Patrones de diseño orientado a objetos

Programación Orientada a Objetos
Facultad de Informática

Juan Pavón Mestras
Dep. Ingeniería del Software e Inteligencia Artificial
Universidad Complutense Madrid

Hacer software no es fácil

Diseñar software orientado a objetos es difícil, y diseñar software orientado a objetos reutilizable es todavía más difícil

Chapter 1: Introduction. Design Patterns, *The Gang of Four*

...y un software capaz de evolucionar tiene que ser reutilizable (al menos para las versiones futuras)

Diseñar para el cambio

- El software cambia
- Para anticiparse a los cambios en los requisitos hay que diseñar pensando en qué aspectos pueden cambiar
- Los patrones de diseño están orientados al cambio

Patrones

Cómo llegar a ser un maestro de ajedrez

- Primero aprender las reglas del juego
  - nombres de las piezas, movimientos legales, geometría y orientación del tablero, etc.
- A continuación aprender los principios
  - relativo valor de las piezas, valor estratégico de las casillas centrales, jaque cruzado, etc.
- Sin embargo, para llegar a ser un maestro, hay que estudiar las partidas de otros maestros
  - Estas partidas contienen patrones que deben ser entendidos, memorizados y aplicados repetidamente
- Hay cientos de estos patrones

Patrones

Cómo llegar a ser un maestro del software

- Primero aprender las reglas
  - algoritmos, estructuras de datos, lenguajes de programación, etc.
- A continuación aprender los principios
  - programación estructurada, programación modular, programación OO, programación genérica, etc.
- Sin embargo, para llegar a ser un maestro, hay que estudiar los diseños de otros maestros
  - Estos diseños contienen patrones que deben ser entendidos, memorizados y aplicados repetidamente
- Hay cientos de estos patrones

Patrones

- Christopher Alexander (1977):

  Cada patrón describe un problema que ocurre una y otra vez en nuestro entorno, y describe la esencia de la solución a ese problema, de tal modo que pueda utilizarse esta solución un millón de veces más, sin siquiera hacerlo de la misma manera dos veces

- Un patrón es:
  - una solución a un problema en un contexto particular
  - recurrente (lo que hace la solución relevante a otras situaciones)
  - enseña (permite entender cómo adaptarlo a la variante particular del problema donde se quiere aplicar)
  - tiene un nombre para referirse al patrón
- Los patrones facilitan la reutilización de diseños y arquitecturas software que han tenido éxito

Motivación de los Patrones

- Capturan la experiencia y la hacen accesible a los no expertos
- El conjunto de sus nombres forma un vocabulario que ayuda a que los desarrolladores se comuniquen mejor
  - Lenguajes de patrones
- Ayudan a la gente a comprender un sistema más rápidamente cuando está documentado con los patrones que usa
- Los patrones pueden ser la base de un manual de ingeniería de software

  Si el software se convierte en una ingeniería, las prácticas exitosas deben ser documentadas sistemáticamente y ampliamente difundidas

Motivación de los Patrones

- Facilitan la reestructuración de un sistema tanto si fue o no concebido con patrones en mente
- Reutilización:
  - Los patrones de diseño soportan la reutilización de arquitecturas software
  - Los armazones soportan la reutilización del diseño y del código
- El software cambia
  - Para anticiparse a los cambios en los requisitos hay que diseñar pensando en qué aspectos pueden cambiar
  - Los patrones de diseño están orientados al cambio

Clasificación de patrones

- Patrones *arquitecturales*
  - Expresan un paradigma fundamental para estructurar un sistema software
  - Proporcionan un conjunto de subsistemas predefinidos, con reglas y guías para organizar las relaciones entre ellos
- Patrones *de diseño*
  - Compuestos de varias unidades arquitecturales más pequeñas
  - Describen el esquema básico para estructurar subsistemas y componentes
- Patrones elementales (*idioms*)
  - Específicos de un lenguaje de programación
  - Describen cómo implementar componentes particulares de un patrón

Ejemplos de patrones

- Patrones arquitecturales
  - Jerarquía de capas
  - Tuberías y filtros
  - Cliente/Servidor
  - Maestro-Esclavo
  - Control centralizado y distribuido
- Patrones de diseño
  - Proxies
  - Factorías
  - Adaptadores
  - Composición
  - Broker
- Patrones elementales (*idioms*)
  - Modularidad
  - Interfaces mínimas
  - Encapsulación
  - Objetos
  - Acciones y Eventos
  - Concurrencia

Patrones y Armazones

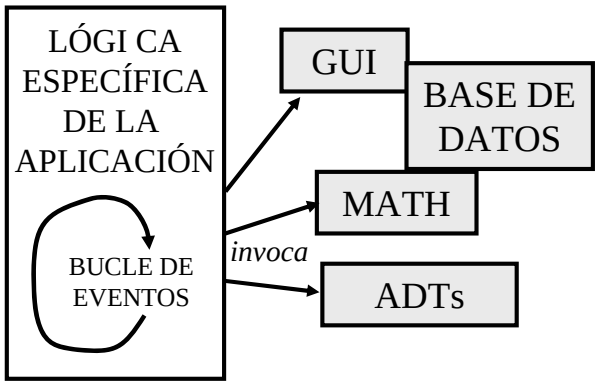
- Los patrones de diseño tienen descripciones más abstractas que los armazones
  - Las descripciones de patrones suelen ser independientes de los detalles de implementación o del lenguaje de programación (salvo ejemplos usados en su descripción)
  - Los armazones están implementados en un lenguaje de programación, y pueden ser ejecutados y reutilizados directamente
- Los patrones de diseño son elementos arquitecturales más pequeños que los armazones
  - Un armazón incorpora varios patrones
  - Los patrones se pueden usar para documentar armazones
- Los patrones de diseño están menos especializados que los armazones
  - Los armazones siempre se aplican a un dominio de aplicación particular

Armazones (Frameworks)

- Características
  - Un armazón ofrece un conjunto integrado de funcionalidad específica de un dominio
    - P.ej.: aplicaciones financieras, servicios de telecomunicación, sistemas de ventanas, bases de datos, aplicaciones distribuidas, núcleos de SO
  - Los armazones invierten el control en ejecución entre la aplicación y el software sobre el que está basada
    - El armazón determina qué métodos se invocan en respuesta a eventos (se reusa el código del cuerpo principal y se escribe el código al que llama)
  - Un armazón es una aplicación medio-acabada
    - Las aplicaciones completas se desarrollan mediante herencia, e instanciando componentes parametrizados del armazón

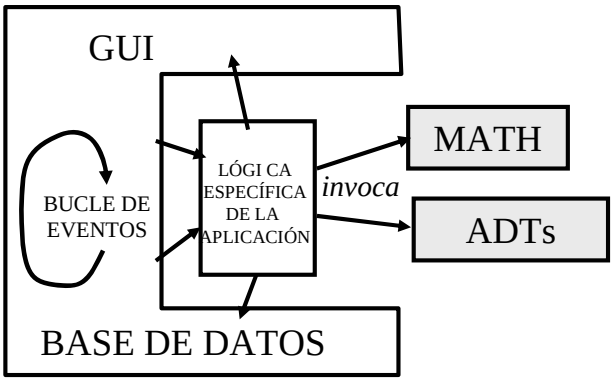
Armazones (*Frameworks*)

Diferencias con bibliotecas



Arquitectura basada en biblioteca de clases

☞ ***reutilización de código***

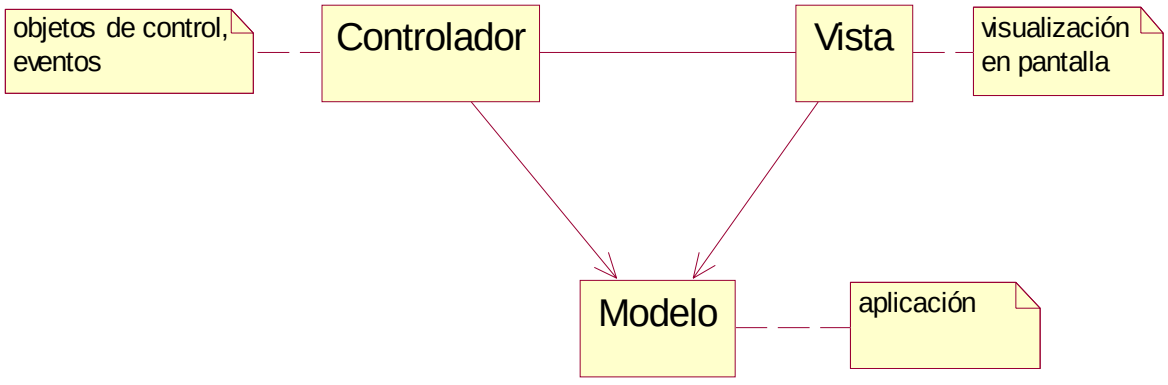


Arquitectura basada en armazón

☞ ***reutilización de diseño y código***

Ejemplo de patrón de diseño

- Modelo-Vista-Controlador
  - Utilizado en Smalltalk [Krasner and Pope, 1988]
  - Distribución de responsabilidades
  - Utilizado en JFC (componentes Swing)



Descripción de patrones de diseño

- Nombre e intención del patrón
  - Referencia al patrón
  - Incrementa el vocabulario de diseño
- Problema y contexto
  - Cuándo aplicar el patrón
- Solución
  - Estructura: elementos que conforman el diseño, sus relaciones, responsabilidades y colaboraciones
    - es una descripción abstracta de cómo una disposición de elementos (clases y objetos) solucionan el problema
  - Se ilustra con un ejemplo de código
- Consecuencias (positivas y negativas)
  - Necesidades (tiempo, memoria), aspectos de implementación y lenguaje de programación, flexibilidad, extensibilidad, portabilidad
- Patrones relacionados

Categorías de patrones de diseño

- Patrones de creación
  - Tratan de la inicialización y configuración de clases y objetos
- Patrones estructurales
  - Tratan de desacoplar interfaz e implementación de clases y objetos:
    - ¿Cómo se componen clases y objetos?
- Patrones de comportamiento
  - Tratan de las interacciones dinámicas entre sociedades de clases y objetos:
    - ¿Cómo interaccionan y se distribuyen responsabilidades los objetos?

Cómo seleccionar un patrón de diseño

- Considerar cómo los patrones de diseño solucionan problemas de diseño
- Buscar las intenciones de cada patrón
- Estudiar cómo se interrelacionan los patrones
- Estudiar patrones de propósito similar
- Examinar la causa de un rediseño
- Considerar qué debería ser variable en un diseño

Cómo usar un patrón de diseño

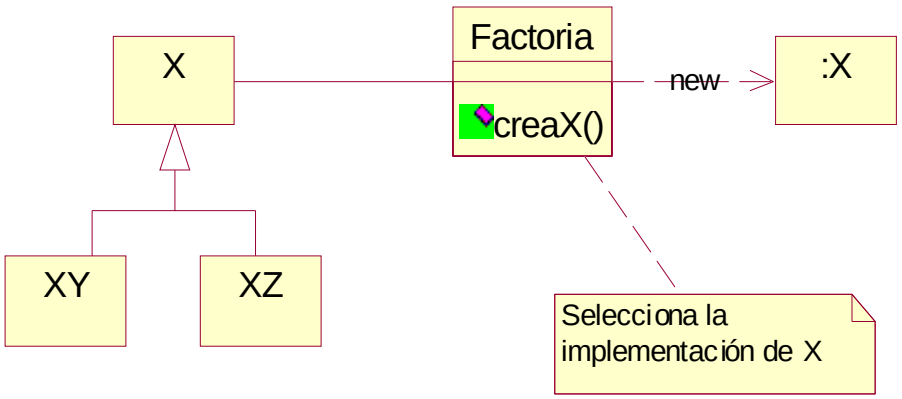
1. Leer el patrón una vez para tener una visión general
2. Volver y estudiar la estructura, los participantes y las colaboraciones
3. Ver un ejemplo concreto codificado del patrón
4. Elegir nombres para los participantes del patrón que sean significativos en el contexto de la aplicación
5. Definir las clases
6. Definir nombres específicos de la aplicación para las operaciones en el patrón
7. Implementar las operaciones que realizarán las responsabilidades y colaboraciones del patrón

Patrones de creación

- En Java para crear un objeto se utiliza el operador new

  unObjeto = new ClaseObjetos ();

