

# 16. Patrones de diseño orientado a objetos

# Índice

- Referencias
- Introducción
- Patrones GRASP
  - Introducción
  - Experto.
  - Creador.
  - Alta cohesión.

# Índice

- Bajo acoplamiento.
- Controlador.
- Polimorfismo.
- Fabricación pura.
- Indirección.
- No hables con extraños
- Nota.

# Índice

- Patrones GoF
  - Clasificación
- Patrones de creación
  - Introducción
  - Abstract factory.
  - Builder.
  - Factory method.

# Índice

- Prototype.
- Singleton.
- Patrones estructurales
  - Introducción
  - Adapter.
  - Bridge.
  - Composite.
  - Decorator.
  - Façade.

# Índice

- Flyweight.
- Proxy.
- Patrones de comportamiento
  - Introducción
  - Chain of responsibility.
  - Command.
  - Interpreter.
  - Iterator.
  - Mediator.

# Índice

- Memento.
- Observer.
- State.
- Strategy.
- Template method.
- Visitor.
- Relaciones entre patrones GoF
- Conclusiones

# Referencias

- Gamma, E., Helm, R., Johnson, R., Vlissides, J. *Patrones de Diseño: Elementos de Software Orientado a Objetos Reutilizables*. Addison-Wesley, 2006
- Larman, C. *UML y Patrones: Introducción al Análisis y Diseño Orientado a Objetos y al Proceso Unificado. 2<sup>a</sup> edición*. Prentice-Hall, 2004

# Referencias

- Stelting, S., Maassen, O. *Patrones de diseño aplicados a Java*. Pearson Educación, 2003
- Metsker S.J. *Design patterns. Java workbook*. Addison-Wesley, 2002
- Shalloway A., Trott J.R. *Design Patterns Explained. A New Perspective on Object-Oriented Design*. Addison-Wesley, 2002

# Introducción

- Según Christopher Alexander, “un *patrón* describe un problema que ocurre una y otra vez en nuestro entorno, así como la solución a ese problema de tal modo que se puede aplicar esta solución un millón de veces, sin hacer lo mismo dos veces”

# Introducción

- Aunque Alexander se refería a patrones en ciudades y edificios, lo que dice también es válido para patrones de diseño OO
- Podemos decir que los patrones de diseño:
  - Son soluciones simples y elegantes a problemas específicos del diseño de software OO.
  - Representan soluciones que han sido desarrolladas y han ido evolucionando a través del tiempo.

# Introducción

- Los patrones de diseño no tienen en cuenta cuestiones tales como:
  - Estructuras de datos.
  - Diseños específicos de un dominio.
- Son descripciones de clases y objetos relacionados que están particularizados para resolver un problema de diseño general en un determinado contexto

# Introducción

- Cada patrón de diseño identifica:
  - Las clases e instancias participantes.
  - Los roles y colaboraciones de dichas clases e instancias.
  - La distribución de responsabilidades
- Veremos dos familias de patrones:
  - GRASP\*, de Craig Larman.
  - Los patrones GoF#, de Eric Gamma et al.

\*General Responsibility Assignment Software Patterns

#Gang of Four

# Patrones GRASP

## Introducción

- Un sistema orientado a objetos se compone de objetos que envían mensajes a otros objetos para que lleven a cabo operaciones
- La calidad de diseño de la interacción de los objetos y la asignación de responsabilidades presentan gran variación

# Patrones GRASP

## Introducción

- Las decisiones poco acertadas dan origen a sistemas y componentes frágiles y difíciles de mantener, entender, reutilizar o extender
- Los patrones GRASP recogen principios/directrices para obtener buenos diseños OO que se aplican al preparar los diagramas de interacción y/o asignar responsabilidades
- Son menos concretos que los patrones GoF

# Patrones GRASP

## Introducción

- Los patrones GRASP son:
  - Experto.
  - Creador.
  - Alta cohesión.
  - Bajo acoplamiento.
  - Controlador.
  - Polimorfismo.
  - Fabricación pura.
  - Indirección.
  - No hables con extraños.

# Patrones GRASP

## Experto

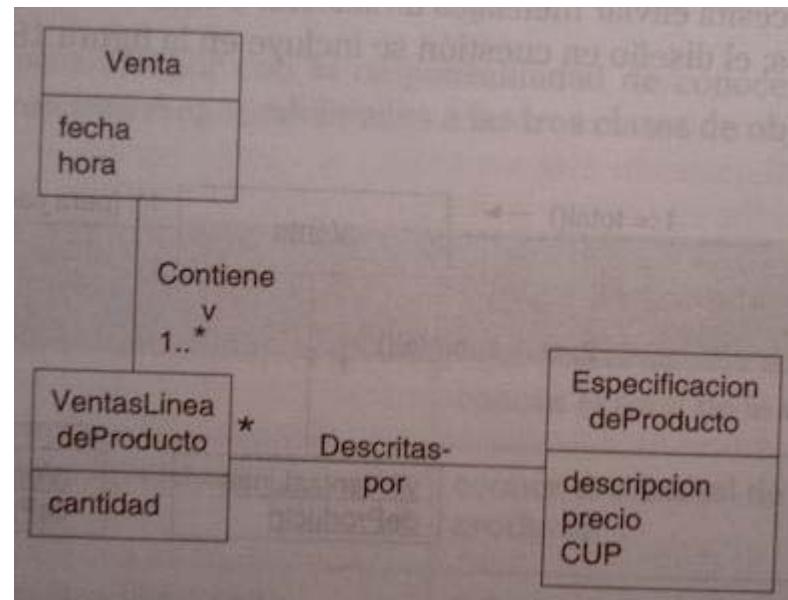
- Problema: ¿cuál es el principio fundamental en virtud del cual se asignan las responsabilidades en el diseño OO?
- Solución: asignar una responsabilidad al *experto en información*: la clase que cuenta con la información necesaria para cumplir la responsabilidad

# Patrones GRASP

## Experto

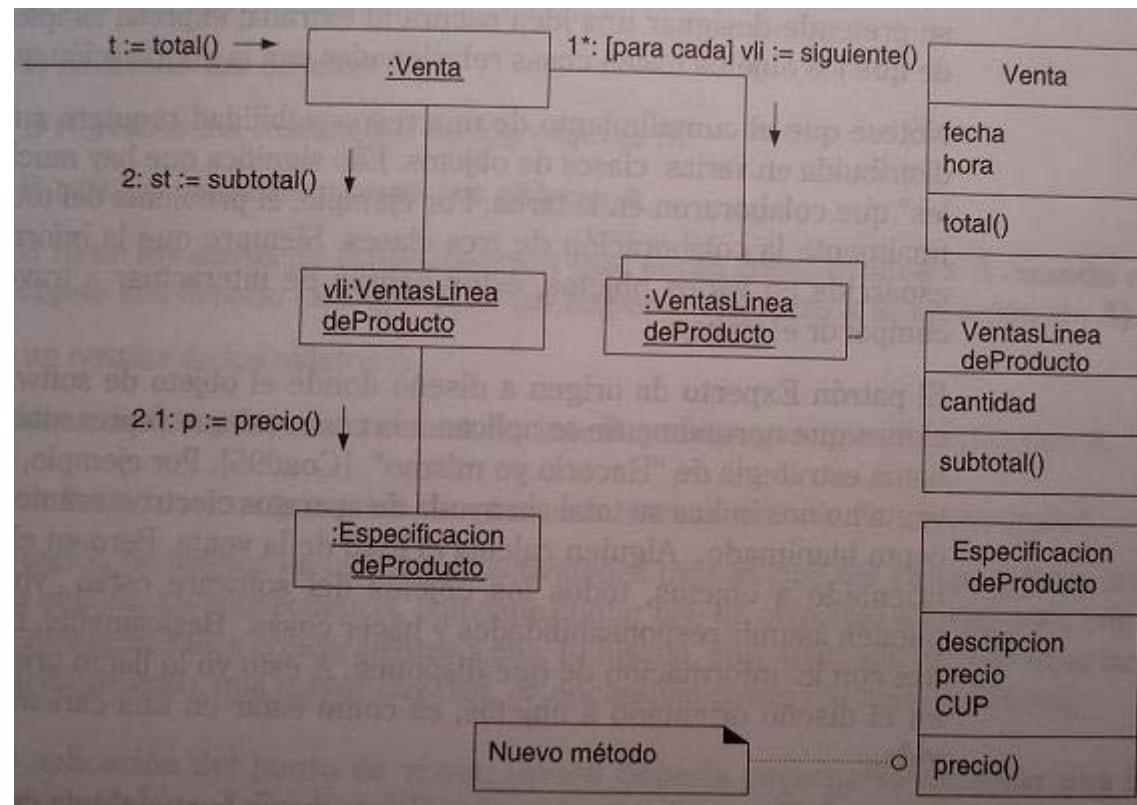
- Ejemplo
  - Supongamos que tenemos una aplicación que gestiona ventas.
  - Queremos calcular el total de una venta sobre las siguientes clases:

Asociaciones  
de Venta



# Patrones GRASP

## Experto



Cálculo del total de la venta

# Patrones GRASP

## Experto

- Es decir, se ha decidido el siguiente reparto de responsabilidades:
  - Venta: conoce el total de la venta.
  - VentasLineadeProducto: conoce el subtotal de la línea de producto.
  - EspecificacióndeProducto: conoce el precio del producto.

# Patrones GRASP

## Experto

- Comentarios
  - Experto es el patrón GRASP más usado.
  - Expresa la idea de que los objetos hacen cosas relacionadas con la información que poseen.
  - El cumplimiento de una responsabilidad frecuentemente requiere información distribuida en varias clases → puede haber expertos parciales.

# Patrones GRASP

## Experto

- Beneficios
  - Se conserva el encapsulamiento → bajo acoplamiento.
  - Promueve clases sencillas y cohesivas que son más fáciles de mantener y comprender

# Patrones GRASP

## Creador

- Problema: ¿Quién debería ser responsable de crear una nueva instancia de alguna clase?
- Solución: La clase B es responsable de crear una instancia de la clase A en uno de los siguientes casos:
  - B *agrega* los objetos A.
  - B *contiene* los objetos A.

# Patrones GRASP

## Creador

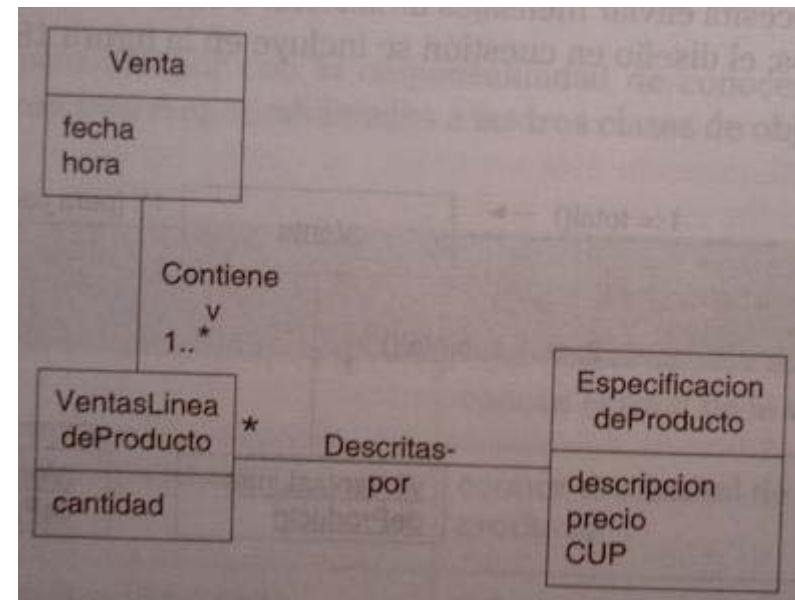
- B *registra* a las instancias de los objetos A.
- B *utiliza* específicamente los objetos A.
- B *tiene* los *datos de inicialización* que serán transmitidos a A cuando este objeto sea creado (B es un experto respecto a la creación de A).

# Patrones GRASP

## Creador

- Ejemplo
  - En la aplicación de punto de venta, ¿quién debería encargarse de crear una instancia de VentasLineadeProducto?

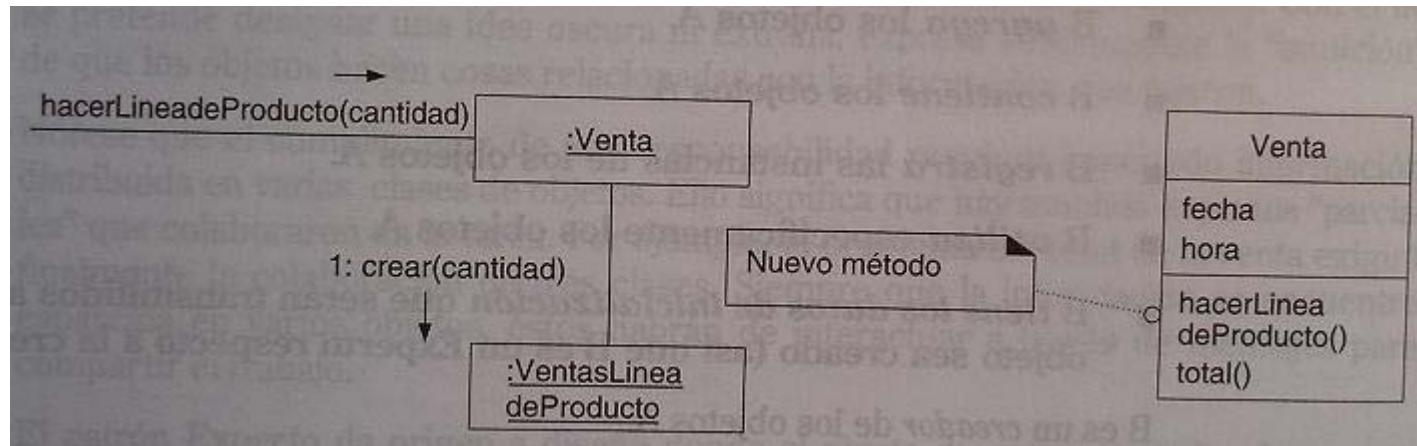
Asociaciones  
de Venta



# Patrones GRASP

## Creador

- Una Venta contiene muchos objetos VentasLineadeProducto, por lo que es idónea para asumir la responsabilidad de la creación.



Creación de un objeto `VentasLineadeProducto`

# Patrones GRASP

## Creador

- Comentarios
  - El patrón creador guía la asignación de responsabilidades relacionadas con la creación de objetos.
  - El propósito fundamental es encontrar un creador que debemos conectar con el objeto producido, dando soporte a un bajo acoplamiento.
  - Nótese que el creador es un experto en creación.

# Patrones GRASP

## Creador

- Beneficios
  - Proporciona un bajo acoplamiento, pues la clase creada tiende a ser visible a la clase creador.

# Patrones GRASP

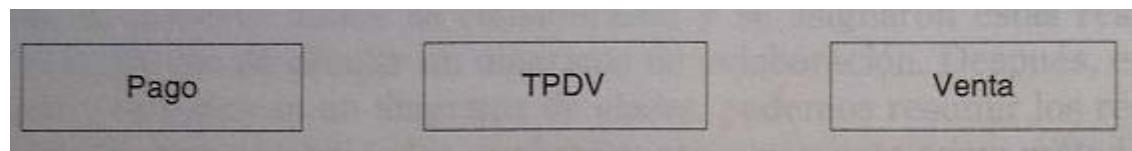
## Bajo acoplamiento

- Problema: ¿Cómo dar soporte a una dependencia escasa y a un aumento de la reutilización?
- Solución: asignar una responsabilidad para mantener bajo acoplamiento

# Patrones GRASP

## Bajo acoplamiento

- Ejemplo
  - En el ejemplo del terminal del punto de venta, ¿quién debe crear un Pago?

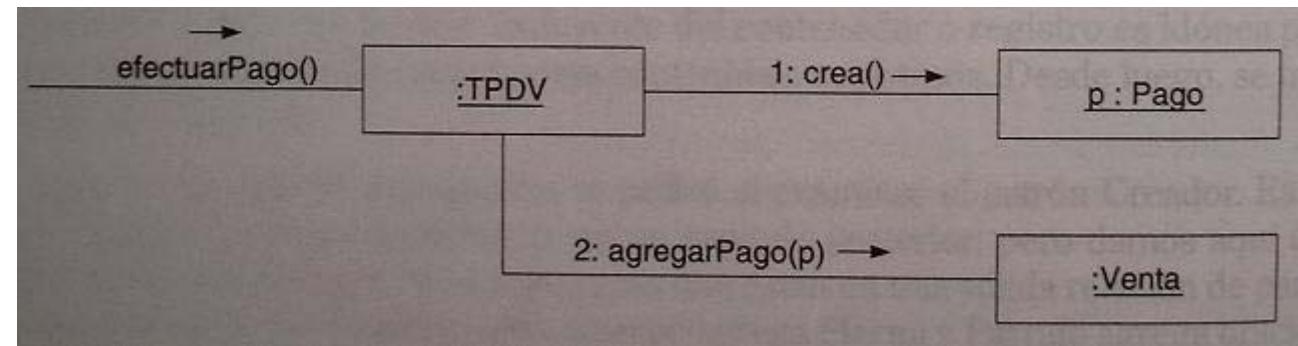


Clases involucradas

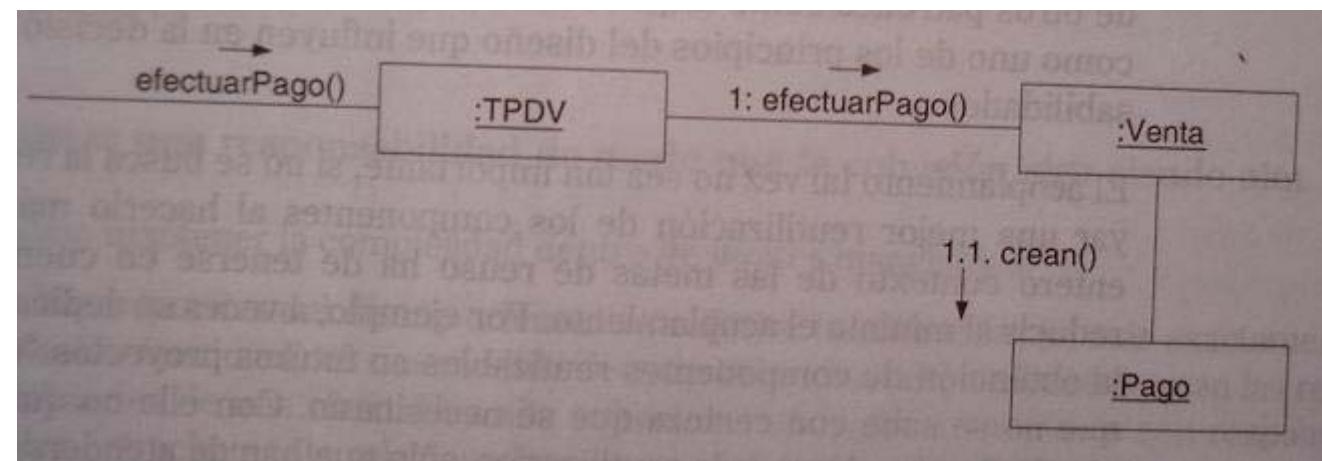
# Patrones GRASP

## Bajo acoplamiento

Diseño 1



Diseño 2



# Patrones GRASP

## Bajo acoplamiento

- El patrón experto podría recomendar el diseño 1.
- El patrón bajo acoplamiento podría recomendar el diseño 2.
- Comentarios
  - En los lenguajes orientados a objetos, las formas más comunes de acoplamiento de TipoA y TipoB son las siguientes:
    - TipoA tiene un atributo de TipoB.

# Patrones GRASP

## Bajo acoplamiento

- `TipoA` tiene un método que involucra a una instancia de `TipoB` (entrada, salida, cuerpo).
- `TipoA` es subclase de `TipoB`.
- `TipoB` es un interfaz y `TipoA` lo implementa.
- El bajo acoplamiento estimula asignar una responsabilidad de modo que su colocación no incremente el acoplamiento tanto que produzca los resultados negativos propios de un alto acoplamiento.

# Patrones GRASP

## Bajo acoplamiento

- Bajo acoplamiento soporta el diseño de clases más independientes, que reducen el impacto de los cambios, y también más reutilizables, que acrecientan la oportunidad de una mayor productividad.
- El acoplamiento no es tan importante si no se busca la reutilización (análisis coste-beneficio).
- Una subclase está acoplada con su superclase, luego la derivación será estudiada con cuidado.

# Patrones GRASP

## Bajo acoplamiento

- Beneficios
  - Los cambios en otros componentes no afectan.
  - Clases fáciles de entender por separado.
  - Clases fáciles de reutilizar.

# Patrones GRASP

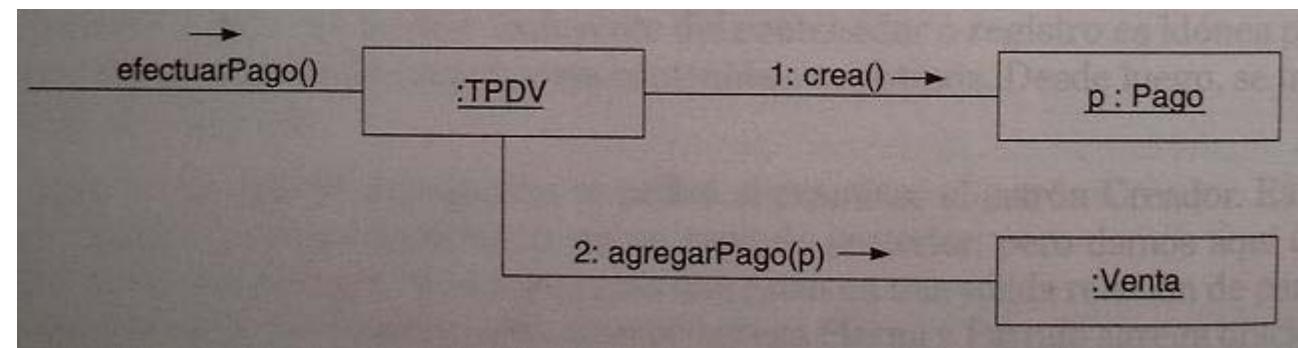
## Alta cohesión

- Problema: ¿cómo mantener la complejidad dentro de límites manejables?
- Solución: asignar una responsabilidad a una clase de modo que la cohesión siga siendo alta
- Ejemplo:
  - Consideremos los diseños anteriores:

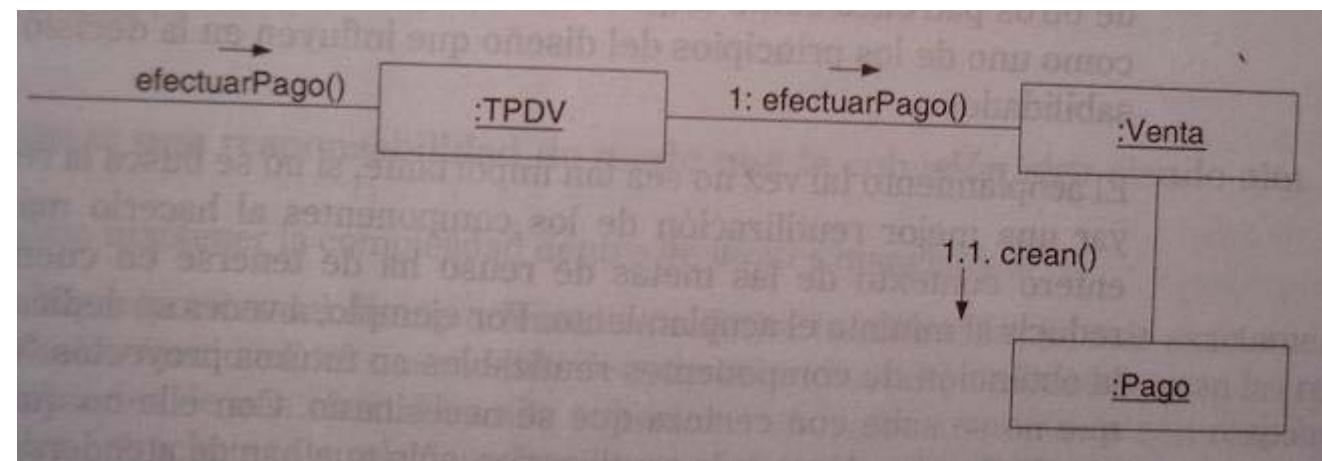
# Patrones GRASP

## Alta cohesión

Diseño 1



Diseño 2



# Patrones GRASP

## Alta cohesión

- El diseño 1 puede llevar a una clase TPDV saturada y sin cohesión
- El diseño 2 proporciona una mayor cohesión de las clases involucradas.
- Comentarios
  - Una clase de alta cohesión posee un número relativamente pequeño de responsabilidades, con una importante funcionalidad relacionada y poco trabajo por hacer. Colabora con otros objetos para compartir el esfuerzo si la tarea es grande.

# Patrones GRASP

## Alta cohesión

- Una clase con mucha cohesión es útil porque es bastante fácil darle mantenimiento, entenderla y reutilizarla.
- No tiene nada que ver una clase cohesiva con una fachada, ya que la fachada delega en las clases a las que oculta.
- Beneficios
  - Mejoran la claridad y facilidad con que se entiende el diseño.

# Patrones GRASP

## Alta cohesión

- Se simplifica el mantenimiento y las mejoras en la funcionalidad.
- A menudo se genera un bajo acoplamiento.

# Patrones GRASP

## Controlador

- Problema: ¿quién debería encargarse de atender un evento del sistema?
- Un *evento del sistema* es un evento de alto nivel generado por una actor externo, es un evento de entrada externa
- Se asocia a *operaciones del sistema*, las que emite en respuesta a los eventos del sistema.

# Patrones GRASP

## Controlador

- Solución: asignar la responsabilidad del manejo de un mensaje de los eventos de un sistema a una clase que represente una de las siguientes opciones:
  - El sistema global (controlador de fachada).
  - La empresa u organización global (controlador de fachada).

# Patrones GRASP

## Controlador

- Algo en el mundo real que es activo y que pueda participar en la tarea (controlador de tareas).
- Un manejador artificial de todos los eventos del sistema de un caso de uso, generalmente denominados “Manejador<NombreCasodeUso>” (controlador de casos de uso).

# Patrones GRASP

## Controlador

- Ejemplo
  - En el caso del terminal del punto de venta, ¿quién debería encargarse de introducir un producto?:
    - El representante del sistema: TPDV.
    - El representante de la empresa: Tienda.
    - Algo activo del mundo real: Cajero.
    - Manejador caso de uso: ManejadordeComprarProductos.

# Patrones GRASP

## Controlador



Distintas opciones para el controlador

# Patrones GRASP

## Controlador

- Comentarios
  - El patrón ofrece una guía para tomar decisiones sobre los eventos de entrada.
  - Sea como fuere, los elementos de interfaz, y sus controladores de eventos de interfaz, no deben ser responsables de controlar los eventos del sistema
  - En términos multicapa, este patrón está más cercano a un servicio de aplicación que a un controlador

# Patrones GRASP

## Controlador

- Beneficios
  - Mayor potencial de los componentes reutilizables.
  - Reflexionar sobre el estado del caso de uso.

# Patrones GRASP

## Polimorfismo

- Problema: ¿cómo manejar las alternativas basadas en el tipo?
- Solución: cuando por el tipo varían las alternativas o los comportamientos afines, las responsabilidades del comportamiento se asignarán mediante operaciones polimórficas a los tipos en el que el comportamiento presenta variantes

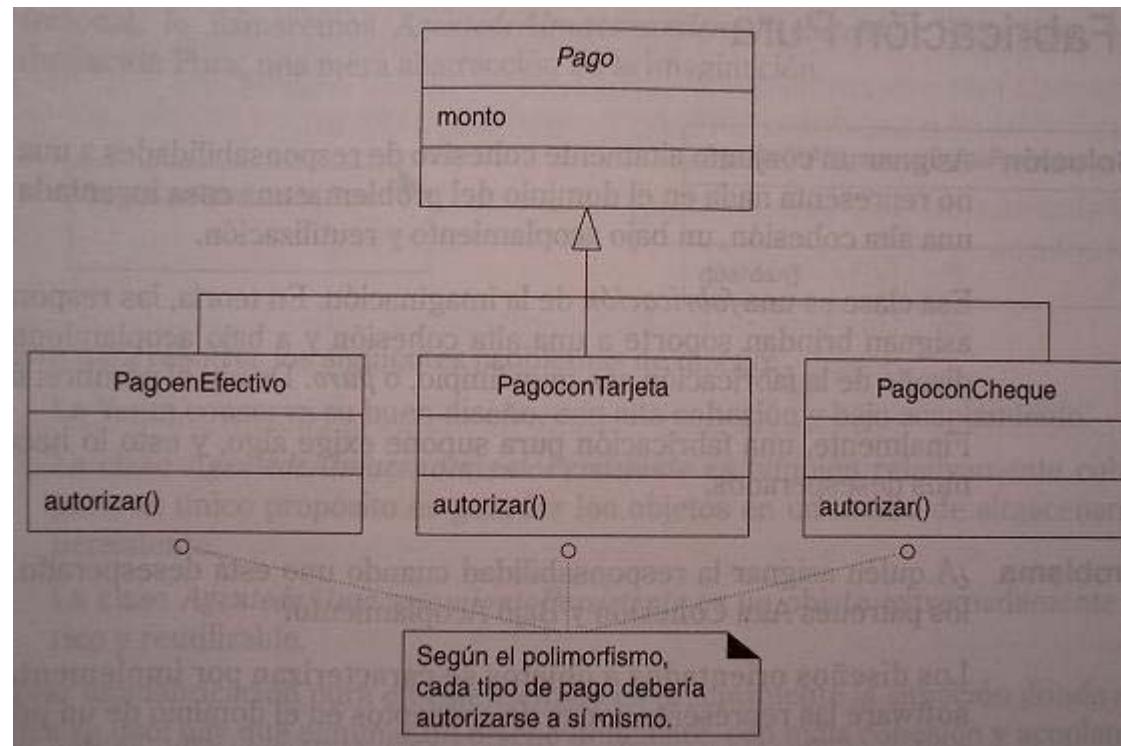
# Patrones GRASP

## Polimorfismo

- Por lo tanto no se deben realizar pruebas con el tipo de un objeto ni utilizar lógica condicional para plantear diversas alternativas basadas en el tipo
- Ejemplo
  - En el caso del terminal del punto de venta, ¿quién debería encargarse de autorizar las diversas clases de pagos?

# Patrones GRASP

## Polimorfismo



Polimorfismo en la autorización de pagos

# Patrones GRASP

## Polimorfismo

- Comentarios
  - Experto es un patrón táctico, mientras que polimorfismo es estratégico.
  - Representa un principio fundamental del modelo de objetos.
- Beneficios
  - Es fácil incluir futuras extensiones.
  - Permite tratar de manera uniforme objetos distintos.

# Patrones GRASP

## Fabricación pura

- Problema: ¿a quién asignar la responsabilidad cuando uno está *desesperado* y no quiere violar los patrones alta cohesión y bajo acoplamiento?
- Solución: asignar un conjunto altamente cohesivo de responsabilidades a una clase artificial que no representa nada en el dominio del problema

# Patrones GRASP

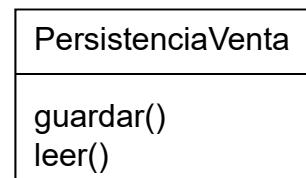
## Fabricación pura

- Ejemplo
  - Supongamos que queremos dar persistencia a las ventas.
  - Si la clase Venta es responsable de su persistencia:
    - La clase Venta reduce su cohesión.
    - La clase Venta aumenta su acoplamiento.

# Patrones GRASP

## Fabricación pura

- Es decir, aunque venta parece el experto para su persistencia, en base a la cohesión y el acoplamiento decidimos crear una clase artificial cuya finalidad única es guardar al objeto.



Fabricación pura

# Patrones GRASP

## Fabricación pura

- Nótese que en un entorno polimórfico, esta solución obliga a vincular a cada objeto con su clase de persistencia correspondiente. De otra forma habría que razonar directamente sobre el tipo.
- Comentarios
  - Clases con responsabilidades de granularidad fina.

# Patrones GRASP

## Fabricación pura

- Patrones como el adaptador, observador, visitante son ejemplos de fabricaciones puras.
- Beneficios
  - Alta cohesión.
  - Aumenta el potencial de reutilización

# Patrones GRASP

## Indirección

- Problema: ¿a quién se asignarán las responsabilidades con el fin de evitar el acoplamiento directo?
- Solución: se asigna la responsabilidad a un objeto intermedio para que medie entre otros componentes o servicios, y éstos no terminen directamente acoplados

# Patrones GRASP

## Indirección

- Ejemplo
  - La clase PersistenciaVenta es también un ejemplo de indirección.
  - Venta delega en otra clase una funcionalidad con el fin de disminuir el acoplamiento.
- Beneficios
  - Bajo acoplamiento.

# Patrones GRASP

## No hables con extraños

- Problema: ¿a quién asignar las responsabilidades para evitar conocer la estructura de los objetos indirectos?
- Solución: se asigna la responsabilidad a un objeto directo del cliente para que colabore con un objeto indirecto, de modo que el cliente no necesite saber nada del objeto indirecto

# Patrones GRASP

## No hables con extraños

- El patrón, también conocido con el nombre de *ley de Demeter*, impone restricciones a los objetos a los cuales deberíamos enviar mensajes dentro de un método
- El patrón establece que en un método, los mensajes sólo deberían ser enviados a los siguientes objetos:
  - El objeto `this`.

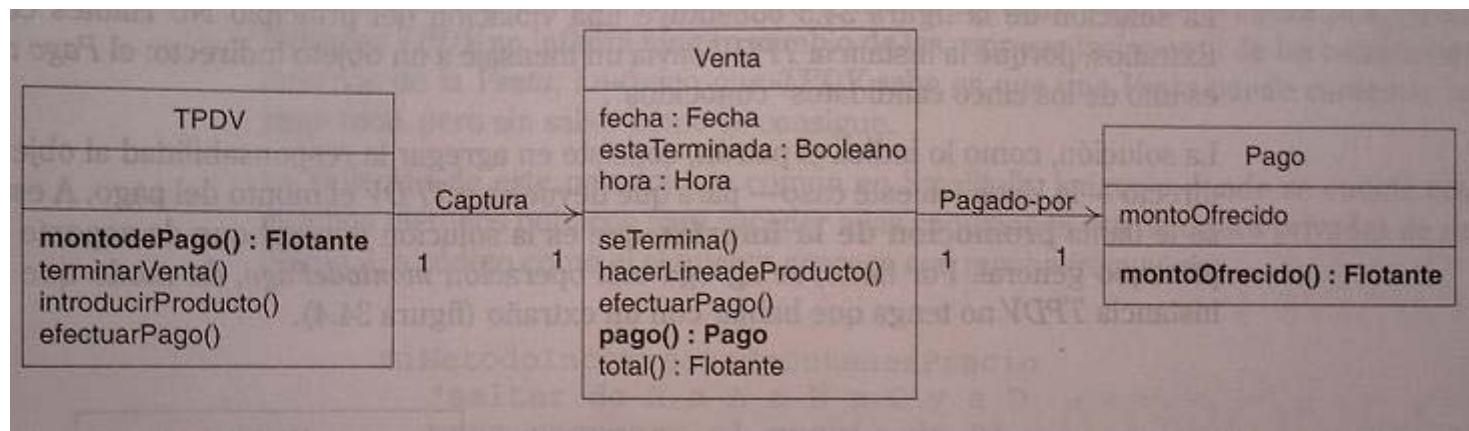
# Patrones GRASP

## No hables con extraños

- Un parámetro del método.
- Un atributo de `this`.
- Un elemento de una colección que sea atributo de `this`.
- Un objeto creado en el interior del método.
- Ejemplo
  - Supongamos que queremos conocer el total (monto) de un pago en una venta.

# Patrones GRASP

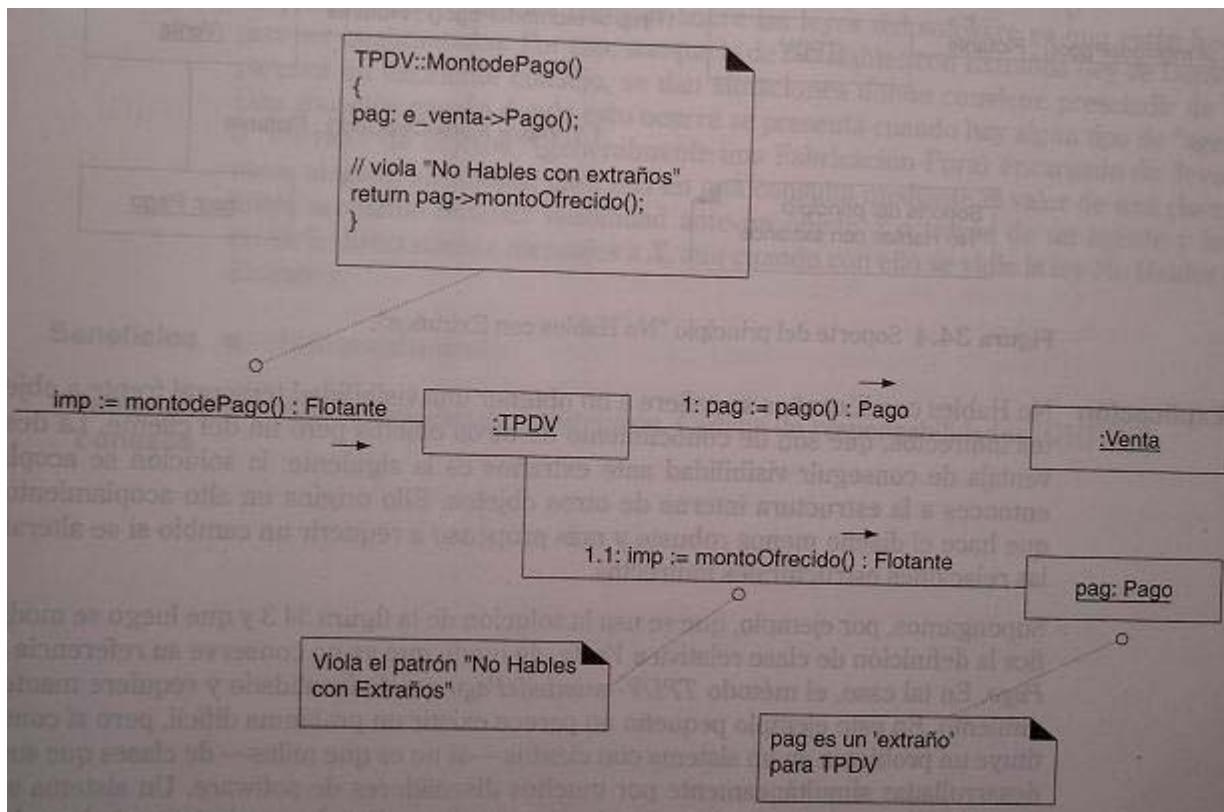
## No hables con extraños



Asociaciones entre clases

# Patrones GRASP

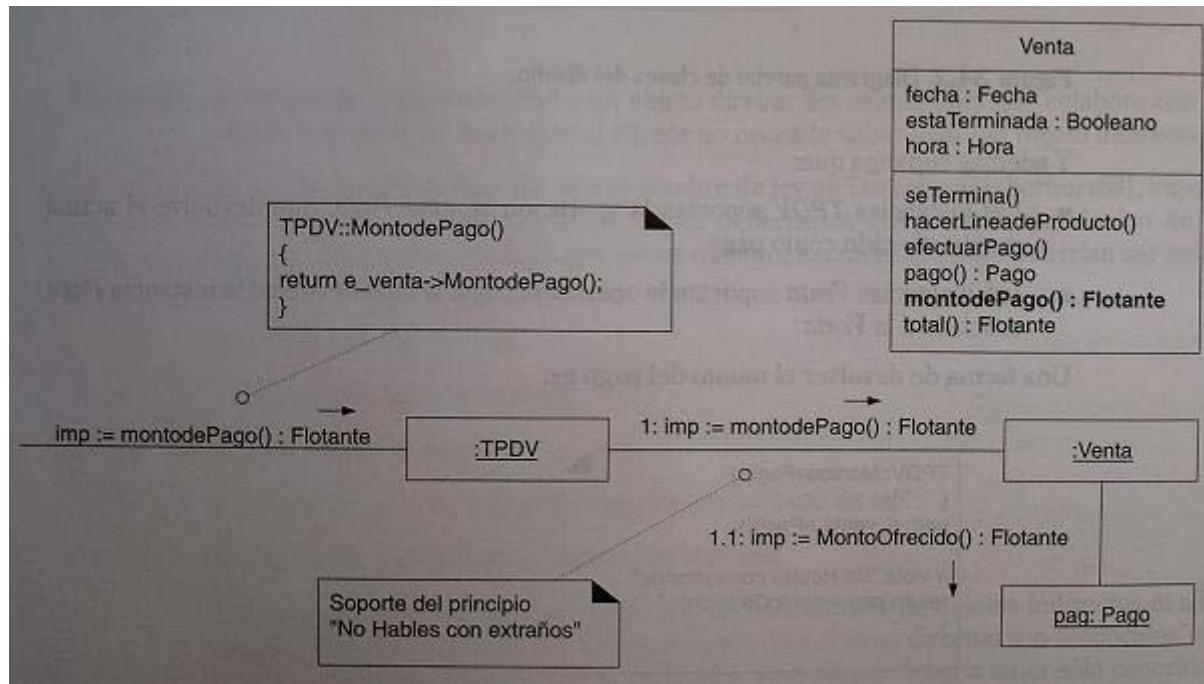
## No hables con extraños



### Violación del patrón

# Patrones GRASP

## No hables con extraños



Implementación del patrón

# Patrones GRASP

## No hables con extraños

- Venta agrega la responsabilidad de obtener el total.
- A esto se le conoce como *promoción de la interfaz*.
- Comentarios
  - El patrón evita una visibilidad temporal frente a objetos indirectos, que son de conocimiento de otros objetos, pero no del cliente.

# **Patrones GRASP**

## **No hables con extraños**

- De esta forma se evitan acoplamientos innecesarios.
- El patrón puede violarse en el caso de que el objeto intermedio sea un agente encargado de extraer datos.
- Beneficios
  - Bajo acoplamiento.

# Patrones GRASP

## Nota

- Los patrones GRASP podemos asimilarlos a *directrices* del modelo de objetos:
  - Las clases deben tener responsabilidades definidas, en base a los papeles que desempeñen.
    - Patrón experto.
    - Patrón creador.
    - Patrón controlador.

# Patrones GRASP

## Nota

- Las clases no deben ser monolíticas, deben *delegar* en otras clases para llevar a cabo su funcionalidad.
  - Patrón indirección.
- Cuando objetos ligados por tramas de herencia usen lógicas case en base al tipo, debemos utilizar polimorfismo.
  - Patrón polimorfismo.

