden establecer grupos de ellos que no están fuertemente relacionados funcionalmente .

-Difíciles de comprender, reutilizar, mantener -> Beneficios de aplicar el patrón.

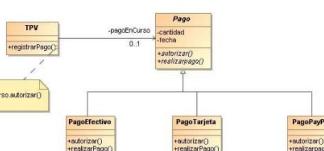
## Polimorfismo

### ¿Cómo manejar las alternativas basadas en el tipo?

Cuando existe comportamiento que depende de una variación de tipos, se debe asignar la responsabilidad de cada comportamiento al correspondiente tipo (**Curso de POO básico**).

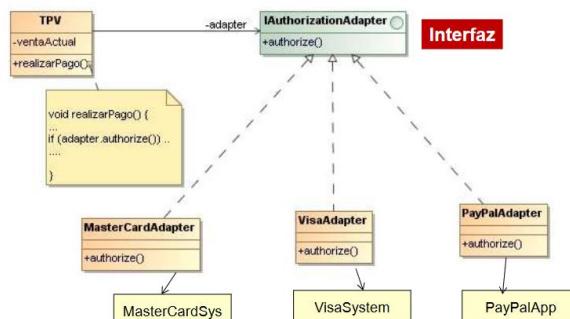
- No realizar un análisis de casos basado en el tipo de los objetos, sino aplicar una jerarquía de herencia.

```
switch (tipoPago) {
    case "Tarjeta":
        autorizarPagoTarjeta();
        break;
    case "Paypal":
        autorizarPagoPaypal();
        break;
    case "Transferencia":
        autorizarPagoTransferencia();
        break;
    default:
}
```



### ¿Cómo conseguir que una clase pueda interactuar con clases no previstas que ofrecen una funcionalidad que requiere?

Definir una **interfaz** que proporcione los métodos que permitan la interacción sin conocer clases concretas.



## Las interfaces son claves para la “pluggability”

- Se añaden fácilmente nuevas alternativas
- Los clientes no conocen las implementaciones concretas que usan.

## Ocultación de la información

### Minimizar la accesibilidad de las clases y sus elementos.

- No declarar atributos públicos, sino que sean privados y la clase ofrezca métodos para acceder y modificarlos.
- Ocultar detalles de implementación al código cliente.

```
public class Vehiculo {
    private double velocidad;
    public double getVelocidad() {
        return velocidad;
    }
    public void setVelocidad(double velocidad) {
        this.velocidad = velocidad;
    }
}
```

## Beneficios

- Es posible establecer restricciones sobre los valores de los atributos.
  - Controladas en métodos set
- Es posible cambiar la representación de los datos sin afectar al código cliente.
- Es posible ejecutar efectos laterales, por ejemplo notificar el cambio a otros objetos.

## Principio Abierto-Cerrado

### Favorece la extensibilidad

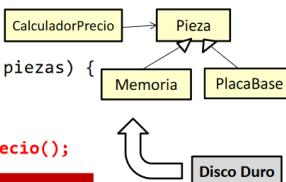
- Añadir nuevo comportamiento sin modificar el código existente.
- Los módulos deberían ser **cerrados para la modificación** (y listos para usarse) al tiempo que **abiertos para ser extendidos**.

### Conviene tener el mayor número posible de clases abiertas y cerradas al mismo tiempo:

- Uso de Herencia y Polimorfismo

```
public class CalculadorPrecio {
    public double precioTotal(Pieza[] piezas) {
        double total = 0.0;
        for (Pieza pieza: piezas)
            total = total + pieza.getPrecio();
        return total;
    }
}
```

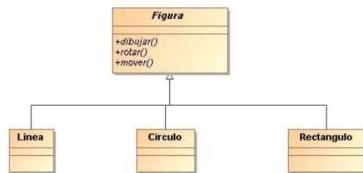
Satisface el principio abierto-cerrado



¿Y si se introduce una política de descuentos que depende del tipo de Pieza?

## Principio de Sustitución de Liskov (LSP)

- Es una consecuencia de la herencia y el polimorfismo.
- Siempre que una entidad tiene como tipo A, dinámicamente podrá referenciar a instancias de cualquier subclase de A.
- Un programador puede violarlo si una subclase no es consistente con la superclase.



```
void pintar (Collection<Figura> figuras) {
    for (Figura figura: figuras)
        figura.dibujar();
}
```

La clase CalculadorPrecio **NO satisface** el principio abierto-cerrado

```
public double precioTotal(Pieza[] piezas) {
    double total = 0.0;
    for (Pieza pieza: piezas) {
        if (pieza instanceof PlacaBase)
            total = total + (0.90 * pieza.getPrecio());
        else if (pieza instanceof Memoria)
            total = total + (0.75 * pieza.getPrecio());
        else
            total = total + pieza.getPrecio();
    }
    return total;
}
```

por el cálculo del precio final

por usar los nombres de estas clases

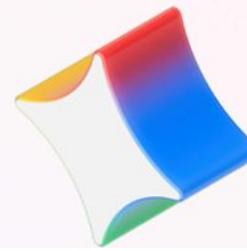
# Google Gemini: Plan Pro a 0€ durante 1 año.

Tu ventaja por ser estudiante.

Oferta válida hasta el 9 de diciembre de 2025

Consegue la oferta

Después 21,99€/mes



```
public class Pieza {..  
    private double precio;  
    public Pieza(double precio) {  
        this.precio = precio; }  
    public void setPrecio(double precio) {  
        this.precio = precio; }  
    public double getPrecio() {  
        return precio; }  
}
```

Las clases Pieza NO satisfacen el principio abierto-cerrado

```
public class PlacaBase extends Pieza {..  
    public double getPrecio() {  
        // return (0.85 * precio);  
        return (0.90 * precio);  
    }
```

El descuento es implementado en el método getPrecio()  
¡Hay que cambiar estas clases si cambia el descuento!

## No hables con extraños

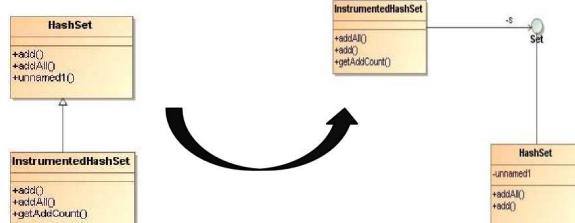
- No enviar mensajes sobre objetos indirectos, sino sobre la instancia actual (**self**), **parámetros** de métodos, **atributos** de la instancia actual y **elementos de colecciones** de la instancia actual. Evitar recorrer largos caminos en la estructura de objetos: "diseño frágil"



```
class TPV {  
    private Venta venta;  
    public void metodoFragil () {  
        Dinero cantidad =  
            venta.getPago().getCantidadEntregada();  
        ...  
    }  
}
```

Dinero cantidad =  
venta.getCantidadEntregadaEnPago()

## Favorecer composición frente a herencia ("Effective Java", Joshua Bloch, item 16)



Relación fija  
Reutilización "caja blanca"

Relación variable  
Reutilización "caja negra"

¡¡La herencia viola la encapsulación !!



Una **subclase depende** de los detalles de implementación de su **superclase** para funcionar correctamente:

- Más difícil de comprender
- Si la implementación de la superclase cambia, podría afectar al comportamiento de la subclase aunque no se haya tocado el código de esta.



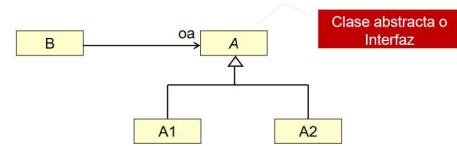
```
public double precioTotal(Pieza[] piezas){  
    double total = 0.0;  
    for (Pieza pieza: piezas) {  
        total = total + pieza.getPrecio();  
    }  
    return total;
```

```
abstract class Pieza {..  
    double precio;  
    PoliticaPrecio politica;  
    public double getPrecio() {  
        return politica.getPrecio(precio);  
    }  
}
```

**CalculadorPrecio y Pieza y sus subclases satisfacen el Principio Abierto-Cerrado**

## • "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.



Una instancia de B puede estar ligada a una instancia de cualquier subclase de A.

## Ejemplo ("Effective Java", Joshua Bloch)

```
public class InstrumentedSet<E> implements Set<E>{  
    private final Set<E> s;  
    private int addCount = 0;  
    public InstrumentedSet(Set s) { this.s = s; }  
    public boolean add(Object o) {  
        addCount++;  
        return s.add(o);  
    }  
    public boolean addAll(Collection c) {  
        addCount += c.size();  
        return s.addAll(c);  
    }  
    public int getAddCount() {  
        return addCount;  
    }  
    // resto de métodos de Set<E> que delegan en el método de igual  
    // nombre en Set<E>  
    public void clear() { s.clear(); }  
    public boolean contains(Object o) { return s.contains(o); }  
    ...  
}
```

```

classDiagram
    class Set {
        -Set<E>
    }
    class HashSet {
        +add()
        +addAll()
        +remove()
        +contains()
    }
    class TreeSet {
        +add()
        +addAll()
        +getAddCount()
    }
    class InstrumentedSet {
        +add()
        +addAll()
        +getAddCount()
    }

    Set <|--> HashSet : -Set<E>
    Set <|--> TreeSet : -Set<E>
    InstrumentedSet --> Set : +add()
    InstrumentedSet --> HashSet : +add()
    InstrumentedSet --> TreeSet : +add()
  
```

## Etapas de un proceso software

- Estudio de viabilidad del proyecto
- Análisis de Requisitos con el cliente
- Diseño de la solución
- Implementación
- Testing
- Despliegue
- Mantenimiento

Procesos actuales están basados en historias de usuario y son iterativos e incrementales.

Adopción de los métodos ágiles como Scrum!

## Diseño Dirigido por el Dominio (DDD)

- Un modelo del dominio dirige el desarrollo.
- Analista representa el conocimiento del dominio en colaboración con el experto del dominio.
- Representación con gráficos y/o texto
- Crean un lenguaje común compartido
- El modelo es orientado a la creación de código
- Entidades del dominio son clases
- Relaciones entre entidades son asociaciones
- Entidad
- Objetos que tienen una identidad que se mantiene a lo largo de la vida del sistema sin importar que cambios tengan lugar. Ejemplos: Usuario, Cuenta, Mensaje, Viaje, y Hospital.
- Value Object
- Objetos que no tienen una identidad. Pueden ser compartidos y deberían ser inmutables. Ejemplos: Fecha, Punto, Complejo, Ruta,..
- Servicios
- Objetos que denotan “algún proceso o transformación en el dominio que no es una responsabilidad natural de una entidad o value object”, y no parece adecuado asignarlo a una de ellas. No tienen estado. Ejemplos: Calculador de descuentos, Generador de números de factura, generador de código de pedido, ...
- Módulo
- Organizar el modelo en varios módulos para dominar la complejidad. Favorece la cohesión. Usar una interfaz (fachada) para reducir el acoplamiento --> Controlador
  
- Agregados
- Grupo de objetos relacionados que son considerados como una unidad.
- El acceso a través de un objeto raíz que es de tipo entidad.
- El objeto raíz controla que los objetos que agrega satisfacen el invariante tras un cambio.
- Se aplica patrón Creador: el objeto raíz crea a los objetos que agrega
  
- Factorías
- Clases encargadas de la construcción de objetos. Usar en vez de constructores para una creación sin especificar el tipo concreto o para crear un objeto complejo, por ejemplo agregados o lógica de creación compleja
- Patrones de diseño GoF: Factoría Abstracta, Método Factoría y Builder.
- Repositorios (Catálogos)
- Encapsulan la lógica necesaria para recuperar objetos de un tipo de Entidad (por ejemplo, Cliente, Vuelo,...)
- Lugar de almacenamiento para objetos que es accesible globalmente: patrón Singleton.
- Métodos para añadir/eliminar y para buscar objetos a través de un identificador, uno o más atributos, o un criterio de selección.
- No son responsables de crear objetos.

En nuestro caso, los repositorios tendrán todos los objetos almacenados de la entidad, se cargarán al arrancar la aplicación. Se podrían cargar bajo demanda.

## DDD. Refactoring

- Cambios al código o modelo que mejoran la calidad del código o descubren nuevos conceptos o relaciones en el modelo del dominio, sin cambiar el comportamiento.
- Dos tipos:
- Técnico. Mejora la calidad del código aplicando patrones de refactorings. Normalmente, el término se refiere a este tipo.
- Dominio. Mejora del modelo del dominio para descubrir nuevos conceptos.
- Ejemplos: Extraer método, Mover método, Eliminar switch basado en tipo, Dividir una clase, ...

```
public class Bookshelf {
    private int capacity = 20;
    private Collection<Book> content;
    public void add(Book book) {
        if(content.size() + 1 <= capacity) {
            content.add(book);
        } else {
            throw new IllegalStateException(
                "The bookshelf has reached its limit.");
        }
    }
}

public class Bookshelf {
    private int capacity = 20;
    private Collection<Book> content;
    public void add(Book book) {
        if(isSpaceAvailable()) {
            content.add(book);
        } else {
            throw new IllegalStateException(
                "The bookshelf has reached its
                limit.");
        }
    }
    private boolean isSpaceAvailable() {
        return content.size() < capacity;
    }
}
```



Extraer una restricción o regla de negocio en un método para mejorar legibilidad y reutilización

## 1 Clases anidadas

- Algunos lenguajes OO, como C++, C# y Java, permiten anidar la definición de una clase dentro de otra clase cuando sólo es necesaria en ese ámbito.
- En Java se incluyó, además, el mecanismo de clases internas para parametrizar clases con funciones.
- En el caso de Java hay 4 mecanismos de anidamiento de clases:
  - Clases anidadas estáticas
  - Clases internas
  - Clases miembro, clases anónimas, y clases locales.

## Clases anidadas en Java

### Clases anidadas **estáticas**

- Reflejan una **relación sintáctica entre dos clases**
- Una clase se declara en el ámbito de otra clase lógicamente relacionada.

```
public class Externa {..
    static class AnidadaEstatica {..}
}
```

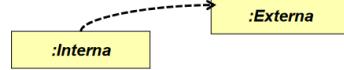
- Un clase anidada estática tiene **visibilidad** sobre los miembros de la clase que la engloba (incluso los privados), y al revés.

```
public class Externa {..
    private int atrib;
    public static class Anidada {
        void m1(Externa pExt) {
            pExt.atrib = 10;
        }
    }
}
```

## Clases internas en Java

- Reflejan una **relación entre instancias de dos clases**.
- Una instancia de una clase interna debe existir en el contexto de una instancia de su clase externa** y tiene acceso directo a sus atributos y métodos de instancia.
- Tres tipos: **clases miembro, anónimas y locales**.
- Los **objetos de la clase interna** en general son **creados en algún método de instancia de la clase externa**.
- Puede implementar sus propias interfaces.
- Útiles para parametrizar una clase por una acción a invocar:** (callback o patrón Command)

```
class Externa {..
    class Interna implements Interfaz {...}
}
```



## Clases internas: Ejemplo framework eventos



(Tomado de "Thinking in Java",  
Bruce Eckel, 1<sup>a</sup> Edición)

Para crear una aplicación concreta es necesario crear una **subclase** de Controller y **subclases** de Event para cada tipo de evento.  
Estas subclases **necesitan actualizar estado del controlador creado**.

```
// implementación de pilas antes de incluir clases genéricas
public class Stack {
    private Node head;
```

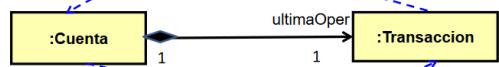
```
private static class Node {
    private Object value;
    private Node next;
    public Node(Object v, Node s) {
        value = v; next = s;
    }
}
```

```
public void push (Object obj) {
    head = new Node (obj,head);
}
public Object pop () {
    Object first = head.value;
    head = head.next;
    return first;
}
}
```

```
class CuentaBancaria{..
    private long codigo;
    private long saldo;
    private Transaccion ultimaOper;
```

(Tomado de "The Java Programming Language", K. Arnold et al., 3<sup>a</sup> Edición)

```
public class Transaccion {
    private String descripcion;
    private long cantidad;
    ...
    public String toString(){
        return codigo + ":" + descripcion+ " " +cantidad;
    }
    public void reintegro(long cantidad){
        saldo = saldo - cantidad;
        ultimaOper= this.new Transaccion("reintegro",cantidad);
    }
}
```



```
abstract public class Event {
    private long evtTime;
    public Event (long eventTime) {
        this.evtTime = eventTime;
    }
    public boolean ready() {
        return System.currentTimeMillis() >= evtTime;
    }
    abstract public void action();
    abstract public String description();
}
class EventSet {
    private Event[] events = new Event[100];
    private int index = 0;
    private int next = 0;
    public void add() {...}
    public Event getNext () {...}
    public void removeCurrent() {...}
}
```



# LA NUESTRA DURA MÁS



```

abstract public class Controller {
    private EventSet es = new EventSet();
    public void addEvent(Event e) {
        es.add(e);
    }
    public void run() {
        Event ev;
        while (ev = es.getNext() != null) {
            if (ev.ready()) {
                ev.action();
                System.out.println(ev.description());
                ev.removeCurrent();
            }
        }
    }
}

public class LinkedList<E>
    extends AbstractSequentialList<E>
    implements List<E>, Deque<E>, Cloneable, java.io.Serializable {
    transient int size = 0; transient Node<E> first; transient Node<E> last;

    private static class Node<E> {
        E item;
        Node<E> next;
        Node<E> prev;
        Node<E> prev, E element, Node<E> next) {
            this.item = element; this.next = next; this.prev = prev;
        }
    }

    private class ListItr implements ListIterator<E> {
        private Node<E> lastReturned;
        private Node<E> next;
        private int nextIndex;
        ListItr(int index) {
            next = (index == size) ? null : node(index);
            nextIndex = index;
        }
        public boolean hasNext() ...
        return nextIndex < size;
    }
}

```

## Clases anónimas. Ejemplo “manejadores de eventos” en Swing

- Una clase interna anónima se crea “al vuelo” en el punto en que se necesita un objeto de una clase que implemente una interfaz o clase abstracta.
- Se usa normalmente como un **object function**
- Se crea si sólo se necesita en un punto del programa y debería ser pequeña, unas 10 líneas de código.

```

public class ViewContador extends JFrame {
    private Contador contador;
    private JTextField tf;
    public initialize()...
    JButton incrBoton = new JButton("Incrementar");
    incrBoton.addActionListener(new ActionListener() {
        public void actionPerformed(ActionEvent ev) {
            contador.incrementar();
            tf.setText(contador.getValor() + "");
        }
    });
}

```

## Closure (Function object)

- ❖ En programación funcional, función anónima que puede ser tratada como un objeto (registrado en una variable y pasado de un contexto a otro).
- ❖ Puede ser invocada en un contexto diferente al que se define sin conocer su nombre.
- ❖ Lleva ligada el entorno (variables) en el que se define y puede acceder a ellas desde otro contexto.
- ❖ Lenguajes OO tipados dinámicamente como Smalltalk, Python y Ruby proporcionan closures (bloques de código), también disponibles en JavaScript y otros lenguajes OO.
- ❖ Disponibles en Java 8.

```

public class ControladorInvernadero
    extends Controller {
    private boolean luz = false;
    private boolean agua = false;

    private class LuzOn extends Event {
        public LuzOn(long eventTime) {super(eventTime);}
        public void action() { luz = true; }
        public void description() {
            return "luz encendida"; }
    }

    private class LuzOff extends Evento {...}
    private class AguaOn extends Evento {...}
    private class AguaOff extends Evento {...}
    ...

    class CalculatorPanel extends JPanel {
        public CalculatorPanel() {
            result = 0;
            lastCommand = "=";
            start = true;
            display = new JLabel("0");
            ActionListener insert = new InsertAction();
            ...
            JButton button = new JButton("1");
            button.addActionListener(insert);
            ...
            private class InsertAction implements ActionListener {
                public void actionPerformed(ActionEvent event) {
                    String input = event.getActionCommand();
                    if (start) {
                        display.setText("");
                        start = false;
                    }
                    display.setText(display.getText() + input);
                }
            }
        }
    }
}

```

## Clases anónimas. Restricción acceso variables

- Sólo pueden usar variables del método en que se ha definido si las variables han sido declaradas como **finales**.

```

final String nombre = getNombre();
boton1.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent event) {
        System.out.println("hola " + nombre);
    }
});

```

**Clases anónimas** de Java con un método proporcionan una forma **limitada** de closure.

- La función **es un método con nombre no anónima** y puede usar variables del método en que se definen si son declaradas **final**.

```

final String nombre = getNombre();
boton1.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent event) {
        System.out.println("hola " + nombre);
    }
});

```

**Expresiones lambda** de Java 8 son closures con la restricción de que a las variables del método en que se definen sólo se les puede asignar valor una vez (**effectively final**)

```

String nombre = getNombre();
boton1.addActionListener(event ->
    System.out.println("hola " + nombre));

```

## Java 8. Motivación

- Java no proporcionaba bloques o closures a diferencia de otros lenguajes OO.
- Proporcionar construcciones para facilitar un estilo de programación funcional con la POO.
- Proporcionar mecanismos para escribir código que tome ventaja de CPU multicore para procesar colecciones.

### Expresiones Lambda y Streams

```
public class Usuario {...  
    private String nombre;  
    private Date nacimiento;  
  
    public String getNombre() {  
        return nombre;  
    }  
    public int getEdad() {  
        // Propiedad calculada ...  
    }  
}
```

```
public class ComparadorUsuariosPorEdad  
    implements Comparator<Usuario> {  
    @Override  
    public int compare(Usuario o1, Usuario o2) {  
        return o1.getEdad() - o2.getEdad();  
    }  
}
```

```
List<Usuario> usuarios;  
...  
Collections.sort(usuarios, new ComparadorUsuariosPorEdad());  
  
Collections.sort(usuarios,  
    (Usuario u1, Usuario u2) -> {  
        return u2.getNombre().compareTo(u1.getNombre());  
    }  
);
```

- Se puede simplificar la expresión dado que el compilador realiza **inferencia de tipos** y el bloque de código es una única sentencia.

```
Collections.sort(usuarios,  
    (u1, u2) -> u2.getNombre().compareTo(u1.getNombre()));
```

- Las expresiones lambda tienen acceso a las variables (locales y parámetros) y campos del contexto del código.

Una **interfaz funcional** es una interfaz con un **único método** que es utilizada como tipo de una expresión lambda (no necesario en un lenguaje tipado dinámicamente).

Una expresión lambda es un bloque de código (función sin nombre) compatible con una **interfaz funcional**.

```
@FunctionalInterface  
interface Comparator<T> {  
    int compare(T o1, T o2);  
}
```

- La declaración **@FunctionalInterface** es opcional.

- La clase **Collections** ofrece el método **static sort** para ordenar listas de acuerdo al **orden establecido por un objeto comparador**:

```
public static <T> void sort(List<T> list, Comparator<T> c)
```

- Necesidad de crear una subclase de **Comparator<T>** para implementar el criterio de orden.

```
interface Comparator<T> {  
    int compare(T o1, T o2);  
}
```

- Ejercicio: Posibilidad de usar introspección para crear una clase genérica que implemente **Comparator<T>**

- Evitan proliferación de clases pequeñas.
- Alternativa a no ofrecer expresiones lambda.

```
Collections.sort(usuarios,  
    new Comparator<Usuario>() {  
        @Override  
        public int compare(Usuario o1, Usuario o2) {  
            return o1.getNombre().compareTo(o2.getNombre());  
        }  
    }  
);
```

## Formas de expresar una lambda

```
Runnable process = () -> System.out.println("Hello World");  
    Sin argumentos      Bloque de código de una línea
```

```
ActionListener al =  
    event -> System.out.println("button clicked");  
    1 arg sin tipo      Bloque de código de una línea
```

```
Runnable process = () -> {  
    System.out.print("Hello");  
    System.out.println(" World");  
};  
    Sin argumentos      Bloque de código de 2 líneas
```

```
BinaryOperator<Long> add = (x, y) -> x + y;  
BinaryOperator<Long> addExplicit = (Long x, Long y) -> x + y;  
    2 argumentos con tipo      Bloque de 1 línea
```

## Interfaz funcional ActionListener

Todas las interfaces con un único método son ahora **interfaces funcionales**

```
public interface ActionListener extends EventListener {  
    // Invoked when an action occurs.  
    public void actionPerformed(ActionEvent e);  
}
```

```
JLabel resultado = new JLabel();  
  
 JButton botonSaludo = new JButton("Poner título");  
 botonSaludo.addActionListener(ev ->  
     resultado.setText("Hola alumnos")));
```

Antes con clases anónimas, ahora con lambdas

```

package java.util.function;

public interface Predicate<T> {
    // comprueba la condición sobre el argumento dado
    boolean test(T t);
}

public interface Consumer<T> {
    // realiza la operación sobre el argumento dado
    void accept(T t);
}

public interface Function<T,R> {
    // retorna el valor obtenido al aplicar la función sobre el argumento dado
    R apply(T t);
}

public interface Supplier<T> {
    // retorna un resultado de tipo T, una alternativa al patrón método factoría
    T get();
}

```

- Representa un suministrador de resultados (objetos)

```

interface Supplier<T> {
    T get();
}

```

- Representa una función que acepta un argumento y devuelve un resultado.

```

interface Function<T, R> {
    R apply(T obj);
}

```

## Implementar un filtro condición

```

public class FiltroCuentas{
    public static List<Cuenta> filtrarCuentas1(List<Cuenta> lista, ITest filtro){
        //cualquier filtro para listas de Cuentas
        ArrayList<Cuenta> result = new ArrayList<Cuenta>();
        for(Cuenta cu:lista)
            if (filtro.test(cu)) result.add(cu);
        return result;
    }

    public static void main(String[] args) {
        TestSaldoCuenta filtro = new TestSaldoCuenta(1000);
        for(Cuenta cu:FiltroCuentas.filtrarCuentas1(lista, filtro))
            System.out.println(cu.getCCC());
    }
}

public interface ITest<T> {
    boolean test (T o);
}

public class TestSaldoCuenta implements ITest<Cuenta> {
    private int cantidad;
    TestSaldoCuenta(int cant) {
        this.cantidad = cant;
    }
    public boolean test(Cuenta c) {
        return c.getSaldo() > cantidad;
    }
}

```

Filtros son clases que implementan una interfaz

## Implementar un filtro condición con Predicate<T>

```

public class FiltroCuentas{
    public static List<Cuenta> filtrarCuentas3(
        List<Cuenta> lista, Predicate<Cuenta> filtro){
        ArrayList<Cuenta> result = new ArrayList<Cuenta>();
        for(Cuenta cu:lista)
            if (filtro.test(cu))
                result.add(cu);
        return result;
    }

    public static void main(String[] args) {
        for(Cuenta cu:FiltroCuentas.filtrarCuentas3(
            lista, (Cuenta c) -> c.getSaldo()>1000))
            System.out.println(cu.getCCC());
    }
}

```

## Interfaces funcionales de propósito general en java.util.function

- Representa una función que comprueba si un objeto satisface una condición.

```

interface Predicate<T> {
    boolean test(T obj);
}

```

- Representa operaciones que dado un único argumento realizan una acción y no retornan nada.

```

interface Consumer<T> {
    void accept(T obj);
}

```

## Implementar un filtro (saldo>cantidad)

```

public class FiltroCuentas{
    public static List<Cuenta> filtrarCuentasPorSaldo(List<Cuenta> lista,int cant){
        ArrayList<Cuenta> result = new ArrayList<Cuenta>();
        //filtrar cuentas con saldo mayor que una cantidad
        for(Cuenta cu:lista)
            if (cu.getSaldo()>cant)
                result.add(cu);
        return result;
    }

    public static void main(String[] args) {
        Cuenta c1 = new Cuenta (10000, "123", "JGM");
        Cuenta c2 = new Cuenta (200, "345", "IGM");
        Cuenta c3 = new Cuenta (3000, "567", "EGM");
        Cuenta c4 = new Cuenta (300, "890", "CMT");
        Cuenta c5 = new Cuenta (2000, "111", "JLGM");

        ArrayList<Cuenta> lista = new ArrayList<Cuenta>();
        lista.add(c1); lista.add(c2); lista.add(c3); lista.add(c4); lista.add(c5);

        for(Cuenta cu:FiltroCuentas.filtrarCuentasPorSaldo(lista,1000))
            System.out.println(cu.getCCC());
    }
}

```

## Implementar un filtro condición con interfaz y clases anónimas

```

public class FiltroCuentas{
    public static List<Cuenta> filtrarCuentas2( List<cuenta> lista, ITest filtro){
        ArrayList<Cuenta> result = new ArrayList<Cuenta>();
        //cualquier filtro para listas de Cuentas
        for(Cuenta cu:lista)
            if (filtro.test(cu))
                result.add(cu);
        return result;
    }

    public static void main(String[] args) {
        for(Cuenta cu:FiltroCuentas.filtrarCuentas2(
            lista,
            new ITest<Cuenta>(){
                @Override
                public Boolean test(Cuenta c) {
                    return c.getSaldo()>1000;
                }
            }))
            System.out.println(cu.getCCC());
    }
}

```

## Realizar una acción sobre objetos que cumplen una condición

```

public class IteradorCuentas{

    public static void hacerSi(List<Cuenta> lista,
        Predicate<Cuenta> filtro,
        Consumer<Cuenta> action) {
        for(Cuenta cu:lista)
            if (filtro.test(cu))
                action.accept(cu);
    }

    public static void main(String[] args) {
        ArrayList<Cuenta> result = new ArrayList<Cuenta>();
        IteradorCuentas.hacerSi(
            lista,
            (Cuenta c) -> c.getSaldo()>1000,
            (Cuenta c) -> result.add(c));
        for (Cuenta cu:result)
            System.out.println(cu.getCCC());
    }
}

```



# **SUDADERAS PARA GRUPOS PERSONALIZADAS**



```
package tds.jesus;

import java.util.ArrayList;
import java.util.function.Function;

public class PruebaExplamda {

    public static void main (String[] args){
        Cuenta c1 = new Cuenta (100, "123", "JGM");
        Cuenta c2 = new Cuenta (200, "345", "IGM");
        Cuenta c3 = new Cuenta (300, "567", "EGM");

        // asignar expresión lambda a variable
        Function<Cuenta,Integer> exp = (Cuenta c) -> c.getSaldo();
        // invocar expresión lambda
        System.out.println(exp.apply(c3));
    }
}

public class Rectangle implements Graphic {
    private int width;
    private int height;
    private String name;

    public Rectangle(String name, int width, int height) {
        super();
        this.name = name;
        this.width = width;
        this.height = height;
    }
    @Override
    public String getName() {
        return name;
    }

    @Override
    public void drag() {...}
}

public class App {

    public static void main(String[] args) {
        Supplier<Graphic> g = () -> new Rectangle("Caja 1",3,2);
        //se crea e inicializa un GraphicTool para rectángulos
        GraphicTool gt = new GraphicTool();
        gt.setSupplier(g);

        gt.printGraphicName();
    }
}
```

## *Ejemplos de Supplier con funciones*

```
public void supplier1(){
    Supplier<Double> doubleSupplier1 = () -> Math.random();
    DoubleSupplier doubleSupplier2 = Math::random;

    System.out.println(doubleSupplier1.get());
    System.out.println(doubleSupplier2.getAsDouble());
}

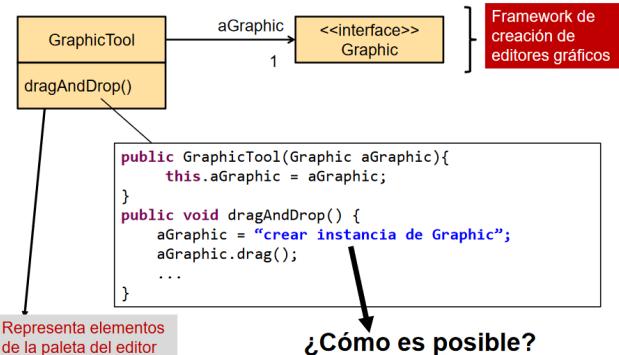
public void supplier2(){
    LocalDateTime ldt = LocalDateTime.now();
    System.out.println(ldt); // Muestra hora actual H1

    Supplier<LocalDateTime> dateSupplier = LocalDateTime::now;

    Thread.sleep(5000);
    System.out.println(ldt); // Muestra hora H1
    System.out.println(dateSupplier.get()); // Muestra hora actual H2

    Thread.sleep(5000);
    System.out.println(ldt); // Muestra hora H1
    System.out.println(dateSupplier.get()); // Muestra hora actual H3
```

## *Método Factoría o Prototype con Supplier* (revisado en tema “Patrones de diseño”)



```
import java.util.function.Supplier;
public class GraphicTool {
    private Supplier<Graphic> supplier;

    public void setSupplier(Supplier<Graphic> supplier) {
        this.supplier = supplier;
    }
    public void dragAndDrop() {..
        supplier.get().drag();
    }
    public void printGraphicName(){
        System.out.println(supplier.get().getName());
    }
}
public interface Graphic {
    public String getName();
}
```

## Referencias a métodos

- Forma abreviada de expresar expresiones lambda que sólo invocan un método sobre el argumento.