- Una clase ***Factoría*** puede ocuparse de los detalles de qué clase de implementación utilizar para crear un objeto de un tipo determinado



Patrones de creación

- Permiten que el sistema sea independiente de cómo se crean, componen o representan sus objetos
  - Sistemas más dependientes de la composición de objetos que de la herencia de clases
    - Se trata de que el comportamiento se defina más por la composición de un conjunto pequeño de comportamientos fundamentales que por la definición mediante herencia de todos los comportamientos posibles
    - Por tanto, la creación de objetos es algo más que instanciar una clase

Patrones de creación

- Temas recurrentes en los patrones de creación:
  - Encapsulan el conocimiento sobre las clases concretas que se van a utilizar
  - Ocultan la manera de crear objetos de estas clases y cómo se juntan
    - La visión global de los objetos del sistema son sus interfaces (que pueden definirse como clases abstractas o interfaces)
    - Se da independencia de
      - qué se crea
      - quién lo crea
      - cómo se crea
      - cuándo se crea

Patrones de creación

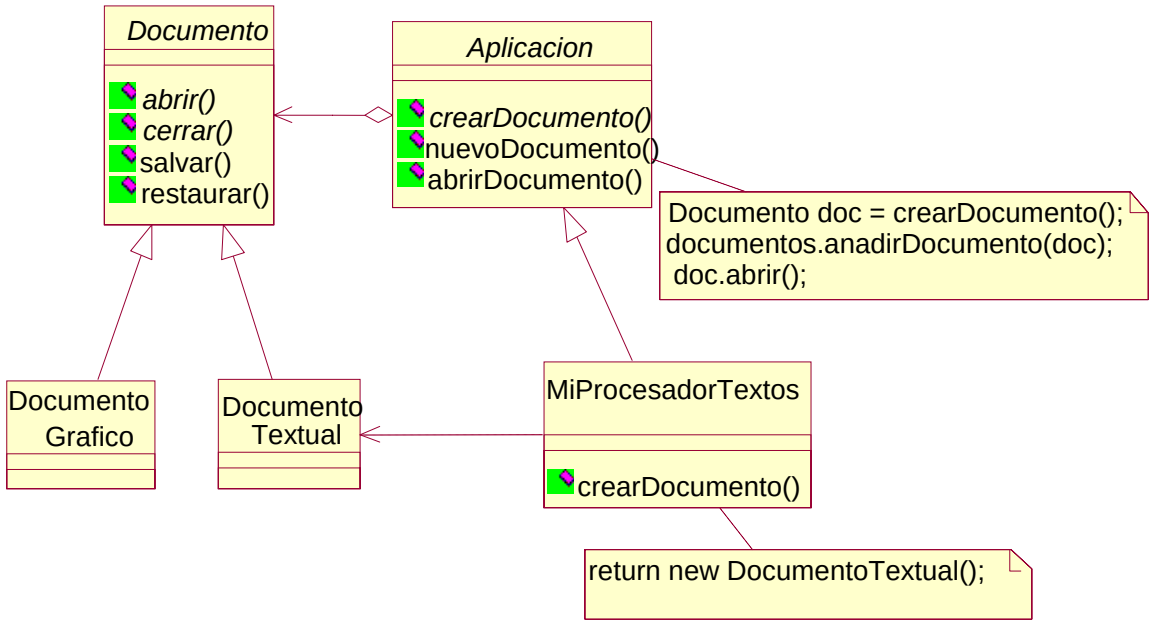
- Tipos de patrones de creación
  - De clase: usa herencia para variar la clase del objeto creado
    - Factoría abstracta
      - Factoría para construir familias de objetos
    - Builder
      - Factoría para construir objetos complejos de forma incremental
  - De objeto: delega la creación en otro objeto
    - Método Factoría
      - Interfaz que permite que sean las subclases las que determinen qué clase instanciar
    - Prototype
      - Factoría para clonar nuevos ejemplares copiando de un prototipo
    - Singleton
    - Object Pool
      - Factoría que asegura que sólo hay un miembro (singleton) o un conjunto determinado (object pool) de una clase, y proporciona un punto global de acceso a él

Método Factoría

- Propósito:

  Permite que una clase difiera la instanciación a las subclases (son éstas las que deciden qué clase instanciar)
- Otras denominaciones:
  - *Factory Method*
  - *Virtual Constructor*
- Motivación
  - Un armazón utiliza clases abstractas para definir y mantener relaciones entre objetos
    - El armazón también es responsable de crear esos objetos
    - En principio el armazón no conoce las clases concretas que se van a definir en una aplicación concreta
    - El método factoría es un método abstracto que encapsula el conocimiento de qué subclase se va a crear y pone este conocimiento fuera del armazón

Método Factoría

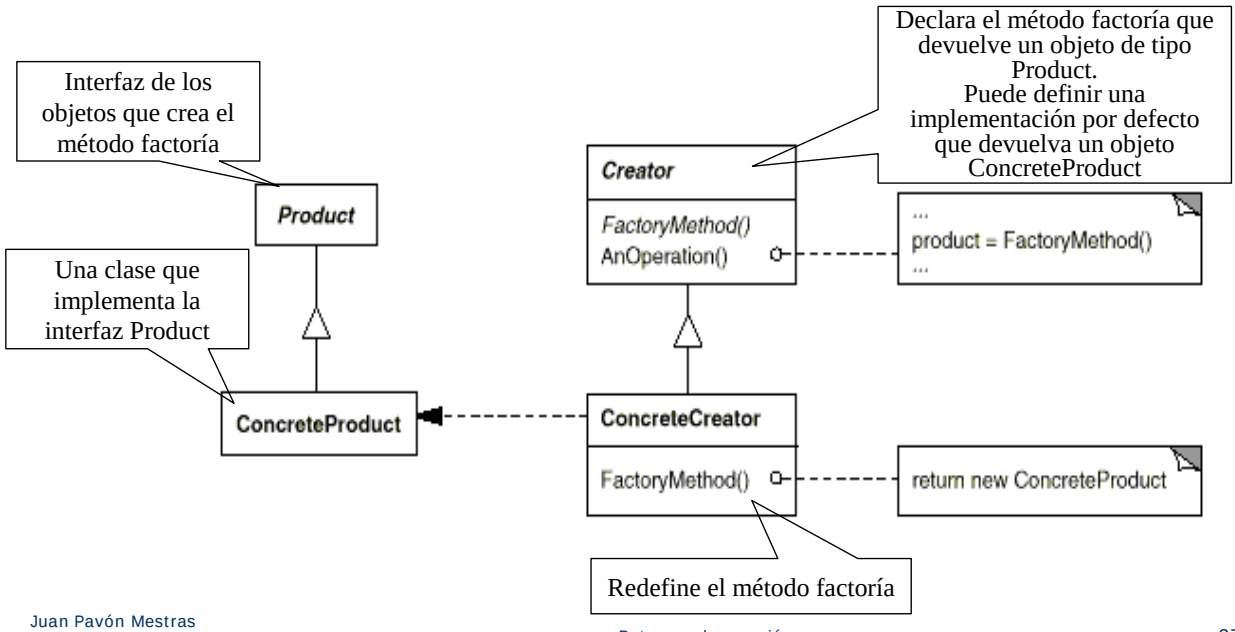


Método Factoría

- Aplicación
  - Cuando una clase no puede anticipar el tipo de objetos que debe crear
  - Cuando una clase quiere que sus clases especifiquen los objetos que deben crear
  - Cuando una clase delega la responsabilidad a una subclase de ayuda (entre varias), y se quiere localizar el conocimiento de qué subclase de ayuda es la delegada

Método Factoría

- Esquema, participantes y colaboraciones
  El Creator deja que sus subclases definan el método factoría que construye y devuelve un objeto del ConcreteProduct apropiado



Método Factoría

- Consecuencias
  - Elimina la necesidad de enlazar clases específicas de la aplicación en el código
  - Es más flexible crear un objeto con un método factoría que directamente
    - Un método factoría puede dar una implementación por defecto, por ejemplo, para crear un archivo, mostrar un diálogo que permita abrir uno ya existente
    - En este ejemplo el método factoría no es abstracto
  - Conectar jerarquías de clases paralelas
    - Los métodos factoría pueden ser llamados por los clientes, no sólo por los Creator

      Ejemplo: manipuladores de figuras: un tipo de manipulador con varios métodos factoría para cada tipo de figura

Método Factoría

- Implementación
  - Implementación de la clase Creator
    - El método factoría es abstracto
    - El método factoría proporciona una implementación por defecto: Permite extensibilidad: se pone la creación de objetos en una operación separada por si el usuario quiere cambiarla
  - Métodos factoría parametrizados
    - Para crear distintos tipos de objetos

```
public class Creator {
   public Product create(ProductId);
};
```


Factoría Abstracta

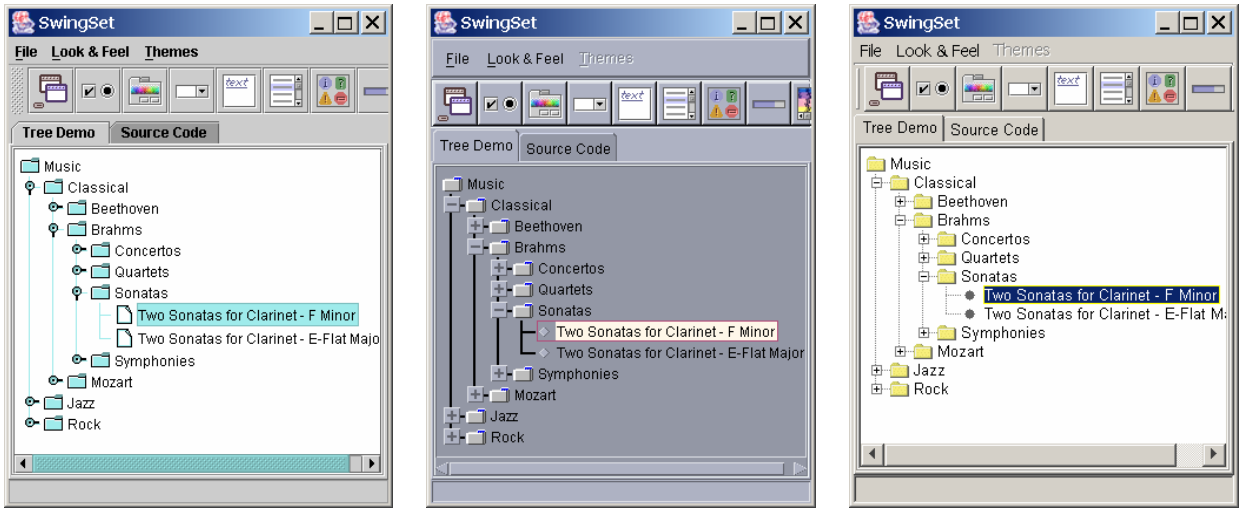
- Propósito:

  Proporcionar una interfaz para crear familias de objetos relaciones o dependientes, sin especificar sus clases concretas
- Otras denominaciones:
  - Abstract Factory
  - Kit
- Motivación
  - Definir una interfaz que soporte diferentes sistemas de ventanes, p.ej. Motif, Windows, Openview, ...

