Ingeniería del Software

Tema 6: Diseño orientado a objetos

Dr. Francisco José García Peñalvo
(fgarcia@usal.es)
Miguel Ángel Conde González
(mconde@usal.es)
Sergio Bravo Martín
(ser@usal.es)

3º I.T.I.S.
Fecha de última modificación: 16-10-2008

Universidad de Salamanca – Departamento de Informática y Automática

Resumen

Resumen	Este tema introduce el diseño orientado a objetos, incidiendo en tres aspectos como son la disciplina de diseño dentro del Proceso Unificado, el diseño de la arquitectura del software, destacando la utilización de un patrón Capas para estructurar la arquitectura de los sistemas, y, por último, se introducen los patrones de diseño, tomando como referencias principales los patrones de GoF (*Gang of Four*) [Gamma et al., 1995] y los patrones POSA (*Pattern Oriented Software Architecture*) [Buschmann et al., 1996], aunque también se hará mención a los patrones GRASP (*General Responsibility Assignment Software Patterns*) [Larman, 2002]
Descriptores	Diseño orientado a objetos; Diseño arquitectónico; Proceso Unificado; Subsistema de diseño; Clase de diseño; Modelo de despliegue; Arquitectura del software; Patrón software; Patrón de diseño; Responsabilidad
Bibliografía	[Gamma et al., 2003] [Jacobson et al., 2000] Capítulo 9 [Larman, 2003] Capítulos 16 y 22 [Sommerville, 2005] Capítulo 14

Esquema

- Introducción
- Diseño en el Proceso Unificado
- Diseño de la arquitectura
- Patrones de diseño orientado a objetos
- Aportaciones principales del tema
- Ejercicios
- Lecturas complementarias
- Referencias

1. Introducción

Definición de DOO

> DOO es el proceso que modela el dominio de la solución, lo que incluye a las clases semánticas con posibles añadidos, y las clases de interfaz, aplicación y utilidad identificadas durante el diseño
>
> [Monarchi y Puhr, 1992]

- El **objetivo** es reexaminar las clases del dominio del problema, refinándolas, extendiéndolas y reorganizándolas, para mejorar su reutilización y tomar ventaja de la herencia
- El diseño casa con el dominio de la solución
- Los objetos del dominio de la solución incluyen: objetos de interfaz, objetos de aplicación y objetos base o de utilidad
- El énfasis del diseño está en **DEFINIR LA SOLUCIÓN**

Generalidades (i)

- Construye los productos desarrollados en el AOO
  - Refinamiento de las clases
  - Definición de los protocolos de los mensajes para todos los objetos
  - Definición de estructuras de datos y procedimientos
- Los objetos y relaciones identificadas en la fase de análisis sirven de entrada a la fase de diseño y como primera capa de diseño
- El sistema se concibe como una colección de objetos que interaccionan
- El estado del sistema está descentralizado y cada objeto gestiona su propio estado
- Los objetos son instancias de una clase de objetos y se comunican invocando métodos

Generalidades (ii)

- Diseño de componentes software individuales
  - Objetivo: Representar un concepto que estará en forma ejecutable
  - Basado en objetos: ADT
  - Orientado a objetos: Clase
- Idealmente una clase es una implementación de un ADT
  - Detalles de implementación son privados a la clase
  - Interfaz pública
    - Funciones de acceso: Devuelven abstracciones significativas acerca del estado de las instancias
    - Procedimientos de abstracción: Utilizados para cambiar el estado de una instancia
- Herencia
  - Extensibilidad del sistema
  - Refleja la abstracción y estructura presente en un dominio de la aplicación
  - Identifica y encapsula elementos comunes en abstracciones de alto nivel

Modelo de análisis vs. Modelo de diseño

- Modelo Conceptual
- Genérico -> Diseño
- Menos formal
- Menos caro de desarrollar
- Menos capas
- Centrado en las interacciones
- Bosquejo del diseño
- Puede no necesitar mantenimiento

- Modelo Físico
- Específico -> Implementación
- Más formal
- Más caro de desarrollar
- Más capas
- Centrado en la secuencia
- Manifiesto del diseño
- Necesidad de mantenimiento a lo largo de todo el ciclo de vida

Un proceso de DOO simple

- Comprender y definir el contexto y los modos de utilización del sistema
- Diseñar la arquitectura del sistema
- Identificar los objetos principales del sistema
- Desarrollar los modelos de diseño
- Especificar las interfaces de los objetos

[Sommerville, 2005]

Fases del diseño (i)

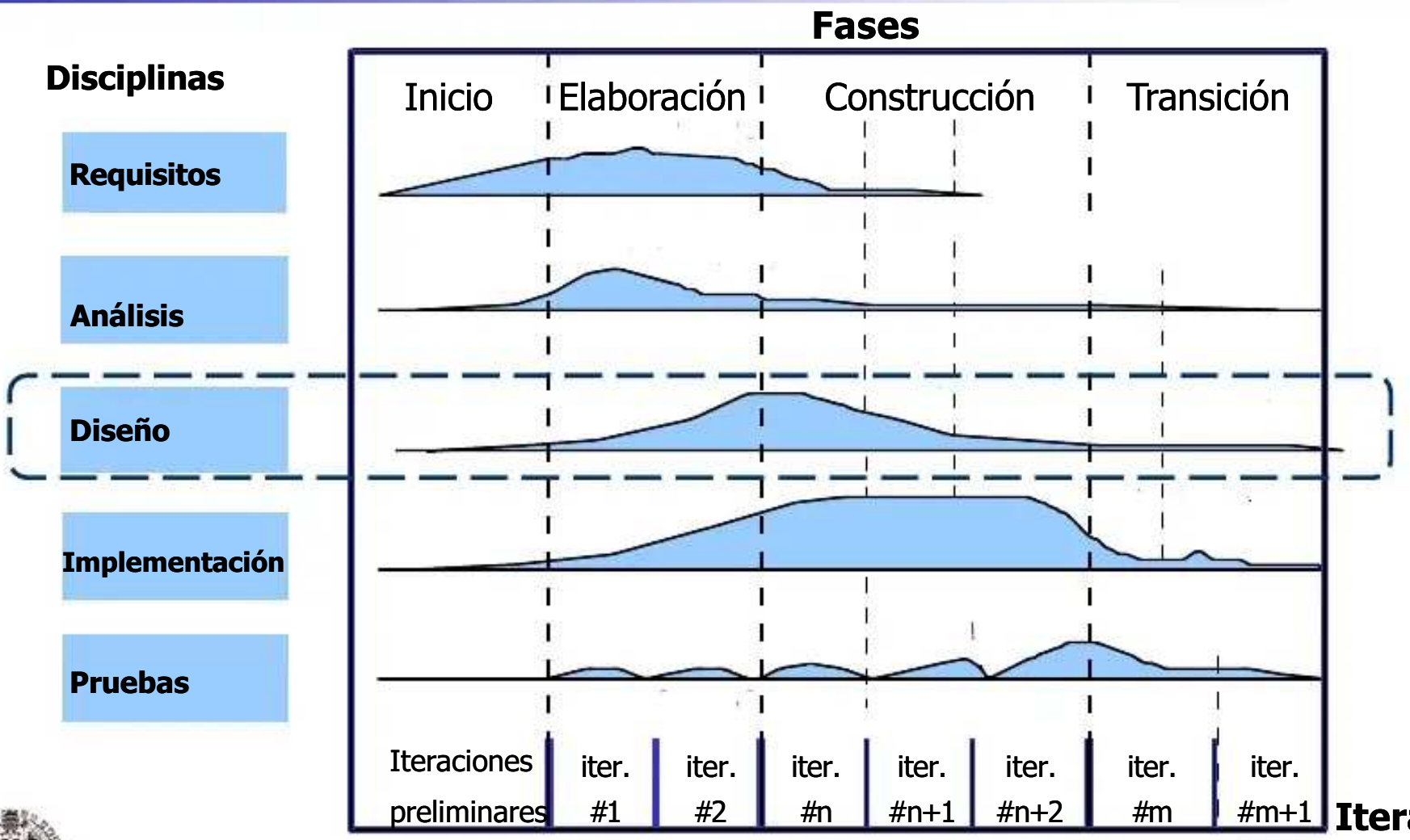
- **Diseño arquitectónico**
  - Identificación y documentación de subsistemas que conforman el sistema completo
  - Identificación y documentación de las relaciones entre subsistemas
    - Se describe la arquitectura de alto nivel (estructura y organización)
  - Definición de las relaciones entre los elementos estructurales principales del software
  - Desarrollo de una estructura modular y de la representación del control entre módulos
  - Adaptación de la estructura lógica del Modelo de Análisis al entorno de implementación y preparación para su implementación
  - Incorporación de los requisitos no funcionales

Fases del diseño (ii)

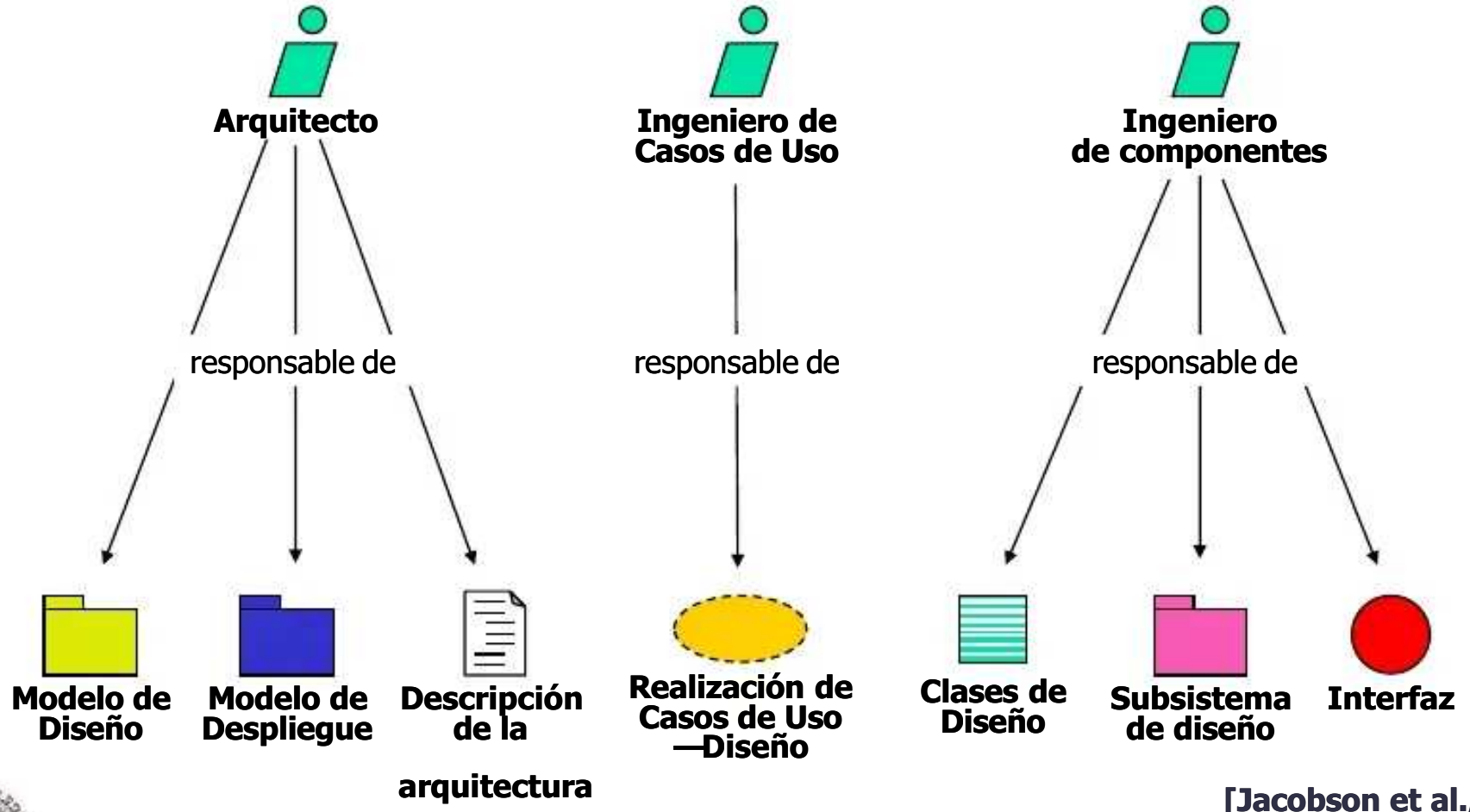
- **Especificaciones abstractas**
  - Para cada subsistema se produce una especificación abstracta de los servicios proporcionados y las restricciones bajo las que tiene que operar
- **Diseño de interfaces**
  - Diseño y documentación de las interfaces entre subsistemas
  - Diseño y documentación de la interfaz entre el sistema software y el usuario
- **Diseño detallado**
  - Asignar servicios a los diferentes componentes y diseñar sus interfaces
  - Detallar cada componente a un nivel suficiente para su codificación

2. Diseño en el Proceso Unificado

Disciplina de diseño



Trabajadores y artefactos involucrados en el diseño



[Jacobson et al., 1999]

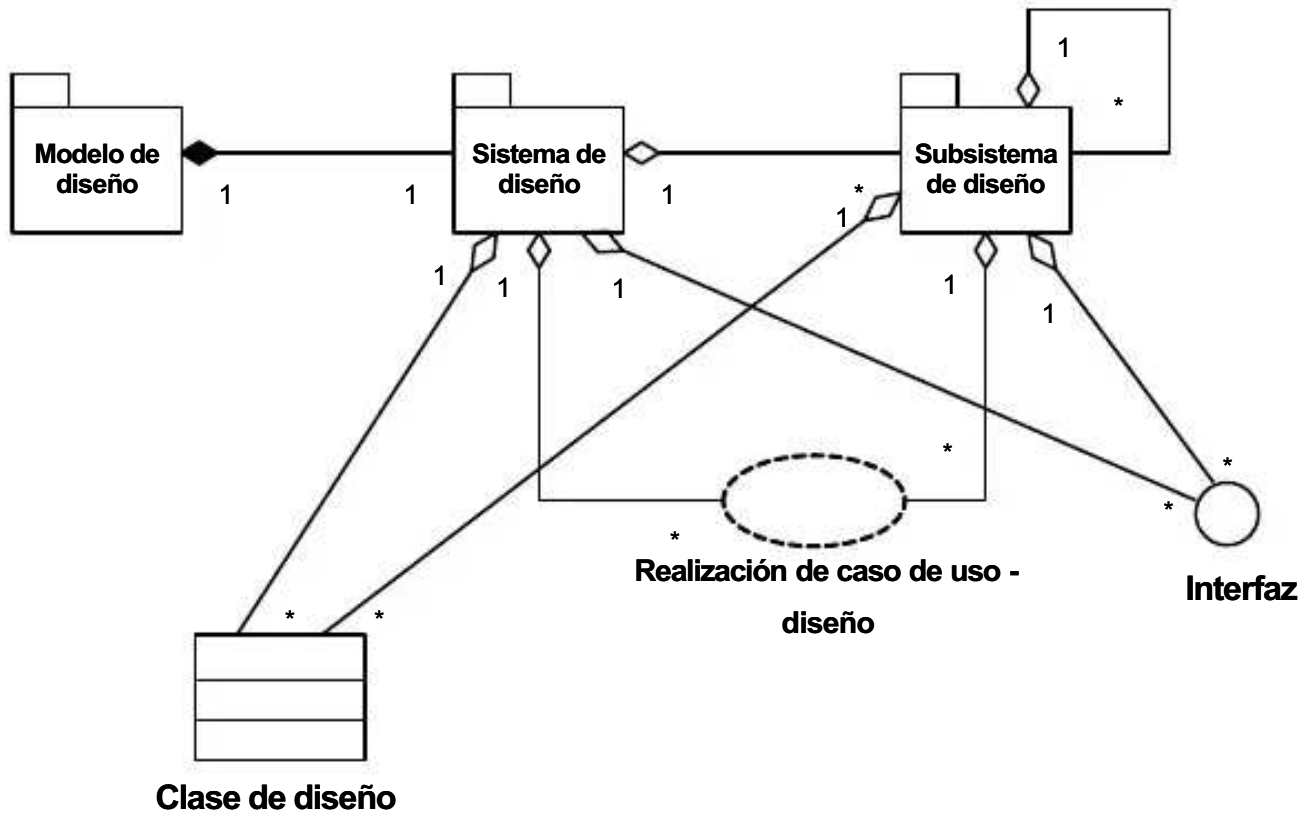
Artefactos propios del diseño en el Proceso Unificado

- Modelo de diseño
- Clase de diseño
- Realización de casos de uso – diseño
- Subsistema de diseño
- Interfaz
- Modelo de despliegue
- Descripción de la arquitectura

Modelo de diseño (i)

- Es un modelo de objetos que describe la realización física de los casos de uso centrándose en cómo los requisitos funcionales y no funcionales, junto con otras restricciones relacionadas con el entorno de implementación, tienen impacto en el sistema a considerar
- El modelo de diseño se representa por un sistema de diseño que denota el subsistema de nivel más alto de modelado
  - La utilización de otro subsistema es una forma de organización del modelo de diseño en partes más manejables
- Los subsistemas de diseño y las clases de diseño representan **abstracciones** del subsistema y componentes de la implementación del sistema
- Los casos de uso son realizados por los objetos de las clases de diseño

Modelo de diseño (ii)



[Jacobson et al., 1999]

Clase de diseño

- Representa una abstracción de una o varias clases (o construcción similar) en la implementación del sistema
- Esta abstracción es sin costuras debido a [Jacobson et al., 1999]
  - El lenguaje utilizado para especificar una clase de diseño es el mismo que el lenguaje de programación utilizado
  - Se especifica la visibilidad de atributos y operaciones, utilizando con frecuencia los términos derivados de C++
  - Las relaciones entre clases de diseño suelen tener un significado directo cuando la clase es implementada
  - Los métodos de una clase de diseño tienen correspondencia directa con el correspondiente método en la implementación de las clases
  - Una clase de diseño puede aparecer como un estereotipo que se corresponde con una construcción en el lenguaje de programación dado
  - Una clase de diseño se puede realizar, esto es, proporcionar interfaces si tiene sentido hacerlo en el lenguaje de programación
  - Una clase de diseño se puede activar, implicando que objetos de la clase mantengan su propio hilo de control y se ejecuten concurrentemente con otros objetos activos

Realización de caso de uso – diseño

- Es una colaboración en el modelo de diseño que describe cómo se realiza un caso de uso específico
- Una vista del modelo de diseño centrado en los siguientes artefactos significativos desde el punto de vista de la arquitectura
  - Diagramas de clases: Clases de diseño, sus características y relaciones
  - Diagramas de interacción: Diagramas de secuencia, diagramas de estado
  - Flujo de eventos – diseño
  - Requisitos de implementación

Subsistema de diseño

- Son una forma de organizar los artefactos del modelo del diseño en piezas más manejables
- Deben ser una colección cohesiva de clases de diseño, realizaciones de caso de uso, interfaces y otros subsistemas
- El acoplamiento entre subsistemas ha de ser mínimo
- Utilizados para separar los aspectos del diseño
- Las dos capas de aplicación de más alto nivel y sus subsistemas dentro del modelo de diseño suelen tener trazas directas hacia paquetes del análisis
- Los subsistemas pueden representar componentes de grano grueso en la implementación del sistema, es decir, elementos de grano más grueso que se convierten ellos mismos en ejecutables, ficheros binarios o entidades similares que pueden distribuirse en diferentes nodos
- Pueden representar productos software reutilizados que han sido encapsulados en ellos
- Pueden representar sistemas heredados o partes de ellos

Interfaz

- Especifica una colección de operaciones públicas, tipos y parámetros necesarios para acceder y usar las capacidades de una clase de diseño o un subsistema
- Una clase de diseño que realice una interfaz debe proporcionar los métodos que implementen las operaciones de la interfaz
- Un subsistema que realice una interfaz debe conectar también clases del diseño u otros subsistemas (recursivamente) que proporcionen la interfaz
- Las interfaces son una forma de separar la especificación de la funcionalidad
- La mayoría de las interfaces entre subsistemas se consideran relevantes para la arquitectura debido a que definen las interacciones permitidas entre los subsistemas

Modelo de despliegue

- Es un modelo de objetos que describe la distribución física del sistema en términos de cómo se distribuye la funcionalidad entre los nodos de cómputo
  - Correspondencia entre la arquitectura software y la arquitectura del sistema
- Cada nodo representa un recurso de cómputo, normalmente un procesador o un dispositivo hardware similar
- Los nodos poseen relaciones que representan medios de comunicación entre ellos
- El modelo de despliegue puede describir diferentes configuraciones de red
- La funcionalidad de un nodo se define por los componentes que se distribuyen en ese nodo