{artista -> artista.getNombre();}  Artista::getNombre

- También con lambdas que crean un objeto.

```
// Constructor no-arg //constructor (String name)
{() -> new Artista();} Function(String, Artista) f:
    _____
        (name)-> new Artista(name);
```

1

```
Function(String, Artist) f:  
    Artista::new
```

## Streams

- Un stream (`java.util.Stream`) es una secuencia de elementos que soporta operaciones secuenciales y paralelas.
- Las colecciones incluyen dos métodos para obtener streams (`stream` y `parallelStream`) y la clase Stream ofrece métodos para iterar sobre su elementos (iteradores internos)

Ejemplo: contar las cuentas con un saldo positivo en una Colección cuentas de objetos Cuenta.

```
long contador = cuentas.stream()
    .filter(cu -> cu.getSaldo() > 0)
    .count();
```

## Streams: Ejemplo 1

Obtener la lista de identificadores de transacciones que son de cierto tipo (p.e., `COMPRADIVISAS`) ordenadas por el valor de la transacción de mayor a menor.

```
//filtro
List<Transaction> compradivisas = new ArrayList<>();
for(Transaction t: transactions){
    if(t.getType() == Transaction.COMPRADIVISAS)
        compradivisas.add(t);
}
//ordeno
Collections.sort(compradivisas, new Comparator(){
    public int compare(Transaction t1, Transaction t2){
        return t2.getValue().compareTo(t1.getValue());
    }
});
//construyo nueva lista
List<Integer> transactionIds = new ArrayList<>();
for(Transaction t: compradivisas){
    transactionIds.add(t.getId());
}
```

Contar elementos que satisfacen una condición

**Java 7**

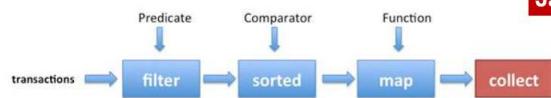
**Select / Filter**

**Sort**

**Map**

**Java 8**

Obtener lista de identificadores de transacciones de cierto tipo ordenadas por el valor de la transacción de mayor a menor.



```
List<Integer> transactionIds = transactions.stream()
    .filter(t -> t.getType() == Transaction.COMPRADIVISAS)
    .sorted(comparing(Transaction::getValue).reversed())
    .map(Transaction::getId)
    .collect(toList());
```

```
static import Comparator.*;
static import Collectors.*;
```

## Operaciones más comunes sobre streams

**Java <8**

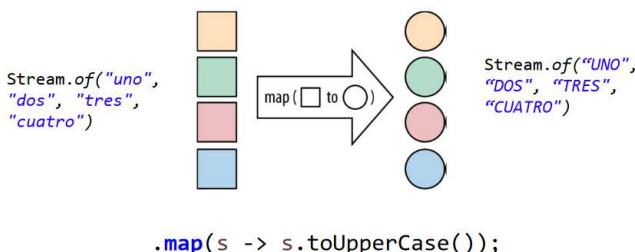
```
List<Cuenta> cuentas;
int num = 0;
for (Cuenta cu: cuentas)
    if (cu.getSaldo()>1000)
        num = num + 1;
```

**Java 8**

```
List<Cuenta> cuentas;
lista.stream()
    .filter(cu-> cu.getSaldo()> 1000)
    .count();
```

### Método `map`:

- Aplica una función sobre cada elemento de un stream y retorna otro con los elementos que resultan de aplicar la función sobre cada elemento de entrada (mismo número de elementos).
- Acepta como parámetro una *función* (`Function<T, R>`)

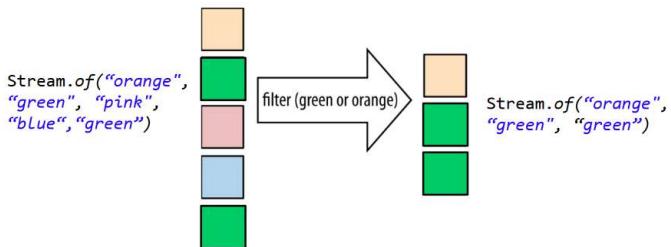


**import static java.util.stream.Collectors.\*;**

```
..
public static void main(String[] args) {
    // obtener una lista de los elementos de un stream
    List<String> collected1 = Stream.of("uno", "dos", "tres", "cuatro")
        .collect(Collectors.toList());
    // retorna una secuencia que es resultado de aplicar una acción sobre
    // cada elemento de la secuencia original
    List<String> collected2 = collected1.stream()
        .map(s -> s.toUpperCase()).collect(toList());
    // retorna la secuencia de elementos que cumplen una condición
    List<String> collected3 = collected1.stream()
        .filter(s ->s.endsWith("s")).collect(toList());
    // ejecuta una acción sobre cada elemento de la secuencia
    collected1.stream()
        .forEach(s -> System.out.println(s));
    // ordena según orden natural o recibe como argumento un Comparator
    collected1.stream()
```

### Método `filter`:

- Retorna un stream con los elementos del stream original que cumplen una condición.
- Acepta como parámetro un *predicado* (`Predicate<T>`)



### Método `forEach`:

- **Operación terminal** que aplica una acción sobre cada elemento.
- Acepta como parámetro un *consumidor* (`Consumer<T>`)

### Métodos `anyMatch`, `noMatch` y `allMatch`:

- **Operaciones terminales** que retornan un booleano indicando si se cumple un predicado en alguno (any), ninguno (no) o todos (all) los elementos.
- Acepta como parámetro un *predicado* (`Predicate<T>`)

### Método `sorted`:

- Ordena el stream según un criterio de ordenación.
- Acepta como parámetro un *comparador* (`Comparator<T>`).
- **Retorna un stream con la secuencia ordenada.**
- Tiene una versión sobrecargada sin parámetros que aplica el orden natural de los elementos.

### Método `count`:

- **Operación terminal** que retorna el número de objetos del stream resultado del procesamiento del stream fuente.

### Método `collect`:

- **Operación terminal** que transforma el resultado del procesamiento del stream en una colección.
- Acepta como parámetros métodos que transforman streams en colecciones, ejemplos: `Collectors.toList()`, `Collectors.toSet()`, `Collectors.toCollection()`, ...

Pueden ser eager or lazy, según generen o no, respectivamente, un valor a partir del stream manejado. `filter()` and `map()` son métodos **lazy** mientras que son métodos **eager** aquellos que retornan un valor como `count()` y `collect()`.

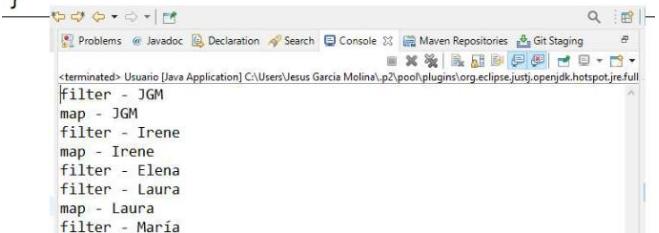
(leer <https://www.amitph.com/java-8-streams-laziness-and-performance/>)

```
usuarios.stream()
    .filter(u -> {
        System.out.println(u.getNombre());
        return u.getEdad() > 18
    });
}
```



No se muestran los nombres, **sí** al añadir `count()`

```
List<String> ids = usuarios.stream()
    .filter(s -> {
        System.out.println("filter - " + s);
        return s.getEdad() > 20;
    })
    .map(s -> {
        System.out.println("map - " + s);
        return s.getNombre();
    })
    .collect(Collectors.toList());
}
```



## Collectors

- Retorna implementaciones de la interfaz Collector que realizan **operaciones de reducción** útiles:

```
'/ Lista de Person a lista de nombres de persona
List<String> list = people.stream()
    .map(Person::getName)
    .collect(Collectors.toList());

'/ Lista de Person a TreeSet de nombres de persona
Set<String> set = people.stream()
    .map(Person::getName)
    .collect(Collectors.toCollection(TreeSet::new));

'/ Calcular suma de salarios de empleados en clase Employee
int total = employees
    .stream()
    .collect(Collectors.summingInt(Employee::getSalary));

'/ Agrupar empleados por departamento
Map<Department, List<Employee>> byDept = employees.stream()
    .collect(Collectors.groupingBy(Employee::getDepartment));
```

- Obtener los **nombres de los usuarios mayores de 18 años**

```
Set<String> resultado = usuarios.stream()
    .filter(u -> u.getEdad() > 18)
    .map(u -> u.getNombre())
    .collect(Collectors.toSet());
```

- Dada una **lista de cuentas**, aplicar reintegro de 1000 euros a cuentas con saldo > de 1000 euros

```
lista.stream()
    .filter(cu -> cu.getSaldo() > 1000)
    .forEach(cu -> cu.reintegro(1000));
```

- Obtener la **lista ordenada por nombre de los titulares de cuentas con un saldo mayor a 1000 euros.**

```
List<String> lista2 = lista.stream()
    .filter(cu -> cu.getSaldo() > 1000)
    .map(cu -> cu.getTitular())
    .sorted()
    .collect(toList());
```

## max y min

- Encontrar el mayor o menor elemento en un stream
- Retornan un Optional

```
Track trackmascorto = tracks.stream()
    .min(Comparator.comparing(track ->
        track.getLength()))
    .get();
```

- Dada una lista de usuarios, **obtiene una lista con sus nombres**:

```
List<String> nombres = usuarios.stream()
    .map(u -> u.getNombre())
    .collect(Collectors.toList());
```

- Muestra los nombres de los usuarios en orden alfabético:**

```
usuarios.stream()
    .map(u -> u.getNombre())
    .sorted()
    .forEach(System.out::println);
```

- Comprueba si algún usuario tiene más de 18 años:**

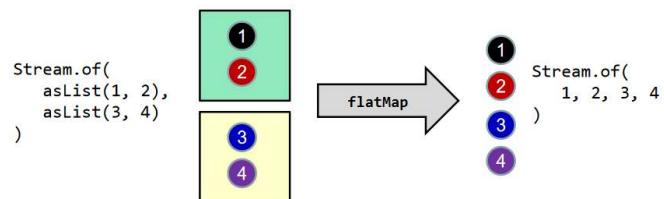
```
boolean resultado = usuarios.stream()
    .anyMatch(u -> u.getEdad() > 18);
```

## flatMap

- Convertir un stream de colecciones en un único stream con todos los elementos concatenados (operación **flatten**). Ejemplo:

```
List<Integer> lista = Stream.of(asList(1, 2), asList(3, 4))
    .flatMap(listanum -> listanum.stream())
    .collect(toList());
```

```
assertEquals(asList(1, 2, 3, 4), lista);
```



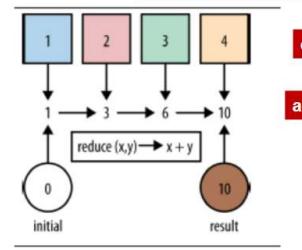
`flatMap(listanum -> listanum.stream())`

## reduce

```
T reduce(T identity,
        BinaryOperator<T> accumulator)
```

- Genera un resultado a partir de una secuencia de elementos.
- `count()`, `max` y `min()` son ejemplos de operaciones `reduce`.
- Se suele combinar con la operación `map` (por ejemplo, `mapReduce` en Hadoop).

```
int count = Stream.of(1, 2, 3, 4)
    .reduce(0, (accum, elem) -> accum + elem);
```



# Si estás en tu **spending era...**

mejor tener una app que te diga en qué tiendas se ha quedado registrada tu tarjeta.

¡Como la app de ING!

Saber más



## Ejercicios: Refactoring sugerido en "Java 8 Lambdas" de R. Warburton



-- En una lista de albums, encontrar nombres de canciones (track) de duración mayor de un minuto

```
public Set<String> findLongTracks(List<Album> albums) {
    Set<String> trackNames = new HashSet<>();
    for(Album album : albums) {
        for (Track track : album.getTrackList()) {
            if (track.getLength() > 60) {
                String name = track.getName();
                trackNames.add(name);
            }
        }
    }
    return trackNames;
}
```

-- Encontrar nombres de canciones que duran más de un minuto

```
public Set<String> findLongTracks(List<Album> albums) {
    Set<String> trackNames = new HashSet<>();

    albums.stream()
        .flatMap(album -> album.getTracks().stream())
        .filter(track -> track.getLength() > 60)
        .map(track -> track.getName())
        .forEach(name -> trackNames.add(name));

    return trackNames;
}
```

¿Se puede evitar la variable local trackNames?



-- Encontrar nombres de canciones que duran más de un minuto

```
public Set<String> findLongTracks(List<Album> albums) {
    Set<String> trackNames = new HashSet<>();
    for(Album album : albums) {
        for (Track track : album.getTrackList()) {
            if (track.getLength() > 60) {
                String name = track.getName();
                trackNames.add(name);
            }
        }
    }
    return trackNames;
}
```

!! Se viola patrón Experto !!

```
public class Track {
    String name;
    int length;

    public String getTitle() {
        return name;
    }

    public int getLength() {
        return length;
    }

    public boolean isLong(int length){
        return this.length > length;
    }
}
```

-- Encontrar nombres de canciones que duran más de un minuto

```
public Set<String> findLongTracks(List<Album> albums) {
    Set<String> trackNames = new HashSet<>();
    albums.stream()
        .forEach(album -> {
            album.getTracks().stream()
                .filter(track -> track.getLength() > 60)
                .map(track -> track.getName())
                .forEach(name -> trackNames.add(name));
        });
    return trackNames;
}
```

¿Puede conseguirse un código más compacto?

-- Encontrar nombres de canciones que duran más de un minuto

```
public Set<String> findLongTracks(List<Album> albums) {
    return albums.stream()
        .flatMap(album -> album.getTracks().stream())
        .filter(track -> track.getLength() > 60)
        .map(track -> track.getName())
        .collect(toSet());
}
```

```
public class Album { ..
    String title;
    List<Track> tracks;
```

```
public Album(String title) {
    super();
    this.title = title;
    this.tracks = new ArrayList<Track>();
}
public List<Track> getTracks() {
    return tracks;
}
public void addTrack(Track t) {
    tracks.add(t);
}
public List<Track> getLongTracks(int length) {
    return tracks.stream()
        .filter(t -> t.isLong(length))
        .collect(Collectors.toList());
}
```

```
public class Artist {..
    List<Album> albums;
    String name;
```

```
public Artist(String name) {
    super();
    this.name = name;
    this.albums = new ArrayList<Album>();
}
public void add(Album a) {
    albums.add(a);
}
public String getName() {
    return name;
}
public Set<Track> getLongTracks (int length) {
    return albums.stream()
        .flatMap(a -> a.getLongTracks(length).stream())
        .collect(Collectors.toSet());
}
```



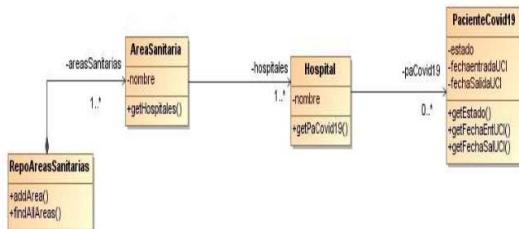
```

21     al.addTrack(t1); al.addTrack(t2);
22     Album a2 = new Album("Born in the USA");
23     a2.addTrack(t3); a2.addTrack(t4);
24     Album a3 = new Album("Tunnel of Love");
25     a3.addTrack(t5); a3.addTrack(t6);
26
27     Artist artist = new Artist ("Bruce Springsteen");
28     artist.add(a1); artist.add(a2); artist.add(a3);
29     EjemploWarburtonBook e = new EjemploWarburtonBook();
30     Set<String> result = e.getLongTracks1(artist, 240);
31
32     for (String s: result)
33         System.out.println(s);
34 }
35 public Set<String> getLongTracks1(Artist a, int lenght) {
36     return (a.getAlbums()).stream()
37         .map(t -> t.getTitle())
38         .collect(Collectors.toSet());
39 }
40 public Set<String> getLongTracks2(Artist a, int length) {
41     return (a.getAlbums()).stream()
42         .flatMap(album -> album.getTracks().stream())
43         .filter(track -> track.getLength() > length)
44         .map(track -> track.getTitle())
45         .collect(Collectors.toSet());
46 }

```

Markers Properties Servers Data Source Explorer Snippets Console  
<terminated> EjemploWarburtonBook [Java Application] C:\Program Files\Java\jre1.8.0\_201\bin  
Born in the USA  
Tunnel of Love  
Dancing in the dark

Sea una aplicación de gestión de la pandemia Covid-19 en una región que tiene varias áreas de salud, y cada hospital pertenece a un área. El diagrama de clases de abajo muestra las principales clases del dominio de esa aplicación. Cada objeto de la clase **AreaSanitaria** registra una lista de objetos **Hospital**, y éstos una lista de **PacienteCovid19**. **PacienteCovid19** tiene, entre otros, un campo **estado** que es una lista de enumerados que registra los estados de un paciente Covid-19 (INGRESO, UCI, PLANTA, ALTA), y los campos **fechaEntradaUCI** y **fechaSalidaUCI** que registran la fecha de entrada y salida de la UCI (se ignora que un paciente podría ingresar varias veces en UCI hasta su alta). **RepoAreasSanitarias** es un catálogo que almacena las instancias de **AreaSanitaria** y está implementado como un singleton, su método **findAllAreas()** retorna todas las áreas sanitarias.

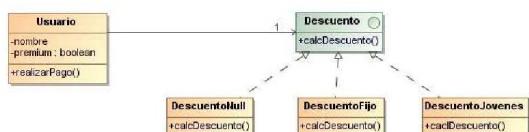


## Problemas con Null

- null o nil existe en la mayoría de lenguajes para representar la falta de un valor (the absence of a value) desde que se introdujo en Algol W en 1965.
- Su creador Tony Hoare se refiere a él como “my billion dólar mistake”.
- Problemas en el uso de null:
  - Fuente de error, excepción **NullPointerException** en Java
  - Código menos legible por las comprobaciones.
  - No tiene significado semántico y puede ser asignado a una variable de cualquier tipo referencia (agujero en el sistema de tipos)
  - Rompe la filosofía de lenguajes OO tipados estáticamente de ocultar los punteros (pointer).
- Soluciones:
  - Patrón Null Object
  - Safe Navigation operator en Groovy o tipo Option[T] en Scala.

## Patrón “Null Object”

Alternativa al uso de null para indicar la ausencia de un objeto.  
Evitar comprobaciones sobre si una variable es null antes de una llamada.



```

class DescuentoNull implements Descuento ...
// no hay descuento
public double calcDescuento() {
    return 0;
}

```

## Programación funcional con lambdas

- Código más simple y conciso
- Se expresa qué transformación se realiza en vez del cómo se realiza la transformación.
- Código funcional que expresa la intención del programador.
- Facilitan escribir código libre de efectos laterales – Se manejan variables que no pueden ser modificadas (actúan como finales)

## Ejercicio de Examen POO, 2017

Utilizando el procesamiento basado en streams, implementa un método transformar que reciba como parámetro una colección de strings y una función a aplicar sobre los objetos de dicha colección. El resultado de la ejecución será una nueva colección de string que contiene los objetos obtenidos al aplicar la función. Como ejemplo de uso, declara e inicializa una lista de cadenas y aplica el método transformar utilizando esa lista como parámetro con el fin de obtener una colección de esas cadenas en letras mayúsculas.

```

public class PruebaCodigo {
    public static <String,R> List<R> transformar(Collection<String> col,
                                                Function<String,R> function) {
        return col.stream()
            .map(function)
            .collect(Collectors.toList());
    }
    public static void main(String[] args) {
        List<String> als = Arrays.asList("orange","red","green",
                                         "pink","blue","green");
        Function<String, String> f = String::toUpperCase;
        PruebaCodigo.transformar(als, f).stream()
            .forEach(s -> System.out.println(s));
        .forEach(System.out::println);
    }
}

```

- Dado el diagrama de clases anterior, reescribe el código de abajo en Java 8 utilizando los métodos de la clase Stream. Este código calcula el **número total de pacientes que han estado y salido de UCI en un intervalo de fechas**. Reescribe dicho código en forma del siguiente método: **public long findTotalPacsUCI (LocalDate fe, LocalDate fs)**. El parámetro fe corresponde a la fecha de entrada a UCI y fs a la de salida.

```

long count = 0;
for (AreaSanitaria area:
     RepoAreasSanitarias.INSTANCE.findAllAreas())
    for (Hospital ho: area.getHospitales())
        for (PacienteCovid19 pac: ho.getPacientesCovid())
            if ((pac.getEstado().includes(Estado.UCI)) &&
                (fe.isBefore(pac.getFechaAltaUCI()) ||
                 fe.equals(pac.getFechaAltaUCI())))
                count++;
public long findTotalPacsUCI(LocalDate fe, LocalDate fs) {
    return RepoAreasSanitarias.INSTANCE.findAllAreas()
        .stream()
        .flatMap(area -> area.getHospitales().stream())
        .flatMap(hos -> hos.getPacientesCovid().stream())
        .filter(pac -> pac.isUCI(fe,fs))
        .count();
}

```

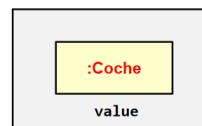
## Optional

Alternativa al uso de null para expresar que un valor puede o no estar presente.

Facilita comprobar si una variable referencia o no a un objeto.  
La clase **Optional<T>** representa que una variable de tipo T puede referenciar o no a un objeto.

Un optional puede estar en uno de dos estados posibles.

Optional<Coche> coche;



Optional<Coche> con un valor



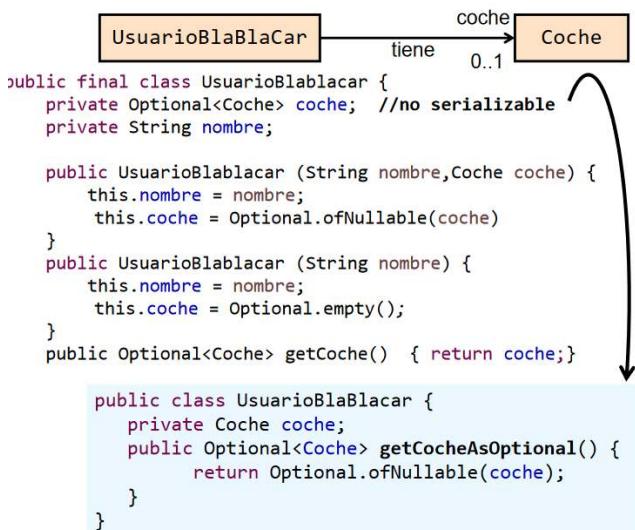
Optional<Coche> empty

## Clase Optional: Principales métodos

```

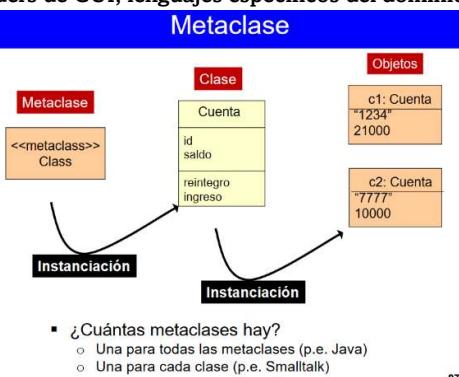
public final class Optional<T> {}.
private static final Optional<?> EMPTY = new Optional<>();
private final T value;
public static <T> Optional<T> empty() {
    Optional<T> t = (Optional<T>) EMPTY;
    return t;
}
public static <T> Optional<T> of(T value) { Crea un Optional con un valor
    return new Optional<>(value);
}
public static <T> Optional<T> ofNullable(T value) { Crea un Optional con un valor que
    return value == null ? empty() : of(value);
}
public T get() { No aconsejable para
    if (value == null) {
        throw new NoSuchElementException("No value present");
    }
    return value;
}
public boolean isPresent() { Comprobar si tiene un valor distinto de
    return value != null;
}
public T orElse(T other) { Retorna el valor distinto
    return value != null ? value : other;
}

```



## Metaprogramación e Introspección

- ✓ Una clase puede ser considerada un objeto para ser manipulada programáticamente ¿Cuál es su clase?
- ✓ Concepto de Metaclasa: clase que describe clases, sus instancias son clases.
- ✓ Un lenguaje proporciona metaprogramación si ofrece la posibilidad de manipular código de forma programática.
  - Analizar y modificar la estructura y comportamiento de un programa, p.e. crear o modificar clases.
- ✓ Un lenguaje proporciona introspección o reflexión si ofrece la posibilidad de analizar código de forma programática.
- ✓ Smalltalk, Python y Ruby son lenguajes OO tipados dinámicamente que permiten metaprogramación, Java ofrece introspección, todos ellos manejan metaclasses pero de distinto modo.
- ✓ Con meta-programación o introspección, los elementos software (clases, atributos, métodos, ..) son representados por clases.
- ✓ Una metaclasa tiene atributos que representan propiedades de una clase (campos y métodos en Java) y métodos para su acceso y modificación (en meta-programación).
- ✓ Posibilidades de la metaprogramación e introspección
  - Parametrizar métodos por clases o métodos.
  - Consultar estructura y comportamiento de una clase.
  - Además la metaprogramación permite:
    - Crear o modificar clases en tiempo de ejecución.
- ✓ Útil en programación avanzada, cuando se manejan entidades software, por ejemplo, crear inspectores de código OO, browsers de clases, builders de GUI, lenguajes específicos del dominio (DSL), ..



## Ejemplo: findAny()

- **findAny()** (clase Stream) retorna un Optional con algún elemento del stream o un Optional empty si el stream está vacío.
- Operación **no determinística** proporcionada para procesamiento paralelo.

```

Optional<Usuario> usuario = usuarios.stream()
    .filter(u -> "JesusGM".equals(u.getName()))
    .findAny();
//no recomendado
if (usuario.isPresent())
    System.out.println(usuario);

//recomendada A
usuario.ifPresent(usuario -> System.out.println(usuario));
A

Usuario usuario = usuarios.stream()
    .filter(u -> "JesusGM".equals(u.getName()))
    .findAny()
    .orElse("null");
B
System.out.println(usuario);

```

```

public final class Controlador ...
    private UsuarioBlablar usuarioActual;

    public void crearViaje(..) ...
        if (usuarioActual.getCoche().isPresent()) {
            Viaje viaje = new Viaje(..)
        }
    }

O mejor definimos tieneCoche() en UsuarioBlablar

public boolean tieneCoche() {
    return coche.isPresent()
}

Entonces ..

public void crearViaje(..) ...
    if (usuarioActual.tieneCoche()) {
        Viaje viaje = new Viaje(..)
    }
}

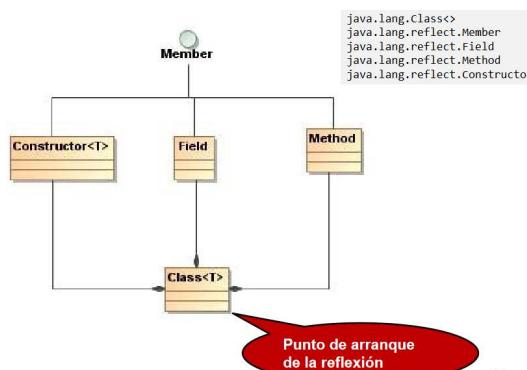
```

Portátiles desde

549€



msi  
BLACK FRIDAY



- ✓ Member ♦ getName, getDeclaringClass,...
- ✓ Field representa atributos (implementa Member): ♦ getType, get(Object o), set(Object o),..
- ✓ Method representa métodos (implementa Member): ♦ getName, getReturnType, getParameterTypes,..

- **Class<T>.forName(String className)**
  - Retorna la metaclasa (instancia de Class<T>) de la clase cuyo nombre es dado como argumento. Método estático.
- **newInstance() (deprecated en Java 9)**
  - Retorna una instancia de la clase que es instancia de la metaclasa que es el objeto receptor. Método de instancia.

```
String nombreClase; //path de la clase
Class<?>.forName(nomClase).newInstance() Java <=8
Class<?>.forName(nomClase).getDeclaredConstructor().newInstance() Java >=9
```

Formas de obtener una instancia de Class<T> (metaclasa)

- **getClass()** en clase Object
  - Cuenta oc;
  - ..
  - Class<Cuenta> c = oc.getClass();
- **forName()** en clase Class<T>
  - Class<Cuenta> c = Class<?>.forName ("com.banco.Cuenta");
- **Operador class**
  - Class<Cuenta> c = Cuenta.class
  - Class<String> c = String.class
- **Métodos que retornan** objetos de tipo **Class**
  - Class<?> c = javax.swing.JButton.class.getSuperclass();

- **getMethod()** retorna un método (instancia de Method) a partir del nombre del método y los tipos de sus parámetros
- **invoke()** invoca a un método (objeto receptor de la invocación) y tiene como parámetros el objeto receptor del método y los argumentos.

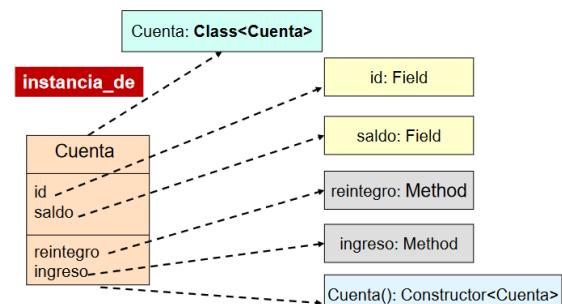
```
public static void main(String[] args) {
    try {
        Class<Cuenta> classCuenta = Cuenta.class;
        Method m = classCuenta.getMethod("ingreso", new Class[] {int.class});
        Cuenta cu = new Cuenta("jesusGM");
        m.invoke(cu, new Object[]{new Integer(1000)});
        m.invoke(cu, new Object[]{new Integer(400)});
        System.out.println(cu.saldo);
    } catch (NoSuchMethodException e) {}
    catch (InvocationTargetException e) {}
    catch (SecurityException e) {}
}
```

✓ Cada clase e interface es instancia de Class<T>.

-Por ejemplo, la clase Cuenta es instancia de Class<Cuenta>

✓ Class<T> representa clases e incluye métodos que retornan información sobre las clases:

- Nombre (getName)
- Campos (getFields, getDeclaredFields) y métodos (getMethods, getDeclaredMethods())
- Superclase (getSuperClass)
- Constructores (getConstructors, getConstructor(Class<T> .. parameters), getDeclaredConstructors())
- Interfaces que implementa (getInterfaces)
- Paquete de la clase (getPackage)
- Consultar tipo de clase (isPrimitive, isEnum, isInterface,..)



"En resumen, la reflexión es una utilidad muy potente que es requerida para algunas tareas de programación sofisticadas, pero que tiene muchas desventajas. Si estás escribiendo un programa que debe manipular clases desconocidas en tiempo de compilación, deberías, en todo lo posible, usar reflexión sólo para instanciar objetos y acceder a esos objetos utilizando alguna interfaz o superclase conocida en tiempo de compilación"

#### Desventajas:

-Se pierden beneficios del control de tipos en compilación, errores en tiempo de ejecución.

-Código verbose y tedioso de escribir.

**Ejemplo Introspección.** Dada una clase imprimir los nombres de sus campos y métodos. (Ken Arnold et al., El lenguaje de programación Java, 3<sup>a</sup> edición, Addison-Wesley, 2001)

```
package tds.reflexion.examples;
import java.lang.reflect.*;
import java.util.Arrays;

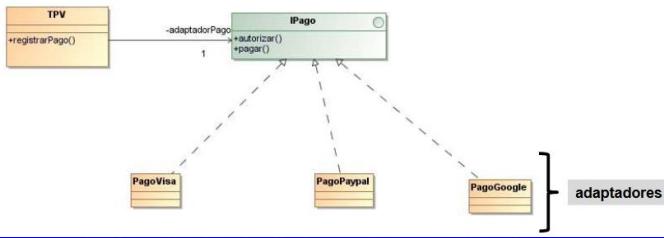
public class EjemploReflexion {
    public static void main (String[] args) {
        try {
            Class<?> c = Class.forName(args[0]);
            System.out.println(c);
            printMiembros (c.getFields());
            printMiembros (c.getConstructors());
            printMiembros (c.getMethods());
        }
        catch (ClassNotFoundException e) {
            System.out.println("Clase desconocida: " + args[0]);
        }
    }
    private static void printMiembros(Member[] miembros) {
        // imprimir miembros excepto los miembros de Object
        Arrays.asList(miembros).stream()
            .filter(m -> m.getDeclaringClass() != Object.class)
            .map(m-> m.getName())
            .forEach(s -> System.out.println(s));
    }
}
```

105



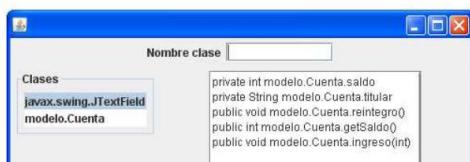
VER OFERTAS

WUOLAH



## Ejercicio 1. Introspección Java

Crea una ventana como la que se muestra abajo para mostrar los métodos y campos declarados en una clase dada. En el campo de texto de arriba se introduce un nombre de clase que se añade a la lista de la izquierda y se muestran sus campos y métodos (a excepción de los que hereda de Object) en el área de texto de la derecha, al añadir una nueva clase o al seleccionar un elemento de la lista.