# Patrones GRASP

## Nota

- En el proceso de análisis y diseño pueden aparecer clases que no están relacionadas con el dominio del problema, pero sí con la solución.
  - Fabricación pura.
- Las clases (y en general los módulos/subsistemas/paquetes), deben tener una alta cohesión.
  - Alta cohesión.

# Patrones GRASP

## Nota

- Las clases (y en general los módulos/subsistemas/paquetes), deben tener un bajo acoplamiento.
  - Bajo acoplamiento.
  - No hables con extraños.
- En este sentido, son más *normas* que *patrones*

# Patrones GoF

## Clasificación

- Podemos clasificar los patrones de diseño GoF en base a su:
  - *Propósito*: lo que hace el patrón
    - *Creación*: creación de objetos
    - *Estructural*: composición de clases u objetos.
    - *Comportamiento*: modo en que las clases y objetos interactúan y se reparten la responsabilidad

# Patrones GoF

## Clasificación

- *Ámbito*: especifica si el patrón se aplica principalmente a clases o a objetos
  - *Clases*: centrados en relaciones entre clases y sus subclases, es decir, relaciones estáticas de compilación.
  - *Objetos*: relaciones entre objetos, que pueden cambiarse en tiempo de ejecución y son más dinámicas

# Patrones GoF

## Clasificación

- De esta forma los patrones:
  - *creación + clases*: delegan alguna parte del proceso de creación de objetos en las subclases.
  - *creación + objetos*: delegan alguna parte del proceso de creación de objetos en otros objetos.
  - *estructurales + clases*: usan la herencia para componer clases.
  - *estructurales + objetos*: describen formas de ensamblar objetos.

# Patrones GoF

## Clasificación

- *comportamiento + clases*: usan la herencia para describir algoritmos y flujos de control.
- *comportamiento + objetos*: describen cómo cooperan un grupo de objetos para realizar una tarea que ningún objeto puede llevar a cabo por si solo.

# Patrones GoF

## Clasificación

		Propósito		
Ámbito	Clase	De Creación	Estructurales	De comportamiento
	Objeto	Factory Method (99) Abstract Factory (79) Builder (89) Prototype (109) Singleton (119)	Adapter (de clases) (131) Adapter (de objetos) (131) Bridge (141) Composite (151) Decorator (161) Facade (171) Flyweight (179) Proxy (191)	Interpreter (225) Template Method (299) Chain of Responsibility (205) Command (215) Iterator (237) Mediator (251) Memento (261) Observer (269) State (279) Strategy (289) Visitor (305)

## Patrones de diseño GoF

# Patrones GoF

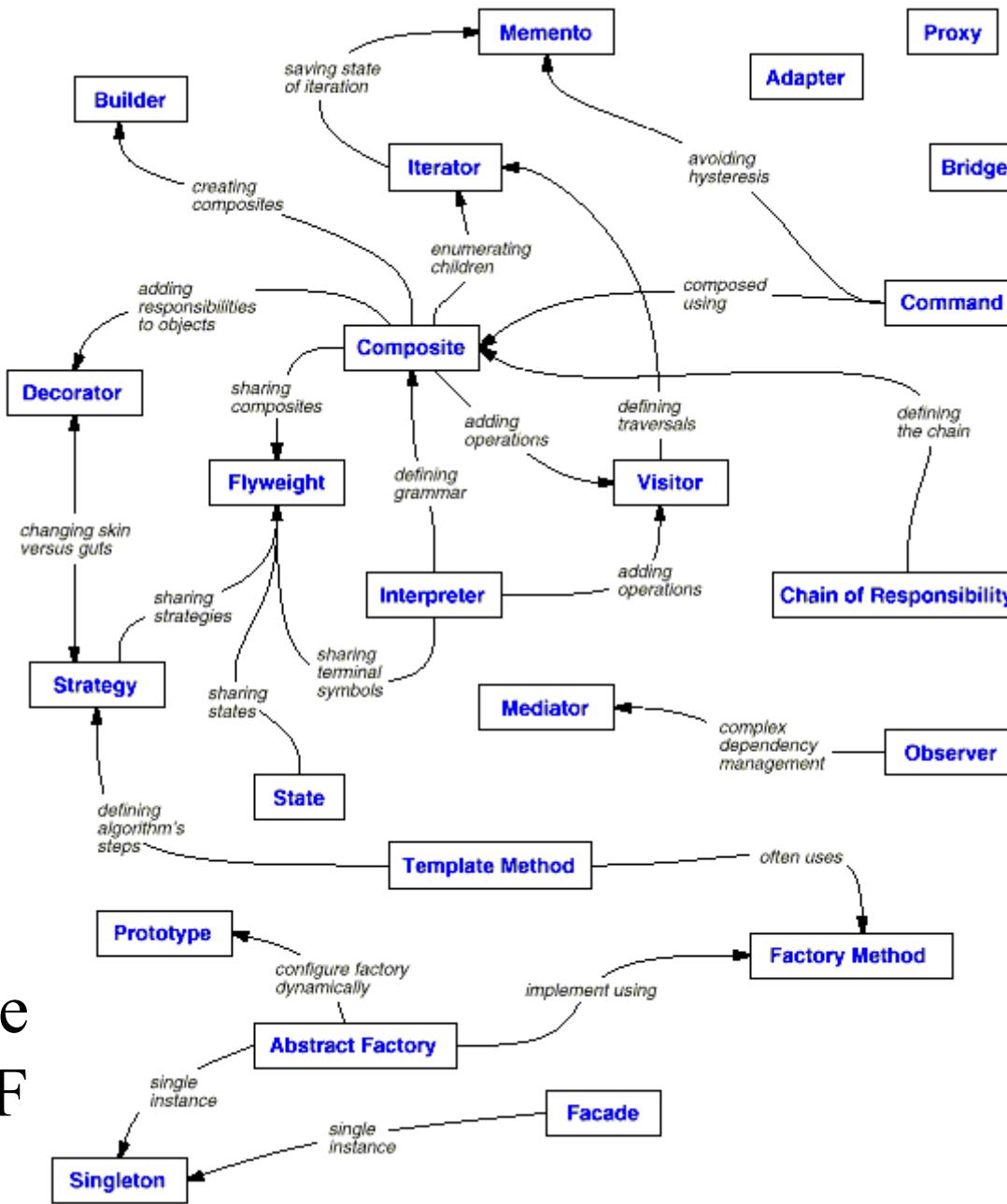
## Clasificación

- En cualquier caso, la clasificación no es única
- Otra clasificación podría hacerse es atendiendo a las relaciones existentes entre los patrones de diseño

# Relaciones entre patrones GoF

Relaciones entre patrones GoF

Ingeniería del Software  
Antonio Navarro



# Patrones GoF

## Clasificación

- Para cada patrón veremos:
  - Propósito.
  - También conocido como.
  - Motivación.
  - Aplicabilidad.
  - Descripción abstracta.
  - Consecuencias
  - Código de ejemplo.

# Patrones GoF

## Clasificación

- Nota: Los patrones extraídos del libro de GoF están en notación OMT, similar a UML, pero distinta

# Patrones de creación

## Introducción

- Los patrones de creación abstraen el proceso de creación de instancias
- Ayudan a que un sistema sea independiente de cómo se crean, componen y representan sus objetos.
- Los de clases, utilizan la herencia para cambiar la clase de la instancia a crear

# Patrones de creación

## Introducción

- Los de objetos, delegan la creación de las instancias en otros objetos
- Estos patrones:
  - Encapsulan el conocimiento sobre las clases concretas que usa el sistema.
  - Ocultan cómo se crean y se asocian las instancias de estas clases.

# Patrones de creación

## Introducción

- Por tanto, lo único que conoce el sistema de sus objetos son sus interfaces.

# Patrones de creación

## Abstract Factory

- Propósito
  - Proporciona una interfaz para crear familias de objetos relacionados o que dependen entre sí, sin especificar sus clases concretas
- También conocido como
  - Fábrica Abstracta
  - Kit

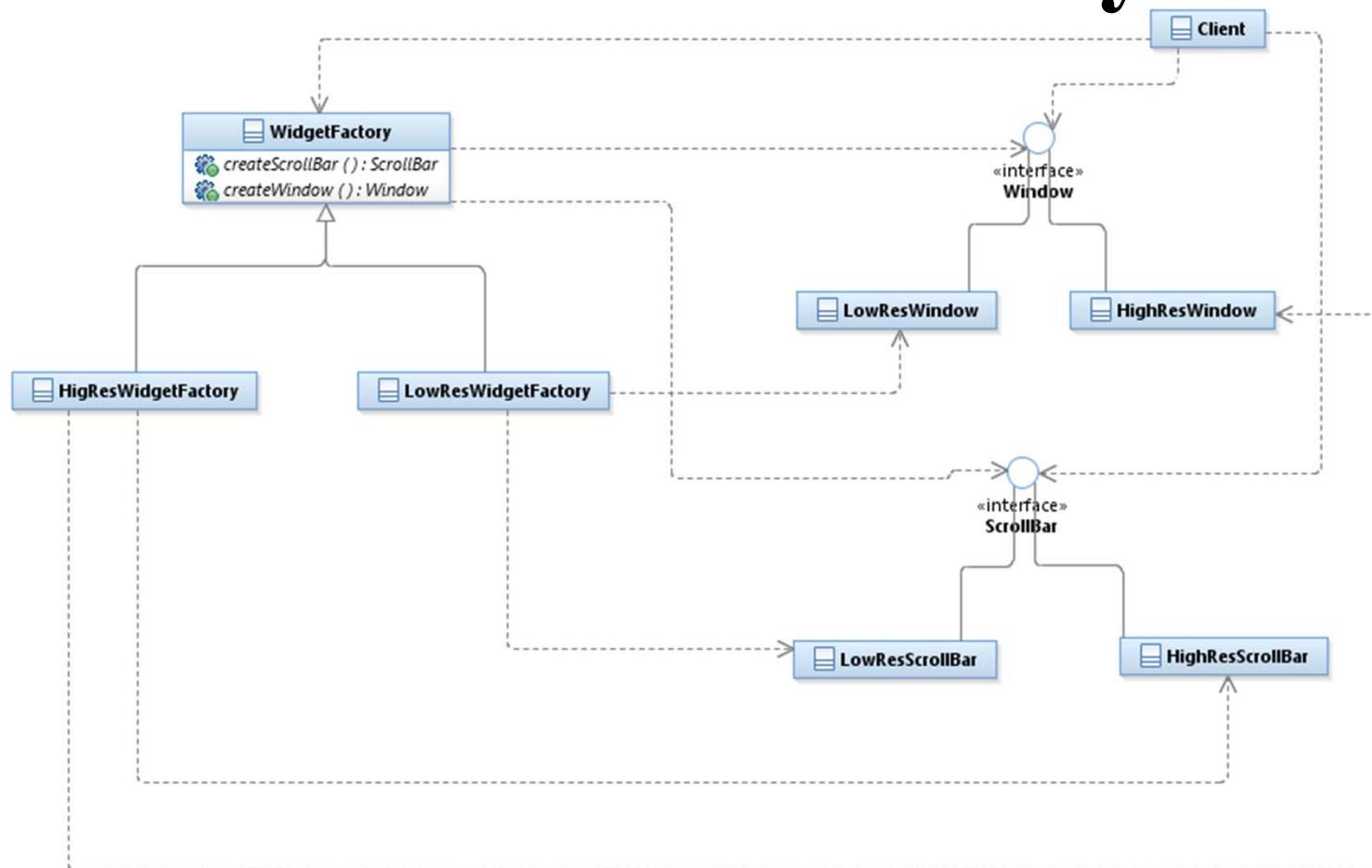
# Patrones de creación

## Abstract Factory

- Motivación
  - Supongamos que deseamos tener una interfaz de usuario independiente de los objetos concretos que la componen.
  - Si la aplicación crea instancias de clases o útiles específicos de la interfaz de usuario será difícil cambiar ésta más tarde.

# Patrones de creación

## Abstract Factory



# Patrones de creación

## Abstract Factory

- Debemos aplicar el patrón cuando:
  - Un sistema debe ser independiente de cómo se crean, componen y representan sus productos.
  - Un sistema debe ser configurado con una familia de productos de entre varias.
  - Una familia de objetos producto relacionados está diseñada para ser usada conjuntamente, y es necesario hacer cumplir esta restricción.

# Patrones de creación

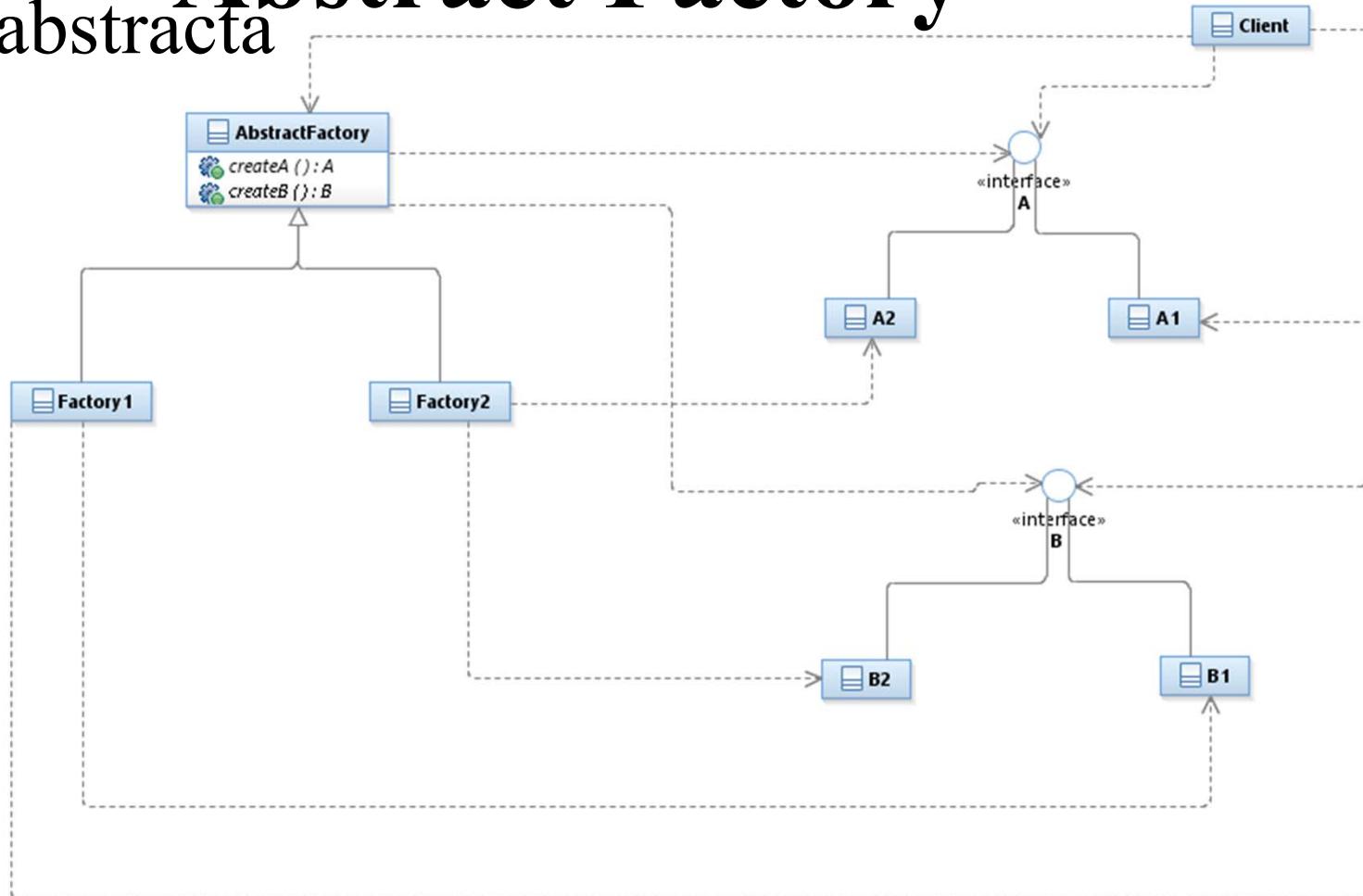
## Abstract Factory

- Quiere proporcionar una biblioteca de clases de productos, y sólo quiere revelar sus interfaces, no sus implementaciones.

# Patrones de creación

## Abstract Factory

- D. abstracta



# Patrones de creación

## Abstract Factory

- Consecuencias de Abstract Factory:
  - Ventajas:
    - Aísla las clases concretas de sus clientes.
    - Facilita el intercambio de familias de productos.
    - Promueve la consistencia entre productos.
  - Inconvenientes:
    - Es difícil dar cabida a nuevos tipos de productos, ya que hay que modificar FabricaAbstracta.

# Patrones de creación

## Abstract Factory

- Código de ejemplo
  - Supongamos que deseamos construir un juego de laberintos.
  - Deseamos que los laberintos que construyamos no dependan de los objetos (e.g. pared) concretos que lo componen.
  - Así podemos tener distintos niveles, para los mismos escenarios.

# Patrones de creación

## Abstract Factory

```
public interface FabricaDeLaberintos {  
    public Laberinto hacerLaberinto();  
    public Pared hacerPared();  
    public Habitacion hacerHabitacion();  
    public Puerta hacerPuerta();  
};
```

# Patrones de creación

## Abstract Factory

```
public class JuegoDelLaberinto {  
    .....  
    Laberinto crearLaberinto(FabricaDeLaberinto fabrica)  
    {  
        Laberinto l= fabrica.hacerLaberinto();  
        Habitacion h1= fabrica.hacerHabitacion();  
        Habitacion h2= fabrica.hacerHabitacion();  
        Puerta p= fabrica.hacerPuerta(h1, h2);  
  
        l.anadirHabitacion(h1);  
        l.anadirHabitacion(h2);  
    }  
}
```

# Patrones de creación

## Abstract Factory

```
h1.establecerLado(Norte, fabrica.hacerPared());
h1.establecerLado(Este, p);
h1.establecerLado(Sur, fabrica.hacerPared());
h1.establecerLado(Oeste, fabrica.hacerPared());

h2.establecerLado(Norte, fabrica.hacerPared());
h2.establecerLado(Este, fabrica.hacerPared());
h2.establecerLado(Sur, fabrica.hacerPared());
h2.establecerLado(Oeste, p)

return 1; }
```

# Patrones de creación

## Abstract Factory

```
class FabricaDeLaberintosEncantados implements  
FabricaDeLaberintos {  
    .....  
    Habitacion hacerHabitacion(int n)  
    { return new HabitacionEncantada(n); }  
  
    Puerta hacerPuerta(Habitacion h1, Habitacion h2)  
    { return new PuertaEncantada(h1, h2); }  
    .....  
};
```

# Patrones de creación

## Abstract Factory

```
class FabricaDeLaberintosExplosivos implements  
FabricaDeLaberintos {  
    .....  
    Habitacion hacerHabitacion(int n)  
    { return new HabitacionExplosiva(n); }  
  
    Puerta hacerPuerta(Habitacion h1, Habitacion  
    h2)  
    { return new PuertaExplosiva(h1, h2); }  
    .....  
};
```

# Patrones de creación

## Abstract Factory

```
JuegoDelLaberinto juego= new JuegoDelLaberinto( );  
FabricaDeLaberintosExplosivos fabrica = new  
FabricaDeLaberintosExplosivos();  
  
juego.crearLaberinto(fabrica);
```

# Patrones de creación

## Builder

- Propósito
  - Separa la construcción de un objeto complejo de su representación, de forma que el mismo proceso de construcción pueda crear diferentes representaciones del objeto.
- También conocido como
  - Constructor.

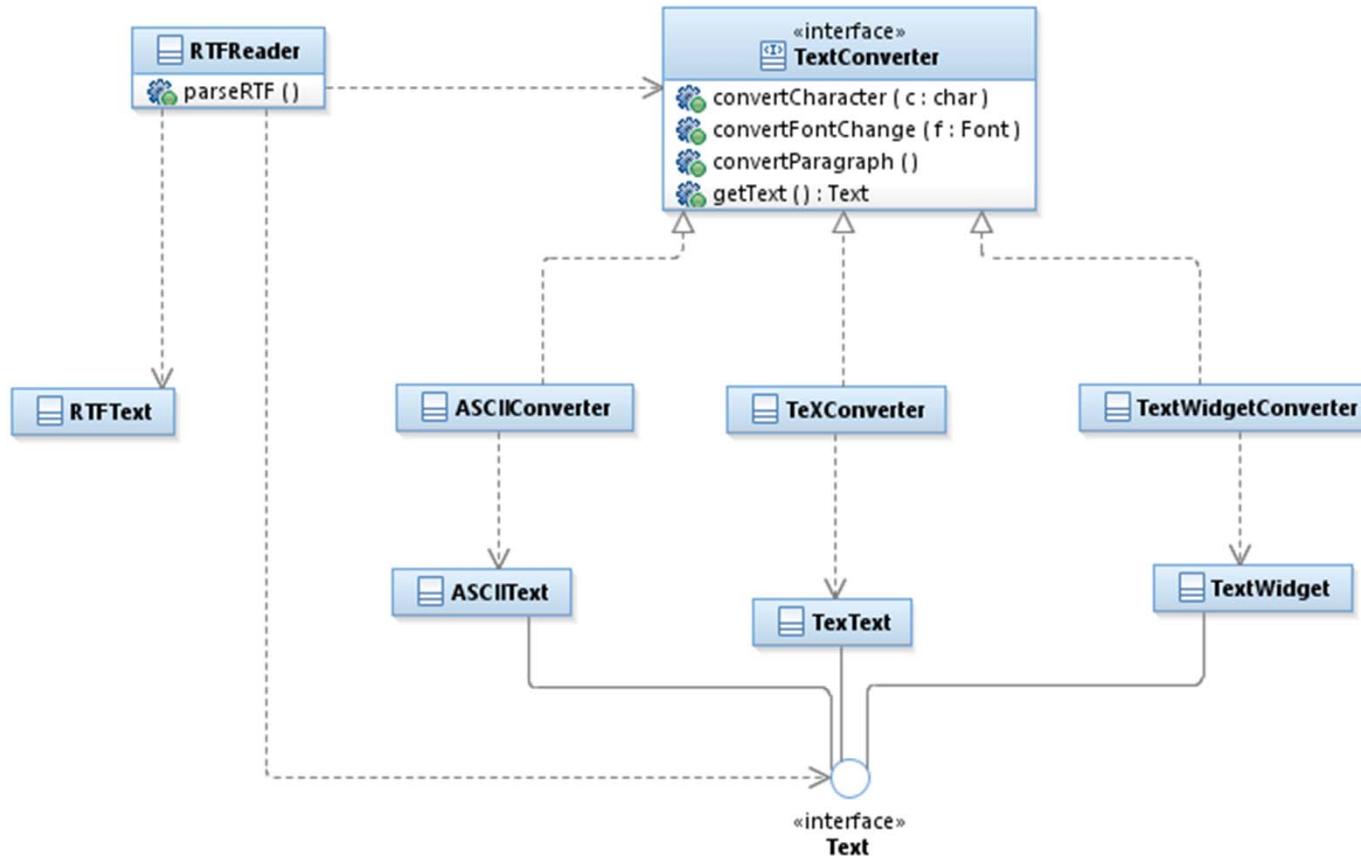
# Patrones de creación

## Builder

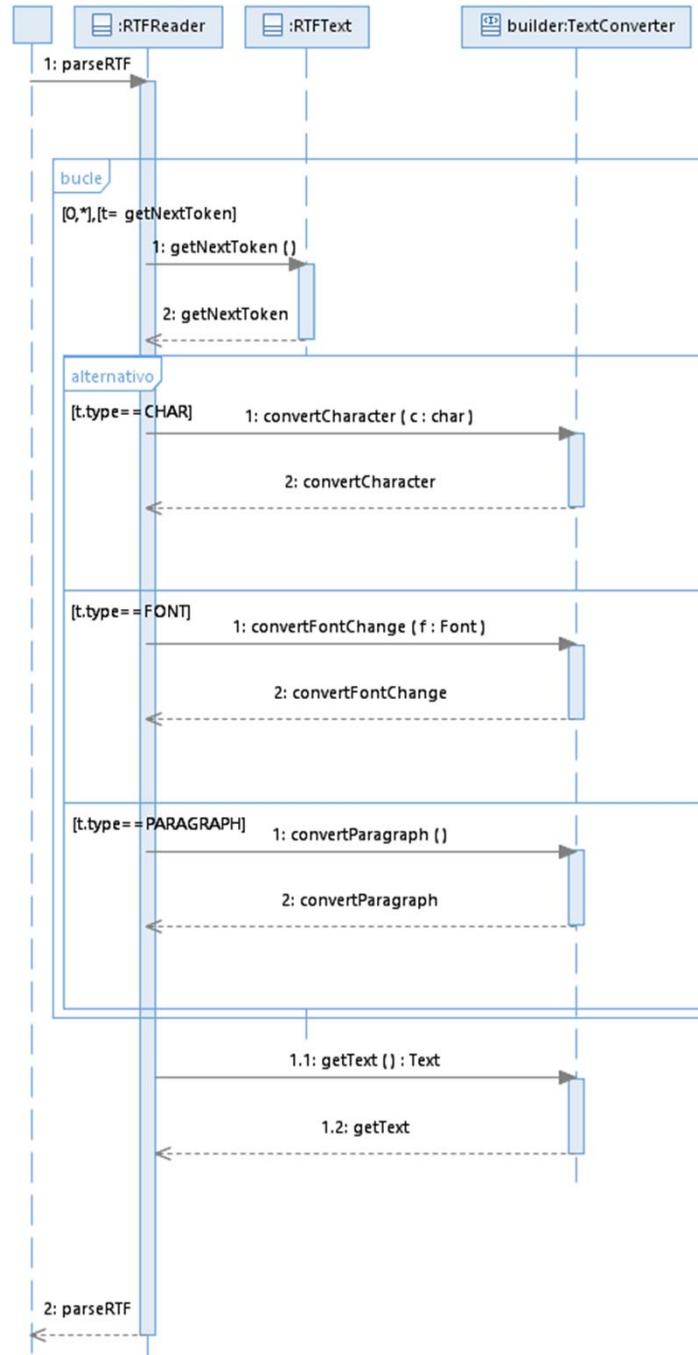
- Motivación
  - Supongamos que deseamos construir un editor RTF que pueda convertir un texto a distintos formatos.
  - El número de formatos no debería estar determinado a priori.

# Patrones de creación

## Builder



# Ejemplo comportamiento Builder



# Patrones de creación

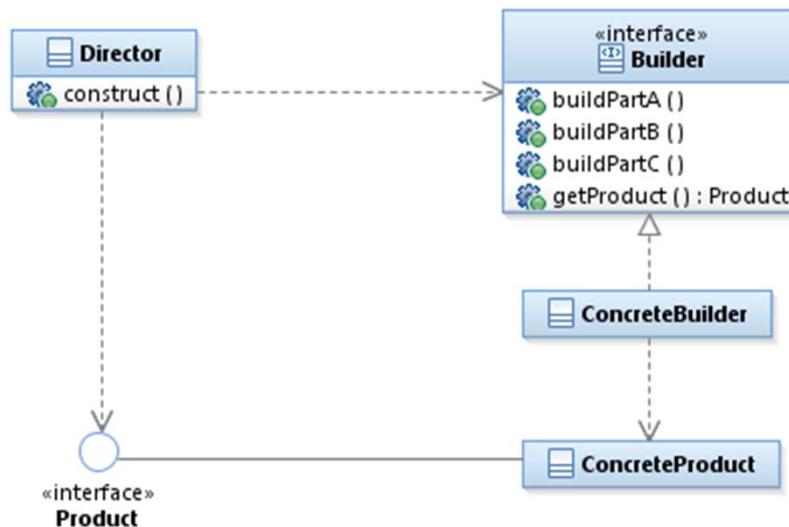
## Builder

- El patrón Builder se debe aplicar cuando:
  - El algoritmo para crear un objeto complejo debería ser independiente de las partes de que compone dicho objeto y de cómo se ensamblan.
  - El proceso de construcción debe permitir diferentes representaciones del objeto que está siendo construido.

# Patrones de creación

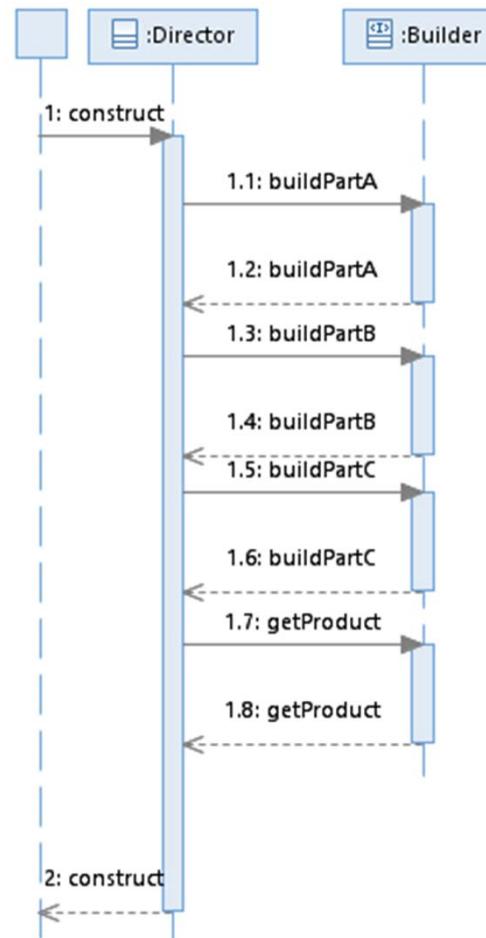
## Builder

- Descripción abstracta



Estructura del patrón Builder

# Patrones de creación



Colaboraciones en el patrón Builder

# Patrones de creación

## Builder

- Consecuencias
  - Ventajas
    - Permite variar la representación interna de un producto.
    - Aísla el código de construcción y representación.
    - Proporciona un control más fino sobre el proceso de construcción.

# Patrones de creación

## Builder

- Código de ejemplo
  - Reinterpretaremos el juego del laberinto desde la óptica del Builder.

# Patrones de creación

## Builder

```
public interface ConstructorLaberinto {  
    public void construirLaberinto();  
    public void construirHabitacion(int habitacion);  
    public void construirPuerta(int puerta);  
    public Laberinto obtenerLaberinto(); }
```

# Patrones de creación

## Builder

```
public class JuegoDelLaberinto {  
    .....  
    Laberinto crearLaberinto(ConstructorLaberinto  
        constructor)  
    {  
        constructor.construirLaberinto();  
        constructor.construirHabitacion(1);  
        constructor.construirHabitacion(2);  
        constructor.construirPuerta(1, 2);  
  
        return constructor.obtenerLaberinto(); }  
    .....};
```

# Patrones de creación

## Builder

```
public class ConstructorLaberintoEstandar
    implements ConstructorLaberinto {
    Laberinto laberintoActual;
    .....
    public ConstructorLaberintoEstandar () {
        laberintoActual = new LaberintoEstandar(); }
    public Laberinto obtenerLaberinto()
    {
        return laberintoActual;
    }
```

# Patrones de creación

## Builder

```
void construirHabitacion (int n)
{
    if (laberintoActual.habitacion(n) == null)
    {
        Habitacion h= new HabitacionEstandar(n);
        laberintoActual.anadirHabitacion(n);
        h.establecerLado(Norte, new ParedEstandar());
        h.establecerLado(Sur, new ParedEstandar());
        h.establecerLado(Este, new ParedEstandar());
        h.establecerLado(Oeste, new ParedEstandar());
    }
}
```

# Patrones de creación

## Builder

```
void construirPuerta(int h1, int h2)
{
    Habitacion h1= laberintoActual.habitacion(h1);
    Habitacion h2= laberintoActual.habitacion(h2);
    Puerta p= new PuertaEstandar(h1, h2);

    h1.establecerLado(Este, p);
    h2.establecerLado(Oeste, p);
}

.......
```

# Patrones de creación

## Builder

```
Laberinto laberinto;  
JuegoDelLaberinto juego = new JuegoDelLaberinto();  
ConstructorLaberinto constructor = new  
ConstructorLaberintoEstandar();  
  
juego.crearLaberinto(constructor);
```

# Patrones de creación

## Factory Method

- Propósito
  - Define una interfaz para crear un objeto, pero deja que sean las subclases quienes decidan qué clase instanciar.
  - Permite que una clase delegue en sus subclases la creación de objetos.
- También conocido como:
  - Método de fabricación.
  - Virtual Constructor (Constructor Virtual)

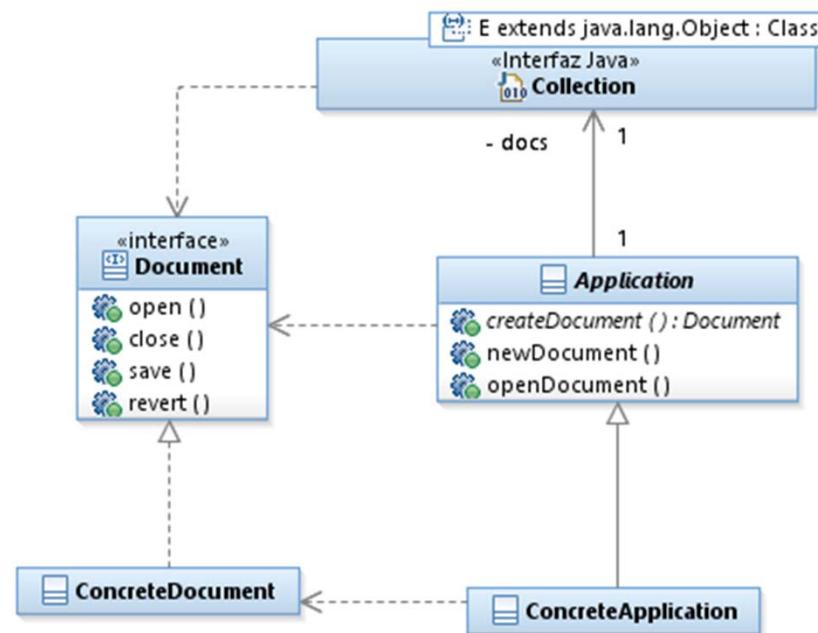
# Patrones de creación

## Factory Method

- Motivación
  - Los marcos usan clases abstractas/interfaces para definir y mantener relaciones entre objetos y también son muchas veces responsables de crear esos mismos objetos.
  - Por ejemplo, un marco de aplicaciones puede presentar distintos tipos de documentos al usuario.
  - Una aplicación puede saber *cuándo* crear un documento, pero no el *tipo* del mismo.

# Patrones de creación

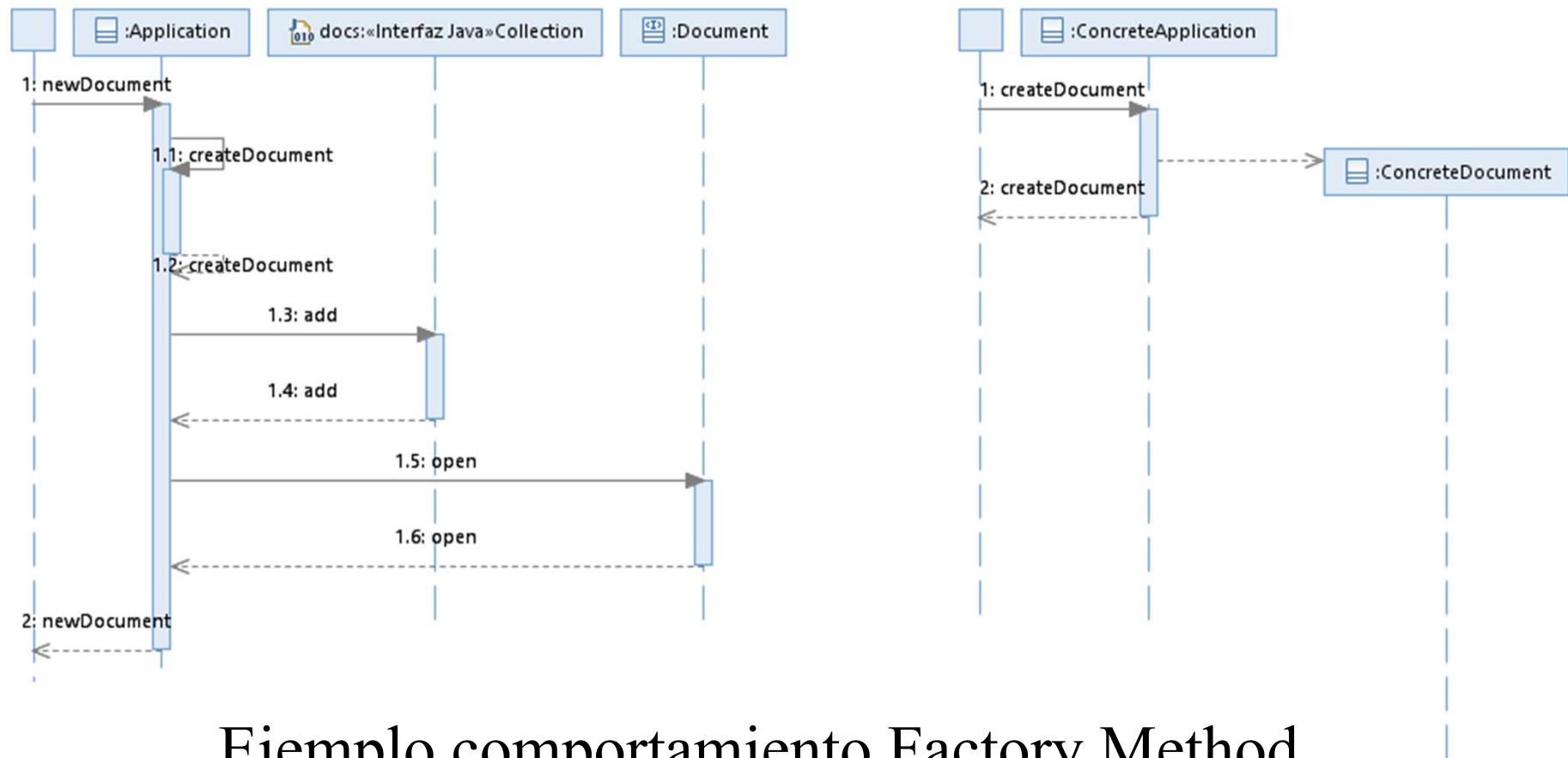
## Factory Method



Ejemplo estructura Factory Method

# Patrones de creación

## Factory Method



Ejemplo comportamiento Factory Method

# Patrones de creación

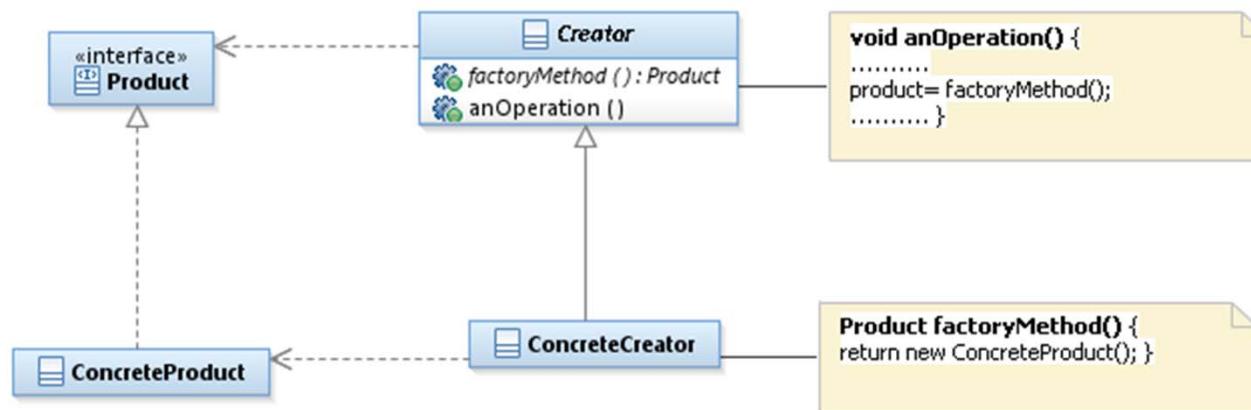
## Factory Method

- El patrón FM debe aplicarse cuando:
  - Una clase no puede prever la clase de objetos que debe crear.
  - Una clase quiere que sean sus subclases quienes especifiquen los objetos que ésta crea.
  - Las clases delegan la responsabilidad en una de entre varias clases auxiliares, y queremos localizar en una subclase dicha delegación.

# Patrones de creación

## Factory Method

- Descripción abstracta



Estructura del patrón Factory Method

# Patrones de creación

## Factory Method

- Consecuencias
  - Ventajas:
    - Elimina la necesidad de ligar nuestro código con clases específicas.
  - Inconvenientes:
    - Los clientes pueden tener que heredar de la clase Creador simplemente para crear un determinado objeto ProductoConcreto.