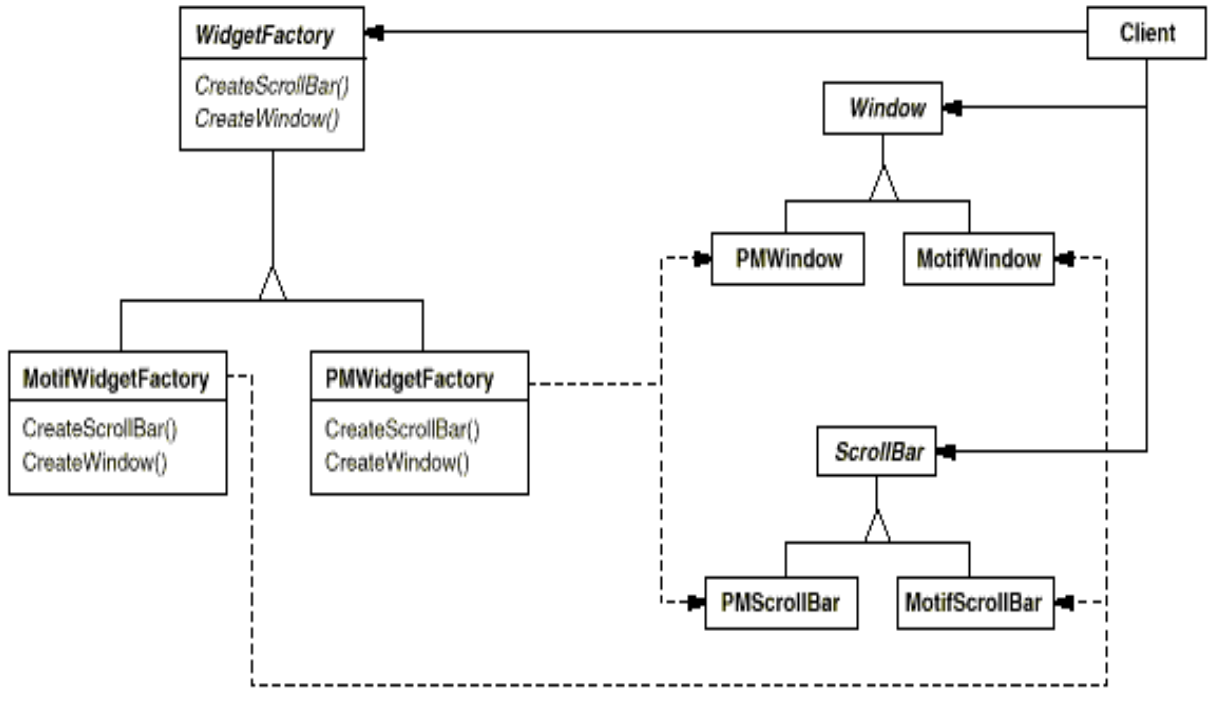
Factoría abstracta: motivación

- Varias apariencias (Look & Feel) para la interfaz gráfica:

JAVA L&F Motif L&F Windows L&F



Factoría Abstracta

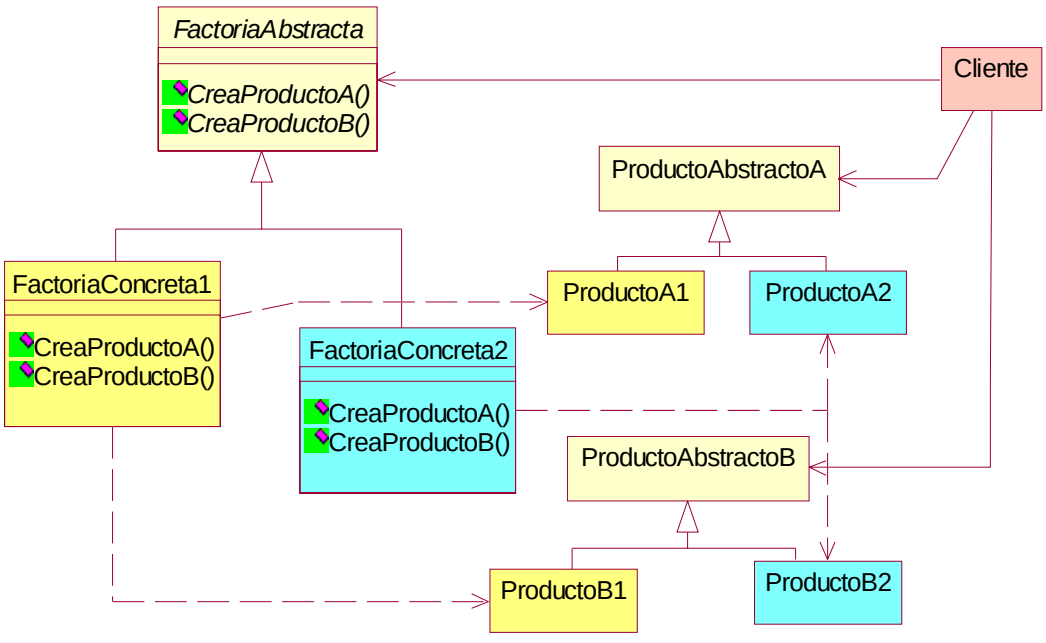


Factoría Abstracta

- Aplicación
  - Para sistemas independientes de cómo se crean, componen y representan sus productos
  - Para sistemas que pueden configurarse con una de varias familias de productos
  - Para obligar a utilizar juntos objetos de una familia de productos
  - Para ofrecer una librería de clases, mostrando sólo sus interfaces y no sus implementaciones

Factoría Abstracta

- Esquema, participantes y colaboraciones



Factoría Abstracta

- Consecuencias
  - Se oculta a los clientes las clases de implementación
    - los clientes manipulan los objetos a través de las interfaces o clases abstractas
  - Facilita el intercambio de familias de productos
    - Al crear una familia completa de objetos con una factoría abstracta, es fácil cambiar toda la familia de una vez simplemente cambiando la factoría concreta
  - Mejora la consistencia entre productos
    - El uso de la factoría abstracta permite forzar a utilizar un conjunto de objetos de una misma familia
  - No es fácil soportar nuevos tipos de productos
    - Si se tiene que extender la interfaz de la Factoría abstracta

Factoría Abstracta

- Implementación
  - Factorías como *singletons*
    - En muchas ocasiones basta con una factoría concreta por cada familia de productos
  - Creación de productos:
    - Método factoría por cada producto:
      La factoría abstracta sólo define una interfaz para crear productos, y es la subclase factoría concreta la que los crea. Ésta especificará sus productos redefiniendo el método factoría para cada uno.
    - Prototipo para cada factoría concreta
      Se crea la factoría concreta inicializada con un objeto prototipo de cada producto. Los nuevos se crean haciendo clones del prototipo
  - Definición de factorías extensibles
    - ¿Qué ocurre cuando se quiere añadir un nuevo tipo de producto?
    - Una solución es añadir un parámetro a las operaciones que crean objetos para especificar el tipo de objeto que se va a crear. (Sólo haría falta una operación de creación que tendría este parámetro)
      - Esta solución requiere también que el cliente haga narrowing o dynamic-cast

Aplicación en Swing

- Abstract factory : LookAndFeel
- Elección de la fábrica concreta : UIManager
- Producto abstracto : ComponentUI y sus clases derivadas (UIDelegate)
- Producto concreto : BasicListUI, MetalListUI…

- Selección de apariencia en Swing

```
private void seleccionarApariencia() {
    // Fuerza al sistema a adoptar una apariencia concreta
    String laf = UIManager.getSystemLookAndFeelClassName();
    try {
      UIManager.setLookAndFeel(laf);
    }
    catch (UnsupportedLookAndFeelException exc)
      {System.err.println("Unsupported: " + laf);}
    catch (Exception exc)
      {System.err.println("Error loading " + laf);}
}
```


Builder

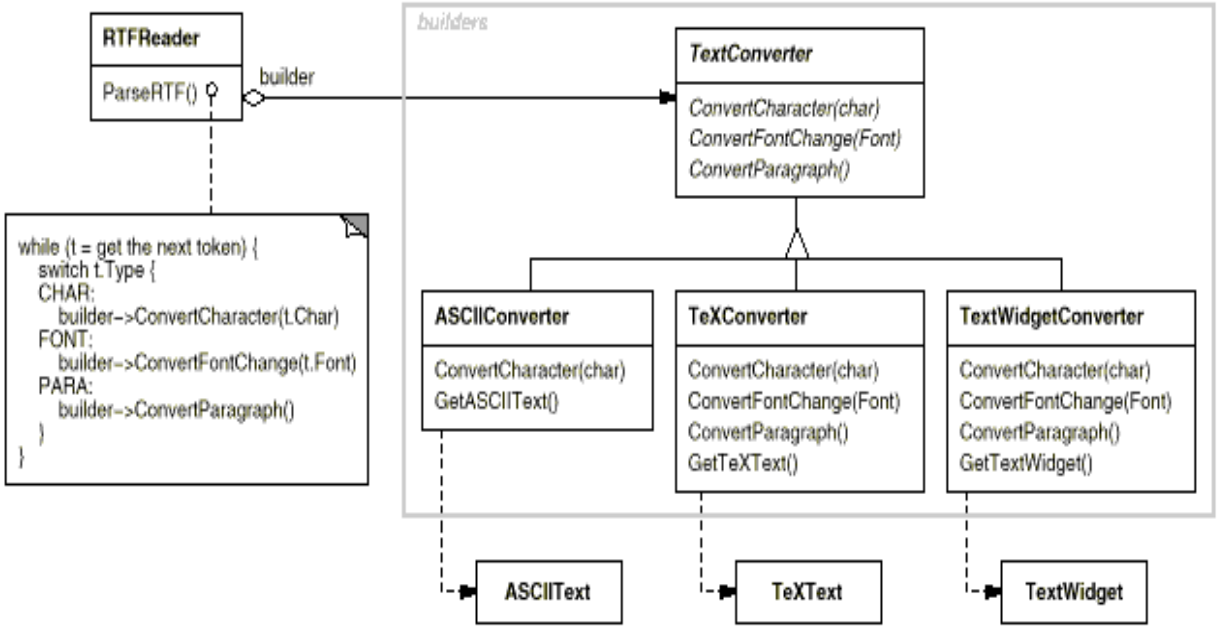
- Propósito:

  Permite a un cliente construir un objeto complejo especificando sólo su tipo y contenido, ocultándole todos los detalles de la construcción del objeto

Builder

- Motivación
  - Supongamos un editor que quiere convertir un tipo de texto (p.ej. RTF) a varios formatos de representación diferentes, y que puede ser necesario en el futuro definir nuevos tipos de representación
    - El lector de RTF(RTFReader) puede configurarse con una clase de Conversor de texto (TextConverter) que convierta de RTF a otra representación
    - A medida que el RTFReader lee y analiza el documento, usa el conversor de texto para realizar la conversión: cada vez que reconoce un token RTF llama al conversor de texto para convertirlo
    - Hay subclases de TextConverter para cada tipo de representación
    - El conversor (builder) está separado del lector (director): se separa el algoritmo para interpretar un formato textual de cómo se convierte y se representa

Builder

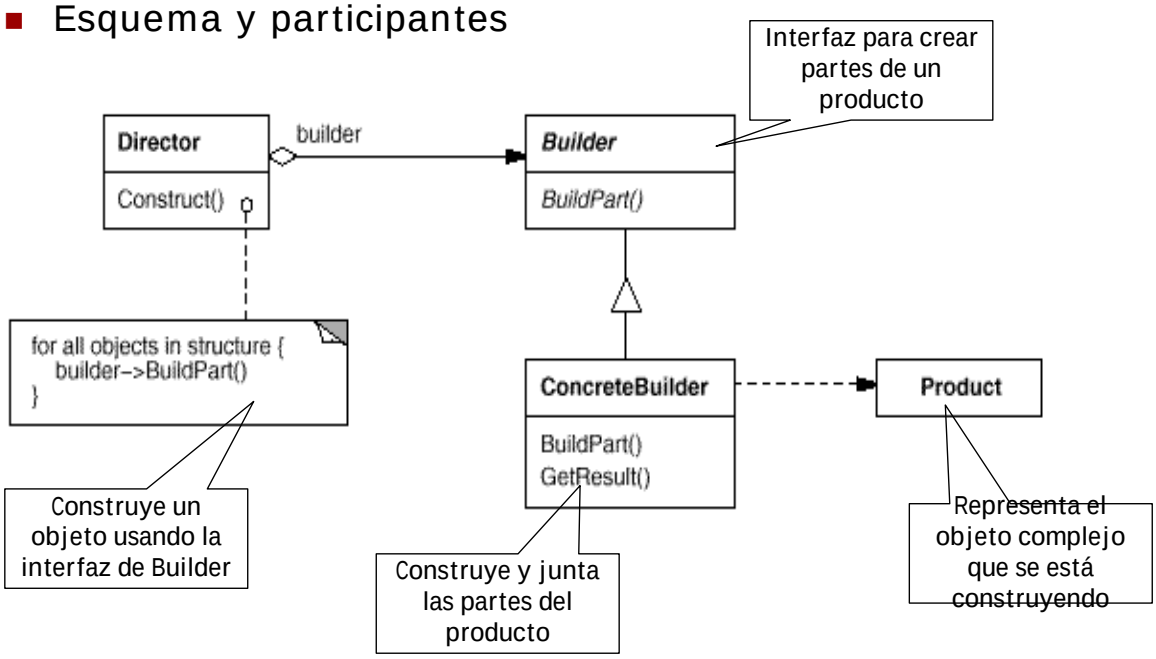


Builder

- Aplicación
  - Cuando el algoritmo para crear un objeto complejo debe ser independiente de las partes que constituyen el objeto y cómo se juntan
  - Cuando el proceso de construcción debe permitir representaciones diferentes para el objeto que se está construyendo