### java.util Class Collections

```
sort(List<T> list, Comparator<? super T> c)
```

Ordena la lista especificada de acuerdo al orden establecido por el comparador establecido.

### java.util Interface Comparator<T>

```
compare(T o1, T o2)
```

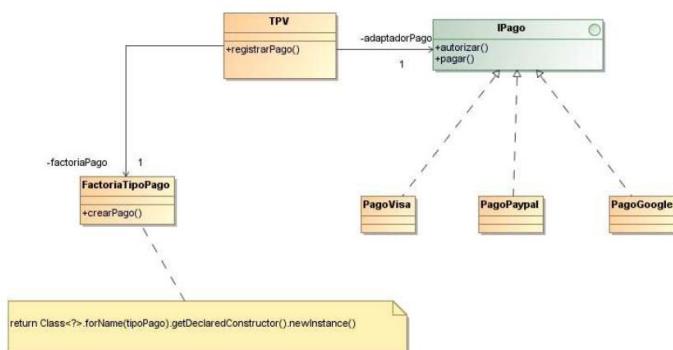
Compara argumentos para ordenar

### Anotaciones Java

- ✓ Mecanismo facilitado por Java para proporcionar información (metadatos) que se especifica separada del código del programa y que es útil para:
  - ◆ Compilador,
  - ◆ Herramientas software (por ejemplo para generar código)
  - ◆ Datos runtime
- ✓ La facilidad consiste de:
  - ◆ Sintaxis para declarar anotaciones
  - ◆ Sintaxis para escribir anotaciones
  - ◆ API para leer anotaciones,
  - ◆ Un tipo especial de archivo para almacenarlas
  - ◆ Una herramienta de procesamiento de anotaciones.

- Programadores no necesitarán habitualmente crear sus propias anotaciones sino que usarán aquellas proporcionadas por herramientas o librerías.
- Una declaración de anotación se declara como una interfaz y usa el símbolo @ para distinguirlas.
- Los métodos de la interfaz “anotación” no tienen parámetros y retornan valores de tipos primitivos, String, enums, o un array y pueden tener valores por defecto.

Las anotaciones son un modificador más y por convención se escriben precediendo a todos los modificadores del elemento anotado.



## Ejercicio 2. Introspección Java

La clase Collections proporciona el método `sort(List<T>, Comparator)` que ordena la lista que se pasa como primer argumento (`List<T>`) utilizando como criterio de ordenación el comparador que se pasa como segundo argumento (instancia de una clase que implementa la interfaz `Comparator<T>`).

Dada una clase C, se desea ordenar listas de objetos de C de acuerdo al valor de uno de sus campos que debe ser de tipo dato primitivo, por ejemplo una lista de objetos Usuario por su campo dni o nombre de tipo String. Utiliza la introspección de Java para idear una forma que evite tener que crear una clase que implemente `Comparator<T>` por cada par <tipo de objeto, campo>, sino una única implementación que sea independiente de un par concreto <clase y campo>.

Este ejercicio estaba planteado para Java 7 y puede reconsiderarse también para Java 8.

### Ejercicio

```
class Cliente {
    private String dni;
    private String nombre;
    private int sueldo;
    // getters and setters
}
```

Ordenar ArrayList<Cliente> por dni, o nombre, o sueldo, ..

```
class Cuenta {
    private String ccc;
    private String titular;
    private int saldo;
    // getters and setters
}
```

Ordenar ArrayList<Cuenta> por ccc, o titular,..

Crear una clase que implemente `Comparator<T>` que evite crear un comparador concreto por cada par clase/campo, por ejemplo, `Cliente/dni`. Esto es, que esté parametrizada por esa información.

**OFERTAS  
BLACK FRIDAY**

**msi®**

# **Esta oferta**

**es como una doble victoria  
tuya: disfrútala ahora porque  
no pasa dos veces.**

**Ver ofertas**



Tu viejo portátil ya dio lo que tenía que dar. Pásate a MSI: rápido, potente y sin dramas. Lo enciendes y estás listo para todo. Aprovecha las ofertas y despídete del modo “se cuelga cada dos por tres”.

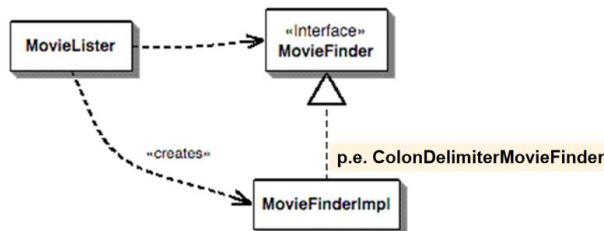
## Inyección de dependencias

(Martin Fowler, [martinfowler.com/articles/injection.html](http://martinfowler.com/articles/injection.html))

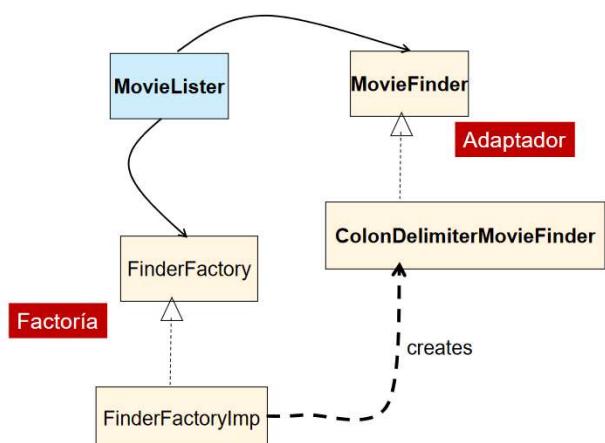
¿Cómo conecto la clase MovieLister con el servicio finder?

```
class MovieLister {..  
    ??? finder;  
    public Movie[] moviesDirectedBy(String arg) {  
        List<Movie> allMovies = finder.findAll();  
        for (Iterator<Movie> it = allMovies.iterator();it.hasNext();) {  
            Movie movie = it.next();  
            if (!movie.getDirector().equals(arg)) it.remove();  
        }  
        return (Movie[]) allMovies.toArray(new Movie[allMovies.  
            size()]);  
    }  
}
```

finder es un servicio de registro de películas findAll() retorna todas las películas registradas



- MovieLister no debería estar acoplado a una implementación concreta como ColonDelimiterMovieFinderImpl
- Deseable una dependencia de sólo la interfaz.
- El servicio debería conocerse en run-time.



- GUICE es un framework open source para Java liberado por Google con licencia Apache (<http://code.google.com/p/google-guice/>)
- Se puede añadir dependencia al POM en proyecto Maven.
- Usa anotaciones para inyectar las dependencias
- Ejemplo para un constructor:

```
class MovieLister {..  
    public MovieLister(MovieFinder finder) {  
        this.finder = finder;  
    }  
}  
  
import com.google.inject.Inject;  
public class MovieLister {..  
    private final MovieFinder finder;  
    @Inject  
    public MovieLister(MovieFinder finder) {  
        this.finder = finder;  
    }  
}
```

Conectarlo a una implementación concreta

```
public class ColonDelimiterMovieFinder implements MovieFinder {  
    public ColonDelimiterMovieFinder(@FileName String fileName) {  
        ...  
    }  
}
```

- Se puede usar una interfaz ... (patrón Adaptador)

```
public interface MovieFinder {  
    public List<Movie> findAll();  
}
```

- ... pero se necesitaría crear una instancia de una clase que sea una implementación de esa interfaz, una posibilidad sería

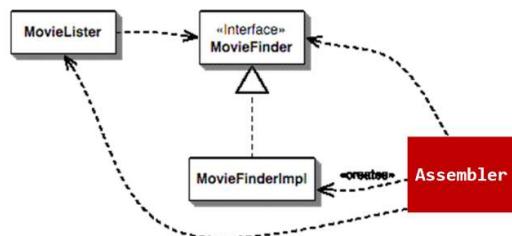
```
class MovieLister { ...  
    private MovieFinder finder;  
    public MovieLister() {  
        finder = new ColonDelimitedMovieFinder("movies1.txt");  
    }  
}
```

- Pero otros servicios pueden retornar las películas en otros formatos: XML, JSON,..., --> diferentes clases de implementación de la interfaz MovieFinder, ¿cómo evitar acoplamiento?

### Utilización de factorías

```
class MovieLister {...  
    private MovieFinder finder;  
    public MovieLister(String factImp) {  
        finder = FinderFactory.getInstancia(factImp)  
            .createFinder();  
    }  
}
```

- La factoría FinderFactory se encarga de todo lo relativo a la creación e inicialización de la instancia concreta de MovieFinderImpl.
- MovieLister u otra clase debe encargarse de crear la instancia concreta de FinderFactory, si es necesario.
- Relaciones directas e indirectas con muchas factorías en el código real.



La clase "Assembler" se encarga de conectar MovieLister y una MovieFinderImpl (por ejemplo ColonDelimitedMovieFinder). Containers son frameworks que automatizan la inyección de dependencias (DI), por ejemplo GUICE. Varias técnicas: basadas en constructores, métodos setter e interfaces, y en anotaciones.

- Módulos encapsulan las dependencias.

```
public class MovieListerModule extends AbstractModule {  
    @Override  
    protected void configure() {  
        bind(MovieFinder.class)  
            .to(ColonDelimiterMovieFinder.class);  
        bind(String.class)  
            .annotatedWith(FileName.class)  
            .toInstance("movies1.txt");  
    }  
}
```

### Iniciar GUICE:

```
public static void main(String[] args) {  
    Injector injector = Guice.createInjector(new MovieListerModule());  
    MovieLister lister = injector.getInstance(MovieLister.class);  
    lister.moviesDirectedBy("Sergio Leone");  
}
```

# Google Gemini: Plan Pro a 0€ durante 1 año. Tu ventaja por ser estudiante.

Oferta válida hasta el 9 de diciembre de 2025

Consigue la oferta



Después 21,99€/mes

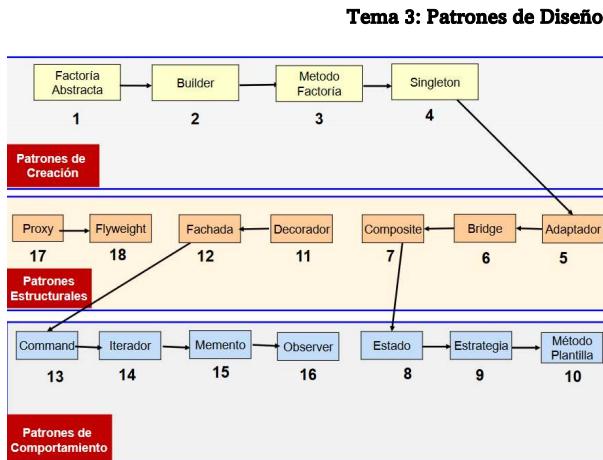
Domina cualquier tema con el Aprendizaje Guiado.

Puedes explicarme como se crea un eclipse lunar completo y sus fases?

¡Claro vamos paso a paso para que lo entiendas a la perfección! ☺ ☺ ☺ ☺ ☺ ☺ ☺ ☺ ☺ ☺

2:44

+ Aprendizaje Guiado x



“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”

Programadores inexpertos podrían aprovechar experiencia de expertos a través de 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.”

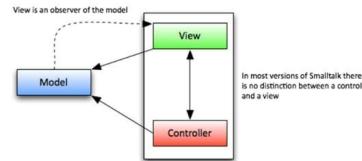
“Los patrones conducen a arquitecturas más pequeñas, simples y comprensibles”

✓ Una solución reutilizable a un problema recurrente en la construcción de software y que funcionó en el pasado.

✓ Lenguajes de patrones

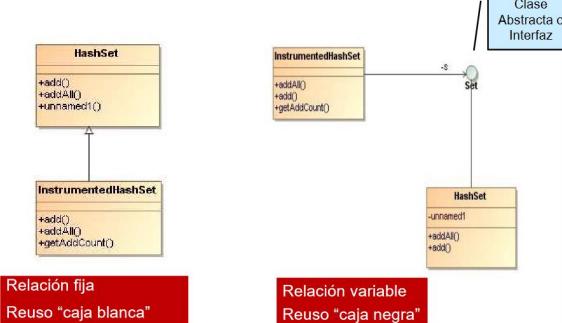
♦ Cada patrón es identificado por un nombre y se describe

## Framework Modelo-Vista-Control (MVC)

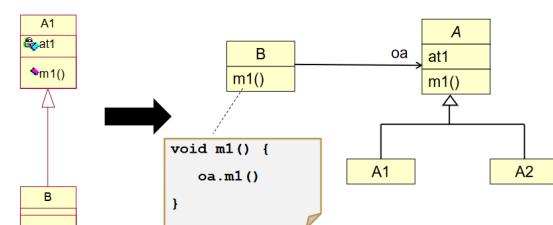


- Creado para construir aplicaciones interactivas gráficas en Smalltalk-80 (Trygve Reenskaug).
- Basado en tres tipos de objetos:
  - **Modelo:** objetos del dominio
  - **Vista:** objetos presentación en pantalla (interfaz de usuario)
  - **Controlador:** objetos que define cómo reacciona la interfaz a la entrada del usuario.
- Utiliza patrones Observer, Método Factoría, Estrategia, Decorador, Composite, entre otros.

## Favorecer composición



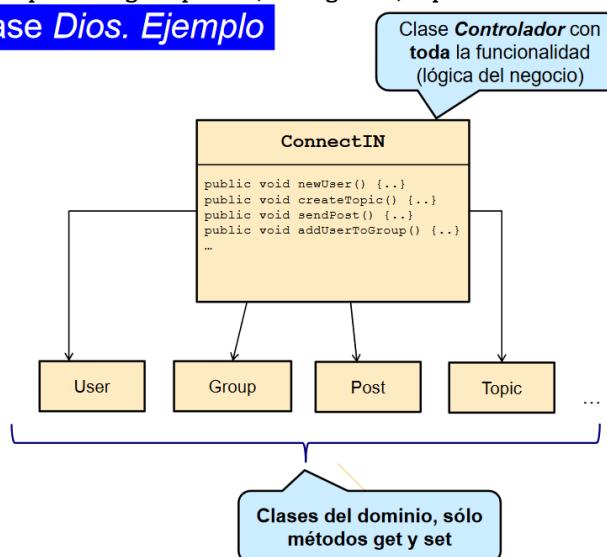
## Delegación



Presente en muchos patrones: State, Strategy, Visitor,..

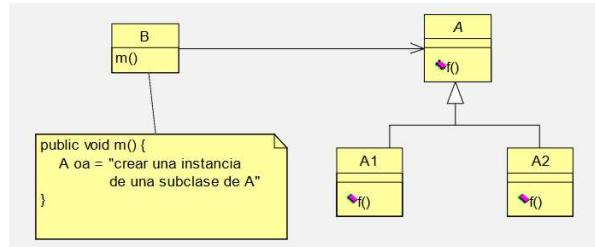
- Un anti-patrón es una mala práctica en el desarrollo de software (análisis, diseño, e implementación).
- Ejemplo: clase Dios
- Un refactoring es un cambio en la estructura interna de un programa para mejorar su calidad (comprensión, evolución,..) sin cambiar su comportamiento.
- Ejemplos: código duplicado, clase grande, expresión condicional compleja, eliminar jerarquía,

## Clase Dios. Ejemplo

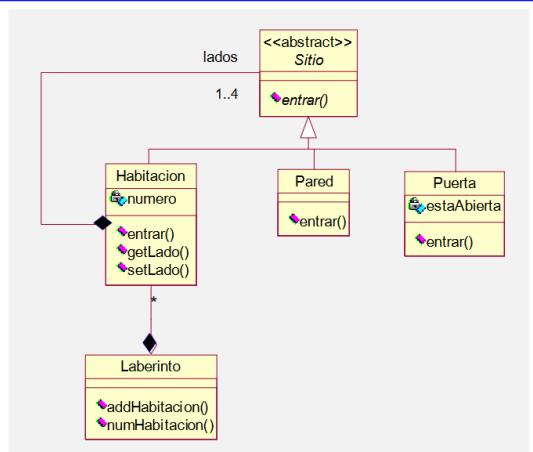


## Patrones de Creación

- Ayudan a crear aplicaciones independientes de cómo los objetos son creados, compuestos y representados.
- Se conoce una clase abstracta o interfaz **pero no se conoce la clase concreta a instanciar**.
- Flexibilidad** en qué se crea, quién lo crea, y cómo se crea.



## Ejemplo: Creación de un objeto Laberinto



49

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

        lab.addHabitacion (h1);
        lab.addHabitacion (h2);

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

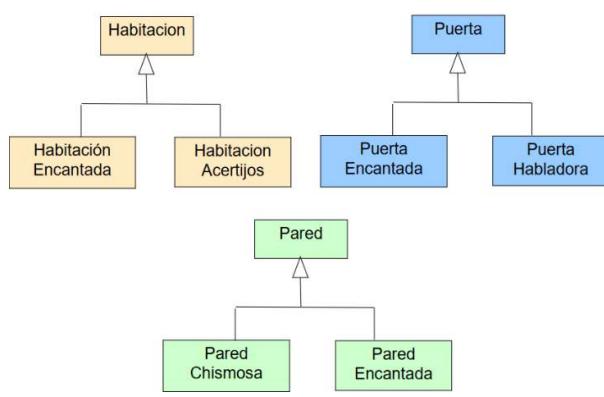
        h2.setLado(Norte, new Pared());
        ...
        return lab;
    }
}
```

**Dependencia de clases concretas**

**Estructura del laberinto es fija**

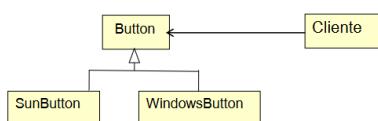
Poco flexible:

¿Qué implicaciones tiene 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.

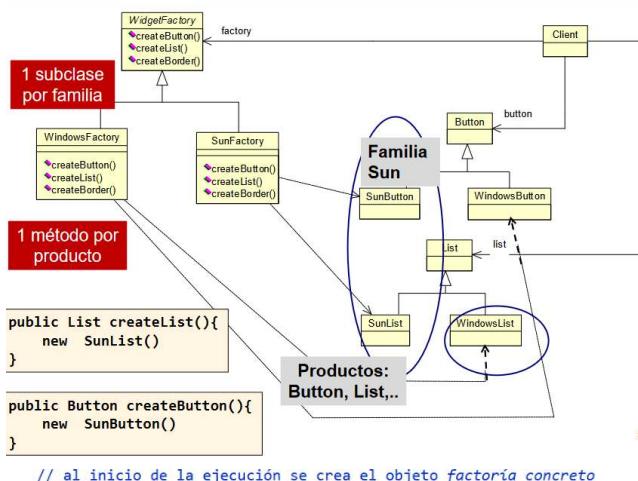


## Abstract Factory (Factoría Abstracta)

- 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 plataformas: Sun, Windows, Ubuntu, Mac,..
  - Se desea independencia de la plataforma (cross-platform).

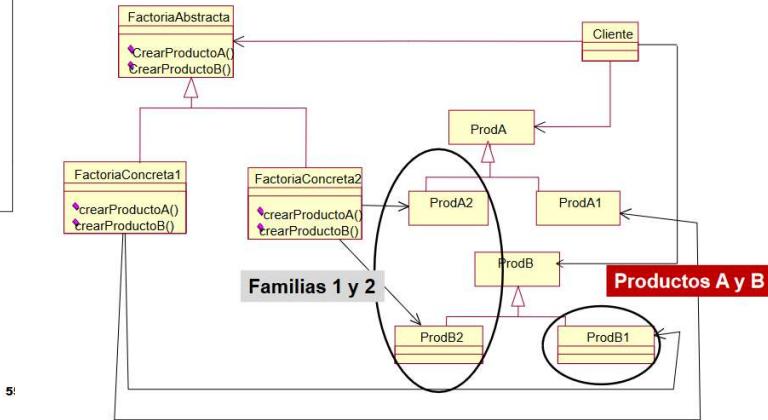


## Factoría Abstracta: Motivación



- Las factorías son realmente clases **singletons**, como veremos más adelante.

## Factoría Abstracta: Estructura



### v Aplicabilidad

- Un sistema debería ser configurado para una familia de productos y ser independientes de familias y productos concretos.

### v Consecuencias

- Desacopla a la aplicación de las clases concretas de implementación.
- Facilita el intercambio de familias de productos.
- Favorece la consistencia entre productos
- Un nuevo producto requiere modificar todas las clases factorías.

### v Implementación

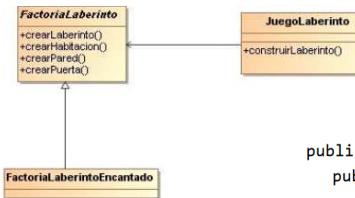
- Factorías como singleton.

```

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

public class FactoriaLaberintoEncantado extends FactoriaLaberinto {
    public ParedEncantada crearPared {
        return new ParedEncantada(); }
    public HabitacionEncantada crearHabitacion(int n){
        return new HabitacionEncantada(n); }
    public PuertaEncantada crearPuerta (Habitacion h1, Habitacion h2) {
        return new PuertaEncantada(h1,h2); }
}

```



```

public class JuegoLaberinto {
    public Laberinto construirLaberinto (FactoriaLaberinto factoria) {
        Laberinto unLaberinto = factoria.crearLaberinto();
        Habitacion h1 = factoria.crearHabitacion(1);
        Habitacion h2 = factoria.crearHabitacion(2);
        Puerta unaPuerta = factoria.crearPuerta(h1,h2);
        unLaberinto.addHabitacion(h1);
        unLaberinto.addHabitacion(h2);
        h1.setLado(Norte, factoria.crearPared());
        h1.setLado(Este, unaPuerta);
        ...
        h2.setLado(Oeste, unaPuerta);
        h2.setLado(Sur, factoria.crearPared());
        return unLaberinto; }
}

```

59

## Builder (Constructor)

### v Propósito

- Construcción de un objeto complejo, separando el proceso de construcción de su representación, así que el mismo proceso puede crear diferentes representaciones.

### v Motivación

- Un traductor de documentos docx a otros formatos.
- ¿Es posible añadir una nueva conversión sin modificar el traductor?

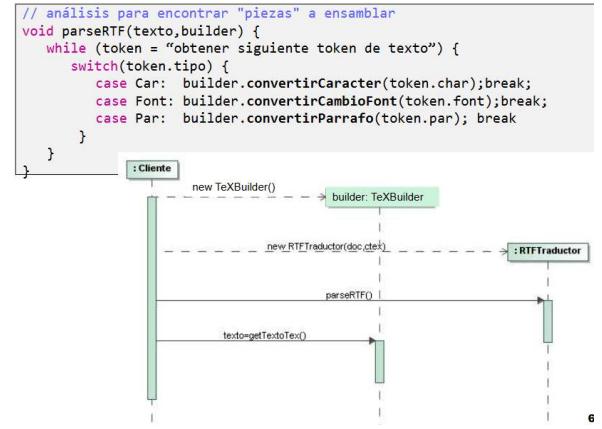
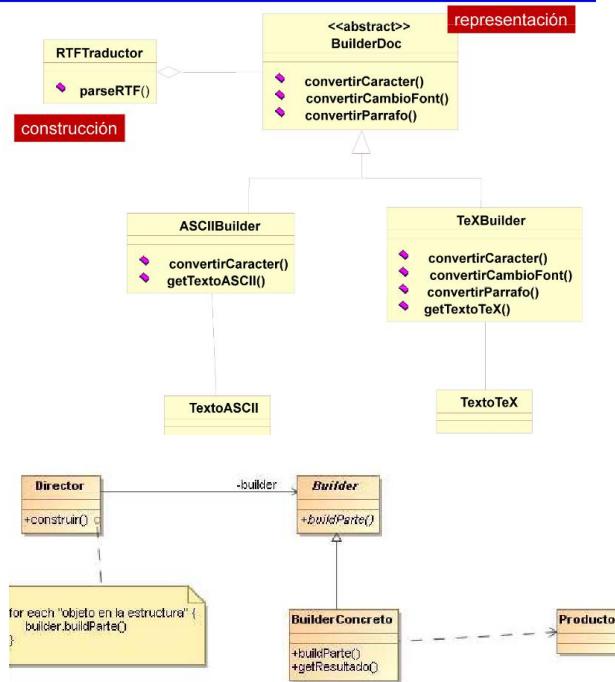


# Organiza tu futuro: estudia hoy para destacar mañana.

En Carpe Diem te esperan cursos adaptados a ti.  
Una forma fácil y real de avanzar profesionalmente.



## Builder: Motivación



### v Aplicabilidad

- ♦ Crear un objeto complejo y el algoritmo de construcción debe ser independiente de las piezas que conforman el objeto complejo y de cómo se ensamblan.

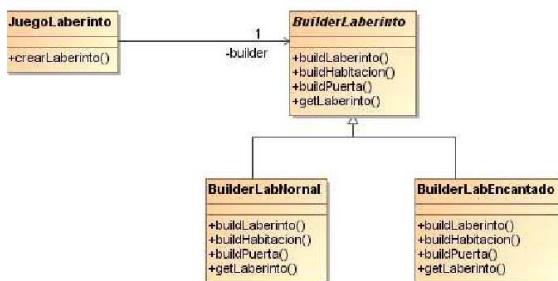
- ♦ El proceso de construcción debe permitir diferentes representaciones para el objeto que se construye.

### v Consecuencias

- ♦ Permite cambiar la representación interna del producto.
- ♦ Reúne el código para la representación y ensamblaje.
- ♦ Clientes no necesitan conocer nada sobre estructura int.
- ♦ Proporciona un control fino del proceso de construcción.
- ♦ Diferentes "directores" pueden reutilizar un mismo builder

### v 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 que añadir el nuevo token al producto en construcción.
- ♦ Los métodos de la clase Builder pueden no ser abstractos sino vacíos.
- ♦ Las clases de los productos no siempre tienen una clase abstracta común.



```

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

```

class Director {
    public class JuegoLaberinto {
        public Laberinto construirLaberinto (BuilderLaberinto builder) {
            builder.buildLaberinto();
            builder.buildHabitacion(1);
            builder.buildHabitacion(2);
            builder.buildPuerta(1,2);
            ...
            return builder.getLaberinto();
        }
    }
    class BuilderLabNormal extends BuilderLaberinto {
        private Laberinto labActual;
        public BuilderLabNormal () { labActual = null; }
        public void buildLaberinto () { labActual = new Laberinto(); }
        public void buildHabitacion (int n) {
            if ( labActual.numeroHabitacion(n) == null ) {
                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.numeroHabitacion(n1);
            Habitacion h2 = labActual.numeroHabitacion(n2);
            Puerta p = new Puerta(h1,h2);
            h1.setLado(getParedComun(h1,h2), p);
            h2.setLado(getParedComun(h2,h1), p);
        }
        Laberinto getLaberinto () { return labActual; }
    }
}
  
```

Oculta la estructura interna

**"Considera un builder cuando te enfrentes a un constructor con muchos parámetros"** ("Effective Java", 2<sup>a</sup> Ed., Joshua Bloch)

### ¡¡ NO CONFUNDIR CON PATRÓN BUILDER !!

```
public class DatosNutricionales {
    private final int tamañoUnidad;           //obligatorio
    private final int unidades;               //obligatorio
    private final int calorías;               //opcional
    private final int grasas;                //opcional
    private final int sodio;                 //opcional
    private final int carbohidratos;          //opcional

    public static class Builder {...}
    // clase anidada estática en siguiente diapositiva

    private DatosNutricionales (Builder builder) {
        tamañoUnidad = builder.tamañoUnidad;
        unidades = builder.unidades;
        calorías= builder.calorías;
        grasas= builder.grasas;
        sodio = builder.sodio;
        carbohidratos = builder.carbohidratos;
    }
}
```

Muchos campos opcionales

```
public static class Builder {
    private final int tamañoUnidad;
    private final int unidades;
    private final int calorías = 0;
    private final int grasas = 0;
    private final int sodio = 0;
    private final int carbohidratos = 0;

    public Builder (int tamañoUnidad, int unidades) {
        this.tamañoUnidad = tamañoUnidad;
        this.unidades = unidades;
    }
    public Builder calorías (int val) {
        calorías = val; return this;
    }
    public Builder grasas(int val) {
        grasas= val; return this;
    }
    public Builder sodio(int val) {
        sodio= val; return this;
    }
    public Builder carbohidratos(int val) {
        carbohidratos= val; return this;
    }
    public DatosNutricionales build () {
        return new DatosNutricionales(this);
    }
}
```

Facilidad de uso

Fluent-API  
basada en técnica  
"method chaining"

72

73

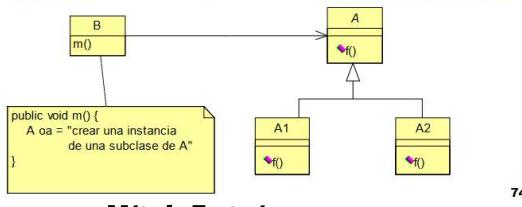
## Factory Method (Método Factoría)

### ■ 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 B que usa 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, pero **no se conoce de qué tipo**.

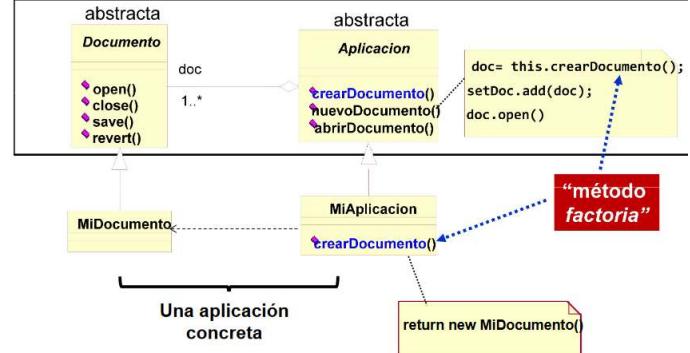


Método Factoría

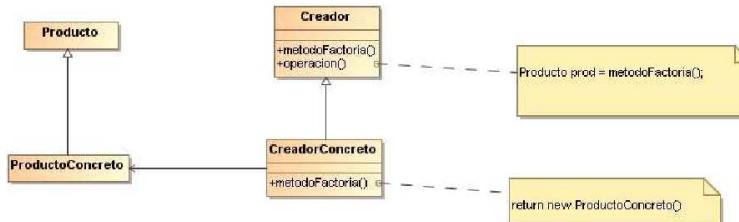
74

## Método Factoría: Motivación

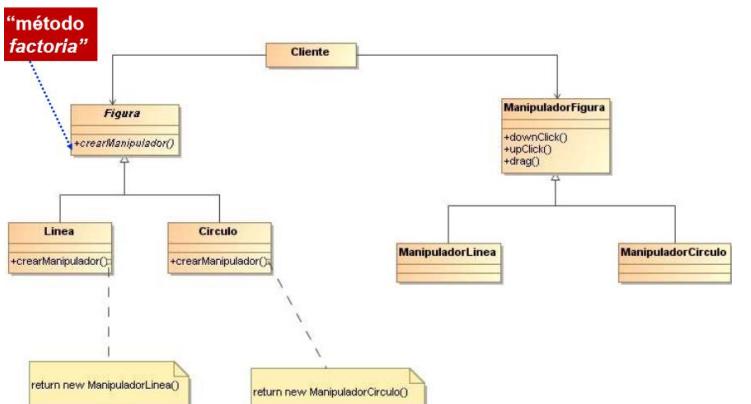
Framework "OpenDocuments"



Una aplicación concreta

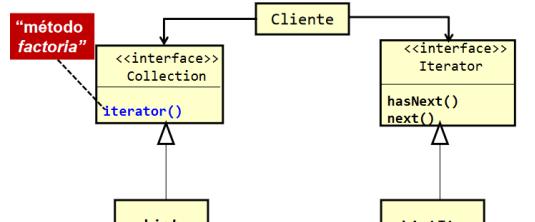


## Método Factoría: Jerarquías paralelas



```
class Cliente {
    void moverFigura (Figura fig) {
        ManipuladorFigura mf = fig.crearManipulador()
        mf.draw();
        ...
    }
}
```

## Método Factoría: Jerarquías paralelas en caso de un iterador externo en Java

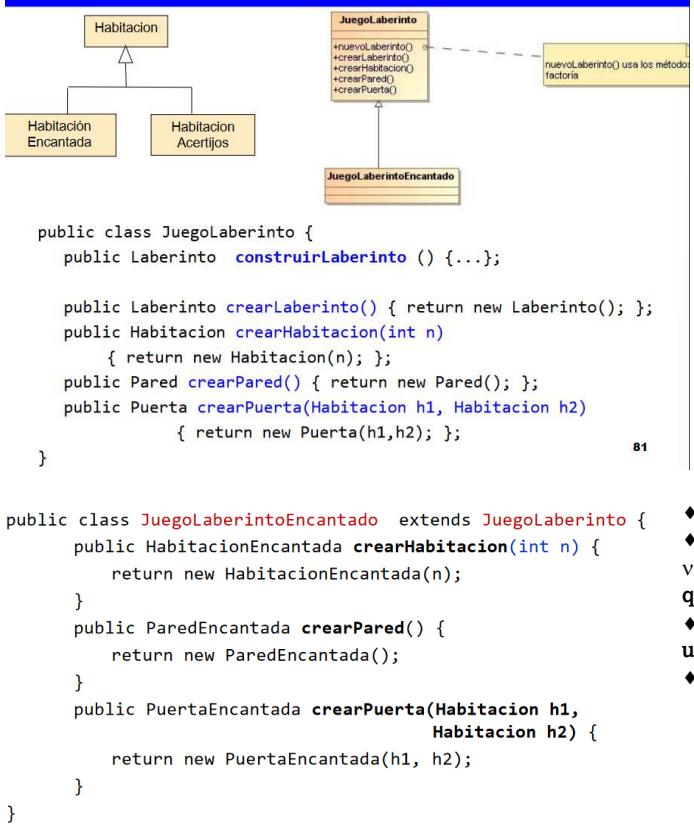


```
class Cliente{
    void printCollection (Collection c) {
        Iterator it = c.iterator();
        while (it.hasNext()) {
            System.out.println(it.next());
        }
    }
}
```

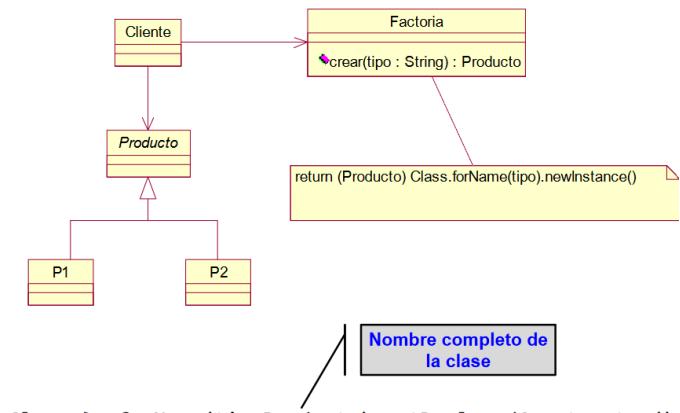
80

WUOLAH

## Método Factoría: Ejemplo Laberinto



## Método Factoría con newInstance()



```

public Laberinto construirLaberinto () {
    Laberinto unLab = this.crearLaberinto();
    Habitacion h1 = this.crearHabitacion(1);
    Habitacion h2 = this.crearHabitacion(2);
    Puerta unaPuerta = this.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;
}

```

✓ Dos posibilidades para método factoría en la clase raíz de la jerarquía de productos.