# Patrones de creación

## Factory Method

- Código de ejemplo
  - Reinterpretaremos el juego del laberinto desde la óptica del patrón Factory Method

# Patrones de creación

## Factory Method

```
public abstract class JuegoDelLaberinto {  
    ..... // atributos del laberinto  
    public abstract Laberinto fabricarLaberinto();  
    public abstract Habitacion fabricarHabitacion();  
    public abstract Pared fabricarPared();  
    public abstract Puerta fabricarPuerta(Habitacion  
h1, Habitacion h2);  
    ..... // otros métodos del laberinto
```

# Patrones de creación

## Factory Method

```
public Laberinto crearLaberinto() {  
    Laberinto l= fabricarLaberinto();  
    Habitacion h1= fabricarHabitacion();  
    Habitacion h2= fabricarHabitacion();  
    Puerta p= fabricarPuerta(h1, h2);  
  
    l.anadirHabitacion(h1);  
    l.anadirHabitacion(h2);
```

# Patrones de creación

## Factory Method

```
h1.establecerLado(Norte, fabricarPared());
h1.establecerLado(Este, p);
h1.establecerLado(Sur, fabricarPared());
h1.establecerLado(Oeste, fabricarPared());

h2.establecerLado(Norte, fabricarPared());
h2.establecerLado(Este, fabricarPared());
h2.establecerLado(Sur, fabricarPared());
h2.establecerLado(Oeste, p);

return l; }
.....};
```

# Patrones de creación

## Factory Method

```
class JuegoDelLaberintoExplosivo extends  
JuegoDelLaberinto {  
    .....  
    public Pared fabricarPared()  
    { return new ParedExplosiva( ); }  
  
    public Habitacion fabricarHabitacion(int n)  
    { return new HabitacionExplosiva(n); }  
    .....  
};
```

# Patrones de creación

## Prototype

- Propósito
  - Especifica los tipos de objetos a crear por medio de una instancia prototípica, y crea nuevos objetos copiando dicho prototipo.
- También conocido como
  - Prototipo

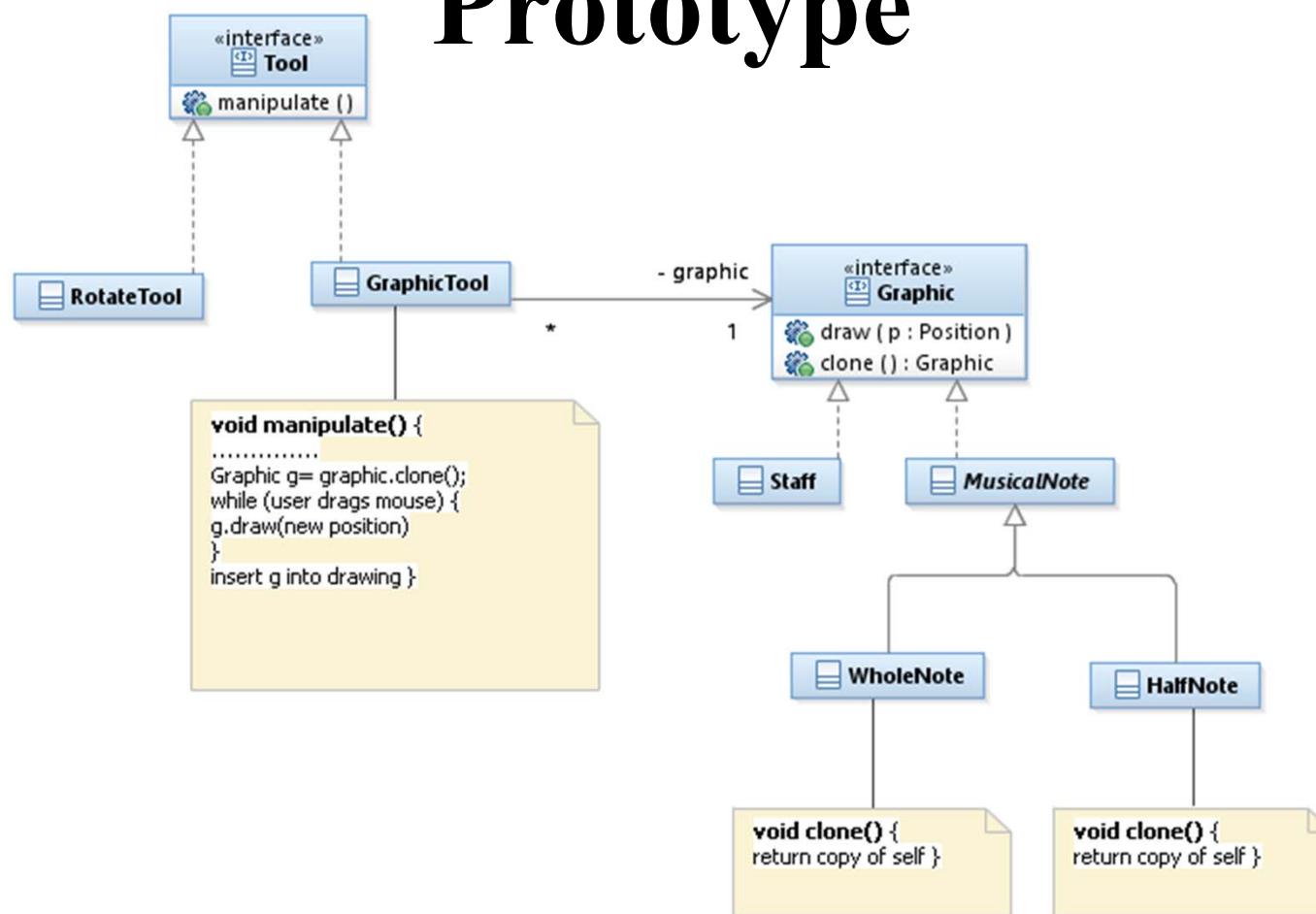
# Patrones de creación

## Prototype

- Motivación
  - Supongamos que deseamos reutilizar un kit de herramientas gráficas.
  - La reutilización demanda que el uso de los objetos sea independiente de su proceso de creación.

# Patrones de creación

## Prototype



# Patrones de creación

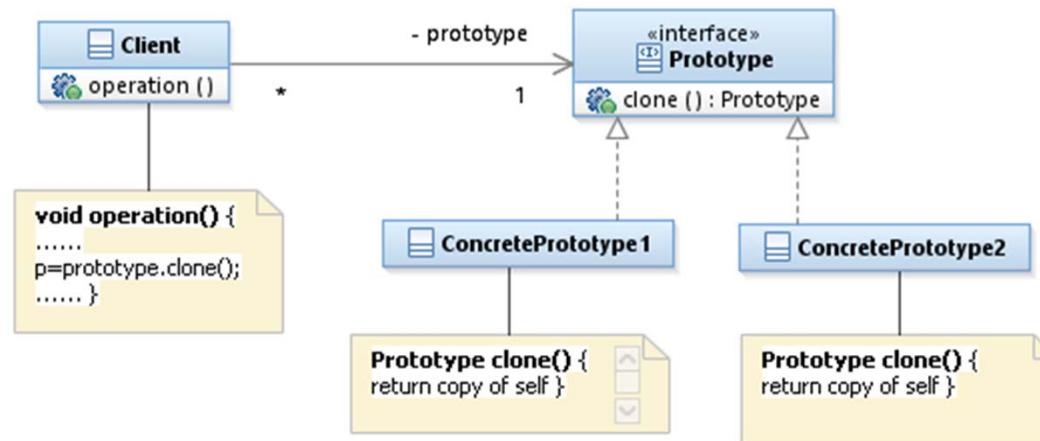
## Prototype

- El patrón Prototype debe aplicarse cuando:
  - Las clases a instanciar sean especificadas en tiempo de ejecución.
  - Para evitar construir una jerarquía de clases fábricas paralela a la jerarquía de clases de los productos.
  - Cuando las instancias de una clase puedan tener uno de entre sólo unos pocos estados diferentes.

# Patrones de creación

## Prototype

- Descripción abstracta



Estructura y comportamiento del patrón Prototype

# Patrones de creación

## Prototype

- Consecuencias
  - Ventajas:
    - Similares a las de Abstract Factory y Builder: oculta al cliente las clases producto concretas, reduciendo así el número de nombres que conocen los clientes y permitiendo que las usen sin cambios. Además permite:
    - Añadir y eliminar productos en tiempo de ejecución.
    - Especificar objetos nuevos modificando valores.

# Patrones de creación

## Prototype

- Especificar nuevos objetos variando la estructura.
- Reduce la herencia.
- Permite configurar dinámicamente una aplicación con clases, utilizando un gestor de prototipos.
- Inconvenientes
  - Cada subclase de prototipo debe implementar la operación clonar, lo cual puede ser difícil.

# Patrones de creación

## Prototype

- Código de ejemplo

```
public interface FabricaDeLaberintos {  
    public Laberinto hacerLaberinto();  
    public Pared hacerPared();  
    public Habitacion hacerHabitacion();  
    public Puerta hacerPuerta();  
};
```

# Patrones de creación

## Prototype

```
public class FabricaDePrototiposLaberinto  
    implements FabricaDeLaberintos {  
  
    Laberinto prototipoLaberinto;  
    Habitacion prototipoHabitación;  
    Pared prototipoPared;  
    Puerta prototipoPuerta;
```

# Patrones de creación

## Prototype

```
FabricaDePrototiposLaberinto(Laberinto l, Pared  
p, Habitacion h, Puerta pu)  
{  
    prototipoLaberinto= l;  
    prototipoPared= p;  
    prototipoHabitacion= h;  
    prototipoPuerta= pu;  
  
}
```

# Patrones de creación

## Prototype

```
Pared hacerPared()
{ return prototipoPared.clonar(); }

Puerta hacerPuerta(Habitacion h1, Habitacion h2)
{
    Puerta puerta= prototipoPuerta.clonar();
    puerta.inicializar(h1, h2);
    return puerta;
}
```

# Patrones de creación

## Prototype

```
JuegoDelLaberinto juego;
```

```
FabricaDeLaberintos fabricaDeLaberintosSimples=
    new FabricaDePrototiposLaberinto(new
        LaberintoSimple(), new ParedSimple(), new
        HabitacionSimple(), new PuertaSimple());
```

```
Laberinto l=
    juego.crearLaberinto(fabricaDeLaberintosSimples
);
```

# Patrones de creación

## Prototype

```
public class PuertaSimple implements Puerta {  
    Habitacion h1;  
    Habitacion h2;  
  
    public PuertaSimple(Puerta p)  
    {    h1= p.h1;    h2= p.h2; }  
  
    public void inicializar (Habitacion h1P, Habitacion  
h2P)  
    { h1= h1P; h2= h2P; }  
  
    public Puerta clonar()  
    { return new PuertaSimple(this); }
```

# Patrones de creación

## Prototype

- Cuestiones

- ¿Por qué no hemos definido un interfaz?

```
public interface FabricaDePrototiposLaberinto  
extends FabricaDeLaberintos {  
    public fijarPrototipos(Laberinto l, Pared p,  
    Habitacion h, Puerta p);  
}
```

- ¿Por qué no hemos utilizado la función clone( ) de Object?

# Patrones de creación

## Prototype

- ¿Por qué el constructor de `PuertaSimple` recibe una `Puerta` como parámetro, si al final, la factoría tiene que inicializarla? Es decir, ¿por qué no tenemos lo siguiente?

```
public class PuertaSimple implements Puerta {  
  
    public Puerta clonar()  
    { return new PuertaSimple(); }  
}
```

# Patrones de creación

## Singleton

- Propósito
  - Garantiza que una clase solo tenga una instancia, y proporciona un punto de acceso global a ella.
- Tambien conocido como
  - Único.

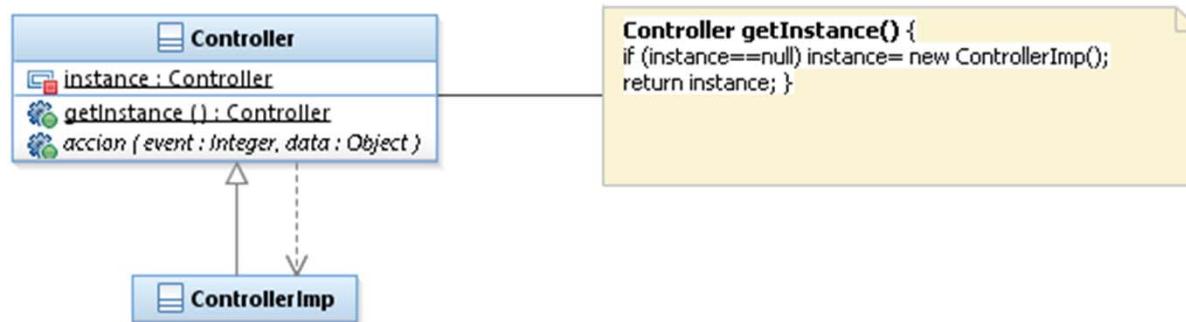
# Patrones de creación

## Singleton

- Motivación
  - Es importante que algunas clases sólo tengan una instancia, e.g. cola de impresión, biblioteca, etc.
  - ¿Cómo garantizamos que una clase tenga una única instancia fácilmente accesible?
  - La propia clase es responsable de su única instancia.

# Patrones de creación

## Singleton



Ejemplo del patrón Singleton

# Patrones de creación

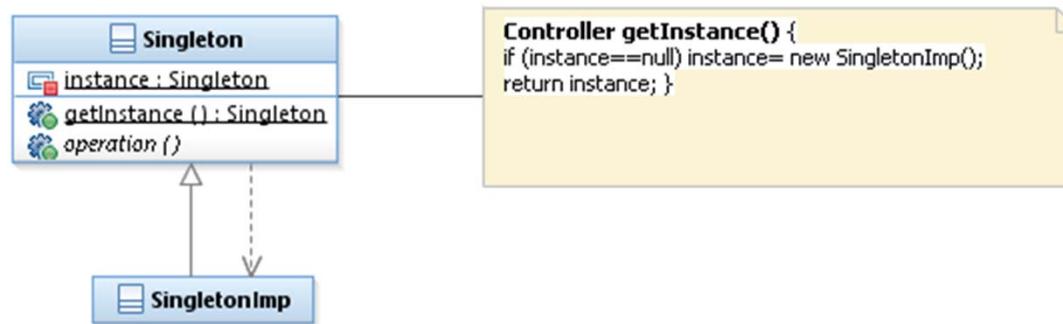
## Singleton

- El patrón Singleton debe aplicarse cuando:
  - Debe haber exactamente una instancia de clase, y ésta debe ser accesible a los clientes desde un punto de acceso conocido.
  - La única instancia debería ser extensible mediante herencia, y los clientes deberían ser capaces de usar una instancia extendida sin modificar su código.

# Patrones de creación

## Singleton

- Descripción abstracta



Estructura y comportamiento del patrón Singleton

# Patrones de creación

## Singleton

- Consecuencias
  - Ventajas
    - Acceso controlado a la única instancia.
    - Espacio de nombres reducido.
    - Permite el refinamiento de operaciones y la representación.
    - Permite un número variable de instancias.
    - Más flexibles que las operaciones de clase estáticas.

# Patrones de creación

## Singleton

- Código de ejemplo

```
public abstract class FabricaDeLaberintos {  
    private static FabricaDeLaberintos fabrica;  
  
    public static FabricaDeLaberintos obtenerInstancia()  
    {  
        if (fabrica == null)  
            fabrica= new FabricaDeLaberintosSimples();  
        return fabrica;  
    }  
}
```

# Patrones de creación

## Singleton

```
public abstract Laberinto hacerLaberinto();  
public abstract Pared hacerPared();  
public abstract Habitacion hacerHabitacion();  
public abstract Puerta hacerPuerta();  
  
}
```

# Patrones de creación

## Singleton

```
public class FabricaDeLaberintosSimples extends  
FabricaDeLaberintos {  
  
    Habitacion hacerHabitacion(int n)  
    { return new HabitacionSimple(n); }  
  
    Puerta hacerPuerta(Habitacion h1, Habitacion h2)  
    { return new PuertaSimple(h1, h2); }  
    . . . . .  
}
```

# Patrones de creación

## Singleton

- Nota:
  - En el ejemplo anterior, el singleton siempre crea la misma clase de factoría
  - Por lo tanto, si los clientes quieren obtener otra implementación de la factoría, debería cambiarse el código de ésta a nivel paquete
  - Hay opciones más razonables

# Patrones de creación

## Singleton

- Su método de generación lee de un archivo la clase concreta que implementa a dicha factoría y que debe generar, la carga dinámicamente y se la devuelve al cliente

# Patrones de creación

## Singleton

```
public abstract class FabricaDeLaberintos {  
    private static FabricaDeLaberintos fabrica;  
  
    public static FabricaDeLaberintos obtenerInstancia()  
    {  
        String claseFactoria= null;  
        try { BufferedReader in= new BufferedReader(new  
                FileReader( "configFactoria.txt" ));  
  
            claseFactoria= in.readLine();  
            in.close();  
        } catch ( java.io.IOException e )  
        { System.out.println( "Problema de E/S" ); }  
    }  
}
```

# Patrones de creación

## Singleton

```
try {  
    if (fabrica == null) fabrica=  
        (FabricaDeLaberintos)  
    Class.forName(claseFactoria).newInstance();  
} catch (Exception e) {  
    System.out.println("Implementación de  
FabricaDeLaberintos no encontrada"); }  
  
return fabrica;  
}
```

# Patrones de creación

## Singleton

```
public abstract Laberinto hacerLaberinto();  
public abstract Pared hacerPared();  
public abstract Habitacion hacerHabitacion();  
public abstract Puerta hacerPuerta();  
  
}
```

# Patrones estructurales

## Introducción

- Los patrones estructurales se preocupan de cómo se combinan las clases y los objetos para formas estructuras más grandes
- Los de clases hacen uso de la herencia para componer interfaces o implementaciones
- Los de objetos describen formas de componer objetos para obtener nueva funcionalidad

# Patrones estructurales

## Introducción

- En los patrones de objetos, la flexibilidad añadida de la composición de objetos viene dada por la capacidad de cambiar la composición en tiempo de ejecución, lo que es imposible con la composición de clases, estática

# Patrones estructurales

## Adapter

- Propósito
  - Convierte la interfaz de una clase en otra interfaz que es la que esperan los clientes.
  - Permite que cooperen clases que de otra forma no podrían tener interfaces compatibles.
- También conocido como
  - Adaptador.
  - Wrapper (Envoltorio).

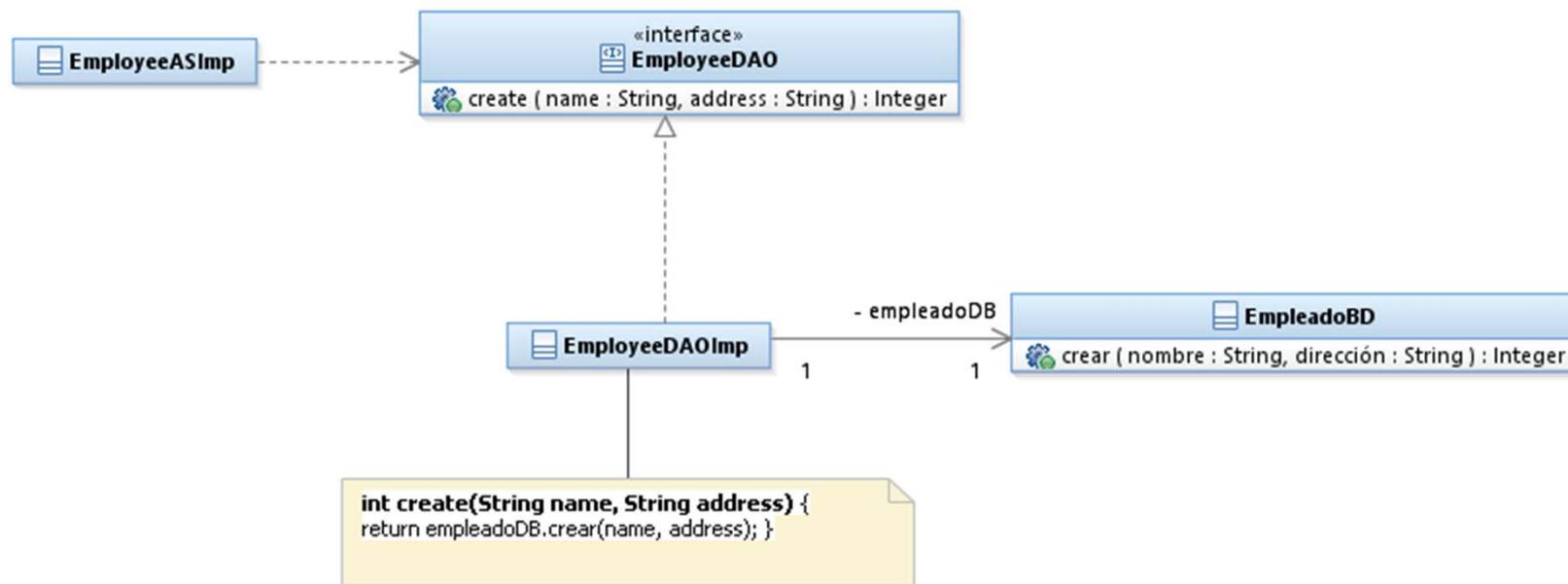
# Patrones estructurales

## Adapter

- Motivación
  - A veces una clase de utilidad que ha sido diseñada para reutilizarse, no puede hacerlo porque su interfaz no coincide con la interfaz específica del dominio que requiere la aplicación.
  - Por eso es necesario adaptarla para que pueda ser utilizada

# Patrones estructurales

## Adapter



Ejemplo patrón Adapter

# Patrones estructurales

## Adapter

- El patrón Adapter debe aplicarse cuando
  - Se quiere usar una clase existente y su interfaz no concuerda con la que se necesita.
  - Se quiere crear una clase reutilizable que coopere con clases no relacionadas o que no han sido previstas, es decir, clases que no tiene porque tener interfaces compatibles.

# Patrones estructurales

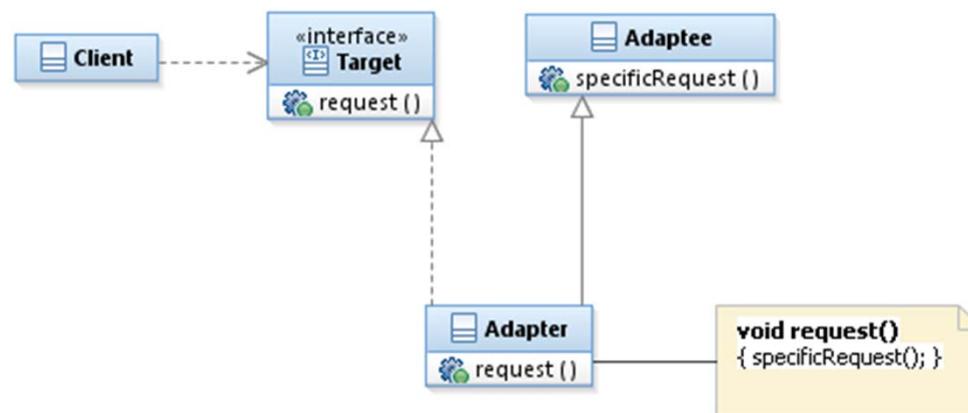
## Adapter

- En el caso de un adaptador de objetos, es necesario usar varias subclases existentes, pero no resulta práctico adaptar su interfaz heredando de cada una de ellas. Un adaptador de objetos puede adaptar la interfaz de su clase padre.

# Patrones estructurales

## Adapter

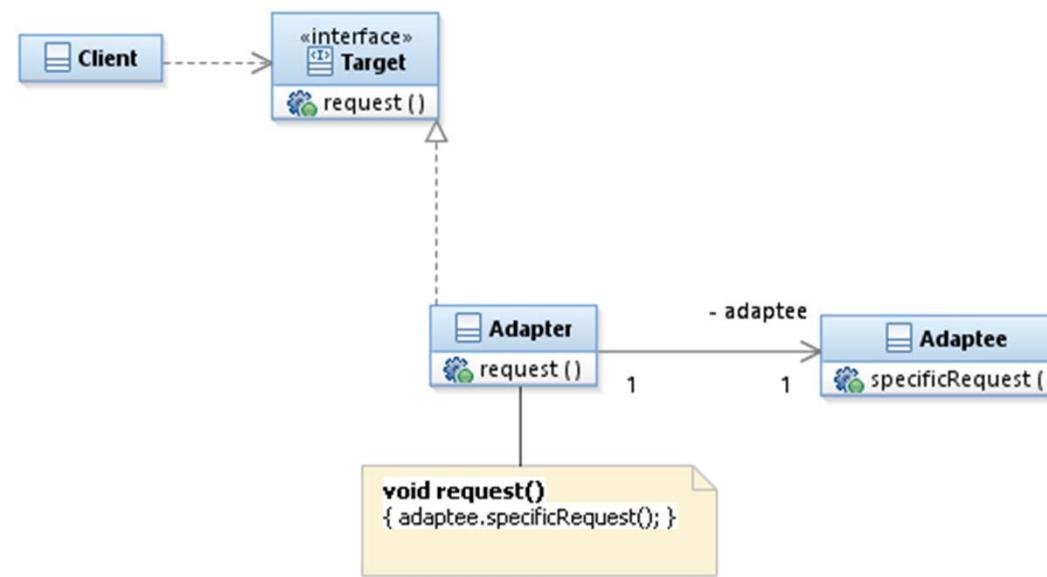
- Descripción abstracta



Estructura y comportamiento de un Adaptador de Clases

# Patrones estructurales

## Adapter



Estructura y comportamiento de un Adaptador de objetos

# Patrones estructurales

## Adapter

- Consecuencias adaptador clases
  - Ventajas
    - Permite que el adaptador redefina parte del comportamiento del adaptable, al ser subclase suya.
    - Introduce un solo objeto, y no se necesita ningún puntero de indirección adicional para obtener el objeto adaptado.
  - Inconvenientes
    - La adaptación es de una clase, pero no de sus subclases.

# Patrones estructurales

## Adapter

- Consecuencias adaptador objetos
  - Ventajas:
    - Permite que un adaptador funcione con muchos adaptables, es decir, con sus subclases. Además, el adaptador puede añadir funcionalidad a todos los adaptables.
  - Inconvenientes:
    - Obliga a vincular al adaptador a las clases que pudieran aparecer del adaptable.

# Patrones estructurales

## Adapter

- Código de ejemplo (objetos)

```
public interface IED {  
    public int insertar(Comparable objetoP);  
    public int eliminar(Object idObjeto);  
    public Comparable obtenerPorId(Object idObjeto);  
    public Comparable obtenerPorPos(int pos);  
    public int obtenerNumEletos();  
};
```

# Patrones estructurales

## Adapter

```
public class DynamicList {  
    public int insert(Comparable objectP) {...};  
    public int delete(Object idObject) {...};  
    public Comparable getById(Object idObject) {...};  
    public Comparable getByPos(int pos) {...};  
    public int getNumElements() {...};  
};
```

# Patrones estructurales

## Adapter

```
public class ListaDinamica implements IED {  
    DynamicList lista;  
  
    ListaDinamica ()  
    { lista= new DynamicList(); }  
  
    public int insertar(Object o)  
    { return lista.insert(o); }  
    .....  
};
```

# Patrones estructurales

## Adapter

- Código de ejemplo (clases)

```
public class ListaDinamica extends DynamicList
    implements IED {

    public int insertar(Object o)
    { return insert(o); }

    .....
}
```

# Patrones estructurales

## Bridge

- Propósito
  - Desacopla una abstracción de su implementación, de modo que ambas puedan variar de forma independiente.
- También conocido como
  - Puente
  - Handle/Body (Manejador/Cuerpo)

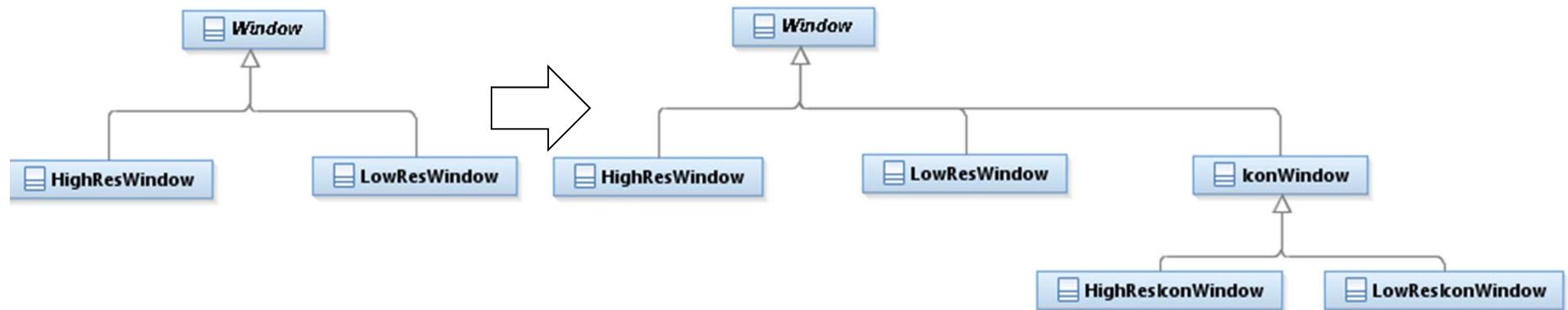
# Patrones estructurales

## Bridge

- Motivación
  - Cuando una abstracción (clase abstracta) puede tener varias implementaciones posibles, la forma más habitual de darle cabida es mediante la herencia.
  - La herencia liga las implementaciones a las abstracciones.

# Patrones estructurales

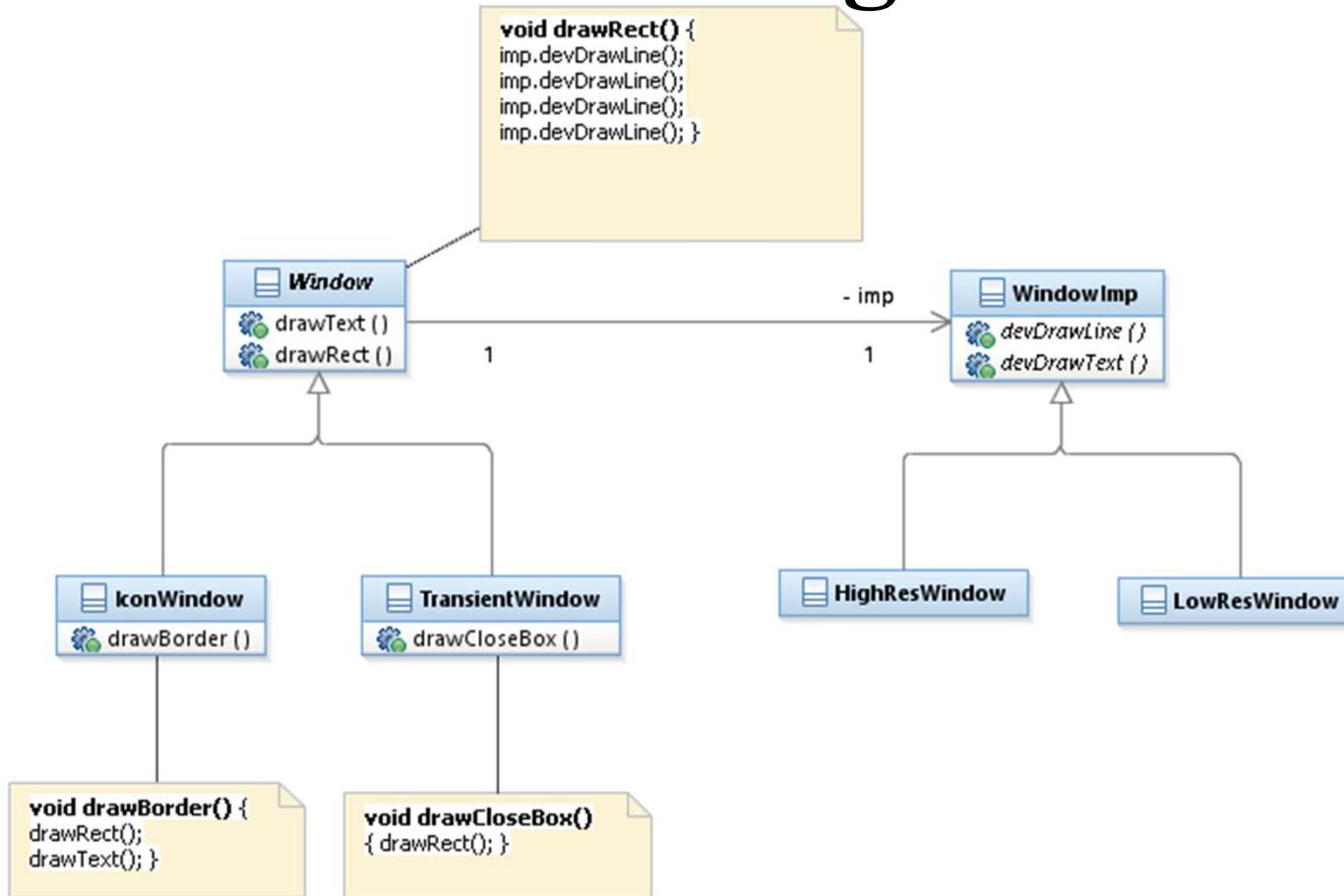
## Bridge



Duplicación de jerarquía de implementaciones al  
ligarla a la jerarquía de abstracciones

# Patrones estructurales

## Bridge



# Patrones estructurales

## Bridge

- El patrón Bridge debe aplicarse cuando
  - Se quiera evitar un enlace permanente entre una abstracción y su implementación (e.g. cuando deba seleccionarse o cambiarse en tiempo de ejecución).
  - Tanto las abstracciones como sus implementaciones deban ser extensibles mediante subclases y de manera independiente.

# Patrones estructurales

## Bridge

- Los cambios en la implementación de una abstracción no deberían tener impacto en los clientes, es decir, no es necesario recompilar su código (alternativa a las factorías abstractas). Nótese que en vez de hacer news de objetos que especializan a la abstracción, se hacen news de las abstracciones, y éstas reciben a sus implementadores por parámetro.
- Se pretenda ocultar completamente a los clientes de las abstracciones la implementación (atributos) de una abstracción (C++).

# Patrones estructurales

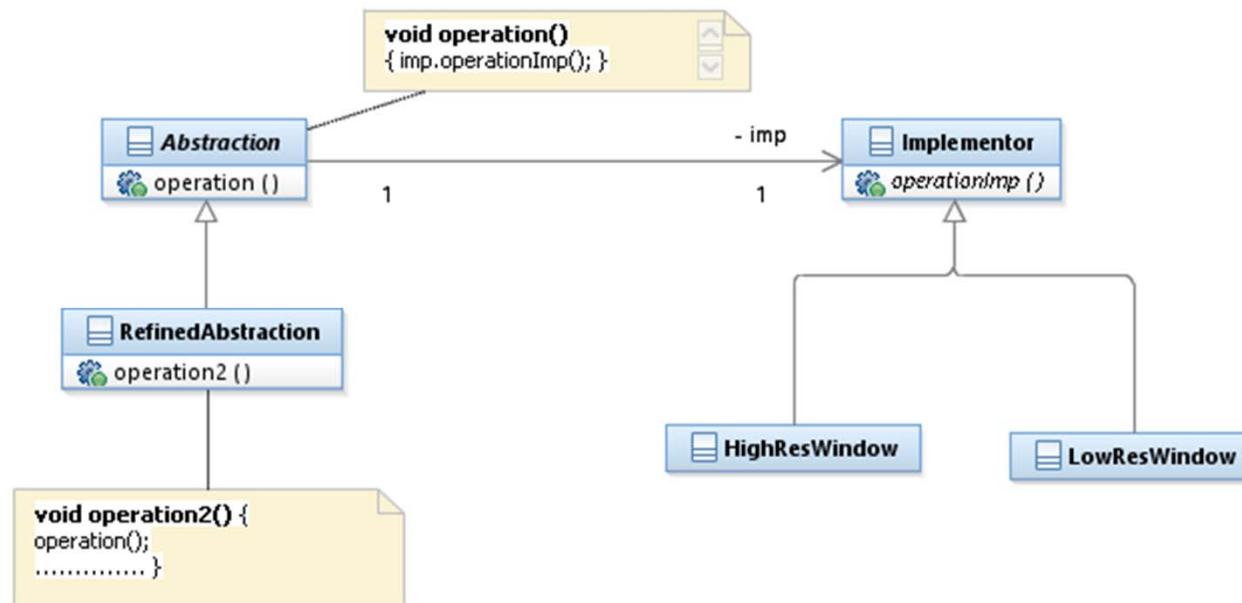
## Bridge

- Se quiera evitar una proliferación de clases de implementación que duplica la estructura de herencia de las abstracciones.
- Se quiera compartir una implementación entre varios objetos y este hecho deba permanecer oculto al cliente (también se podría resolver con factorías que producen Singletons).

# Patrones estructurales

## Bridge

- Descripción abstracta



Estructura y comportamiento del patrón Bridge

# Patrones estructurales

## Bridge

- Consecuencias
  - Ventajas
    - Desacopla la interfaz de la implementación, permitiendo cambios en ejecución y evitando dependencias de compilación.
    - Mejora la extensibilidad al independizar las jerarquías de abstracciones e implementaciones.
    - Oculta detalles de implementación a los clientes, como el compartir objetos de implementación.

# Patrones estructurales

## Bridge

- Inconvenientes
  - Fuerza a que la implementación proporcione todas las operaciones que las posibles subclases de la abstracción pudieran necesitar para la implementación de sus operaciones.
  - Así, nuevas subclases de la abstracción, podrían necesitar nuevas implementaciones para soportar dichas operaciones.

# Patrones estructurales

## Bridge

- Código de ejemplo

```
public class Ventana {  
  
    //peticiones manejadas por la ventana  
    public void DibujarContenido() {...};  
    public void Abrir() {...};  
    public void Cerrar() {...};  
    public void Minimizar() {...};  
    public void Maximizar() {...};
```

# Patrones estructurales

## Bridge

```
//peticiones reenviadas a su implementación
public void EstablecerOrigen(Punto en)
    {imp.establecerOrigen(en); }
public void EstablecerArea (Punto area) { ... };
public void TraerAlFrente() { ... };
public void EnviarAlFondo() { ... };

public void DibujarLinea(Punto p1, Punto p2) { ... };
public void DibujarRect(Punto p1, Punto p2) { ... };
public void DibujarPoligono(Punto []pts, int n) { ... };
public void DibujarTexto(String s, Punto p) { ... };
```

# Patrones estructurales

## Bridge

```
protected VentanaImp obtenerVentanaImp() { ... }  
protected Vista obtenerVista() { ... };
```

```
VentanaImp imp;  
Vista contenido; //el contenido de la ventana  
};
```

# Patrones estructurales

## Bridge

```
public interface VentanaImp {  
    public void ImpSuperior();  
    public void ImpInferior();  
    public void ImpEstablecerArea(Punto p);  
    public void ImpEstablecerOrigen(Punto p);  
  
    public void DispositivoRect(Coord c1, Coord c2,  
        Coord c3, Coord c4);  
    public void DispositivoTexto(String s, Coord  
        c1, Coord c2);  
    public void DispositivoMapaDeBits(String s,  
        Coord c1, Coord c2);  
    //resto de funciones para dibujar en ventanas  
};
```

# Patrones estructurales

## Bridge

```
class VentanaAplicacion extends Ventana {  
    .....  
    public void dibujarContenido()  
    {  
        obtenerVista().dibujarEn(this);  
    }  
};
```

# Patrones estructurales

## Bridge

```
class VentanaIcono extends Ventana {  
    .....  
    String nombreMapaDeBits;  
    public void dibujarContenido()  
    {  
        VentanaImp imp= obtenerVentanaImp();  
        if (imp != null) {  
  
            imp.dispositivoMapaDeBits(nombreMapaDeBits,  
                                         0, 0);  
        }  
    }  
};
```

# Patrones estructurales

## Bridge

```
public class VentanaSencilla implements  
VentanaImp{  
    .....  
};  
  
public class VentanaCalidad implements VentanaImp  
{  
    .....  
};
```

# Patrones estructurales

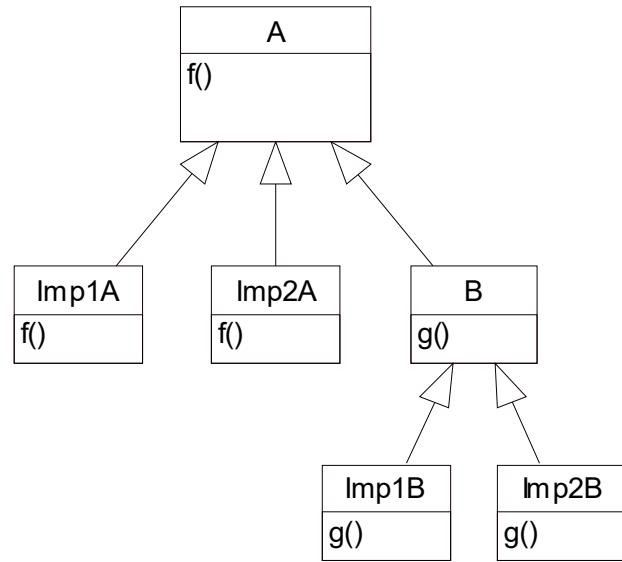
## Bridge

- Nótese que sin la presencia de las clases que refinan a la abstracción, el patrón bridge se limita a delegar en un objeto visto a través de un interfaz, en vez de delegar en subclases. Es algo así como delegar en subclases para crear (factory method) o delegar en objetos (abstract factory).

# Patrones estructurales

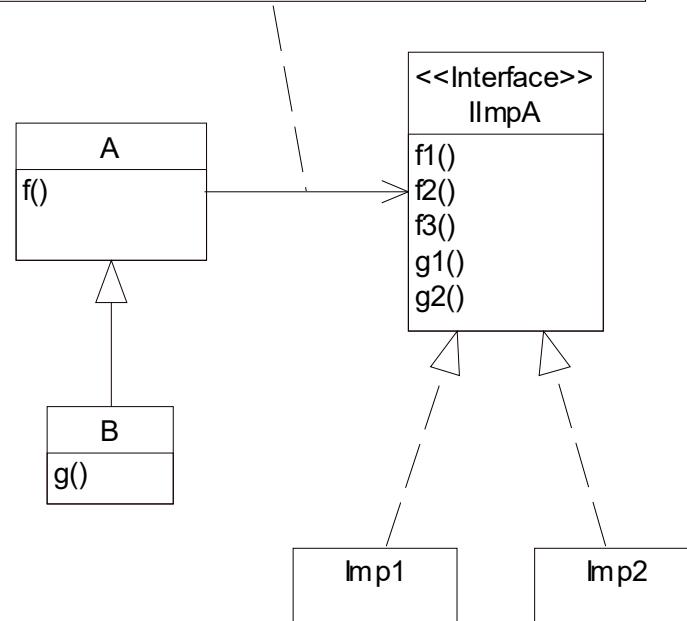
## Bridge

f() utiliza a f1(), f2() y f3() para su implementación  
g() utiliza a g1() y g2() para su implementación



(i)

Bridge pasa de (i) a (ii)

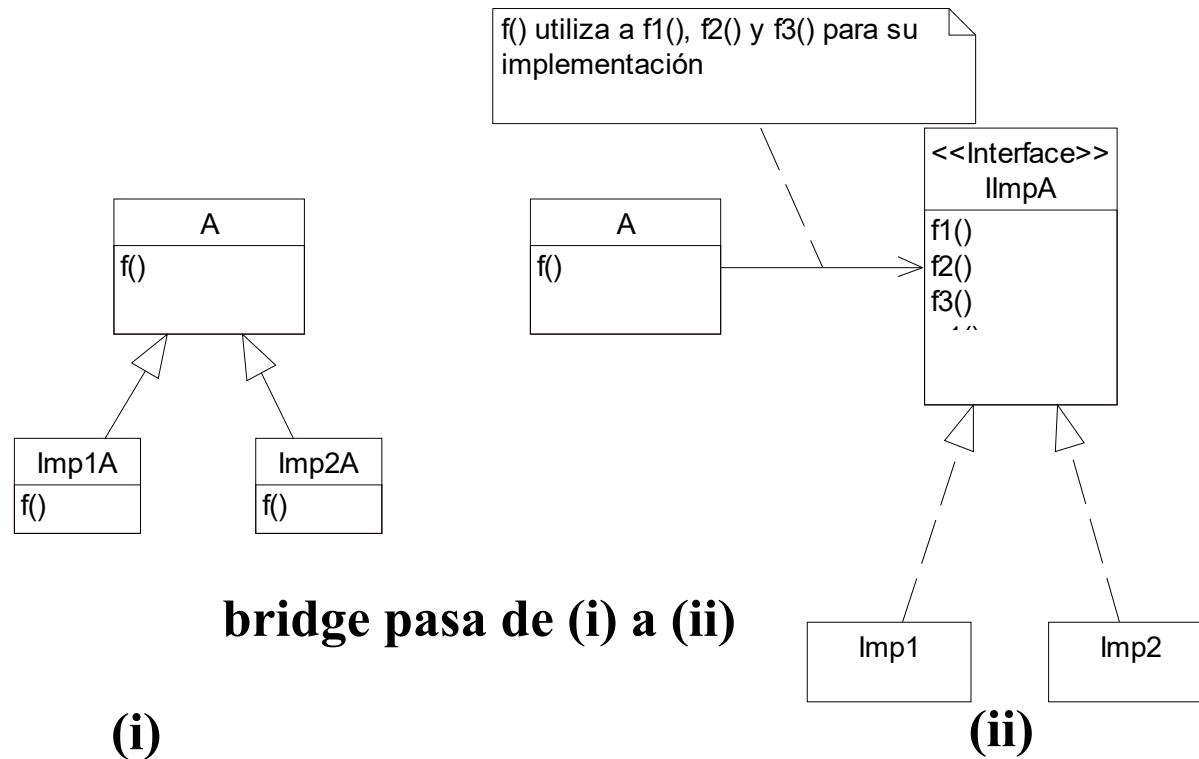


(ii)

El patrón bridge con una clase que refina a la abstracción

# Patrones estructurales

## Bridge



**El patrón bridge sin una clase que refine a la abstracción**

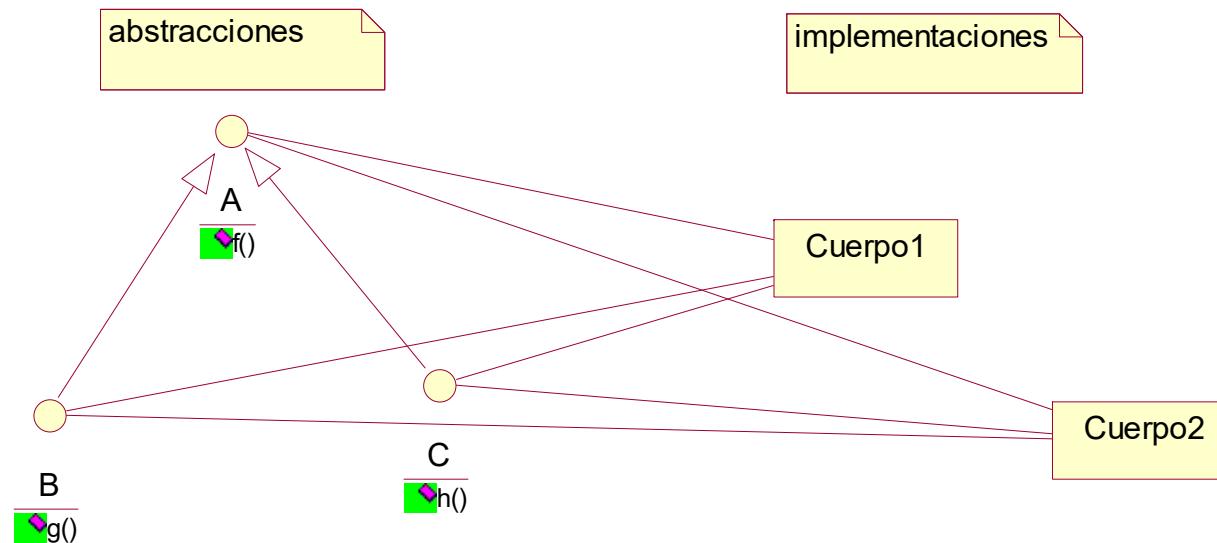
# Patrones estructurales

## Bridge

- ¿Cómo se implementa el patrón Bridge si las abstracciones son interfaces, y no clases?
- ¿Serviría la siguiente implementación?