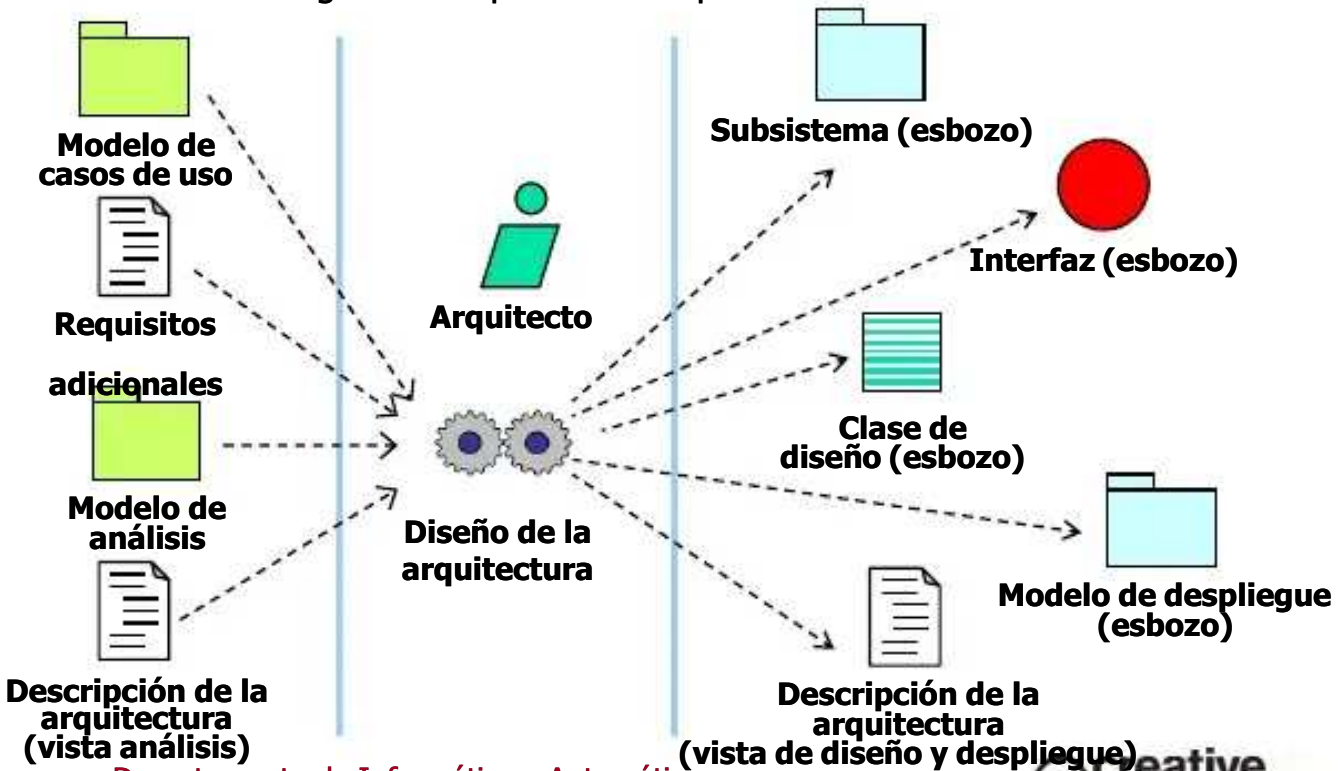
Descripción de la arquitectura

- Contiene una vista de la arquitectura del modelo de diseño que muestra sus artefactos relevantes para la arquitectura
  - Subsistemas, sus interfaces y las dependencias entre ellos
  - Clases de diseño fundamentales con una traza a las clases de análisis significativas y clases activas
  - Realizaciones de caso de uso – diseño que describan una funcionalidad importante y crítica y que deban desarrollarse pronto en el ciclo de vida
- Contiene una vista de la arquitectura del modelo de despliegue que muestra los artefactos relevantes para la arquitectura

3. Diseño de la arquitectura

Introducción

- El objetivo es esbozar los modelos de diseño y despliegue y su arquitectura mediante la identificación de
  - Los nodos y sus configuraciones de red
  - Los subsistemas y sus interfaces
  - Las clases de diseño significativas para la arquitectura
  - Los mecanismos de diseño genéricos que tratan requisitos comunes



Identificación de nodos y configuraciones de red

- Las configuraciones de red suelen tener una gran influencia sobre la arquitectura del software
- Las configuraciones de red habituales utilizan un patrón de tres capas (o un derivado del mismo)
  - Capa de interacción con el usuario
  - Capa de lógica de negocio
  - Capa de acceso a datos
- Aspectos a destacar [Jacobson et al., 1999]
  - ¿Qué nodos se necesitan y cuál debe ser su capacidad en términos de potencia de procesamiento y tamaño de memoria?
  - ¿Qué tipo de conexiones debe haber entre los nodos y qué protocolos de comunicaciones deben utilizarse?
  - ¿Qué características deben tener las conexiones y los protocolos de comunicaciones, en aspectos tales como ancho de banda, disponibilidad y calidad?
  - ¿Es necesario tener alguna capacidad de proceso redundante, modos de fallo, migración de procesos, mantenimiento de copias de seguridad de los datos, o aspectos similares?

Identificación de subsistemas y de sus interfaces (i)

- Los subsistemas son un medio para organizar el modelo de diseño
- No todos los subsistemas se desarrollan internamente en el proyecto en curso
- Los subsistemas se organizan siguiendo un patrón de capas [Buchmann et al., 1996; Shaw y Garlan, 1996]
  - Este patrón facilita la organización jerárquica de los subsistemas en capas
  - Sigue la máxima de que los subsistemas de una capa sólo pueden referenciar subsistemas de un nivel igual o inferior
  - La comunicación entre los subsistemas de diferentes capas se lleva a cabo mediante un conjunto de interfaces bien definidas
- Identificación de subsistemas de aplicación
  - Se identifican los subsistemas de las capas de la aplicación (dos capas superiores)
  - Si se hizo una división adecuada en paquetes durante el análisis, se pueden utilizar éstos tanto como sea posible, e identificar los correspondientes subsistemas dentro del modelo de diseño
    - Se pueden refinar estos subsistemas para tratar temas relativos al diseño

Identificación de subsistemas y de sus interfaces (ii)

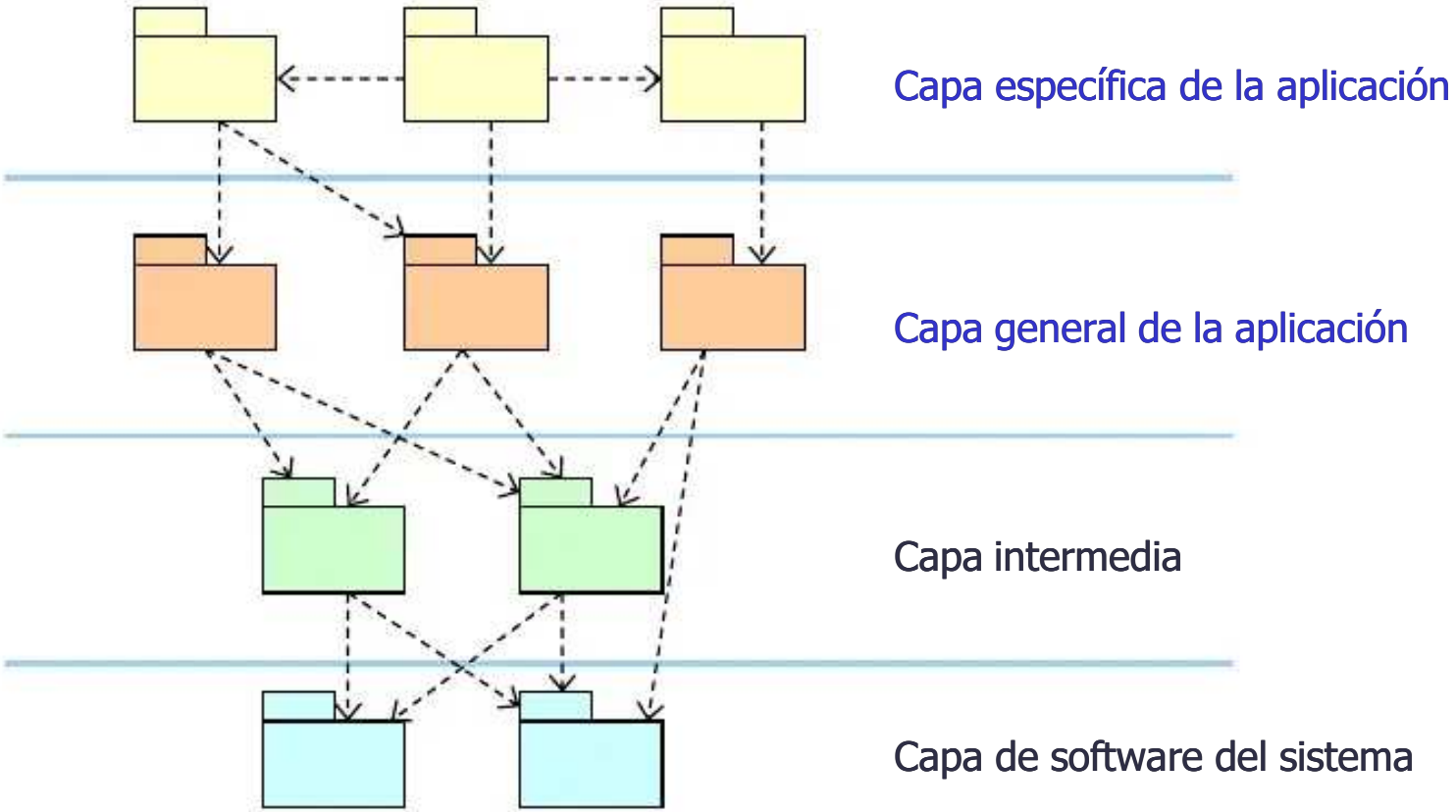
- La descomposición inicial de los subsistemas del análisis se refina cuando [Jacobson et al., 1999]
  - Una parte de un paquete del análisis se corresponde con un subsistema por sí misma
    - Esa parte puede ser compartida y utilizada por otros subsistemas
  - Algunas partes de un paquete del análisis se realizan mediante productos software reutilizados
    - Estas funcionalidades pueden asignarse a capas intermedias o subsistemas de software del sistema
  - Los paquetes del análisis no representan una división adecuada del trabajo
  - Los paquetes del análisis no representan la incorporación de un sistema heredado
    - Se puede encapsular un sistema heredado, o parte de él, mediante un subsistema de diseño independiente
  - Los paquetes del análisis no están preparados para una distribución directa sobre los nodos

Identificación de subsistemas y de sus interfaces (iii)

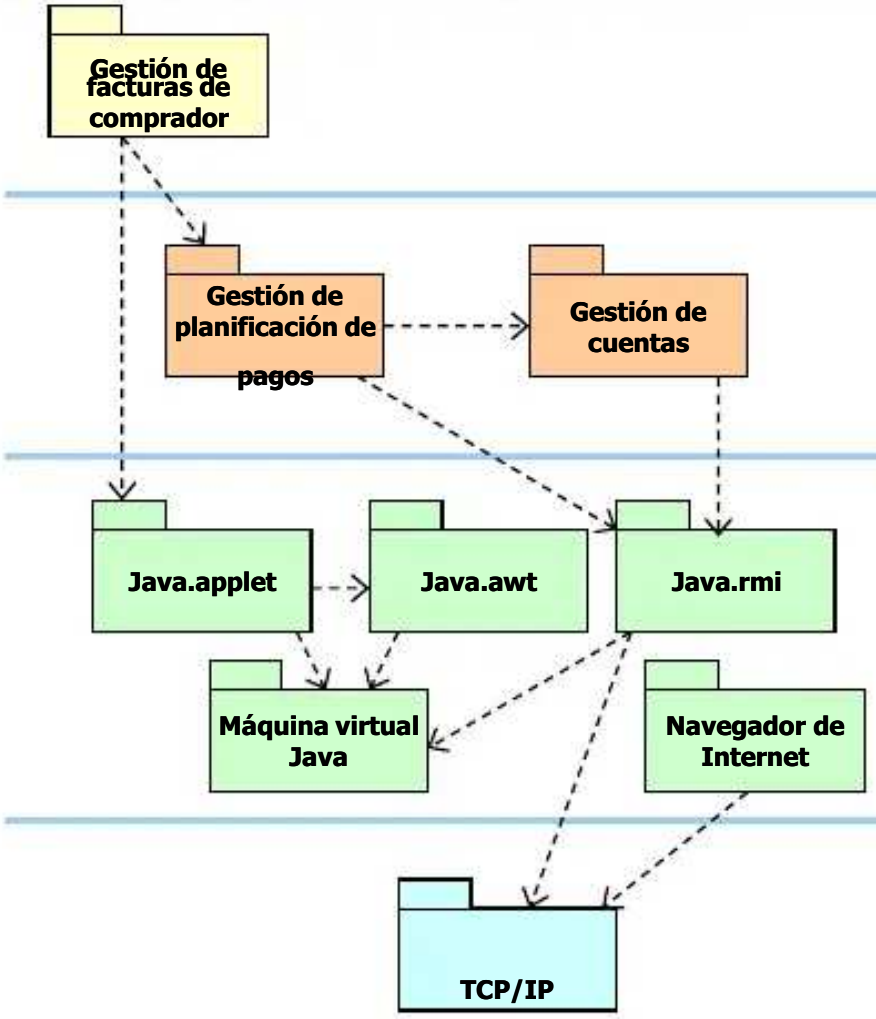
- Identificación de subsistemas intermedios y de software del sistema
  - Estos subsistemas constituyen los cimientos de un sistema
    - Toda la funcionalidad descansa sobre software como sistemas operativos, sistemas de gestión de bases de datos, software de comunicaciones, tecnologías de distribución de objetos, bibliotecas de componentes para el diseño de interfaces gráficas de usuario, y tecnologías de gestión transacciones [Jacobson et al., 1997]
  - La selección e integración de productos software que se compran o se construyen son dos de los objetivos fundamentales durante las fases de inicio y elaboración
- Definición de las dependencias
  - Deberían definirse dependencias entre subsistemas si sus contenidos tienen relación
  - La dirección de la dependencia debería ser la misma que la dirección de la navegabilidad de la relación
    - Si se utilizan interfaces entre subsistemas, las dependencias deberían ir hacia las interfaces, no hacia los subsistemas

Identificación de subsistemas y de sus interfaces (iv)

- Patrón de capas propuesto en [Jacobson et al., 1999]



Identificación de subsistemas y de sus interfaces (v)



Capa específica de la aplicación

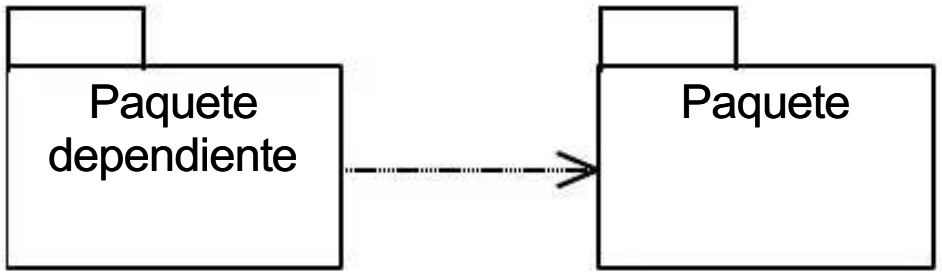
Capa general de la aplicación

Capa intermedia

Capa de software del sistema

Identificación de subsistemas y de sus interfaces (vi)

- El principio de dependencia acíclica (i)
  - La estructura de dependencia entre los paquetes debe ser un grafo dirigido acíclico (DAG). Esto es, no debe haber ciclos en la estructura de dependencia (Robert C. Martin)



Identificación de subsistemas y de sus interfaces (vii)

- El principio de dependencia acíclica (ii)

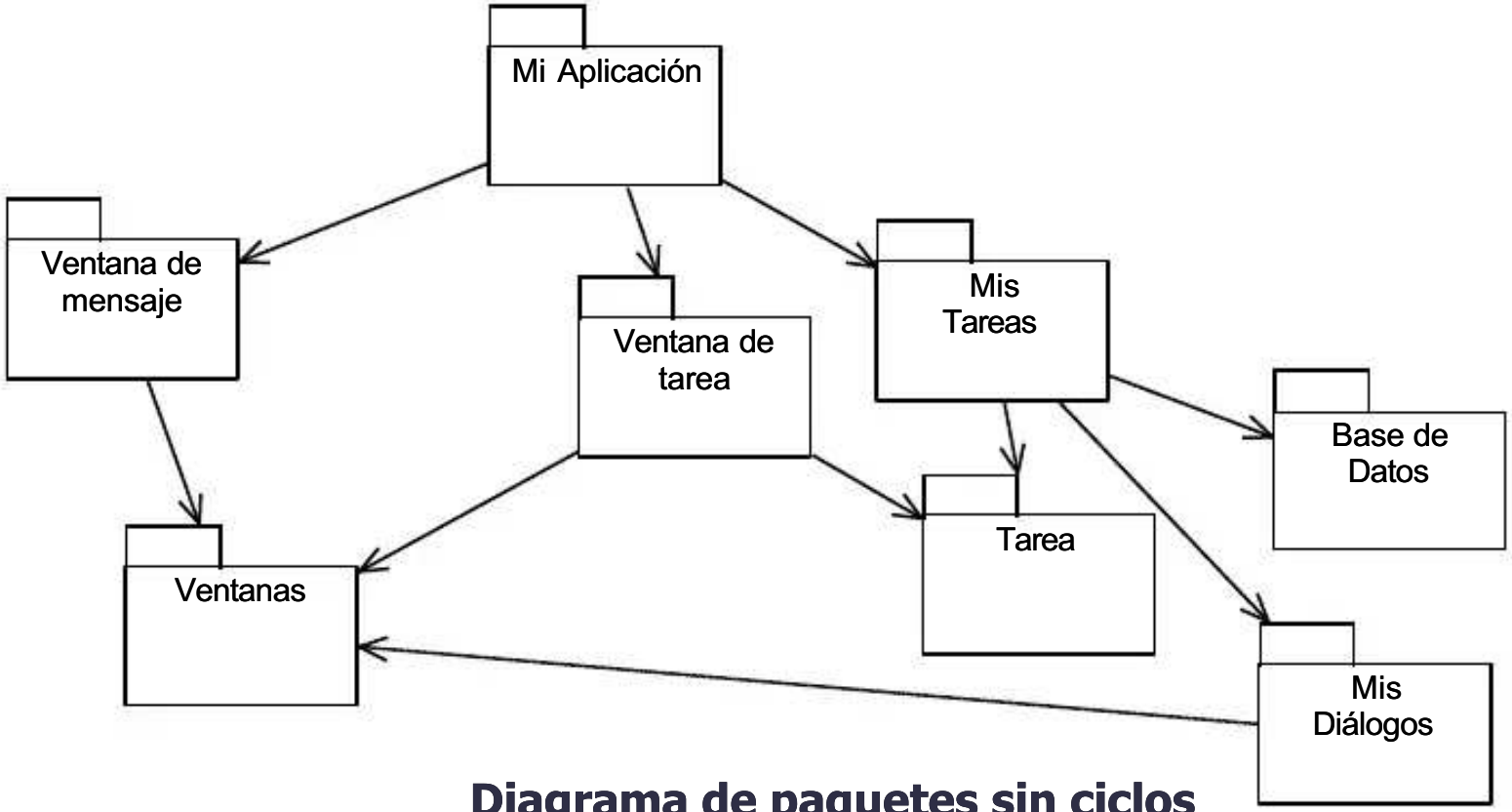


Diagrama de paquetes sin ciclos

Identificación de subsistemas y de sus interfaces (viii)

- El principio de dependencia acíclica (iii)