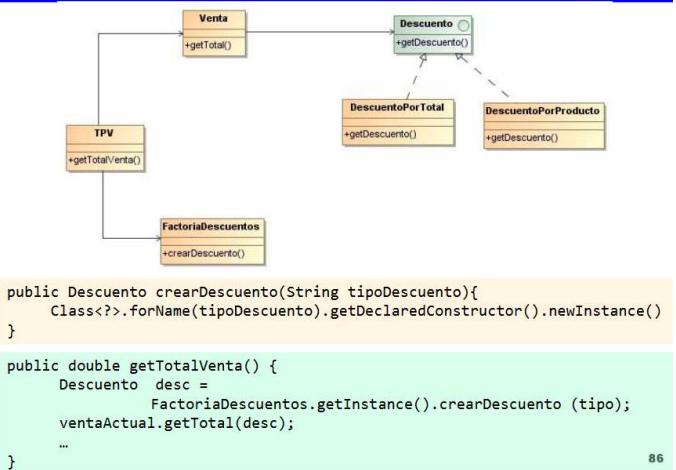
◆ Método abstracto

◆ Una implementación por defecto.

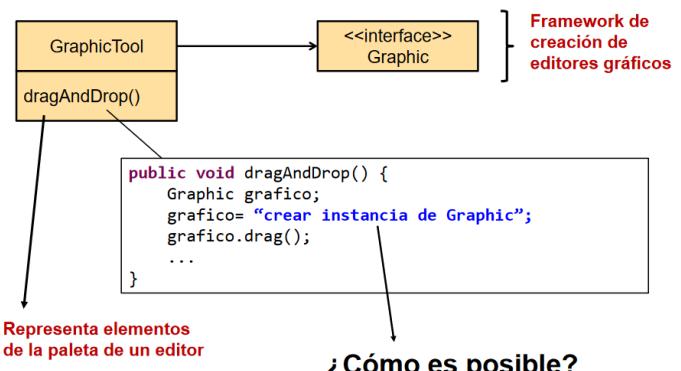
✓ Cuando son innecesarias las subclases de Creador, se evita tener que crearlas con una clase Factoría que tiene el método factoría:

- ◆ Un parámetro identifica a la clase del objeto a crear: Switch o uso de Introspección (uso de la metaclasa)
- ◆ Alternativa en Java 8: Interface funcional Supplier<>

## Método Factoría con newInstance()



## Método Factoría con Supplier



¿Cómo es posible?

```

public class GraphicTool {
    private Supplier<Graphic> supplier;

    public void setSupplier(Supplier<Graphic> supplier) {
        this.supplier = supplier;
    }
    public void dragAndDrop() {..}
    supplier.get().drag();
    }

    public void printGraphicName(){
        System.out.println(supplier.get().getName());
    }
}

public interface Graphic {
    public String getName();
}

```

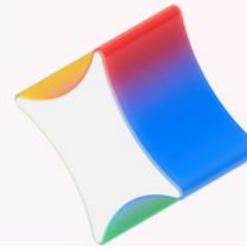
# Google Gemini: Plan Pro a 0€ durante 1 año.

Tu ventaja por ser estudiante.

Oferta válida hasta el 9 de diciembre de 2025

Consigue la oferta

Después 21,99€/mes



```

public class Rectangle implements Graphic {
    private int width;
    private int height;
    private String name;

    public Rectangle(String name, int width, int height) {
        super();
        this.name = name;
        this.width = width;
        this.height = height;
    }
    @Override
    public String getName() {
        return name;
    }
}

```

```

public class App {
    public static void main(String[] args) {
        Supplier<Graphic> g = () -> new Rectangle("Caja 1",3,2);
        //se crea e inicializa un GraphicTool para rectángulos
        // tendré uno por objeto en paleta
        GraphicTool gt = new GraphicTool();
        gt.setSupplier(g);
        gt.printGraphicName();
    }
}

```

```

1 public class ShapeFactory {
2
3     final static Map<String, Supplier<Shape>> map = new HashMap<>();
4     static {
5         map.put("CIRCLE", Circle::new);
6         map.put("RECTANGLE", Rectangle::new);
7     }
8     public Shape getShape(String shapeType){
9         Supplier<Shape> shape = map.get(shapeType.toUpperCase());
10        if(shape != null) {
11            return shape.get();
12        }
13        throw new IllegalArgumentException("No such shape " + shapeType.toUpperCase());
14    }
15 }

```

```

public class Singleton {
    public static Singleton getInstance() { ③
        if (unicaInstancia == null)  unicaInstancia = new Singleton();
        return unicaInstancia;
    }
    private Singleton() { } ①
    private static Singleton unicaInstancia = null; ②
    private int dato = 0;
    public int getData() {
        return dato;
    }
    public void setData(int i) {dato = i;}
}

```

Clase "normal"

- ✓ Aplicabilidad
  - ♦ Debe existir una única instancia de una clase, accesible globalmente.
- ✓ Consecuencias
  - ♦ Acceso controlado a la única instancia
  - ♦ Evita usar variables globales
  - ♦ Es posible generalizar a un número variable de instancias
  - ♦ No es lo mismo que declarar todos los métodos estáticos
    - ✓ No es posible definir una jerarquía de clases
    - ✓ Una clase puede cambiar y dejar de ser Singleton
    - ♦ La clase Singleton puede tener subclases.

## Singleton: Ejemplo "Repositorio de circuitos"

```

public class RepositorioCircuitos {
    public static RepositorioCircuitos getInstance() {
        if (unicaInstancia == null)
            unicaInstancia = new RepositorioCircuitos();
        return unicaInstancia;
    }
    private static RepositorioCircuitos unicaInstancia = null;

    private RepositorioCircuitos() {
        elems = new Hashtable<Circuito>();
    }
    private Hashtable<String,Circuito> elems;

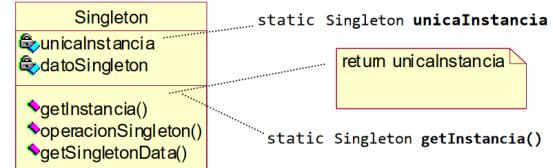
    public Circuito getCircuito(String id) {
        Circuito c = elems.get(id);
        return c.clone();
    }
    public void addCircuito(String id, Circuito c) {
        elems.put(id,c);
    }
}

Circuito sumador= RepositorioCircuitos.getInstance()
               .getCircuito("sumador");

```

- Singleton**
- ✓ Propósito
    - ♦ Asegurar que una clase tiene una única instancia y asegurar un punto de acceso global.
  - ✓ Motivación
    - ♦ Ejemplos: un catálogo o repositorio, una factoría abstracta, un manejador de ventanas,...
    - ♦ La clase se encarga de asegurar que existe una única instancia y de su acceso.

Singleton.getInstance().operacionSingleton()



## Singleton: Inicialización temprana

```

public class Singleton {
    private static Singleton unicaInstancia = new Singleton();
    private Singleton() {}
    public static Singleton getInstance() {
        return unicaInstancia;
    }
    private int dato = 0;
    public int getData() {return dato;}
    public void setData(int i) {dato = i;}
}

```

### Inconvenientes:

- Siempre se crea la instancia, aunque luego no se necesite.
- La inicialización puede necesitar lógica de creación compleja y condicional

```

public enum Planeta {
    MERCURIO (3.303e+23, 2.4397e6),
    VENUS (4.869e+24, 6.0518e6),
    TIERRA (5.976e+24, 6.37814e6),
    MARTE (6.421e+23, 3.3972e6),
    JUPITER (1.9e+27, 7.1492e7),
    SATURNO (5.688e+26, 6.02568e7),
    URANO (8.686e+25, 2.5559e7),
    NEPTUNO (1.024e+26, 2.4746e7);

    private final double masa;
    private final double radio;
    Planet(double masa, double radio) {
        this.masa = masa;
        this.radio = radio;
    }
    private double masa() {
        return masa;
    }
    private double radio() {
        return radio;
    }
    public static final double G = 6.67300E-11;
    double gravedadSuperficie() {
        return G * massa() / (radius * radius);
    }
}

for (Planeta p : Planeta.values())
{
    System.out.printf(
        "Su peso es:",
        p.gravedadSuperficie());
}

```

WUOLAH

## Singletons con enums

```
public enum CatalogoCircuitos {
    INSTANCE;
    private Hashtable elems;
    private CatalogoCircuitos() {
        elems = new Hashtable();
    }
    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 sumador = CatalogoCircuitos.INSTANCE.getCircuito("sumador");

### Effective Java (2ª edición)

**Item 3:** Enforce the singleton property with a private constructor or an enum type  
While this approach has yet to be widely adopted,  
a single-element enum type is the best way to implement a singleton.

Pero, enums no pueden ser extendidos

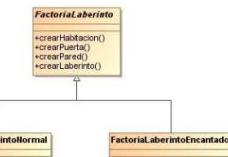
102

## Singleton (Con subclases): Análisis de casos

```
public abstract class FactorialLaberinto {
    private static FactorialLaberinto unicaInstancia = null;

    public static FactorialLaberinto getInstancia() {
        if (unicaInstancia == null) return getInstancia ("encantado");
        else return unicaInstancia;
    }

    // invocar sólo la primera vez
    public static FactorialLaberinto getInstancia(String nombre) {
        if (unicaInstancia == null)
            if (nombre.equals("encantado"))
                unicaInstancia = new FactorialLaberintoEncantado();
            else if (nombre.equals("normal"))
                unicaInstancia = new FactorialLaberintoNormal();
        return unicaInstancia;
    }
    protected FactorialLaberinto () { }
    ...
}
```



## Singleton (Con subclases): Alternativa

Cada subclase proporciona un método getInstancia()

```
public abstract class FactorialLaberinto {
    protected static FactorialLaberinto unicaInstancia = null;
    public static FactorialLaberinto getInstancia() {
        return unicaInstancia;
    }
    protected FactorialLaberinto () { }
    ...
}

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

## Tema 3: Patrones de Diseño (Parte 2. Patrones Estructurales)

### Patrones Estructurales

- ✓ Describen **formas de combinar objetos** para obtener una funcionalidad que resuelve problemas recurrentes en programación orientada a objetos.
- Basadas en la composición.
- Posibilidad de cambiar la estructura en tiempo de ejecución.

### Adapter / Wrapper (Adaptador)

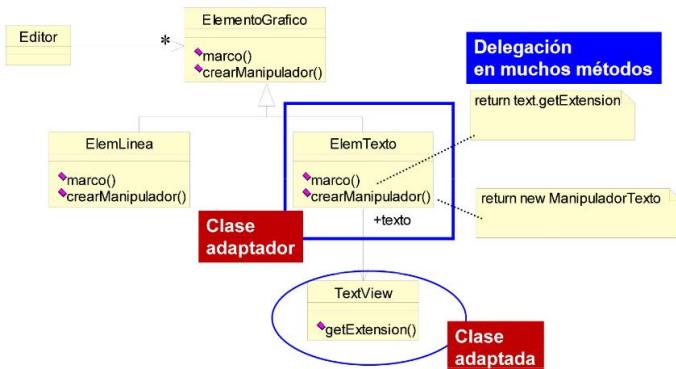
#### Propósito

Convertir la interfaz de una clase en otra que una clase cliente espera. Permite la colaboración de ciertas clases a pesar de tener interfaces incompatibles

#### Motivación (1)

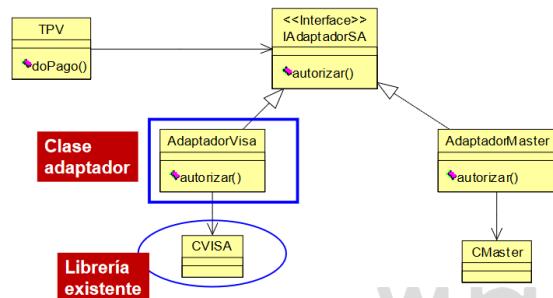
Un editor gráfico incluye una jerarquía que modela elementos gráficos (líneas, polígonos, texto,...) y se desea reutilizar una clase existente TextView para implementar la clase que representa elementos de texto, pero que tiene una interfaz incompatible con la raíz de la jerarquía.

## Adaptador: Motivacion (1). Reuso

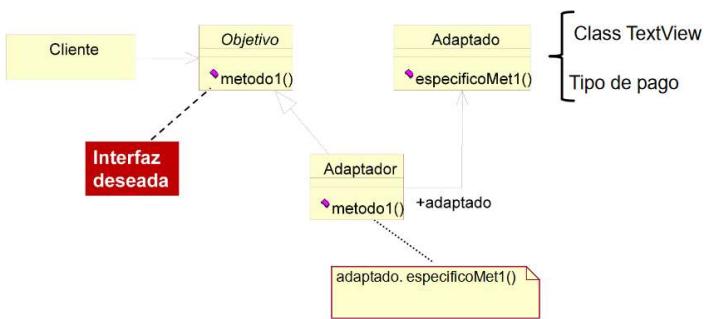


## Adaptador: Motivacion (2). Variación en tipos

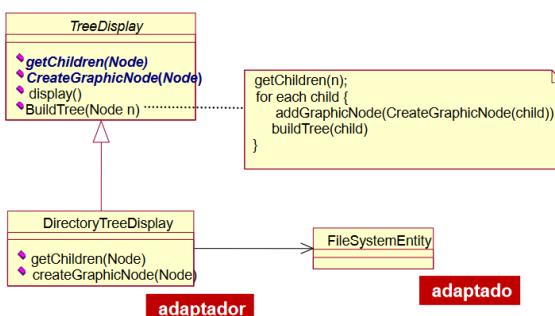
En una aplicación de TPV se deben soportar diferentes tipos de servicios externos de realización de pagos o de inventario y se quiere ser **independiente de un servicio concreto**.



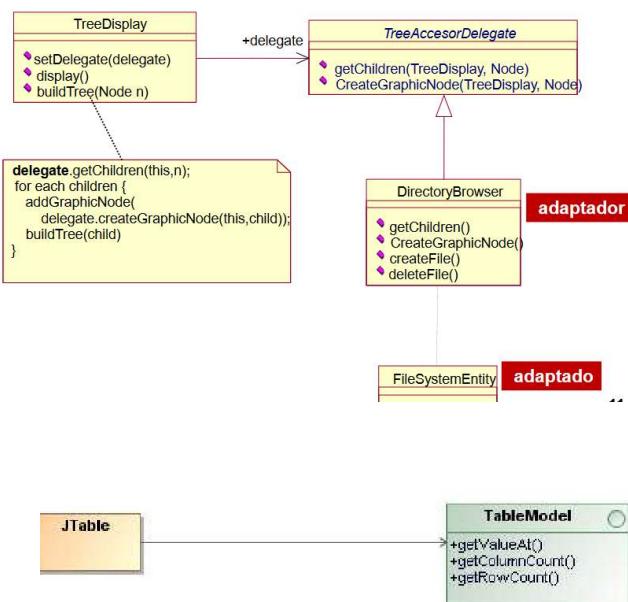
## Adaptador: Estructura



**TreeDisplay** es una clase abstracta y los adaptadores son subclases (ejemplo en “Design Patterns”)



**TreeDisplay** es una clase concreta que delega en un interfaz que implementan los adaptadores



### Adaptador en 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 aislan 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 (Similar a Treedisplay basado en composición).

### Adaptador en Swing: TableModel y ListModel

```

public interface ListModel {
    int getSize();
    Object getElementAt(int index);
}

public interface TableModel {
    // obtener número de filas
    public int getRowCount();

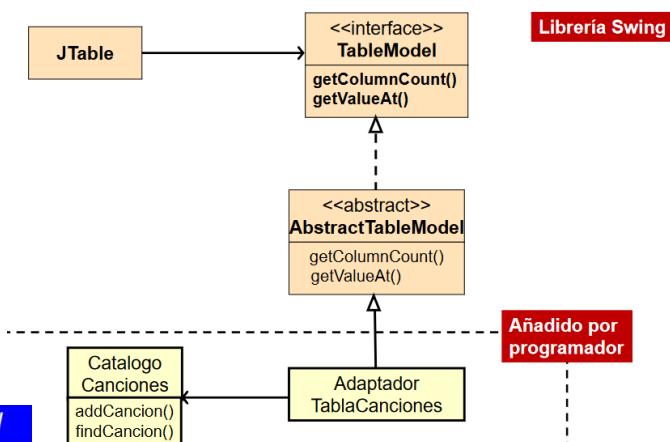
    // obtener número de columnas
    public int getColumnCount();

    // obtener valor del elemento fila fil y columna col
    public Object getValueAt(int fil, int col);

    // obtener nombre de la columna col
    public String getColumnName(int col);
}

Adaptador en Swing: TreeModel
  
```

### Adaptador en Swing: JTable



```

public interface TreeModel {
    // retorna nodo raíz
    public Object getRoot();

    // retorna nodo hijo del padre en posición index
    public Object getChild(Object parent, int index);

    // retorna número de hijos del nodo padre
    public int getChildCount(Object parent);

    // retorna true si es un nodo hoja
    public boolean isLeaf(Object node);
}
  
```

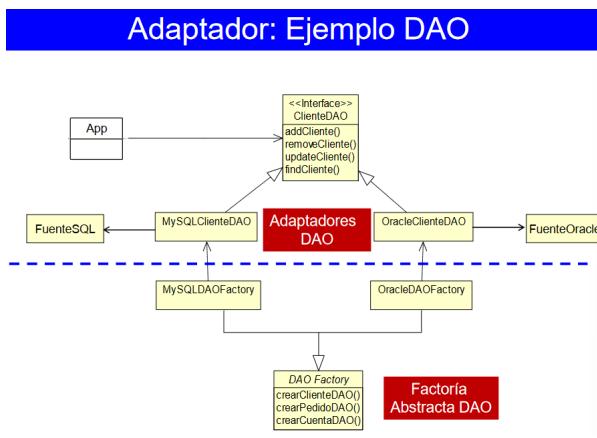


# LA NUESTRA DURA MÁS

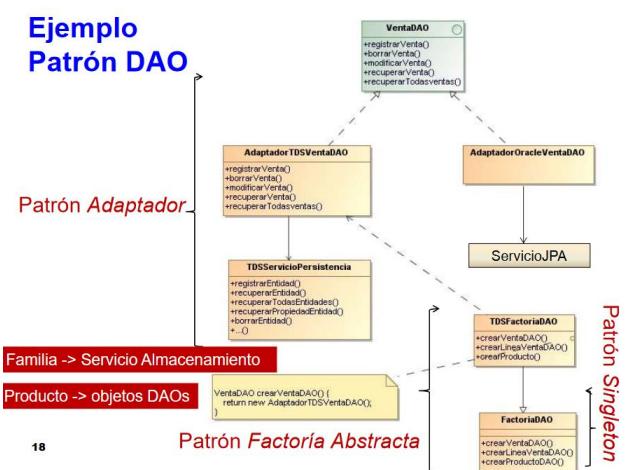


## Patrón DAO: Adaptador + Factoría Abstracta + Singleton

- El patrón DAO (Data Access Object) se utiliza para conseguir que una aplicación sea independiente del sistema de almacenamiento utilizado (p.e., un sistema relacional de uno u otro fabricante, o JDBC, o una base de datos MongoDB, etc.).
- Para cada clase del negocio (por ejemplo Cuenta y Cliente) se crea una interfaz DAO que proporciona los métodos CRUD para crear, recuperar, modificar y eliminar datos almacenados.
- La interfaz DAO es implementada por clases que se encargan de establecer las conexiones a una fuente de datos concreta y encapsular cómo se accede a dicha fuente.
- El patrón DAO utiliza una factoría abstracta para crear las instancias de las clases DAO que requiere la aplicación, para conseguir que ésta sea independiente de un sistema concreto de almacenamiento.



## Ejemplo Patrón DAO



18

## Bridge / Handle

### Propósito

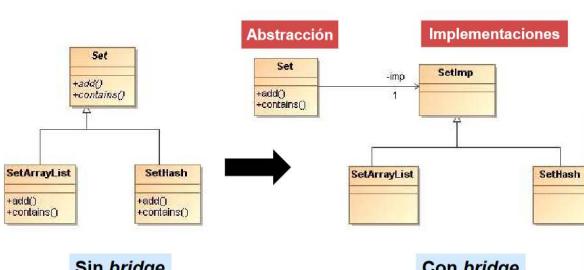
Desacoplar una abstracción de su implementación, de modo que los dos puedan cambiar independientemente.

### Motivación

- Sea una clase que modela una abstracción (p.e., una estructura de datos o una ventana).
- Si se usa herencia para definir subclases que implementan la abstracción, probablemente de distintas formas, entonces es difícil reutilizar abstracciones e implementaciones de forma independiente
  - Si refinamos la abstracción en una nueva subclase, está tendrá tantas subclases-implementación como tenía su superclase.
  - El código cliente es dependiente de la plataforma.

## Bridge: Motivación (1)

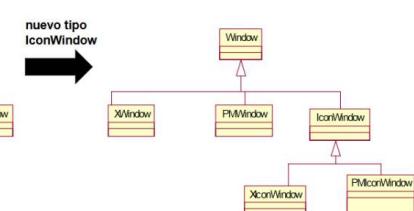
El tipo *conjunto* puede ser implementado de distintas maneras.



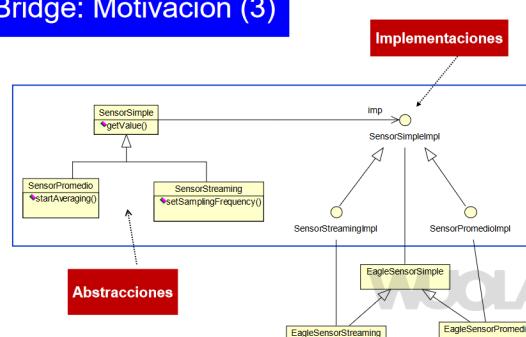
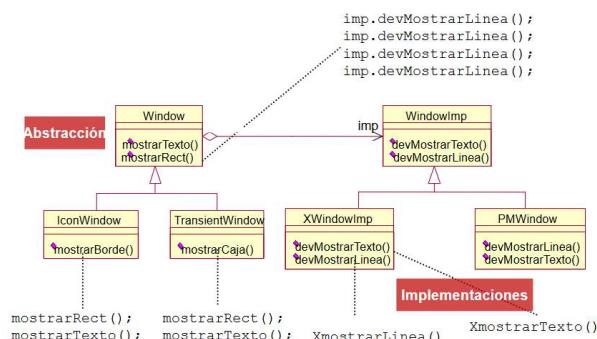
## Bridge: Motivación (2)

Ejemplo en libro GoF

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



## Bridge: Motivación (3)



## Bridge

### v 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.
- ♦ Se tiene una proliferación de clases.

### v Consecuencias

- ♦ Un objeto puede cambiar su implementación en tiempo de ejecución.
- ♦ Se mejora la extensibilidad.
- ♦ Se ocultan detalles de implementación a los clientes.

## Bridge. Ejemplo en Java AWT

- ♦ Java 1.1 AWT (Abstract Window Toolkit) fue diseñado para proporcionar una interfaz GUI en un entorno heterogéneo.
- Component es una clase abstracta que encapsula la lógica común a todos los componentes GUI y tiene subclases como Button, List y TextField que encapsulan la lógica para estos componentes GUI de forma independiente de una plataforma concreta (Sun, 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.

Cada clase componente (por ejemplo Button) tiene un atributo **peer** que se inicializa del siguiente modo:

```
ButtonPeer peer =
    Toolkit.getDefaultToolkit().createButton(this);
```

AWT utiliza una **fábrica abstracta** (**Toolkit** es la clase raíz) para crear los componentes **peer** de la plataforma concreta que sea usada.

```
public static Toolkit getDefaultToolkit() {
    String nm;
    Class cls;
    if (toolkit == null) {
        try {
            nm = "obtener nombre de la subclase de Toolkit";
            try {
                cls = Class.forName(nm);
            } catch (ClassNotFoundException e){...}
            if (cls != null) toolkit = (Toolkit) cls.newInstance();
        } catch (Exception e) {}
    }
    return toolkit;
}
```

## Composite

### v Propósito

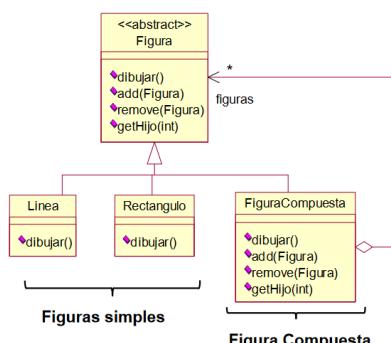
- ♦ Componer objetos en estructuras jerárquicas para representar jerarquías “parte/todo”. Permite al código cliente manejar a los objetos primitivos y compuestos de forma uniforme.

### v Motivación

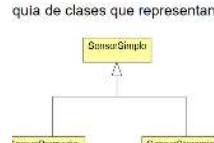
- ♦ Representar figuras compuestas: una figura compuesta es una figura formada por figuras primitivas y compuestas
- ♦ Representar circuitos compuestos: un circuito compuesto es un circuito formado por circuitos primitivos y compuestos.
- ♦ Representar carteras de valores: un paquete es un valor formado por valores simples (acciones, bonos,...) y por paquetes

## Composite: Motivación (1)

### Figuras compuestas



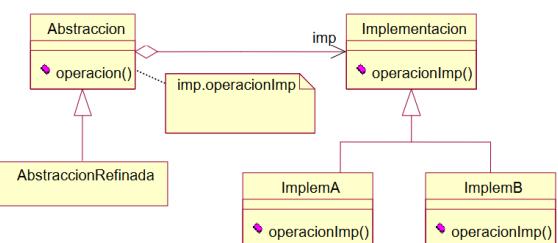
que de clases que representan tipos de se



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

**¡Explosión de clases con herencia!**

## Bridge: Estructura

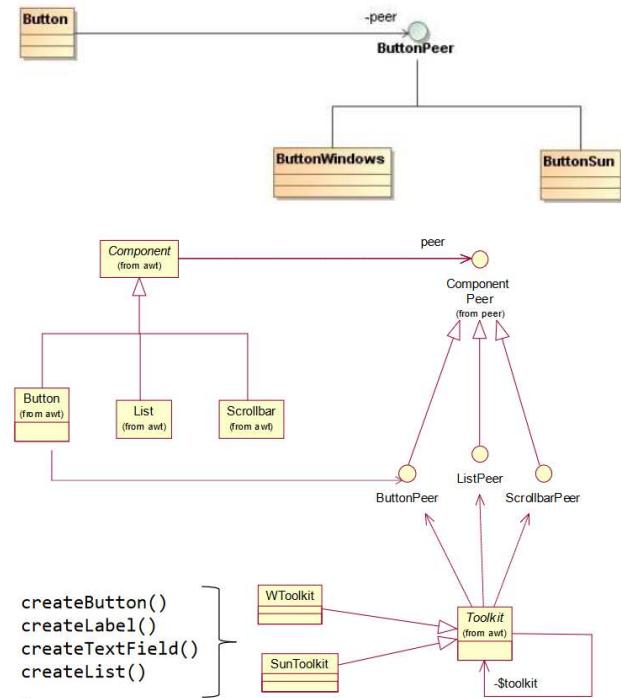


## Bridge: Implementación

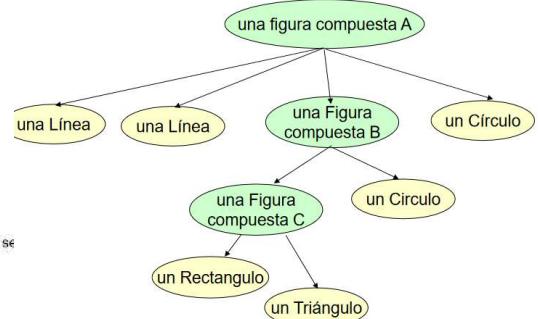
### ¿Cómo, cuándo y dónde se decide qué implementación usar?

- Constructor de la clase **Abstraccion** + método `setImpl()`
- Elegir una **implementación por defecto** + método `setImpl()`
- Delegar a otro objeto, por ejemplo un objeto **fábrica**

Component es una clase abstracta que encapsula la lógica común a todos los componentes GUI y tiene subclases como Button, List y TextField que encapsulan la lógica para estos componentes GUI de forma independiente de una plataforma concreta (Sun, W32, GTK,...).

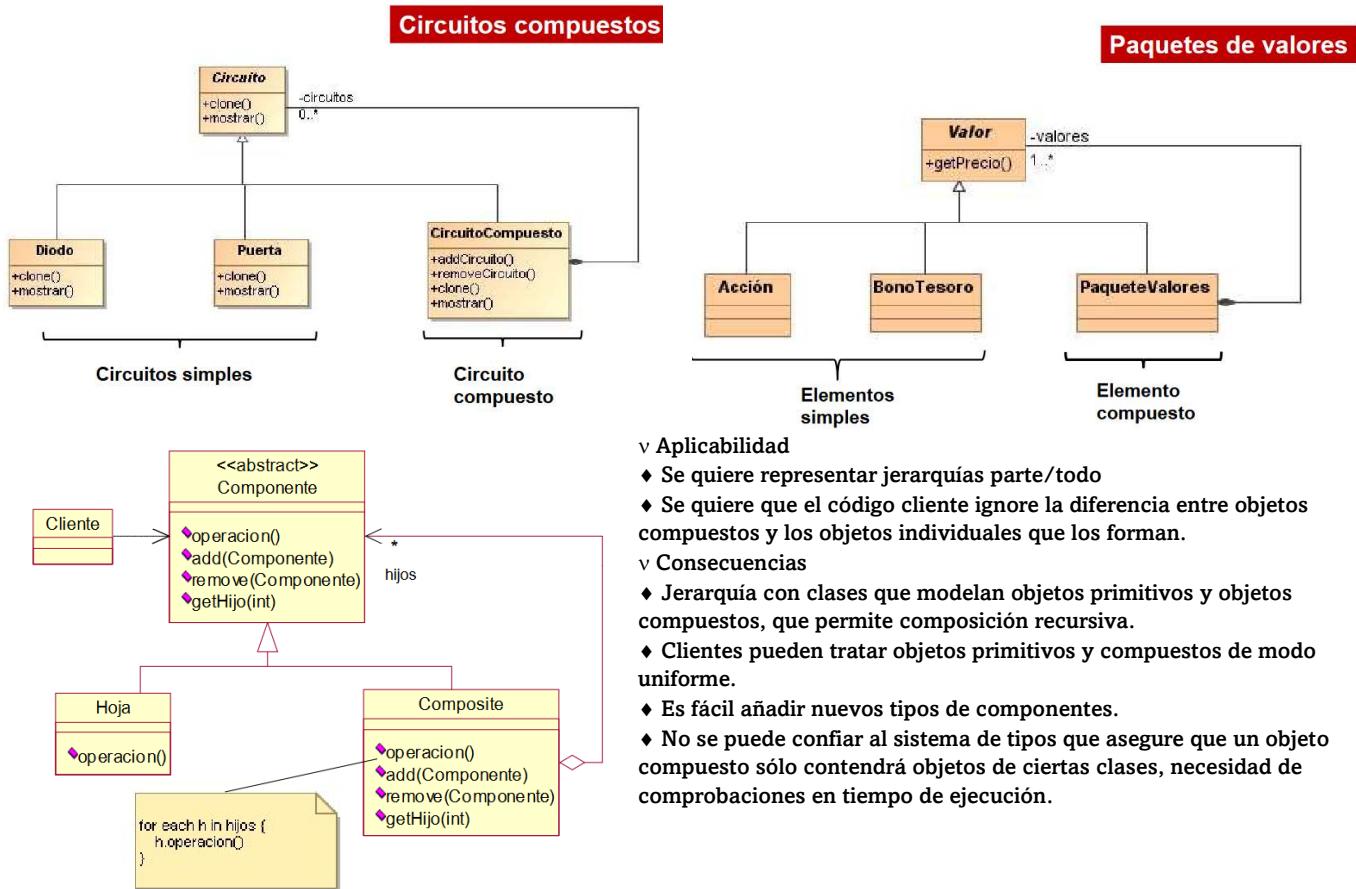


## Composición recursiva



## Composite: Motivación (2)

## Composite: Motivación (3)



### v Aplicabilidad

- ♦ Se quiere representar jerarquías parte/todo
- ♦ Se quiere que el código cliente ignore la diferencia entre objetos compuestos y los objetos individuales que los forman.