# Patrones estructurales

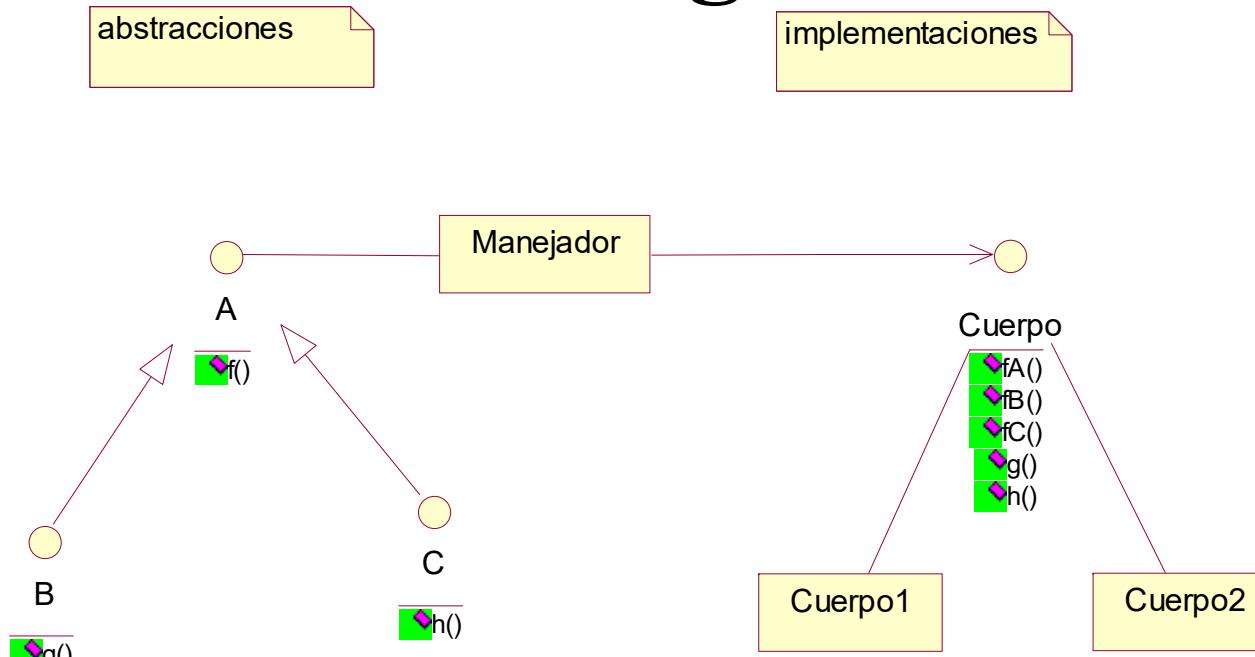
## Bridge



¿Patrón Bridge con interfaces?

# Patrones estructurales

## Bridge



¿Patrón Bridge con interfaces?

# Patrones estructurales

## Bridge

- Bridge vs. factoría abstracta
  - En bridge, VentanaImpX no es una Ventana, es una VentanaImp. En cambio, en factoría abstracta, VentanaX sí es una Ventana
- Bridge vs. strategy
  - En bridge, VentanaImp da soporte a TODAS las operaciones de Ventana y sus subclases. En cambio en strategy Componedor da soporte a UNA ÚNICA OPERACIÓN de Composicion

# Patrones estructurales

## Bridge

- Es decir, VentanaImp es un objeto que *simula* (implementa en su totalidad) a Ventana y sus subclases mientras que Componedor es un objeto que es utilizado puntualmente (implementa una única operación) por Composición
- Bridge vs. adapter
  - Según lo anterior, Ventana es una especie de adaptador de VentanaImp

# Patrones estructurales

## Bridge

- La diferencia radica en un adaptador se aplica a objetos existentes, mientras que un puente se aplica para permitir que implementaciones y abstracciones varíen independientemente unas de otras.
- Bridge vs. Template Method
  - En la medida en que las operaciones de las abstracciones contengan código común a todas las implementaciones, el Bridge es también una especie de Template Method

# Patrones estructurales

## Bridge

- Eso sí, en vez de tener la implementación de los métodos plantilla en las subclases, la tiene en el cuerpo (herencia vs. puntero).

# Patrones estructurales

## Composite

- Propósito
  - Compone objetos en estructuras de árbol para representar jerarquías de parte-todo. Permite que los clientes traten de manera uniforme a los objetos individuales y a los compuestos.
- También conocido como
  - Compuesto.

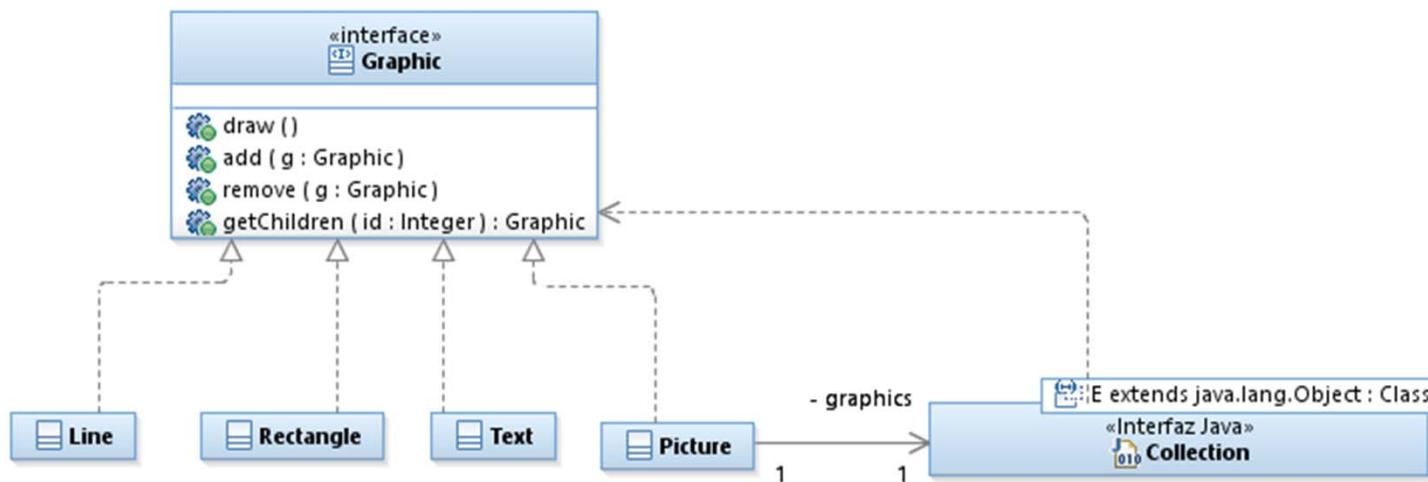
# Patrones estructurales

## Composite

- Motivación
  - Las aplicaciones gráficas como los editores de dibujo permiten a los usuarios construir diagramas complejos a partir de componentes simples.
  - El código que usa estas clases debe tratar de forma diferente a los objetos primitivos y a los contenedores, aunque se traten de forma idéntica.

# Patrones estructurales

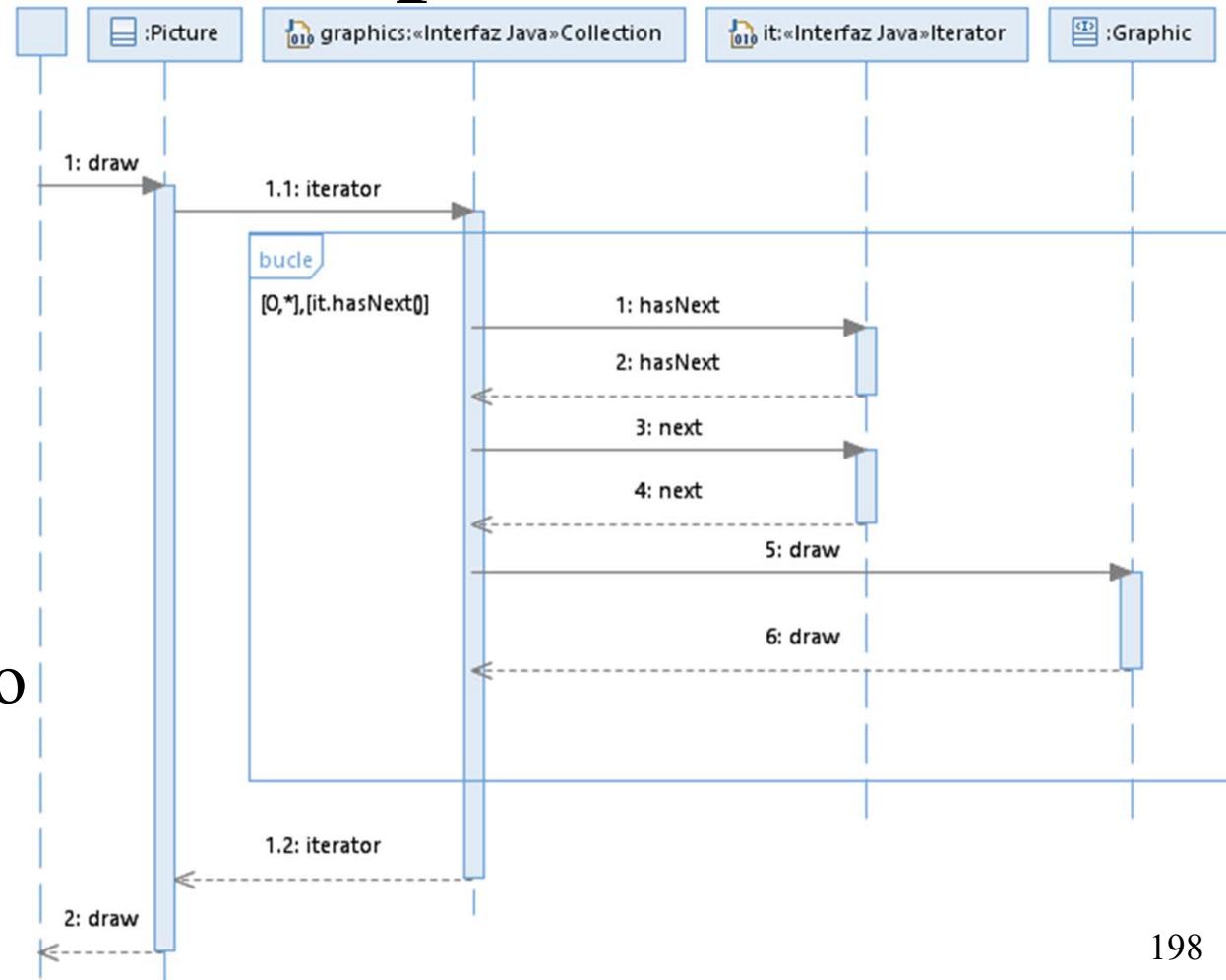
## Composite



Ejemplo estructura patrón Composite

# Patrones estructurales

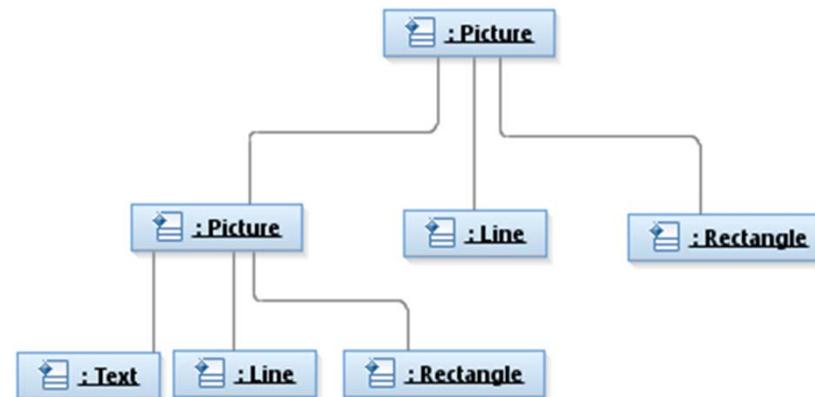
## Composite



Ejemplo  
comportamiento  
Composite

# Patrones estructurales

## Composite



Objetos compuestos

# Patrones estructurales

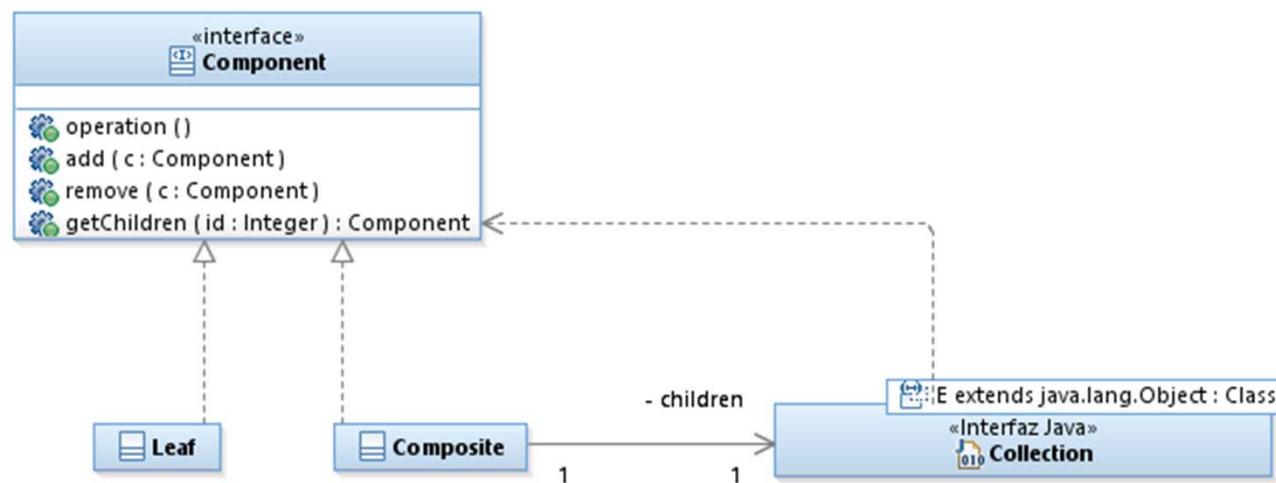
## Composite

- El patrón Composite debe aplicarse cuando
  - Se quiera representar jerarquías de objetos parte-todo.
  - Se quiera que los clientes sean capaces de obviar las diferencias entre composiciones de objetos y los objetos individuales. Los clientes tratarán a todos los objetos de la estructura compuesta de manera uniforme.

# Patrones estructurales

## Composite

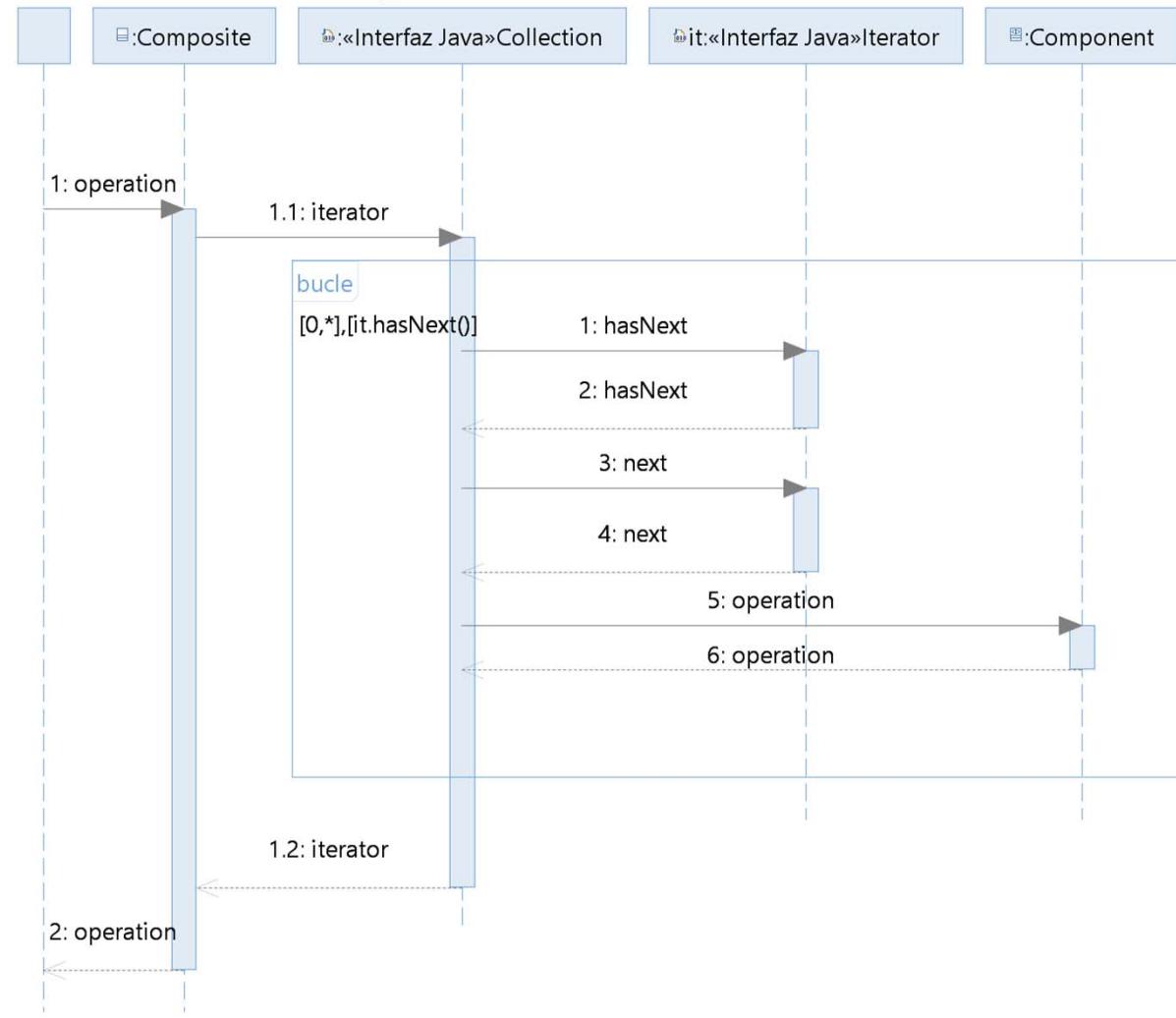
- Descripción abstracta



Estructura del patrón Composite

# Patrones estructurales

## Composite

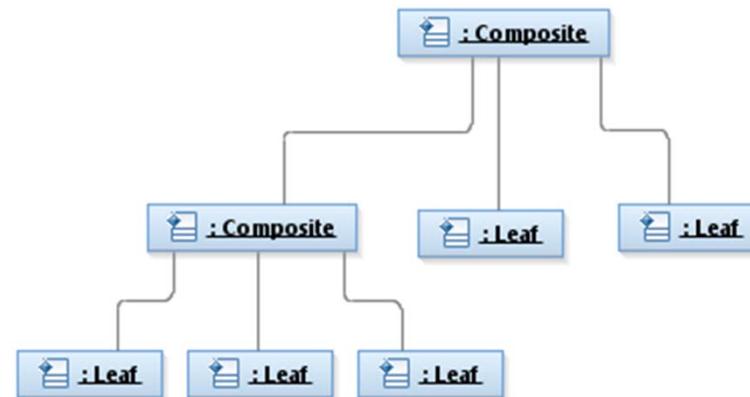


Comportamiento  
Composite

Ingeniería del Software  
Antonio Navarro

# Patrones estructurales

## Composite



Estructura de objetos compuestos

# Patrones estructurales

## Composite

- Consecuencias
  - Ventajas
    - Define jerarquías de clases formadas por objetos primitivos y compuestos.
    - Permite un tratamiento uniforme por parte del cliente.
    - Facilita añadir nuevos tipos de componentes.
  - Inconvenientes
    - Oculta los tipos de los objetos dentro del compuesto.

# Patrones estructurales

## Composite

- Código de ejemplo

```
public interface IComponente {
```

```
    public String nombre();
```

```
    float potencia();
```

```
    float precioNeto();
```

```
    float precioDescuento();
```

# Patrones estructurales

## Composite

```
public void anadir(IComponente e);  
public void eliminar(IComponente e);  
public Iterador crearIterador();  
  
}  
  
public class Procesador implements IComponente {  
    .....  
}  
public class DVD implements IComponente {  
    .....  
}
```

# Patrones estructurales

## Composite

```
//clase base para todos los compuestos
public class Compuesto implements IComponente {
    String nombre;
    Float precioComponenteCompuesto;
    Lista listaIComponente;

    float precioNeto()
    { Iterador i= crearIterador();

        float total= precioComponenteCompuesto;
        for (i.primero; !i.haTerminado(); i.siguiente())
            total += i.elementoActual().precioNeto();
        return total;
    }
    ...
}
```

# Patrones estructurales

## Composite

```
public class Ordenador extends Compuesto { ... }  
public class Placa extends Compuesto { ... }  
  
IComponente ordenador= new Ordenador("PC low", ...);  
IComponente placa= new Placa("B75", ...);  
IComponente procesador= new Procesador("i5-3570T",  
...);  
IComponente dvd= new DVD("+-RW", ...);  
placa.anadir(procesador);  
ordenador.anadir(placa);  
ordenador.anadir(dvd);  
ordenador.precioNeto();
```

# Patrones estructurales

## Composite

- Nota:
  - El código anterior es el típico ejemplo a evitar de *utilizo un interfaz y creo directamente su implementación*

# Patrones estructurales

## Decorator

- Propósito
  - Asigna responsabilidades adicionales a un objeto dinámicamente, proporcionando una alternativa flexible a la herencia para extender la funcionalidad.
- También conocido como
  - Decorador.
  - Wrapper (envoltorio).

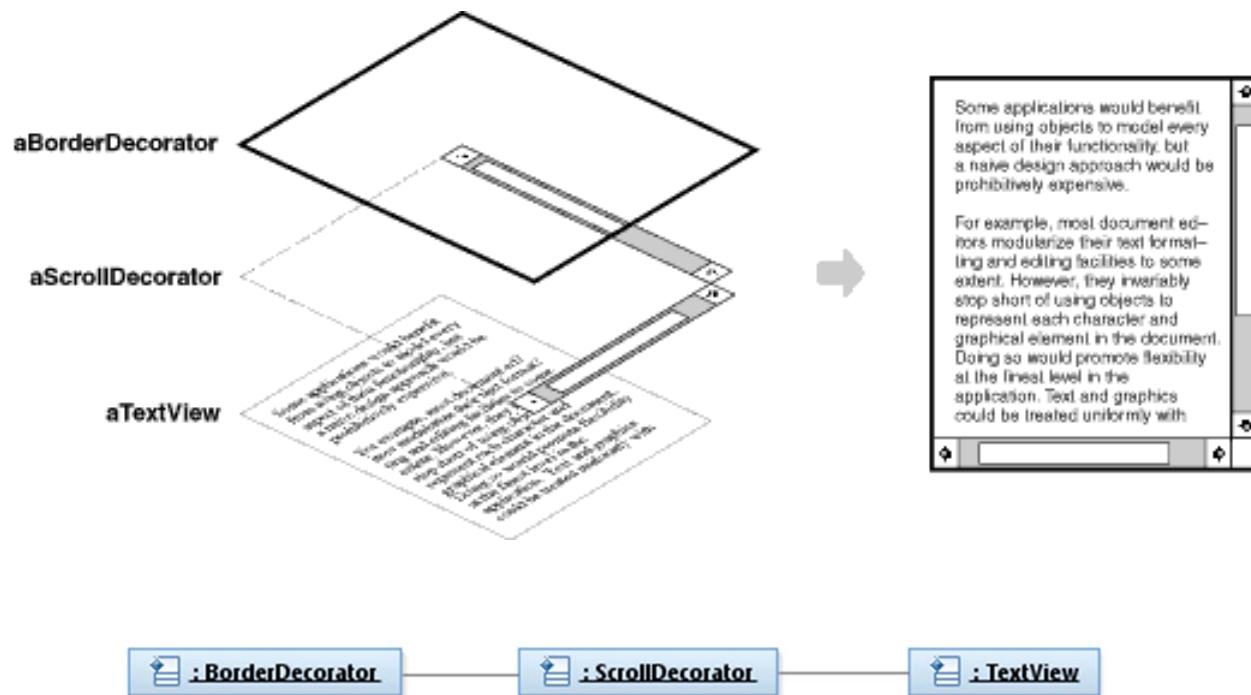
# Patrones estructurales

## Decorator

- Motivación
  - A veces queremos añadir responsabilidades a objetos individuales en vez de a toda una clase.
  - Por ejemplo, los bordes o desplazamientos asociados a componentes de una IGU.
  - La herencia es una solución estática.
  - Un enfoque más flexible es encerrar el componente en otro objeto que añada el borde.
  - Así, podemos seguir una aproximación recursiva.

# Patrones estructurales

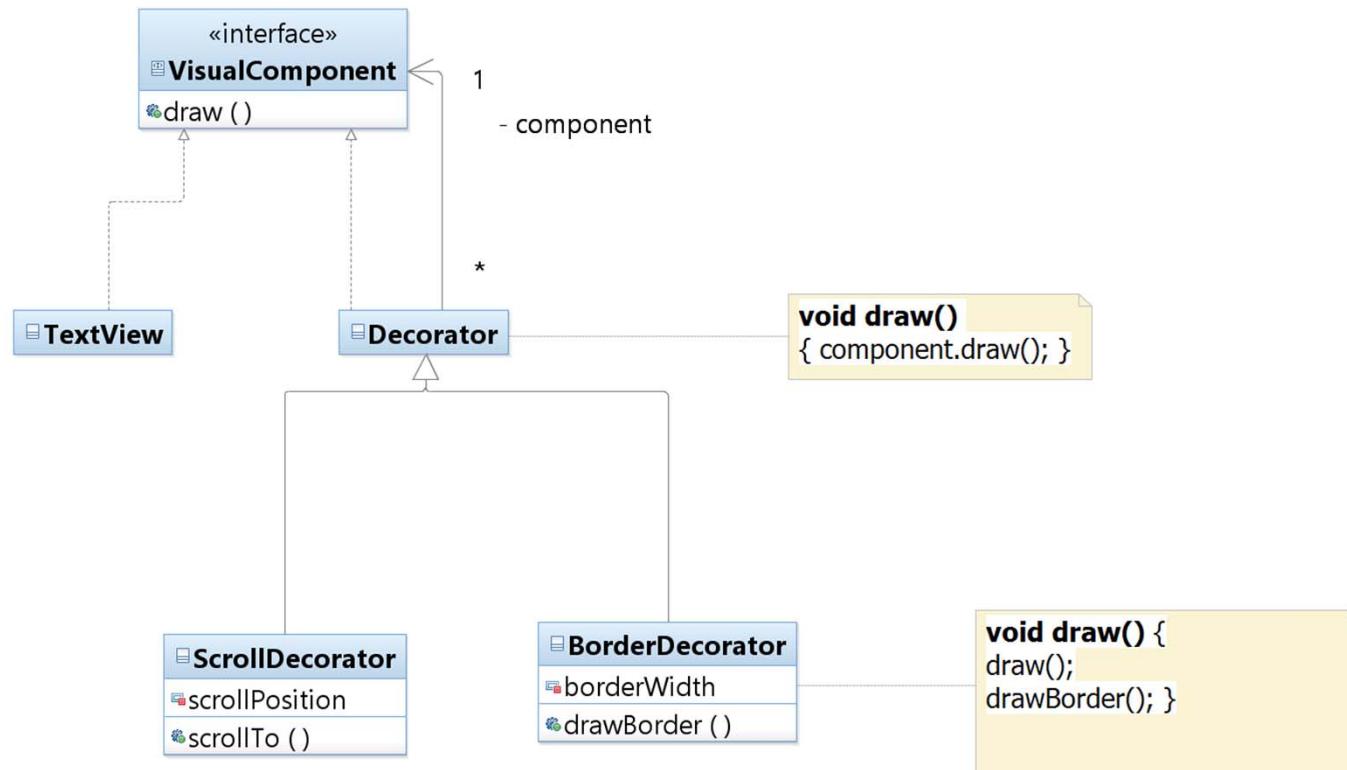
## Decorator



## Composición de decoradores

# Patrones estructurales

## Decorator



### Ejemplo patrón Decorator

# Patrones estructurales

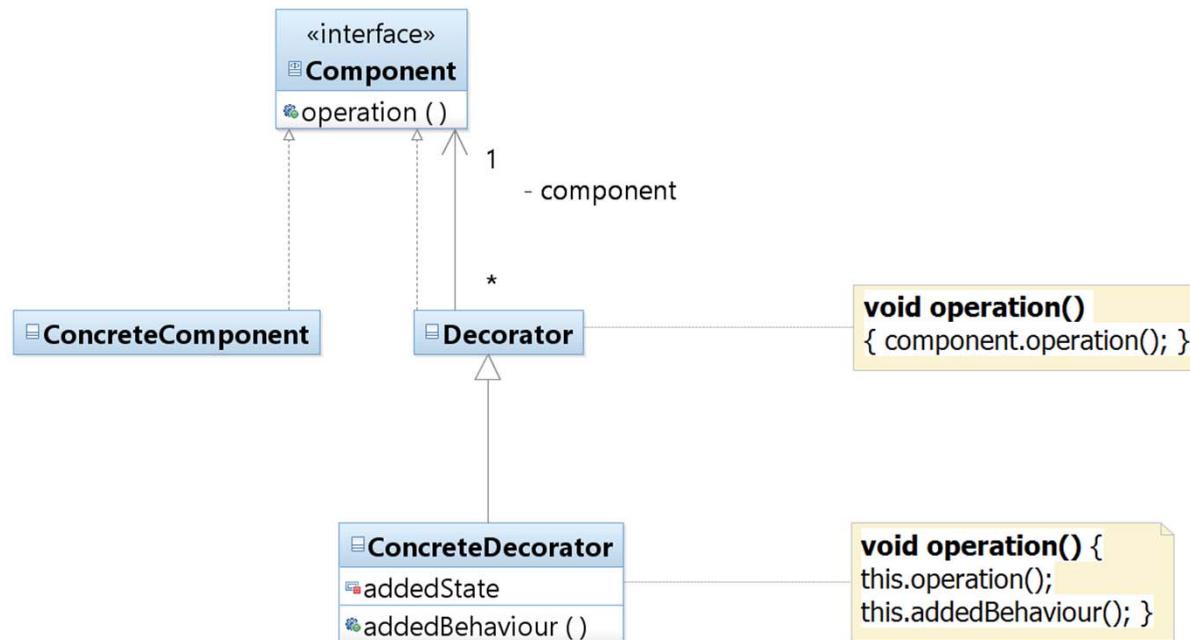
## Decorator

- El patrón Decorator debe aplicarse cuando
  - Se desee añadir objetos individuales de forma dinámica y transparente, es decir, sin afectar a otros objetos.
  - Se desee la posibilidad de eliminar responsabilidades.
  - Cuando la extensión mediante herencia no es viable por la explosión de subclases que se producen.

# Patrones estructurales

## Decorator

- Descripción abstracta



Estructura y comportamiento del patrón Decorator

# Patrones estructurales

## Decorator

- Consecuencias
  - Ventajas
    - Más flexibilidad que la herencia múltiple estática.
    - Evita clases cargadas con funciones en la parte de arriba de la jerarquía, ya que pueden añadirse en el decorador.
  - Inconvenientes
    - Un decorador y su componente no son idénticos.
    - Aparecen muchos objetos pequeños.

# Patrones estructurales

## Decorator

- Código de ejemplo

```
public interface ComponenteVisual {  
    public void dibujar();  
    public void cambiarTamano();  
};
```

# Patrones estructurales

## Decorator

```
class Decorador implements ComponenteVisual {  
    ComponenteVisual componente;  
  
    public Decorador(ComponenteVisual c)  
    { componente= c; }  
  
    public void dibujar()  
    { componente.dibujar(); }  
  
    public void cambiarTamano()  
    { componente.cambiarTamano(); }  
};
```

# Patrones estructurales

## Decorator

```
public class DecoradorBorde extends Decorator {  
    int anchoBorde;  
  
    public DecoradorBorde(ComponenteVisual c, int a)  
    { super(c); anchoBorde= a; }  
  
    private void dibujarBorde(int ancho) {...};  
  
    public void dibujar(){  
        super.dibujar();  
        dibujarBorde(anchoBorde);  
    }  
};
```

# Patrones estructurales

## Decorator

- Discusión: en el ejemplo anterior, ¿la clase Decorador es imprescindible?
- ¿Cuándo es imprescindible la clase Decorador?

# Patrones estructurales

## Façade

- Propósito
  - Proporciona una interfaz unificada para un conjunto de interfaces de un subsistema. Define una interfaz de alto nivel que hace que el subsistema sea más fácil de usar.
- También conocido como
  - Fachada.

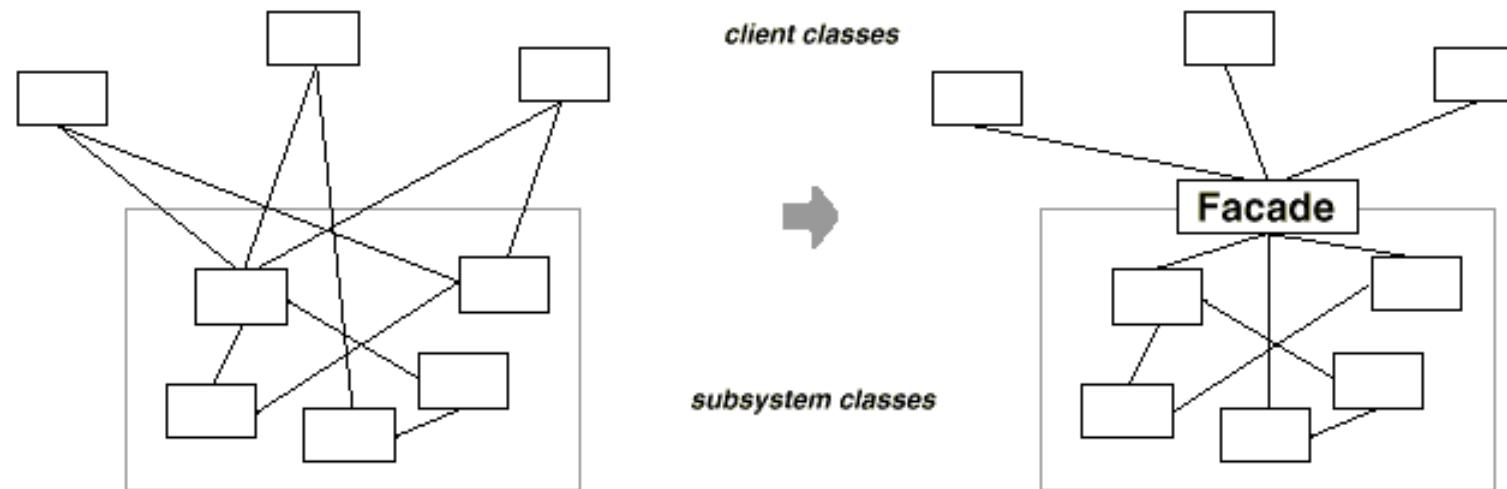
# Patrones estructurales

## Façade

- Motivación
  - Estructurar un sistema en subsistemas ayuda a reducir la complejidad.
  - Un objetivo clásico en diseño es minimizar la comunicación y dependencias entre subsistemas.
  - Un modo de lograr esto es introduciendo un objeto fachada que proporcione una interfaz única y simplificada para los servicios más generales del subsistema.

# Patrones estructurales

## Façade



El patrón Façade

# Patrones estructurales

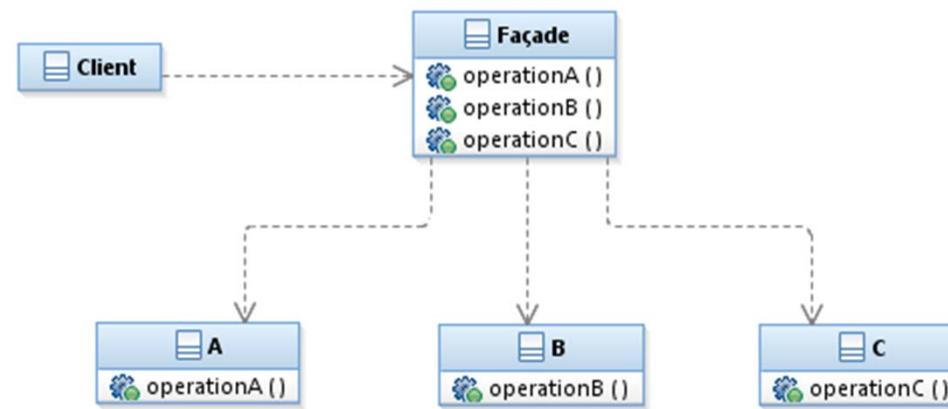
## Façade

- El patrón Façade debe aplicarse cuando
  - Queramos proporcionar una interfaz simple para un subsistema complejo.
  - Haya muchas dependencias entre los clientes y las clases que implementan una abstracción.
  - Queramos dividir en capas nuestros subsistemas.

# Patrones estructurales

## Façade

- Descripción abstracta



Estructura del patrón Façade

# Patrones estructurales

## Façade

- Consecuencias
  - Ventajas
    - Oculta a los clientes los componentes del subsistema, reduciendo así el número de objetos con los que traten los clientes y haciendo que el subsistema sea más fácil de usar.
    - Promueve un débil acoplamiento entre el subsistema y los clientes.
    - No impide que las aplicaciones utilicen clases del subsistema si es necesario.

# Patrones estructurales

## Façade

- Inconvenientes

- Nuevas operaciones de los componentes deben promocionar hacia la interfaz de la fachada.

# Patrones estructurales

## Façade

- Código de ejemplo:

```
public interface Biblioteca {  
    public Integer insertaUsuario(TUsuario usuario);  
    public Boolean daDeBajaUsuario(Integer id);  
    public TDAOUsuario obtenUsuario(Integer id);  
    public Integer insertaPublicacion(TPublicacion  
publicacion);  
    public Boolean daDeBajaPublicacion(Integer id);  
    public TDAOPublicacion obtenPublicacion(Integer id);  
    public TPrestamo prestamo(TPrestamo tPrestamo);  
    public Boolean devolucion(Integer ejemplar); }
```

# Patrones estructurales

## Façade

```
public class BibliotecaImp implements Biblioteca {  
    public Integer insertaUsuario(TUsuario usuario)  
    { ..... }  
  
    public Boolean daDeBajaUsuario(Integer id)  
    { //nótese que no se ha hecho explícito  
        //el acceso a estos servicios por parte de la  
        //Biblioteca  
        serviciosUsuario.daDeBajaUsuario(id);  
    }  
.....  
}
```

# Patrones estructurales

## Flyweight

- Propósito
  - Comparte objetos de grano fino para permitir la existencia un gran número de estos objetos de forma eficiente.
- También conocido como
  - Peso ligero

# Patrones estructurales

## Flyweight

- Motivación
  - Utilizar copias de objetos para todo es muy costoso en recursos e integridad.
  - Por ejemplo, los caracteres de un documento.
  - El patrón Flyweight describe cómo compartir objetos para permitir su uso con granularidades muy finas, sin un coste prohibitivo.

# Patrones estructurales

## Flyweight

- Un *peso ligero* es un objeto compartido que puede usarse a la vez en varios contextos.
- El peso ligero es un objeto independiente en cada contexto, no se puede distinguir de una instancia del objeto que no esté compartida.
- Los pesos ligeros no pueden hacer suposiciones sobre el contexto en el cual operan.
- Lo fundamental es la distinción entre estado *intrínseco* y *extrínseco*.