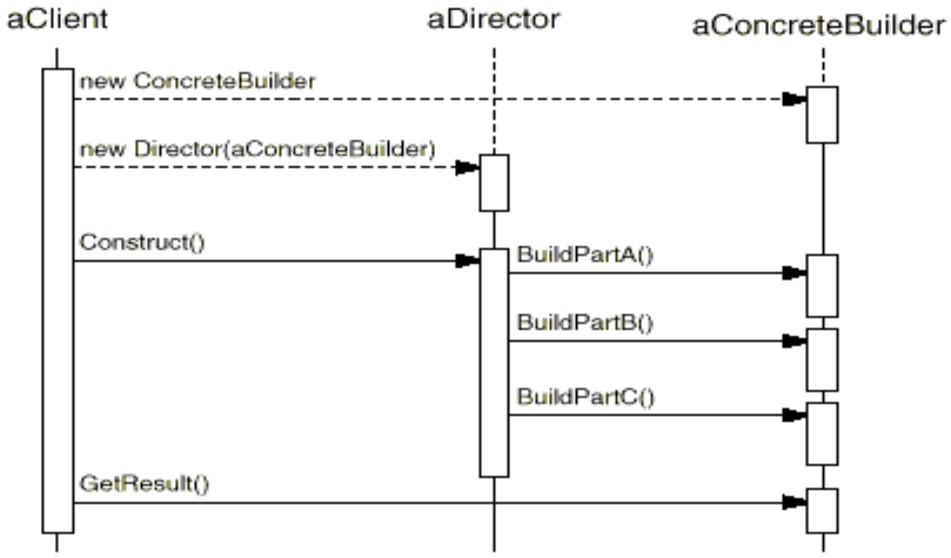
Builder

- Esquema y participantes



Builder

- Colaboraciones



Builder

- Consecuencias
  - Permite variar la representación interna de un producto
    - El Builder ofrece una interfaz al Director para construir un producto y encapsula la representación interna del producto y cómo se juntan sus partes.
    - Si se cambia la representación interna basta con crear otro Bulider que respete la interfaz
  - Separa el código de construcción del de representación
    - Las clases que definen la representación interna del producto no aparecen en la interfaz del Bulider
    - Cada ConcreteBuilder contiene el código para crear y juntar una clase específica de producto
    - Distintos Directors pueden usar un mismo ConcreteBuilder
  - Da mayor control en el proceso de construcción
    - Permite que el Director controle la construcción de un producto paso a paso
    - Sólo cuando el producto está acabado lo recupera el director del builder

Builder

- Implementación y patrones relacionados
  - El Builder define las operaciones para construir cada parte
    - El ConcreteBuilder implementa estas operaciones
  - Con la Factoría Abstracta también se pueden construir objetos complejos
    - Pero el objetivo del patrón Builder es construir paso a paso
    - El énfasis de la Factoría Abstracta es tratar familias de objetos
  - El objeto construido con el patrón Builder suele ser un Composite
  - El Método Factoría se puede utilizar por el Builder para decidir qué clase concreta instanciar para construir el tipo de objeto deseado
  - El patrón Visitor permite la creación de un objeto complejo, en vez de paso a paso, dando todo de golpe como objeto visitante

Singleton

- Propósito:

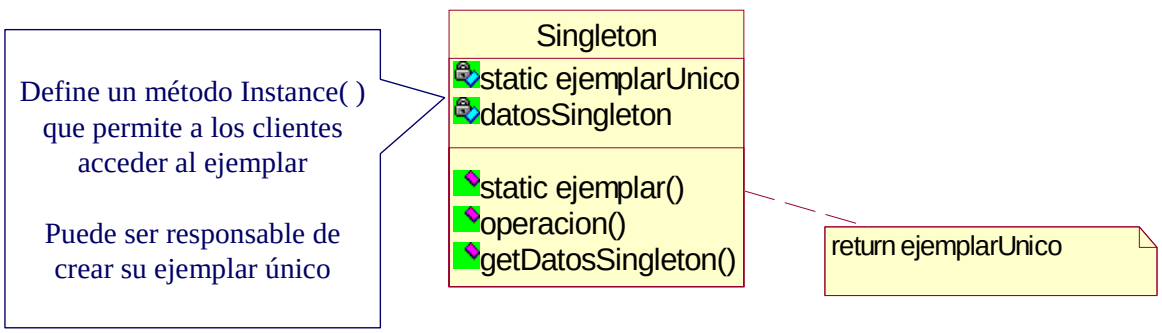
  Asegurar que una clase sólo tiene un ejemplar, y proporcionar un punto de acceso global a éste
- Motivación
  - Algunas clases sólo necesitan exactamente un ejemplar
    - Un *spooler* de impresión en un sistema, aunque haya varias impresoras
    - Un sólo sistema de archivos
    - Un sólo gestor de ventanas
    - ...
  - En vez de tener una variable global para acceder a ese ejemplar único, la clase se encarga de proporcionar un método de acceso

Singleton

- Aplicación
  - Cuando sólo puede haber un ejemplar de una clase, y debe ser accesible a los clientes desde un punto de acceso bien conocido
  - Cuando el único ejemplar pudiera ser extensible por herencia, y los clientes deberían usar el ejemplar de una subclase sin modificar su código

Singleton

- Esquema, participantes y colaboraciones
  - Los clientes acceden al ejemplar de Singleton únicamente a través del método Instance de la clase Singleton



Singleton

- Consecuencias
  - Acceso controlado a un ejemplar único
    - Cómo y cuando los clientes pueden acceder al ejemplar
  - Espacio de nombres reducido
    - Evita la necesidad de utilizar variables globales
  - Permite refinar las operaciones y la representación
    - Se puede heredar de la clase Singleton para configurar el ejemplar para una aplicación concreta
  - Permite un número de ejemplares variable
    - Es posible tener un conjunto de ejemplares en vez de uno sólo: Object Pool
  - Más flexible que las operaciones de clase (static)
    - Con static no es posible considerar que hubiera más de un solo ejemplar
    - En C++ las funciones static no pueden ser virtuales, y por tanto las subclases no pueden redefinirlas polimórficamente

Singleton

- Implementación
  - Definición de la clase: asegurar que sólo hay un ejemplar

```
class Singleton {
   private static Singleton ejemplar = null;
   public static Singleton getEjemplar() {
            if ( ejemplar == null )
                     ejemplar = new Singleton();
            return ejemplar;
   }
   protected Singleton() {
            // lo que sea necesario
   }
   public void metodo() {...}
}
```

Singleton

- Implementación
  - Utilización:

```
Singleton instance =  Singleton.getEjemplar();
// …
instance.metodo();
```

  - Herencia de la clase Singleton
    - ¿Cómo determinar en *instance()* qué subclase utilizar?
      - Usando variables de entorno
      - Poner la implementación de instance() en las subclases
      - Usando un *registro de singletons*

Discusión

- Hay dos maneras de parametrizar un sistema por las clases de objetos que crea:
  - Usando una subclase de la clase que crea los objetos:
    - Método Factoría
      - Requiere una nueva subclase por cada nuevo tipo de producto
  - Mediante composición de objetos: un objeto es el responsable de conocer la clase de los objetos de producto:
    - Factoría Abstracta, que puede producir objetos de varias clases
    - Builder, que puede construir un producto complejo paso a paso
    - Prototipo, construye un producto copia de otro prototipo
      - Son más flexibles que el Método Factoría pero más complejos de implementar

Patrones estructurales

Patrones estructurales

- Lo fundamental son las relaciones de uso entre objetos
- Se trata de conseguir que cambios en los requisitos de la aplicación no ocasionen cambios en las relaciones entre los objetos
- Las relaciones de uso están determinadas por las ***interfaces*** que soportan los objetos

Patrones estructurales

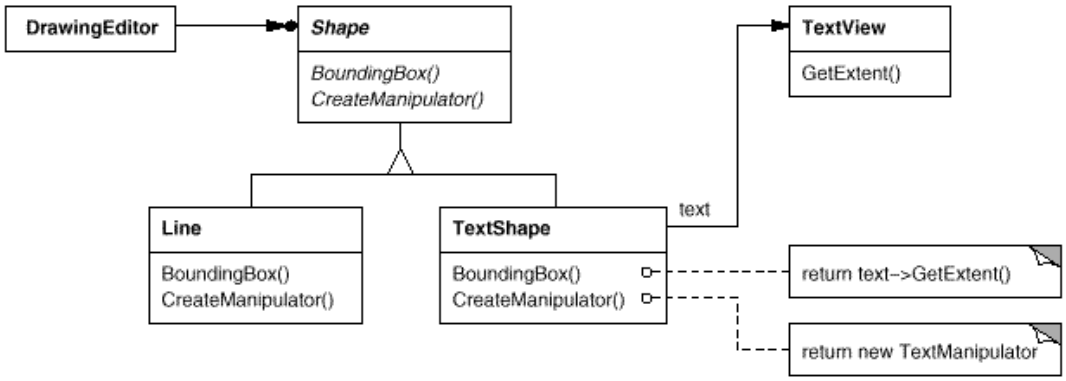
- Tipos de patrones estructurales
  - De clase: usa herencia para componer interfaces o implementaciones
    - Herencia múltiple: una clase que hereda de otras combina sus propiedades
    - Class Adapter
  - De objeto: composición de objetos en tiempo de ejecución
    - Object Adapter
    - Bridge
    - Composite
    - Decorator
    - Facade
    - Flyweight
    - Proxy

Adaptador

- Propósito:
  - Convertir la interfaz de una clase en otra que esperan los clientes
- Otras denominaciones:
  - Class Adapter y Object Adapter
  - Wrapper (Envolvente)
- Motivación
  - Para reutilizar una clase de una biblioteca aunque su interfaz no correspondiera exactamente con el que requiere una aplicación concreta
    - En el ejemplo, la clase TextView
  - Para añadir funcionalidad que la clase reutilizada no proporciona
    - En el ejemplo, la operación CreateManipulator()

Adaptador

- La clase TextShape adapta TextView a la interfaz Shape, de manera que el editor gráfico DrawingEditor puede reutilizar la clase TextView, que de otra manera sería incompatible.
- TextShape se puede hacer de dos maneras:
  1) Heredando la interfaz Shape y la clase TextView (Adaptador de clases)
  2) Componiendo un objeto TextView en un TextShape, que se implementa utilizando el objeto TextView (Adaptador de objeto)

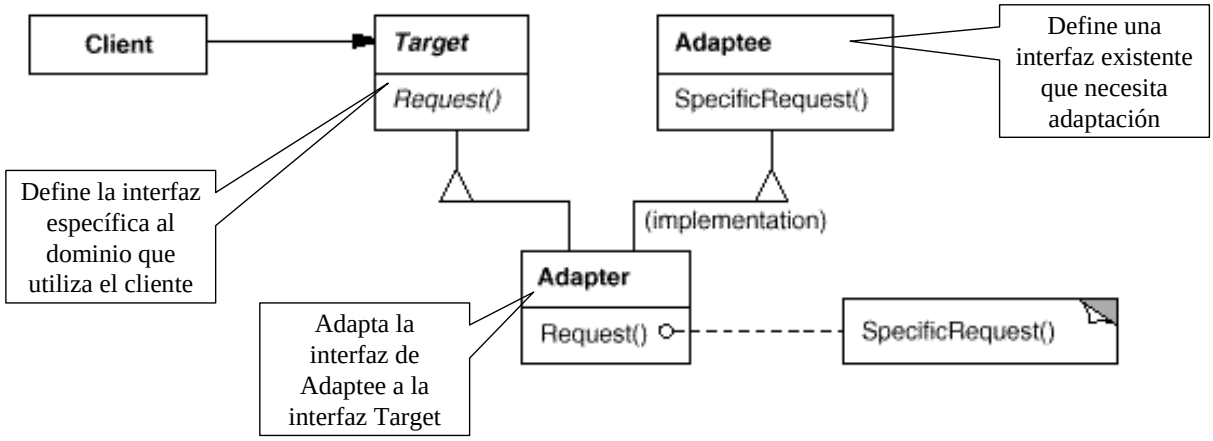


Adaptador

- Aplicación
  - Para usar una clase existente cuya interfaz no se corresponde con el que se necesita
  - Para crear una clase reutilizable que coopera con clases imprevistas (esto es, que no tienen necesariamente interfaces compatibles)
  - El adaptador de objeto sólo: para utilizar varias subclases existentes para las que sería poco práctico adaptar su interfaz heredando de cada una. Un adaptador de objeto puede adaptar la interfaz de su clase padre