### v 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.

### Transparencia

```

Figura figuraActual;
public void manejarFigura (Figura nuevaFigura) {
    // figuraActual debe ser compuesta si no excepción es lanzada
    figuraActual.addFigura(nuevaFigura)
    ...
}
  
```

### Seguridad

```

Figura figuraActual;
public void manejarFigura (Figura nuevaFigura) {
    if (figuraActual instanceof FiguraCompuesta)
        (FiguraCompuesta)figuraActual.addFigura(nuevaFigura)
    ...
}
  
```



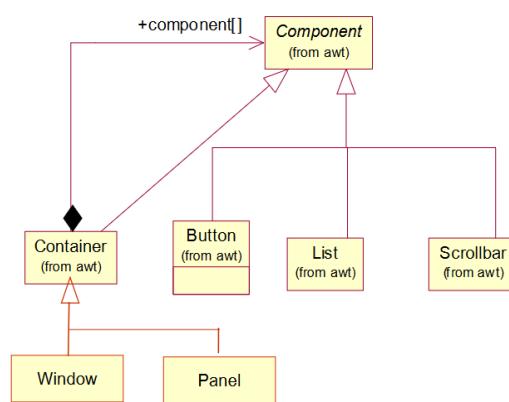
# SUDADERAS PARA GRUPOS PERSONALIZADAS



**KRONIX**  
Sudaderas



Diseñar Ahora



## Decorador (Decorador)

### v Propósito

- ◆ Asignar dinámicamente nuevas responsabilidades a un objeto. Alternativa más flexible a crear subclases para extender la funcionalidad de una clase.

### v 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.

```

public class Container extends Component {..}
public Component add (Component c) {...}
public void remove (Component c) {...}
}
  
```

Role  
Composite

```

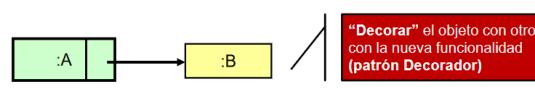
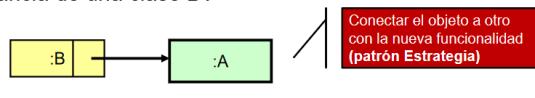
public class Window extends Container {
    Component[] components;
    public void update() {
        if (components != null)
            for (int k = 0; k < components.length; k++)
                components[k].update();
    }
}
  
```

En la clase composite las operaciones heredadas normalmente se implementan recorriendo los elementos contenidos y aplicando la operación.

## Decorador

## Decorador: Motivación

¿Cómo extender dinámicamente la funcionalidad de una instancia de una clase B?



y siempre que espero un B debería poder recibir un objeto de la clase A



Dos decoradores sobre un objeto TextView

### v Aplicabilidad

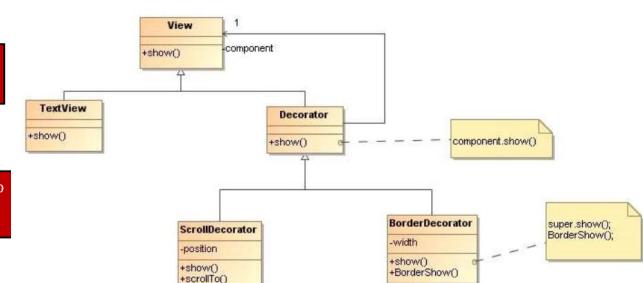
- ◆ Añadir dinámicamente responsabilidades a objetos individuales de forma transparente, sin afectar a otros objetos.
- ◆ Evitar una explosión de clases.
- ◆ Más flexible que la herencia: responsabilidades pueden añadirse y eliminarse en tiempo de ejecución.
- ◆ Diferentes decoradores pueden ser conectados 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.
- ◆ Puede dar lugar a aplicaciones con muchos y pequeños objetos

### Implementación:

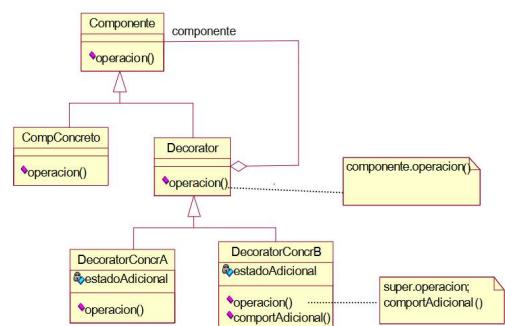
v Componentes y decoradores deben heredar de una clase común que debe ser "ligera" en funcionalidad. Si no es ligera es mejor usar el patrón Estrategia.

### v Diferencias entre Decorador y Estrategia:

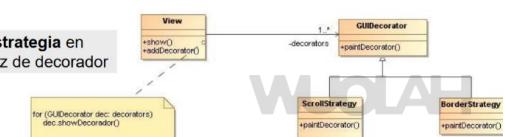
- ◆ Un componente no sabe nada acerca de sus decoradores, con Estrategia sucede lo contrario.



## Decorador: Estructura



Estrategia en vez de decorador

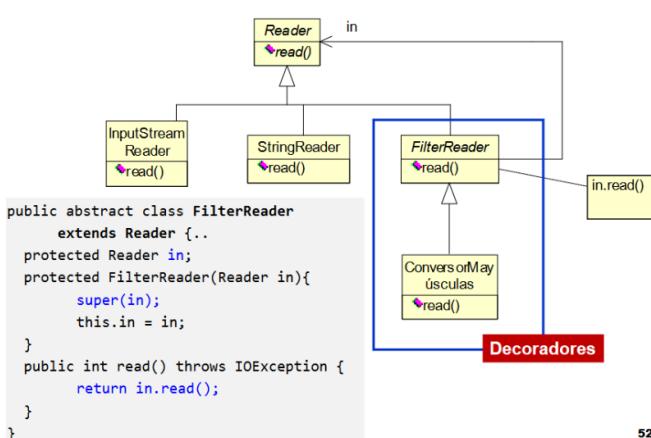


- ♦ El decorador envía los mensajes al componente que decora, pudiendo extender la operación con nuevo comportamiento.
- ♦ Las clases que modelan los decoradores concretos pueden añadir responsabilidades a las que heredan de la clase Decorador.

#### Decorador: Ejemplo filtros de streams (entrada/salida) en 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.

#### Decorador: Filtros de streams en 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;
    }
}

```

52

#### Facade (Fachada)

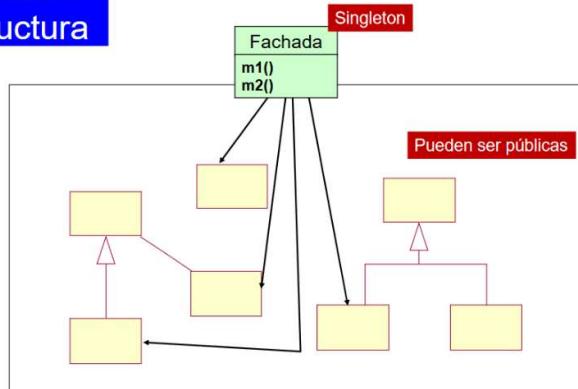
##### v Propósito

- ♦ Proporciona una única interfaz a un conjunto de clases de un subsistema que ofrece una funcionalidad. Define una interfaz de más alto nivel que facilita el uso de un subsistema.

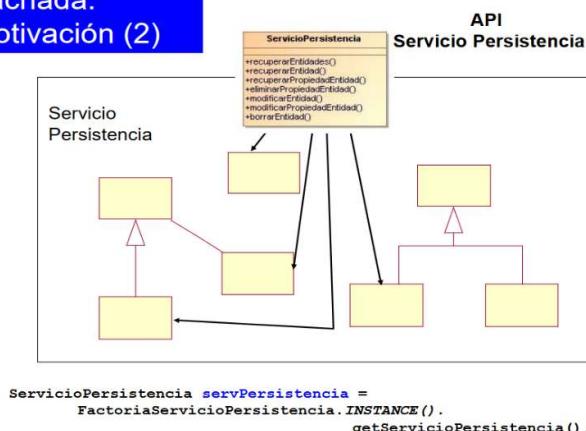
##### v Motivación

- ♦ Reducir las dependencias (acoplamiento) entre subsistemas.
- ♦ (1) Un entorno de programación que ofrece una librería de clases que proporcionan acceso a su subsistema compilador: Scanner, Parser, ProgramNode, ByteCodeStream y ProgramNodeBuilder. La clase Compiler actúa como fachada.
- ♦ (2) Servicio Persistencia utilizado en el caso práctico de la asignatura
- ♦ (3) Controlador entre la capa GUI y la capa de negocio.

#### Fachada: Estructura



#### Fachada: Motivación (2)

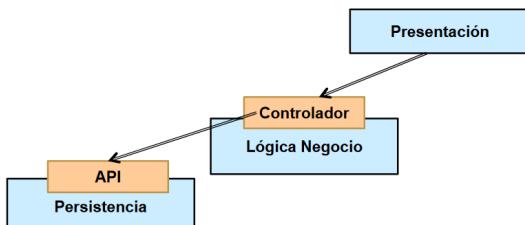


##### v Consecuencias

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

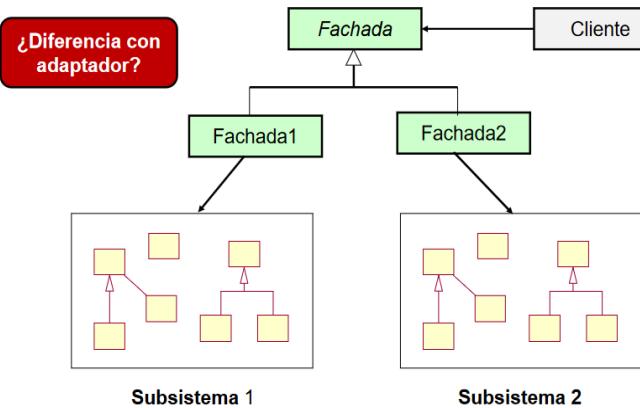
##### v Implementación

- 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++)
- Una fachada es normalmente un Singleton.



## Fachada: Jerarquía

Diferentes implementaciones del subsistema

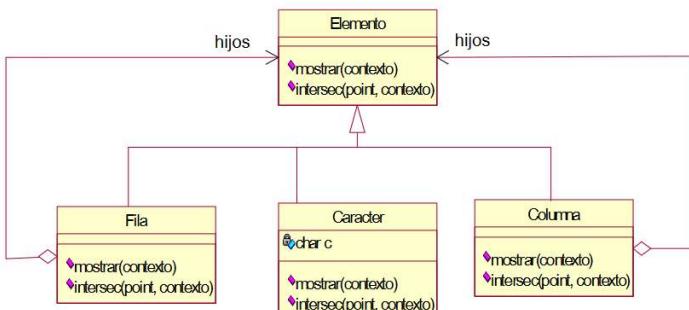


- ◆ Objetos clientes son responsables de pasar el estado extrínseco al flyweight cuando lo necesita.
- ✓ Objetos flyweight se usan para modelar conceptos o entidades de los que se necesita una gran cantidad en una aplicación, por ejemplo los caracteres de un texto.

## Flyweight: Ejemplo Carácter

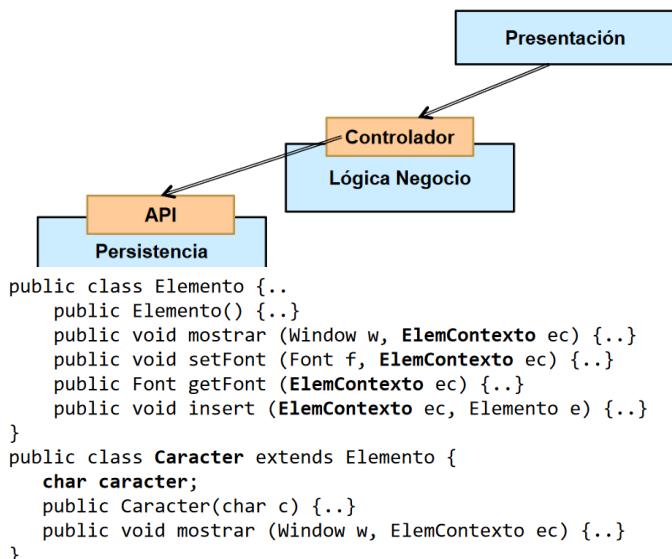
### Flyweight carácter de un texto:

- estado intrínseco: código del carácter
- estado extrínseco: información sobre posición y formato

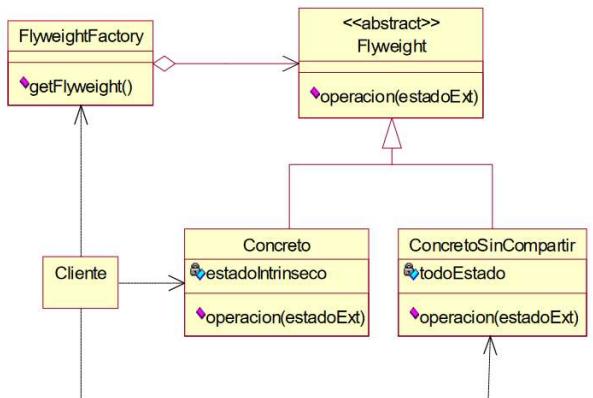


## Fachada: Aplicabilidad

- 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.



## Flyweight: Estructura



```

Flyweight getFlyweight(key) {
    if "existe (flyweightPool[key])"
        return "flyweight existente";
    else {
        "crear nuevo flyweight";
        "añadirlo al pool de flyweights";
        return "nuevo flyweight creado";
    }
}
  
```

FlyweightFactory es un pool de objetos Flyweight  
Aplicarlo siempre que se cumplan las siguientes condiciones:

```

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 () {...}
}
  
```

Repository de estado extrínseco

# Si estás en tu **spending era...**

mejor tener una app que te diga en qué tiendas se ha quedado registrada tu tarjeta.

¡Como la app de ING!

Saber más



- 1. Una aplicación utiliza un gran número de objetos de cierto tipo.
- 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.

#### v 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.
- v 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 flyweights son compartidos, no deberían ser instanciados directamente por los clientes: uso de una factoría

## Flyweight. Ejemplos en Java

Strings

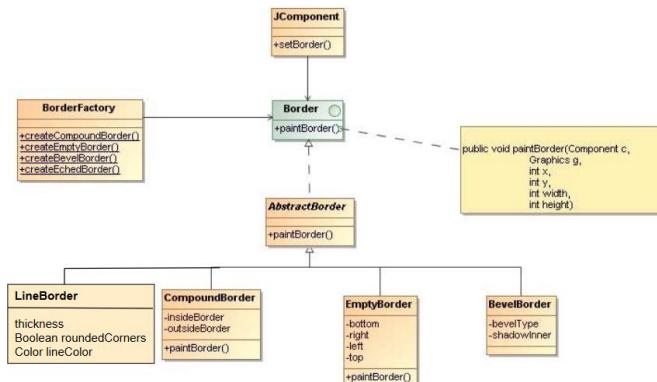
```
new String("TDS") == new String ("TDS") --> False  
new String("TDS").equal(new String ("TDS")) --> True
```

Strings son almacenados en un pool de objetos compartidos (literales)

```
"hola" == "hola" --> true  
"ho"+ "la" == "hola" --> false
```

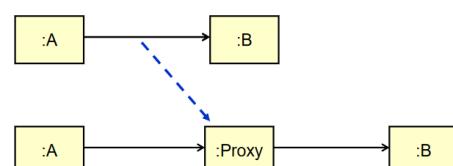
Método `intern()` en clase `String` recupera string del pool, si no existe lo registra y siempre retorna su referencia. Favorece uso de `==` en vez de `equals`

```
String c = ("ho" + "la").intern  
c == "hola" --> true
```



## Proxy

I Se convierte una referencia en un objeto que conforma con el objeto sustituido (smart reference).



El objeto proxy tiene la **misma interfaz** como el objeto destino.

El objeto proxy mantiene una referencia al objeto destino y puede pasársela a él los mensajes recibidos (delegación).

## Flyweight. Ejemplos en Java

Bordes de components Swing

```
public class BorderFactory {  
    public static Border createLineBorder();  
    public static Border createCompoundBorder()  
    ...  
}  
  
JPanel panel = new JPanel();  
panel.setBorder(BorderFactory.createLineBorder(Color.black));
```

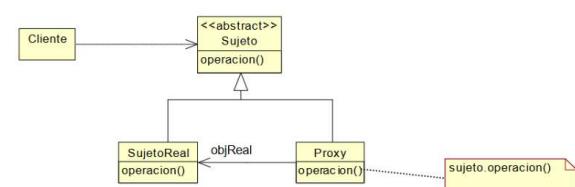
line border

```
public void paintBorder(Component c, Graphics g, int x, int y, int width, int height)
```

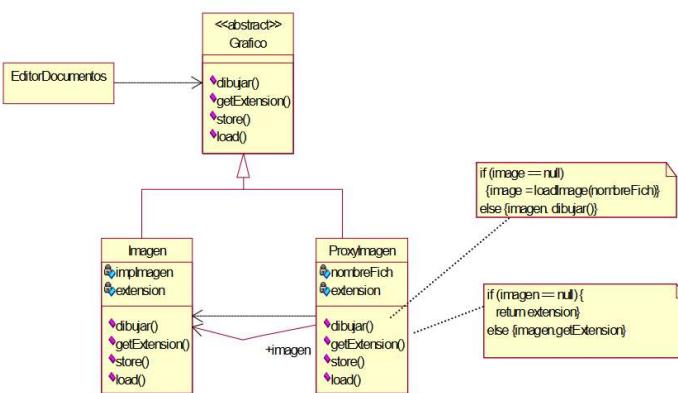
#### Estado extrínseco

#### Proxy (Sustituto)

- v Propósito
  - ♦ Proporcionar un sustituto (surrogate) de un objeto para controlar el acceso a dicho objeto.
- v Motivación
  - ♦ (1) 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.
  - ♦ (2) Hay situaciones en las que un objeto cliente no referencia o no puede referenciar a otro objeto directamente, pero necesita interactuar con él (p.e. arquitectura distribuida).
  - ✓ Un objeto proxy puede actuar como intermediario entre el objeto cliente y el objeto destino.



# Proxy: Motivación (1)



demanda.

3. Un proxy para protección o las referencias inteligentes permiten realizar tareas de control sobre los objetos accedidos.

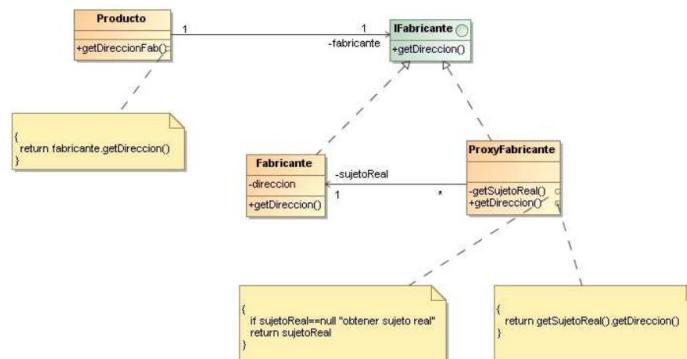
✓ Ejemplos

◆ Mantenimiento de los servicios ante fallos.

◆ Materialización perezosa de tuplas en objetos.

◆ 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.

## Proxy para materialización perezosa (ver texto "UML y Patrones", Craig Larman)

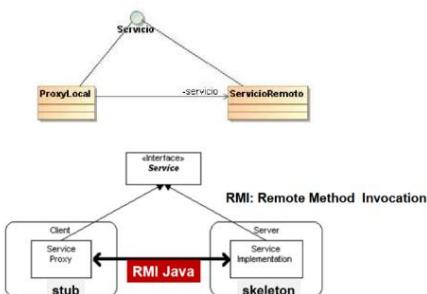


```
public class LazyCloneMap implements Map, Cloneable {
    private Map underlyingMap;
    private MutableInteger refCount;

    private Method cloneMethod;
    private static Class[] cloneParams = new Class[0];

    public LazyCloneMap(Map aMap)
        throws NoSuchMethodException, InvocationTargetException {
        Class mapClass = aMap.getClass();
        cloneMethod = mapClass.getMethod("clone", cloneParams);
        try {
            this.underlyingMap = (Map) cloneMethod.invoke(aMap, null);
        } catch (IllegalAccessException e) {
        }
    }
    refCount = new MutableInteger(1);
}
```

## Proxy remoto



## Aplicabilidad

✓ Siempre que hay necesidad de referenciar a un objeto mediante una referencia más "rica" que un puntero o una referencia normal.

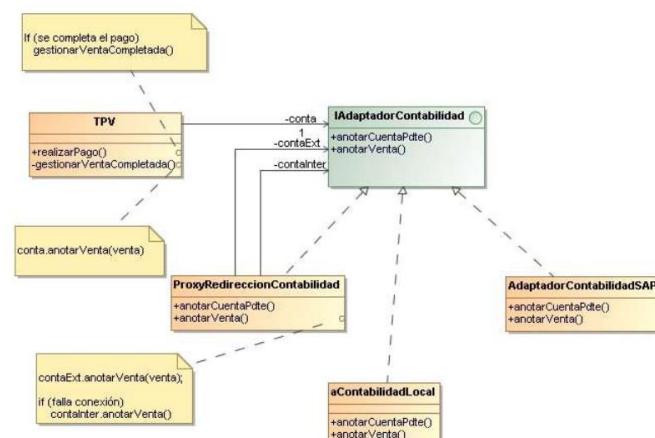
✓ Situaciones más comunes:

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)

✓ 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.



## Proxy para clonación perezosa

✓ Una razón para clonar un objeto Map es evitar mantener un bloqueo sobre la colección un largo tiempo, si sólo se desean realizar operaciones de consulta.

✓ Utilizar métodos synchronized para obtener el acceso exclusivo puede resultar inaceptable en algunas situaciones.

✓ Algunas clases que implementan Map permiten la clonación, como HashMap y TreeMap, pero una clonación previa puede ser innecesaria, mejor aplicar una clonación perezosa y sólo clonar cuando es necesario.

```
public Object clone() {
    LazyCloneMap theclone;
    try {
        theClone = (LazyCloneMap) super.clone();
    }
    catch (CloneNotSupportedException e) {
        theClone = null;
    }
    refCount.setValue(1+refCount.getValue());
    return theClone;
}

private void ensuredUnderlyingMapNotShared() {
    if (refCount.getValue() > 1)
        try {
            underlyingMap = (Map) cloneMethod.invoke(underlying, null);
            refCount.setValue(refCount.getValue() - 1);
            refCount = new MutableInteger(1);
        } catch (...) }
```

```

public int size() {
    return underlyingMap.size();
}
public boolean isEmpty() {
    return underlyingMap.isEmpty();
}
public Object get(Object key) {
    return underlyingMap.get(key);
}
public void put(Object key, Object value) {
    ensureUnderlyingMapNotShared();
    return underlyingMap.put(key, value)
}
public void remove(Object key) {
    ensureUnderlyingMapNotShared();
    return underlyingMap.remove(key)
}
} // class LazyCloneMap

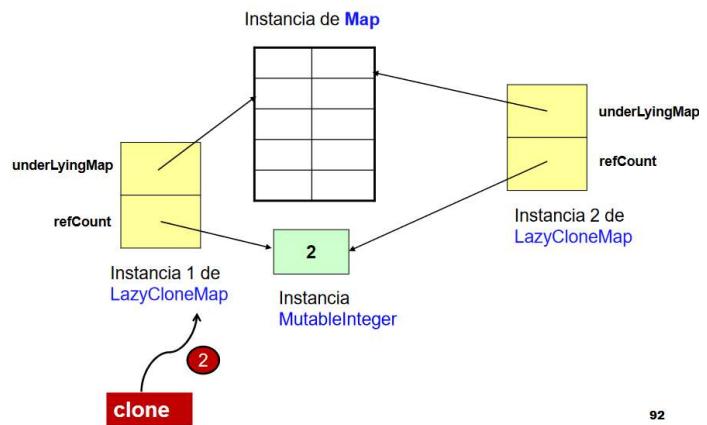
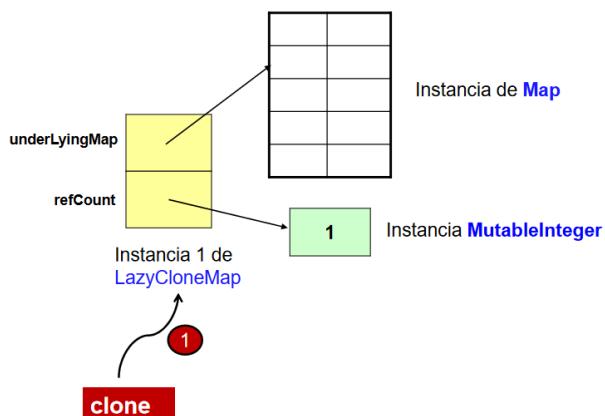
```

```

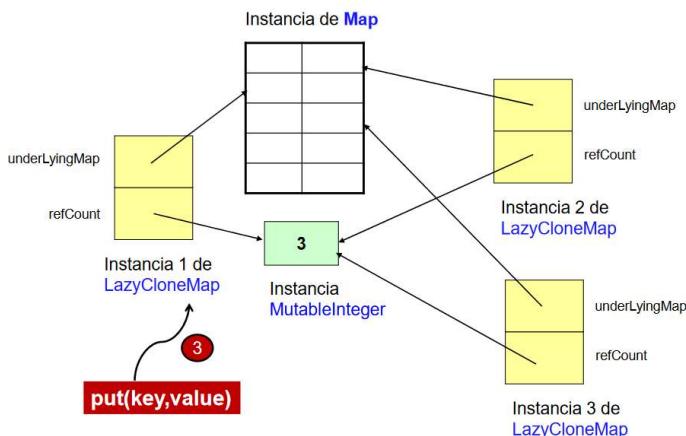
public class MutableInteger {
    private int val;
    public MutableInteger (int value) {
        setValue (value);
    }
    public MutableInteger getValue() {
        return val;
    }
    public void setValue(int value) {
        val = value
    }
    ...
}

```

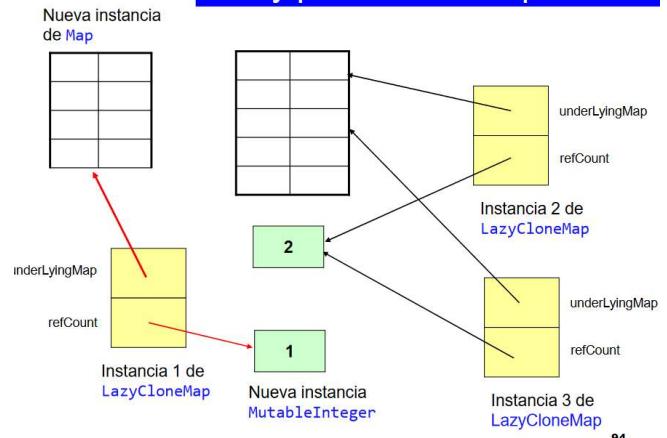
## Proxy para clonación perezosa



92



## Proxy para clonación perezosa



94

Portátiles desde

549€



msi  
BLACK FRIDAY

### Tema 3: Patrones de Diseño (Parte 3. Patrones de Comportamiento)

#### Patrones de comportamiento

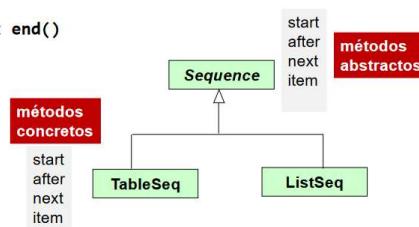
- ✓ Enfatizan la colaboración entre objetos.
- ✓ Caracterizan un flujo de control más o menos complejo que será transparente en el uso del patrón.
- ✓ Basados en composición excepto Template Method que basa en la herencia.

#### Template Method (Método Plantilla)

- ✓ Propósito
  - ♦ Definir operaciones como esquemas de algoritmos que difieren algunos pasos a operaciones implementadas en las subclases.
  - Permite a las subclases redefinir ciertos pasos de un algoritmo sin cambiar la estructura del algoritmo.
- ✓ Motivación
  - ♦ Esencial para escribir código en frameworks.
  - ✓ Clase abstracta Aplicación que maneja objetos de la clase abstracta Documento: método OpenDocument, puede incluir métodos basados en métodos definidos en sus subclases
- ♦ Técnica fundamental para la reutilización en jerarquías de clases: factorizar comportamiento común.

#### Método Plantilla: Esquema algorítmico

```
class Sequence<T>{..  
    // es código Eiffel, lenguaje creado antes de Java  
    has(v:T): boolean {  
        from start()  
        until after() or else item().equal(v)  
        loop next()  
        end  
        Result = not end()  
    }  
}
```



#### Método Plantilla

- ✓ Aplicabilidad
  - ♦ Una clase implementa el esquemas de un algoritmo y deja 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 (refactoring)
- ✓ Consecuencias
  - ♦ Un método plantilla (template) invoca a los siguientes tipos de métodos:
    - ✓ operaciones abstractas
    - ✓ operaciones concretas en la clase abstracta o en clientes
    - ✓ métodos factoría

#### Método Plantilla. Ejemplo java.util

```
public abstract class AbstractCollection<E> implements Collection<E> {  
  
    public abstract Iterator<E> iterator();  
    public abstract int size();  
  
    public boolean contains(Object o) {  
        Iterator<E> it = iterator();  
        if (o==null) {  
            while (it.hasNext())  
                if (it.next()==null)  
                    return true;  
        } else {  
            while (it.hasNext())  
                if (o.equals(it.next()))  
                    return true;  
        }  
        return false;  
    }  
}
```

#### Método Plantilla y expresiones lambda

#### Método Plantilla y expresiones lambda (Java 8 Lambdas, Richard Warburton, 2014)

```
public abstract class LoanApplication {  
  
    public void checkLoanApplication() throws ApplicationDenied {  
        checkIdentity();  
        checkCreditHistory();  
        checkIncomeHistory();  
        reportFindings();  
    }  
  
    protected abstract void checkIdentity() throws ApplicationDenied;  
    protected abstract void checkIncomeHistory() throws ApplicationDenied;  
    protected abstract void checkCreditHistory() throws ApplicationDenied;  
    private void reportFindings() {...}  
}
```

LoanApplication no usa expresiones lambda  
sino método plantilla.