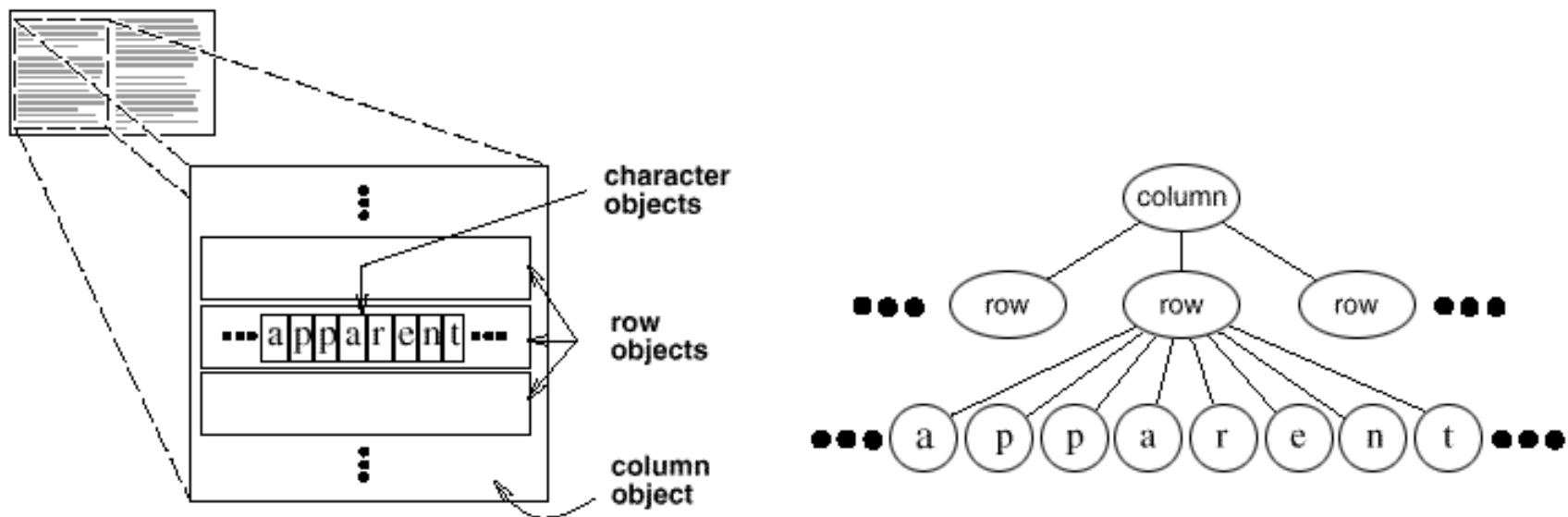
# Patrones estructurales

## Flyweight

- El estado intrínseco se guarda en el propio objeto. Consisten en información que es independiente de su contexto y que puede ser compartida.
- El estado extrínseco depende del contexto y cambia con él, por lo que no puede ser compartido.
- Los objetos cliente son responsables de pasar al peso ligero su estado extrínseco cuando lo necesite.

# Patrones estructurales

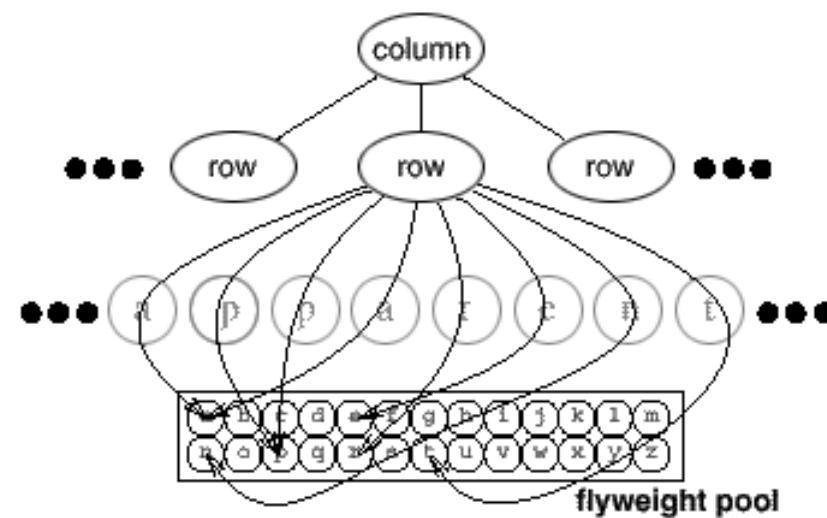
## Flyweight



El patrón Flyweight

# Patrones estructurales

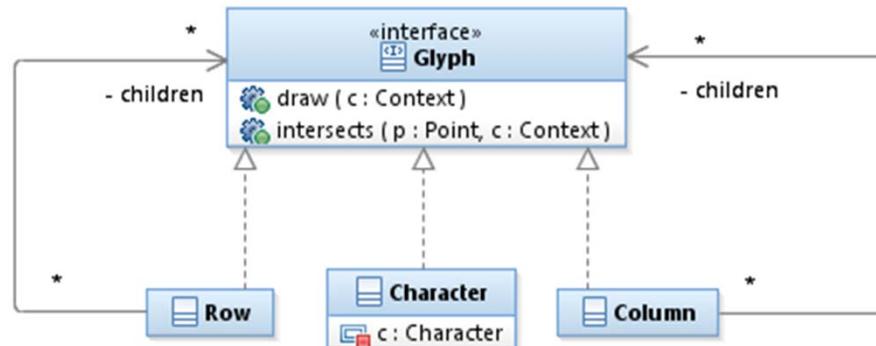
## Flyweight



El patrón Flyweight

# Patrones estructurales

## Flyweight



Ejemplo estructura patrón Flyweight

# Patrones estructurales

## Flyweight

- El patrón Flyweight debe aplicarse cuando
  - Una aplicación utilice un gran número de objetos, y
  - Los costes de almacenamiento sean elevados debido a la gran cantidad de objetos, y
  - La mayor parte del estado del objeto pueda hacerse extrínseco, y

# Patrones estructurales

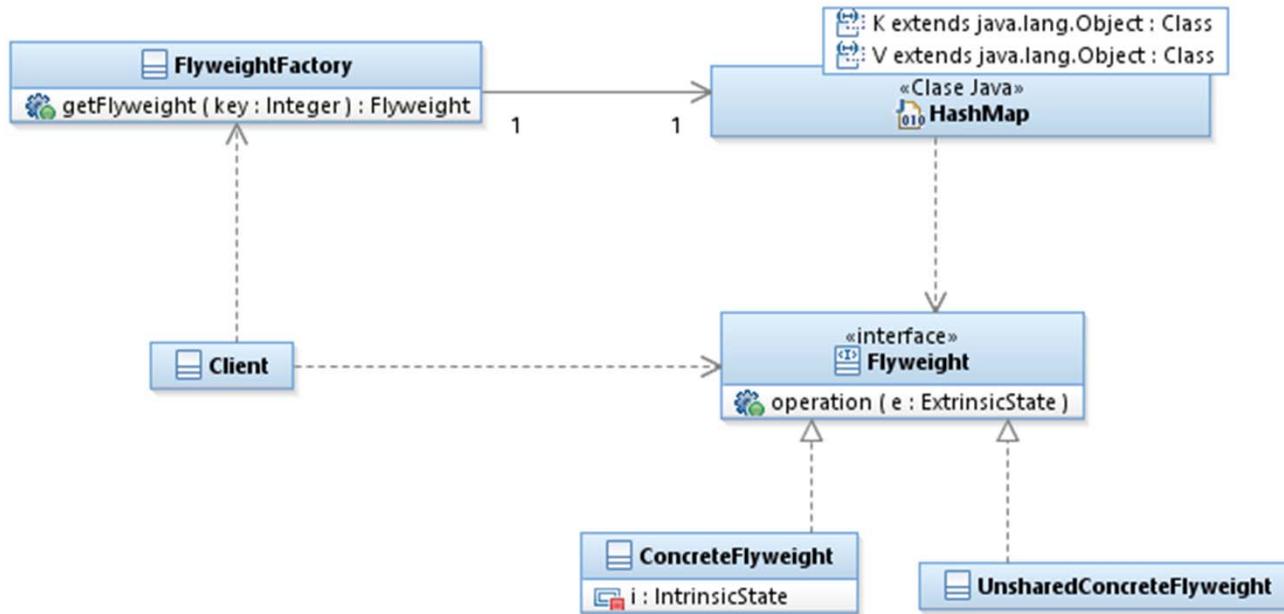
## Flyweight

- Muchos grupos de objetos puedan reemplazarse por relativamente pocos objetos compartidos, una vez que se ha eliminado el estado extrínseco, y
- La aplicación no depende de la identidad de un objeto. Puesto que los objetos peso ligero pueden ser compartidos, las comprobaciones de identidad devolverán verdadero para objetos conceptualmente distintos.

# Patrones estructurales

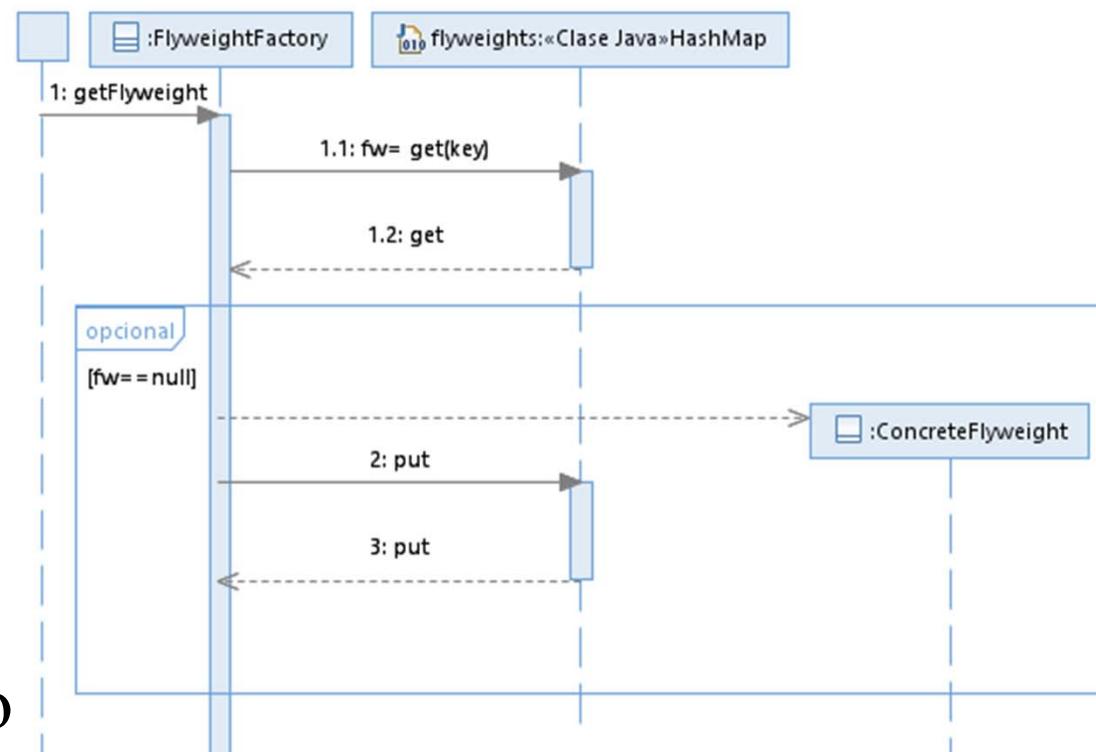
## Flyweight

- Descripción abstracta



# Patrones estructurales

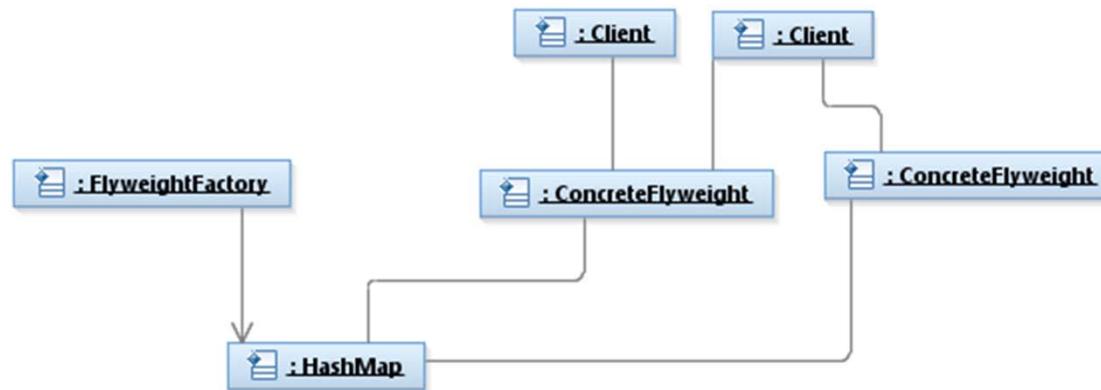
## Flyweight



Comportamiento  
Flyweight

# Patrones estructurales

## Flyweight



Objetos Flyweight compartidos

# Patrones estructurales

## Flyweight

- Consecuencias
  - Ventajas
    - Ahorro de almacenamiento.
    - Integridad.
  - Inconvenientes
    - Costes de tiempo de ejecución.

# Patrones estructurales

## Flyweight

- Código de ejemplo

```
public interface Glifo {  
    public void dibujar(Ventana v, ContextoGlifo cg);  
    public void establecerFuente(Fuente f, ContextoGlifo cg);  
    public void obtenerFuente(ContextoGlifo cg);  
    public void primero(ContextoGlifo cg);  
    public void siguiente(ContextoGlifo cg);  
    public Boolean haTerminado(ContextoGlifo cg);  
    public Glifo actual(ContextoGlifo cg);  
    public void insertar(Glifo g, ContextoGlifo cg);  
    public void borrar(ContextoGlifo cg); };
```

# Patrones estructurales

## Flyweight

```
public Class Caracter implements Glifo {  
    char codigoCaracter;  
  
    public Carácter(char c)  
    { codigoCaracter= c; }  
  
    public void Dibujar(Ventana v, ContextoGlifo cg)  
    {...};  
};
```

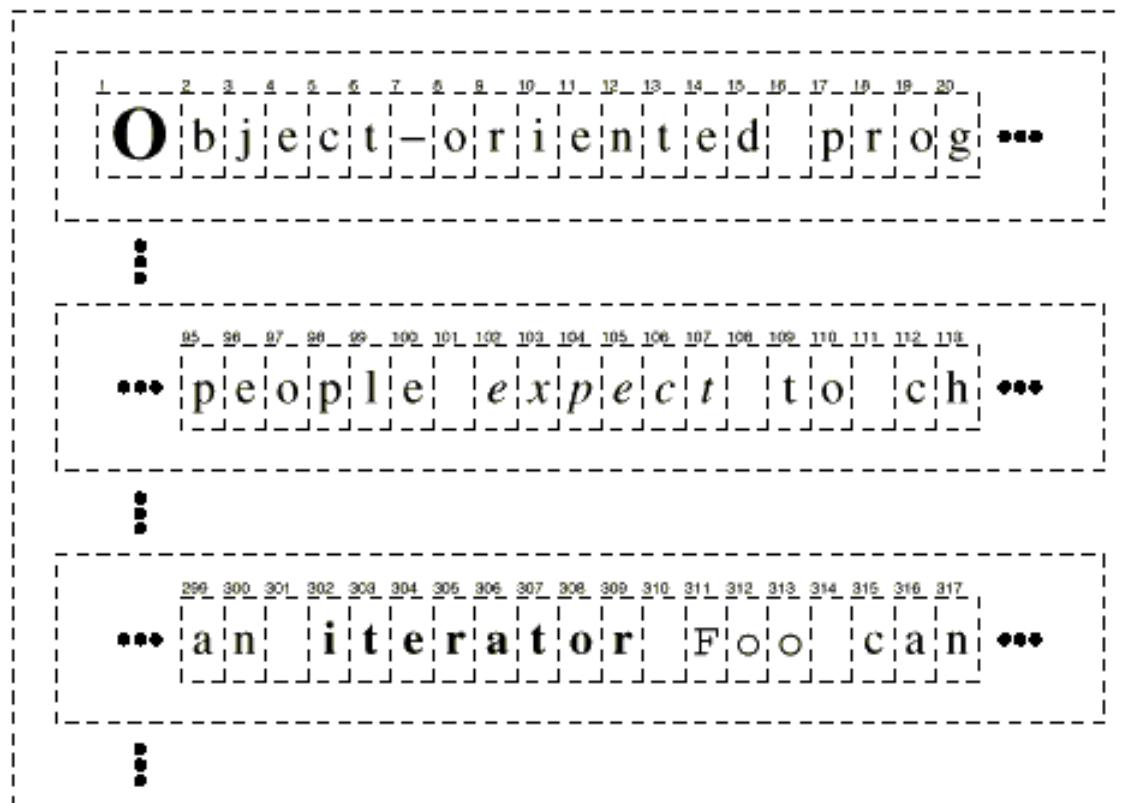
# Patrones estructurales

## Flyweight

```
class ContextoGlifo {  
    int indice;  
    Arbol fuentes;  
    public ContextoGlifo() {...};  
    public void siguiente(int incremento)  
        {indice += incremento};  
    public void insertar (int cantidad) {...};  
    public Fuente obtenerFuente();  
    public void establecerFuente(Fuente f, int  
                                extension) {...};  
};
```

# Patrones estructurales

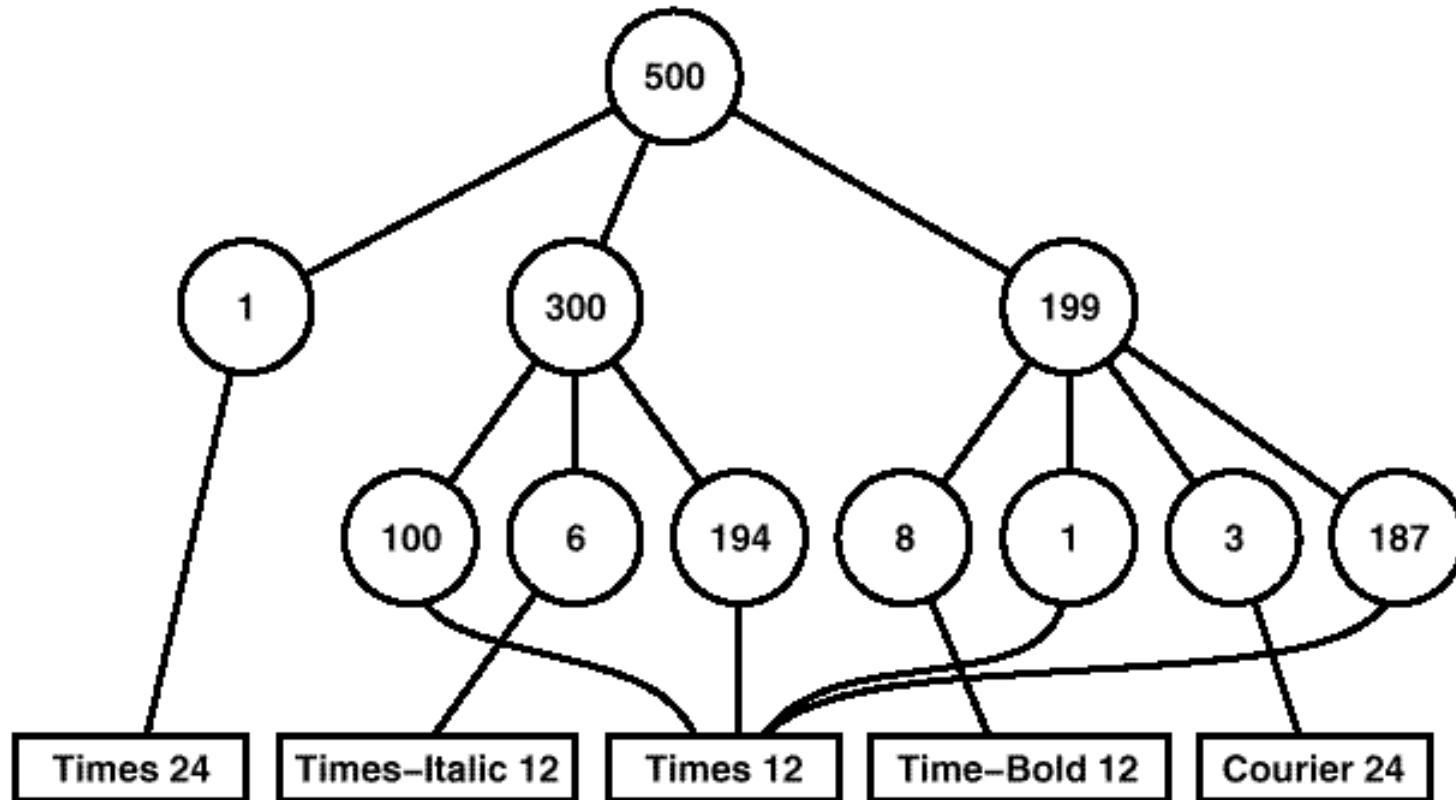
## Flyweight



Texto

# Patrones estructurales

## Flyweight



Representación del Contexto para el texto anterior

# Patrones estructurales

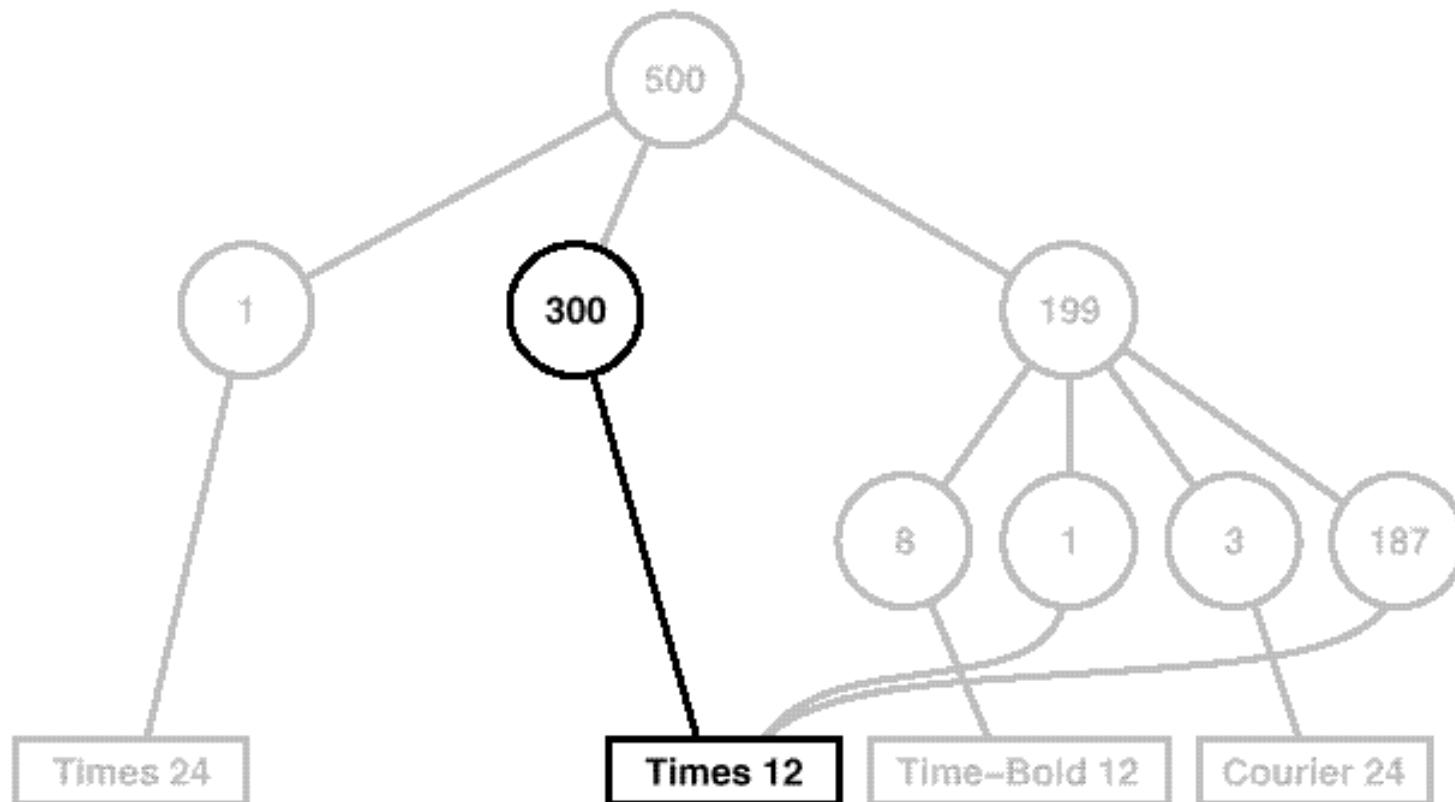
## Flyweight

Si en el índice 102 queremos que expect sea Times Roman de 12 ptos:

```
Fuente times12= new Fuente ("Times-Roman-12");  
Fuente timesItalic12= new Fuente("Times-Italic-  
12");  
.....  
contextoGlifo.establecerFuente(times12, 6);
```

# Patrones estructurales

## Flyweight



Nuevo estado del contexto

# Patrones estructurales

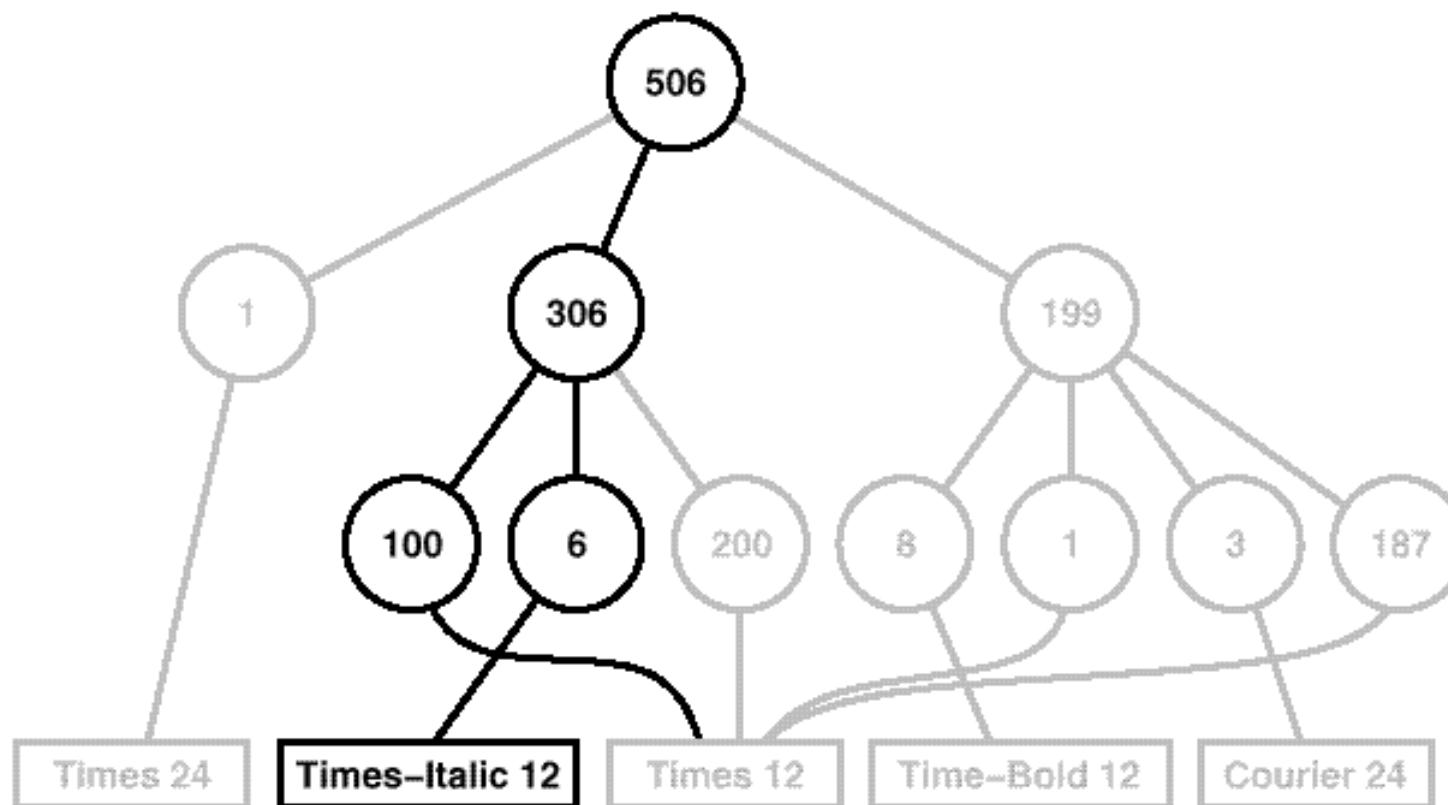
## Flyweight

Si deseamos añadir la palabra don't (con el espacio en blanco) en Times Italic de 12 puntos, antes de expect, y suponiendo que estemos en el índice 102:

```
contextoGlifo.insertar(6);
contextoGlifo.establecerFuente(timesItalic12, 6);
```

# Patrones estructurales

## Flyweight



Nuevo estado del contexto

# Patrones estructurales

## Flyweight

```
public class FabricaDeGlifos {  
    int NCODIGOSC= 128;  
    Caracter caracter[NCODIGOSC];  
  
    public FabricaDeGlifos()  
    { for (int i= 0; i<NCODIGOSC; i++)  
        caracter[i]= null; }  
  
    public Caracter crearCaracter(char c)  
    { if (caracter[c] == null) caracter[c]=  
        new Caracter(c);  
    return caracter[c]; }
```

# Patrones estructurales

## Flyweight

```
public Fila crearFila()
{ return new Fila(); }

public Columna crearColumna()
{ return new Columna(); }

. . . . .

};
```

# Patrones estructurales

## Proxy

- Propósito
  - Proporciona un representante o sustituto de otro objeto para controlar el acceso a éste.
- También conocido como
  - Apoderado.
  - Surrogate (Sustituto).

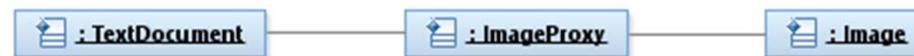
# Patrones estructurales

## Proxy

- Motivación
  - Una razón para controlar el acceso a un objeto es retrasar todo el coste de su creación e inicialización hasta que sea realmente necesario usarlo.
  - Por ejemplo, en el contexto de recuperar varios objetos de un archivo.
  - Por ejemplo, para manejar imágenes en un procesador de textos.

# Patrones estructurales

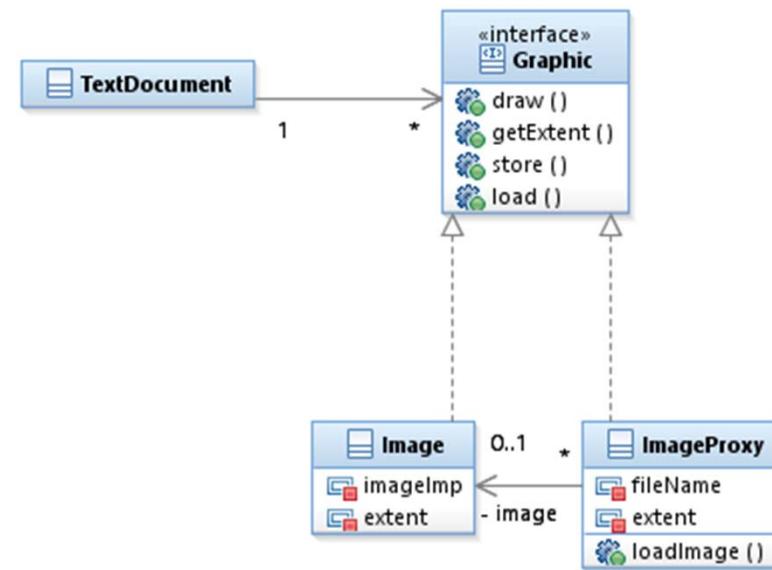
## Proxy



Ejemplo de proxy

# Patrones estructurales

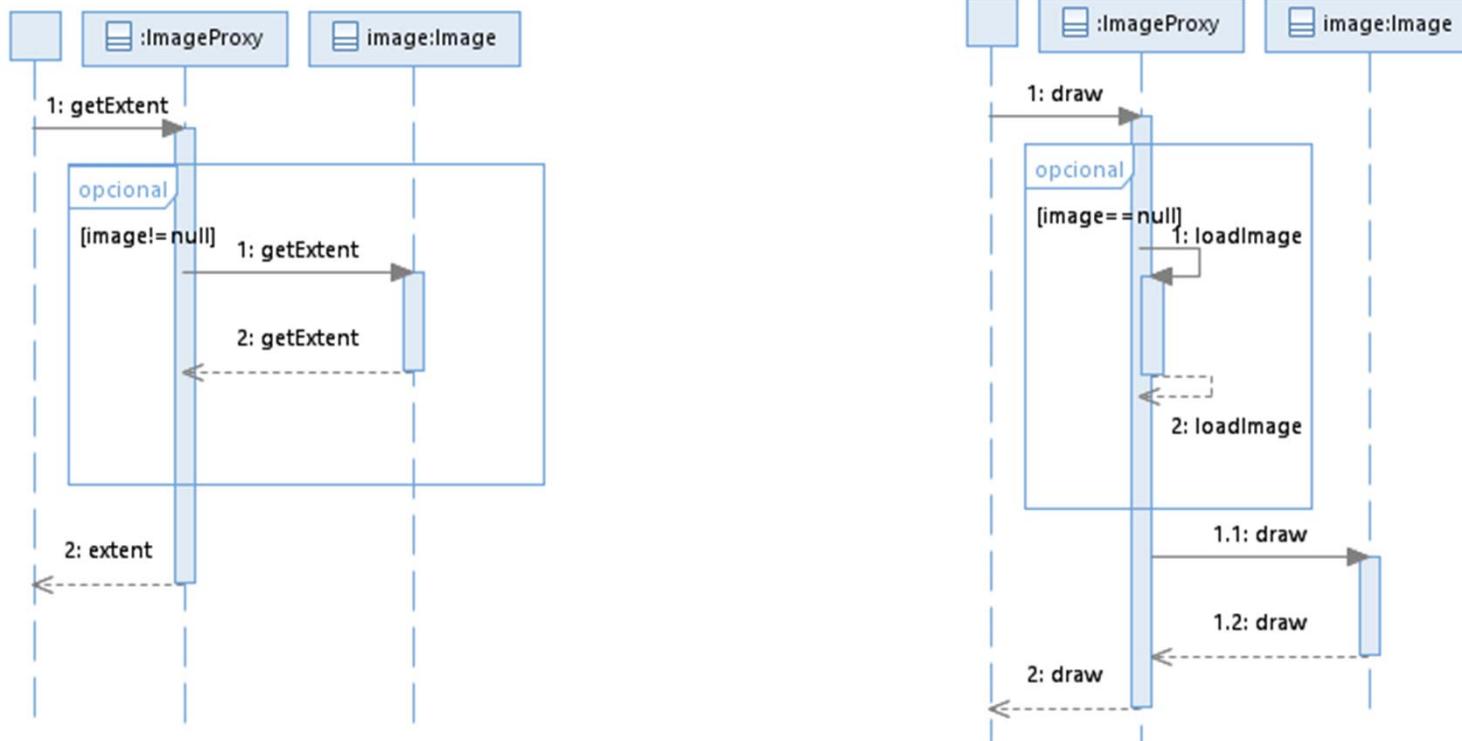
## Proxy



Ejemplo estructura patrón Proxy

# Patrones estructurales

## Proxy



Ejemplo comportamiento patrón Proxy

# Patrones estructurales

## Proxy

- El patrón Proxy debe aplicarse cuando
  - Hay necesidad de una referencia a un objeto más versátil o sofisticada que un simple puntero.
  - Tipos de Proxy:
    - *Remoto*. Proporciona un representante local de un objeto situado en otro espacio de direcciones.
    - *Virtual*. Crea objetos costosos por encargo.
    - *De protección*. Controla el acceso al objeto original.

# Patrones estructurales

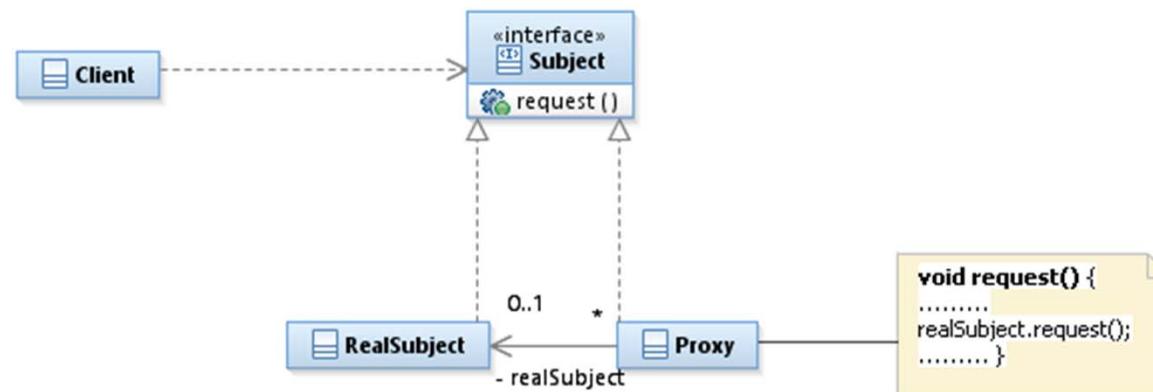
## Proxy

- *Referencia inteligente*. Es un sustituto de un simple puntero que lleva a cabo operaciones adicionales cuando se accede a un objeto. Por ejemplo:
  - Contar el número de referencias al objeto real de forma que éste pueda liberarse automáticamente cuando no haya ninguna referencia apuntándole (punteros inteligentes).
  - Cargar un objeto persistente en la memoria cuando es referenciado por primera vez.
  - Comprobar que se bloquea el objeto real antes de acceder a él para garantizar que no pueda ser modificado por ningún otro objeto.

# Patrones estructurales

## Proxy

- Descripción abstracta



Estructura y comportamiento del patrón Proxy

# Patrones estructurales

## Proxy



Objetos según el patrón Proxy

# Patrones estructurales

## Proxy

- Consecuencias
  - Ventajas
    - Un proxy remoto oculta el hecho de residir en un espacio de direcciones diferente.
    - Un proxy virtual puede llevar a cabo optimizaciones tales como crear objetos por encargo.
    - Un proxy de protección o una referencia inteligente, permite realizar tareas adicionales cuando se accede a un objeto.
    - Copia de escritura: flyweight + copia a clientes que quieren modificarlo

# Patrones estructurales

## Proxy

- Código de ejemplo (proxy virtual)

```
public interface Grafico extends Serializable {  
    public void dibujar(Punto en);  
    public void manejarRaton(Evento e);  
    public Punto obtenerExtension();  
};
```

# Patrones estructurales

## Proxy

```
public class Imagen implements Grafico {  
    Imagen(String fichero) {...};  
    public void dibujar(Punto en) {...};  
    public void manejarRaton(Evento e) {...};  
    public Punto obtenerExtension() {...};  
    .....  
};
```

# Patrones estructurales

## Proxy

```
public class ProxyImagen implements Grafico {  
    Imagen imagen;  
    Punto extension;  
    String nombreFichero;  
  
    public ProxyImagen(String nombreFicheroP)  
    { nombreFichero= nombreFicheroP;  
        //aquí leería la extensión de la imagen  
        imagen= null;  
    }
```

# Patrones estructurales

## Proxy

```
protected Imagen obtenerImagen()
{ if (imagen==null) imagen= new Imagen(nombreFichero);
  return imagen; }

public Punto obtenerExtension()
{ if (imagen==null) return extension;
  else return obtenerImagen().obtenerExtension();
}

}
```

# Patrones estructurales

## Proxy

```
public void dibujar (Punto en)
{ obtenerImagen( ).dibujarEn( ); }

public void manejarRaton(Evento e)
{ obtenerImagen( ).manejarRaton(e); }
```

# Patrones estructurales

## Proxy

```
public class DocumentoDeTexto {  
    ....  
    public void insertar(Grafico g) {...};  
    ....  
};
```

```
DocumentoDeTexto texto= new DocumentoDeTexto();  
texto.insertar(new  
    ProxyImagen("nombreFicheroImagen"));
```

# Patrones de comportamiento

## Introducción

- Los patrones de comportamiento tienen que ver con algoritmos y con la asignación de responsabilidades a objetos
- Estos patrones describen patrones de clases objetos y los patrones de comunicación entre estas clases y objetos

# Patrones de comportamiento

## Introducción

- Los de clases usan la herencia para distribuir el comportamiento entre clases
- Los de objetos usan la composición para distribuir dicho comportamiento

# Patrones de comportamiento

## Chain of Responsibility

- Propósito
  - Evita acoplar el emisor de una petición a su receptor, dando a más de un objeto la posibilidad de responder a la petición.  
Encadena los objetos receptores y pasa la petición a través de una cadena hasta que es procesada por algún objeto.
- También conocido como
  - Cadena de responsabilidad.