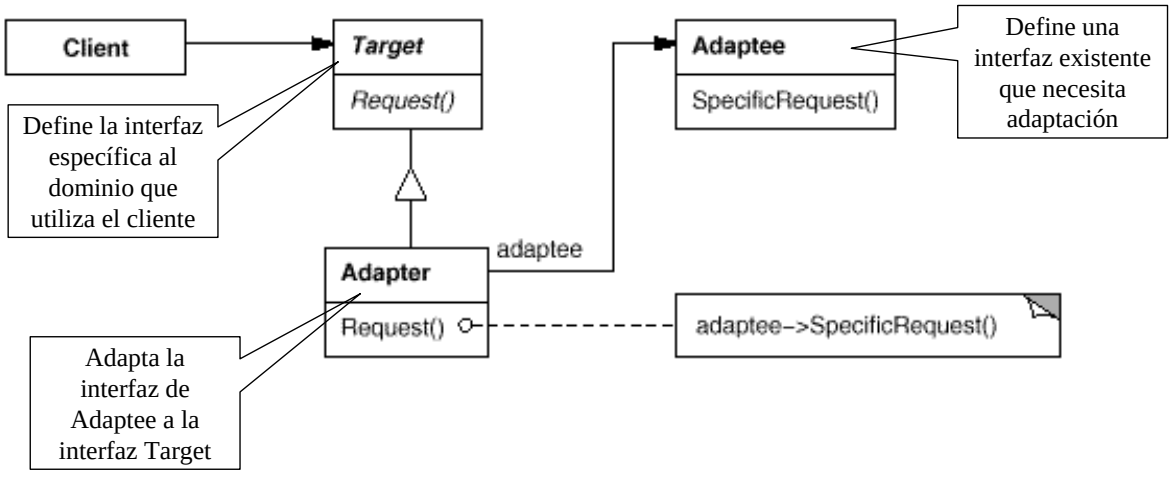
Adaptador

- Esquema, participantes y colaboraciones (*Adaptador de clase*)
  - Los clientes llaman a las operaciones de un objeto Adaptador.
  - A su vez, el Adaptador llama a las operaciones heredadas de la clase Adaptada que tratan la petición



Adaptador

- Esquema, participantes y colaboraciones (*Adaptador de objeto*)
  - Los clientes llaman a las operaciones de un objeto Adaptador.
  - A su vez, el Adaptador llama a las operaciones del Adaptado que tratan la petición



Adaptador

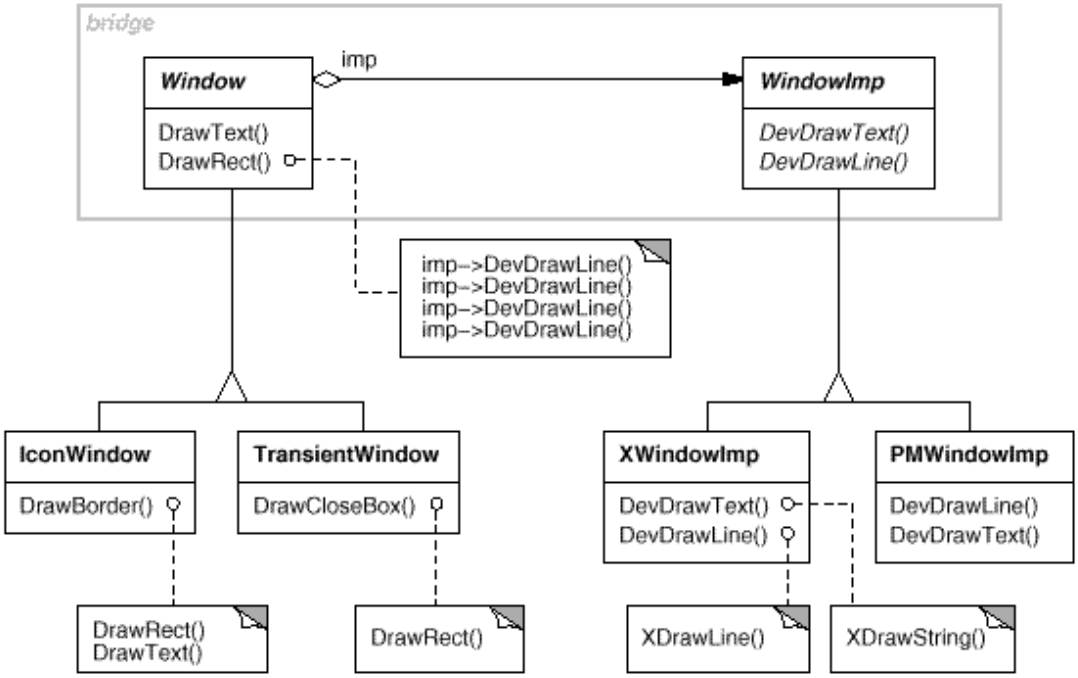
- Consecuencias
  - Un adaptador de clase:
    - Adapta una clase Adaptada a una interfaz Objetivo reutilizando los métodos de la clase Adaptada. Por tanto, no funcionará cuando se quieran adaptar la clase adaptada y todas sus subclases
    - La clase adaptadora puede redefinir algunos de los métodos de la clase adaptada
    - Sólo se introduce un objeto, y no hace falta delegar en otro adaptado
  - Un adaptador de objeto:
    - Permite trabajar un sólo adaptador con muchos adaptados (esto es, de la clase adaptada y sus subclases)
    - Se puede añadir funcionalidad a todos los adaptados de una vez
    - Es más difícil si se necesita redefinir el comportamiento del adaptado
  - ¿Cuanta adaptación hace un adaptador?
    - Desde cambiar el nombre de los métodos hasta añadir nuevas operaciones

Bridge

- Propósito:

  Desacopla una abstracción de su implementación de manera que las dos puedan evolucionar independientemente
- Otras denominaciones:
  - Handle/Body
- Motivación
  - La herencia permite que una abstracción tenga varias implementaciones: esta relación se define en tiempo de compilación
  - Hay casos en los que puede haber una gran jerarquía de clases de una abstracción (p.ej. Window, IconWindow, TransientWindow) y varias jerarquías de implementación (en diferentes plataformas)
    - *Es necesario desacoplar la abstracción de la implementación*

Bridge

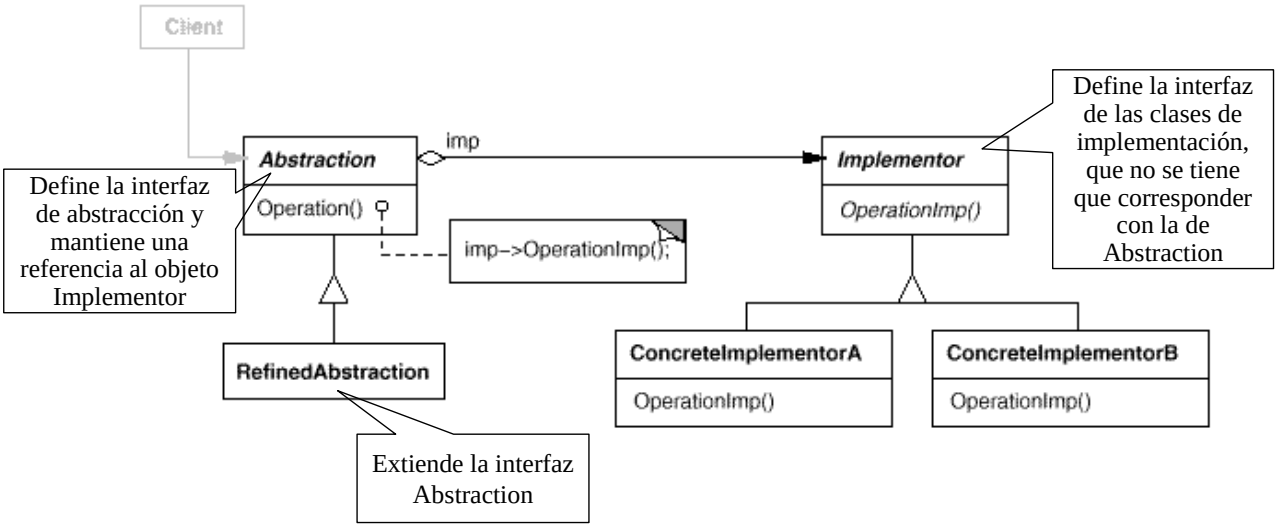


Bridge

- Aplicación
  - Para evitar una vinculación permanente entre una abstracción y su implementación
    - Permite cambiar la implementación en tiempo de ejecución
  - Cuando tanto las abstracciones como las implementaciones podrían ser susceptibles de extensión mediante subclases
    - El patrón Bridge permite combinar las abstracciones e implementaciones y extenderlas independientemente
    - Permite además reducir la proliferación de clases (ver ejemplo de motivación)
  - Para que cambios en la implementación de una abstracción no hagan recompilar el código de los clientes
    - En C++ es más crítico porque la representación de una clase es visible en su definición
  - Para compartir una implementación entre múltiples objetos, sin que lo noten los clientes

Bridge

- Esquema, participantes y colaboraciones
  - La Abstracción pasa las peticiones del cliente al objeto Implementor



Bridge

- Consecuencias
  - Se desacopla interfaz e implementación
    - La implementación no está vinculada permanentemente a la interfaz, y se puede determinar en tiempo de ejecución (incluso cambiar)
    - Se eliminan dependencias de compilación
    - Se consigue una arquitectura más estructurada en niveles
  - Se mejora la extensibilidad
    - Las jerarquías de abstracción y de implementación pueden evolucionar independientemente
  - Se esconden detalles de implementación a los clientes

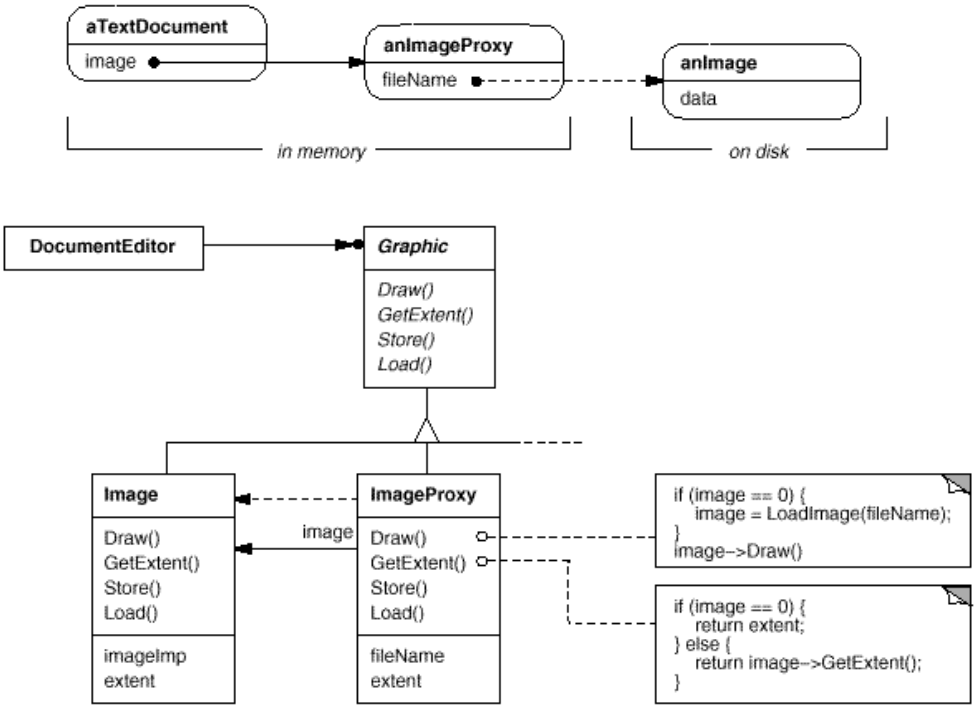
Bridge

- Implementación y patrones relacionados
  - Una sola clase de implementación
    - En este caso no es necesario salvo casos donde no se quiere tener que recompilar el cliente ante cambios de la implementación (en C++ sólo habría que volver a linkar)
  - Creación del objeto de implementación apropiado
    - ¿Cómo, cuándo y dónde se decide qué clase de implementación instanciar?
    - La Factoría Abstracta permite crear y configurar un Bridge particular (y esta factoría puede ser un Singleton): ver ejemplo de código en el GoF
  - Compartir objetos de implementación
  - El patrón Adaptador tiene también el objetivo de hacer trabajar juntas clases con distinta interfaz, pero en general se aplica a sistemas que ya existen. El patrón Bridge suele aplicarse al empezar un diseño, para permitir que las abstracciones e implementaciones evolucionen independientemente