```
public class LoanApplication {  
    private final Criteria identity;  
    private final Criteria creditHistory;  
    private final Criteria incomeHistory;  
  
    public LoanApplication(Criteria identity,  
                          Criteria creditHistory,  
                          Criteria incomeHistory) {  
        this.identity = identity;  
        this.creditHistory = creditHistory;  
        this.incomeHistory = incomeHistory;  
    }  
    public void checkLoanApplication() throws ApplicationDenied {  
        identity.check();  
        creditHistory.check();  
        incomeHistory.check();  
        reportFindings();  
    }  
    private void reportFindings() {...}  
}
```

Criteria es una  
interfaz funcional

```
public interface Criteria {  
    public void check() throws ApplicationDenied;  
}  
  
public class CompanyLoanApplication extends LoanApplication {  
    public CompanyLoanApplication(Company company) {  
        super(company::checkIdentity,  
              company::checkHistoricalDebt,  
              company::checkProfitAndLoss);  
    }  
}
```

11



## Strategy/Policy (Estrategia)

### v Propósito

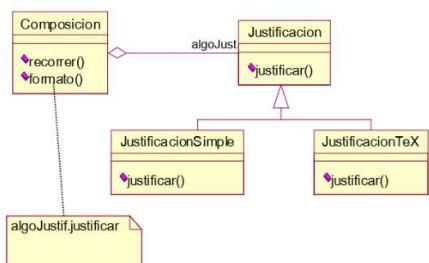
- ♦ Define una familia de algoritmos, encapsula cada uno, y permite intercambiarlos. Permite variar los algoritmos de forma independiente a los clientes que los usan

### v Motivación

- ♦ Existen muchos algoritmos para justificación de texto, ¿debe implementar un algoritmo el cliente que lo necesita?
- ♦ Cómo implementar las políticas de descuento en una aplicación de ventas.

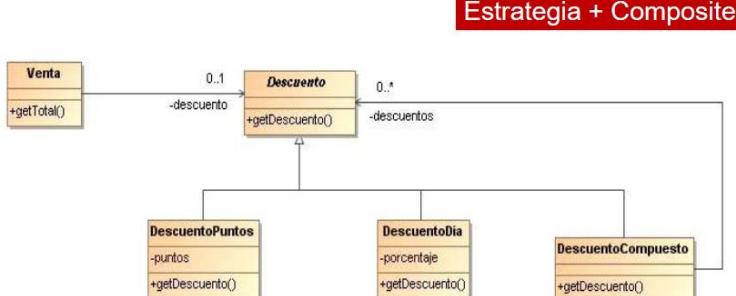
## Estrategia. Ejemplo “Justificación de texto”

DP GoF

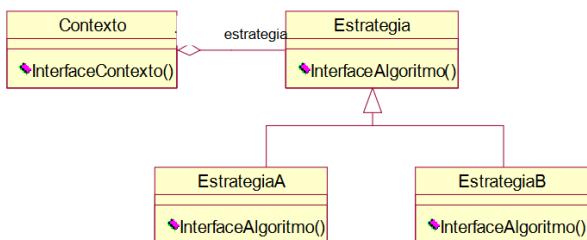


## Estrategia. Ejemplo “Política de Descuentos”

UML y Patrones, Craig Larman



## Estrategia. Estructura



## Estrategia: Implementación

- ¿Cómo una estrategia concreta accede a los datos del contexto?
  - Pasar datos o un objeto Contexto como argumentos
  - Estrategia almacena una referencia al contexto
- ¿Cómo se crea una instancia de una estrategia concreta?
  - Uso de una factoría

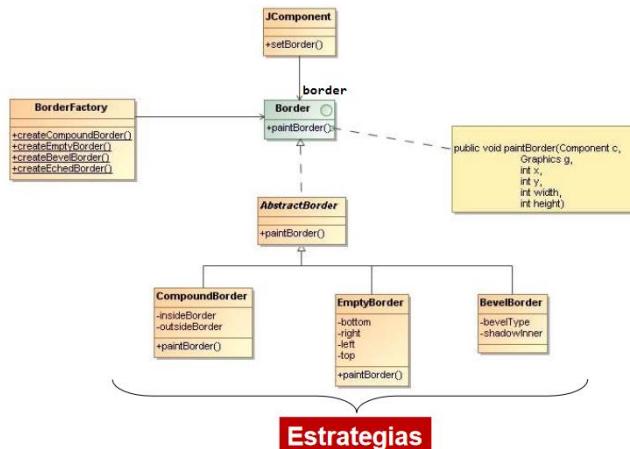
### v Aplicabilidad

- ♦ 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 (refactoring).

### v Consecuencias

- ♦ Define una familia de algoritmos relacionados.
- ♦ Una alternativa a crear subclases de la clase Contexto.
- ♦ Elimina sentencias case
- ♦ En el código cliente se puede elegir entre diferentes estrategias o implementaciones: debe conocer detalles
- ♦ Estado y Estrategia son similares, cambia el Propósito: ejemplos de composición con delegación

## Estrategia. Ejemplo de Bordes en Swing

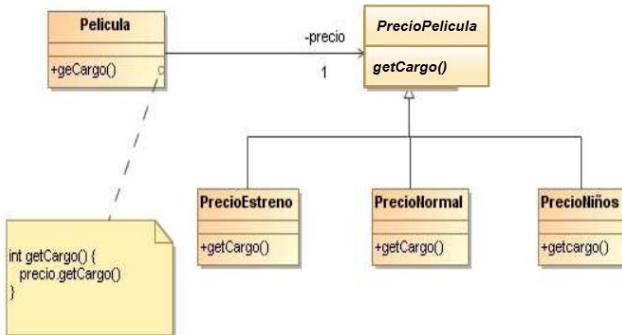


### Cambiar switch por patrón Estrategia

```

cantidadTotal = 0;
for(Alquiler alq:_alquileres) {
    double unaCantidad = 0;
    //determinar cantidad para cada linea
    switch (alq.getCodigoPrecio()) {
        case Pelicula.NORMAL:
            unaCantidad += 2;
            if (alq.getDiasAlquiler() > 2)
                unaCantidad += (alq.getDiasAlquiler()-2)*1.5;
            break;
        case Pelicula.ESTRENO:
            unaCantidad += alq.getDiasAlquiler() * 3;
            break;
        case Pelicula.NIÑOS:
            unaCantidad += 1.5;
            if (alq.getDiasAlquiler() > 3)
                unaCantidad += (alq.getDiasAlquiler()-3)*1.5;
            break;
    }
    cantidadTotal = cantidadTotal + unaCantidad;
}
  
```

## Cambiar switch por patrón Estrategia



## Estrategia "Precios Viaje"

```

public interface ReglaPrecioViaje {
    public double getPrecio(Viaje viaje);
}

public class Viaje {
    int distancia;
    Confort tipoCoche;
    ReglaPrecioViaje reglaPrecio;

    public double calcularPrecio() {
        return reglaPrecio.getPrecio(this);
    }

    public void setReglaPrecio(ReglaPrecioViaje regla) {
        this.reglaPrecio = regla;
    }
}
// métodos getters y setters para distancia y tipoCoche
  
```

Estrategia es una interfaz funcional

### 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

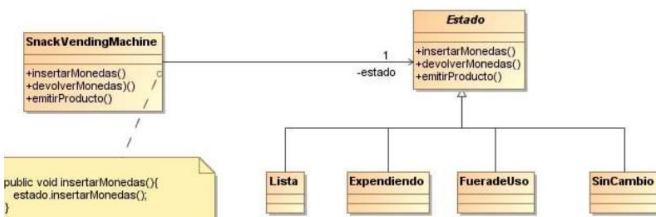
mensajes de otros objetos para solicitudes tales como abrir, cerrar o establecer conexión.

◆ Un cajero automático realiza sus operaciones dependiendo de su estado interno: disponible, sin dinero, sin papel, procesando trabajo interno, sin conexión,...

◆ Una máquina vendedora de productos también tiene diferentes estados.

## Estado: Motivación

Eliminar CASE con una jerarquía de estados



```

public class Pelicula {...}
private String _titulo;
private PrecioPelicula _precio;
public Pelicula(String titulo, String tipo) {
    _titulo = titulo;
    try {
        _precio = (PrecioPelicula)
            Class.forName("videoclubFowler.precioPelicula"
                +tipo.toUpperCase()).newInstance();
    } catch(Exception e) {
        e.printStackTrace();
    }
}

public double getCosteAlquilerPelicula(Alquiler alquiler) {
    return _precio.getPrecioPelicula(alquiler);
}
  
```

## Estrategia con expresiones lambda

```

public class App {
    public static void main(String[] args) {
        Viaje viaje = new Viaje();
        viaje.setDistancia(500);
        viaje.setTipoCoche(Confort.MEDIO);
        ReglaPrecio regla = (Viaje v) -> {
            double precio;
            switch (v.getTipoCoche()){
                case BAJO: precio = 5;break;
                case MEDIO: precio = 10;break;
                case ALTO: precio = 15;break;
                default: precio =0;
            }
            precio = precio + v.getDistancia()*0.1;
            return precio;
        };
        viaje.setReglaPrecio(regla);
        double precio = viaje.calcularPrecio();
        System.out.println(precio);
    }
}
  
```

¿Cuándo interesa usar expresiones lambda?

## Estado: Motivación

```

public class SnackVendingMachine {
    final static int LISTA = 0;
    final static int EXPENDIENDO = 1;
    final static int SIN_CAMBIO = 2;
    final static int INSERTANDO_MONEDAS = 3;
    final static int PRODUCTO_SERVIDO = 4;
    final static int FUERA_DE_USO = 5;

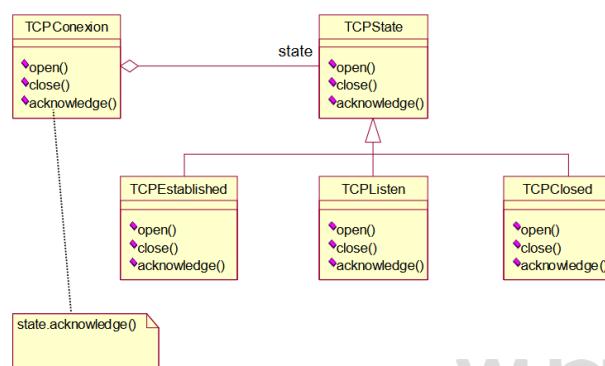
    private int estado = LISTA;

    private Product[][] products;

    public SnackVendingMachine() { ... }
    public boolean insertarMoneda(int cant){
        if (estado == LISTA) {
            ...
        } else if (estado == EXPENDIENDO) {
            ...
        } else if (estado == FUERA_DE_USO) {
            ...
        } else if (estado == SIN_CAMBIO) {
            ...
        }
        public boolean devolverMonedas() {
            if (estado == INSERTANDO_MONEDAS) {
                ...
            } else if (estado == EXPENDIENDO) {
                ...
            } else if (estado == FUERA_DE_USO) {
                ...
            } else if (estado == LISTA) {
                ...
            }
        }
        // otros métodos cuya implementación también está basada en un análisis
        // casos sobre el atributo estado
    }
}
  
```

## Estado. Ejemplo TCP

(en DP-GoF)



# Google Gemini: Plan Pro a 0€ durante 1 año. Tu ventaja por ser estudiante.

Oferta válida hasta el 9 de diciembre de 2025

[Consigue la oferta](#)



Después 21,99€/mes

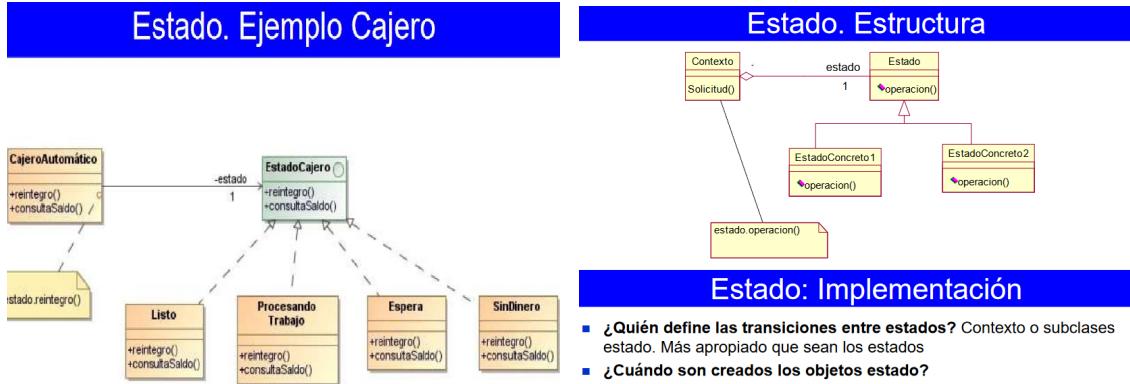
Domina cualquier tema con el Aprendizaje Guiado.

Puedes explicarme como se crea un eclipse lunar completo y sus fases?

¡Claro vamos paso a paso para que lo entiendas a la perfección! ☺ ☺ ☺ ☺ ☺ ☺ ☺ ☺

2:44

+ ☰ Aprendizaje Guiado x



### Estado: Implementación

- ¿Quién define las transiciones entre estados? Contexto o subclases estado. Más apropiado que sean los estados
- ¿Cuándo son creados los objetos estado?
  - Pueden crearse cuando se necesiten o con antelación y que Contexto tenga una referencia a ellos (una tabla).

#### v 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 (refactoring).

#### v Consecuencias

- Todo el comportamiento asociado a un particular estado es embebido en una clase.
- Subclases en vez de Sentencias CASE.
- Ayuda a evitar estados inconsistentes en caso de estado representado por varias variables.
- Transiciones de estado son más explícitas.
- Incrementa el número de objetos. Los objetos estado pueden ser Singleton

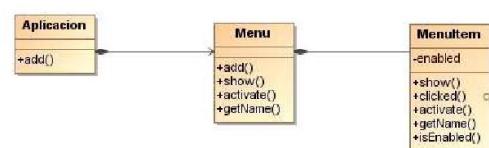
### Command (Orden)

#### v Propósito

Encapsula un mensaje como un objeto, permitiendo parametrizar métodos y objetos con mensajes, añadir mensajes a una cola y soportar funcionalidad deshacer/rehacer (undo/redo)

#### v Motivación

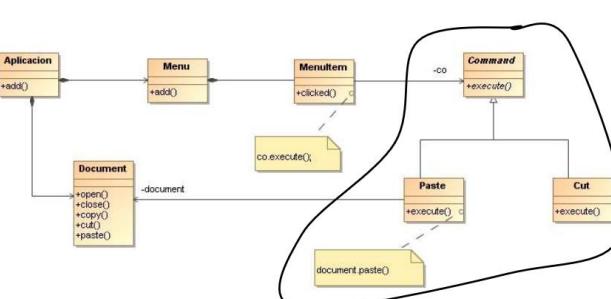
Algunas veces es necesario enviar un mensaje a un objeto sin conocer el selector del mensaje ni el objeto receptor. Por ejemplo widgets como botones o menús deben ejecutar una acción cuando el usuario hace clic sobre ellos.



¿Cómo parametrizar MenuItem con la acción que debe ejecutarse?

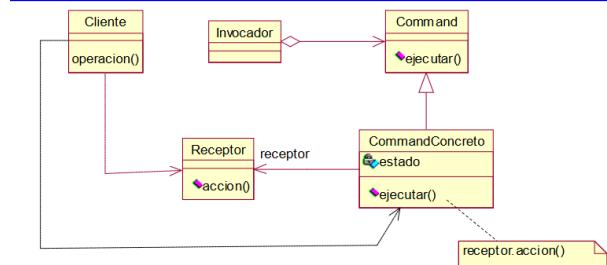
### Command: Motivación

¿Cómo parametrizar MenuItem con la acción que debe ejecutarse?



Lenguaje sin lambdas ni metaprogramación, p.e., Java < 8

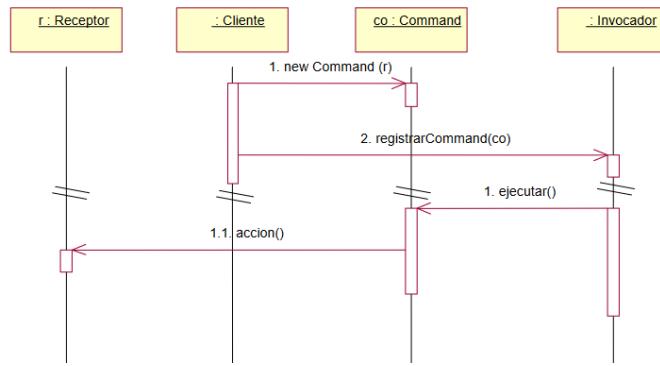
### Command: Estructura



### Command: Implementación

- El lenguajes que soportan expresiones lambda (Java, Python, Scala,...), punteros a funciones (C++) o metaprogramación, el patrón Command sólo sería útil para soportar undo/redo.

# Command: Colaboración



- ♦ Recuperación de fallos.

## Command en Java con expresiones lambda (sin considerar interfaz Consumer)

```

// interfaz funcional
public interface Command {
    public void perform();
}

public class MenuItem {
    public void clicked(Command co) {
        co.perform();
    }
}

public class App {
    public static void main(String[] args) {
        Cuenta c1 = new Cuenta(800000, "123", "Juan");
        MenuItem item1 = new MenuItem();
        item1.clicked(() -> System.out.println(c1.getSaldo()));
    }
}
  
```

```

public interface Command<T> {
    public void perform(T t);
}
public class MenuItem<T> {
    Command<T> action;
    T obj;
    public MenuItem(Command<T> action, T obj){
        this.action = action;
        this.obj= obj;
    }
    public void clicked(){
        action.perform(obj);
    }
}
public class App {

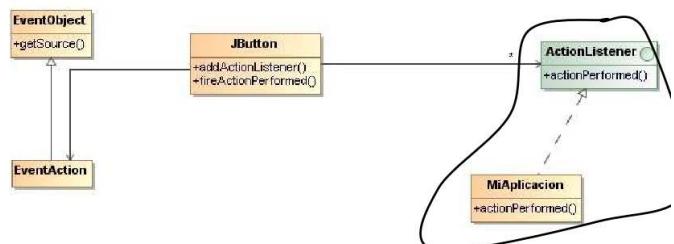
    public static void main(String[] args) {
        Cuenta cuenta = new Cuenta(800000, "123", "Juan");
        Command<Cuenta> action = c-> System.out.println(c.getSaldo());
        MenuItem<Cuenta> item1 = new MenuItem<Cuenta>(action,cuenta);
        item1.clicked();
    }
}
  
```

## Command con interfaz Consumer

```

public class MenuItem<T> {
    Consumer<T> action;
    T obj;
    public MenuItem(Consumer<T> action, T obj){
        this.action = action;
        this.obj= obj;
    }
    public void clicked(){
        action.accept(obj);
    }
}
public class App {
    public static void main(String[] args) {
        Cuenta c1 = new Cuenta(800000, "123", "Juan");
        Consumer<Cuenta> action = c->
            System.out.println(c.getSaldo());
        MenuItem<Cuenta> item1 = new MenuItem<Cuenta>(action,c1);
        item1.clicked();
    }
}
  
```

## Button en Swing



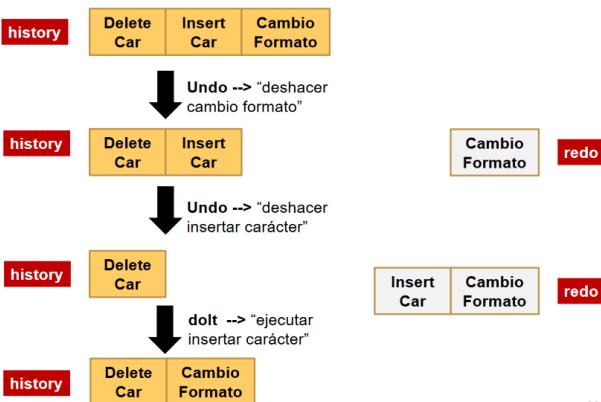
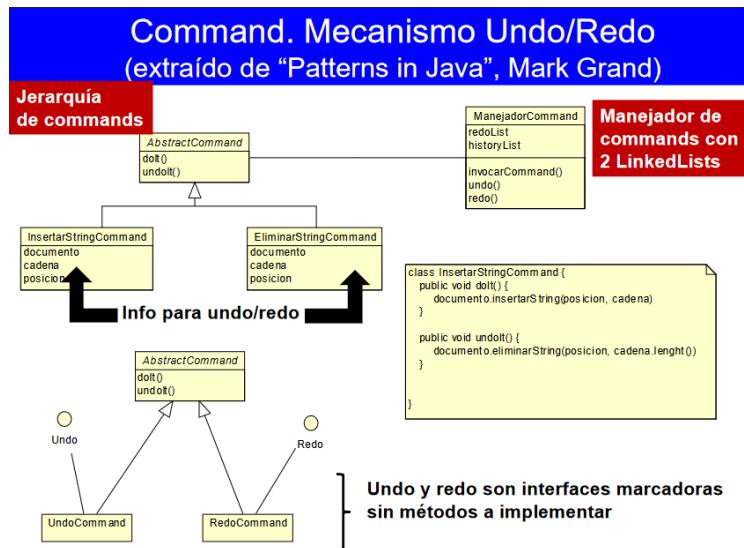
ActionListener juega el papel de un **Command** pero veremos que es parte de una implementación del patrón **Observer**.

## Command: Consecuencias

- ✓ Desacopla el objeto que invoca la operación del objeto que sabe cómo realizarla.
- ✓ Cada subclase CommandConcreto encapsula 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)

## Command. Mecanismo Undo/Redo

- Una jerarquía de Commands representa las operaciones que se pueden deshacer/rehacer.
- Los command deben tener atributos para registrar la información que permita deshacer/rehacer.
- Una lista “historia” de objetos commands registra copias de las operaciones realizadas (una única lista doblemente enlazada o tener dos listas: historia y redo)
- Una clase “manager” soporta el manejo de las listas de commands y de la recepción de operaciones.
- Las operaciones undo y redo se tratan de forma distinta al resto de commands.



## Command. Ejercicio “Mando de Control Remoto”

Una empresa desea construir una API para un mando de control remoto programable de casas inteligentes. El mando podrá ser programado con 7 funciones diferentes y tendrá para cada una de ellas dos botones, uno de encendido (on) y otro de apagado (off). Cada función puede ser asignada a uno de los dispositivos controlables existentes en la casa (luces, termostato, puerta del garaje, ventilador techo, TV, etc.). El mando también tiene un botón que permite “deshacer” la acción ejecutada por el último botón presionado. Para cada dispositivo existe una clase que implementa su control (por ejemplo, las clases Luz, VentiladorTecho, PuertaGaraje, etc.) que son proporcionadas por los fabricantes.

a) Diseña una solución basada en patrones de diseño para el software de control del mando teniendo en cuenta que debe poder controlar dispositivos ya existentes u otros que puedan aparecer en el futuro.

Dibuja el diagrama de clases y escribe el código Java de la solución. b) Modifica la solución anterior para que sea posible asignar el control de varios dispositivos a un botón del mando.

```

public class ControlRemoto {
    Command[] onCommands;
    Command[] offCommands;
    Command undoCommand;

    public ControlRemoto() {
        onCommands = new Command [7];
        offCommands = new Command [7];
        Command noCommand = new NoCommand();
        for (int i = 0; i < 7; i++) {
            onCommands[i] = noCommand; offCommands[i] = noCommand;
        }
        undoCommand = noCommand
    }

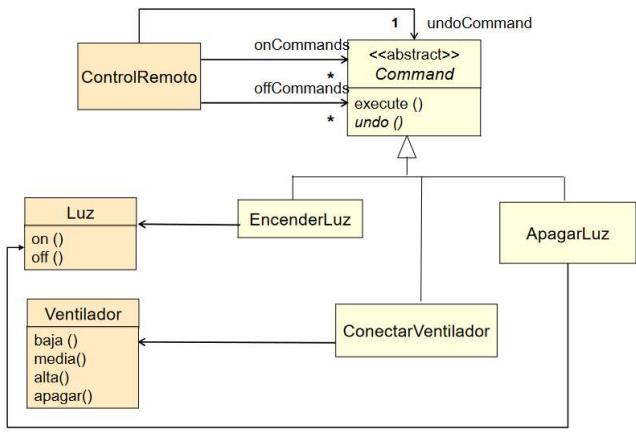
    public void setCommand (int slot, Command onCommand, Command offCommand){
        onCommands[slot] = onCommand; offCommands[slot] = offCommand;
    }

    public void botonOnPulsado (int slot) {
        onCommands[slot].execute();
        undoCommand = onCommands[slot];
    }

    public void botonOffPulsado (int Slot) {
        offCommands[slot].execute();
        undoCommand = offCommands[slot];
    }

    public void botonUndoPulsado () {
        undoCommand.undo();
    }
}
  
```

## Command. Ejercicio “Mando de Control Remoto”





# Organiza tu futuro: estudia hoy para destacar mañana.

En Carpe Diem te esperan cursos adaptados a ti.  
Una forma fácil y real de avanzar profesionalmente.

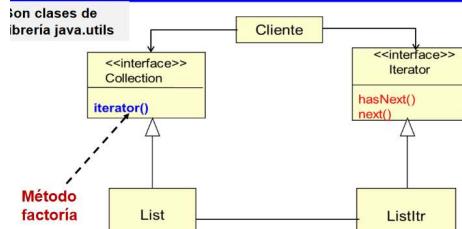


## Iterator (Iterador)

### v Propósito

- Proporciona una forma para acceder a los elementos de una estructura de datos sin exponer los detalles de la representación.
- Proporcionados por mayoría de lenguajes de programación
- v Motivación
  - Un objeto contenedor (colección) debe permitir una forma de recorrer sus elementos sin exponer su estructura interna.
  - Debería permitir diferentes métodos de recorrido y recorridos concurrentes
  - Esta funcionalidad no es parte de la interfaz de la colección.
  - Iteradores externos vs. Iteradores internos.
    - v Externos: recorrido controlado por el código que usa el iterador (tarea del programador)
    - v Internos: recorrido controlado por el propio iterador (se libera al programador que sólo dice lo que quiere)

## Iterator Externo + Método Factoría (Java)



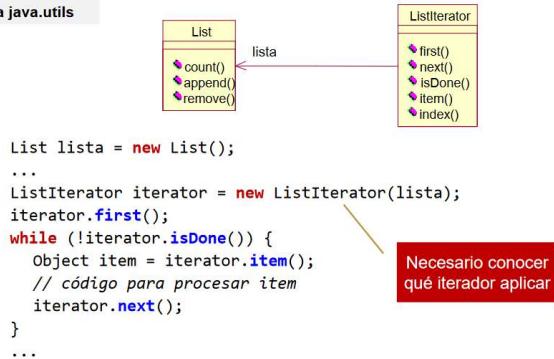
```

List<Cuenta> cuentas;
...
Iterator<Cuenta> it = cuentas.iterator();
while (it.hasNext()) {
    Cuenta cu = it.next();
    if (cu.getSaldo()>10000) cu.reintegro(600)
}
  
```

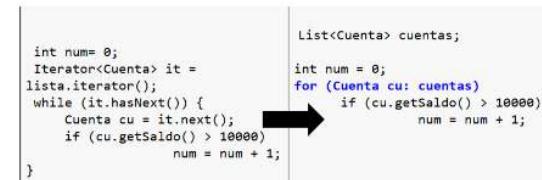
52

## Iterator Externo: Ejemplo

No son clases de  
biblioteca java.util



## Iterator Externo for en Java 5



## Patrones de operaciones con colecciones

for (T:e col1)  
 if (test(e)) col2.add(e);

Select / Filter

for (T:e col1)  
 col2.add(accion(e));

Map

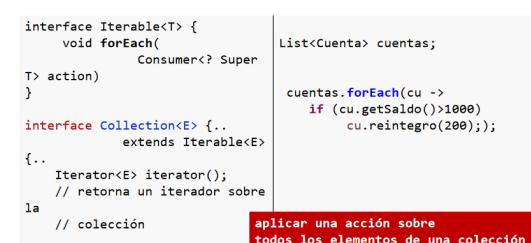
for (T:e col1)  
 accion(e);

Do / Foreach

existe = false;  
for (T:e col1)  
if (test(e)) {  
 existe = true;  
break;
}

Any?

## Iterator Interno en Java 8



## Iterator interno

- v El recorrido es controlado internamente, el programador es liberado.
- v Permiten iterar sobre una colección y ejecutar acciones como “seleccionar todos los elementos que cumplen una condición” o “aplicar una operación sobre cada elemento”.
- v Requieren el mecanismo llamado “bloque de código”, closure, expresiones lambda.
- v Comunes en lenguajes OO tipados dinámicamente como Smalltalk, Python, y Ruby.
- Seleccionar todas las cuentas con saldo mayor que 1000  
`colCuentas select: [:cu | cu getsaldo > 1000]`
- Aplicar la operación reintegro: a todas las cuentas  
`colCuentas do: [:cu | cu reintegro: 1000]`
- v Un iterador interno requiere parámetros que son operaciones: condición y/o acción.
  - ♦ En lenguajes OO tipados dinámicamente se usan bloques y en Java 8 se usan expresiones lambda en la clase Stream que ofrece iteradores internos.
  - v Cuando no se puede pasar código como parámetro, como sucedía en Java.
    - ♦ Se debe encapsular los “parámetros” como objetos y es posible aplicar los patrones Método Plantilla y Estrategia.
    - ♦ Método plantilla
  - v Una clase abstracta “Iterador” contiene el código de los iteradores y tiene métodos abstractos para “acción” y “condición”.
  - ♦ Estrategia
  - v Una clase “Iterador” contiene el código de los iteradores y delega en unas interfaces Acción y Condición

## Iteradores internos en clase Stream

- Un stream (`java.util.Stream`) es una secuencia de elementos que soporta operaciones secuenciales y paralelas, entre ellas iteradores internos.
- Las colecciones incluyen dos métodos para obtener streams (`stream` y `parallelStream`) y la clase `Stream` ofrece métodos para iterar sobre su elementos (*iteradores internos*)

**Ejemplo:** contar las cuentas con un saldo positivo en una colección cuentas de objetos Cuenta.

```
long contador = cuentas.stream()
    .filter(cu -> cu.getSaldo() > 0)
    .count();
```

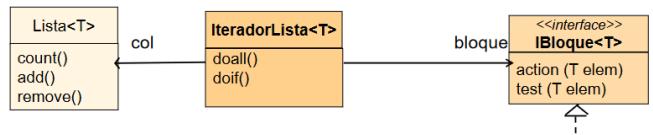
## Código Java 7 equivalente al anterior

```
//filtro
List<Transaction> compradivisas = new ArrayList<>();
for(Transaction t: transactions){
    if(t.getType() == Transaction.COMPRADIVISAS)
        compradivisas.add(t);
}
//ordeno
Collections.sort(compradivisas, new Comparator(){
    public int compare(Transaction t1, Transaction t2){
        return t2.getValue().compareTo(t1.getValue());
    }
});
//construyo nueva lista
List<Integer> transactionIds = new ArrayList<>();
for(Transaction t: compradivisas){
    transactionIds.add(t.getId());
}
```

## Ejemplo Iterador Interno

```
// imprimir cuentas cuyo saldo supere cierta cantidad
class IteradorSaldoCuenta extends IteradorLista<Cuenta> {
    int cantidad;
    IteradorSaldoCuenta (List<Cuenta> cc, int cant){
        super(cc);
        cantidad = cant;
    }
    protected boolean test (Cuenta elem){
        return (elem.getSaldo())>cantidad;
    }
    protected void action (Cuenta elem){
        System.out.println(elem.getCodigo()+" "
                           + elem.getSaldo());
    }
}
```

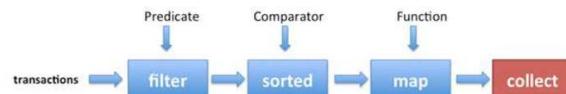
## Iterador interno basado en composición (i)



```
public abstract class IteradorLista<T> {
    private List<T> lista;
    public IteradorLista(List<T> lista) {
        this.lista = lista;
    }
    public void doIf(IBloque bloque) {
        for (T o : col)
            if (bloque.test(o)) bloque.action(o);
    }
}
```

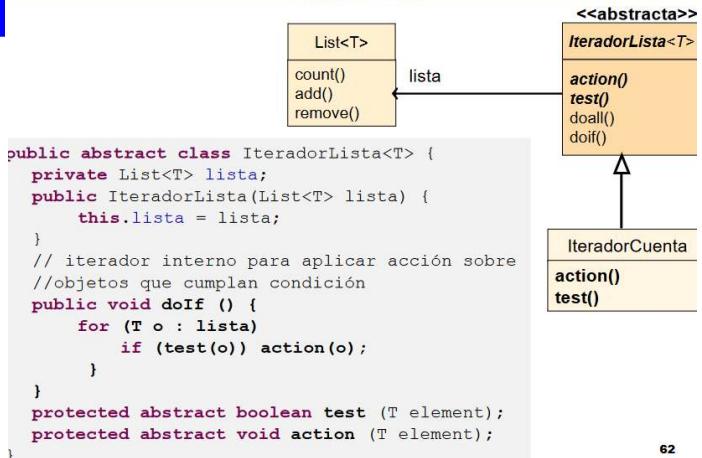
## Streams

"Obtener lista de identificadores de transacciones de cierto tipo ordenadas por el valor de la transacción de mayor a menor."