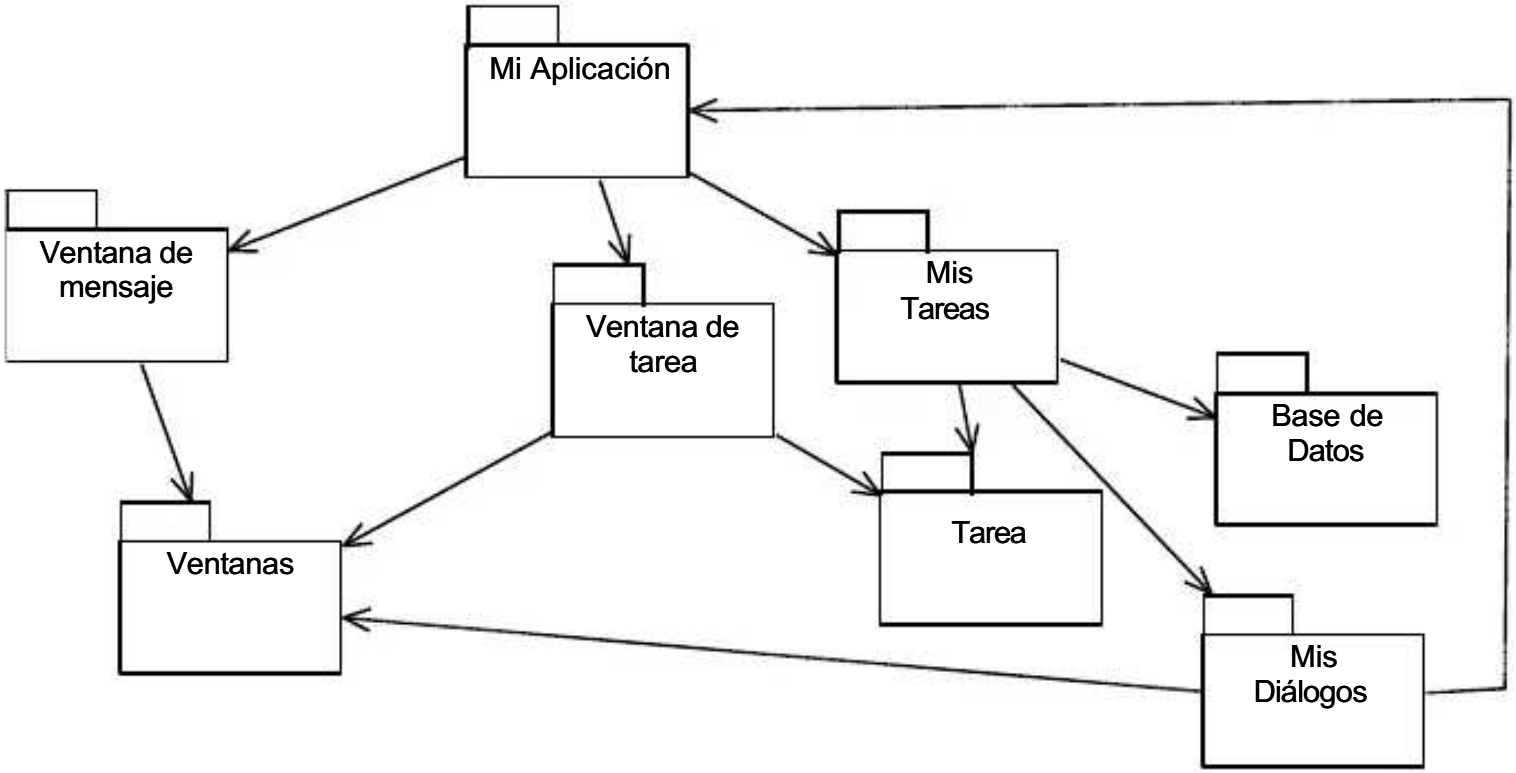
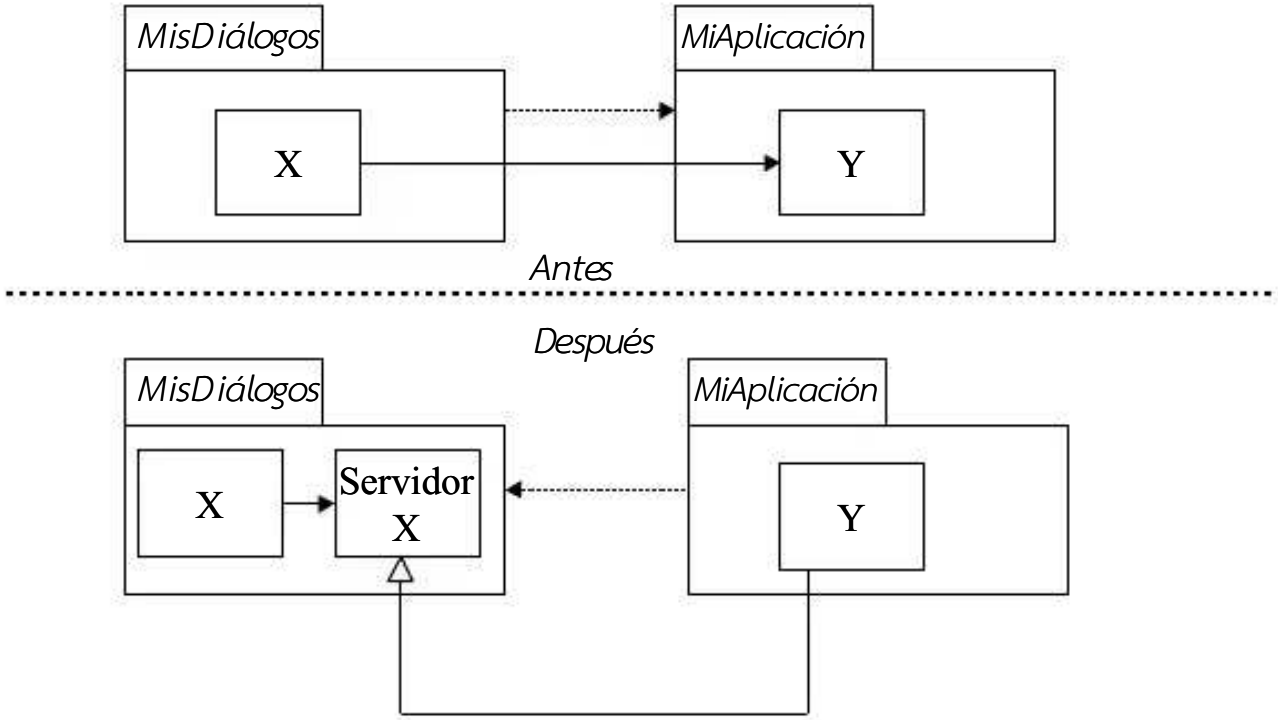


Diagrama de paquetes con ciclos

Identificación de subsistemas y de sus interfaces (ix)

- El principio de dependencia acíclica (iv)
  - Eliminación del ciclo con el principio de inversión de la dependencia
    - Dependencia de las abstracciones de las concreciones [Martin, 1996]



Identificación de clases de diseño

- Se pueden esbozar inicialmente algunas clases a partir de las clases del análisis
  - Se pueden utilizar las relaciones entre esas clases para identificar un conjunto tentativo de relaciones entre las correspondientes clases de diseño
- Se deben identificar también las clases activas necesarias en el sistema, considerando los requisitos de concurrencia del mismo
  - Requisitos de rendimiento, tiempo de respuesta y disponibilidad de los diferentes actores en su interacción con el sistema
  - Distribución del sistema sobre los nodos
  - Otros requisitos sobre el arranque y terminación del sistema, progresión, evitar interbloqueos, evitar la inanición, reconfiguración de los nodos y la capacidad de los nodos

Identificación de mecanismos genéricos de diseño

- Se tratan los requisitos especiales identificados durante el análisis
  - Se tienen en cuenta las tecnologías de diseño e implementación disponibles
- El resultado es un conjunto de mecanismos genéricos de diseño que pueden manifestarse como clases, colaboraciones o incluso como subsistemas
- Los requisitos que deben tratarse suelen estar relacionados con aspectos como
  - Persistencia
  - Distribución transparente de objetos
  - Características de seguridad
  - Detección y recuperación de errores
  - Gestión de transacciones

4. Patrones de diseño orientado a objetos

Introducción (i)

- La experiencia es tan importante como las notaciones para recoger y representar los diseños y las reglas de uso de esas notaciones
- Los patrones de diseño identifican y entrañan abstracciones de diseño orientado al objeto
- Usos de los patrones de diseño en el proceso de desarrollo orientado al objeto
  - Proporcionan un vocabulario común
  - Constituyen una base de experiencias reutilizables
  - Ayudan a la reducción del tiempo de aprendizaje de una biblioteca de clases
  - Proporcionan un objetivo para la reorganización o la refactorización

Introducción (ii)

- Un patrón es
  - Una solución a un problema en un contexto determinado
    - Codifica conocimiento específico recogido a partir de la experiencia en un dominio
  - Una forma de documentar resolución de problemas de la ingeniería del software
  - Una idea reutilizable
  - Una relación entre un contexto, un problema y una solución
  - Identificación de buenas estructuras de diseño que se repiten en la práctica

Definición

- Un **patrón** es una regla que establece una relación entre un contexto, un sistema de fuerzas que aparecen en el contexto y una configuración [Alexander, 1979]
- Un **patrón** es una descripción en un formato fijo de cómo solucionar un cierto tipo de problemas [Reenskaug et al., 1996]
- Cada **patrón** es una regla constituida por tres partes, la cual expresa una relación entre un cierto contexto, un cierto sistema de fuerzas que ocurren repetidamente en ese contexto, y una cierta configuración software que permite a estas fuerzas resolverse así mismas [Coplien, 2007]
- Un **patrón** es una unidad de información instructiva con nombre que captura la estructura esencial y la comprensión de una familia de soluciones exitosas probadas para un problema recurrente que ocurre dentro de un cierto contexto y de un sistema de fuerzas [Appleton, 2000]
- Un **patrón** es un par problema/solución que tiene un nombre y que puede aplicarse en contextos nuevos, con las instrucciones correspondientes acerca de cómo utilizarlo en una situación nueva [Larman, 2002]
- Un **patrón de diseño** es una descripción de clases y objetos comunicándose entre sí, adaptada para resolver un problema de diseño general en un contexto particular [Gamma et al., 1995]
- Un **patrón de diseño** es una abstracción de un problema de diseño general, que ocurre recurrentemente en contextos específicos no arbitrarios [Aklecha, 1999]

Descripción de un patrón (i)

- Los patrones se describen utilizando formatos consistentes
  - Plantilla dividida en secciones, que ofrece uniformidad
  - Existen diferentes formatos de descripción de patrones
    - Formato de Alexander [Alexander, 1979]
    - Formato canónico [Buschmann et al., 1996]
    - Formato del GoF [Gamma et al., 1995]

Descripción de un patrón (ii)

- Nombre
  - Una abstracción significativa que sugiere su aplicabilidad e intención
- Contexto
  - Para describir cuando ha de aplicarse el patrón e indicar el entorno y condiciones que han de existir para que el patrón de diseño sea aplicable y la solución funcione
- Problema
  - Declaración del problema y/o intención de la solución
- Características (fuerzas)
  - Indica los atributos del diseño que han de ajustarse para permitir que el patrón se acomode a una gran variedad de problemas
  - Son objetivos y restricciones; factores que lo motivan; indicaciones de por qué el problema es complicado

Descripción de un patrón (iii)

- Consecuencias
  - Asociadas a la utilización del patrón
  - Proporcionan una indicación de las ramificaciones de las decisiones de diseño
  - Resultados y costes (inconvenientes) de la aplicación del patrón
- Solución
  - Para describir una solución de propósito general al problema que contempla el patrón
  - Indica cómo generar la solución, la estructura y sus participantes y colaboraciones
    - Estructural: Diagrama de clases (estática)
    - Comportamiento: Diagramas de interacción (dinámica)
- Ejemplo
  - Ejemplo de utilización del patrón
- Patrones relacionados
  - Patrones que son similares; patrones que usa o que son usados por él
- Usos conocidos
  - Relación de ejemplos reales de utilización del patrón

Tipos de patrones

- **Análisis**
  - Describe la estructura y el flujo de trabajo en un dominio específico [Fowler, 1996]
- **Organizativos**
  - Ayudan en la organización del desarrollo del software
- **Arquitectónico**
  - Un patrón arquitectónico expresa un esquema organizativo estructural fundamental para un sistema software
  - Proporciona un conjunto de subsistemas predefinidos, sus responsabilidades, e incluye reglas y recomendaciones para organizar las relaciones entre ellos
- **Diseño**
  - Un patrón de diseño proporciona un esquema para el refinamiento de subsistemas o componentes de un sistema software, o las relaciones entre ellos
  - Describe un estructura recurrente común de componentes que se comunican y que resuelven un problema general de diseño dentro de un contexto particular
- **Idiomas**
  - Un idioma es un patrón de bajo nivel específico de un lenguaje de programación
  - Un idioma describe cómo implementar un aspecto particular de los componentes o de sus relaciones utilizando las características del lenguaje concreto

Clasificación de los patrones de diseño

- POSA [Buschmann et al., 1996]
  - Categoría del patrón
    - Relacionado con la principales fases y actividades en el desarrollo del software
  - Categoría del problema
    - Los diferentes tipos de problemas que pueden surgir en el desarrollo de sistemas software
- GoF [Gamma et al., 1995]
  - Propósito
    - Refleja qué hace el patrón
  - Ámbito
    - Si el patrón se aplica a clases o a objetos
- GRASP [Larman, 2002]
  - Aspectos fundamentales del diseño
    - Básicos y avanzados

POSA – Categorías de patrones

- Patrones arquitectónicos
  - Para expresar el esquema de una organización estructural fundamental para los sistemas software
  - Son plantillas para arquitecturas software concretas
  - Proporcionan un conjunto de subsistemas predefinidos, especifican sus responsabilidades e incluye reglas y recomendaciones para organizar las relaciones entre los subsistemas
  - La selección de un patrón arquitectónico es una decisión de diseño fundamental cuando se desarrolla un sistema software
  - Ejemplos: Modelo-Vista-Controlador, Capas (*Layers*), *Broker*, *Pipes and Filter*
- Patrones de diseño
  - De menor escala que los patrones arquitectónicos y sin efectos sobre la estructura fundamental de los sistemas software
  - Proporcionan un esquema para refinar los subsistemas o componentes de un sistema software o las relaciones entre ellos
  - Describen un estructura frecuente de componentes que se comunican y que resuelven un problema general de diseño en un contexto particular
  - Ejemplos: *Observer*, *Publisher-Subscriber*
- Idiomas
  - Son patrones de bajo nivel específicos de lenguajes de programación
  - Describen cómo implementar aspectos particulares de componentes o las relaciones entre ellos utilizando las características de un lenguaje dado
  - Ejemplo: `while (*destino++ = *src++);`

POSA – Categorías de problemas (i)

- De estructuración (*from mud to structure*)
  - Patrones que soportan una descomposición adecuada de la tarea general de un sistema en subtareas cooperativas
- Sistemas distribuidos
  - Patrones que proporcionan infraestructuras para sistemas que tienen componentes situados en procesos diferentes o en varios subsistemas y componentes
- Sistemas interactivos
  - Patrones que ayudan a estructurar sistemas con interacción hombre-máquina
- Sistemas adaptables
  - Patrones que proporcionan infraestructuras para la extensión y adaptación de aplicaciones en respuesta a requisitos funcionales que cambian y evolucionan
- Descomposición estructural
  - Patrones que soportan una descomposición adecuada de subsistemas y componentes complejos en partes cooperativas

POSA – Categorías de problemas (ii)

- Organización del trabajo
  - Patrones que definen cómo colaboran los componentes para proporcionar un servicio complejo
- Control de acceso
  - Patrones de vigilan y controlan el acceso a los servicios o componentes
- Gestión
  - Patrones para el manejo de colecciones homogéneas de objetos, servicios y componentes en su totalidad
- Comunicación
  - Patrones que ayudan a organizar la comunicación entre componentes
- Manejo de recursos
  - Patrones que ayudan a gestionar componentes y objetos compartidos

POSA – Patrones

POSA	Arquitectónicos	Diseño	Idiomas
Estructuración	*Layers; Pipes&Filters; Blackboard*		
Sistemas distribuidos	*Broker* *Pipes&Filters* *Microkernel*		
Sistemas interactivos	*MVC; PAC*		
Sistemas adaptables	*Microkernel, Reflection*		
Descomposición estructural		*Whole-Part*	
Organización del trabajo		*Master-Slave*	
Control acceso		*Proxy*	
Gestión		*Command Processor* *View Handler*	
Comunicación		*Publisher-Subscriber* *Forwarder-Receiver* *Client-Dispatcher-Server*	
Manejo recursos			*Counted Pointer*

GoF – Introducción

- Tres partes esenciales de un patrón de diseño
  - Una descripción abstracta de una clase o una colaboración de objetos y su estructura
  - El tema del diseño del sistema abordado por la estructura abstracta
  - Las consecuencias de la aplicación de la estructura abstracta a la arquitectura del sistema
- Plantilla básica
  - Nombre del patrón de diseño
  - Intención
  - También conocido como
  - Motivación
  - Aplicabilidad
  - Estructura
  - Participantes
  - Colaboraciones
  - Consecuencias
  - Implementación
  - Código de ejemplo
  - Usos conocidos
  - Patrones relacionados

GoF – Clasificación de patrones (i)

- Ámbito
  - Dominio de aplicación
    - Clase
      - Relaciones entre clases base y sus subclases
    - Objeto
      - Relaciones de iguales entre objetos (*peer objects*)
    - Composición
      - Estructuras recursivas de objetos
- Propósito
  - Qué hacer
    - Creación
      - Proceso de creación de objetos
    - Estructural
      - Composición de clases y objetos
    - Comportamiento
      - Las formas en las que las clases y objetos interaccionan y se distribuyen las responsabilidades

GoF – Clasificación de patrones (ii)

- Ámbito de clase
  - Patrones de Creación – Cómo instanciar objetos ocultando la parte específica del proceso de creación
    - *Factory Method*
  - Patrones Estructurales – Cómo utilizar la herencia para componer protocolos o código
    - *Adapter*
  - Patrones de comportamiento – Cómo cooperan las clases con sus subclases para ajustarse a su semántica
    - *Template Method*
- Ámbito de objeto
  - Patrones de creación – Cómo se crean conjuntos de objetos
    - *Abstract Factory*
  - Patrones estructurales – Cómo juntar objetos para hacerse cargo de una funcionalidad nueva
    - *Proxy, Flyweight*
  - Patrones de comportamiento – Cómo un grupo de objetos "equivalentes" (*peer*) coperan para llevar a cabo una tarea que un objeto por sí solo no puede llevar a cabo
    - *Mediator, Chain of Responsibility, Observer, Model-View, Strategy*
- Ámbito de composición
  - Patrones de creación – Cómo crear una estructura recursiva de objetos
    - *Builder*
  - Patrones estructurales – Cómo crear estructuras recursivas de objetos
    - *Composite, Wrapper*
  - Patrones de comportamiento – El comportamiento de estructuras recursivas de objetos
    - *Iterator*

GoF – Patrones

		Propósito		
		Creación	**Estructural**	**Comportamiento**
Ámbito	**Clase**	*Factory Method*	*Adapter (class)* *Bridge (class)*	*Template method*
	Objeto	*Abstract Factory* *Prototype* *Solitaire*	*Adapter (object)* *Bridge (object)* *Flyweight* *Glue* *Proxy*	*Chain of Responsibility* *Command* *Iterator (object)* *Mediator* *Memento* *Observer* *State* *Strategy*
	Composición	*Builder*	*Composite* *Wrapper*	*Interpreter* *Iterator (compound)* *Walker*

POSA y GoF – Patrones (i)

POSA y GOF (1)	Arquitectónicos	Diseño	Idiomas
Estructuración	*Layers* (POSA) *Pipes&Filters* (POSA) *Blackboard* (POSA)	*Interpreter* (GOF)	
Sistemas distribuidos	*Broker* (POSA) *Pipes&Filters* (POSA) *Microkernel* (POSA)		
Sistemas interactivos	*MVC* (POSA) *PAC* (POSA)		
Sistemas adaptables	*Microkernel* (POSA) *Reflection* (POSA)		
Creación		*Abstract Factory* (GOF) *Prototype* (GOF) *Builder* (GOF)	*Singleton* (GOF) *Factory Method* (GOF)
Descomposición estructural		*Whole-Part* (POSA) *Composite* (GOF)	
Organización del trabajo		*Master-Slave* (POSA) *Chain of Reponsibility* (GOF) *Command* (GOF) *Mediator* (GOF)	

POSA y GoF – Patrones (ii)