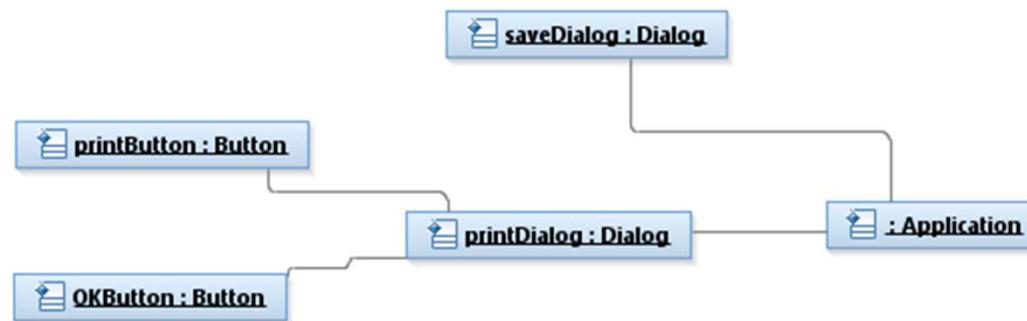
# Patrones de comportamiento

## Chain of Responsibility

- Motivación
  - Supongamos una ayuda contextual para una interfaz gráfica de usuario.
  - Si no hay información específica para un determinado contexto, se puede dar una más general.
  - El problema es que el objeto que proporciona la ayuda no conoce el objeto que inicializa la petición de ayuda

# Patrones de comportamiento

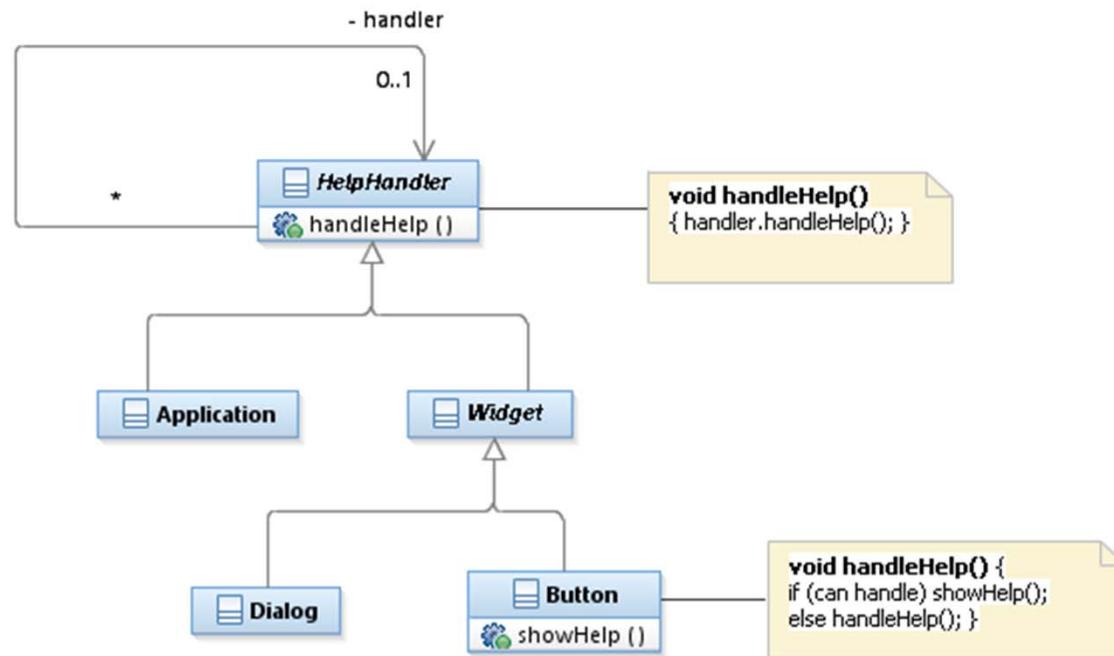
## Chain of Responsibility



Objetos enlazados por una Chain of Responsibility

# Patrones de comportamiento

## Chain of Responsibility



Ejemplo de Chain of Responsibility

# Patrones de comportamiento

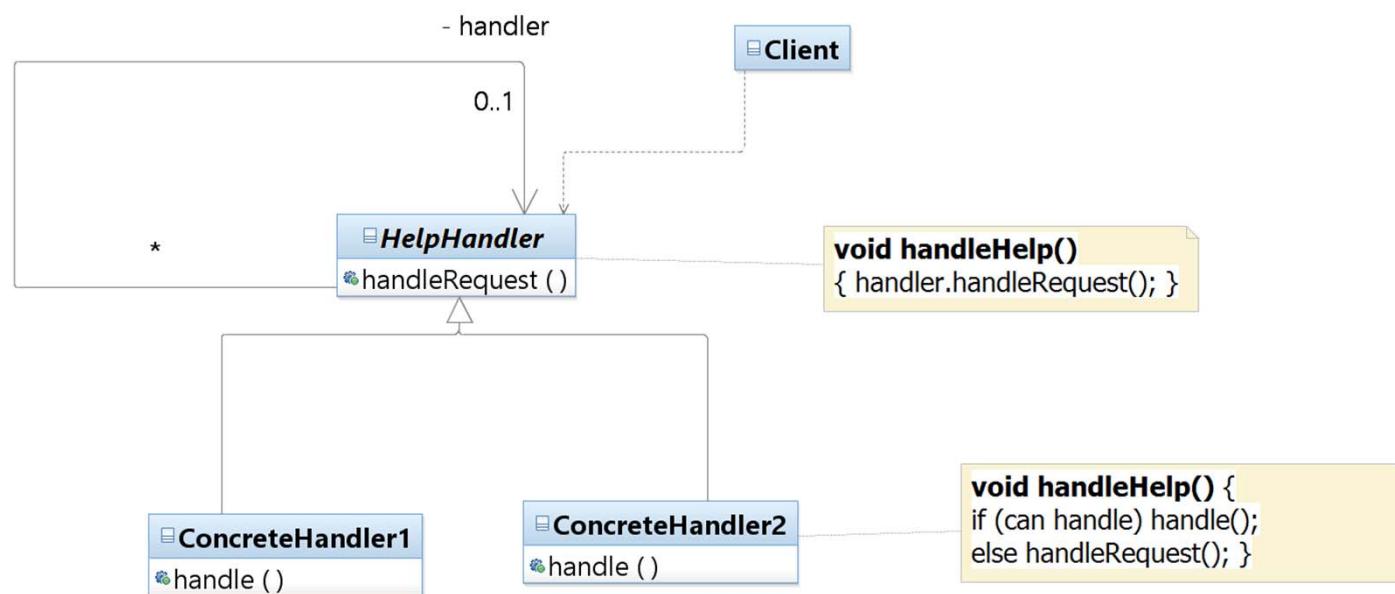
## Chain of Responsibility

- El patrón CoR debe aplicarse cuando
  - Hay más de un objeto que pueda manejar una petición, y el manejador no se conoce a priori, sino que debería determinarse automáticamente.
  - Se quiere enviar una petición a un objeto entre varios sin especificar explícitamente el receptor.
  - El conjunto de objetos que pueden tratar una petición debería ser especificado dinámicamente.

# Patrones de comportamiento

## Chain of Responsibility

- Descripción abstracta



Estructura y comportamiento del patrón Chain of Responsibility

# Patrones de comportamiento

## Chain of Responsibility



Estructura de objetos conectados por  
una Chain of Responsibility

# Patrones de comportamiento

## Chain of Responsibility

- Consecuencias
  - Ventajas
    - Reduce el acoplamiento.
    - Añade flexibilidad para asignar responsabilidades a objetos.
  - Inconvenientes
    - No se garantiza la recepción de la petición.

# Patrones de comportamiento

## Chain of Responsibility

- Código de ejemplo

```
//esta clase juega el papel de manejador y de
//analizador de documentos
public class Analizador {
    Analizador sucesor;
    String extension;

    public Analizador(Analizador m, String e)
    { sucesor= m;
        extension= e      }
```

# Patrones de comportamiento

## Chain of Responsibility

```
public void establecerManejador(Analizador m)
{ sucesor= m; }

public void manejarAnalisis()
{ if (sucesor != null) sucesor.analizar(); }

public abstract Boolean analizar(String documento);

public String obtenerExtension(String fichero)
{ ... }

}
```

# Patrones de comportamiento

## Chain of Responsibility

```
public class AnalizadorDoc extends Analizador {  
    public AnalizadorDoc(Analizador m, String  
extension)  
    { super(m, e); }  
  
    public Boolean analizar(String fichero)  
    { if  
(extension.equals(obtenerExtension(fichero))  
        {  
            //analiza el documento Word  
        }  
        else {super.manejarAnalisis();}  
  
    }  
}
```

# Patrones de comportamiento

## Chain of Responsibility

```
public class AnalizadorRTF extends Analizador {  
    public AnalizadorRTF(Analizador m, String  
extension)  
    { super(m, e); }  
  
    public Boolean analizar(String fichero)  
    { if  
(extension.equals(obtenerExtension(fichero))  
        {  
            //analiza el documento RTF  
        }  
        else {super.manejarAnalisis();}  
    }  
}
```

# Patrones de comportamiento

## Chain of Responsibility

```
public class AnalizadorPDF extends Analizador {  
    public AnalizadorPDF(Analizador m, String  
        extension)  
    { super(m, e); }  
  
    public Boolean analizar(String fichero)  
    { if  
        (extension.equals(obtenerExtension(fichero))  
            {  
                //analiza el documento PDF}  
            }  
        else {super.manejarAnalisis();}  
    }  
}
```

# Patrones de comportamiento

## Chain of Responsibility

```
AnalizadorPDF aPDF= new AnalizadorPDF(null,  
    "pdf");  
AnalizadorRTF aRTF= new AnalizadorRTF(aPDF,  
    "rtf");  
AnalizadorDoc aDoc= new AnalizadorDoc(aRTF,  
    "doc");  
  
aDoc.analizar(fichero);  
//suponemos que la mayoría de archivos son doc
```

# Patrones de comportamiento

## Chain of Responsibility

- Si surgen nuevas posibles extensiones a analizar:

```
public class AnalizadorXML extends Analizador {  
    public AnalizadorXML(Analizador m, String  
        extension)  
    { super(m, e); }  
  
    public Boolean analizar(String fichero)  
    { if (extension.equals(obtenerExtension(fichero))  
        {  
            //analiza el documento XML  
        }  
        else {super.manejarAnalisis();}  
    }  
}
```

# Patrones de comportamiento

## Chain of Responsibility

```
AnalizadorXML aXML= new AnalizadorXML(null,  
    "xml");  
  
AnalizadorPDF aPDF= new AnalizadorPDF(aXML,  
    "pdf");  
  
AnalizadorRTF aRTF= new AnalizadorRTF(aPDF,  
    "rtf");  
  
AnalizadorDoc aDoc= new AnalizadorDoc(aRTF,  
    "doc");  
  
aDoc.analizar(fichero);
```

# Patrones de comportamiento

## Chain of Responsibility

- Nótese, que esto es más flexible que una lógica basada en `if/switch`, ya que en dicha lógica el número de analizadores es fijo, y aquí variable
- También podemos reorganizar la cadena (estática o dinámicamente) en base a la probabilidad de recibir ficheros de un determinado tipo

# Patrones de comportamiento

## Command

- Propósito
  - Encapsula una petición en un objeto, permitiendo así parametrizar a los clientes con diferentes peticiones, hacer cola o llevar un registro de las peticiones, y poder deshacer las operaciones.
- También conocido como
  - Orden.
  - Action (Acción).
  - Transaction (Transacción).

# Patrones de comportamiento

## Command

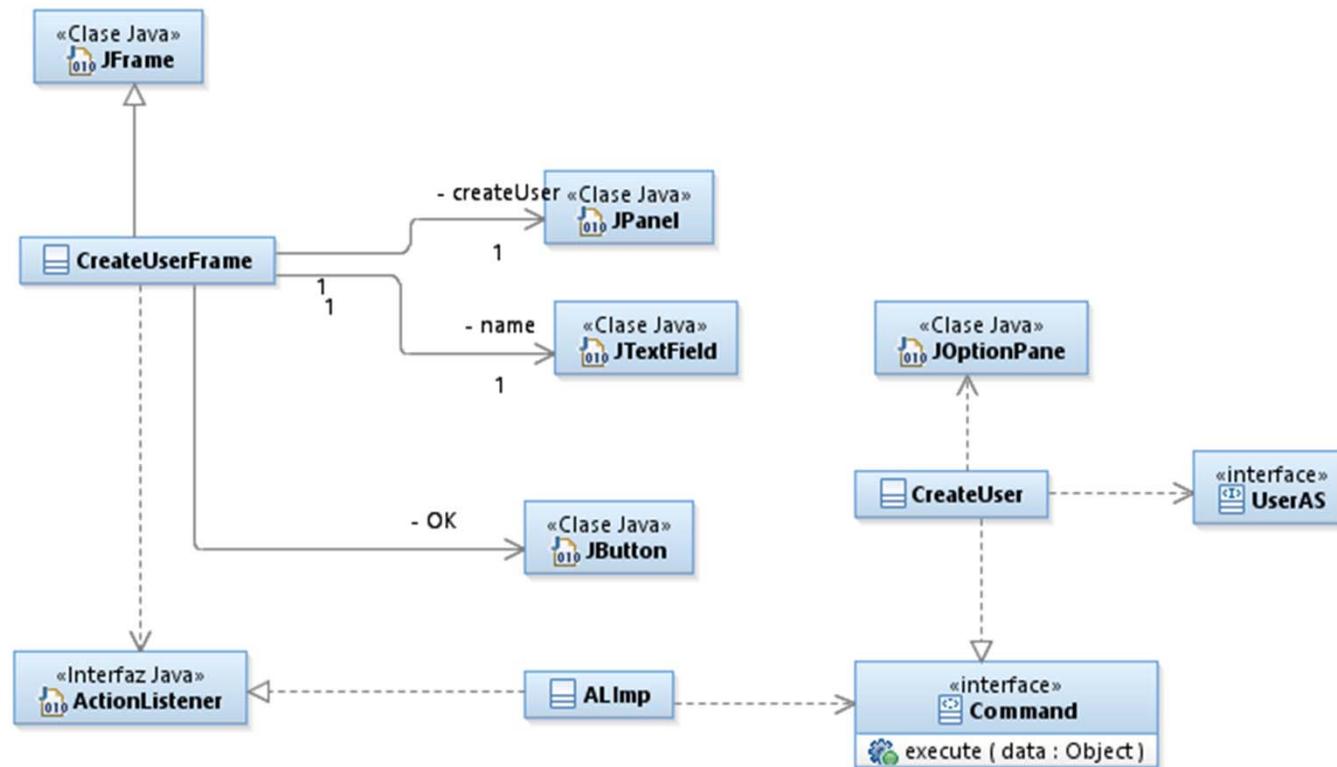
- Motivación
  - A veces es necesario enviar peticiones a objetos sin saber nada acerca de la operación solicitada o quién es el receptor de la petición.
  - Por ejemplo, los elementos visuales de las interfaces de usuario no tienen asociado operaciones.

# Patrones de comportamiento

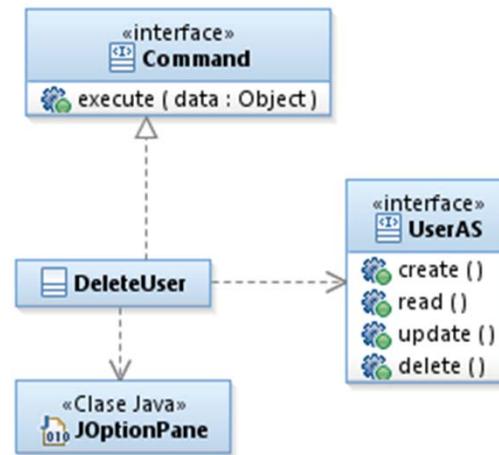
## Command

- Este patrón permite que los objetos de la interfaz hagan peticiones a objetos de la aplicación no especificados, convirtiendo la petición en un objeto, el cual se puede guardar y enviar exactamente igual que cualquier otro objeto.

# Patrones de comportamiento Command

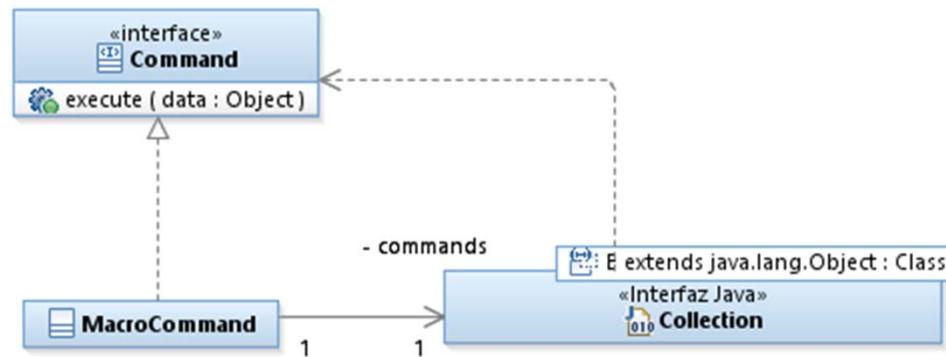


# Patrones de comportamiento Command



## Órdenes concretas

# Patrones de comportamiento Command



## Órdenes concretas

# Patrones de comportamiento

## Command

- El patrón Command debe aplicarse cuando se quiera
  - Parametrizar objetos con una acción a realizar. Los objetos Orden son un sustituto orientado a objetos de las funciones de *callback*.
  - Especificar, poner en cola y ejecutar peticiones en diferentes instantes de tiempo.
  - Permitir deshacer, incorporando esta opción en los objetos Orden.

# Patrones de comportamiento

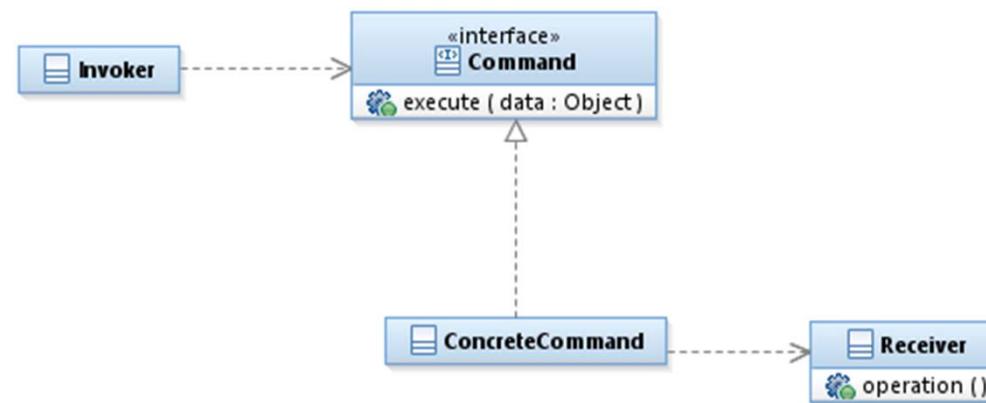
## Command

- Permitir registrar los cambios de manera que se puedan volver a aplicar en caso de una caída del sistema.
- Estructurar un sistema alrededor de operaciones de alto nivel construidas sobre operaciones básicas.

# Patrones de comportamiento

## Command

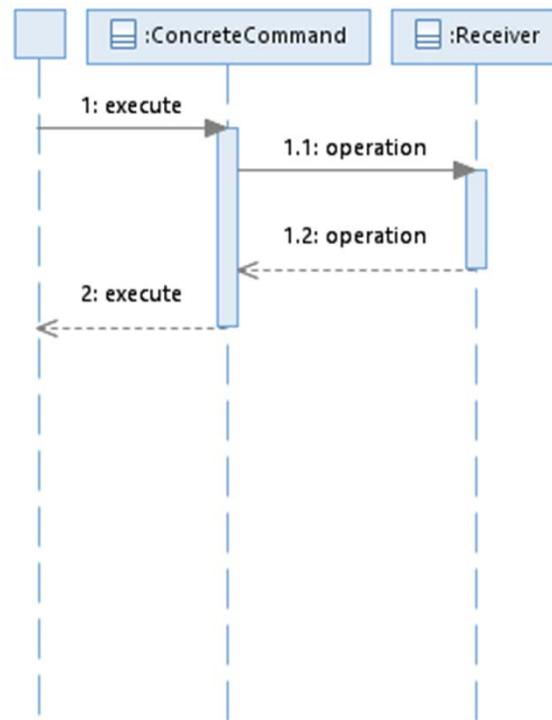
- Descripción abstracta



Estructura del patrón Command

# Patrones de comportamiento

## Command



# Patrones de comportamiento

## Command

- Consecuencias
  - Ventajas
    - Desacopla el objeto que invoca la operación de aquél que sabe cómo realizarla.
    - Las órdenes son objetos de primera clase.
    - Se pueden ensamblar órdenes en una orden compuesta.
    - Es fácil añadir nuevas órdenes, ya que no hay que cambiar las clases existentes.

# Patrones de comportamiento

## Command

- Código de ejemplo

```
public interface Orden {  
    public void ejecutar();  
};
```

# Patrones de comportamiento

## Command

```
public class OrdenAbrir implements Orden {  
    Aplicación aplicación;  
    String respuesta;  
  
    public OrdenAbrir (Aplicación a)  
    { Aplicación= a; }  
  
    protected String preguntarUsuario() {...}
```

# Patrones de comportamiento

## Command

```
public void ejecutar()
{ String nombre= preguntarUsuario();
  if (nombre != 0)
    { Documento doc= new Documento(nombre);
      aplicacion.anadir(doc);
      doc.abrir();
    }
}
```

# Patrones de comportamiento

## Command

```
public class OrdenPegar implements Orden {  
    Documento doc;  
  
    public OrdenPegar(Documento d)  
    { doc= d;  
  
        public void ejecutar()  
        { documento.pegar( ); }  
  
    } ;
```

# Patrones de comportamiento

## Command

- En los ejemplos anteriores no se utilizaba un controlador
- Las Action de Struts 1.x\*, son un claro caso de patrón Command con controlador

\*<http://struts.apache.org/>

# Patrones de comportamiento

## Interpreter

- Propósito
  - Dado un lenguaje, define una representación de su gramática junto con un intérprete que usa dicha representación para interpretar sentencias del lenguaje.
- También conocido como
  - Intérprete.

# Patrones de comportamiento

## Interpreter

- Motivación
  - Si hay un tipo de problema que ocurre con cierta frecuencia, pude valer la pena expresar las apariciones de ese problema como instrucciones de un lenguaje simple. A continuación puede construirse un intérprete que resuelva el problema interpretando dichas instrucciones.

# Patrones de comportamiento

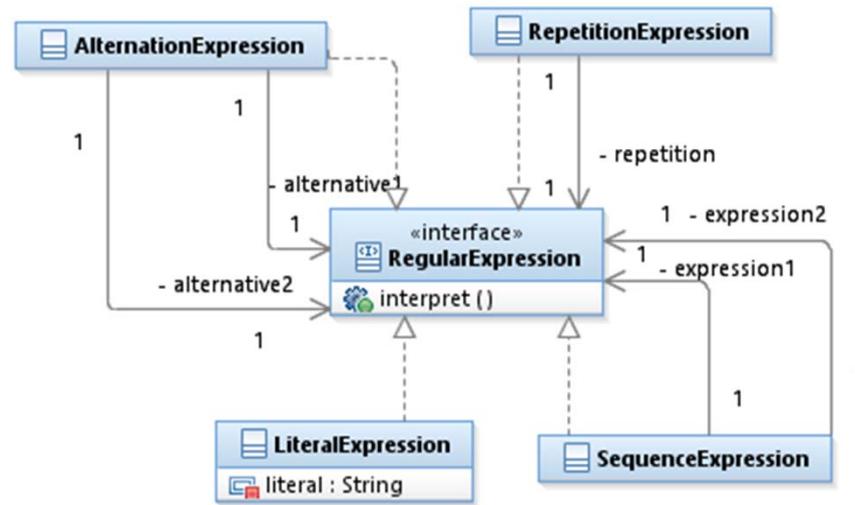
## Interpreter

```
expresion ::= literal | alternativa | secuencia |
            repeticion
alternativa ::= expresion ' | ' expresion
secuencia ::= expresion '&' expresion
repeticion ::= expresion '*'
literal ::= 'a' | 'b' | ... { 'a' | 'b' | ... } *
```

### Ejemplo de lenguaje

# Patrones de comportamiento

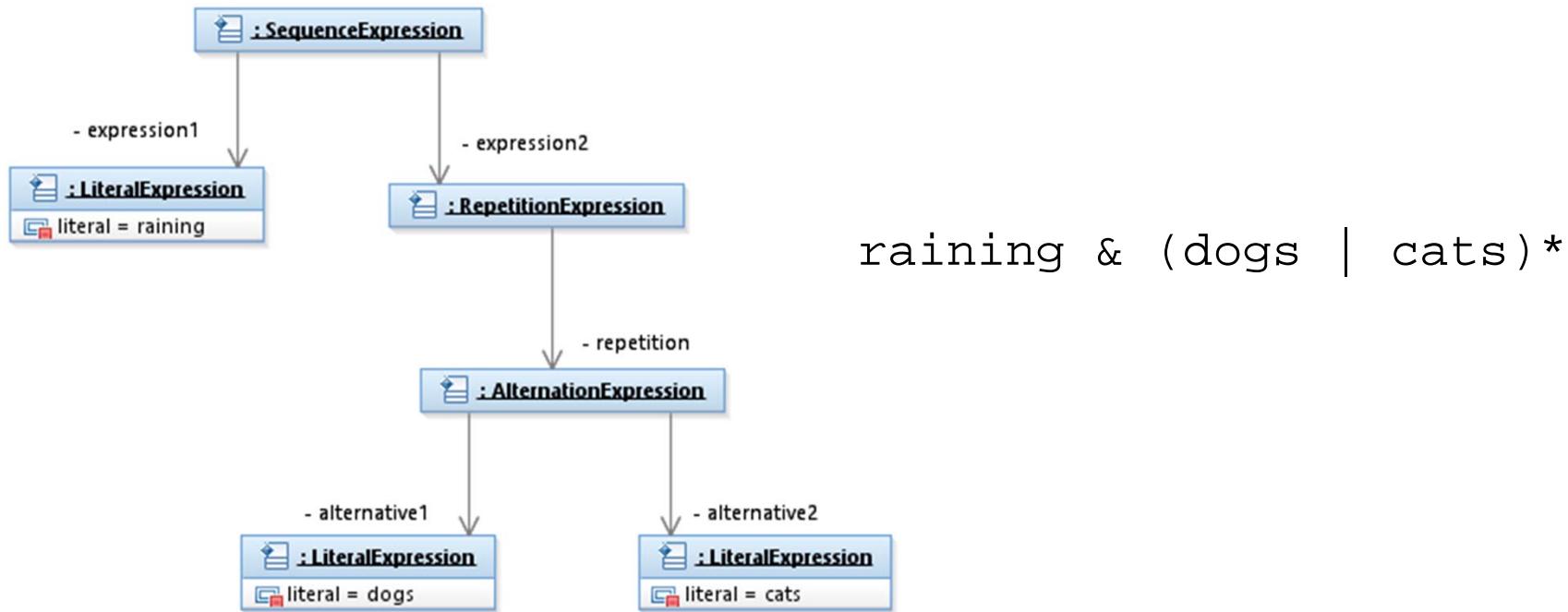
## Interpreter



Ejemplo estructura patrón Interpreter

# Patrones de comportamiento

## Interpreter



Árbol sintaxis abstracta para una expresión

# Patrones de comportamiento

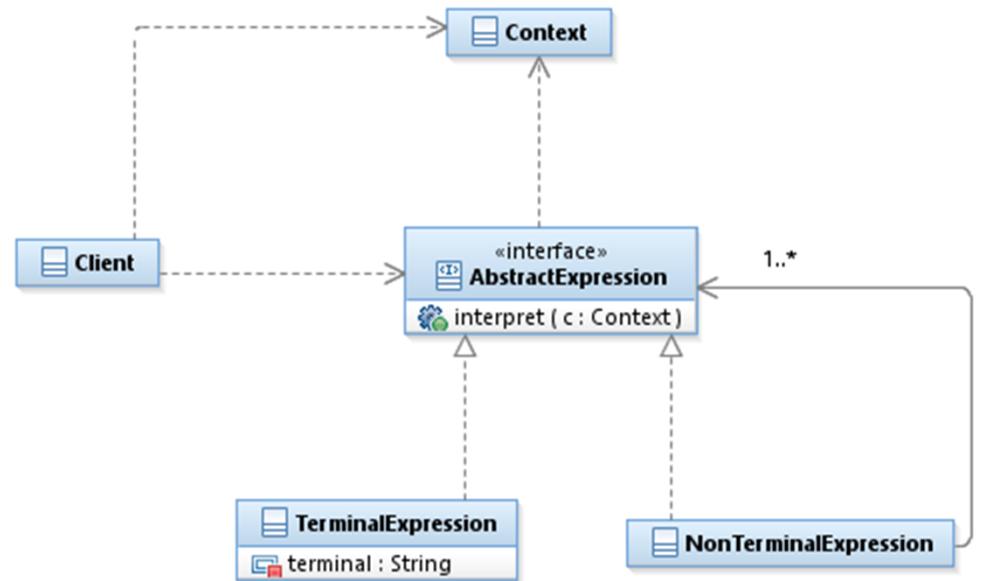
## Interpreter

- El patrón Interpreter debe aplicarse cuando
  - Hay un lenguaje que interpretar y se pueden representar las sentencias del lenguaje como árboles sintácticos abstractos. Funciona mejor cuando:
    - La gramática es simple.
    - La eficiencia no es una preocupación crítica.

# Patrones de comportamiento

## Interpreter

- Descripción abstracta



Estructura del patrón Interpreter

# Patrones de comportamiento

## Interpreter

- Consecuencias
  - Ventajas
    - Es fácil cambiar y ampliar la gramática.
    - Es fácil implementar la gramática.
    - Es fácil añadir nuevos modos de interpretar expresiones.
  - Inconvenientes
    - Las gramáticas complejas son difíciles de mantener.

# Patrones de comportamiento

## Interpreter

- Código de ejemplo

```
ExpBooleana ::= ExpVariable | Constante | ExpOr |
               ExpAnd | ExpNot | '(' ExpBooleana ')'
ExpAnd ::= ExpBooleana 'and' ExpBooleana
ExpOr ::= ExpBooleana 'or' ExpBooleana
ExpNot ::= 'not' ExpBooleana
Constante ::= 'true' | 'false'
ExpVariable ::= 'A' | 'B' | ... | 'Z'
```

# Patrones de comportamiento

## Interpreter

```
public abstract class ExpBooleana {  
    public abstract Boolean evaluar(Contexto c);  
    public abstract ExpBooleana sustituir(String c,  
                                         ExpBooleana e);  
    public abstract ExpBooleana copiar();  
};  
  
public class Contexto {  
    public boolean buscar(char variable) {...}  
    public void asignar(ExpVariable var, Boolean valor)  
    {...}  
};
```

# Patrones de comportamiento

## Interpreter

```
public class ExpVariable extends ExpBooleana {  
    char nombre;  
  
    public ExpVariable (char n)  
    { nombre= n; }  
  
    public Boolean evaluar(Contexto c)  
    { return c.buscar(nombre); }  
  
    public ExpBooleana copiar()  
    { return new ExpVariable(nombre); }
```

# Patrones de comportamiento

## Interpreter

```
public ExpBooleana sustituir(String n,  
                           ExpBooleana exp)  
{ if (nombre.equals(n)) return exp.copiar();  
    else return new ExpVariable(nombre);  
}  
  
};
```

# Patrones de comportamiento

## Interpreter

```
public class ExpAnd extends ExpBooleana {  
    ExpBooleana operando1;  
    ExpBooleana operando2;  
  
    public ExpAnd(ExpBooleana o1, ExpBooleana o2)  
    { operando1= o1; operando2= o2; }  
  
    public Boolean evaluar(Contexto c)  
    { return operando1.evaluar(c) &&  
        operando2.evaluar(c); }
```

# Patrones de comportamiento

## Interpreter

```
public ExpBooleana copiar()
{ return new ExpAnd(operando1.copiar(),
operando2.copiar) ; }
```

```
public ExpBooleana sustituir (char n,
ExpBooleana exp)
{ return new ExpAnd(operando1.sustituir(n,
exp), operando2.sustituir(n, exp));
}
```

# Patrones de comportamiento

## Interpreter

(true and x) or (y and (not x))

```
ExpBooleana expresion;
Contexto contexto;
ExpVariable x= new ExpVariable("X");
ExpVariable y= new ExpVariable("Y");
expresion=
    new ExpOr(new ExpAnd(new Constante(true), x),
              new ExpAnd(y, new ExpNot(x)));
contexto.asignar(x, false);
contexto.asignar(y, true);
Boolean resultado= expresion.evaluar(contexto);
```

# Patrones de comportamiento

## Interpreter

```
ExpVariable z= new ExpVariable("Z");
```

```
ExpNot not_Z= new ExpNot(z);
```

```
ExpBooleana sustitucion= expresion.sustituir("Y",  
not_z);
```

```
contexto.asignar(z, true);
```

```
resultado= sustitucion.evaluar(context);
```

# Patrones de comportamiento

## Iterator

- Propósito
  - Proporciona un modo de acceder secuencialmente a los elementos de un objeto agregado sin exponer su representación interna.
- También conocido como
  - Iterador.
  - Cursor.

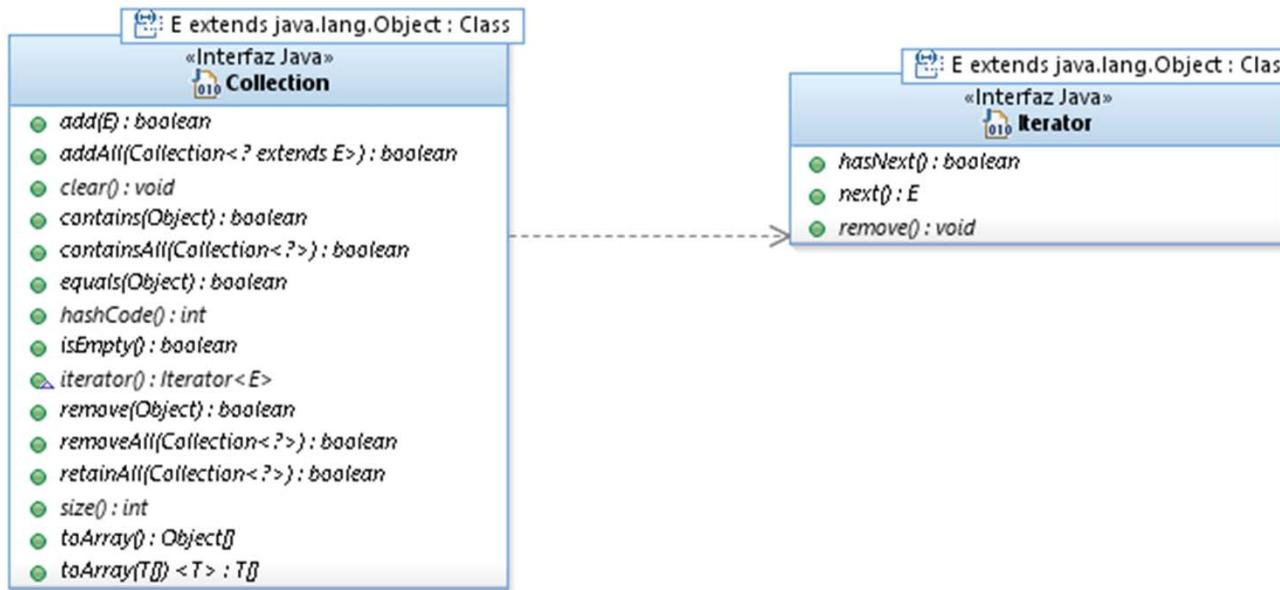
# Patrones de comportamiento

## Iterator

- Motivación
  - Un objeto agregado (e.g. una lista) debería darnos una forma de acceder a sus elementos sin exponer su estructura interna.
  - Además es posible que deseemos hacer diversas cosas con los componentes de la lista o que necesitemos hacer más de un recorrido simultáneamente.

# Patrones de comportamiento

## Iterator



Ejemplo patrón Iterator

# Patrones de comportamiento

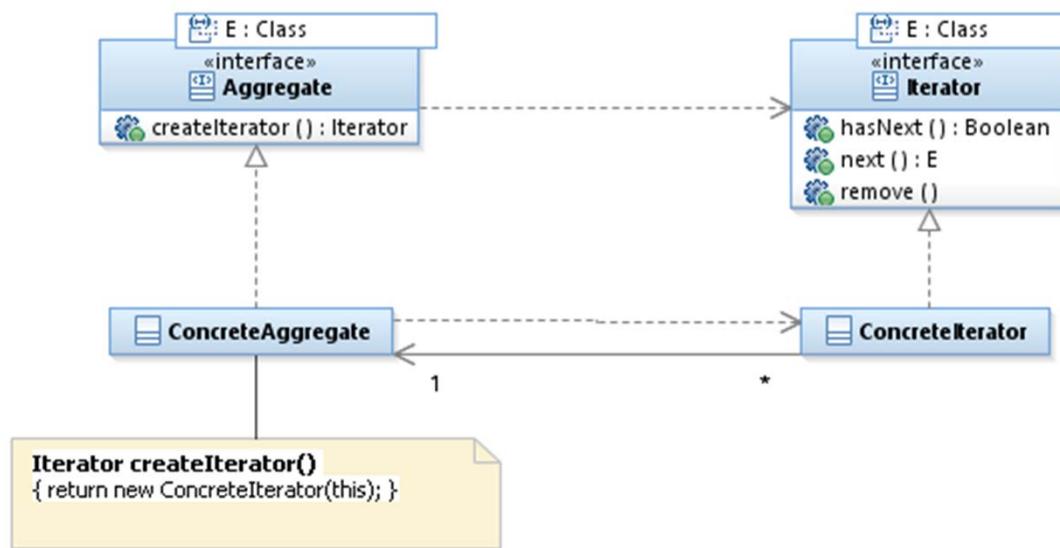
## Iterator

- El patrón Iterator debe aplicarse para
  - Acceder al contenido de un objeto agregado sin exponer su representación interna.
  - Permitir varios recorridos sobre objetos agregados.
  - Proporcionar una interfaz uniforme para recorrer diferentes estructuras agregadas.

# Patrones de comportamiento

## Iterator

- Descripción abstracta



Estructura y comportamiento del patrón Iterator

# Patrones de comportamiento

## Iterator

- Consecuencias
  - Ventajas
    - Permite variaciones en el recorrido de un agregado.
    - Simplifica la interfaz del objeto agregado.
    - Se puede hacer más de un recorrido a la vez sobre un agregado.

# Patrones de comportamiento

## Iterator

- Código de ejemplo

```
public interface IED {  
    public int insertar(Comparable objetoP);  
    public int eliminar(Object idObjeto);  
    public Comparable obtenerPorId(Object idObjeto);  
    protected Comparable obtenerPorPos(int pos);  
    public int obtenerNumEletos();  
    public IIterador crearIterador();  
};
```

# Patrones de comportamiento

## Iterator

```
public interface IIterador {  
    public void primero();  
    public void siguiente();  
    public boolean haTerminado();  
    public Object elementoActual();  
};
```

# Patrones de comportamiento

## Iterator

```
public class IteradorLista implements IIterador {  
    IED lista;  
    int actual;  
  
    public IteradorLista(IED listaP)  
    { lista= listaP; actual= 0; }  
  
    public primero()  
    { actual= 0; }  
  
    public siguiente()  
    { actual++; }
```

# Patrones de comportamiento

## Iterator

```
public boolean haTerminado( )
{ return actual >= lista.obtenerNumEletos( ); }

public Object elementoActual()
{ if (haTerminado( )) return null;
  else return lista.obtenerPorPos(actual);
}
};
```

# Patrones de comportamiento

## Iterator

```
IED lista = new Lista();
IIterador i= lista.crearIterador();
total= 0;
//uso de iterador
for (i.primer() ; !i.haTerminado(); i.siguiente())
{ empleado= (IEmppleado)i.elementoActual();
  total+= empleado.nomina();
}
//si obtenerPorPos() no fuera protegida
for (i=0; i<lista.numEletos(); i++)
{ empleado= (IEmppleado)lista.obtenerPorPos(i);
  total+= empleado.nomina();
}
```

# Patrones de comportamiento

## Iterator

```
//uso de iterador al estilo Java
while (!i.haTerminado( ))
{ i.siguiente();
  empleado= (IEmppleado)i.elementoActual();
  total+= empleado.nomina();
}
```

# Patrones de comportamiento

## Mediator

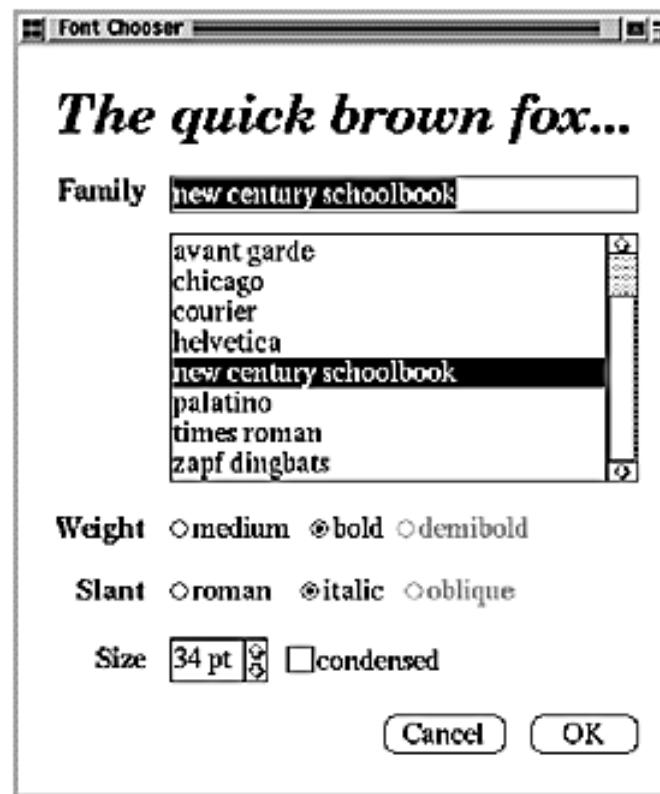
- Propósito
  - Define un objeto que encapsula cómo interactúan una serie de objetos. Promueve un bajo acoplamiento al evitar que los objetos se refieran unos a otros explícitamente, y permite variar la interacción entre ellos de forma independiente.
- También conocido como
  - Mediador.

# Patrones de comportamiento

## Mediator

- Motivación
  - Aunque dividir un sistema en muchos objetos suele mejorar la reutilización, la proliferación de interconexiones tiende a reducir ésta de nuevo.
  - En las interfaces gráficas de usuario suelen aparecer estas interconexiones entre diversos elementos visuales.