```
List<Integer> transactionIds = transactions.stream()
    .filter(t -> t.getType() == Transaction.COMPRADIVISAS)
    .sorted(comparing(Transaction::getValue).reversed())
    .map(Transaction::getId)
    .collect(toList());
```

## Iterador Interno con Método Plantilla (Java <8)

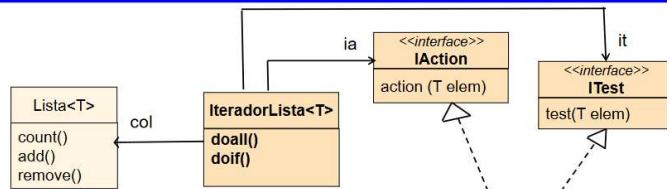


62

```
public class PruebaIterador {
    public class IteradorSaldo extends IteradorLista<Cuenta> {
        ...
    }
    public static void main(String[] args) {
        PruebaIterador p = new PruebaIterador();
        ArrayList<Cuenta> lista = new ArrayList<Cuenta>();
        Cuenta c1 = new Cuenta (100, "123", "JGM");
        ...
        Cuenta c6 = new Cuenta (1000, "222", "MOGM");
        lista.add(c1);.. lista.add(c6);

        p.new IteradorSaldo(lista,100).doIf();
    }
}
```

## Iterador interno basado en composición (ii)



```
public class IteradorLista<T> {
    private List<T> lista;
    public IteradorLista(List<T> lista) {
        this.lista = lista;
    }
    public void doIf (ITest<T> it, IAccion<T> ia) {
        for (T o : lista)
            if (it.test(o)) ia.accion(o);
    }
}
```

```

public class PruebaIterador {
    private int valor;
    private ArrayList<Cuenta> lista;

    public PruebaIterador() {
        lista = new ArrayList<Cuenta>();
        Cuenta c1 = new Cuenta (100, "123", "JGM");
        ...
        Cuenta c6 = new Cuenta (1000, "222", "MOGM");
        lista.add(c1); ... ;lista.add(c6);
    }
    public int getValor() {
        return valor;
    }
    public void setValor(int valor) {
        this.valor = valor;
    }
    public void ejecutar() {
        IteradorLista<Cuenta> it = new IteradorLista<Cuenta>(lista);
        //clase interna anónima
        it.dolif(new IBloque<Cuenta>() {
            @Override
            public boolean test(Cuenta cuenta) {
                return (cuenta.getSaldo() > valor);
            }
            @Override
            public void action(Cuenta cuenta) {
                System.out.println(cuenta.getNombreTitular()+" "+cuenta.getSaldo());
            }
        });
    }
    public static void main(String[] args) {
        PruebaIterador p = new PruebaIterador();
        p.setValor(Integer.parseInt(args[0]));
        p.ejecutar();
    }
}

```

```

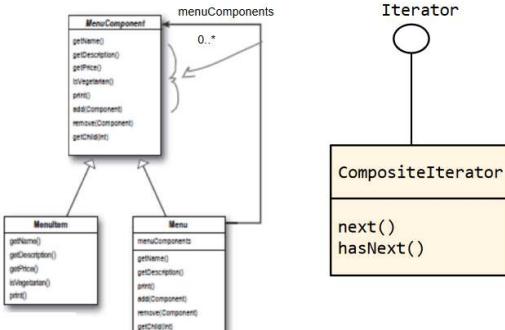
public class Iterador<T> {
    ArrayList<T> col;
    public Iterador(ArrayList<T> c) {
        col = c;
    }
    public ArrayList<T> select(Predicate<T> p){
        ArrayList<T> result = new ArrayList<T>();
        for(T o: col)
            if (p.test(o)) result.add(o);
        return result;
    }
    public ArrayList<T> reject(Predicate<T> p){
        ArrayList<T> result = new ArrayList<T>();
        for(T o: col)
            if (!p.test(o)) result.add(o);
        return result;
    }
    public void dolif(Predicate<T> p, Consumer<T> c) {
        for (T o: col)
            if (p.test(o)) c.accept(o);
    }
    public void forEach(Consumer<T> c){
        for (T o: col)
            c.accept(o);
    }
}

```

Una clase Iterador basada en las interfaces funcionales  
Predicate y Consumer

## Composite + Iterador

(extraído de “Head First Design Patterns”)



68

## Composite + Iterador

```

public class Menu extends MenuComponent {
    ArrayList menuComponents = new ArrayList();
    String name;
    String description;

    // constructor code here
    // other methods here

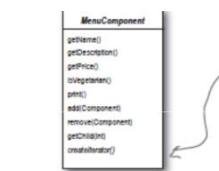
    public void print() {
        System.out.print("\n" + getName());
        System.out.println(", " + getDescription());
        System.out.println("-----");
        Iterator iterator = menuComponents.iterator();
        while (iterator.hasNext()) {
            MenuComponent menuComponent =
                (MenuComponent)iterator.next();
            menuComponent.print();
        }
    }
}

```

All we need to do is change the print() method to make it print not only the information about this Menu, but all of this Menu's components: other Menus and MenuItem.

Look! We get to use an Iterator. We use it to iterate through all the Menu's components - those could be other Menus, or they could be MenuItem. Since both Menus and MenuItem implement print(), we just call print() and the rest is up to them.

NOTE: If, during this iteration, we encounter another Menu object, its print() method will start another iteration, and so on.



Now we need to implement this method in the Menu and MenuItem classes:

```

public class Menu extends MenuComponent {
    Iterator iterator = null;
    // other code here doesn't change
    public Iterator createIterator() {
        if (iterator == null) {
            iterator = new CompositeIterator(menuComponents.iterator());
        }
        return iterator;
    }
}

public class MenuItem extends MenuComponent {
    // other code here doesn't change
    public Iterator createIterator() {
        return new NullIterator();
    }
}

```

We've added a createIterator() method to the MenuComponent. This means that each Menu and MenuItem will need to implement this method. It also means that calling createIterator() on a composite should apply to all children of the composite.

Here we're using a new iterator called CompositeIterator. It knows how to iterate over any composite.

We pass it the current composite's iterator.

Now for the MenuItem... Whoa! What's this NullIterator? You'll see in two pages.

## Iterador

### v Consecuencias

- Simplifica la interfaz de una colección al extraer los métodos de recorrido
- Permite varios recorridos concurrentes
- Soporta variantes en las técnicas de recorrido
- v Implementación
- ¿Quién controla la iteración? : Externos vs. Internos
- ¿Quién define el algoritmo de recorrido?
- v Agregado: iterador sólo almacena el estado de la iteración (Cursor)
- v Iterador: es posible reutilizar el mismo algoritmo sobre diferentes colecciones o aplicar diferentes algoritmos sobre una misma colección
- Es posible modificar la colección durante la iteración?
- Colección e iterador son clases intimamente relacionadas: clases amigas en C++ y clases internas en Java para iteradores externos.

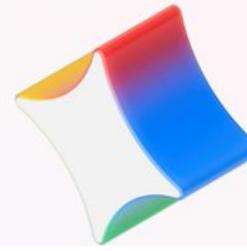
# Google Gemini: Plan Pro a 0€ durante 1 año.

Tu ventaja por ser estudiante.

Oferta válida hasta el 9 de diciembre de 2025

Consegue la oferta

Después 21,99€/mes



```

public class CompositeIterator implements Iterator {
    Stack stack = new Stack();
    public CompositeIterator(Iterator iterator) {
        stack.push(iterator);
    }
    public Object next() {
        if (hasNext()) {
            Iterator iterator = (Iterator) stack.peek();
            MenuComponent component = (MenuComponent) iterator.next();
            if (component instanceof Menu) {
                stack.push(component.createIterator());
            } else {
                return component;
            }
        }
        return null;
    }
    public boolean hasNext() {
        if (stack.empty()) {
            return false;
        } else {
            Iterator iterator = (Iterator) stack.peek();
            if (!iterator.hasNext()) {
                stack.pop();
                return hasNext();
            } else {
                return true;
            }
        }
    }
    public void remove() {
        throw new UnsupportedOperationException();
    }
}
  
```

The iterator of the top level composite we're going to iterate over is passed in. We throw that in a stack data structure.

Okay, when the client wants to get the next element we first make sure there is one by calling hasNext().

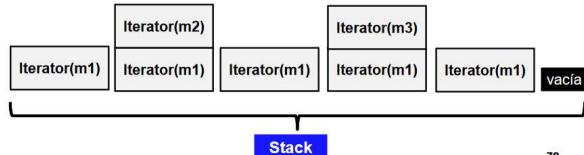
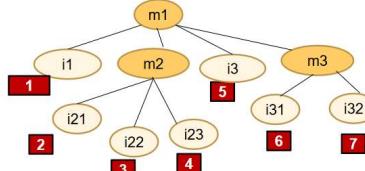
If there is a next element, we get the current iterator off the stack and get its next element.

If that element is a menu, we have another composite that needs to be included in the iteration so we throw it on the stack. In either case, we return the component.

To see if there is a next element, we check to see if the stack is empty; if so, there isn't. Otherwise, we get the iterator off the top of the stack and see if it has a next element. If it doesn't we pop it off the stack and call hasNext() recursively.

Otherwise there is a next element and we return true.

We're not supporting remove, just traversal.



## Observer / Publish-Subscribe (Observador)

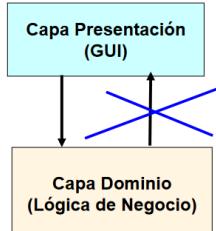
## Observador: Estructura

### Propósito

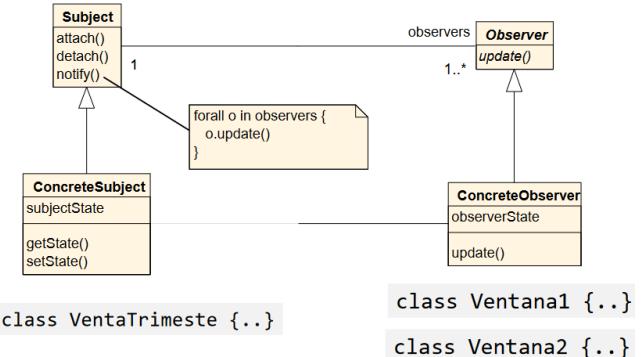
- Define una dependencia uno-a-muchos entre objetos, de modo que cuando cambia el estado de un objeto, todos sus dependientes son automáticamente notificados.

### Motivación

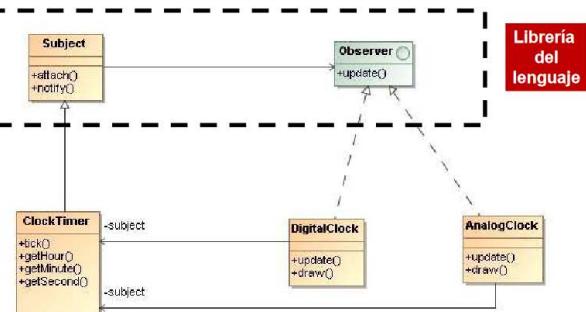
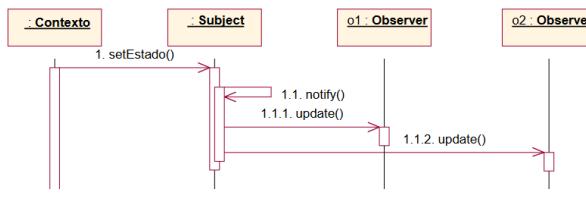
- Ejemplo: Separación Modelo-Vista



Los objetos del dominio no deben conocer a los objetos de la interfaz. ¿cómo les notifican de un cambio?



## Observador. Ejemplo "Alarma" (GoF)



### Aplicabilidad

- Cuando un cambio de estado en un objeto requiere cambios en otros objetos, y no sabe sobre qué objetos debe aplicarlos.
- Cuando un objeto debe ser capaz de notificar algo (p.e., un evento) a otros objetos, sin hacer asunciones sobre quiénes son estos objetos.

### Consecuencias

- Acoplamiento abstracto y mínimo entre Subject y Observer:
  - Puedo reutilizarlos por separado
  - Pueden estar en diferentes capas
  - Pueden añadirse observers sin modificar el subject
  - ConcreteSubject no necesita conocer las clases concretas de Observers
  - Es posible añadir y eliminar observers en cualquier instante.

Es posible que un observer esté ligado a más de un subject: la operación update tendrá como argumento el subject.

Al registrar un observer es posible asociarle el evento sobre el que quiere ser notificado. □ Quién dispara la notificación?

- Normalmente, métodos set en la clase Subject, en vez de clases clientes de la clase Subject

□ Cuánta información sobre el cambio se le envía a los observers con la notificación?

- Conveniente pasárselo el subject o un objeto

## Patrón Observador 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() {...}
```

**Modelo de Delegación de Eventos** es normalmente usado en vez de Observable y Observer.

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

Argumento recibido del método notifyObservers, puede indicar el tipo de cambio

**Observers pueden ser implementados con expresiones lambda** (ejemplo de Java 8 Lambdas, Richard Warburton)

```
public interface LandingObserver {
    public void observeLanding(String name);
}

public class Moon {
    private final List<LandingObserver> observers =
        new ArrayList<>();

    public void land(String name) {
        for (LandingObserver observer : observers) {
            observer.observeLanding(name);
        }
    }
    public void addLandingObserver(LandingObserver obs) {
        observers.add(obs);
    }
}
```

```
public class Producto extends Observable {
    private String nombre;
    private float precio;
    public Producto(String nombre, float precio) {
        this.nombre = nombre;
        this.precio = precio;
    }
    //aquí métodos getters
    public void setNombre(String nombre) {
        this.nombre = nombre;
        setChanged();
        notifyObservers(nombre);
    }
    public void setPrecio(float precio) {
        this.precio = precio;
        setChanged();
        notifyObservers(new Float(precio));
    }
}

public class PrecioObserver implements Observer {
    private float precio;
    public PrecioObserver() {
        this.precio= 0;
    }
    public void update(Observable obj, Object arg) {
        if (arg instanceof Float) {
            precio = ((Float)arg).floatValue();
            System.out.println(
                "PrecioObserver: Precio ha cambiado a " + precio);
        } else {
            System.out.println(
                "PrecioObserver: Algo diferente ha cambiado!");
        }
    }
}

Moon moon = new Moon();

moon.addLandingObserver(name ->
    if (name.contains("Apollo"))
        System.out.println("We made it!"));

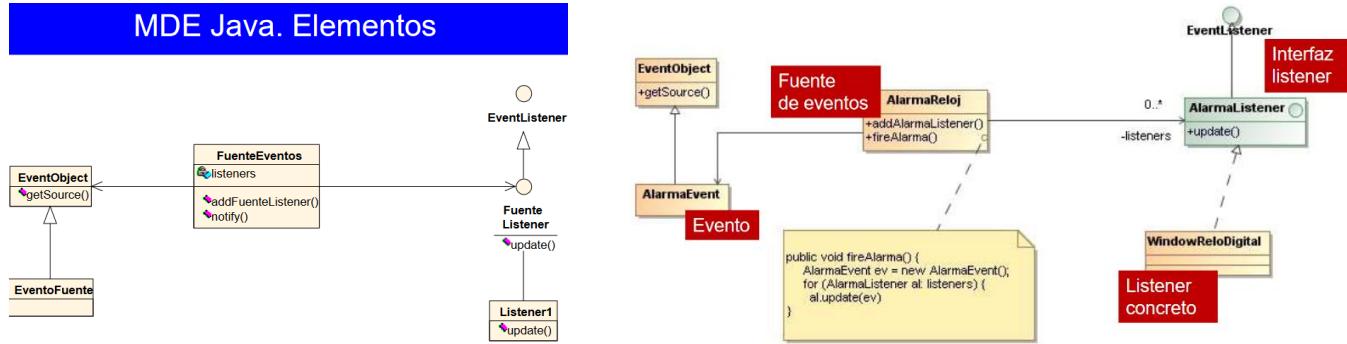
moon.addLandingObserver(name -> {
    if (name.contains("Apollo"))
        System.out.println("They're distracted,
                           lets invade earth!"));

moon.land("An asteroid");
moon.land("Apollo 11");
```

### Modelo de delegación de eventos (MDE)

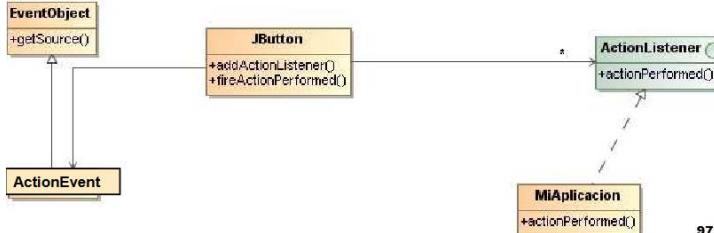
- Java 1.1 introdujo un nuevo modelo de eventos basado en el patrón Observer.
- Objetos (p.e. una GUI) que pueden generar eventos son llamados fuentes de eventos.
- Objetos que desean ser notificados de eventos (p.e. procedentes de una GUI) son llamados oyentes de eventos (event listeners).
- Los listeners deben registrarse en las fuentes e implementan una interfaz con los métodos que deben ser llamados por la fuente cuando ocurre el evento.

### MDE Java. Elementos



Una fuente juega el papel de **ConcreteSubject** y un listener el de **ConcreteObserver**.

- ActionEvent es el evento producido
  - Ejemplos: Clic en botón, pulsar «enter» en un campo de entrada
- ActionListener es la interfaz listener:
  - public void actionPerformed(ActionEvent e)
- Método para añadir un oyente de ese evento
  - public addActionListener(ActionListener al)

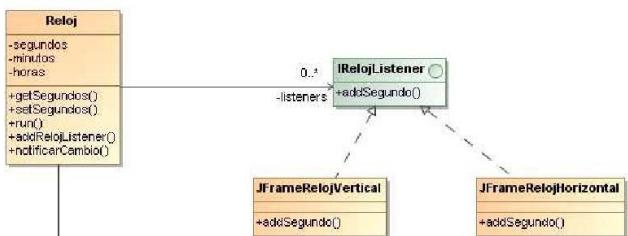


```

public class Reloj extends Thread {
    private int hora = 0;
    private int minutos = 0;
    private int segundos = 0;
    private ArrayList<IRelojListener> listeners =
        new ArrayList<IRelojListener>();
    public synchronized void addRelojListener(IRelojListener listener) {
        listeners.add(listener);
    }
    private void notificarCambio(RelojEvent event) {
        for (IRelojListener list : listeners)
            list.addSegundo(event);
    }
    // faltan get y set para minutos y horas
    public int getSegundos() {
        return segundos;
    }

    public void setSegundos(int seg) {
        if (seg==60) segundos = 0;
        else segundos = seg;
        RelojEvent ev = new RelojEvent(new Integer(segundos));
        notificarCambio(ev);
    }
}

```



```

97
public void run () {
    while (true){
        try {
            sleep(1000);
            segundos = segundos + 1;
            setSegundos(segundos);
        }catch (InterruptedException e) { return;}
    }
}

public static void main(String[] args) {
    Reloj reloj = new Reloj();
    JFrameRelojVertical rv = new JFrameRelojVertical(reloj);
    JFrameRelojHorizontal rv = new JFrameRelojHorizontal(reloj);
    reloj.start();
}

```

Listeners podrían ser implementados con expresiones lambda

```

public class JFrameRelojVertical {
    private JFrame frmReloDigitalDs;
    private JTextField textField;
    private JTextField textField_1;
    private JTextField textField_2;
    private Reloj reloj;
    private int segundos, minutos, hora;

    public JFrameRelojVertical(Reloj reloj) {
        initialize();
        reloj.addRelojListener(new IRelojListener(){
            @Override
            public void addSegundo(RelojEvent ev) {
                segundos = ev.getSegundos();
                if (segundos == 0) {
                    minutos = minutos + 1;
                    if (minutos == 60) {
                        minutos = 0;
                        hora = hora + 1;
                        if (hora == 24) hora = 0;
                    }
                }
                textField_2.setText(""+segundos);
                textField_1.setText(""+minutos);
                textField.setText(""+hora);
            }
        });
    }
}

102

```

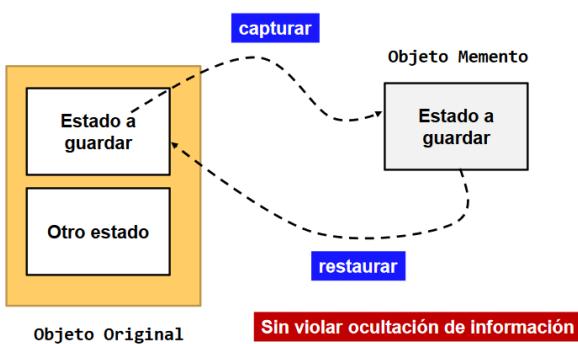
## Memento

### v Propósito

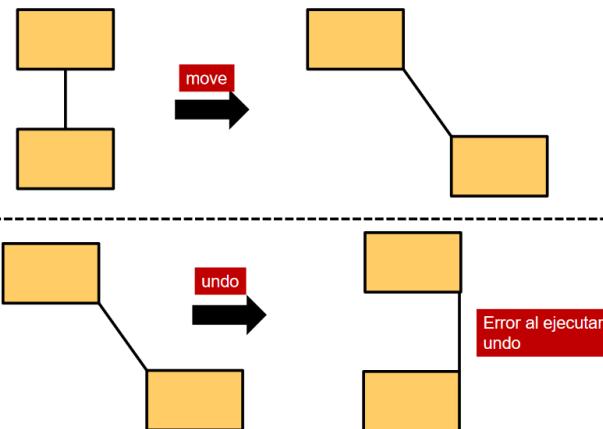
- ◆ Captura y externaliza el estado interno de un objeto, sin violar la encapsulación, de modo que el objeto puede ser restaurado a este estado más tarde.

### v 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.
- ◆ En la implementación de undo/redo, los commands registrados en la lista history podrían referir a un memento del objeto receptor.



## Memento: Motivación (DP de GoF)

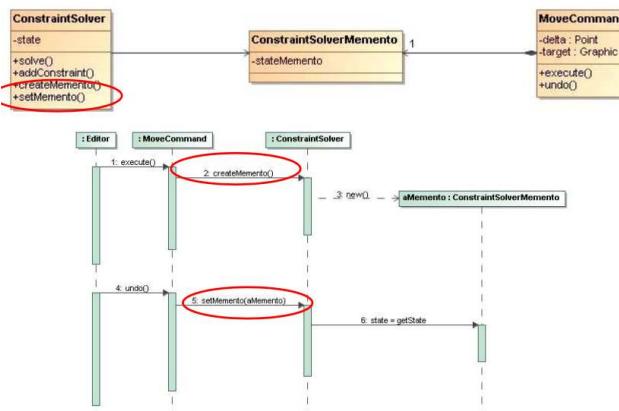




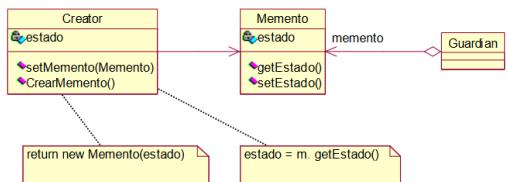
# LA NUESTRA DURA MÁS



## Memento. Ejemplo del libro GoF



## Memento: Estructura



## Memento: Implementación

**Memento tiene dos interfaces**, una para los creadores y otra para el resto de clases: clases amigas en C++, clase privada estática o acceso paquete en Java.

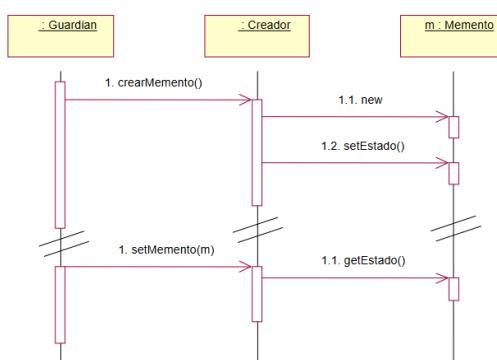
### Aplicabilidad

- 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.

### Consecuencias

- Mantiene la encapsulación
- Simplifica la clase Creador ya que no debe preocuparse de mantener las versiones del estado interno.
- Encapsular y restaurar el estado no debe ser costoso.
- El lenguaje debe permitir asegurar que sólo el Creador tenga acceso al estado del Memento.

## Memento: Comportamiento



## Memento: Guardar estado de un juego

(adaptado de "Patterns in Java", Mark Grand)

```

public class Game {
    private String description;
    ...
    private MilestoneMementoManager mementoManager;

    public MilestoneMemento createMemento(String description){
        MilestoneMemento memento = new MilestoneMemento(description);
        ...
        return memento;
    }
    public void setMemento(MilestoneMemento memento){
        description = memento.description;
    }
    public static class MilestoneMemento {
        private String description;
        MilestoneMemento (String description){
            this.description = description;
        }
    }
}
  
```

```

public class MilestoneMementoManager {
    private ArrayList<MilestoneMemento> mementos =
        new ArrayList<MilestoneMemento>();
    public void addMemento (MilestoneMemento m) {
        mementos.add(m);
    }
    public Memento getMemento (int index) {
        return mementos.get(index);
    }
}
public class TestMemento {
    public static void main(String[] args) {
        Game game = new Game("Juego Star War");
        MilestoneMementoManager manager = new MilestoneMementoManager();
        ...
        manager.addMemento(game.createMemento());
        ...
        manager.addMemento(game.createMemento());
        ...
        g.setMemento( manager.getMemento(1));
    }
}
  
```

## Tema 4: Desarrollo de Software dirigido por pruebas

“Nadie quiere hablar sobre las pruebas (testing). El testing es el hijastro feo del desarrollo de software. Eso es un problema. Todo el mundo sabe lo importante que son las pruebas, pero nadie quiere hacer suficientes pruebas. Y luego lo lamentamos – nuestros proyectos no van tan bien como deberían y somos conscientes de que más pruebas ayudarían a resolver el problema -. Entonces leemos un libro de pruebas y nos quedamos atascados en los diferentes tipos y formas de pruebas, porque pensamos que no hay forma en la que podamos hacer todo lo que se dice en el libro y todavía desarrollar la aplicación”

- Testing es el proceso de comprobar si un software satisface los requisitos funcionales y no funcionales.
- “Probar(Testing) es el proceso de ejecutar un programa para encontrar errores”
- Un error es un estado interno incorrecto motivado por un defecto estático en el software que se manifiesta como un fallo o comportamiento incorrecto externo con respecto a los requisitos.
- Se escriben casos de pruebas (test cases) separados del código de producción.
- Depurar consiste en localizar el lugar exacto en el que se encuentra un error y modificar el software para corregirlo.
- Objetivo de calidad del software: Entregar software sin defectos

## Estrategias de pruebas

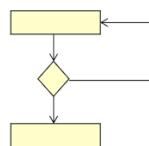
### Caja negra (más comunes)

- Las pruebas derivan de la especificación externa del software (requisitos, diseño, especificaciones de interfaces, ..)



### Caja blanca

- Las pruebas derivan de la estructura del código fuente a probar: bifurcaciones, iteraciones, llamadas a funciones, ...



Se escriben conociendo el código, por ejemplo para ver si se realiza efectivamente una llamada a una función

### Pruebas Unitarias (o de Módulo)

0 Comprueban la funcionalidad de un módulo o función (p.e., una clase o método en OO)

#### • Pruebas de Integración

0 Comprobar las interfaces e interacción entre los módulos o componentes del sistema.

#### • Pruebas de Sistema

0 Testing de requisitos no funcionales

0 Test de funcionalidad, test de rendimiento, test de seguridad, test de usabilidad, test de configuración, test de GUIs,..

#### • Pruebas de Aceptación

0 Comprobar si una aplicación satisface los requisitos (punto de vista del usuario)

• Realizar pruebas es una tarea muy laboriosa por lo que existen herramientas destinadas a su automatización.

- Pruebas unitarias: xUnit (JUnit, SUnit, NUnit, ..) , JMock, EasyMock,..

- Pruebas de integración: Jenkins, JTest, TestNG,..

- Pruebas de aceptación: Cucumber y “Robot framework” para “behaviour-driven development”

- Pruebas de GUI: Jemmy, Squish,

- Pruebas de aplicaciones web: Selenium

- Análisis de rendimiento

### Pruebas unitarias

- Se prueba la corrección de unidades elementales (clases y métodos en POO) no triviales.
- Se necesita una especificación del módulo (entradas, salidas y función) y el código fuente del módulo --> Pruebas de caja blanca
- Escritas por los propios desarrolladores en su entorno de trabajo.
- ¿Cuándo se deben escribir?

### Pruebas de integración

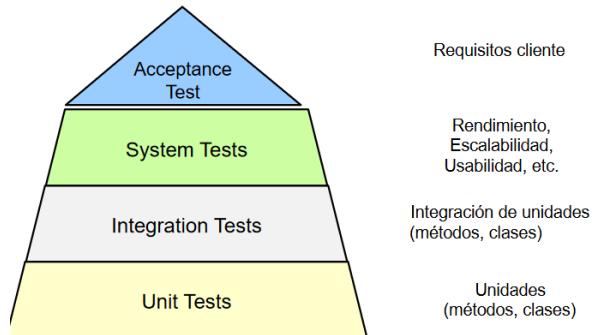
- Garantizan que los módulos funcionen bien cuando se integran

### Pruebas unitarias. Beneficios

- Se reducen los tiempos de depuración – Errores más localizados
- Facilitan el cambio
- Pueden servir como documentación
- Permiten pruebas en paralelo
  - No es necesario disponer de los módulos que requiere el módulo probado. – Programar hacia la interfaz para facilitar la separación
- Facilitan los tests de integración y regresión

- Testing dinámico
  - Se ejecuta el programa.
  - El término testing normalmente se refiere a este tipo.
- Testing estático
  - No se ejecuta el programa, se realizan inspecciones y algunas formas de análisis de código.
  - Se realiza normalmente por personas ajenas al proyecto (departamento de calidad si lo hay).
  - Herramientas de análisis de código

## Tipos de pruebas



## Pruebas unitarias. Limitaciones

- Las pruebas unitarias no pueden detectar todos los errores en un programa, por ejemplo no pueden capturar errores de integración, problemas de rendimiento y otros.
- Sólo son efectivas cuando se usan junto con otras pruebas del software, como las pruebas de integración y las pruebas del sistema.
- No es trivial conocer por anticipado todas las posibles situaciones de entrada de un módulo.
- Exige una disciplina rigurosa por parte de todo el equipo – Uso de un sistema de control de versiones (p.e. Subversion)

## Estrategia de Integración Continua

- Uso de un repositorio con control de versiones (p.e. Git, Github, Gitlab,...)
- Uso de herramientas para automatizar la construcción de la aplicación (p.e. Maven, Gradle)
- Uso de herramientas para pruebas unitarias (p.e. Junit, Selenium, ..)
- Un desarrollador realiza un pull de un proyecto en el repositorio, realiza cambios en el código, ejecuta pruebas unitarias para comprobar que es correcto y realiza un commit+push, entonces comprueba en la "máquina de integración" que la aplicación sigue funcionando bien (integration build).
- Herramientas de integración continua específicas (p.e. Jenkins, CruiseControl, Bamboo, GitLab CI, ...) para automatizar tareas.
- Todo el equipo hace una o más integraciones diarias.

## Pruebas de integración

- Las pruebas unitarias no garantizan que los módulos funcionen bien cuando se integren y son necesarias pruebas de integración.
- Se prueban las interfaces de elementos que ya funcionan y se han integrado para formar el sistema.
- El sistema normalmente está incompleto y se requiere código scaffolding: test stubs y test drivers.
- Test stubs
  - Esqueleto de la implementación de un módulo creado para probar un componente que depende de él.
  - Se denominan comúnmente mock
  - Retornan valores que pueden satisfacer ciertas restricciones
  - Pueden simular un acceso a base de datos o a una red.
- Test drivers
  - Un componente que sustituye a otro que toma el control y/o realiza invocaciones a un componente.
  - Ejemplo: método main() en Java.

## Test de integración. Tipos

- Estructural
  - Pruebas de caja blanca
  - Se analizan llamadas entre módulos y se analizan todos los posibles esquemas.
- Funcional
  - Pruebas de caja negra
  - Se buscan errores en un módulo que depende de otros y se basan en especificaciones.
  - Normalmente por pruebas de integración se entienden este tipo de pruebas funcionales.

## Test de regresión

- Son las pruebas realizadas para detectar errores cuando se modifica un programa.
- Un pequeño cambio en una parte del sistema puede originar problemas en partes "distantes" del sistema.
- Se ejecuta un conjunto de pruebas cuya elección es crucial para que el tiempo necesario no sea excesivo.
- Las pruebas de regresión suponen ejecutar las pruebas de integración de nuevo.

## Pruebas del Sistema

- Se prueba un sistema completo e integrado para comprobar si satisface todos sus requisitos funcionales y no funcionales -->

### Pruebas de caja negra

- Tipos de pruebas de sistema:
  - GUI – Facilidad de uso (Usabilidad) – Seguridad – Rendimiento – Recuperación del sistema
  - Volumen de datos – Sobrecarga – Configuración – Instalación

## Pruebas de aceptación

- Son realizadas por el cliente y son pruebas funcionales sobre el sistema completo que comprueban si se satisfacen los requisitos (especificación de requisitos, contrato, manual del usuario,...).
- La experiencia muestra que aún después del más cuidadoso proceso de pruebas, quedan una serie de errores que sólo aparecen cuando el cliente usa el sistema
- Pruebas alfa y beta cuando una aplicación se vende a un buen número de usuarios.
- Pruebas alfa. Se invita al cliente a la empresa de desarrollo. Se trabaja en un entorno controlado y un desarrollador ayuda al cliente a usar el sistema y analizar los resultados.
- Pruebas beta. Se realizan en el entorno del cliente después de las pruebas alfa. El entorno está fuera de control y el cliente, sin presencia de un desarrollador, trata de encontrar fallos (reales o imaginarios) de los que informa al desarrollador.

## Proceso de Pruebas

- Las pruebas deberían diseñarse y ejecutarse como parte del proceso de desarrollo de software en vez de dejarse para el final.  
¿Cómo se integran?