Proxy

- Propósito:
  - Es un sustituto de otro objeto, para controlar el acceso a él
- Otras denominaciones:
  - Surrogate
  - Virtual proxy
- Motivación
  - Puede haber ocasiones en que se desee posponer el coste de la creación de un objeto hasta que sea necesario usarlo
    - Al abrir un documento, postergar el crear una imagen hasta que se tenga que visualizar
    - En programas de comunicaciones, para un objeto remoto
  - El objeto proxy actúa en lugar del verdadero objeto, y ofrece las misma interfaz, y las solicita en el objeto cuando es necesario
    - Por ejemplo cuando el editor del documento invoca la operación Draw en la imagen para visualizarla

Proxy

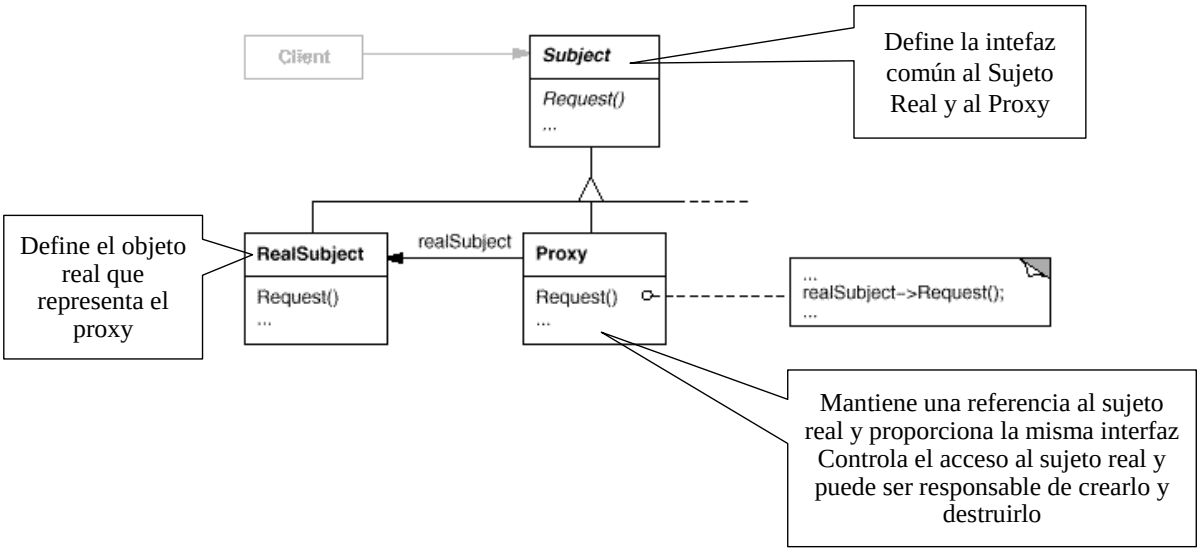


Proxy

- Aplicación: cuando es necesario una referencia a un objeto más sofisticada que el simple puntero o referencia a objeto
  - Proxy remoto, cuando el objeto está en otro espacio de memoria (proceso)
    - Stubs en RPC y CORBA
  - Proxy virtual, para postergar la creación de objetos caros hasta el momento en que se necesitan
    - Imágenes en documentos
  - Proxy de protección, para controlar el acceso al objeto original
    - Controla derechos de acceso diferentes
  - Referencias inteligentes, remplazan a una referencia o puntero para realizar alguna acción adicional:
    - Contar el número de referencias a un objeto para liberarlo cuando nadie lo usa
    - Cargar un objeto persistente en memoria cuando se referencia por primera vez
    - Sincronizar el acceso a un objeto con un cerrojo

Proxy

- Esquema, participantes y colaboraciones
  - El Proxy pasa las peticiones al RealSubject cuando sea apropiado, dependiendo del tipo de proxy



Proxy

- Consecuencias
  - Introduce un nivel de indirección al acceder a un objeto
    - Este nivel de indirección tiene usos distintos dependiendo del tipo de proxy:
    - Un proxy remoto puede ocultar dónde está el objeto real
    - Un proxy virtual puede mejorar la eficiencia por ejemplo al crear un objeto bajo demanda
    - Tanto los proxies de protección como las referencias inteligentes realizan tareas de gestión interna cuando se accede al objeto
  - Permite implementar de manera eficiente copy-on-write
    - Un objeto costoso sólo se copia si se ha modificado

Proxy

- Implementación y patrones relacionados
  - Un Proxy no tiene que conocer el tipo de un sujeto real
    - Puede acceder a él a través de la interfaz abstracta
  - Se pueden utilizar facilidades de los lenguajes para implementarlo:
    - En C++ se puede sobrecargar el operador de acceso a un miembro (->). Así cuando se accede a un miembro se pueden hacer acciones adicionales
      - Esto es válido sólo cuando no es necesario distinguir el tipo de operación que se accede en el objeto
  - El patrón Proxy se puede ver como un caso particular de Bridge:
    - un Proxy sólo tiene una implementación, y un Bridge puede tener más de una
    - un Proxy se suele usar para controlar el acceso a su implementación, el Bridge permite cambiar una implementación dinámicamente

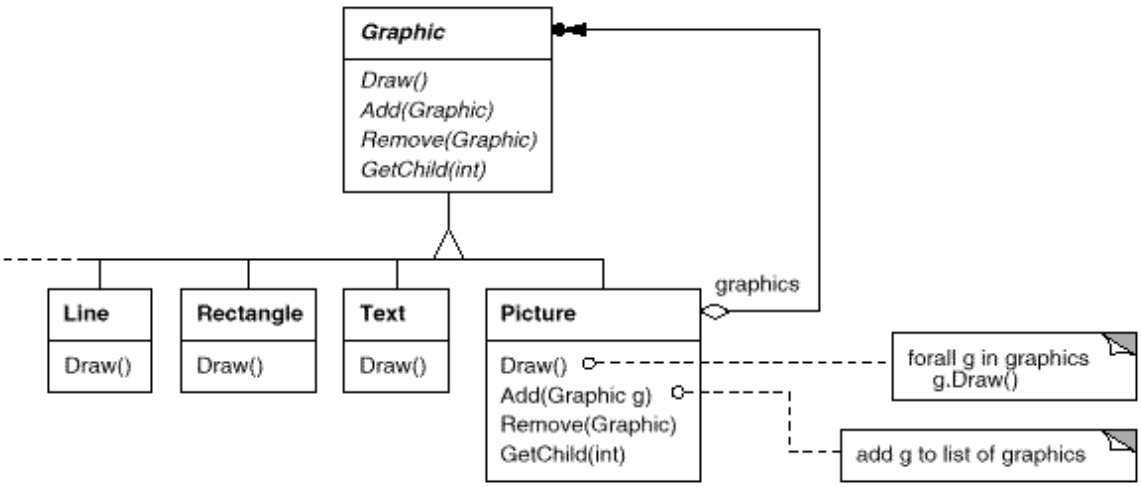
Proxy

- Implementación y patrones relacionados
  - El Adaptador proporciona una interfaz diferente al objeto que adapta, pero el proxy tiene la misma interfaz
    - Aunque el proxy puede rehusar realizar una operación (así su interfaz puede verse como un subconjunto)
  - El Decorador se puede implementar de manera similar al Proxy pero el propósito es diferente: el decorador añade responsabilidades a un objeto, el proxy sólo controla su acceso

Composite

- Propósito:
  - Construir objetos complejos mediante la composición recursiva de objetos similares de manera similar a un árbol
- Motivación
  - Las aplicaciones gráficas tienen componentes que pueden agruparse para formar componentes mayores (contenedores)

Composite

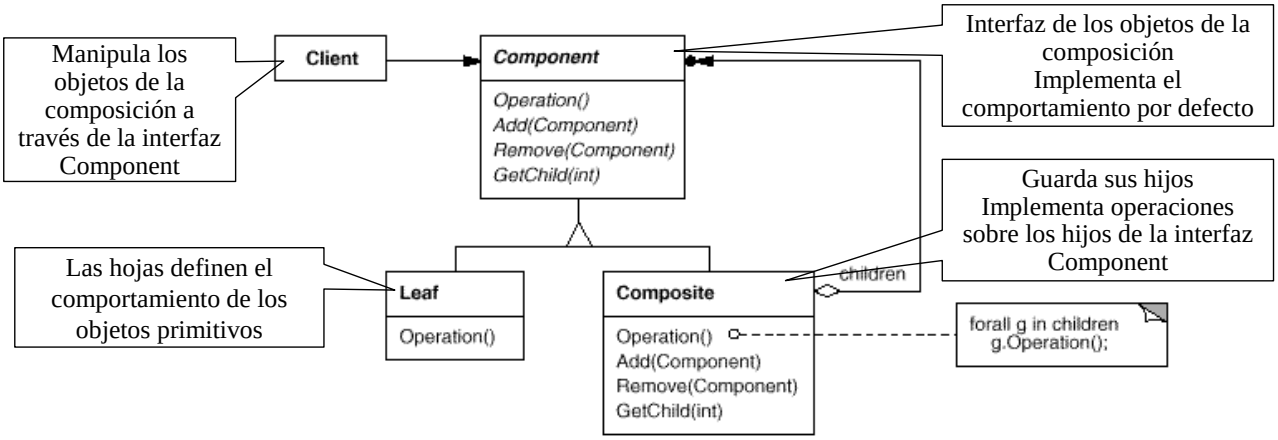


Composite

- Aplicación
  - Para representar jerarquías de objetos parte-todo
  - Para que los clientes puedan manejar indistintamente objetos individuales o composiciones de objetos
    - Los clientes tratan todos los objetos en la estructura composite de manera uniforme

Composite

- Esquema, participantes y colaboraciones
  - Los clientes usan la clase Componente para interaccionar con los objetos de la estructura composite. Si el recipiente es una Hoja la petición se maneja directamente. Si se trata de un Composite, se pasa a sus componentes hijos, pudiéndose realizar operaciones adicionales antes o después



Composite

- Consecuencias
  - Define jerarquías de clases que tienen objetos primitivos y objetos compuestos (composite)
    - La composición puede ser recursiva
  - Hace el cliente simple
    - Puede tratar la estructura y los objetos individuales uniformemente
  - Facilita la adición de nuevas clases de componentes
  - Puede hacer que el diseño sea demasiado general
    - Hace más difícil restringir los componentes de un composite
    - Si se quiere hacer que un composite sólo tenga ciertos componentes hay que codificar las comprobaciones para que se realicen en tiempo de ejecución