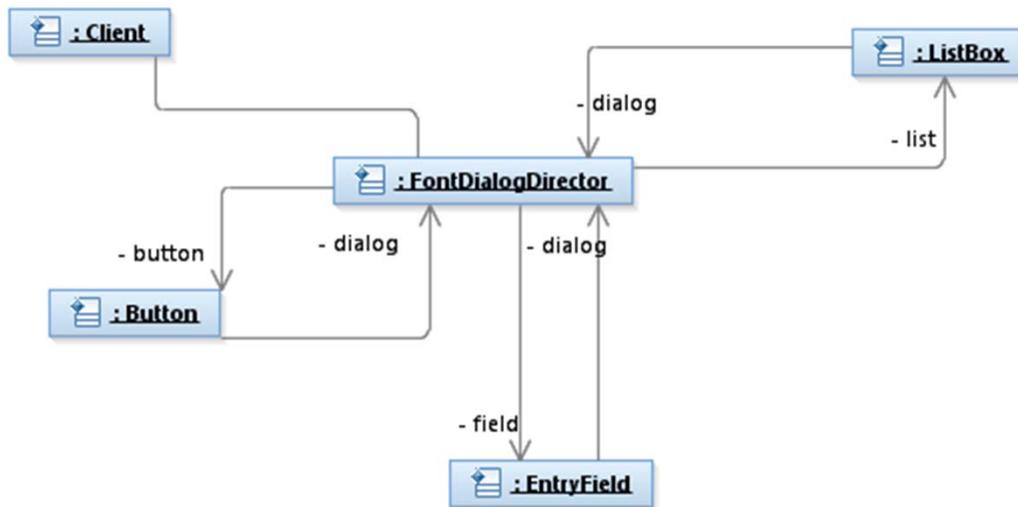
# Patrones de comportamiento Mediator



Relaciones entre elementos

# Patrones de comportamiento

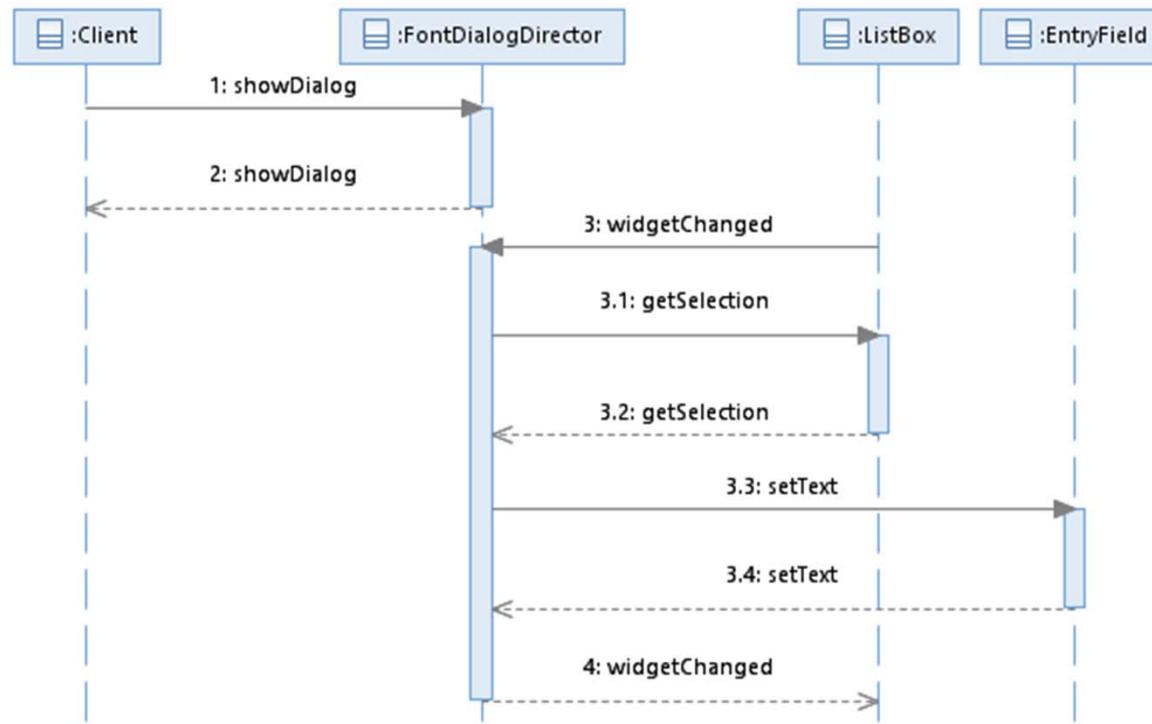
## Mediator



Un mediador entre elementos

# Patrones de comportamiento

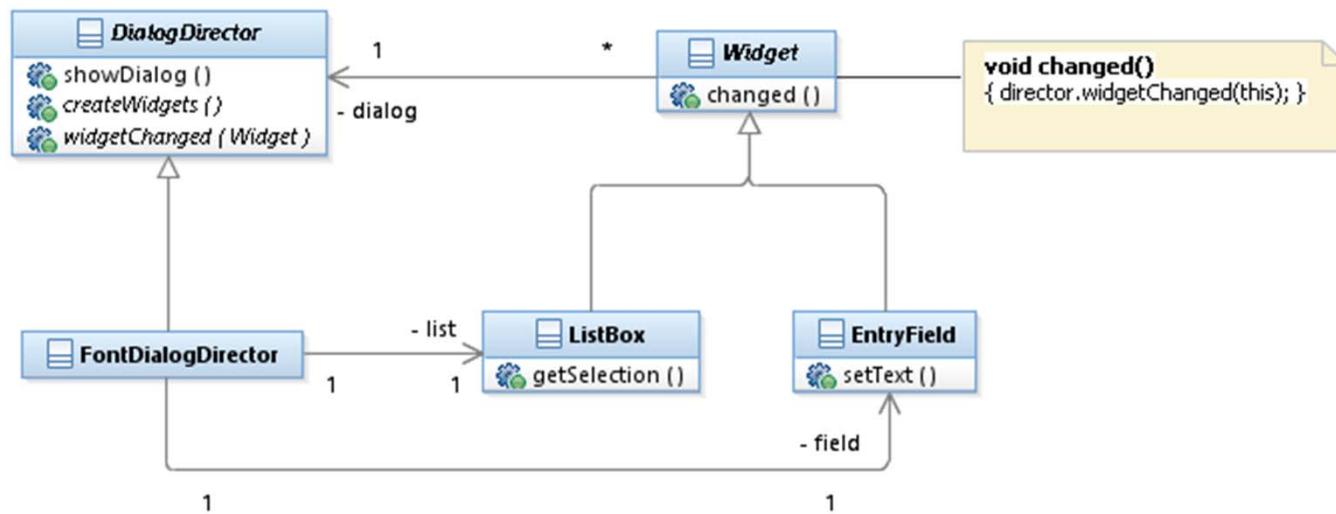
## Mediator



Ejemplo interacción entre el mediador y sus colegas. Nótese que los controladores de eventos se suponen unidos a los elementos visuales

# Patrones de comportamiento

## Mediator



Ejemplo patrón Mediator

# Patrones de comportamiento

## Mediator

- El patrón Mediator debe aplicarse cuando
  - Un conjunto de objetos se comunican de forma bien definida, pero compleja. Las interdependencias resultantes no están estructuradas y son difíciles de comprender.
  - Es difícil reutilizar un objeto, ya que éste se refiere a otros muchos objetos con los que se comunica.

# Patrones de comportamiento

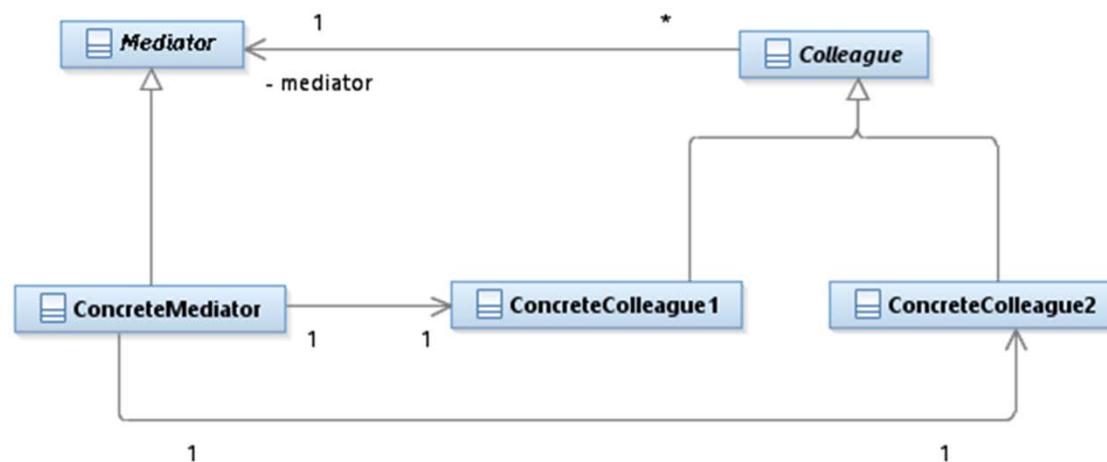
## Mediator

- Un comportamiento que está distribuido entre varias clases debería poder ser adaptado sin necesidad de una gran cantidad de subclases.

# Patrones de comportamiento

## Mediator

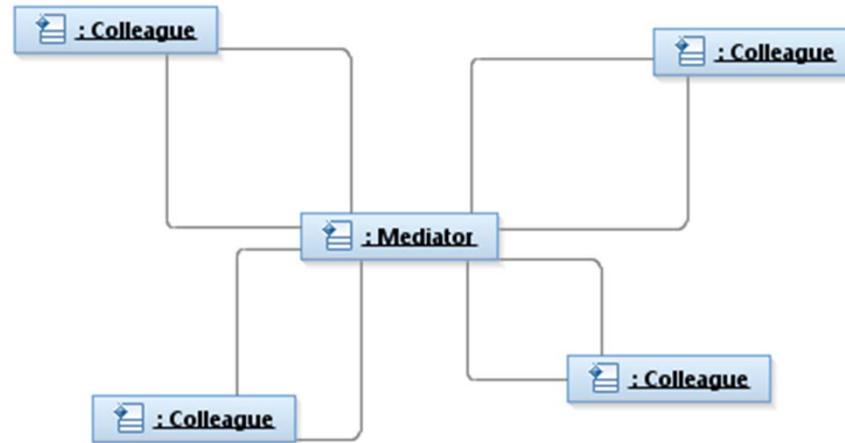
- Descripción abstracta



Estructura del patrón Mediator

# Patrones de comportamiento

## Mediator



Objetos relacionados a través de un mediador

# Patrones de comportamiento

## Mediator

- Consecuencias
  - Ventajas
    - Reduce la herencia.
    - Desacopla a los colegas.
    - Simplifica los protocolos de los objetos.
    - Abstactea como cooperan los objetos.
  - Inconvenientes
    - Centraliza el control.

# Patrones de comportamiento

## Mediator

- Código de ejemplo

```
public interface DirectorDialogo {  
    public void mostrarDialogo();  
    public void utilModificado(Util u);  
    protected void crearUtiles();  
};
```

# Patrones de comportamiento

## Mediator

```
public class Util {  
    DirectorDialogo director;  
  
    public Util (DirectorDialogo d)  
    { director= d; }  
  
    public void modificado()  
    { director.utilModificado(this); }  
  
    public void manejarRaton(EventoRaton e) { . . . }  
    . . . . .  
};
```

# Patrones de comportamiento

## Mediator

```
public class ListaDesplegable extends Util {  
  
    public ListaDesplegable(DirectorDialogo d)  
    { super (d); }  
    public String obtenerSeleccion() {...}  
    public void establecerLista(Lista l) {...}  
    public void manejarRaton(EventoRaton e) {...}  
    ....  
};
```

# Patrones de comportamiento

## Mediator

```
public class CampoDeEntrada extends Util {  
  
    public ListaDesplegable(DirectorDialogo d)  
    { super (d); }  
    public String obtenerSeleccion() {...}  
    public void establecerLista(Lista l) {...}  
    public void manejarRaton(EventoRaton e) {...}  
    ....  
};
```

# Patrones de comportamiento

## Mediator

```
public class Boton extends Util {  
  
    public ListaDesplegable(DirectorDialogo d)  
    { super (d); }  
    public String obtenerSeleccion() {...}  
    public void establecerLista(Lista l) {...}  
    public void manejarRaton(EventoRaton e) {...}  
    ....  
};
```

# Patrones de comportamiento

## Mediator

```
public class DirectorDialogoFuente implements  
DirectorDialogo {  
    Boton aceptar;  
    Boton cancelar;  
    ListaDesplegable fuenteLista;  
    CampoDeEntrada nombreFuente;  
  
    public DirectorDialogoFuente() { ... }
```

# Patrones de comportamiento

## Mediator

```
public void crearUtiles()
{ aceptar= new Boton(this);
cancelar= new Boton(this);
fuenteLista= new ListaDesplegable(this);
nombreFuente= new CampoDeEntrada(this);

//rellenar la lista con los nombres de fuentes

//ensambla los útiles en el dialogo
}
```

# Patrones de comportamiento

## Mediator

```
public void utilModificado(Util modificado)
{
    if (modificado == fuenteLista)
        { nombreFuente.establecerTexto(
                fuenteLista.obtenerSeleccion()); }
    else if (modificado == aceptar)
        { //aplicar cambio de fuente y cerrar }
    else if (modificado == cancelar)
        { //cerrar el diálogo }
}
};
```

# Patrones de comportamiento

## Memento

- Propósito
  - Representa y externaliza el estado interno de un objeto sin violar la encapsulación, de forma que éste pueda volver a dicho estado más tarde.
- También conocido como
  - Recuerdo.
  - Token.

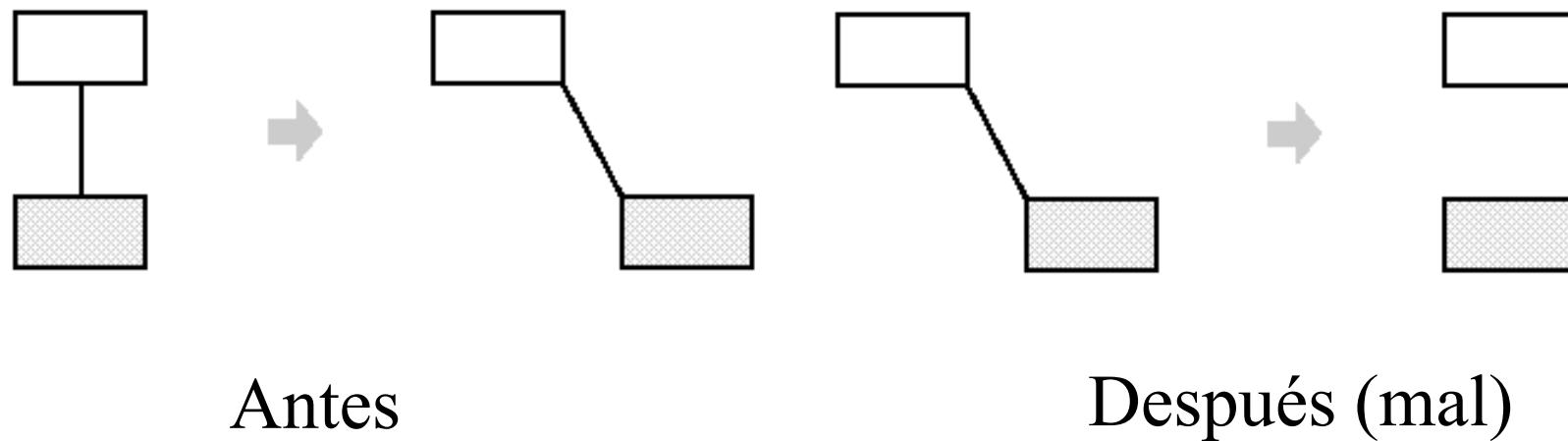
# Patrones de comportamiento

## Memento

- Motivación
  - A veces es necesario guardar el estado interno de un objeto (e.g. deshacer).
  - Debe guardarse información del estado en algún sitio para que los objetos puedan volver a su estado anterior.
  - Sea como fuere, el estado está encapsulado.

# Patrones de comportamiento

## Memento



# Patrones de comportamiento

## Memento

- El patrón Memento debe aplicarse cuando
  - Hay que guardar una instantánea del estado de un objeto (o de parte de éste) para que pueda volver posteriormente a ese estado, y
  - Una interfaz directa para obtener el estado podría exponer detalles de implementación y romper la encapsulación del objeto.

# Patrones de comportamiento

## Memento

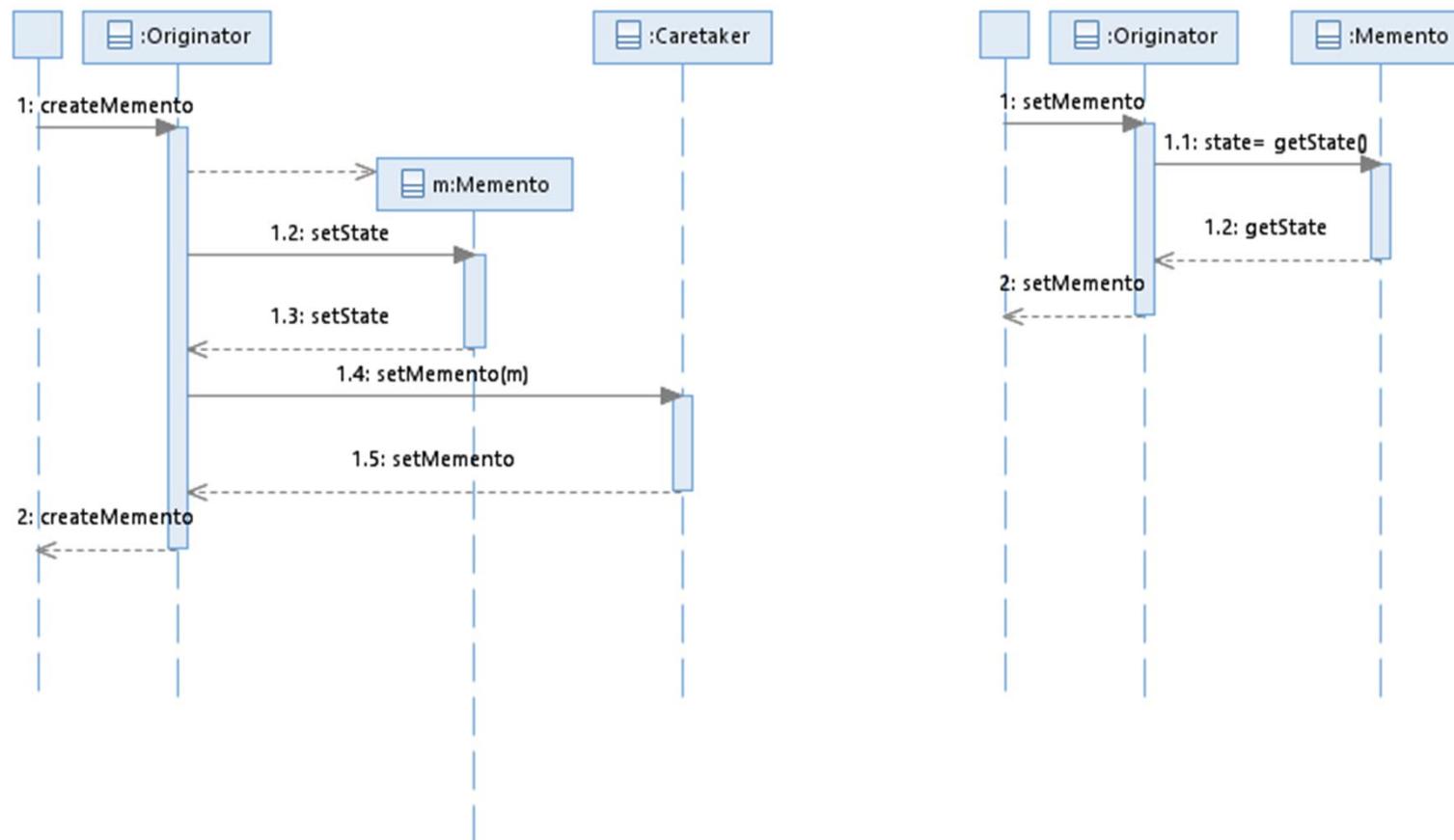
- Descripción abstracta



Estructura del patrón Memento

# Patrones de comportamiento

## Memento



# Patrones de comportamiento

## Memento

- Consecuencias
  - Ventajas
    - Preservación de los límites de la encapsulación.
    - Simplifica al Creador al delegar en el Memento.
  - Inconvenientes
    - El uso de mementos puede ser costoso.
    - Definición de interfaces reducidas y amplias.
    - Costes ocultos en el cuidado de mementos.

# Patrones de comportamiento

## Memento

- Código de ejemplo

```
package ResolventeDeRestricciones;  
public class ResolventeDeRestricciones {  
    //estado no trivial y operaciones para hacer  
    //cumplir la semántica de las conexiones  
  
    void anadirRestriccion (Grafico principioConexion,  
    Grafico finConexion) {...}  
  
    void eliminarRestriccion (Grafico principioConexion,  
    Grafico finConexion) {...}
```

# Patrones de comportamiento

## Memento

```
MemementoDelResolventeDeRestricciones  
crearMemento( ) { . . . }  
  
void EstablecerMemento  
(MemementoDelResolventeDeRestricciones m) { . . . }  
  
} ;
```

# Patrones de comportamiento

## Memento

```
package ResolventeDeRestricciones;

private class MementoDelResolventeDeRestricciones {
    //estado del ResolventeDeRestricciones;
    //funciones para acceder a dicho estado;
}
```

# Patrones de comportamiento

## Memento

```
public interface Grafico { ... };

public class OrdenMover {

    Punto incremento;
    Grafico destino;
    ResolventeDeRestricciones resolvente;
    MementoDelResolventeDeRestricciones estado;

    public OrdenMover(Grafico d, Punto i,
                      ResolventeDeRestricciones r)
    { destino= d; incremento= i; resolvente= r; }
```

# Patrones de comportamiento

## Memento

```
public ejecutar()
{ estado= resolvente.crearMemento();
  destino.mover(incremento);
  resolvente.resolver();
}
public void deshacer()
{ destino.mover(-incremento);
  resolvente.establecerMemento(estado);
  resolvente.resolver();
}
};
```

# Patrones de comportamiento

## Observer

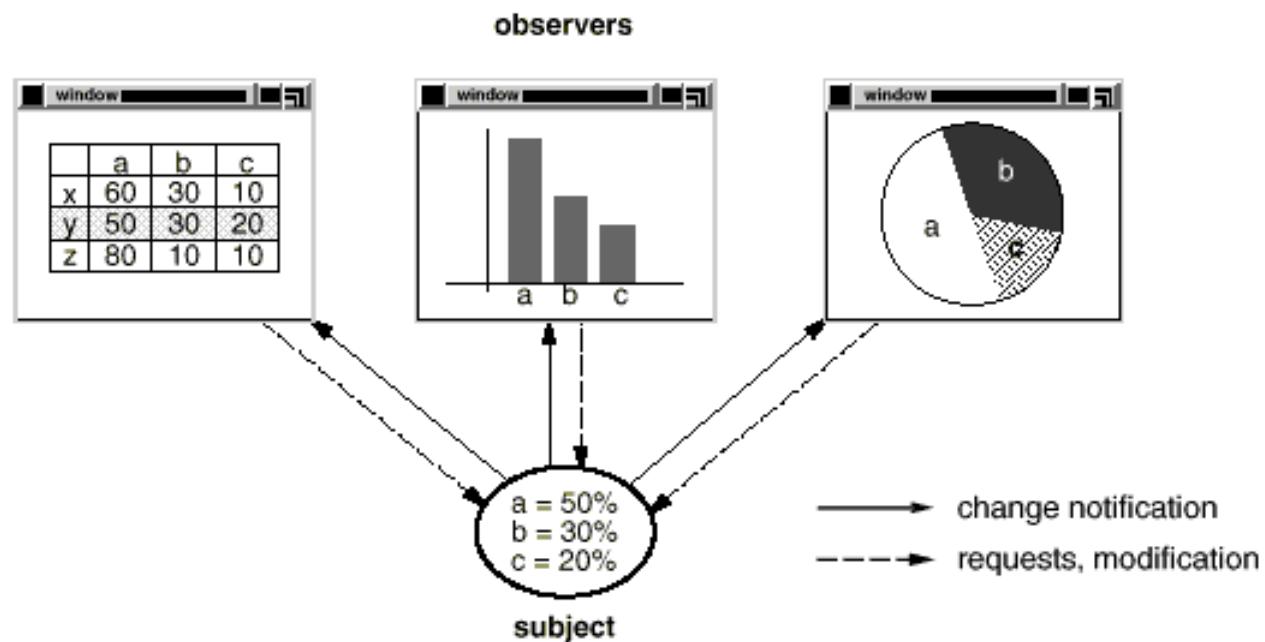
- Propósito
  - Define una dependencia de uno a muchos entre objetos, de forma que cuando un objeto cambie de estado se notifique y se actualicen automáticamente todos los objetos que dependen de él.
- También conocido como
  - Observador.
  - Dependents (dependientes).
  - Publish-Suscribe (publicar-suscribir).

# Patrones de comportamiento

## Observer

- Motivación
  - Si dividimos un sistema en una colección de clases cooperantes debemos mantener la consistencia entre estados relacionados.
  - Esta consistencia no debe lograrse pagando un fuerte acoplamiento.
  - Por ejemplo, en las interfaces de usuario.

# Patrones de comportamiento Observer



Interfaces de usuario como observers

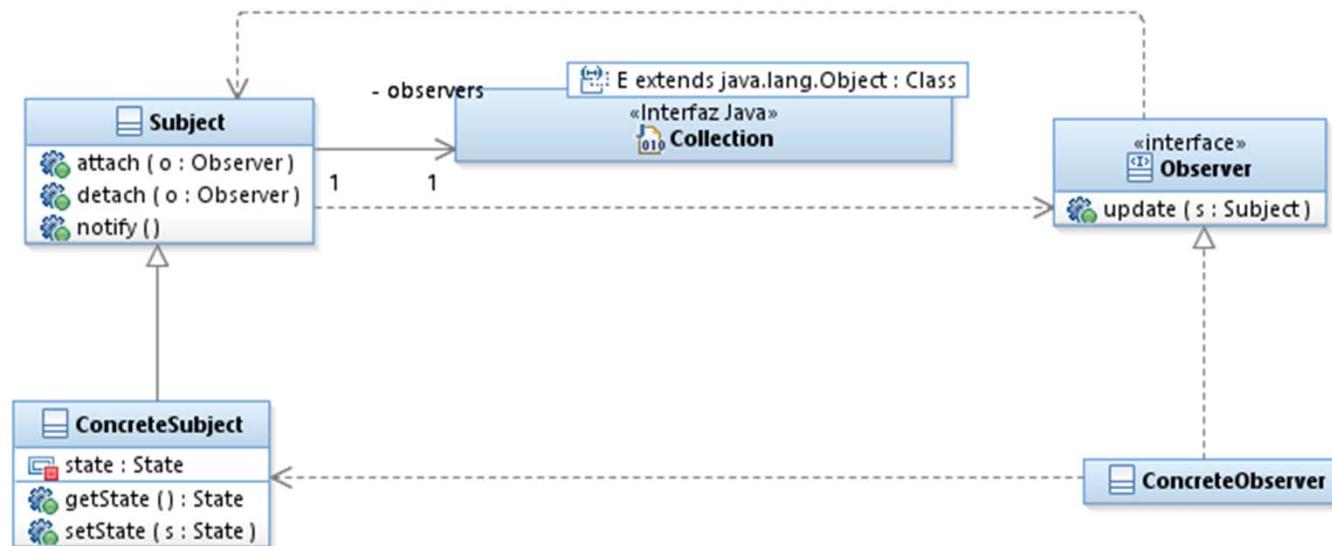
# Patrones de comportamiento

## Observer

- El patrón Observer debe aplicarse cuando
  - Una abstracción tiene dos aspectos y uno depende del otro.
  - Cuando un cambio en un objeto requiere cambiar otros, y no sabemos cuántos objetos necesitan cambiarse.
  - Cuando un objeto debería ser capaz de notificar a otros sin hacer suposiciones sobre quiénes son dichos objetos.

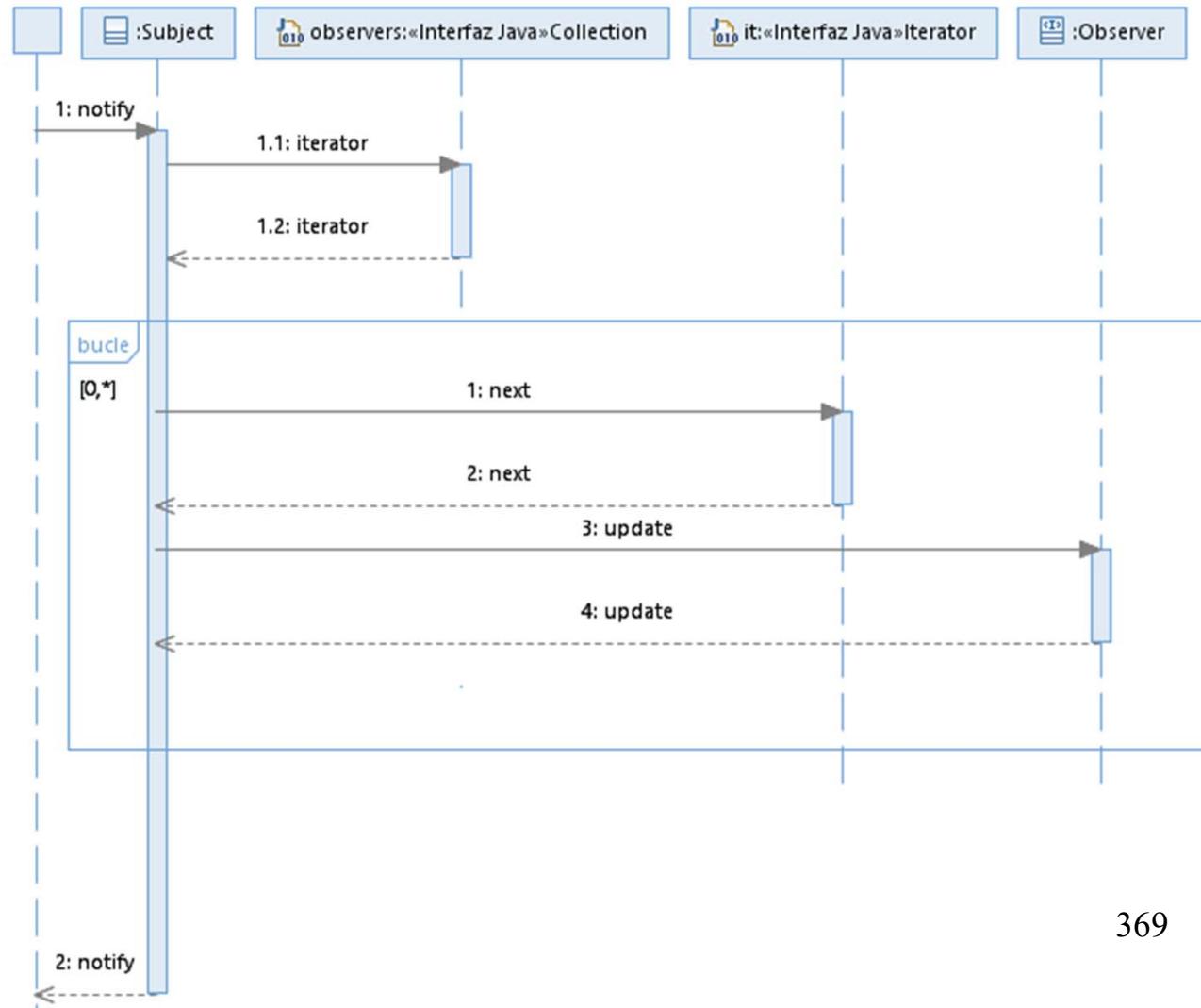
# Patrones de comportamiento Observer

- Descripción abstracta



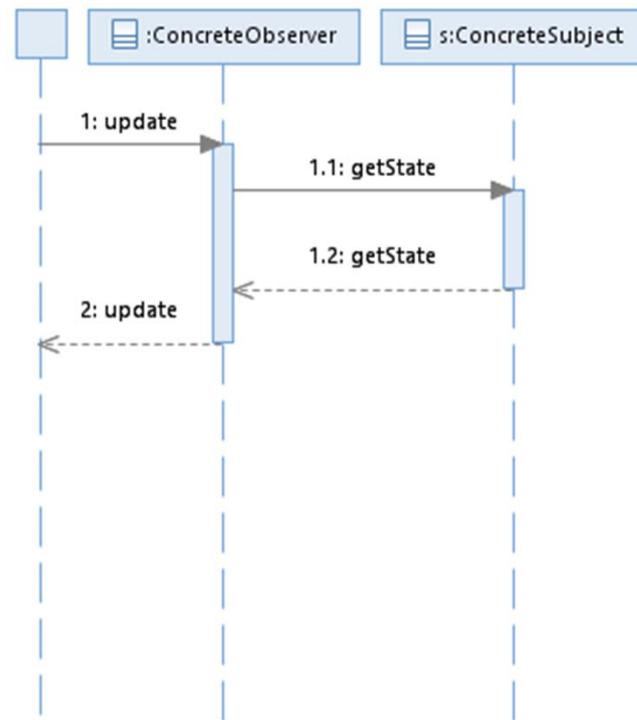
Estructura del patrón Observer

# Patrones de comportamiento Observer



Interacción  
en Observer

# Patrones de comportamiento Observer



# Patrones de comportamiento

## Observer

- Consecuencias
  - Ventajas
    - Permite modificar objetos y observadores de manera independiente.
    - Acoplamiento abstracto entre sujeto y observador.
    - Capacidad de comunicación mediante difusión.
  - Inconvenientes
    - Actualizaciones inesperadas.
    - Protocolo de actualización simple.

# Patrones de comportamiento

## Observer

- Código de ejemplo

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

# Patrones de comportamiento

## Observer

```
public class Observable {  
    public void addObserver(Observer o) {...}  
    protected void clearChanged() {...}  
    public int countObservers() {...}  
    public void deleteObserver(Observer o) {...}  
    public void deleteObservers() {...}  
    public boolean hasChanged() {...}  
    public void notifyObservers() {...}  
    public void notifyObservers(Object arg) {...}  
    public protected void setChanged() {...}  
    ...  
};
```

# Patrones de comportamiento

## Observer

```
//versión naif de un controlador
class Controlador implements ActionListener{
    Modelo modelo;

    public Controlador(Modelo modeloP)
    { modelo= modeloP; }

    public void actionPerformed (ActionEvent e)
    { modelo.sumar(); }

}
```

# Patrones de comportamiento

## Observer

```
class Modelo extends Observable {  
    int valor;  
  
    Modelo( )  
    { valor= 0; }  
  
    void sumar( )  
    { valor++;  
        notifyObservers(); //notify le pasa el objeto  
    }  
    int obtenerValor( )  
    { return valor; } };
```

# Patrones de comportamiento

## Observer

```
class Vista extends JFrame implements Observer {  
    JTextField valor;  
    JButton sumar;  
  
    public Vista(Modelo modelo) {  
        // crea e inicializa sus elementos  
        ActionListener controlador= new Controlador(modelo);  
        sumar.addActionListener(controlador);  
        // termina de configurarse  
    }  
}
```

# Patrones de comportamiento

## Observer

```
public void update (Observable o, Object arg)
{
    Modelo modelo= (Modelo) o;
    Integer i= new Integer(modelo.obtenerValor());
    valor.setText(i.toString());
}

public void activar()
{ setVisible(true); }

};
```

# Patrones de comportamiento Observer

```
public class MVC {  
  
    public static void main (String args[ ]) {  
        Modelo modelo= new Modelo();  
        Vista vista= new Vista(modelo);  
        modelo.addObserver(vista);  
        vista.activar( );  
    }  
};
```

# Patrones de comportamiento

## State

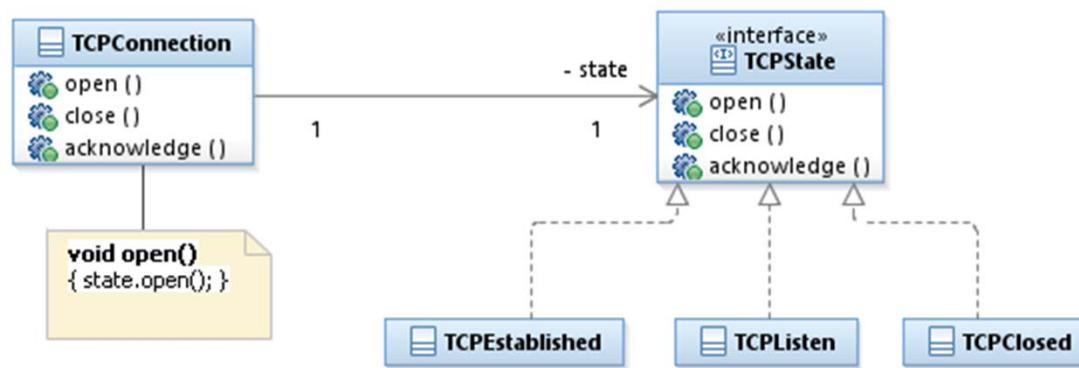
- Propósito
  - Permite que un objeto modifique su comportamiento cada vez que cambie su estado interno. Parecerá que cambia la clase del objeto.
- También conocido como
  - Estado.
  - Objects for States (estados como objetos).

# Patrones de comportamiento

## State

- Motivación
  - Supongamos una clase ConexionTCP que representa una conexión de red.
  - El estado puede ser: establecida, escuchando o cerrada.
  - En función del estado, la clase responde de distinta forma a los mismos clientes.
  - La idea es encapsular el estado en otra clase y delegar en ella.

# Patrones de comportamiento State



Ejemplo patrón State

# Patrones de comportamiento

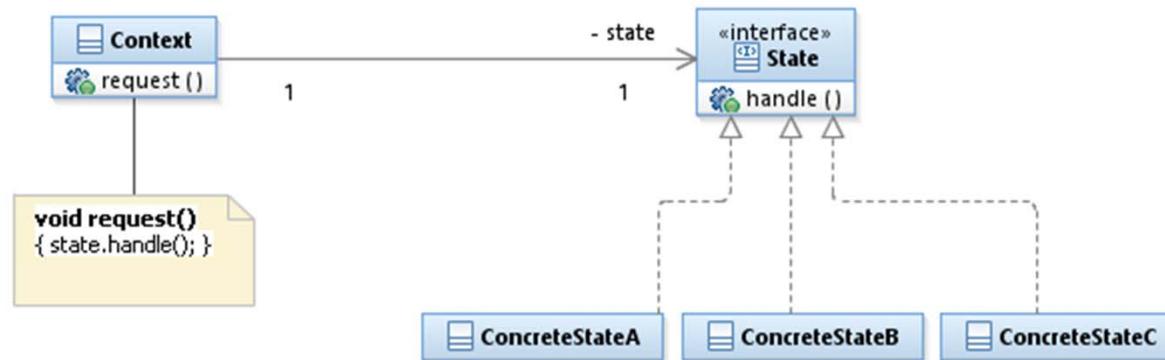
## State

- El patrón State debe aplicarse cuando
  - El comportamiento de un objeto depende de su estado, y debe cambiar en tiempo de ejecución dependiendo de ese estado (esto hace inviable el polimorfismo).
  - Las operaciones tienen largas sentencias condicionales con múltiples ramas que dependen del estado del objeto. El patrón State pone cada rama de la condición en una clase aparte.

# Patrones de comportamiento

## State

- Descripción abstracta



Estructura y comportamiento del patrón State

# Patrones de comportamiento

## State

- Consecuencias
  - Ventajas
    - Localiza el comportamiento dependiente del estado y divide dicho comportamiento en diferentes estados.
    - Hace explícitas las transiciones entre estados.
    - Los objetos estado pueden compartirse

# Patrones de comportamiento

## State

- Código de ejemplo

```
public class ConexionTCP {  
    EstadoTCP estado;  
  
    public ConexionTCP() {  
        //al inicio, cerrado  
        estado= EstadoTCP.getInstancia(Estados.CERRADA);  
    }  
  
    public cambiarEstado(EstadoTCP e)  
    { estado= e; }  
  
    public void abrirActiva()  
    { estado.abrirActiva(this); }
```

# Patrones de comportamiento

## State

```
public void abrirPasiva()
{ estado.abrirPasiva(this); }

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

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

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

};
```

# Patrones de comportamiento

## State

```
public void enviar()  
{ estado.enviar(this); }  
  
}
```

# Patrones de comportamiento

## State

```
public abstract class EstadoTCP {  
    //clase singleton con todos los estados posibles  
    static TCPCerrada instCerrada;  
    static TCPAbierta instEstablecida;  
    .....  
    static EstadoTCP getInstancia(int tipo)  
    { switch (tipo):  
  
        case Estados.CERRADA { if (instCerrada == null)  
                                instCerrada = new TCPCerrada();  
                                return instCerrada; }  
        case Estados.ABIERTA { if (instEstablecida == null)  
                               instEstablecida = new  
                               TCPEstablecida();  
                               return instEstablecida; }  
        .....  
    }  
}
```

# Patrones de comportamiento

## State

```
public void transmitir(ConexionTCP conexion,  
FlujoOctetos flujo) {};  
public void abrirActiva(ConexionTCP conexion) {};  
public void abrirPasiva(ConexionTCP conexion) {};  
public void cerrar(ConexionTCP conexion) {};  
public void sincronizar(ConexionTCP conexion) {};  
public void acuseDeRecibo(ConexionTCP conexion) {};  
public void enviar(ConexionTCP conexion) {};  
protected void cambiarEstado(ConexionTCP  
    conexion, EstadoTCP estado) {};  
public void cambiarEstado(ConexionTCP c, EstadoTCP e)  
{ c.cambiarEstado(e); }  
}
```

# Patrones de comportamiento

## State

```
public class TCPCerrarada extends EstadoTCP {  
    public void abrirActiva (ConexionTCP c)  
    { //abrir conexión  
  
        cambiarEstado(c,  
                      EstadoTCP.getInstancia(Estados.ESTABLECIDA));  
    }  
  
    public void abrirPasiva(ConexionTCP c)  
    { cambiarEstado(c,  
                   EstadoTCP.getInstancia(Estados.ESCUCHANDO));  
    }  
}
```

# Patrones de comportamiento

## State

```
public class TCPEstablecida extends EstadoTCP {  
  
    public void cerrar(ConexionTCP c)  
    { //cerrar conexión  
        cambiarEstado(c,  
                      EstadoTCP.getInstancia(Estados.CERRADA));  
    }  
    public void transmitir(ConexionTCP c, FlujoOctetos f)  
    { c.procesarOcteto(f); }  
}
```

# Patrones de comportamiento

## State

```
public class TCPEscuchando extends EstadoTCP {  
  
    public void enviar(ConexionTCP c)  
    { //enviar  
  
        cambiarEstado(c,  
        EstadoTCP.getInstancia(Estados.ESTABLECIDA));  
    }  
}
```

# Patrones de comportamiento

## Strategy

- Propósito
  - Define una familia de algoritmos, encapsula cada uno de ellos y los hace intercambiables. Permite que un algoritmo varíe independientemente de los clientes que los usan.
- También conocido como
  - Estrategia.
  - Policy (política).

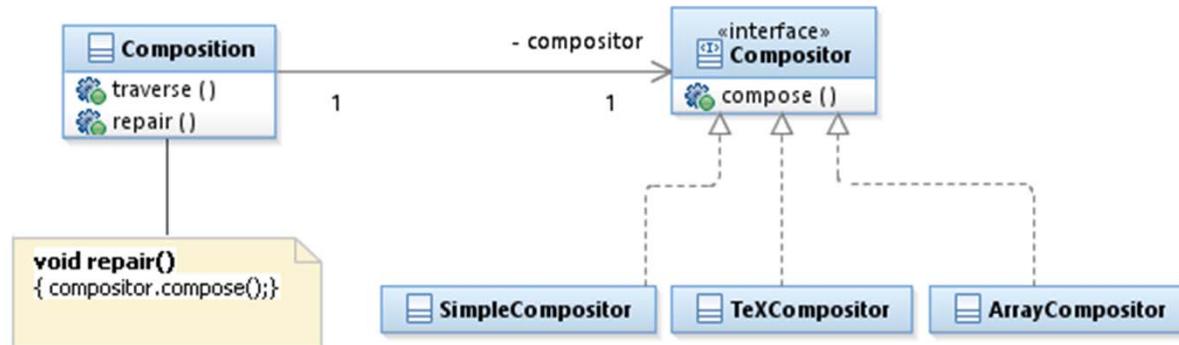
# Patrones de comportamiento

## Strategy

- Motivación
  - Ya hemos comentado que desacoplando funcionalidades obtenemos sistemas con clases más pequeñas, especializadas y fáciles de mantener.
  - Por ejemplo, una composición de texto que puede actuar sobre distintos tipos de textos.