# SUDADERAS PARA GRUPOS PERSONALIZADAS



Extreme Programming, XP

## Valores

- Comunicación – Simplicidad – Realimentación – Valentía

## Principios

- Aceptar el cambio – Asumir Simplicidad – Rápida Realimentación – Cambio incremental – Trabajo de calidad

## Prácticas

- El juego de la planificación (user stories) – Versiones pequeñas (desarrollo iterativo e incremental)

- Diseño simple – Pruebas primero – Refactoring – Programación por parejas

- Código colectivo – Integraciones continuas – Semanas de 40 horas – Cliente en el equipo – Estándares de codificación

## Pruebas Unitarias

- ♣ Pruebas unitarias: uno de los principios básicos de eXtreme Programming (XP) y del Desarrollo de software dirigido por pruebas (TDD, Test-driven development), propuestos por Kent Beck.

- Escribir pruebas antes que el código (test-first programming), y escribir código necesario para pasar el caso de prueba.

- Su disponibilidad favorece la evolución: ¡Sin miedo al cambio!

- Se combinan con integración continua + refactoring.

- Conveniencia de herramientas de automatización de la creación y ejecución de pruebas unitarias

## Ejemplo de prueba unitaria en JUnit

```
import static org.junit.Assert.*;
public class ListaTest {
    @Test
    public void pruebaOrdenar() {
        //Objeto sobre el que vamos a hacer las pruebas
        String[] elems={"e", "d", "c", "b", "a"};
        listaAlReves= new Lista(elems);

        //Objeto con resultado esperado
        String[] expected={"a", "b", "c", "d", "e"};
        Lista result = new Lista(e2);

        //Ejecutamos assert que comprueba test
        assertEquals(result, listaAlReves.ordenar());
    }
}
```

## Manifiesto del Desarrollo Ágil

- ♣ Nosotros estamos descubriendo mejores formas de desarrollar software haciendo lo y ayudando a hacerlo. A través de nuestro trabajo hemos llegado a apreciar:

- Personas e interacciones antes que procesos y herramientas
- Trabajar con el software antes que documentación
- Colaborar con el cliente antes que la negociación de un contrato
- Responder al cambio antes que seguir un plan

- ♣ Esto es, aunque reconocemos que los ítems de la derecha tienen valor, nosotros valoramos más los ítems de la izquierda.

## Métodos ágiles más conocidos

- ♣ Scrum: el más usado

- ♣ Extreme programming (XP): el origen

- ♣ Lean: extraído de prácticas de Toyota

- ♣ Kanban: usado como complemento de Scrum o Lean

- ♣ Crystal: métodos desarrollados por Alistair Cockburn

## Criticas a los métodos ágiles (B. Meyer)

- ♣ No es nuevo y no es bueno

- Historias de usuario o casos de uso no pueden sustituir a los requisitos

- ♣ No es nuevo y es bueno

- Desarrollo iterativo e incremental

- Reconocimiento de la importancia del cambio, pero "no tiene sentido el desprecio hacia las prácticas existentes para facilitar la extensibilidad"

- ♣ Nuevo y no es bueno

- Programación en parejas

- Desarrollar software mínimo para pasar las pruebas

- Despreocuparse de capturar requisitos inicialmente.

- ♣ Nuevo y bueno

- Importancia de las pruebas

- Papel central que concede al equipo de desarrolladores.

- "Congelar los requisitos" durante las iteraciones y tienen un tiempo fijado.

- Énfasis en iteraciones cortas y que el valor está en el código

- Refactoring

## Refactoring

- ♣ Reestructurar un código fuente existente para cambiar su estructura interna sin afectar al comportamiento externo. Se aplican transformaciones que mantienen el comportamiento.

## Catálogo de refactorings:

- Eliminar una jerarquía – Extraer una clase
- Introducir una clase en otra – Extraer una subclase
- Extraer un método – Encapsular una colección
- Ocultar un método – Mover un campo

## Test driven development, TDD

TDD = Pruebas primero + Integración Continua + Refactoring + Automatización

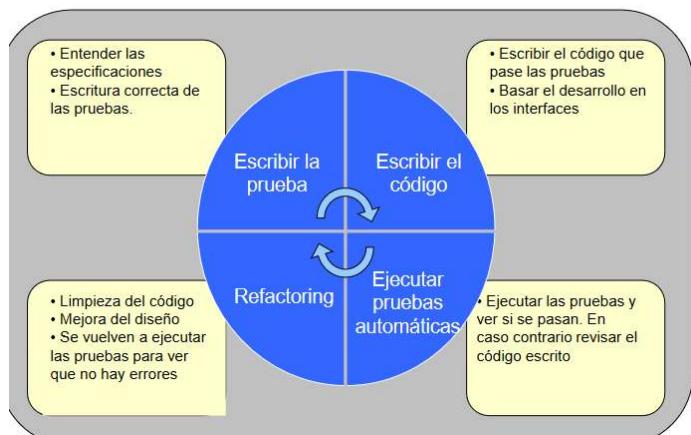
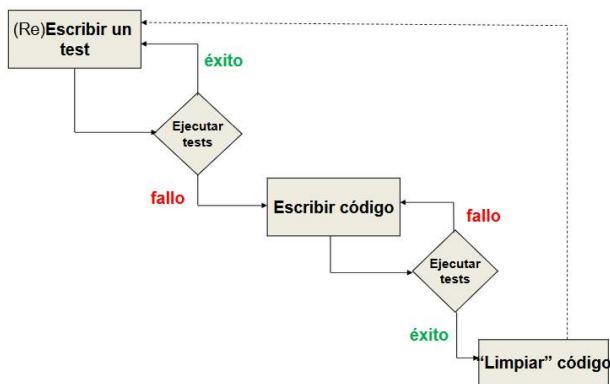
- Desarrollo dirigido por pruebas (Test-driven development, TDD) es un proceso de creación de software basado en la repetición de un ciclo de desarrollo corto:

1. Con la ayuda de una herramienta, el desarrollador escribe en primer lugar una prueba que provoca un fallo, ya que corresponde a una mejora, una nueva funcionalidad o a un defecto.
2. Entonces escribe el código para pasar la prueba y se ejecuta el conjunto de pruebas. Si se produce un fallo se modifica el código para pasar la prueba.
3. Al principio sólo se ejecuta para que el código pase las pruebas, para mejorar más tarde si es necesario (refactoring).

- Ideada o “redescubierta” por K. Beck y relacionada con los conceptos de la “programación con pruebas primero” (test-first programming).

## TDD. Ciclo de trabajo

## TDD. Ciclo de trabajo



- Para cada sesión de trabajo el programador define una lista de pruebas (según funcionalidad pendiente de implementar) y de refactorings.
- Cuando se escribe una prueba, el programador toma la perspectiva de “cliente”: debe comprender la interfaz y el requisito.
- Se debe escribir el código más sencillo que haga que la prueba funcione.
- Cuando se escribe nuevo código para pasar una prueba se puede:
  - detectar que el diseño es inapropiado y es necesario un refactoring.
  - descubrir un nuevo requisito que se añade a la lista
- Una prueba de integración y de regresión de un sistema supone ejecutar las pruebas unitarias para cada clase.
- La automatización de las pruebas unitarias y de integración es crucial para que realmente se escriban.

## TDD. Patrones

- Buenas prácticas de diseño en TDD

- Responden a cuestiones como cuándo probar, qué probar, cómo escribir una prueba, qué datos usar en la prueba, cuándo acabar de escribir pruebas.

- Isolated test – Test List – Test first – Assert first
- Test data – Starter test – Regression test – Mock object

## TDD. Patrón Isolated Test

- ¿Cómo deberían los resultados de un test afectar a los otros?
- No deben afectar de ninguna manera.
- Los tests deben ser capaces de ignorarse unos a otros totalmente.
- Eso implica que deben ser independientes del orden en que se ejecuten.
- “Si un test falla tengo un problema, si fallan dos tests tengo dos problemas”

## TDD. Patrón Assert first

## TDD. Patrón Assert first

- ¿Cuándo escribo los assertos? Lo primero de todo

**Ejemplo:** Una aplicación se comunica con un servicio externo a través de un socket. Una vez realizada la comunicación se debe cerrar el socket y se debe leer la cadena “abc”.

```

testCompleteTransaction () {
    ...
    assertTrue(reader.isClosed());
    assertEquals("abc", reply.contents());
}

testCompleteTransaction () {
    ...
    Buffer reply = reader.contents();
    assertTrue(reader.isClosed());
    assertEquals("abc", reply.contents());
}
  
```

```

testCompleteTransaction () {
    ...
    Socket reader = new Socket ("localhost", defaultPort());
    Buffer reply = reader.contents();
    assertTrue(reader.isClosed());
    assertEquals("abc", reply.contents());
}
  
```

```

testCompleteTransaction () {
    Server writer = new Server(defaultPort(), "abc");
    Socket reader = new Socket ("localhost", defaultPort());
    Buffer reply = reader.contents();
    assertTrue(reader.isClosed());
    assertEquals("abc", reply.contents());
}
  
```

## TDD. Patrón Test Data y Evident Data

### • ¿Qué datos uso en una prueba?

- Usa datos tales que la prueba sea fácil de leer y comprender.
- No disperses los datos
- No uses la misma constante con diferentes significados.
- Haz que las relaciones entre datos sean evidentes

Ejemplo: convertir de dólares a libras : 100\$ = 50 libras (la mitad) y hay un 1.5% de comisión

#### Escribir:

```
Exchange banco = new Exchange();
banco.addCambio("USD", "GBP", 2); banco.comision(0.0015);
Money result = banco.convertir(new Money(100,"USD"), "GBP");
assertEquals(new Money(100/2*(1-0.0015), "GBP"), result);
```

#### en vez de:

```
Exchange banco = new Exchange();
banco.addCambio("USD", "GBP", STANDARD_RATE);
banco.comision(STANDARD_COMMISSION);
Money result = banco.convertir(new Money(100,"USD"), "GBP");
assertEquals(new Money(49.25, "GBP"), result);
```

## TDD. Ventajas (asociadas a test-first)

- ¡Se escriben las pruebas!
- Satisfacción del programador: ¡He superado la prueba!
- Confianza del programador.
- Disminuye stress.
- Ayudan a comprender mejor las interfaces y comportamiento
- No hay miedo a los cambios: ¡existen cientos de pruebas de unidad!
- Menor tiempo dedicado a la depuración
- Mayor calidad: modularidad, extensibilidad, errores se detectan pronto.
- Mayor productividad aunque se escriben pruebas, código con menos defectos.

## TDD. Inconvenientes

- Las pruebas son código por lo que pueden contener errores.
- Complicado automatizar pruebas unitarias para aplicaciones con concurrencia, aplicaciones distribuidas, GUI y bases de datos.
- Ejecutar el conjunto de pruebas (test suite) puede requerir un tiempo considerable.
- Organizarlo en varios niveles
- El conjunto completo se ejecuta por la noche
- Requiere una disciplina de todo el equipo.
- Requiere mantenimiento de los tests.

## JUnit

- JUnit es framework para pruebas unitarias en Java creado por Erich Gamma y Kent Beck.
- Software "Open Source" distribuido bajo la licencia "Common Public" 1.0.
- Facilita la creación y ejecución de pruebas unitarias, de integración y regresión.
- JUnit es un framework basado en patrones de diseño (Command, Composite, Estrategia, Método Plantilla, Método Factoría,...) y que usa la introspección de Java.
- JUnit 4.x utiliza las anotaciones e importación estática de Java 5.0 en vez de introspección.
- JUnit 5.x adapta el framework a Java 8

## Ejemplo prueba unitaria (JUnit 4.x)

```
import static org.junit.Assert.*;
import org.junit.Test;

class VentaTest {
    @Test
    public void getTotal() {
        //Crear objetos sobre los que ejecutar test
        Dinero precio = new Dinero(2.5);
        ItemID id = new ItemId(1);
        Producto p = new Producto(id, precio, "producto 1");
        Venta venta = new Venta();
        venta.addLineaVenta(p,1);
        venta.addLineaVenta(p,2);
        //Establecer resultado
        Dinero total = new Dinero(7.5);
        //Ejecutar assertos del test
        assertEquals(venta.getTotal(), total);
    }
}
```

## TDD. Patrón Mock object

- ¿Cómo probar una clase que depende otra a la que no podemos acceder (p.e., recurso costoso o sólo se conoce interfaz)

- Creamos una versión "falsa" (mock) que retorna constantes.
- Ver [www.mockobjects.com](http://www.mockobjects.com)

Ejemplo: acceso a una base de datos en un servidor remoto  
En vez de crear una base de datos real, creamos un objeto que actúa como tal:

```
public void testOrderLookup() {
    Database db = new MockDatabase();
    db.expectQuery("select order_no from Order
                    where cust_no is 123");
    db.returnResult(new String[] {"Order 2", "Order 3"});
    ...
}
```

- Automatizar creación de mocks: JMocks, EasyMock

## JUnit. Asertos

- Las comprobaciones de errores se realizan mediante assertos.
  - Diferentes tipos de assertos:
    - Dado el valor esperado y el valor real compara si son iguales o distintos
    - Comprueba si una condición es verdadera o falsa
    - La mayoría de los assertos admiten un primer parámetro opcional para mostrar un mensaje en caso de que falle la comparación.
- Ejemplo:
- assertEquals("Prueba 1:", a, b);
  - Cuando se comparan valores reales se puede especificar un valor de precisión para los decimales. Ejemplo:
    - assertEquals("Comparar reales:", 3.141592, 3.14889, 0.001) ¡error!
    - assertEquals("Comparar reales:", 3.141592, 3.14889, 0.01) ¡ok!
  - El método assertEquals(oa, ob) utiliza el método equals() en la clase de los parámetros para realizar la comparación entre los objetos.

# Si estás en tu **spending era...**

mejor tener una app que te diga en qué tiendas se ha quedado registrada tu tarjeta.

¡Como la app de ING!

Saber más



## Aserciones JUnit 4.0

<code>assertArrayEquals(Object[] expected, Object[] actuals)</code>	Asserts that two object arrays are equal.
<code>assertArrayEquals(long[] expected, long[] actuals)</code>	
Resto de tipos: Boolean, short, ...	
<code>assertEquals(Object expected, Object actual)</code>	Asserts that two objects are equal.
<code>assertEquals(double expected, double actual, double delta)</code>	Asserts that two doubles are equal to within a positive delta.
<code>assertNotEquals(String message, Object unexpected, Object actual)</code>	Asserts that two object are not equal
<code>assertFalse(boolean condition)</code>	Asserts that a condition is false.
<code>assertTrue(boolean condition)</code>	Asserts that a condition is true.
<code>assertNotNull(Object object)</code>	Asserts that an object isn't null.
<code>assertNull(Object object)</code>	Asserts that an object is null.
<code>assertSame(Object expected, Object actual)</code>	Asserts that two objects refer to the same object.
<code>fail(String message)</code>	Fails a test with the given message.
<code>@Test public void method()</code>	Identifica que el método es un método de prueba
<code>@Before public void method()</code>	Cuando vaya a ejecutar cada una de las pruebas, ejecutará <code>method()</code> antes
<code>@After public void method()</code>	Ejecutará <code>method()</code> después de ejecutar cada método de prueba
<code>@BeforeClass public void method()</code>	Ejecutará <code>method()</code> una sola vez antes de comenzar con los métodos de prueba. Ej: conectararse a una base de datos
<code>@AfterClass public void method()</code>	Ejecutará <code>method()</code> una vez que hayan finalizado todos los métodos de prueba. Ej: desconectarse de una base de datos
<code>@Ignore</code>	Ignora el método de prueba. Es útil si hemos cambiado el código de nuestra clase yb aún no hemos adaptado el método de prueba a los cambios
<code>@Test(expected=IllegalArgumentException.class)</code>	Comprueba si el método finaliza lanzando una excepción (en el ejemplo se supone que el test es correcto si finaliza con la excepción <code>IllegalArgumentException</code> )
<code>@Test(timeout=100)</code>	La prueba falla si el método tarda más de 100 milisegundos en terminar.

simulan el funcionamiento de las clases no creadas.

• JMock e EasyMock son herramientas que ayudan a escribir mocks en Java.

## EasyMock. Ejemplo

(<https://www.vogella.com/tutorials/EasyMock/article.html>)

```
public enum Position {  
    BOSS, PROGRAMMER, SURFER  
}  
public interface ICalcMethod {  
    double calc(Position position);  
}  
public class IncomeCalculator {  
    private ICalcMethod calcMethod;  
    private Position position;  
  
    public void setCalcMethod(ICalcMethod calcMethod) {  
        this.calcMethod = calcMethod;  
    }  
    public double calc() {  
        if (calcMethod == null) {  
            throw new RuntimeException("CalcMethod not yet maintained");  
        }  
        if (position == null) {  
            throw new RuntimeException("Position not yet maintained");  
        }  
        return calcMethod.calc(position);  
    }  
}
```

```
public class IncomeCalculatorTest {  
    private ICalcMethod calcMethod;  
    private IncomeCalculator calc;  
  
    @Before  
    public void setUp() throws Exception {  
        // NiceMocks return default values for unimplemented methods  
        calcMethod = createNiceMock(ICalcMethod.class);  
        calc = new IncomeCalculator();  
    }  
    @Test  
    public void testCalc1() {  
        // Setting up the expected value of the method call calc  
        expect(calcMethod.calc(Position.BOSS))  
            .andReturn(70000.0).times(2);  
        expect(calcMethod.calc(Position.PROGRAMMER))  
            .andReturn(50000.0);  
        // Setup is finished need to activate the mock  
        replay(calcMethod);  
    }  
}
```

## EasyMock. Ejemplo

## Ejemplo prueba unitaria (JUnit 4.x)

```
import static org.junit.Assert.*;  
public class ListaTest {  
    @Test  
    public void pruebaOrdenar() {  
        //Objeto sobre el que vamos a hacer las pruebas  
        String[] elemes={"e", "d", "c", "b", "a"};  
        listaAlReves=new Lista(elemes);  
        //Objeto con resultado esperado  
        String[] expected={"a", "b", "c", "d", "e"};  
        Lista result = new Lista(e2);  
        //Ejecutamos aserto que comprueba test  
        assertEquals(result, listaAlReves.ordenar());  
    }  
}
```

• Es posible agrupar pruebas unitarias en una “suite” de modo que se ejecuten todas a la vez.

– Utilidad para pruebas de integración y regresivas.

– Organizar pruebas unitarias en un árbol de suites.

– Para crear una suite:

@SuiteClasses({ PruebaListas.class, PruebasListaCircular.class })

JUnit. Mock

• A veces es necesario realizar pruebas sobre clases que utilizan otras clases que aún no han sido implementadas o cuyo acceso es costoso o complejo.

• Las clases mock ayudan en estos casos para crear objetos que

```
import static org.easymock.EasyMock.createNiceMock;  
....
```

```
// ICalcMethod is the object which is mocked  
ICalcMethod calcMethod = createNiceMock(ICalcMethod.class);
```

```
// setup the mock object  
expect(calcMethod.calc(Position.BOSS)).andReturn(70000.0).times(2);  
expect(calcMethod.calc(Position.PROGRAMMER)).andReturn(50000.0);  
// Setup is finished need to activate the mock  
replay(calcMethod);
```

## EasyMock. Ejemplo

```
@Test  
public void testCalc1() {  
    // Setting up the expected value of the method call calc  
    expect(calcMethod.calc(Position.BOSS))  
        .andReturn(70000.0).times(2);  
    expect(calcMethod.calc(Position.PROGRAMMER))  
        .andReturn(50000.0);  
    // Setup is finished need to activate the mock  
    replay(calcMethod);  
  
    calc.setCalcMethod(calcMethod);  
    try {  
        calc.calc();  
        fail("Exception did not occur");  
    } catch (RuntimeException e) {}  
    calc.setPosition(Position.BOSS);  
    assertEquals(70000.0, calc.calc(), 0);  
    assertEquals(70000.0, calc.calc(), 0);  
    calc.setPosition(Position.PROGRAMMER);  
    assertEquals(50000.0, calc.calc(), 0);  
    calc.setPosition(Position.SURFER);  
    verify(calcMethod);  
}
```

WUOLAH



## Tema 5: Desarrollo de Software basado en componentes

### Desarrollo de software. Un proceso industrial

- Principal objetivo de la ingeniería del software: Conseguir un enfoque industrial: "capacidad de producir software de alta calidad a bajo coste"
- La industria del software ha satisfecho unas demandas cada vez más exigentes con un uso intensivo de tareas manuales.
- Avances en abstracción, reutilización, automatización y estándares, aunque no es suficiente.

### Innovación en desarrollo de software

#### v Abstracción

- ◆ Lenguajes de programación: Procedurales, Funcionales, OO,..
- ◆ Lenguajes de modelado y específicos del dominio (DSL)
- v Automatización
- ◆ Compiladores
- ◆ Generadores de código (por ejemplo: diseñadores de GUIs)
- v Reutilización
- ◆ Librerías de rutinas
- ◆ Clases y Herencia
- ◆ Librerías de clases
- ◆ Frameworks
- ◆ Componentes
- ◆ Líneas de Producto Software

### Reutilización

- v No empezar cada nuevo proyecto desde cero sino reutilizar elementos (assets) existentes.
  - ◆ Construir para la reutilización y construir con reutilización
- v Dimensiones del proceso de reutilización
  - ◆ Abstracción
    - v Unidad modular de reutilización (rutina, clase, componente,..)
  - ◆ Selección
    - v Los assets deben ser especificados para su búsqueda
  - ◆ Adaptación
    - v Los assets pueden necesitar cambios
  - ◆ Integración
    - v Los assets deben ser integrados dentro de una aplicación

### Extensibilidad

- v La evolución del software supone una parte muy importante del coste de una aplicación.
- ◆ Eliminar errores, nueva funcionalidad, mejoras de eficiencia,..
- v Interesa que:
  - ◆ Partes de una aplicación puedan ser construidas/mantenidas por terceros.
    - v Pueden ofrecer soluciones de mayor calidad
    - v Se encargan del mantenimiento
- v Arquitecturas basadas en partes (componentes) reemplazables.

### Concepto de componente

- v El desarrollo basado en componentes (Component-based development, CBD) consiste en utilizar componentes software existentes en la implementación de nuevas aplicaciones.
- ◆ Integrar partes más pequeñas, independientes, ya probadas, provenientes de terceros (o de la propia empresa).

### Algunas analogías podrían ser:

- montaje de equipos electrónicos a partir de hardware de distintos fabricantes (ejemplo: ensamblaje de un PC clónico)
- construcción de automóviles en una línea de producción
- montaje de una construcción de piezas de lego
- v La idea ya se aplica con éxito en otras ingenierías como la electrónica.
  - ◆ Se construye un nuevo dispositivo electrónico integrando los componentes adecuados
  - ◆ Existen fabricantes de dispositivos y fabricantes de componentes
  - ◆ Se obtiene un producto final de calidad
  - ◆ Los plazos de fabricación son cortos
- v Estas analogías están bien pero ... el software tiene una naturaleza diferente.
- v Dificultad de ensamblar componentes de diferentes plataformas (o versiones)
  - θ Debería existir un marketplace y muchos estándares.
- v Dificultad de especificar las propiedades de un componente necesarias para la interacción (qué hacen, dependencias, recursos necesarios, vulnerabilidades, etc.)
- v Dificultad de adaptar los componentes.
- v Han surgido técnicas para resolver los problemas :servicios web, auto-descripción, adaptación al contexto, ...

• El DBC se basa en la noción de componente, un bloque modular para la construcción de software.

• Diferentes definiciones con elementos comunes:

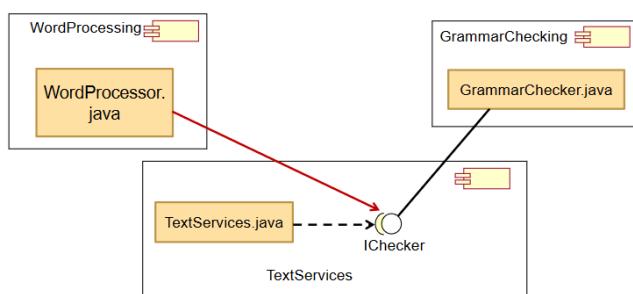
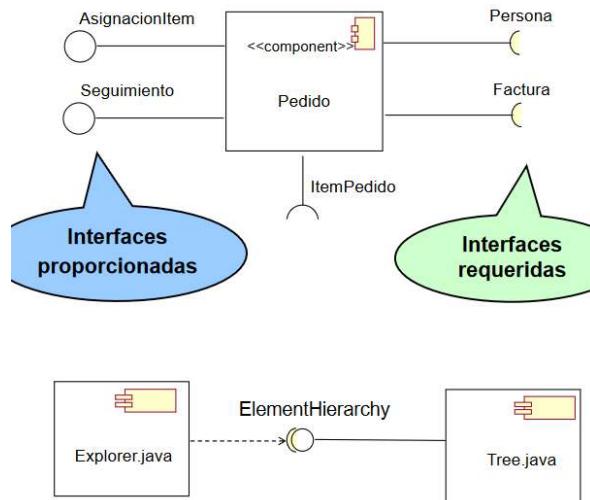
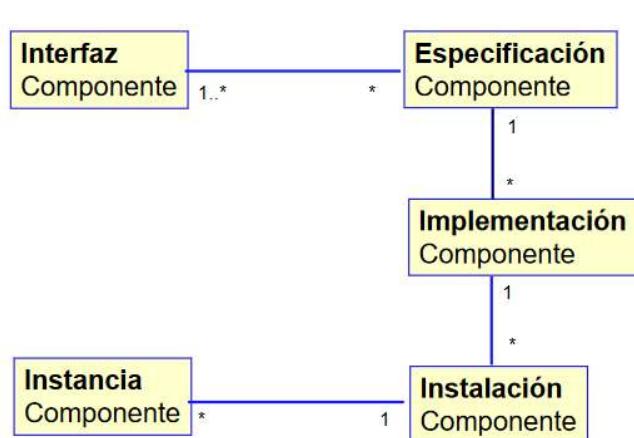
◦ Una parte modular, desplegable y reemplazable que encapsula su implementación y muestra un conjunto de interfaces" [OMG, 2001, componentes en UML]

◦ "Software orientado al cliente que permite ser utilizado por otros elementos de software sin intervención de los creadores. Implica una completa especificación de su comportamiento y funcionalidad." [B. Meyer,1999]

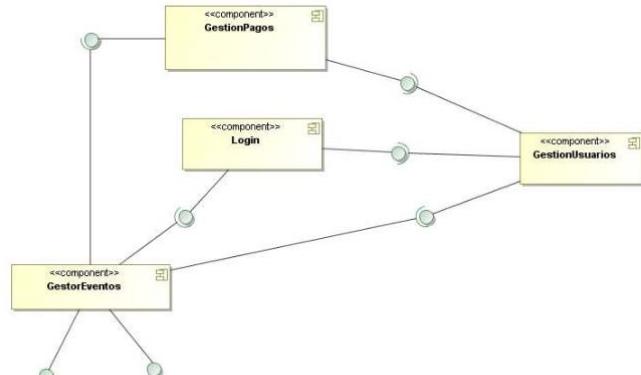
◦ "Unidad binaria de composición de aplicaciones software que posee un conjunto de requisitos, y que ha de poder ser desarrollado, adquirido, incorporado al sistema y compuesto con otros componentes, de forma independiente en tiempo y espacio." [C.Szyperski]

#### Características de un componente

- Es una parte modular de un sistema que encapsula el estado y comportamiento de un conjunto de elementos software (por ejemplo, clases). Realiza una función bien definida
- Es una unidad de despliegue independiente. En una aplicación dada existirá una única copia.
- Especifica un contrato de los servicios que proporciona y de los que requiere en términos de interfaces requeridas y proporcionadas. Especificación vs. Implementación
- Es una unidad reemplazable que se puede sustituir en tiempo de diseño o ejecución por otro componente que ofrezca la misma funcionalidad en base a la compatibilidad de sus interfaces.
- Creado conforme a un modelo de componentes



#### DBC: Composición orientada a servicios



#### Casos de éxito relacionados con componentes

• Existen tecnologías que muestran algunas de las características que tendría una tecnología de componentes exitosa.

◦ Controles de Microsoft Visual Basic

◦ Arquitecturas basadas en plugins (p.e. Eclipse, Maven)

◦ apps para plataformas móviles

◦ Portlets para aplicaciones web

◦ Servicios web

◦ SOA

◦ Microservicios (interés creciente)

◦ Repositorios de código fuente (e.g. Google code, sourceforge, Assembla, ibiblio,...)

• Interfaz de un componente

• Determina tanto las operaciones que el componente implementa como las que precisa utilizar de otros componentes durante la ejecución.

• Contratos

• Especificación que se añade a la interfaz de un componente y que establece las condiciones de uso e implementación que ligan a los clientes y proveedores del componente.

• Interfaces con pre y postcondiciones • Secuencias de interacciones válidas • Requisitos no funcionales: tiempo, almacen, fiabi

Portátiles desde

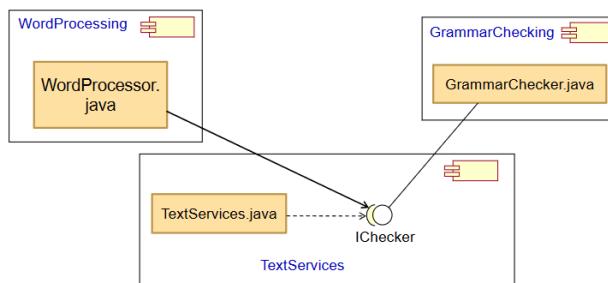
549€



msi  
BLACK FRIDAY

- ❖ Composición tardía
- ❖ Composición en un tiempo posterior al de la compilación del componente y por alguien ajeno a su desarrollo
  - Del componente sólo se conoce su interfaz o contrato.
  - No se conocen detalles de la implementación (ocultación de información).

#### ■ Ligadura dinámica o tardía favorece la reemplazabilidad



- GrammarChecker se registra en TextServices, entonces WordProcessing adquiere una referencia a él en tiempo de ejecución si es el checker por defecto

#### ❖ Eventos

- ❖ Mecanismo de comunicación asincrónica.
- ❖ Especifican la forma en la que el componente notifica al exterior una un cambio en una condición interna.
- ❖ Reflexión
- ❖ Habilidad de un componente para conocer sus elementos.
- ❖ Modelo de componentes
- ❖ Define la forma de las interfaces y los mecanismos para interconectar los componentes, ensamblaje y despliegue, y su entorno.
- Entorno de un componente
- Conjunto de recursos software que soportan a un componente. Hay dos tipos:
  - Ejecución: Ambiente (contenedor) donde se ejecuta normalmente el componente.
  - Diseño: Entorno que se utiliza para localizar, configurar, especializar y probar los componentes que van a formar parte de una aplicación.

#### Modelo de componentes

- ❖ Un modelo de componentes especifica:
  - La forma de los interfaces de sus componentes
  - Los mecanismos de composición y comunicación entre ellos
  - La forma en la que se proveen los servicios
  - La forma de empaquetamiento y despliegue
- ❖ No existe un estándar que domine la industria.
- ❖ Algunos modelos son:
  - Corba CCM de OMG (no usado)
  - COM, OLE, ActiveX (no usados) y .NET de Microsoft
  - JavaBeans (no usado) y J2EE de Sun
  - OSGi para Java
- Una plataforma de componentes ofrece un entorno para utilizar componentes con un modelo de componentes.

#### ❖ Interfaces:

- Cómo se definen (ofrecidas y proporcionadas), eventos generados, cómo se realiza la composición

#### ◦ Interfaces específicas

#### ❖ Información de uso:

- Convenciones de nombres para facilitar la introspección

#### ❖ Despliegue y uso:

- Cómo se empaquetan, Cómo se especifican, Cómo se registran en el sistema, cómo se activan, ...

#### ◦ Soporte de la evolución

- Los componentes se despliegan en un contenedor.

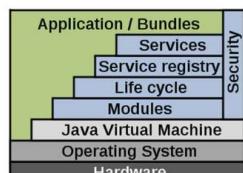
#### ◦ Conjunto de interfaces que ofrece un conjunto de servicios

- Ciclo de vida, persistencia, seguridad, transacciones, concurrencia, etc.

- Manejo de excepciones, localización, comunicaciones, etc

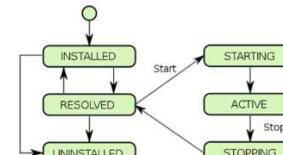
## Modelo de componentes OSGi

- Open Services Gateway Initiative (OSGi) define una arquitectura para desarrollar aplicaciones modulares en Java.
  - Conjunto de servicios que un contenedor OSGi debe implementar
  - Contrato entre aplicaciones y el contenedor
- Ejemplos de implementaciones: Eclipse Equinox, Apache Felix y Knopflerfish.



Cada componente (*plug-in* o *bundle*) define su interfaz (API) a través de un conjunto de paquetes que exporta y las dependencias requeridas.

Los componentes pueden ser dinámicamente instalados/desinstalados, activados/desactivados y actualizados.



VER OFERTAS

**WUOLAH**

Las mejores noches de Murcia. Descubre dónde empieza (y nunca acaba) la noche.