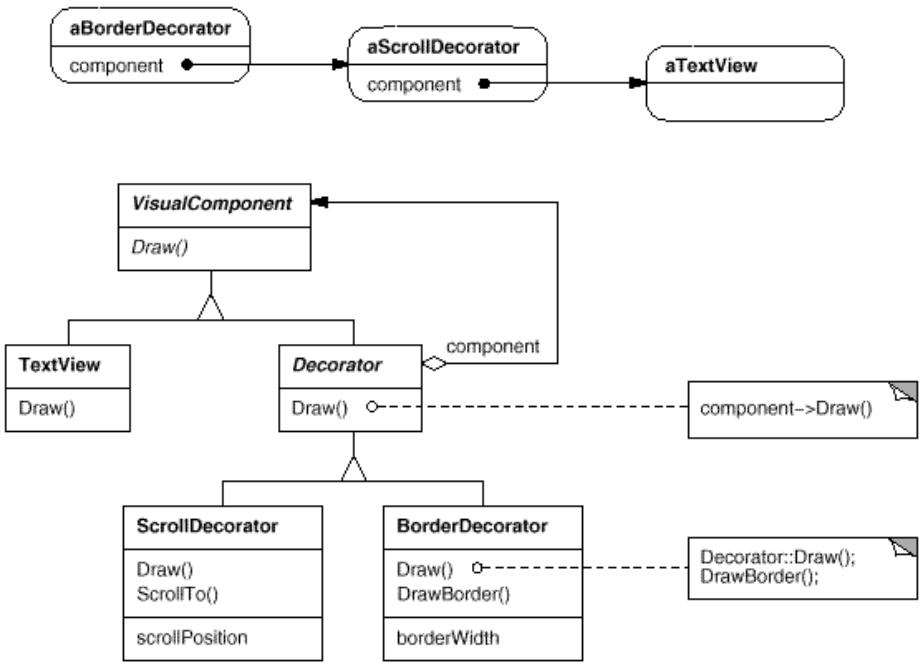
Composite

- Implementación y patrones relacionados
  - Referencias explícitas al padre
  - Compartición de componentes
  - Maximizar la interfaz Component
  - Operaciones de gestión de hijos
  - Debe implementar Component una lista de Components?
  - Orden de los hijos
  - Mejora de la eficiencia con caching
  - ¿Quién debe eliminar los componentes?
  - ¿Cuál es la mejor estructura de datos para almacenar los componentes?
  - Los Iteradores se pueden usar para recorrer composites
  - El Visitor localiza operaciones y comportamiento que de otra manera se distribuiría entre las hojas y el composite

Decorador

- Propósito:
  - Asignar nuevas responsabilidades a un objeto dinámicamente. Es una alternativa a la creación de subclases por herencia
- Otras denominaciones:
  - Decorator
  - Wrapper
- Motivación
  - Para añadir nuevas responsabilidades a objetos individuales y no a toda la clase
    - Por ejemplo, una ventana de una interfaz gráfica puede tener bordes, scrolling, u otros artifactos
  - Esto se puede hacer dinámicamente

Decorador

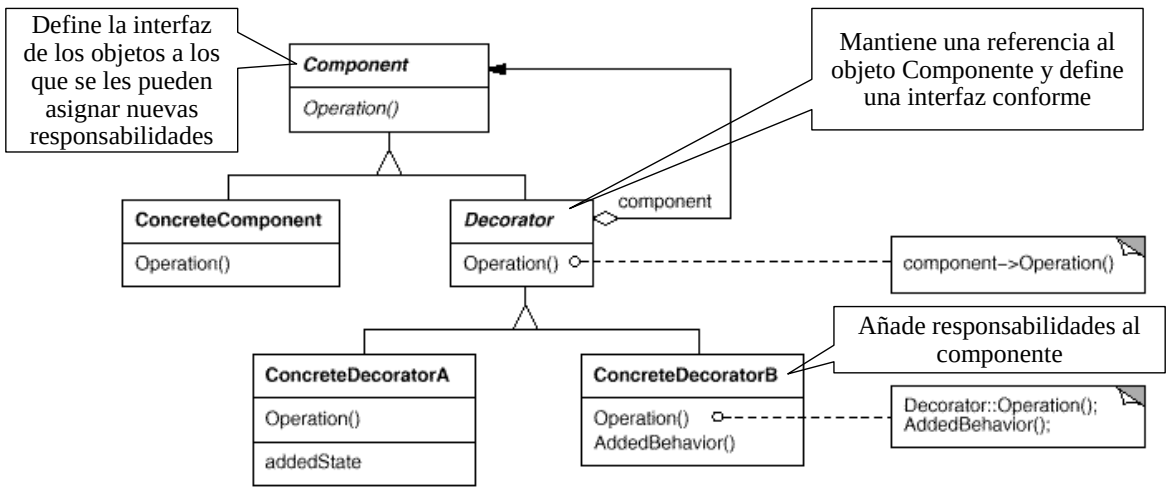


Decorador

- Aplicación
  - Para asignar responsabilidades a objetos individuales de forma dinámica y transparente, sin afectar a otros objetos
  - Para responsabilidades que pueden ser suprimidas
  - Cuando no es práctico utilizar herencia de clases (porque se produciría una explosión del número de clases, o porque no se tiene la definición de la clase)

Decorador

- Esquema, participantes y colaboraciones
  - El Decorador envía las peticiones a su objeto Componente, y puede realizar alguna operación adicional antes o después



Decorador

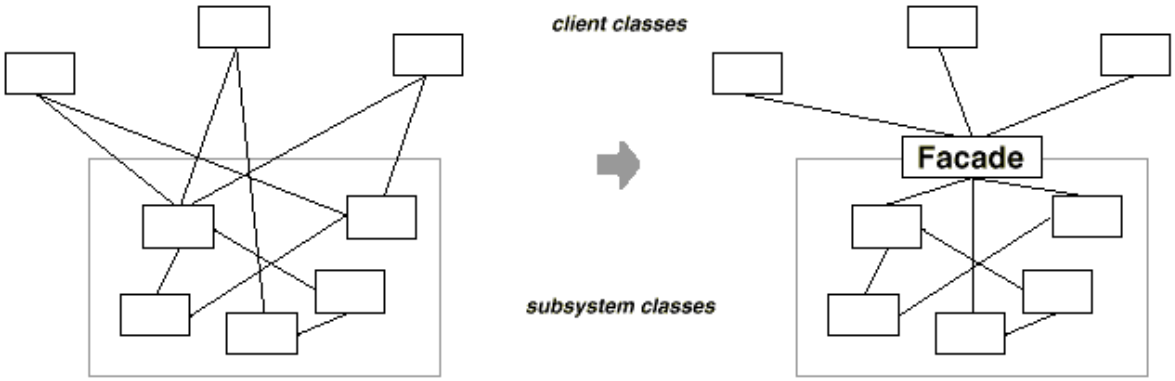
- Consecuencias
  - Más flexibilidad que la herencia de clases (estática)
    - Las responsabilidades se añaden y quitan dinámicamente (en tiempo de ejecución)
    - Evita la explosión de clases
    - Permiten añadir una propiedad más de una vez
      - Por ejemplo, añadir doble borde a una ventana
  - Se puede definir una clase sencilla y añadirle nuevas características mediante decoradores, pudiendo así obtener gran funcionalidad combinando piezas sencillas
    - Sólo se incorpora lo que se usa
  - Muchos componentes pequeños
    - Al usar Decoradores, el sistema tiene muchos objetos pequeños
    - Puede ser difícil de entender y depurar

Decorador

- Implementación y patrones relacionados
  - Conformidad de interfaces
    - La interfaz de un objeto decorador debe ser conforme con la del componente que decora, por tanto un objeto decorador concreto debe heredar de una clase común
  - Omisión de la clase abstracta Decorador
    - Si sólo se va a definir una nueva responsabilidad no hace falta definir la clase abstracta Decorador
  - Las clases Componente deben ser ligeras
    - La definición de los datos internos de la clase se debe dejar a las implementaciones
  - Decorador vs. Strategy: el Decorador puede actuar antes o después de llamar a los métodos de otro objeto, para cambiar lo que se hace en medio de la llamada a un método se usa el patrón Strategy
  - Es distinto del Adaptador ya que sólo añade responsabilidades a un objeto, no su interfaz
  - Puede verse como un caso degenerado de Composite con un solo componente. Sin embargo, el decorador añade responsabilidades

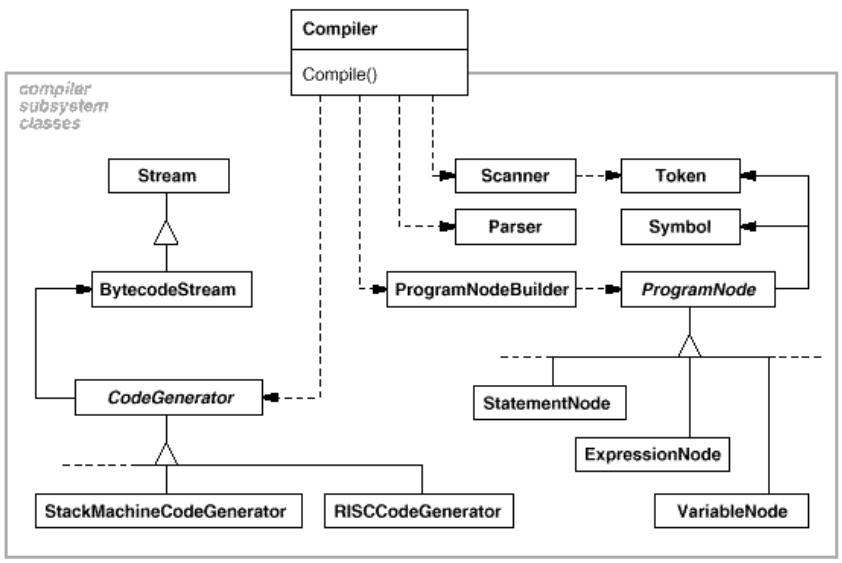
Fachada

- Propósito:
  - Simplifica el acceso a un conjunto de objetos proporcionando uno que todos los clientes pueden usar para comunicarse con el conjunto
- Otras denominaciones:
  - Façade
- Motivación
  - Minimizar las comunicaciones y dependencias entre subsistemas



Fachada

- El subsistema de compilación ofrece scanner, parser, ... a través de una interfaz unificada (la fachada) de la funcionalidad del compilador, sin ocultar completamente las clases que lo implementan

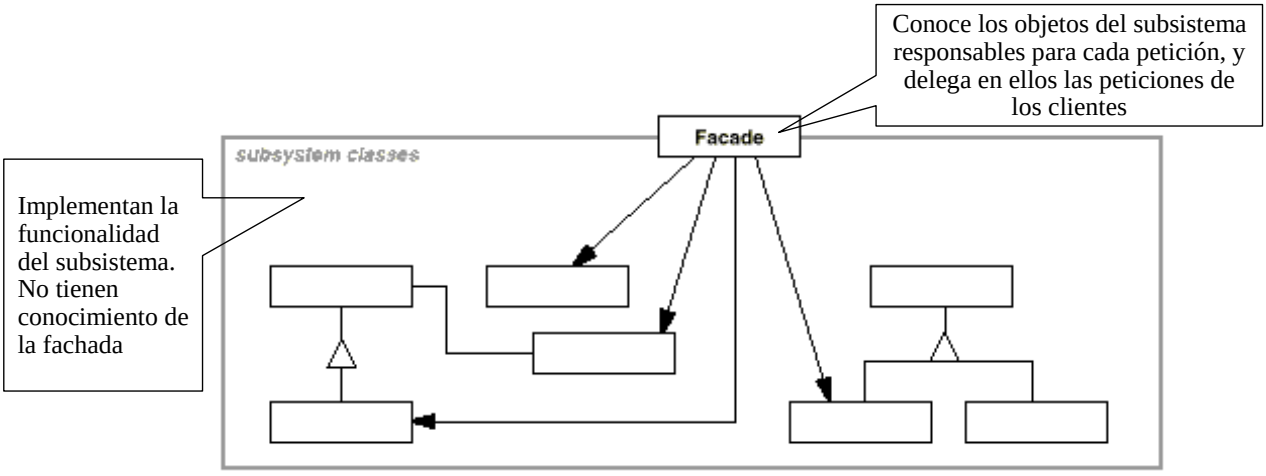


Fachada

- Aplicación
  - Para proporcionar una interfaz sencilla a un subsistema complejo
    - A medida que un subsistema evoluciona va teniendo más clases, más pequeñas, más flexibles y configurables
    - Hay clientes que no necesitan tanta flexibilidad y que quieren una visión más simple del subsistema
    - Sólo los clientes que necesiten detalles de más bajo nivel accederán a las clases detrás de la fachada
  - Cuando hay muchas dependencias entre los clientes y las clases de implementación de una abstracción
    - La fachada desacopla el subsistema de los clientes y de otros subsistemas
    - Esto mejora la independencia de subsistemas y la portabilidad
  - Para estructurar un sistema en capas
    - La fachada define el punto de entrada de cada nivel
    - Se pueden simplificar las dependencias obligando a los subsistemas a comunicarse únicamente a través de sus fachadas