POSA y GOF (2)	Arquitectónicos	Diseño	Idiomas
Control acceso		*Proxy* (POSA) *Facade* (GOF) *Iterator* (GOF)	
Variación Servicio		*Bridge* (GOF) *Strategy* (GOF) *State* (GOF)	*Template Method* (GOF)
Extensión Servicio		*Decorator* (GOF) *Visitor* (GOF)	
Gestión		*Command Processor* (POSA) *View Handler* (POSA) *Memento* (GOF)	
Adaptación		*Adapter* (GOF)	
Comunicación		*Publisher-Subscriber* (POSA) *Forwarder-Receiver* (POSA) *Client-Dispatcher-Server* (POSA)	
Manejo recursos		*Flyewight* (GOF)	*Counted Pointer* (POSA)

GRASP – Introducción

- GRASP (*General Responsibility Assignment Software Patterns*)
- No son patrones "reales"
  - Son descripciones formales de principios fundamentales del diseño orientado a objetos y de la asignación de responsabilidades expresados como patrones
- Definidos en función de la asignación de responsabilidad
  - Una responsabilidad es un contrato u obligación de un clasificador [OMG, 2003]
    - ¿Qué tiene que conocer un objeto o una clase de objetos?
      - Conocer los datos privados encapsulados
      - Conocer los objetos relacionados
      - Conocer las cosas que puede derivar o calcular
    - ¿Qué tiene que hacer un objeto o una clase de objetos?
      - Hacer algo él mismo, como crear un objeto o hacer un cálculo
      - Iniciar una acción en otros objetos
      - Controlar y coordinar actividades en otros objetos

GRASP – Patrones

- Aspectos fundamentales del diseño
  - Experto en información
  - Creador
  - Bajo acoplamiento
  - Alta cohesión
  - Controlador
  - Polimorfismo
- Aspectos avanzados del diseño
  - Fabricación Pura
  - Indirección
  - Variaciones protegidas

Estudio de algunos patrones significativos

- Patrón Modelo-Vista-Controlador [Buschmann et al., 1996]
- Patrón Fábrica abstracta [Gamma et al., 1995]
- Patrón Puente [Gamma et al., 1995]
- Patrón Mediador [Gamma et al., 1995]
- Patrón Experto en información (Experto) [Larman, 2002]
- Patrón Creador [Larman, 2002]
- Patrón Bajo acoplamiento [Larman, 2002]
- Patrón Alta cohesión [Larman, 2002]
- Patrón Controlador [Larman, 2002]
- Patrón Polimorfismo [Larman, 2002]
- Patrón Fabricación pura [Larman, 2002]
- Patrón Indirección [Larman, 2002]
- Patrón Variaciones protegidas [Larman, 2002]
- Patrón DAO [Sun, 2002]

MVC (i)

- El patrón MVC es un patrón arquitectónico [Buschman et al., 1996]
- Pertenece a la categoría de los patrones para sistemas interactivos
- Se encuentra en muchos sistemas interactivos y *frameworks* de aplicación para software con interfaces gráficas (MacApp, ET++, bibliotecas de Smalltalk, MFC)

MVC (ii)

- Problema
  - Propensión que tienen las interfaces de usuario para cambiar, al extender la funcionalidad de una aplicación o al portarla a un entorno gráfico diferente
  - Construir un sistema flexible es caro y propenso a los errores si la interfaz de usuario está altamente entremezclada con el núcleo funcional
- Fuerzas que intervienen
  - La misma información es presentada de maneras diferentes en distintas ventanas
  - La presentación y el comportamiento de una aplicación deben representar los cambios en los datos inmediatamente
  - Los cambios en la interfaz de usuario deben ser sencillos e incluso factibles en tiempo de ejecución
  - Soportar diferentes estándares de interfaces gráficas o portar la interfaz de usuario no debe afectar al código del núcleo de la aplicación

MVC (iii)

- Solución
  - El patrón MVC divide la aplicación en tres áreas: proceso, entrada y salida
    - El componente de modelo encapsula los datos y la funcionalidad central
      - Es independiente de la entrada y la salida
    - Los componentes de vista presentan la información al usuario
      - Una vista obtiene datos del modelo, pudiendo haber múltiples vistas del modelo
    - Los controladores reciben la entrada, normalmente eventos que son trasladados para servir las peticiones del modelo o de la vista
      - El usuario interactúa con el sistema sólo a través de los controladores

MVC (iv)

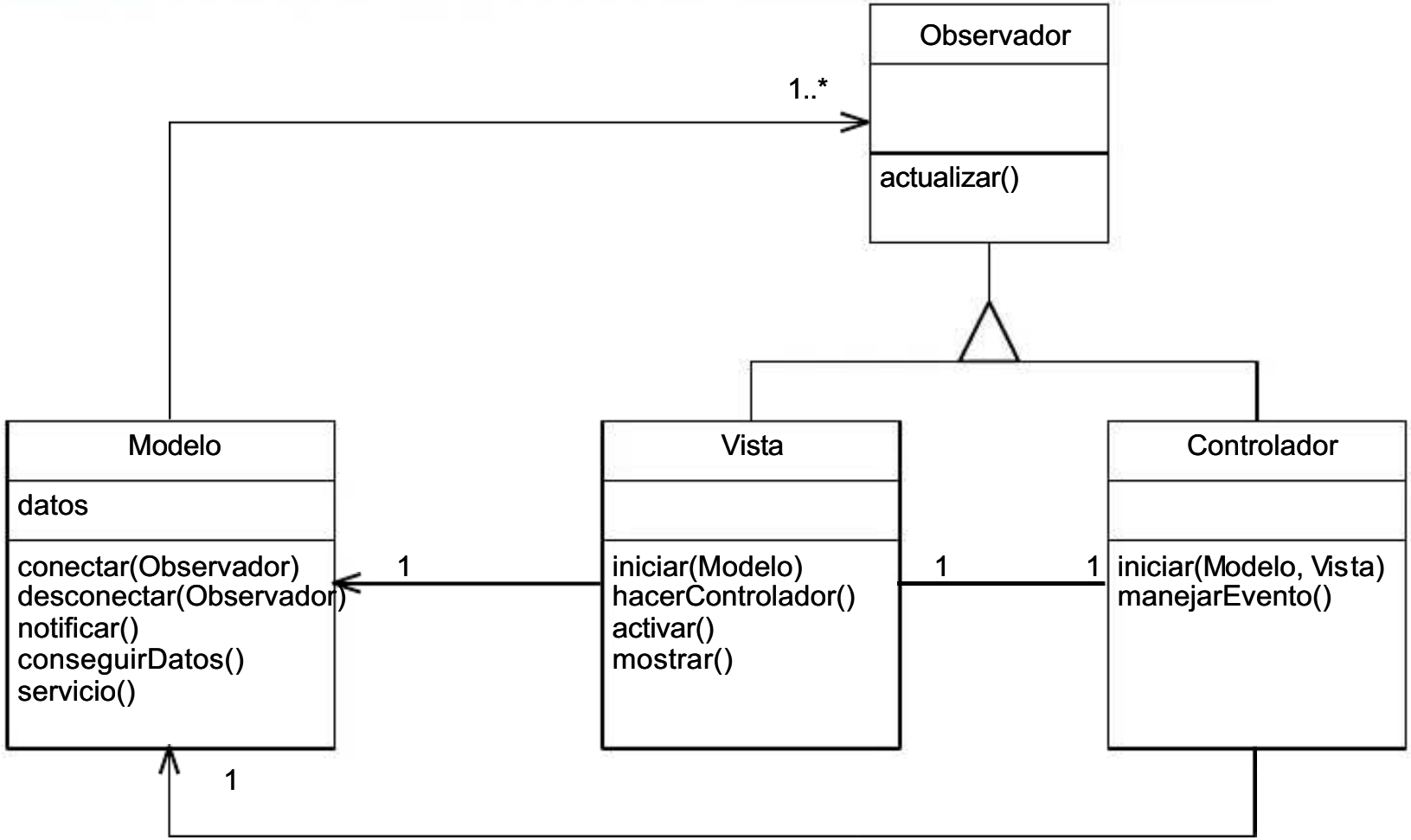
Clase Modelo	**Colaboradores**
Responsabilidades • Ofrece la funcionalidad central de la aplicación • Registrar las vistas y controladores dependientes • Notificar a los componentes dependientes sobre cambio en los datos	• Vista • Controlador

Clase Vista	**Colaboradores**
Responsabilidades • Crear e iniciar su controlador asociado • Mostrar la información al usuario • Implementar el método actualizar() • Recuperar datos del modelo	• Controlador • Modelo

MVC (v)

Clase Controlador	***Colaboradores***
Responsabilidades • Aceptar los eventos de entrada del usuario • Traducir los eventos en peticiones de servicio • Implementar el método actualizar() si se requiere • Recuperar datos del modelo	• Vista • Modelo

MVC (vi)



MVC (vii)

- Ventajas
  - Múltiples vistas del mismo modelo
  - Vistas sincronizadas
  - Cambios de vistas y controles en tiempo de ejecución
  - Cambio del aspecto externo de las aplicaciones
  - Base potencial para construir un *framework*
- Inconvenientes
  - Se incrementa la complejidad
  - Número de actualizaciones potencialmente alto
  - Íntima conexión entre la vista y el controlador
  - Alto acoplamiento de las vistas y los controladores con respecto al modelo
  - Acceso ineficiente a los datos desde la vista
  - Cambio inevitable de las vistas y controladores cuando se porte a otras plataformas

Fabrica abstracta (i)

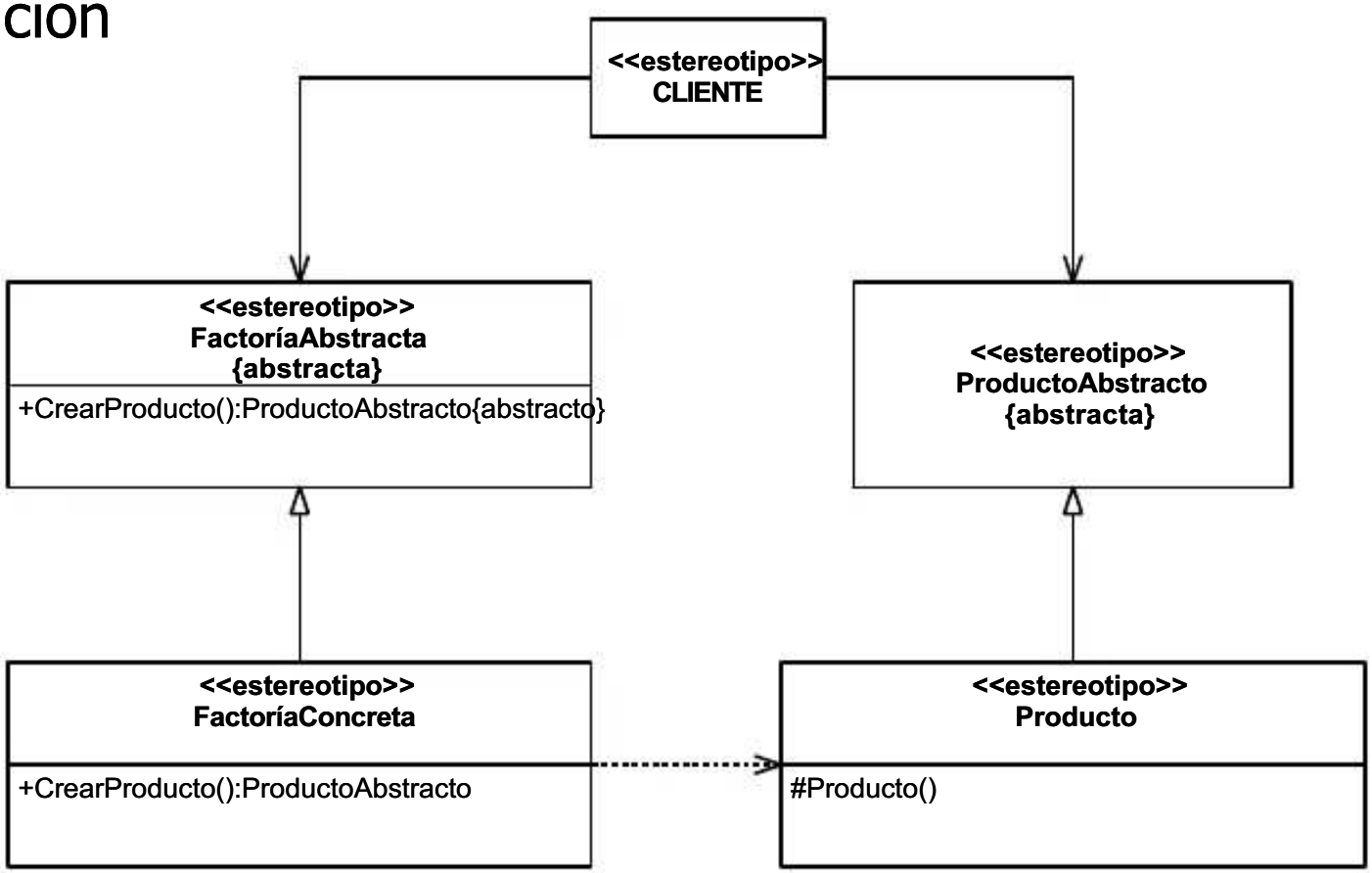
- Patrón Fabrica abstracta (*Abstract Factory*) – Patrón de creación de objetos [Gamma et al., 1995]
  - Los patrones de creación se utilizan para construir una abstracción sobre el proceso de instanciación, introduciendo una gran dosis de flexibilidad en todos los aspectos que involucren creación de objetos
- También conocido como *Kit*
- Ofrece una interfaz para crear familias de objetos relacionados o dependientes sin especificar sus clases concretas

Fabrica abstracta (ii)

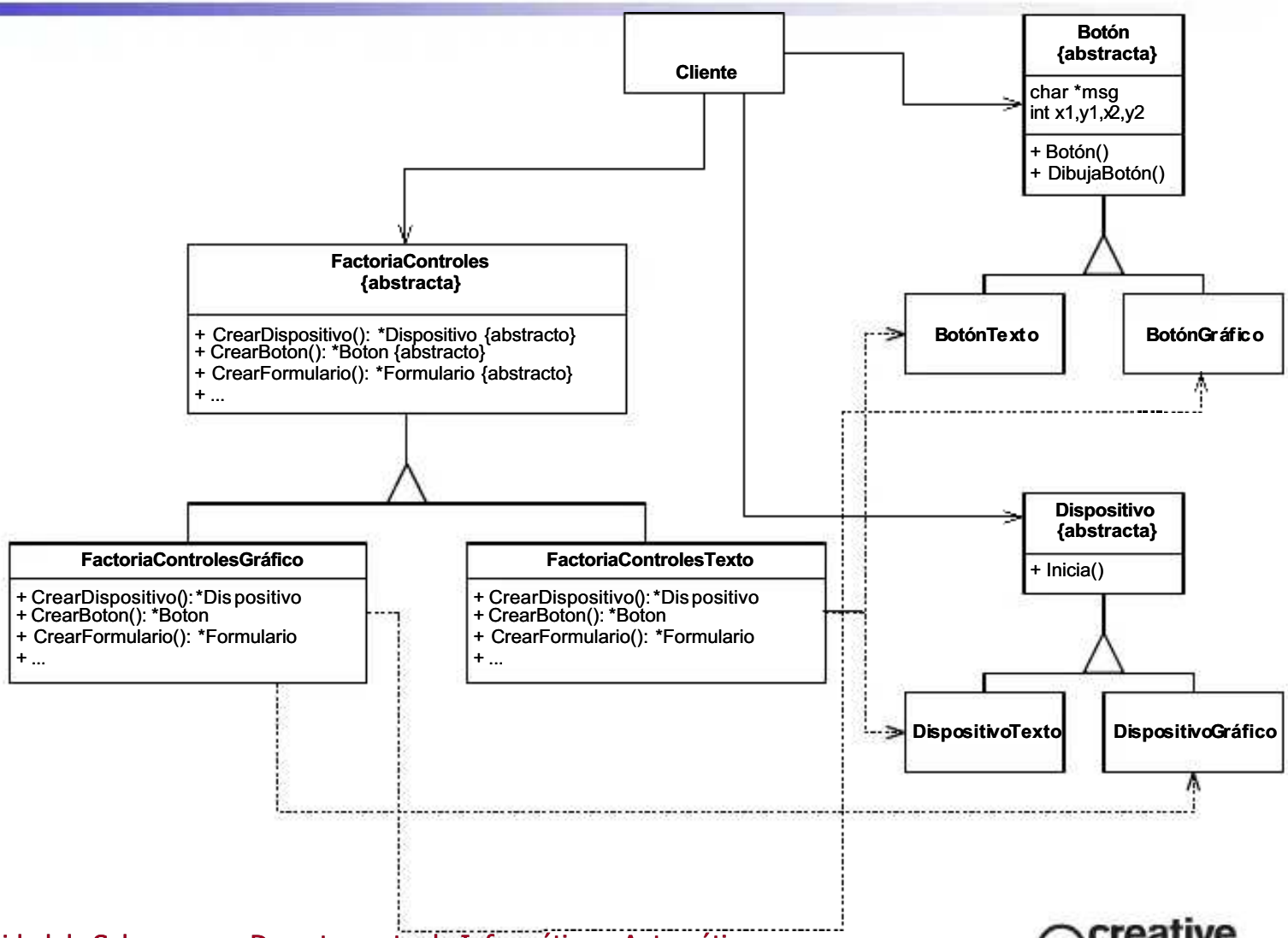
- Aplicabilidad
  - Un sistema debe ser independiente de cómo se creen, se compongan y se representen sus objetos
  - Un sistema debe ser configurado con un producto de múltiples familias de productos
  - Una serie de productos relacionados están diseñados para usarse conjuntamente, queriéndose reforzar esta característica
  - Se quiere distribuir una biblioteca de clases que implementa un determinado producto, queriéndose revelar sólo sus interfaces, no sus implementaciones

Fabrica abstracta (iii)

- Solución



Fabrica abstracta (iv)



Fabrica abstracta (v)

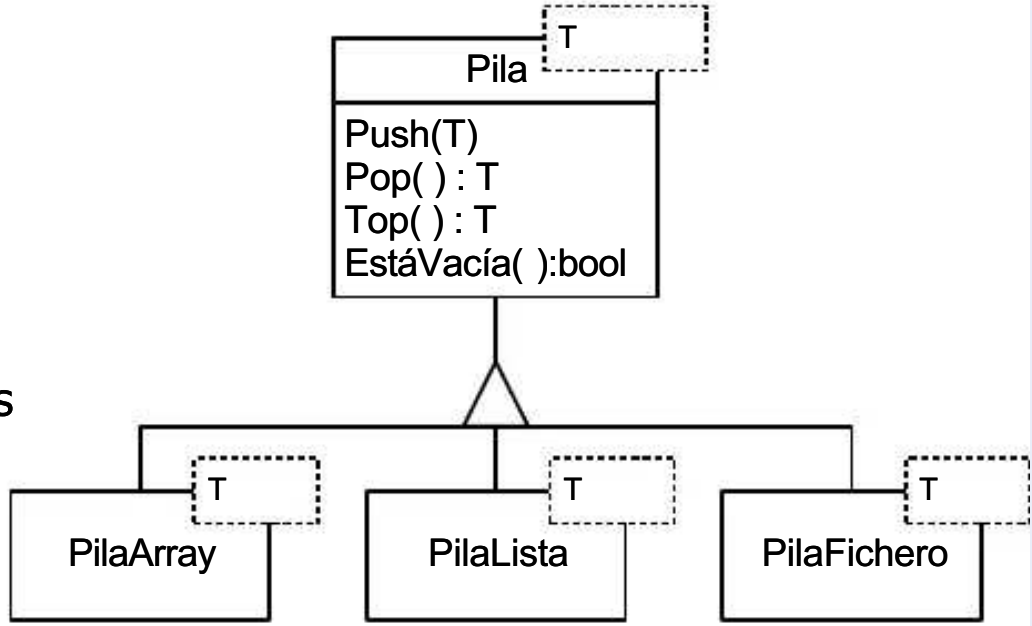
- Ventajas
  - Aísla los clientes de las implementaciones, ya que sólo usan la interfaz
  - Las dependencias se aíslan en las fabricas concretas
  - Facilita el cambio de familias de productos, ya que la clase de familia concreta sólo aparece una vez y es fácil cambiarla
  - Favorece la consistencia entre productos, ya que favorece que sólo se use una familia de productos a la vez
- Inconveniente
  - El soporte de nuevos productos se complica porque afecta a la interfaz de la fábrica abstracta y por consiguiente de todas sus subclases

Puente (i)

- Patrón Puente (*Bridge*) – Patrón estructural [Gamma et al., 1995]
  - Los patrones estructurales expresan cómo las clases y objetos se componen para formar estructuras mayores
- También conocido como *Handle/Body* o *Envelope*
- Desacopla una abstracción de su implementación para que ambas puedan variar independientemente