# Patrones de comportamiento

## Strategy



### Ejemplo patrón Strategy

# Patrones de comportamiento

## Strategy

- El patrón Strategy debe aplicarse cuando
  - Muchas clases relacionadas difieren sólo en su comportamiento. Las estrategias permiten configurar una clase con un determinado comportamiento entre muchos posibles.
  - Se necesitan distintas variantes de un algoritmo.
  - Un algoritmo usa datos que los clientes no deberían conocer. El patrón evita exponer estructuras de datos dependientes del algoritmo.

# Patrones de comportamiento

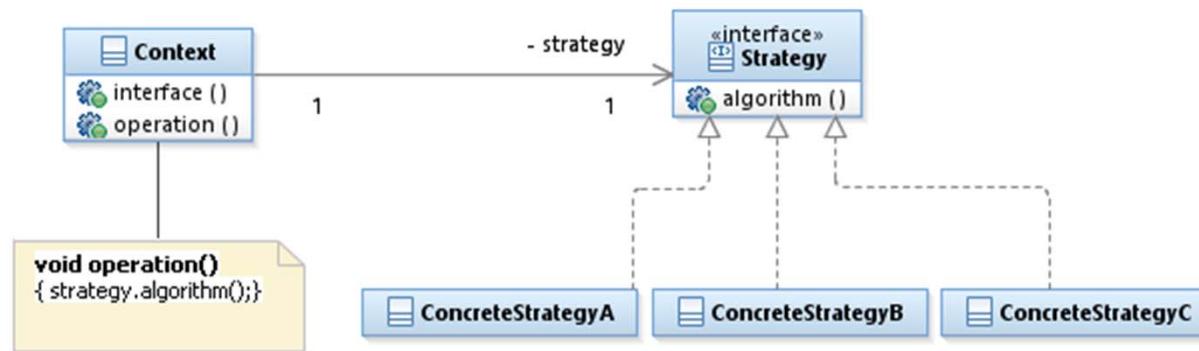
## Strategy

- Una clase define muchos comportamientos, y éstos se representan como múltiples sentencias condicionales en sus operaciones.

# Patrones de comportamiento

## Strategy

- Descripción abstracta



Estructura y comportamiento del patrón Strategy

# Patrones de comportamiento

## Strategy

- Consecuencias
  - Ventajas
    - Permite familias de algoritmos relacionados.
    - Alternativa a la herencia.
    - Las estrategias eliminan las sentencias condicionales.
    - Permite una elección de implementaciones.
  - Inconvenientes
    - Los clientes deben conocer las diferentes estrategias.
    - Costes de comunicación.
    - Mayor número de objetos.

# Patrones de comportamiento

## Strategy

- Código de ejemplo

```
public class Composicion {  
  
    Componedor componedor;  
    Componente componentes;  
    int contadorComponentes;  
    int anchoLinea;  
    int saltosLinea;  
    int contadorLineas  
  
    public Composicion (Componedor c) {...}
```

# Patrones de comportamiento

## Strategy

```
void reparar()
{ Coord natural;
  Coord maxima;
  Coord minima;
  int contadorComponentes;
  int saltos
  Componedor componedor;

  //preparación de los arrays con los tamaños
  //deseados de los componentes
  //...
```

# Patrones de comportamiento

## Strategy

```
//determina donde van los saltos
int contadorSaltos;
contadorSaltos= componedor.componer(natural,
    maxima, minima, contadorComponentes,
    anchoLinea, saltos);
//colocar los componentes en función de los
//saltos
}

};

};
```

# Patrones de comportamiento

## Strategy

```
public interface Componedor {  
    public int componer(Coord natural[], Coord  
    estirado[], Coord encogido[], int  
    contadorComponentes, int anchoLinea, int  
    saltos[]);  
};
```

# Patrones de comportamiento

## Strategy

```
public class ComponedorSimple implements  
    Componedor {  
... };
```

```
public class ComponedorMatriz implements  
    Componedor {  
... };  
.....
```

```
Composición rápida= new Composición (new  
    ComponedorSimple());
```

```
Composición iconos= new Composición (new  
    ComponedorMatriz(100));
```

# Patrones de comportamiento

## Template Method

- Propósito
  - Define en una operación el esqueleto de un algoritmo, delegando en las subclases algunos de sus pasos. Permite que las subclases redefinan ciertos pasos de un algoritmo sin cambiar su estructura.
- También conocido como
  - Método plantilla.

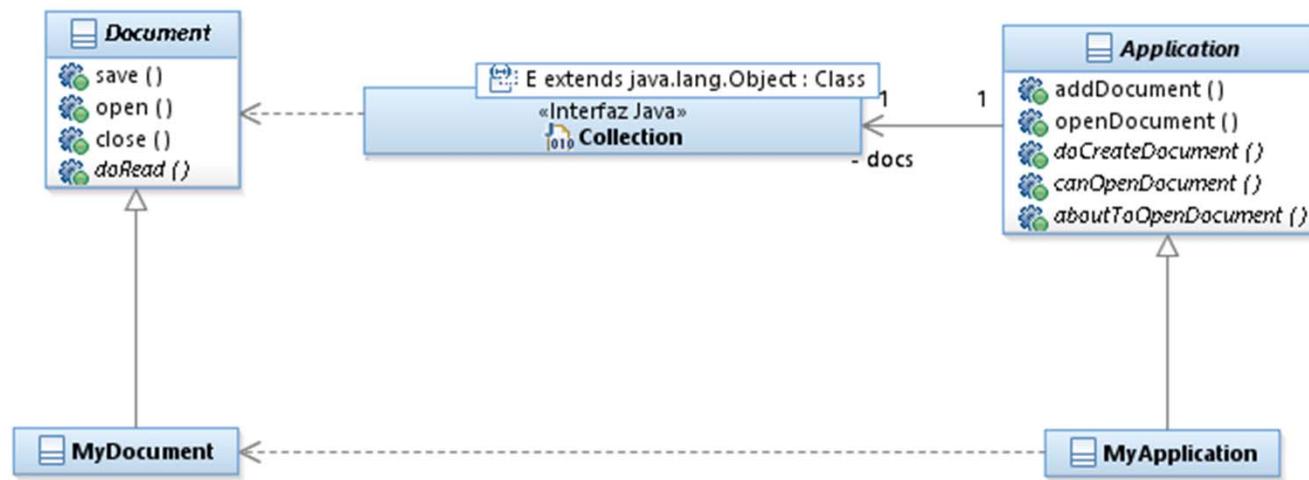
# Patrones de comportamiento

## Template Method

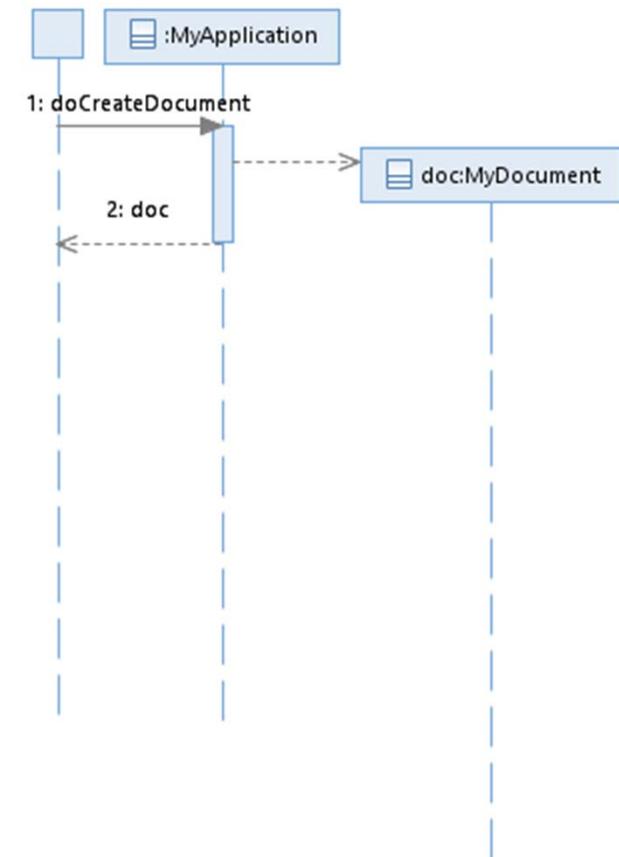
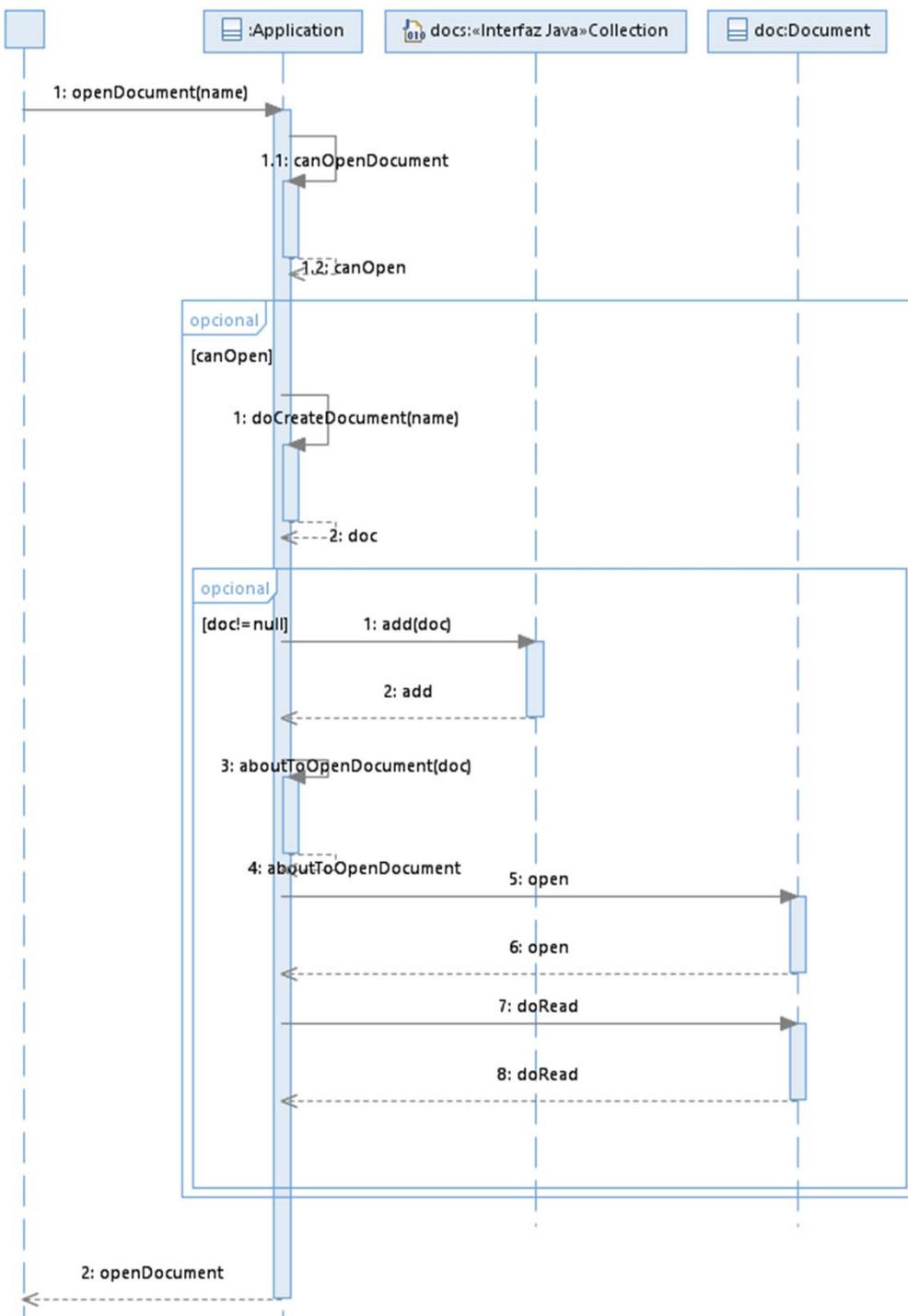
- Motivación
  - Podemos tener un marco de aplicaciones y documentos.
  - La interacción entre estas clases es siempre la misma.
  - Las clases concretas deben responder de distinta forma a la misma interacción.

# Patrones de comportamiento

## Template Method



Ejemplo estructura Template Method



Ejemplo interacción  
Template Method

# Patrones de comportamiento

## Template Method

- El patrón TM debe aplicarse cuando
  - Se quiere implementar las partes de un algoritmo que no cambian, y dejar que sean las subclases quienes implementen el comportamiento que pueda variar.
  - Cuando el comportamiento repetido de varias subclases debería factorizarse y ser localizado en una clase común para evitar el código duplicado.

# Patrones de comportamiento

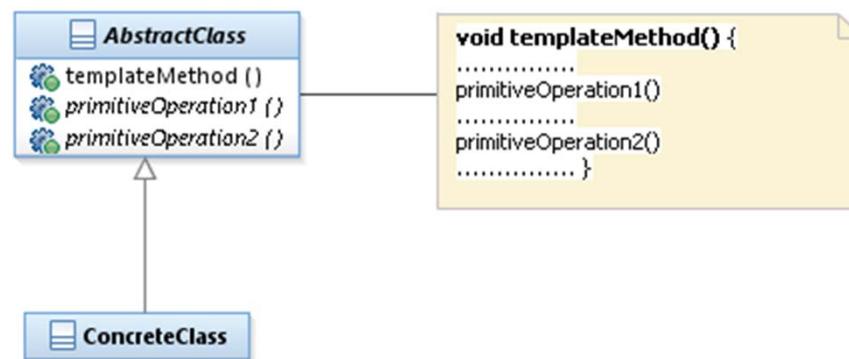
## Template Method

- Para controlar las extensiones de las subclases.

# Patrones de comportamiento

## Template Method

- Descripción abstracta



Estructura y comportamiento del patrón Template Method

# Patrones de comportamiento

## Template Method

- Consecuencias
  - Ventajas
    - Permiten la reutilización de código.

# Patrones de comportamiento

## Template Method

- Código de ejemplo

```
public class Vista {  
    .....  
    public void Mostrar()  
    { asignarFoco();  
        hacerMostrar();  
        quitarFoco();  
    }  
    public void hacerMostrar() { } //no hace nada  
};
```

# Patrones de comportamiento

## Template Method

```
public class MiVista extends Vista {  
    .....  
  
    public void hacerMostrar()  
    { //muestra los contenidos de la vista }  
  
};
```

# Patrones de comportamiento

## Visitor

- Propósito
  - Representa una operación sobre los elementos de una estructura de objetos. Permite definir una nueva operación sin cambiar las clases de los elementos sobre los que opera.
- También conocido como
  - Visitante.

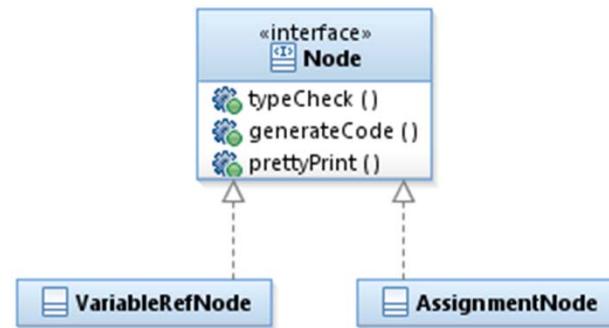
# Patrones de comportamiento

## Visitor

- Motivación
  - Un compilador puede representar programas como árboles de sintaxis abstracta.
  - Necesitamos hacer varias operaciones sobre los nodos de los árboles (e.g. comprobación de tipos, generación de código).
  - Podemos incluir cada operación necesaria para cada nodo en una superclase.

# Patrones de comportamiento

## Visitor



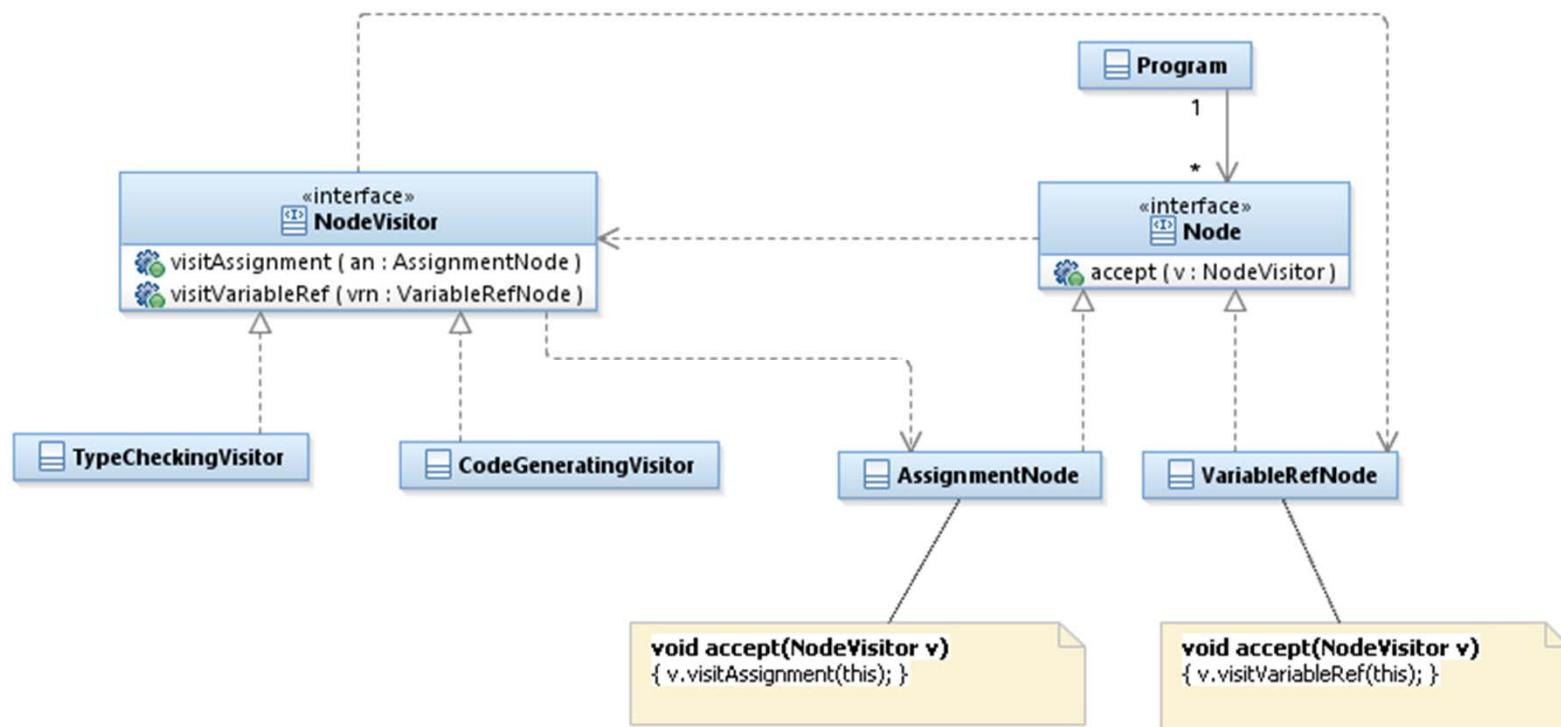
Una opción de construcción

# Patrones de comportamiento

## Visitor

- También podemos agrupar procesamientos similares en clases que procesan cada elemento del lenguaje según sus intereses.

# Patrones de comportamiento Visitor



Ejemplo patrón Visitor

# Patrones de comportamiento

## Visitor

- El patrón Visitor debe aplicarse cuando
  - Una estructura de objetos contiene muchas clases de objetos con diferentes interfaces, y queremos realizar operaciones sobre esos elementos que dependen de sus clases concretas.
  - Se necesitan realizar muchas operaciones distintas y no relacionadas sobre objetos de una estructura de objetos, y queremos evitar “contaminar” su clases con dichas operaciones.

# Patrones de comportamiento

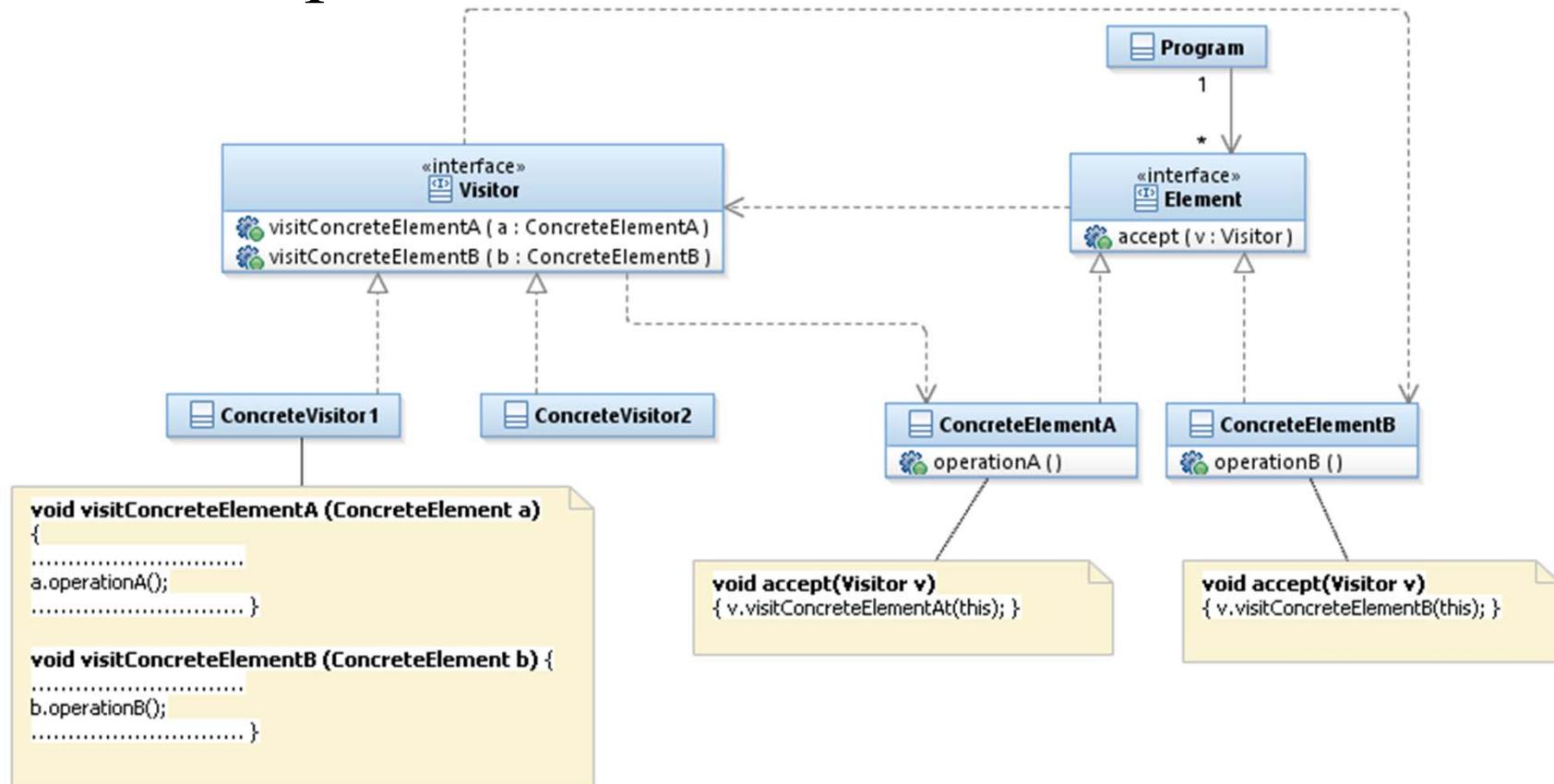
## Visitor

- Las clases que definen la estructura de objetos rara vez cambian, pero muchas veces queremos definir nuevas operaciones sobre la estructura.

# Patrones de comportamiento

## Visitor

- Descripción abstracta



# Patrones de comportamiento

## Visitor

- Consecuencias
  - Ventajas
    - El visitante facilita añadir nuevas operaciones.
    - Un visitante agrupa operaciones relacionadas y separa las que no lo están.
    - Permite visitar jerarquías de clases frente a un iterador que necesita un interfaz común para manejarlas.
    - Permiten acumular el estado.

# Patrones de comportamiento

## Visitor

- Inconvenientes

- Añadir nuevas clases elemento obliga a añadir las clases visitor, y probablemente a redefinir sus operaciones.
- Puede romper la encapsulación.

# Patrones de comportamiento

## Visitor

- Código de ejemplo. Compuesto\*

```
package compuesto;

import datos.IteradorDatos;

public interface Componente
{
    /*no tiene operaciones "precio()" o "aString()"
    porque estás estarán incluidas en el visitante,
    y por supuesto en los nodos */
    public void aceptar(VisitanteComponente v);
}
```

\*A diferencia de GoF, nuestro visitante es el responsable de recorrer al compuesto

# Patrones de comportamiento

## Visitor

```
/*si no se incluyen estas operaciones, forzaríamos a
que los clientes creen objetos de la clase
compuesto, rompiendo la visión homogénea de
componentes. Los nodos podrían lanzar una excepción,
por ejemplo */
public int añadir(Componente c);
public int eliminar(Componente c);
public IteradorDatos iterador();
}
```

# Patrones de comportamiento

## Visitor

```
package compuesto;

import datos.IteradorDatos;

/*incluye las operaciones atómicas que necesita
utilizar el visitante pero que no parece
razonable incluir en componentes compuestos
ya que en el caso de componentes compuestos
dichas operaciones serán implementadas por el
visitante correspondiente
lo de Comparable se debe a la estructura de datos
de Compuesto */

public class Nodo implements Componente, Comparable
{
```

# Patrones de comportamiento

## Visitor

```
String nombre;
int precio;

public Nodo (String nombre, int precio)
{
    this.nombre= new String(nombre);
    this.precio= precio;
}

public String aString()
{
    return nombre;
}
```

# Patrones de comportamiento

## Visitor

```
public float precio()
{
    return precio;
}

public int compareTo(Object o)
{
    ...
}

public void aceptar(VisitanteComponente visitante)
{
    visitante.visitaNodo(this);
}
```

# Patrones de comportamiento

## Visitor

```
public int añadir(Componente c)
{ return -2; }

public int eliminar(Componente c)
{return -2; }

public IteradorDatos iterador()
{return null; }

}
```

# Patrones de comportamiento

## Visitor

```
package compuesto;

public class DVD extends Nodo {
    public DVD (String nombre, int precio)
    {
        super(nombre, precio);
    }

}

public class Memoria extends Nodo {...}

public class Procesador extends Nodo {...}
```

# Patrones de comportamiento

## Visitor

```
package compuesto;

import datos.Datos;
import datos.IteradorDatos;
import datos.ListaEnlazada;

public class Compuesto implements Comparable,
    Componente {

    private String nombre;
    private Datos datos;
    private Float precioComponenteCompuesto;
```

# Patrones de comportamiento

## Visitor

```
public Compuesto (String nombre)
{   this.nombre= new String(nombre);
    datos= new ListaEnlazada(); //sin factoría
}

//a propósito se ha obviado el utilizar el
//nombre "aString()"
public String nombre()
{ return nombre; }

public Float precioComponenteCompuesto()
{ return precioComponenteCompuesto; }
```

# Patrones de comportamiento

## Visitor

```
public void aceptar(VisitanteComponente visitante)
{
    visitante.visitaCompuesto(this);
}

public int añadir(Componente c)
{
    return datos.insertar((Comparable)c);
}

public int eliminar(Componente c)
{
    return datos.eliminar(c);
}
```

# Patrones de comportamiento

## Visitor

```
public IteradorDatos iterador()
{
    return datos.iterador();
}

public int compareTo(Object o)
{
    ...
}
```

# Patrones de comportamiento

## Visitor

```
package compuesto;

public class Ordenador extends Compuesto {

    public Ordenador(String nombre)
    {
        super(nombre);
    }

}

public class Placa extends Compuesto { ... }
```

# Patrones de comportamiento

## Visitor

```
package compuesto;

public interface VisitanteComponente
{
    public void visitaNodo(Nodo nodo);
    public void visitaCompuesto(Compuesto compuesto);
}
```

# Patrones de comportamiento

## Visitor

```
package compuesto;

import datos.IteradorDatos;

public class VisitantePrecio implements
    VisitanteComponente {

    public int precio;

    public VisitantePrecio()
    {
        precio= 0;
    }
```

# Patrones de comportamiento

## Visitor

```
public void visitaNodo(Nodo nodo)
{
    precio+= nodo.precio();
}
```

# Patrones de comportamiento

## Visitor

```
public void visitaCompuesto(Compuesto compuesto)
{
    //el precio de la placa base, por ejemplo
    precio+= compuesto.precioComponenteCompuesto();

    IteradorDatos iter= compuesto.iterador();

    if (iter!=null)
        while (iter.tieneSiguiente())
        { Componente componente= (Componente)
            iter.siguiente();
            componente.aceptar(this);
        }
}
```

# Patrones de comportamiento

## Visitor

```
public int precio()  
{  
    return precio;  
}  
}
```

# Patrones de comportamiento

## Visitor

```
package compuesto;

import datos.IteradorDatos;

public class VisitanteAString implements
    VisitanteComponente {

    public String cadena;

    public VisitanteAString()
    {
        cadena= new String();
    }

    public void visitar(IteradorDatos iterador)
    {
        String linea;
        while((linea= iterador.siguienteLinea())!= null)
            cadena+= linea;
    }
}
```

# Patrones de comportamiento

## Visitor

```
public String string()
{
    return cadena;
}

public void visitaNodo(Nodo nodo)
{
    cadena+= nodo.aString()+' ';
}
```

# Patrones de comportamiento

## Visitor

```
public void visitaCompuesto(Compuesto compuesto)
{
    IteradorDatos iter= compuesto.iterador();

    cadena+= compuesto.nombre() + '\n';
    if (iter!=null)
        while (iter.tieneSiguiente())
        {
            Componente componente= (Componente)
                iter.siguiente();
            componente.aceptar(this);
        }
    }
}
```

# Patrones de comportamiento

## Visitor

```
package compuesto;

public class Principal {

    public static void main(String [] args ){

        Componente ordenador= new Ordenador( "ordenador
            simple" );
        Componente placa= new Placa( "iG965" );
        Componente procesador= new Procesador( "iE6300" );
        Componente dvd= new DVD( "+-RW" , 100 );
        Componente memoria= new Memoria ( "DDR2800" , 300 );
```

# Patrones de comportamiento

## Visitor

```
placa.añadir(procesador);
placa.añadir(memoria);
ordenador.añadir(placa);
ordenador.añadir(dvd);

VisitantePrecio vp= new VisitantePrecio();
ordenador.aceptar(vp);
System.out.println(vp.precio());

VisitanteAString vs= new VisitanteAString();
ordenador.aceptar(vs);
System.out.println(vs.string());

}
```

# Patrones de comportamiento

## Visitor

- Código de ejemplo. Intérprete

```
package visitor;

public interface ExpBooleana
{
    public void acepta(VisitanteExpBooleana v);
}
```

# Patrones de comportamiento

## Visitor

```
package visitor;
public class ExpVariable implements ExpBooleana {
    private String nombre;
    public ExpVariable (String nombre)
    {   this.nombre= new String(nombre); }

    public String getNombre()
    {   return nombre;     }

    public void acepta(VisitanteExpBooleana v)
    {
        v.visitExpVariable(this);
    }
}
```

# Patrones de comportamiento

## Visitor

```
package visitor;
public class ExpCte implements ExpBooleana {
    private Boolean valor;
    public ExpCte(Boolean valor)
    {   this.valor= valor;   }

    public Boolean getValor()
    {   return valor;   }

    public void acepta(VisitanteExpBooleana v)
    {
        v.visitExpCte(this);
    }
}
```

# Patrones de comportamiento

## Visitor

```
package visitor;  
public class ExpAnd implements ExpBooleana {  
    private ExpBooleana exp1;  
    private ExpBooleana exp2;  
    public ExpAnd(ExpBooleana exp1, ExpBooleana exp2)  
    {    this.exp1= exp1;  
        this.exp2= exp2;    }  
  
    public ExpBooleana getExp1()  
    {    return exp1;    }  
  
    public ExpBooleana getExp2()  
    {    return exp2;}
```

# Patrones de comportamiento

## Visitor

```
public void acepta(VisitanteExpBooleana v)
{
    v.visitExpAnd(this);
}
```

# Patrones de comportamiento

## Visitor

```
package visitor;
public class ExpOr implements ExpBooleana {
    private ExpBooleana exp1;
    private ExpBooleana exp2;
    public ExpOr(ExpBooleana exp1, ExpBooleana
exp2)
    {   this.exp1= exp1;
        this.exp2= exp2;  }

    public ExpBooleana getExp1( )
    {   return exp1; }

    public ExpBooleana getExp2( )
    {   return exp2;      }
```

# Patrones de comportamiento

## Visitor

```
public void acepta(VisitanteExpBooleana v)
{
    v.visitExpOr(this);
}
```

# Patrones de comportamiento

## Visitor

```
package visitor;
public class ExpNot implements ExpBooleana {
    private ExpBooleana exp;

    public ExpNot(ExpBooleana exp)
    {   this.exp= exp;           }
    public ExpBooleana getExp( )
    {   return exp;  }

    public void acepta(VisitanteExpBooleana v)
    {
        v.visitExpNot(this);
    }
}
```

# Patrones de comportamiento

## Visitor

```
package visitor;

public interface VisitanteExpBooleana {

    public void visitaExpVariable(ExpVariable expVar);
    public void visitaExpCte(ExpCte expCte);
    public void visitaExpAnd(ExpAnd expAnd);
    public void visitaExpOr(ExpOr expOr);
    public void visitaExpNot(ExpNot expNot);
}
```

# Patrones de comportamiento

## Visitor

```
package visitor;
import java.util.HashMap;
public class VisitanteEval implements
    VisitanteExpBooleana {
    Boolean valor;
    HashMap contexto;

    public VisitanteEval(HashMap contexto)
    {
        this.contexto= contexto;
    }
}
```

# Patrones de comportamiento

## Visitor

```
public void visitaExpVariable(ExpVariable expVar)
{
    valor=
(Boolean)contexto.get(expVar.getNombre( ));

}

public void visitaExpCte(ExpCte expCte)
{
    valor= expCte.getValor();
}
```

# Patrones de comportamiento

## Visitor

```
public void visitaExpAnd( ExpAnd expAnd )
{
    ExpBoleana exp1= expAnd.getExp1( );
    exp1.acepta( this );
    Boolean valor1= valor;

    //se haría solamente si valor1 fuera cierto,
    //pero bueno
    ExpBoleana exp2= expAnd.getExp2( );
    exp2.acepta( this );
    Boolean valor2= valor;

    valor= valor1 & valor2;
}
```

# Patrones de comportamiento

## Visitor

```
public void visitaExpOr(ExpOr expOr)
{
    ExpBooleana exp1= expOr.getExp1();
    exp1.acepta(this);
    Boolean valor1= valor;

    //se haría solamente si valor1 fuera falso,
    //pero bueno
    ExpBooleana exp2= expOr.getExp2();
    exp2.acepta(this);
    Boolean valor2= valor;

    valor= valor1 | valor2;
}
```

# Patrones de comportamiento

## Visitor

```
public void visitaExpNot(ExpNot expNot)
{
    ExpBoleana expNegada= expNot.getExp();
    expNegada.acepta(this);
    valor= !valor;
}

public Boolean getValor()
{
    return valor;
}
```

# Patrones de comportamiento

## Visitor

```
package visitor;  
import java.util.HashMap;  
  
public class VisitanteAString implements  
    VisitanteExpBoleana {  
    String cadena;  
    HashMap contexto;  
  
    public VisitanteAString(HashMap contexto)  
    {  
        this.contexto= contexto;  
        cadena= new String();  
    }  
}
```

# Patrones de comportamiento

## Visitor

```
public void visitaExpVariable(ExpVar expVar)
{
    Boolean valor=
(Boolean)contexto.get(expVar.getNombre( ));
    cadena+=
expVar.getNombre( )+ " [ "+valor.toString( )+" ] ";
}
```

```
public void visitaExpCte(ExpCte expCte)
{
    Boolean valor= expCte.getValor( );
    cadena+= valor.toString( );
}
```

# Patrones de comportamiento

## Visitor

```
public void visitaExpAnd( ExpAnd expAnd )
{
    cadena+= " ( ";
    ExpBooleana exp1= expAnd.getExp1( );
    exp1.acepta( this );
    cadena+= "& ";
    ExpBooleana exp2= expAnd.getExp2( );
    exp2.acepta( this );
    cadena+= " ) ";
}
```

# Patrones de comportamiento

## Visitor

```
public void visitaExpOr(ExpOr expOr)
{
    cadena+= " ( ";
    ExpBoleana exp1= expOr.getExp1();
    exp1.acepta(this);
    cadena+= " | ";
    ExpBoleana exp2= expOr.getExp2();
    exp2.acepta(this);
    cadena+= " ) ";
}
```

# Patrones de comportamiento

## Visitor

```
public void visitaExpNot(ExpNot expNot)
{
    cadena+="!";
    ExpBoleana expNegada= expNot.getExp();
    expNegada.acepta(this);
}
```

# Patrones de comportamiento

## Visitor

```
public String getString( )
{
    return cadena;
}

public void limpiaString( )
{
    cadena= new String( );
}
```

# Patrones de comportamiento

## Visitor

```
package visitor;

import java.util.HashMap;

public class Principal
{
    public static void main(String []args)
    {
        ExpBooleana expresion;
        HashMap contexto= new HashMap( );
    }
}
```

# Patrones de comportamiento

## Visitor

```
ExpVariable x= new ExpVariable( "X" );
ExpVariable y= new ExpVariable( "Y" );
expresion= new ExpOr(
            new ExpAnd(new ExpCte(true), x),
            new ExpAnd(y, new ExpNot(x)));
contexto.put(x.getNombre(), false);
contexto.put(y.getNombre(), true);
```

```
VisitanteEval ve= new VisitanteEval(contexto);
expresion.acepta(ve);
System.out.println(ve.getValor());
```

# Patrones de comportamiento

## Visitor

```
VisitanteAString vas= new  
VisitanteAString(contexto);  
expresion.acepta(vas);  
System.out.println(vas.getString());  
vas.limpiarString();
```

```
ExpNot not_expresion= new ExpNot(expresion);  
not_expresion.acepta(ve);  
System.out.println(ve.getValor());  
  
not_expresion.acepta(vas);  
System.out.println(vas.getString());  
vas.limpiarString();
```

# Patrones de comportamiento Visitor

```
ExpAnd exp= new ExpAnd(expresion, not_expresion);
exp.acepta(ve);
System.out.println(ve.getValor());
exp.acepta(vas);
System.out.println(vas.getString());
}

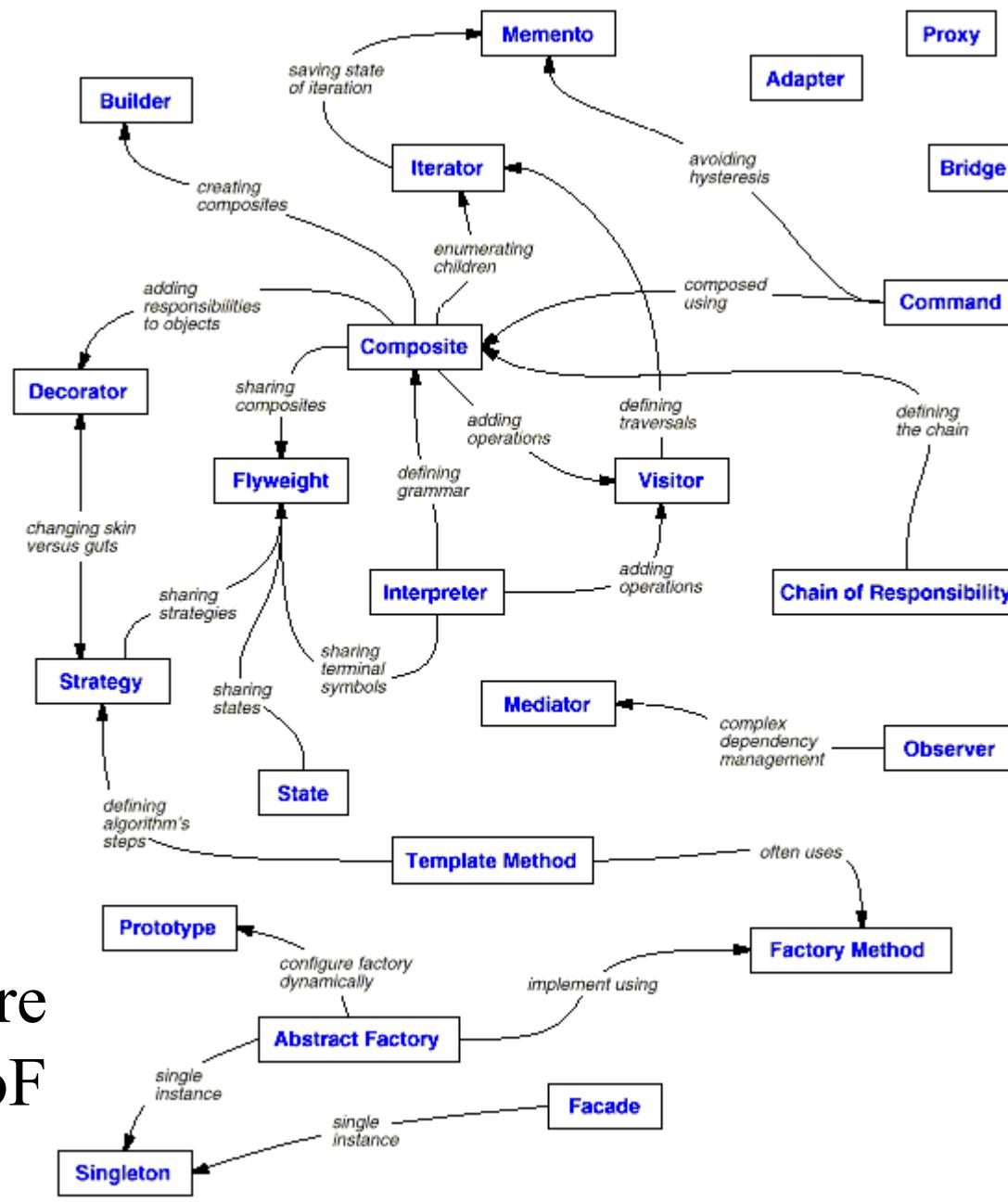
}
```

# Relaciones entre patrones GoF

# Relaciones entre patrones GoF

# Ingeniería del Software

## Antonio Navarro



# Conclusiones

- Patrón: diseño útil
- Patrones GRASP: abstracción de patrones.
- Patrones GoF: extraídos de usos concretos.
- De creación, estructurales de comportamiento
- Relaciones entre patrones GoF.