Fachada

- Esquema, participantes y colaboraciones
  - Los clientes se comunican con el subsistema haciendo peticiones a la Fachada, que las envía a los objetos del subsistema apropiados (la fachada podría también traducir su interfaz a la de las interfaces del subsistema)
  - Los clientes que usan la fachada no tienen que acceder a los objetos del subsistema directamente



Fachada

- Consecuencias
  - Oculta a los clientes los componentes del subsistema
    - Reduce el número de objetos con los que tienen que tratar los clientes
  - Disminuye el acoplamiento entre un subsistema y sus clientes
    - Un menor acoplamiento facilita el cambio de los componentes del subsistema sin afectar a sus clientes
    - Las fachadas permiten estructurar el sistema en capas
    - Reduce las dependencias de compilación
  - No evita que las aplicaciones puedan usar las clases del subsistema si lo necesitan
    - Se puede elegir entre facilidad de uso y generalidad

Fachada

- Implementación y patrones relacionados
  - Reducción del acoplamiento cliente-subsistema
    - Se puede reducir más el acoplamiento haciendo la fachada una clase abstracta con subclases concretas para las distintas implementaciones del subsistema. Los clientes se comunican con el subsistema usando la interfaz de la clase fachada abstracta
    - Otra posibilidad es configurar el objeto fachada con diferentes objetos del subsistema. Para personalizar la fachada basta con reemplazar uno o varios de sus objetos del subsistema
    - La factoría abstracta se puede utilizar junto a la fachada para crear objetos del subsistema de manera independiente al subsistema
  - Clases del subsistema privadas y públicas
    - En Java se puede utilizar los paquetes para determinar las clases que son visibles fuera o no. En C++ los namespaces no soportan la ocultación de clases
  - Normalmente sólo hace falta un objeto Fachada, por lo cual suele implementarse como Singleton

Discusión

- Muchos patrones estructurales tienen bastante similitud en el esquema, los participantes y las colaboraciones
- Las diferencias están principalmente en el propósito
  - Adaptador y Bridge
    - Ambos envían peticiones a un objeto cuya interfaz es distinta
    - El adaptador trata de resolver incompatibilidades entre interfaces existentes, bridge trata de proporcionar una interfaz estable a los clientes, independiente de cómo evolucione la implementación
    - Por tanto el Adaptador suele aparecer cuando ya hay clases desarrolladas, pero el Bridge es algo que el diseñador avanza al principio del ciclo de vida del sistema

Discusión

- Las diferencias están principalmente en el propósito
  - Composite y Decorator
    - Ambos se basan en la composición recursiva para organizar un conjunto de objetos ilimitado
    - El Decorador pretende facilitar el añadir responsabilidades a los objetos sin hacer subclases. Composite trata de que múltiples objetos se puedan tratar de manera uniforme, como uno (el énfasis está en la representación y no en el embellecimiento)
    - Los propósitos son complementarios y por tanto se suelen usar los dos patrones conjuntamente
  - Proxy
    - Similar estructura que Decorator
    - Especialización de Bridge

Patrones de comportamiento

Patrones de comportamiento

- Algoritmos y asignación de responsabilidades entre objetos y patrones de comunicación entre objetos
- Distribución del comportamiento entre varias clases
  - Mediante herencia
    - Método Template: definición abstracta de un algoritmo
    - Intérprete: representa una gramática como una jerarquía de clases
  - Mediante composición de clases
    - Mediador: entre varios objetos para desacoplarlos
    - Cadena de responsabilidad
    - Observador (MVC)
- Encapsulado de comportamientos
    - Estrategia: encapsula un algoritmo en un objeto
    - Comando: encapsula una petición a un objeto
    - Estado: encapsula el estado de un objeto
    - Visitor: encapsula un comportamiento
    - Iterador: el método de recorrer una colección de clases

Patrones de comportamiento

- Chain of responsability
  - Petición delegada al proveedor de servicio responsable
- Command
  - Encapsula una petición como un objeto
- Interpreter
  - Intérprete de lenguaje para una gramática sencilla
- Iterator
  - Para acceder secuencialmente a elementos de una agregación
- Mediator
  - El mediador coordina las interacciones entre sus asociados
- Memento
  - Captura el estado de un objeto y lo restaura posteriormente

Patrones de comportamiento

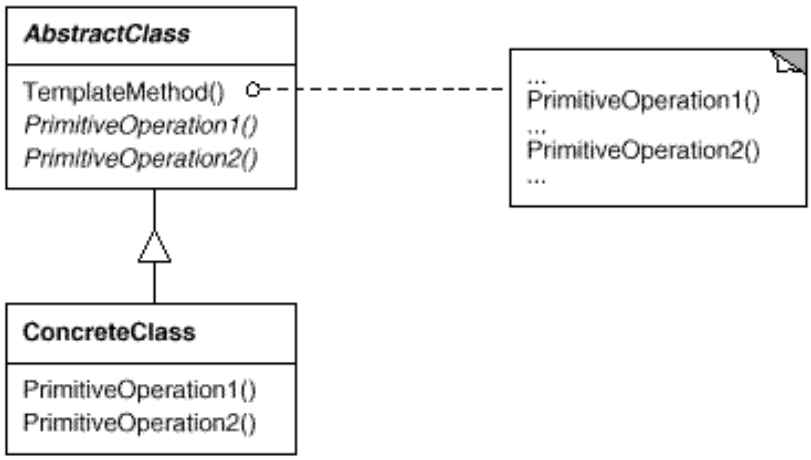
- Observer
  - Para notificar automáticamente cambios de estado de un objeto a todos los dependientes
- State
  - Objeto cuyo comportamiento depende de un estado (el objeto puede aparentar cambiar de clase según un estado)
- Strategy
  - Abstracción para seleccionar uno o varios algoritmos
- Template method
  - Algoritmo con varios pasos suministrados por una clase derivada
- Visitor
  - Operaciones aplicadas a elementos de una estructura de objetos heterogénea

Método Template

- Propósito:
  - Permite construir el esqueleto de un algoritmo dejando los detalles a las subclases
- Aplicación
  - Permite escribir las partes fijas de un algoritmo una sola vez y definir las partes que varían en las subclases
  - Refactorización de código
    - Identificar las diferencias en el código existente
    - Separar las diferencias en nuevas operaciones
    - Reemplazar el código diferente con un método template que llama a esas operaciones nuevas
  - Define los puntos de extensión de las subclases
    - Útil en frameworks: indica en qué puntos se puede extender

Método Template

- Esquema, participantes y colaboraciones
  - La clase abstracta define las operaciones primitivas (pasos del algoritmo) que deben implementar las subclases concretas
  - El método template define el esqueleto del algoritmo

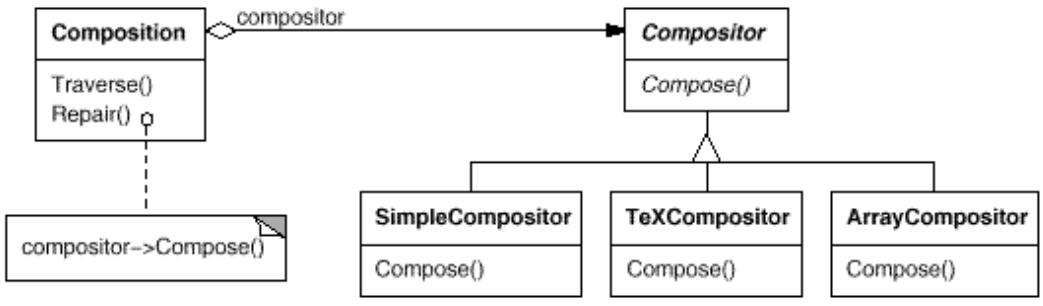


Método Template

- Consecuencias
  - Ayuda a la reutilización de código
    - Por ejemplo, factorizando comportamiento en bibliotecas de clases
  - Soporte de frameworks: invierte el flujo de control
    - Los métodos que nos deja implementar son llamados por el framework
  - El método template puede llamar a:
    - Operaciones concretas (de la clase concreta o de las clientes)
    - Operaciones concretas de la clase abstracta (operaciones útiles para las subclases)
    - Operaciones primitivas (pasos del algoritmo)
    - Métodos factoría
    - Operaciones Hook, que proporcionan el comportamiento por defecto que las subclases pueden extender si fuera necesario (por defecto no suelen hacer nada)

Estrategia

- Propósito:
  - Permite definir una familia de algoritmos, encapsulándolos de manera que se pueda intercambiar uno por otro
- Otras denominaciones:
  - *Policy* (políticas)
- Motivación
  - En un procesador de textos, una clase Composition se encarga de gestionar la división de líneas de un texto y para ello delega en Compositor el algoritmo de división (por líneas, por párrafos, o por un número fijo de caracteres)

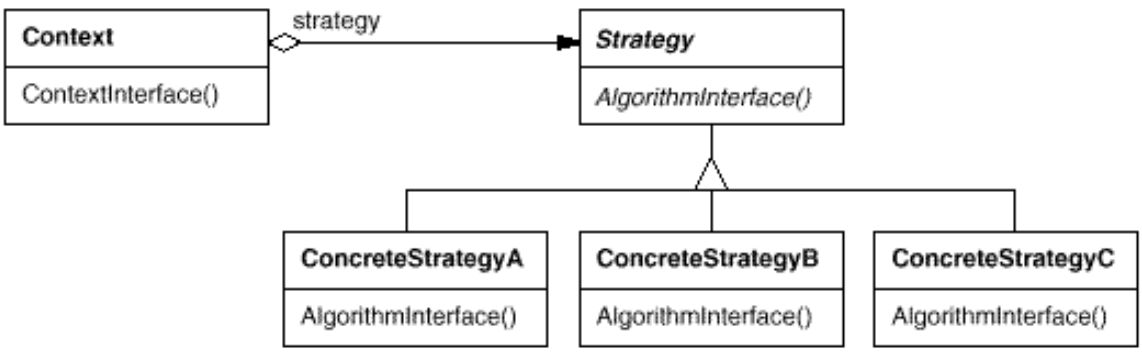


Estrategia

- Aplicación
  - Cuando hay muchas clases similares que sólo difieren en algún comportamiento
    - Se puede configurar una clase con varios comportamientos
  - Se necesitan variantes de un algoritmo
  - Para ocultar estructuras de datos complejas que usa el algoritmo
  - En vez de utilizar muchas instrucciones condicionales poner los comportamientos alternativos en clases estrategia concretas

Estrategia

- Esquema, participantes y colaboraciones
  - La estrategia y el contexto colaboran para implementar un algoritmo concreto:
    - La clase Strategy declara la interfaz común a todos los algoritmos
    - El contexto se configura con una estrategia concreta y mantiene la referencia correspondiente
    - El contexto puede definir una interfaz para que la clase estrategia acceda a sus datos (y le pasará a la estrategia su referencia) o bien le pasa todos los datos necesarios en las operaciones



Estrategia

- Consecuencias
  - Permite gestionar familias de algoritmos
    - Se puede definir una jerarquía de clases de algoritmos
      - Sacar factor común del código
      - Se comprende mejor la relación entre los algoritmos de la familia
  - Elimina instrucciones condicionales
  - Permite definir varias implementaciones del mismo comportamiento
  - Como inconveniente, los clientes tienen que conocer las distintas estrategias para elegir la más apropiada

Discusión

- Template Method vs. Strategy
  - El patrón método template utiliza herencia para variar parte de un algoritmo
  - El patrón estrategia utiliza delegación para variar todo el algoritmo
- Los métodos factoría se llaman frecuentemente por los métodos template

Bibliografía

- Gamma E., Helm R., Johnson R., Vlissides J., ; *Design Patterns. Elements of Reusable Object-Oriented Software;* Addison-Wesley, 1995. Traducción al castellano (2002): Diseño de patrones. Pearson Educación
- James W. Cooper, *The Design Patterns Java Companion*. Addison Wesley, 1998
- Steven J. Metsker, *Design Patterns Java Workbook*, Addison Wesley, 2002