Puente (ii)

- Problema
  - Cuando una abstracción puede tener una de varias implementaciones, la forma usual de tratarlo es con herencia
    - Una clase abstracta define la interfaz de la abstracción, mientras que las subclases concretas la implementan de diferentes maneras
  - Esta aproximación no es flexible
    - La herencia enlaza una implementación a la abstracción de forma permanente, lo que dificulta la modificación, extensión y reutilización de las abstracciones y de las implementaciones independientemente

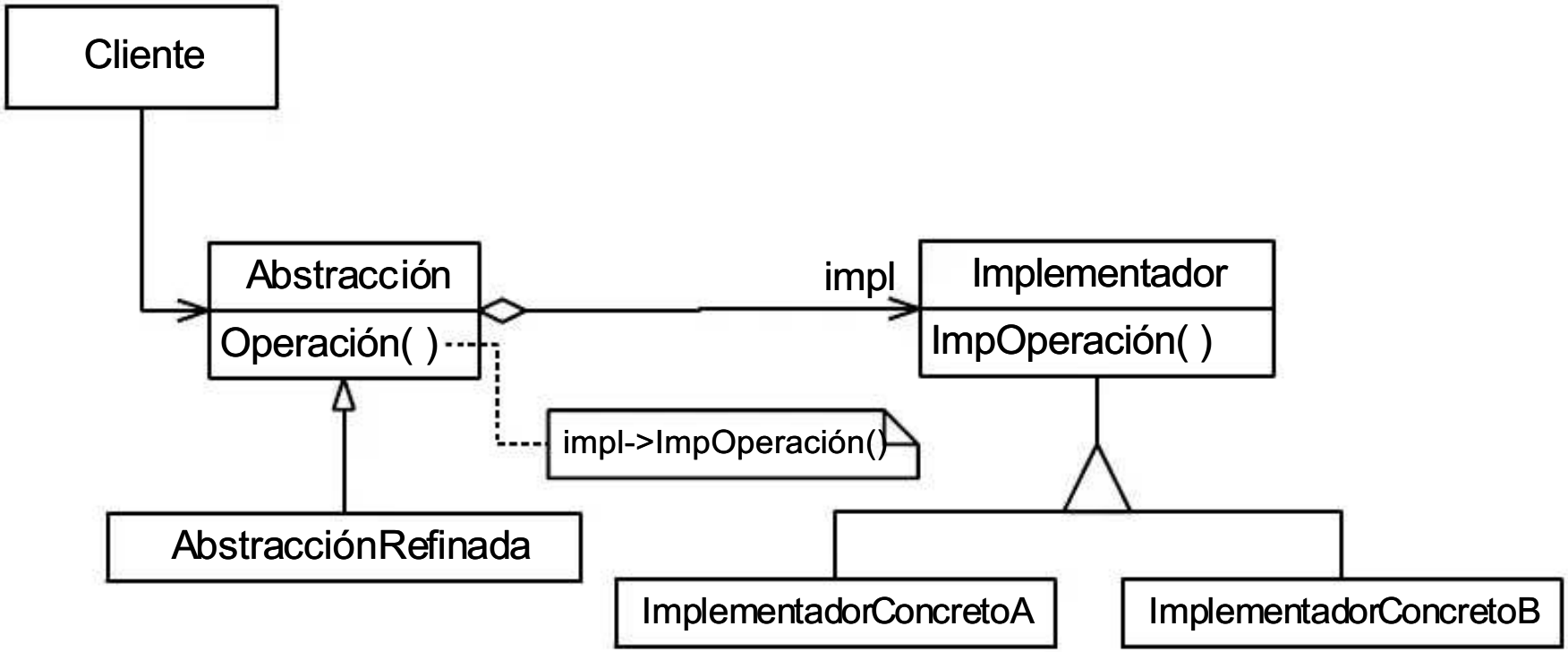


Puente (iii)

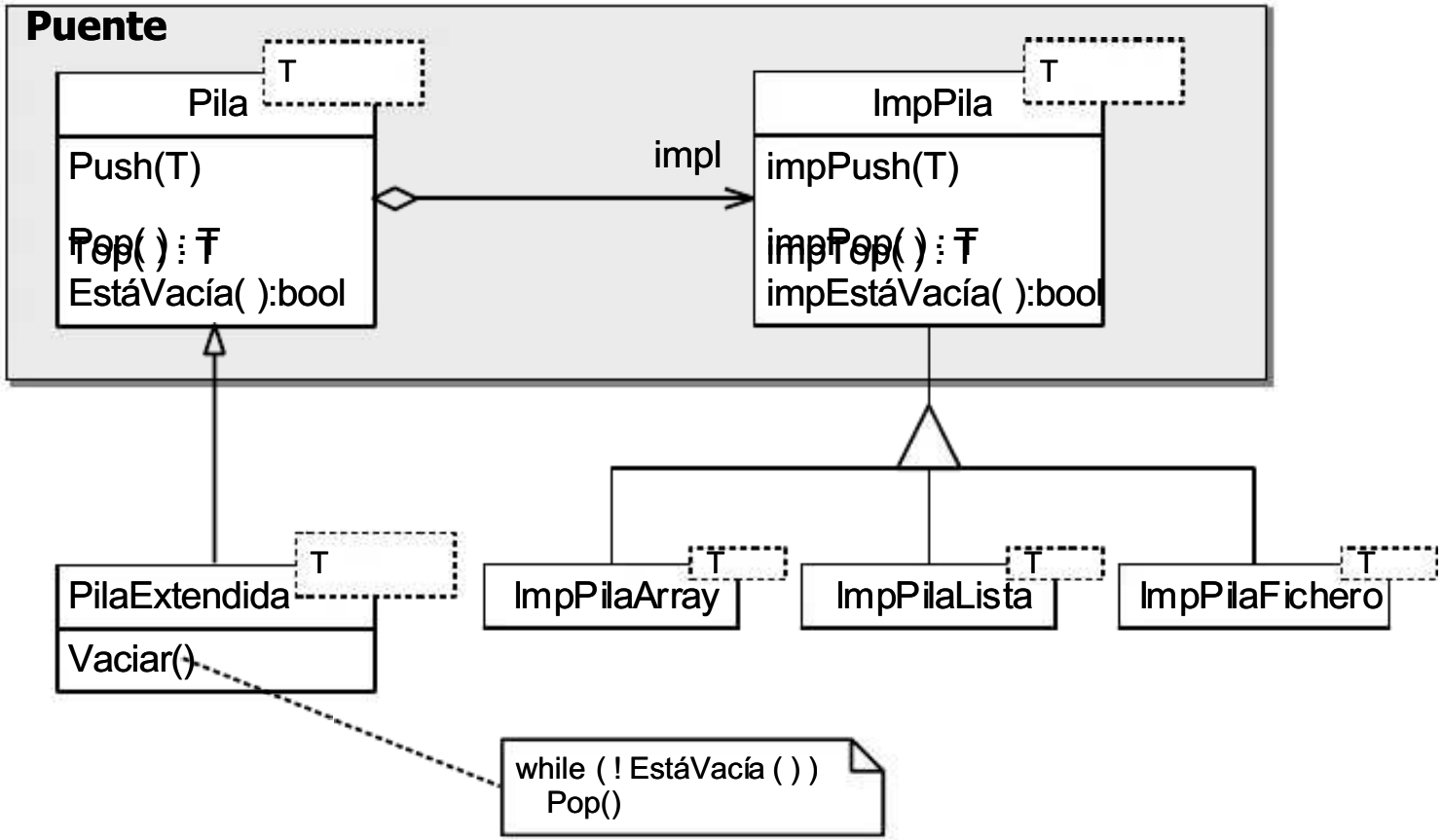
- Aplicabilidad
  - Si se quiere evitar un enlace permanente entre una abstracción y su implementación
  - Tanto la abstracción como la implementación se quiere que sean extensibles por derivación
  - Se quiere aislar a los clientes de cambios en la implementación
  - Se quiere compartir una implementación entre múltiples entidades

Puente (iv)

- Solución



Puente (v)



Puente (vi)

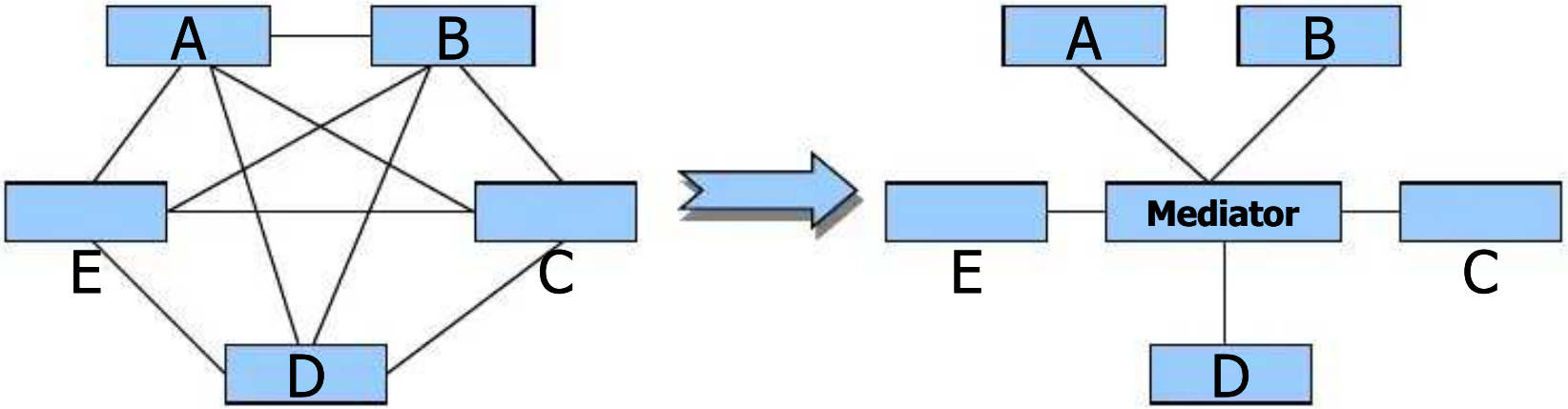
- Consecuencias
  - Desacopla la interfaz y la implementación, pudiendo configurarse o cambiarse en tiempo de ejecución
  - Favorece la división en niveles de un sistema
  - Ambas jerarquías pueden extenderse por herencia independientemente
  - Oculta a los clientes los detalles de implementación

Mediador (i)

- Patrón Mediador (*Mediator*) – Patrón de comportamiento [Gamma et al., 1995]
  - Los patrones de comportamiento están relacionados con los algoritmos y la asignación de responsabilidades. No describen tan sólo patrones de objetos y clases, sino patrones de comunicación entre ellos
- Su objetivo es definir un objeto que encapsule como interactúan un conjunto de objetos, evitando que tengan que conocerse entre ellos y permitiendo cambiar la interacción de forma independiente

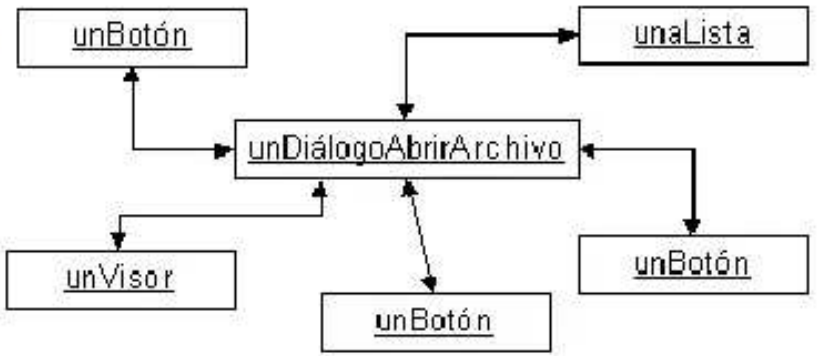
Mediador (ii)

- Problema (i)
  - Que un objeto encapsule todo su comportamiento suele ser adecuado, salvo en aquellos casos que suponga un número excesivo de enlaces



Mediador (iii)

- Problema (ii)
  - Un cuadro de diálogo para abrir un archivo tiene muchos objetos interdependientes: se evita derivar cada uno

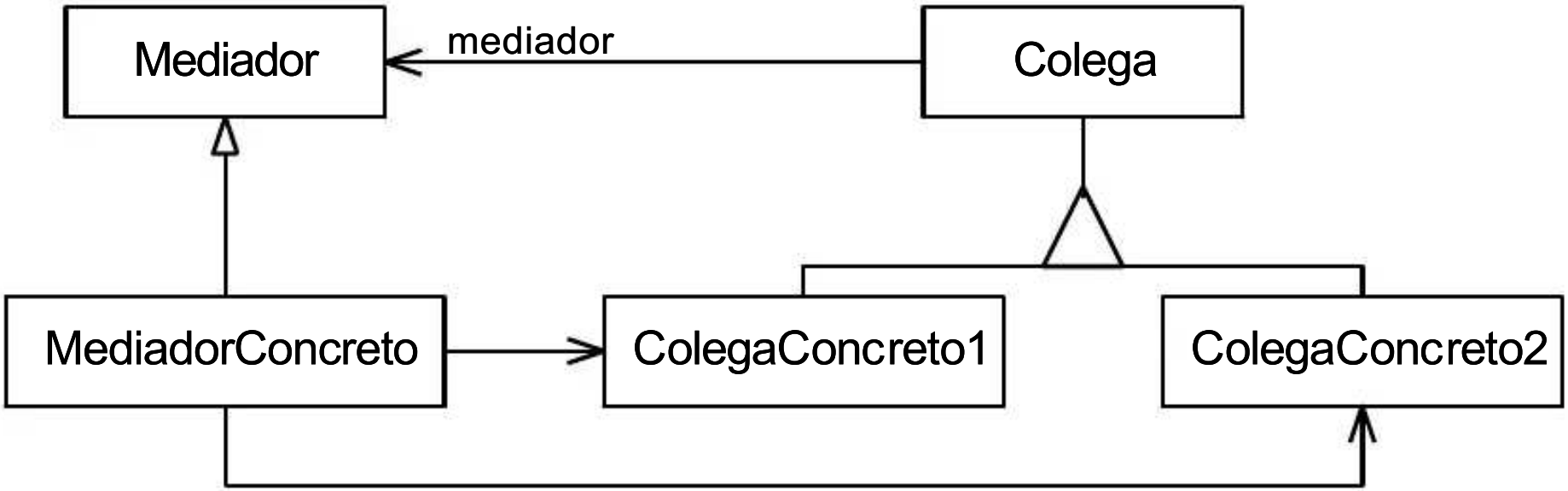


Mediador (iv)

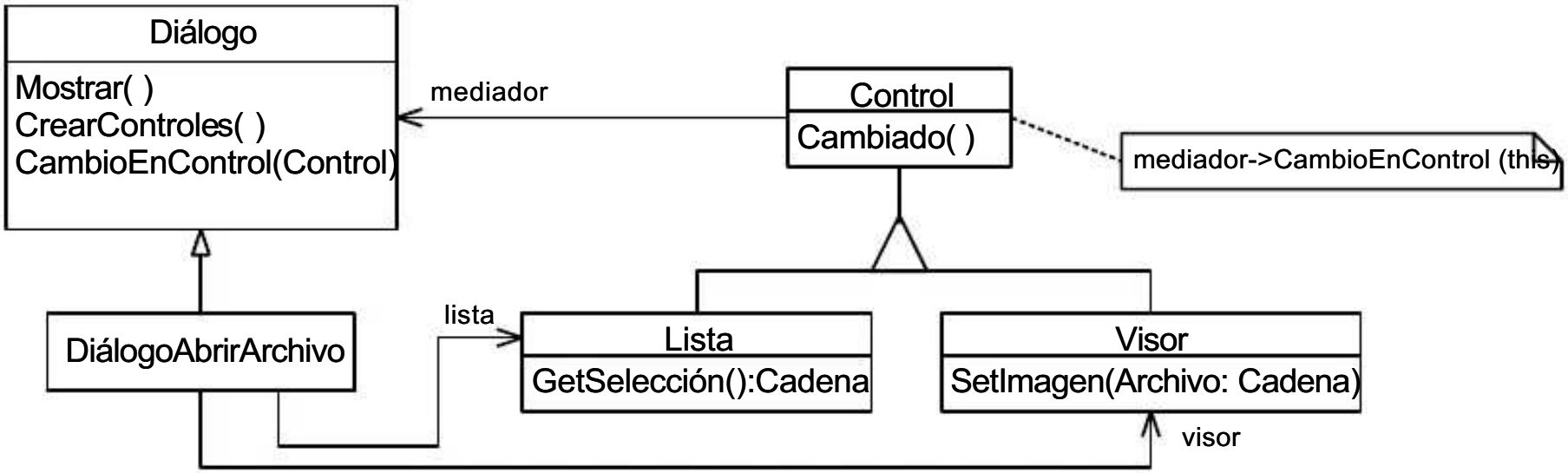
- Aplicabilidad
  - Cuando un conjunto de objetos se comunica de una forma bien definida pero compleja (con interdependencias complicadas)
  - Cuando reutilizar una entidad sea difícil por tener referencias a muchos objetos por comunicarse con ellos
  - Cuando un comportamiento distribuido entre varias clases debe ser adaptado sin tener que derivar muchas clases

Mediador (v)

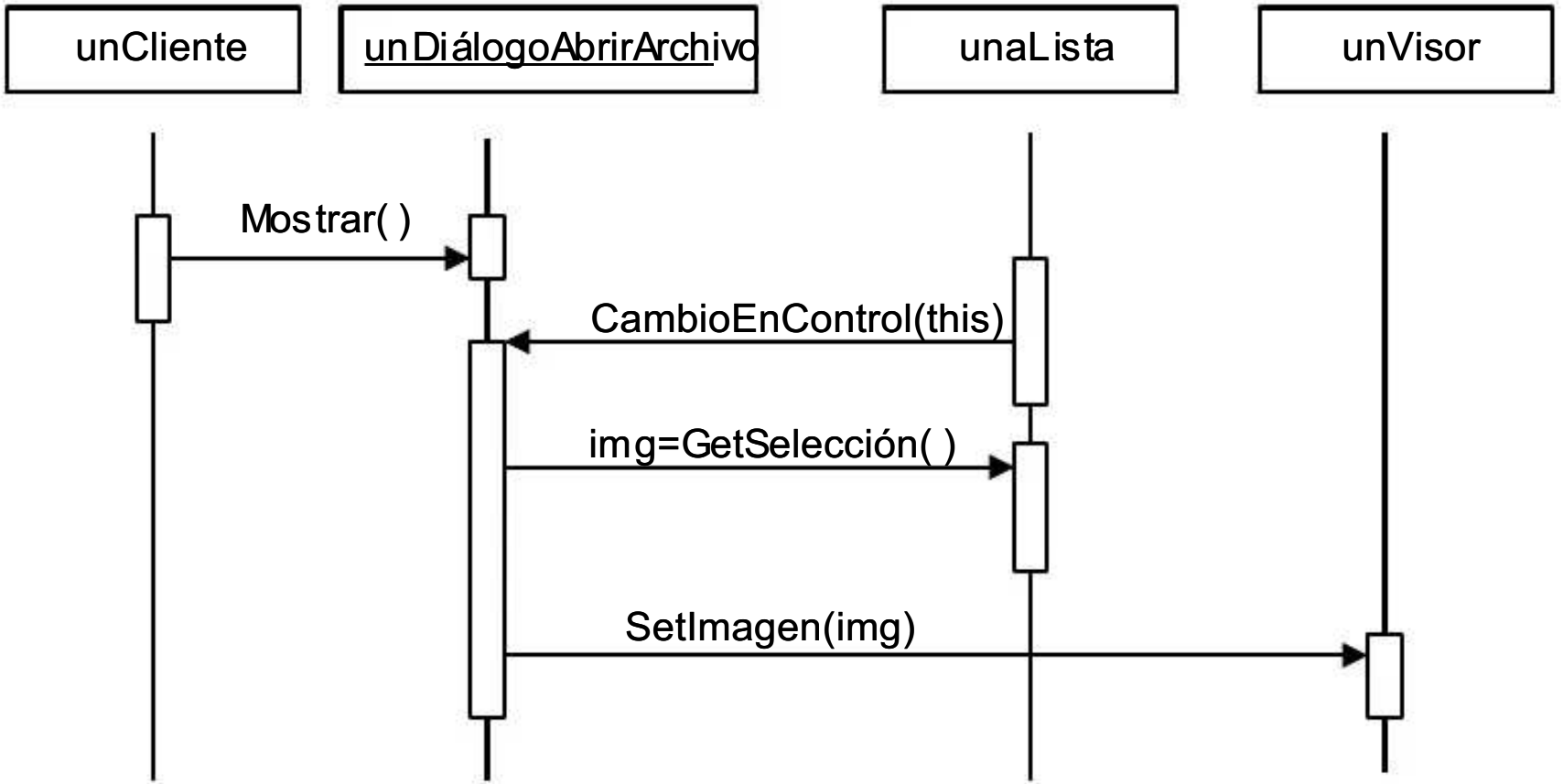
- Solución



Mediador (vi)



Mediador (vii)



Mediador (viii)

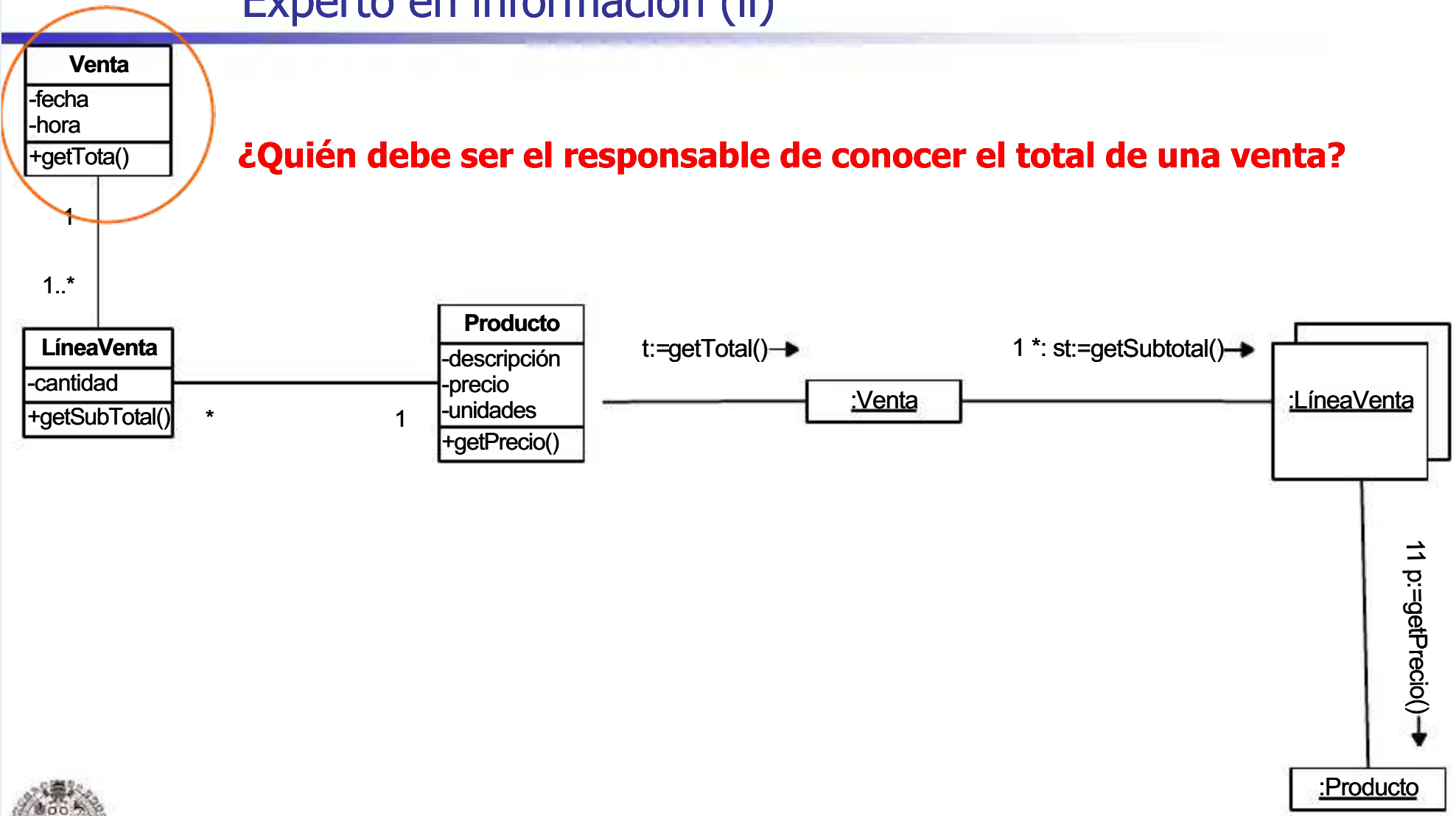
- Ventajas
  - Evita derivar nuevas clases de colegas
    - Para cambiar la conducta sólo hay que derivar un nuevo mediador
  - Desacopla colegas, permitiendo que cambien independientemente
  - Simplifica los protocolos de los objetos, al cambiar las asociaciones m:n por 1:n, más sencillas de entender, mantener y ampliar
  - Abstrae la cooperación entre los objetos y la encapsula en el mediador, siendo más fácil entender cómo funciona el sistema
- Inconveniente
  - Centraliza el control en el mediador, que es un objeto complejo y difícil de entender

Experto en información (i)

- Patrón GRASP [Larman, 2002]
- Clasificado como aspecto fundamental de diseño
- Problema
  - ¿Cuál es un principio general para asignar responsabilidades a los objetos?
    - Un modelo de diseño puede definir numerosas clases y una aplicación puede requerir que se realicen numerosas responsabilidades
    - Durante el diseño de objetos, cuando se definen las interacciones entre los objetos, se toman decisiones sobre la asignación de responsabilidades a las clases
      - Si esta asignación se hace bien, los sistemas tienden a ser más fáciles de entender, mantener y amplia, existiendo más oportunidades para reutilizar componentes en futuras aplicaciones
- Solución
  - Asignar una responsabilidad al experto en información, esto es, la clase que tiene la **información** necesaria para realizar la responsabilidad

Experto en información (ii)

¿Quién debe ser el responsable de conocer el total de una venta?



Experto en información (iii)

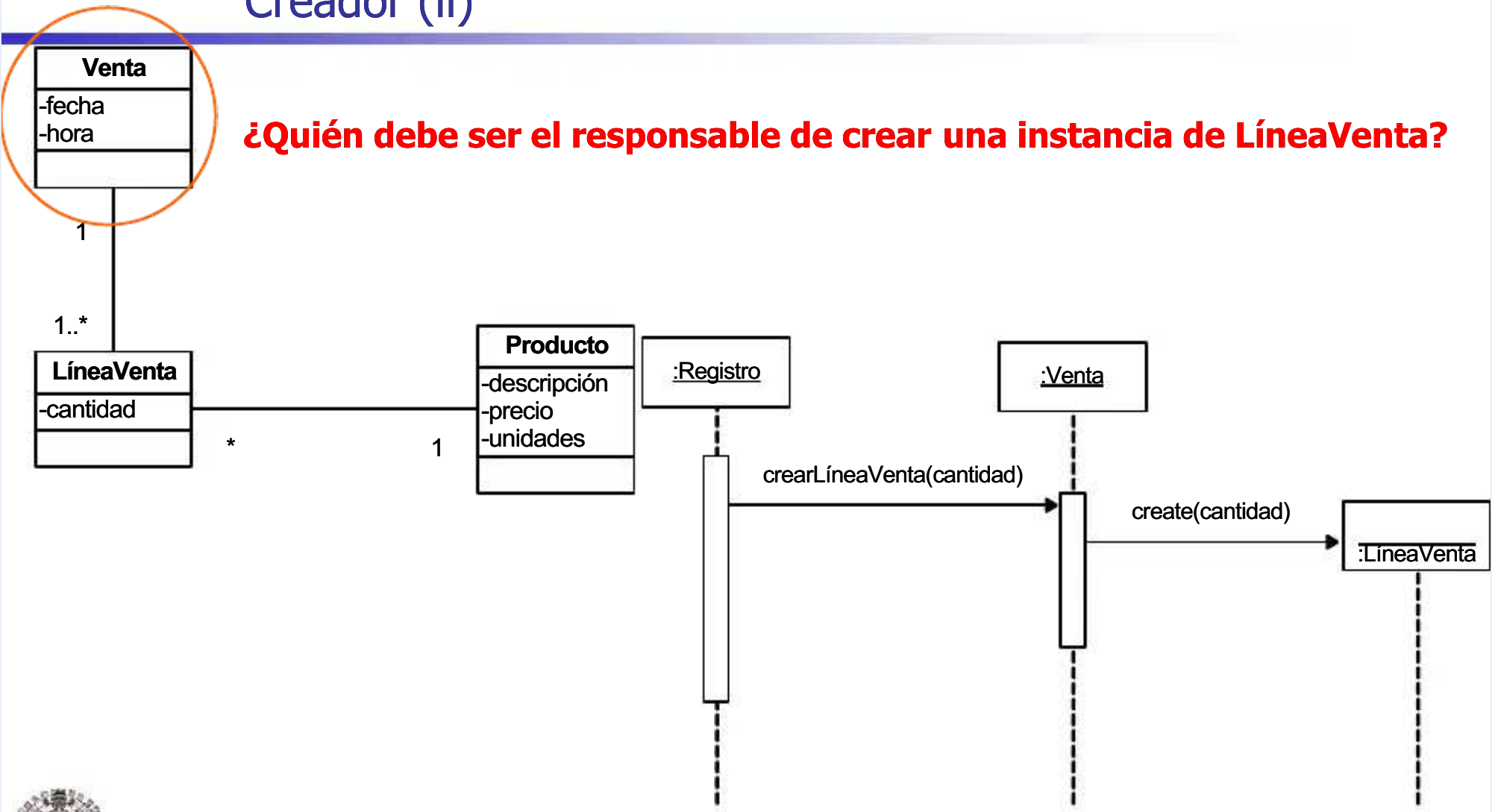
- Contraindicaciones
  - En ocasiones la solución propuesta por este patrón no es deseable por problemas de cohesión y acoplamiento
- Beneficios
  - Se mantiene el encapsulamiento de la información
  - Se distribuye el comportamiento entre las clases que contienen la información requerida
  - Bajo acoplamiento
  - Alta cohesión

Creador (i)

- Patrón GRASP [Larman, 2002]
- Clasificado como aspecto fundamental de diseño
- Problema
  - ¿Quién debería ser el responsable de la creación de una nueva instancia de alguna clase?
    - La creación de instancias es una de las actividades más comunes en un sistema orientado a objetos
    - Es útil contar con un principio general para la asignación de las responsabilidades de creación
      - Si se asignan bien, el diseño puede soportar un bajo acoplamiento, mayor claridad, encapsulación y reutilización
- Solución
  - Asignar a la clase B la responsabilidad de crear una instancia de la clase A si se cumple uno o más de los siguientes casos, donde B sería un creador de objetos de A
    - B agrega objetos de A
    - B contiene objetos de A
    - B registra objetos de A
    - B utiliza más estrechamente objetos de A
    - B tiene los datos de inicialización que se pasarán a un objeto A cuando sea creado (B es un Experto con respecto a la creación de A)

Creador (ii)

¿Quién debe ser el responsable de crear una instancia de LíneaVenta?



Creador (iii)

- Contraindicaciones
  - Frecuentemente, la creación requiere una complejidad significativa, como utilizar instancias recicladas por motivos de rendimiento, crear condicionalmente una instancia a partir de clases similares basados en el valor de alguna propiedad externa…
  - En estos casos es aconsejable delegar la creación a una clase auxiliar denominada Factoría [Gamma et al., 1995] en lugar de utilizar la clase sugerida por el Creador
- Beneficios
  - Bajo acoplamiento

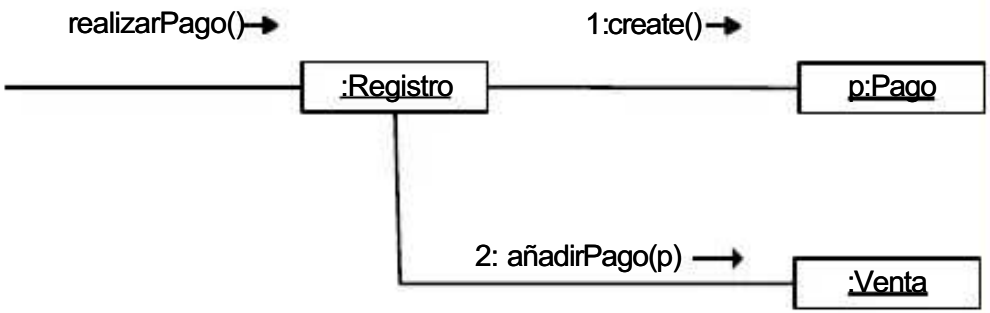
Bajo acoplamiento (i)

- Patrón GRASP [Larman, 2002]
- Clasificado como aspecto fundamental de diseño
- Problema
  - ¿Cómo soportar bajas dependencias, bajo impacto del cambio e incremento de la reutilización?
    - El acoplamiento es una medida de la fuerza con que un elemento *está conectado a*, *tiene conocimiento de*, *confía en*, otros elementos
    - Un elemento con bajo (o débil) acoplamiento no depende de demasiados elementos
    - Una clase con alto (o fuerte) acoplamiento confía en otras clases. Tales clases podrían no ser deseables; algunas adolecen de los siguientes problemas
      - Los cambios en las clases relacionadas fuerzan cambios locales
      - Son difíciles de entender de forma aislada
      - Son difíciles de reutilizar porque su uso requiere la presencia adicional de las clases de las que depende
- Solución
  - Asignar una responsabilidad de manera que el acoplamiento permanezca bajo

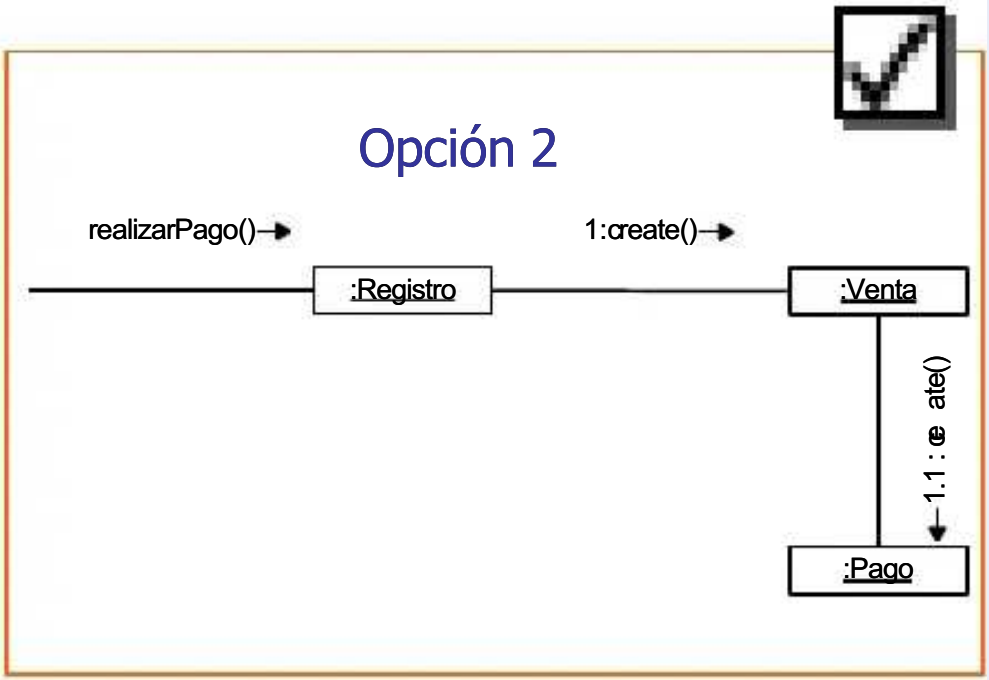
Bajo acoplamiento (ii)

Debe crearse una instancia de Pago y asociarla con Venta, ¿qué clase debería ser la responsable?

Opción 1



Opción 2



Bajo acoplamiento (iii)

- Contraindicaciones
  - No suele ser un problema el acoplamiento alto entre objetos estables y elementos generalizados
  - El alto acoplamiento en sí no es un problema; es el alto acoplamiento entre elementos que no son estables en alguna dimensión, como su interfaz, implementación o su mera presencia
- Beneficios
  - No afectan los cambios en otros componentes
  - Fácil de entender de manera aislada
  - Conveniente para reutilizar

Alta cohesión (i)

- Patrón GRASP [Larman, 2002]
- Clasificado como aspecto fundamental de diseño
- Problema
  - ¿Cómo mantener la complejidad manejable?
    - En el diseño orientado a objetos la cohesión (cohesión funcional) es una medida de la fuerza con la que se relacionan y del grado de focalización de las responsabilidades de un elemento
      - Existe una alta cohesión funcional cuando los elementos de un componente (clase) trabajan todos juntos para proporcionar algún comportamiento bien delimitado [Booch, 1994]
    - Un elemento con responsabilidades altamente relacionadas, y que no hace una gran cantidad de trabajo, tiene alta cohesión
    - Una clase con baja cohesión hace muchas cosas no relacionadas, o hace demasiado trabajo. Tales clases no son convenientes y adolecen de los siguientes problemas
      - Difíciles de entender
      - Difíciles de reutilizar
      - Difíciles de mantener
      - Delicadas, constantemente afectadas por los cambios
    - Con frecuencia las clases con baja cohesión representa un grano grande de abstracción, o se les ha asignado responsabilidad que deberían haberse delegado en otros objetos
- Solución
  - Asignar una responsabilidad de manera que la cohesión permanezca alta

Alta cohesión (ii)

- Contraindicaciones
  - Existen pocos casos en los que esté justificada la aceptación de baja cohesión
  - Un caso de componentes con baja cohesión lo constituyen los objetos servidores distribuidos, ya que suele ser deseable crear menos objetos servidores, de mayor tamaño y menos cohesivos, que proporcionen una interfaz para muchas operaciones
    - Menos llamadas remotas
    - Mejor rendimiento
- Beneficios
  - Se incrementa la claridad y facilita la comprensión del diseño
  - Se simplifican el mantenimiento y las mejoras
  - Se soporta a menudo bajo acoplamiento
  - El grano fino de funcionalidad altamente relacionada incrementa la reutilización porque una clase cohesiva se puede utilizar para un propósito muy específico

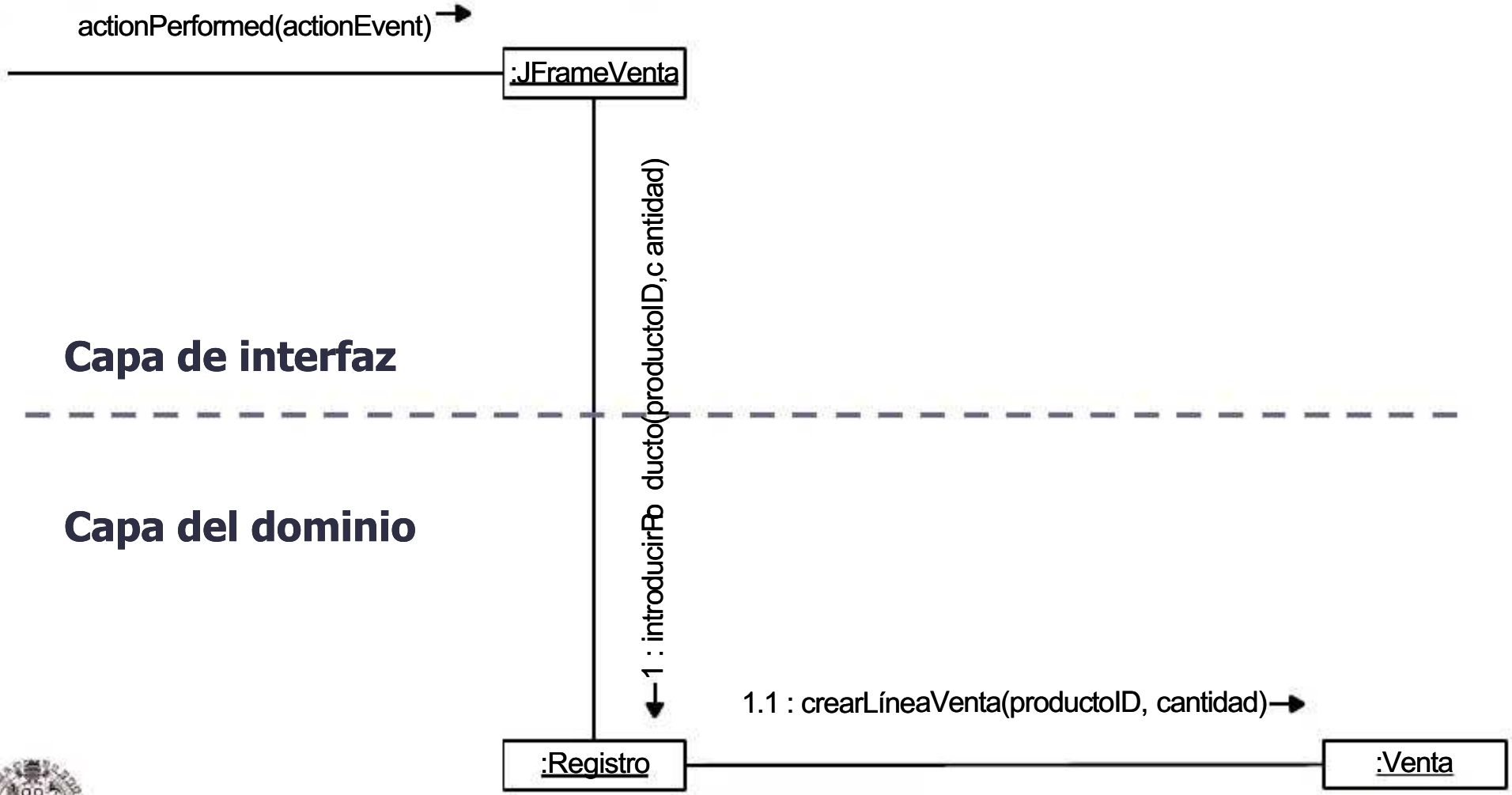
Controlador (i)

- Patrón GRASP [Larman, 2002]
- Clasificado como aspecto fundamental de diseño
- Problema
  - ¿Quién debe ser el responsable de gestionar un evento de entrada al sistema?
    - Un evento de entrada es un evento generado por un actor externo
      - Se asocian con operaciones del sistema tal como se relacionan los mensajes y los métodos
    - Un controlador es un objeto que no pertenece a la interfaz de usuario, responsable de recibir o manejar un evento del sistema
      - Un controlador define el método para la operación del sistema

Controlador (ii)

- Solución
  - Asignar la responsabilidad de recibir o manejar un mensaje de evento del sistema a una clase que representa alguna de las siguientes opciones
    - Representa el sistema global, dispositivo o subsistema (controlador de fachada)
    - Representa un escenario de caso de uso en el que tiene lugar el evento del sistema, a menudo denominado <NombreCasoUso>Manejador, <NombreCasoUso>Coordinador o <NombreCasoUso>Sesión (controlador de sesión o de caso de uso)
      - Debe utilizarse la misma clase controlador para todos los eventos del sistema en miso escenario de caso de uso
      - Informalmente, una sesión es una instancia de conversación con un actor. Las sesiones pueden tener cualquier duración, pero se organizan con frecuencia en función de los casos de uso (sesiones de casos de uso)
    - Nótese que la clases "Ventana", "Applet", "Widget", "Vista" o "Documento" no están en esta lista. Tales clases no deberían abordar tareas relacionadas con los eventos del sistema, normalmente, reciben estos eventos y los delegan a un controlador

Controlador (iii)



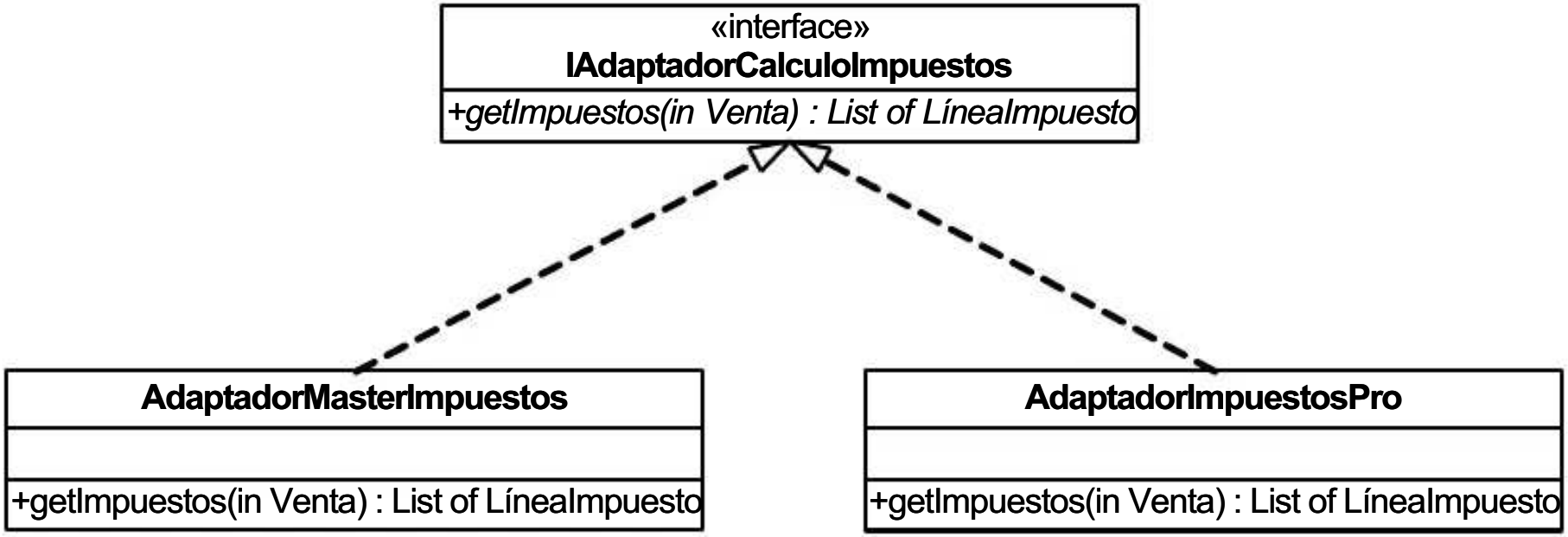
Controlador (iv)

- Beneficios
  - Aumenta el potencial para reutilizar y las interfaces conectables
    - Asegura que la lógica de la aplicación no se maneja en la capa de la interfaz
    - Un diseño de una interfaz como controlador reduce la oportunidad de reutilizar la lógica en futuras aplicaciones, puesto que está ligada a una interfaz particular
  - Razonamiento sobre el estado de los casos de uso
    - A veces es necesario asegurar que las operaciones del sistema tienen lugar en una secuencia válida, o ser capaces de razonar sobre el estado actual de la actividad y operaciones del caso de uso que está en marcha
      - Si es así, es necesario capturar en algún sitio esta información de estado; el controlador es una opción razonable, especialmente si el mismo controlador se utiliza en todo el caso de uso, que es la opción más recomendable

Polimorfismo (i)

- Patrón GRASP [Larman, 2002]
- Clasificado como aspecto fundamental de diseño
- Problemas
  - ¿Cómo manejar las alternativas basadas en el tipo?
    - La variación condicional es fundamental en el desarrollo de software
    - Si se diseña una programa utilizando sentencias de la lógica condicional, en el momento que aparezca una nueva variación, se requerirán modificaciones en la lógica de los casos
    - Esto dificulta que el software se extienda con facilidad como nuevas variaciones porque se tiende a necesitar cambios en varios sitios
  - ¿Cómo crear componentes software conectables?
    - Viendo los componentes en las relaciones cliente-servidor, ¿cómo se puede sustituir un componente servidor por otro, sin afectar al cliente?
- Solución
  - Cuando las alternativas o comportamientos relacionados varían según el tipo (clase) se debe asignar la responsabilidad para el comportamiento (utilizando operaciones polimórficas) a los tipos para los que varía el comportamiento
  - No deben hacerse comprobaciones acerca del tipo del objeto y no debe utilizarse la lógica condicional para llevar a cabo alternativas diferentes basadas en el tipo

Polimorfismo (ii)



Polimorfismo (iii)

- Contraindicaciones
  - Algunas veces, los desarrolladores diseñan sistemas con interfaces y polimorfismo para futuras necesidades especulativas frente a posibles variaciones desconocidas
    - Es conveniente una evaluación crítica
- Beneficios
  - Se añaden fácilmente las extensiones necesarias para nuevas variaciones
  - Las nuevas implementaciones se pueden introducir sin afectar a los clientes

Fabricación pura (i)

- Patrón GRASP [Larman, 2002]
- Clasificado como aspecto avanzado de diseño
- Problema
  - ¿Qué objetos deberían tener la responsabilidad cuando no se quiere violar los objetivos de Alta cohesión y Bajo acoplamiento, u otros, pero las soluciones que ofrece el Experto no son adecuadas?
    - Los diseños orientados a objetos algunas veces se caracterizan por implementar como clases software las representaciones de los conceptos del dominio del mundo real para reducir el salto en la representación
    - Sin embargo, hay muchas veces en las que la asignación de responsabilidades sólo a las clases software de la capa de dominio da lugar a problemas en cuanto a cohesión y acoplamiento pobres, lo cual provoca un potencial de reutilización bajo
- Solución
  - Asignar un conjunto de responsabilidades altamente cohesivo a una dase artificial o de conveniencia que no representa un concepto del dominio del problema
    - Algo inventado para soportar alta cohesión, bajo acoplamiento y reutilización
  - Esta clase es una *fabricación* de la imaginación
  - Idealmente las responsabilidades asignadas a esta fabricación soportan alta cohesión y bajo acoplamiento, de manera que el diseño de la fabricación es muy limpio o puro

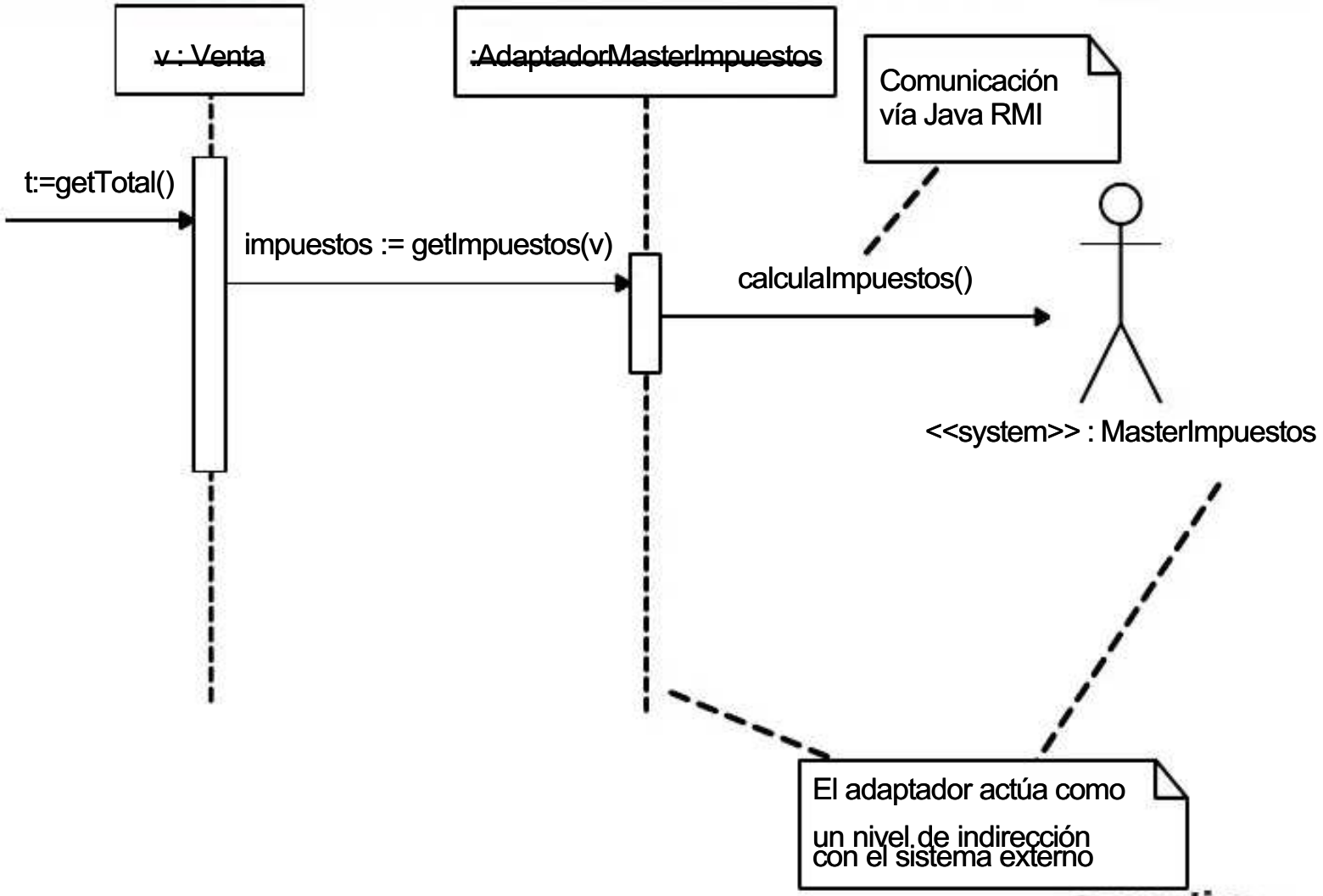
Fabricación pura (ii)

- Contraindicaciones
  - Algunas veces se abusa de la descomposición en comportamiento en los objetos de Fabricación pura
    - Exagerando, sucede que las funciones se convierten en objetos
    - El síntoma habitual es que la mayoría de los datos contenidos en los objetos se pasan a otros objetos para trabajar con ellos
- Beneficios
  - Se soporta **Alta cohesión** puesto que las responsabilidades se factorizan en una clase de grano fino que sólo se centra en un conjunto muy específico de tareas relacionadas
  - El potencial para reutilizar puede aumentar debido a la presencia de clases de Fabricación pura de grano fino, cuyas responsabilidades tienen aplicación en otras aplicaciones

Indirección (i)

- Patrón GRASP [Larman, 2002]
- Clasificado como aspecto avanzado de diseño
- Problemas
  - ¿Dónde asignar una responsabilidad para evitar el acoplamiento directo entre dos (o más) clases?
  - ¿Cómo desacoplar los objetos de manera que se soporte el bajo acoplamiento y el potencial para reutilizar permanezca alto?
- Solución
  - Asignar responsabilidades a un objeto intermedio que medie entre otros componentes o servicios de manera que no se acoplen directamente
  - El intermediario crea una indirección entre los otros componentes
- Beneficio
  - Disminuir el acoplamiento entre los componentes

Indirección (ii)



Variaciones protegidas (i)

- Patrón GRASP [Larman, 2002]
- Clasificado como aspecto avanzado de diseño
- Problema
  - ¿Cómo diseñar objetos, subsistemas y sistemas de manera que las variaciones o inestabilidades en estos elementos no tengan un impacto no deseable en otros elementos?
- Solución
  - Identificar los puntos de variación previstos o de inestabilidad
  - Asignar responsabilidades para crear una interfaz estable alrededor de ellos

Variaciones protegidas (ii)

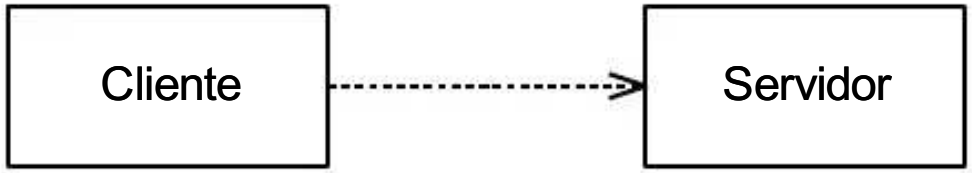
- Contraindicaciones
  - Si la necesidad de flexibilidad y protección de cambios es realista, entonces está motivada la aplicación de este patrón
  - Pero si la probabilidad de reutilización es muy incierta, entonces se requiere un modo de ver las cosas restringido y crítico
- Beneficios
  - Se añaden fácilmente las extensiones que se necesitan para nuevas variaciones
  - Se pueden introducir nuevas implementaciones sin afectar a los clientes
  - Se reduce el acoplamiento
  - Puede disminuirse el impacto o el coste de los cambios

Variaciones protegidas (iii)

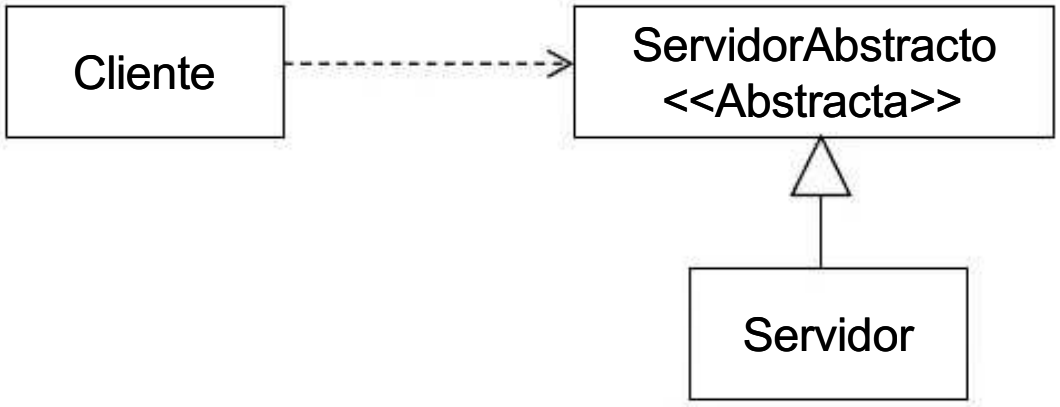
- El principio abierto-cerrado [Meyer, 1997] es esencialmente equivalente al patrón Variaciones Protegidas y a la ocultación de la información
  - Los módulos deberían ser tanto abiertos (para extenderse) como cerrados (a las modificaciones que afecten a los clientes) [Meyer, 1997]
- El principio abiertro-cerrado y el patrón Variaciones Protegidas son expresiones del mismo principio que resaltan diferentes puntos
  - Protección en los puntos de variación
  - Evolución

Variaciones protegidas (iv)

- La clave del principio abierto-cerrado está en la abstracción



Diseño que no se ajusta al principio abierto/cerrado



Diseño que cumple el principio abierto/cerrado

Variaciones protegidas (v)

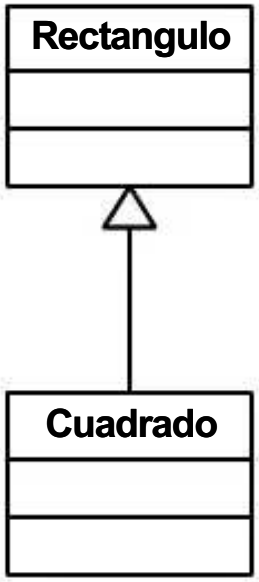
- El principio de sustitución de Liskov [Liskov, 1987] formaliza el principio de protección contra las variaciones en implementaciones diferentes de una interfaz, o una subclase que extiende una superclase
  - Lo que se quiere aquí es algo como la siguiente propiedad de sustitución: Si para cada objeto o1 de tipo S hay un objeto o2 de tipo T tal que para todos los programas P definidos en términos de T, el comportamiento de P no cambia cuando o2 es sustituido por o1,

    entonces S es un subtipo de T [Liskov, 1987]
    - Informalmente, el software (métodos, clases...) que hace referencia a un tipo T (alguna interfaz o superclase) debería trabajar correctamente o como se espera con cualquier implementación o subclase de T que la sustituya, llámese S

Variaciones protegidas (vi)

```
class Rectangulo {
        double alto, ancho;
public:
        Rectangulo(double _alto, double _ancho) {alto=_alto; ancho=_ancho;}
        virtual void EstableceAlto(double tmp)  {alto=tmp;}
        virtual void EstableceAncho(double tmp) {ancho=tmp;}
        double Alto()  const {return alto;}
        double Ancho() const {return ancho;}

};
```




```
class Cuadrado: public Rectangulo {
public:
        Cuadrado(double lado):Rectangulo(lado, lado){}
        virtual void EstableceAlto(double lado) {
                Rectangulo::EstableceAlto(lado);
                Rectangulo::EstableceAncho(lado);
        }
        virtual void EstableceAncho(double lado) {
                Rectangulo::EstableceAlto(lado);
                Rectangulo::EstableceAncho(lado);
        }
};
```

Variaciones protegidas (vii)

```
void main (void) {
        Cuadrado c1(2);
        cout << c1.Alto() << " " << c1.Ancho() << endl; //Salida: 2 2

        c1.EstableceAlto(5);
        cout << c1.Alto() << " " << c1.Ancho() << endl; //Salida 5 5

        c1.EstableceAncho(10);
        cout << c1.Alto() << " " << c1.Ancho() << endl; //Salida 10 10
}
```

Este programa funcionará sin problemas

Variaciones protegidas (viii)

- Pero, ¿qué sucedería con alguna función que recibiese una referencia o un puntero a un objeto **Rectangulo** y modificase su altura o su anchura?

```
void CambiaAspecto(Rectangulo &r) {
        r.EstableceAncho(r.Alto()*.5);
}
```

```
void main (void) {
        Cuadrado c1(2);
        cout << c1.Alto() << " " << c1.Ancho() << endl; //Salida: 2 2

        CambiaAspecto(c1);
        cout << c1.Alto() << " " << c1.Ancho() << endl; //Salida: 1 1
}
```

```
void CambiaAspecto(Rectangulo &r) {
        if (typeid(r) == typeid(Cuadrado))
                throw "\nNo se puede ajustar el aspecto de un cuadrado";

        r.EstableceAncho(r.Alto()*.5);
}
```

Variaciones protegidas (viiii)

- Conclusiones al ejemplo
  - Un modelo, visto por separado, no puede ser validado
  - La validez de un modelo sólo puede expresarse en términos de sus clientes
  - Un cuadrado puede ser un rectángulo, pero un objeto **Cuadrado** no es un objeto **Rectangulo** porque, en términos de comportamiento, el comportamiento de un objeto **Cuadrado** no es consistente con el comportamiento de un objeto **Rectangulo**
- Existe una fuerte relación entre el principio de Liskov y el concepto de diseño por contrato expuesto por Bertrand Meyer
  - "... cuando se redefine un método (en una clase derivada), sólo se puede reemplazar su precondición por una más suave, y su poscondición por una más fuerte" (Bertrand Meyer)

Patrón DAO (i)

- *Data Access Object* [Sun, 2002]
  - También conocido como *Data Access Component*
- Contexto
  -
  - El acceso al almacenamiento persistente, como una base de datos, varía en gran medida dependiendo del tipo de almacenamiento (bases de datos relacionales, bases de datos orientadas a objetos, ficheros planos...) y de la implementación del vendedor
- Aplicabilidad
  - Desacoplar la lógica de negocio de la lógica de acceso a datos, de manera que se pueda cambiar la fuente de datos fácilmente
  - Poder seleccionar el tipo de fuente de datos durante la instalación/configuración de una aplicación

El acceso a los datos varía dependiendo de la fuente de los datos

Patrón DAO (ii)

- Problema (i)
  - Muchas aplicaciones necesitan utilizar datos persistentes en algún momento
  - Para muchas de ellas, este almacenamiento persistente se implementa utilizando diferentes mecanismos
    - Hay marcadas diferencias en las APIS utilizadas para acceder a esos mecanismos de almacenamiento diferentes
  - Otras aplicaciones podrían necesitar acceder a datos que residen en sistemas diferentes
    - Por ejemplo, los datos podrían residir en sistemas *mainframe*, repositorios LDAP (*Lightweight Directory Access Protocol*)...
  - Otro ejemplo es donde los datos los proporcionan servicios a través de sistemas externos como los sistemas de integración negocio-a-negocio (B2B), servicios de tarjetas de crédito...

Patrón DAO (iii)

- Problema (ii)
  - Normalmente, las aplicaciones utilizan componentes distribuidos y compartidos (como los *beans* de entidad) para representar los datos persistentes
  - Las aplicaciones pueden utilizar APIs para acceder a los datos en un sistema de control de bases de datos relacionales
    - Estas APIs permiten una forma estándar de acceder y manipular datos en un almacenamiento persistente, como una base de datos relacional
      - Por ejemplo, la API JDBC permite a las aplicaciones J2EE utilizar sentencias SQL, que son el método estándar para acceder a tablas relacionales
      - Sin embargo, incluso dentro de un entorno relacional, la sintaxis y formatos actuales de las sentencias SQL podrían variar dependiendo de la propia base de datos en particular

Patrón DAO (iv)

- Problema (iii)
  - Incluso hay una mayor variación con diferentes tipos de almacenamientos persistentes
    - Los mecanismos de acceso, las APIs soportadas y sus características varían entre los diferentes tipos de almacenamientos persistentes, como bases de datos relacionales, bases de datos orientadas a objetos, ficheros planos...
    - Las aplicaciones que necesitan acceder a datos de un sistema legado o un sistema dispar (como un *mainframe* o un servicio B2B) se ven obligados a utilizar APIs que podrían ser propietarias
    - Dichas fuentes de datos dispares ofrecen retos a la aplicación y potencialmente pueden crear una dependencia directa entre el código de la aplicación y el código de acceso a los datos
    - Pero introducir el código de conectividad y de acceso a datos dentro de estos componentes genera un fuerte acoplamiento entre los componentes y la implementación de la fuente de datos
    - Dichas dependencias de código en los componentes hace difícil y tedioso migrar la aplicación de un tipo de fuente de datos a otro. Cuando cambia la fuente de datos, también deben cambiar los componentes para manejar el nuevo tipo de fuente de datos

Patrón DAO (v)

- Causas
  - Los componentes necesitan recuperar y almacenar información desde almacenamientos persistentes y otras fuentes de datos como sistemas legados, B2B, LDAP...
  - Las APIs para almacenamiento persistente varían dependiendo del vendedor del producto
    - Otras fuentes de datos podrían tener APIS que no son estándares y/o propietarias. Estas APIs y sus capacidades también varían dependiendo del tipo de almacenamiento - bases de datos relacionales, bases de datos orientadas a objetos, documentos XML, ficheros planos... Hay una falta de APIs uniformes para corregir los requisitos de acceso a sistemas tan dispares
  - Los componentes normalmente utilizan APIs propietarias para acceder a sistemas externos y/o legados para recuperar y almacenar datos
  - La portabilidad de los componentes se ve afectada directamente cuando se incluyen APIs y mecanismos de acceso específicos
  - Los componentes necesitan ser transparentes al almacenamiento persistente real o la implementación de la fuente de datos para proporcionar una migración sencilla a diferentes productos, diferentes tipos de almacenamiento y diferentes tipos de fuentes de datos

Patrón DAO (vi)

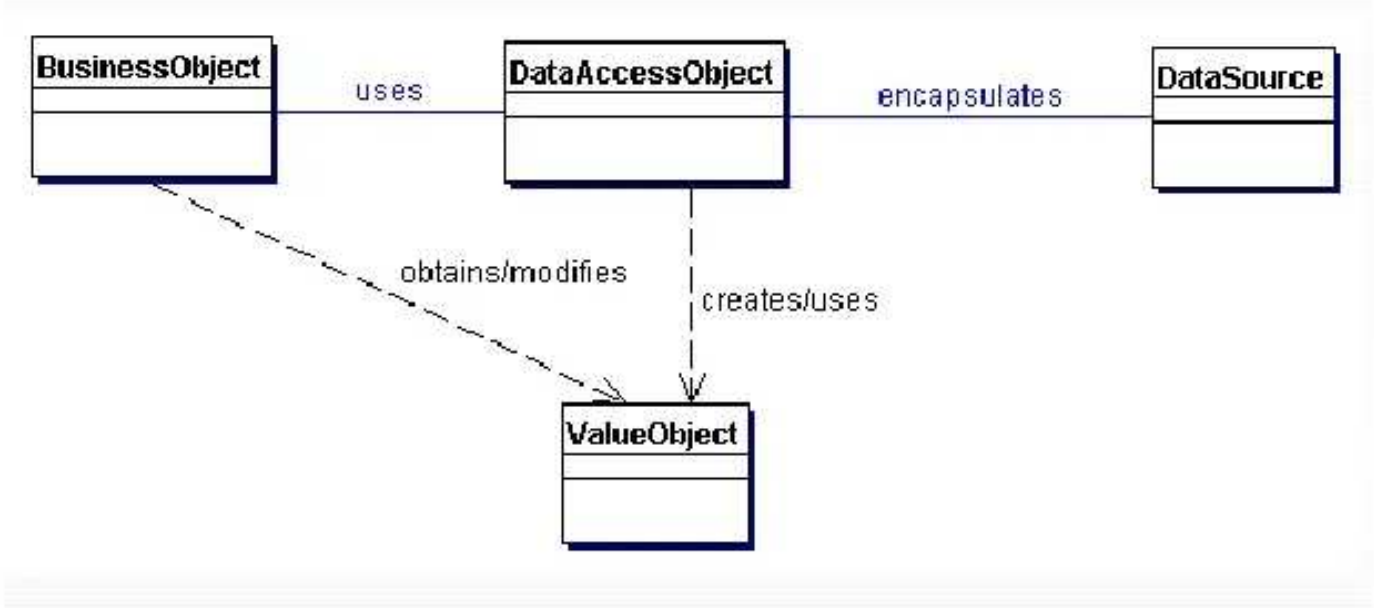
- Solución

Utilizar un *Data Access Object* (DAO) para abstraer y encapsular todos los accesos a la fuente de datos. El DAO maneja la conexión con la fuente de datos para obtener y almacenar datos

  - El DAO implementa el mecanismo de acceso requerido para trabajar con la fuente de datos
  - Esta fuente de datos puede ser un almacenamiento persistente como una base de datos relacional, un servicio externo como un intercambio B2B, un repositorio LDAP...
  - Los componentes de negocio que tratan con el DAO utilizan una interfaz simple expuesto por el DAO para sus clientes
  - El DAO oculta completamente los detalles de implementación de la fuente de datos a sus clientes
  - Como la interfaz expuesta por el DAO no cambia cuando cambia la implementación de la fuente de datos subyacente, este patrón permite al DAO adaptarse a diferentes esquemas de almacenamiento sin que esto afecte a sus clientes o componentes de negocio
  - Esencialmente, el DAO actúa como un adaptador entre el componente y la fuente de datos

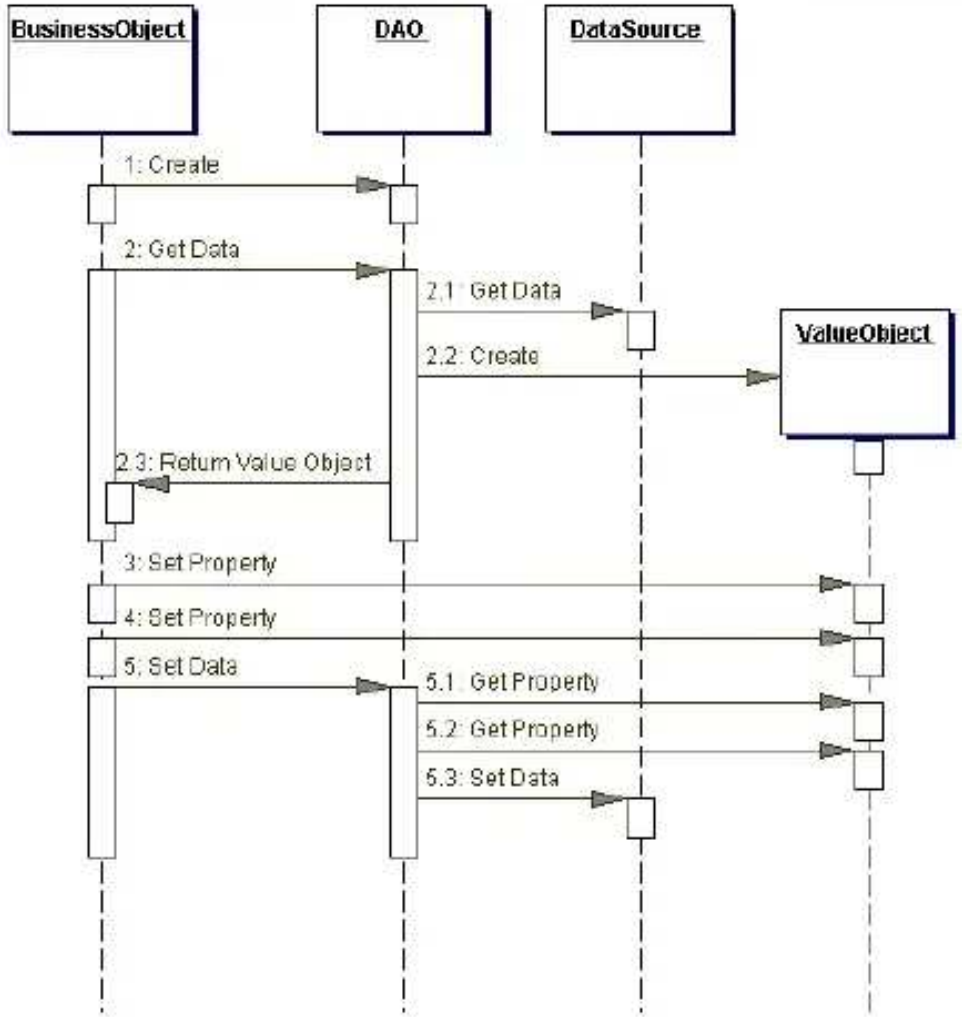
Patrón DAO (vii)

- Estructura



Patrón DAO (viii)

- Participantes y responsabilidades (i)

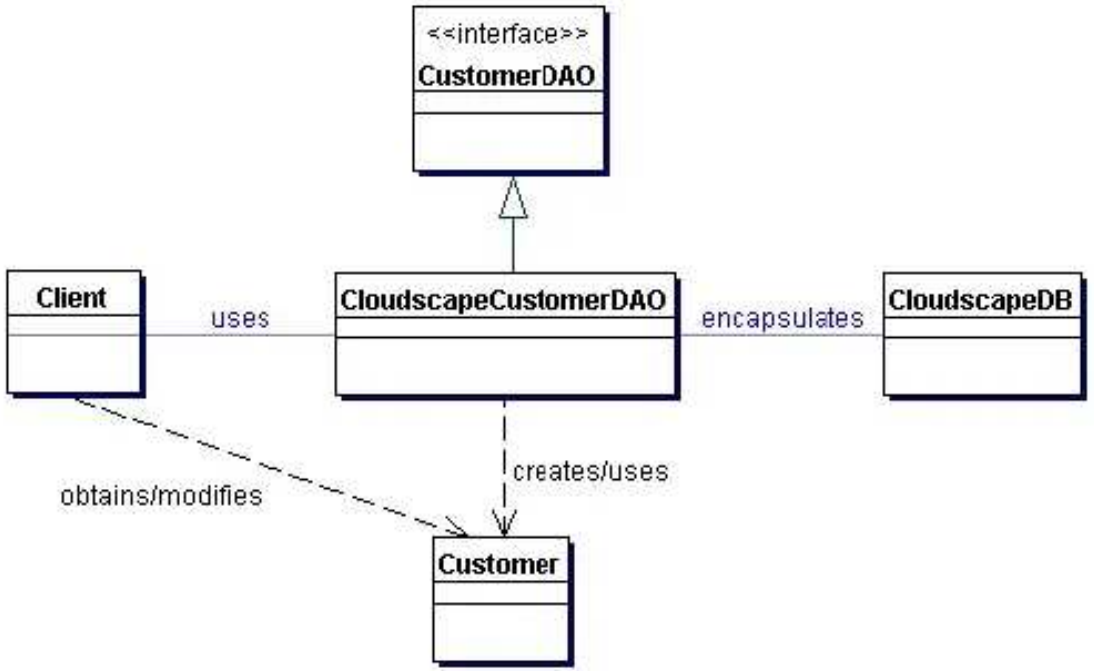


Patrón DAO (ix)

- Participantes y responsabilidades (ii)
  - **BusinessObject** representa los datos del cliente
    - Es el objeto que requiere el acceso a la fuente de datos para obtener y almacenar datos
  - **DataAccessObject** es el objeto principal de este patrón
    - Abstrae la implementación del acceso a datos subyacente al **BusinessObject** para permitirle un acceso transparente a la fuente de datos
    - El **BusinessObject** también delega las operaciones de carga y almacenamiento en el **DataAccessObject**
  - **DataSource** representa la implementación de la fuente de datos
    - Una fuente de datos podría ser una base de datos como una base de datos relacional u objetual, un repositorio XML, un fichero plano...
  - **ValueObject** representa un objeto utilizado para el transporte de datos
    - El **DataAccessObject** podría utilizar un **ValueObject** para devolver los datos al cliente
    - El **DataAccessObject** también podría recibir datos desde el cliente en un **ValueObject** para actualizar los datos en la fuente de datos

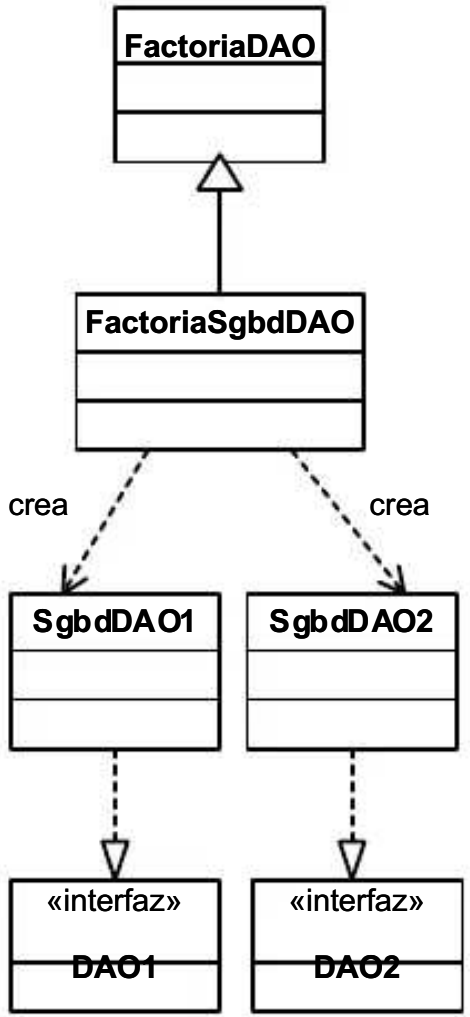
Patrón DAO (x)

- Ejemplo básico



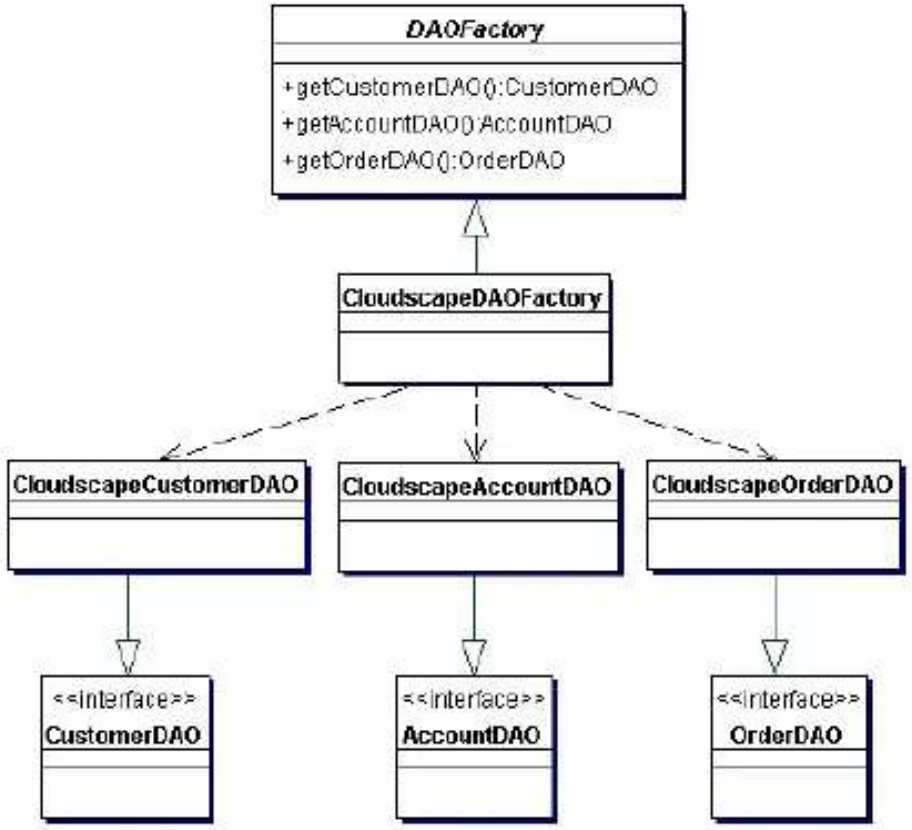
Patrón DAO (xi)

- Estrategia de factoría para DAOs (i)
  - El patrón DAO se puede flexibilizar adoptando los patrones **Abstract Factory** [Gamma et al., 1995] y **Factory Method** [Gamma et al., 1995]
  - Cuando el almacenamiento subyacente no está sujeto a cambios de una implementación a otra, esta estrategia se puede implementar utilizando el patrón **Factory Method** para producir el número de DAOs que necesita la aplicación



Patrón DAO (xii)

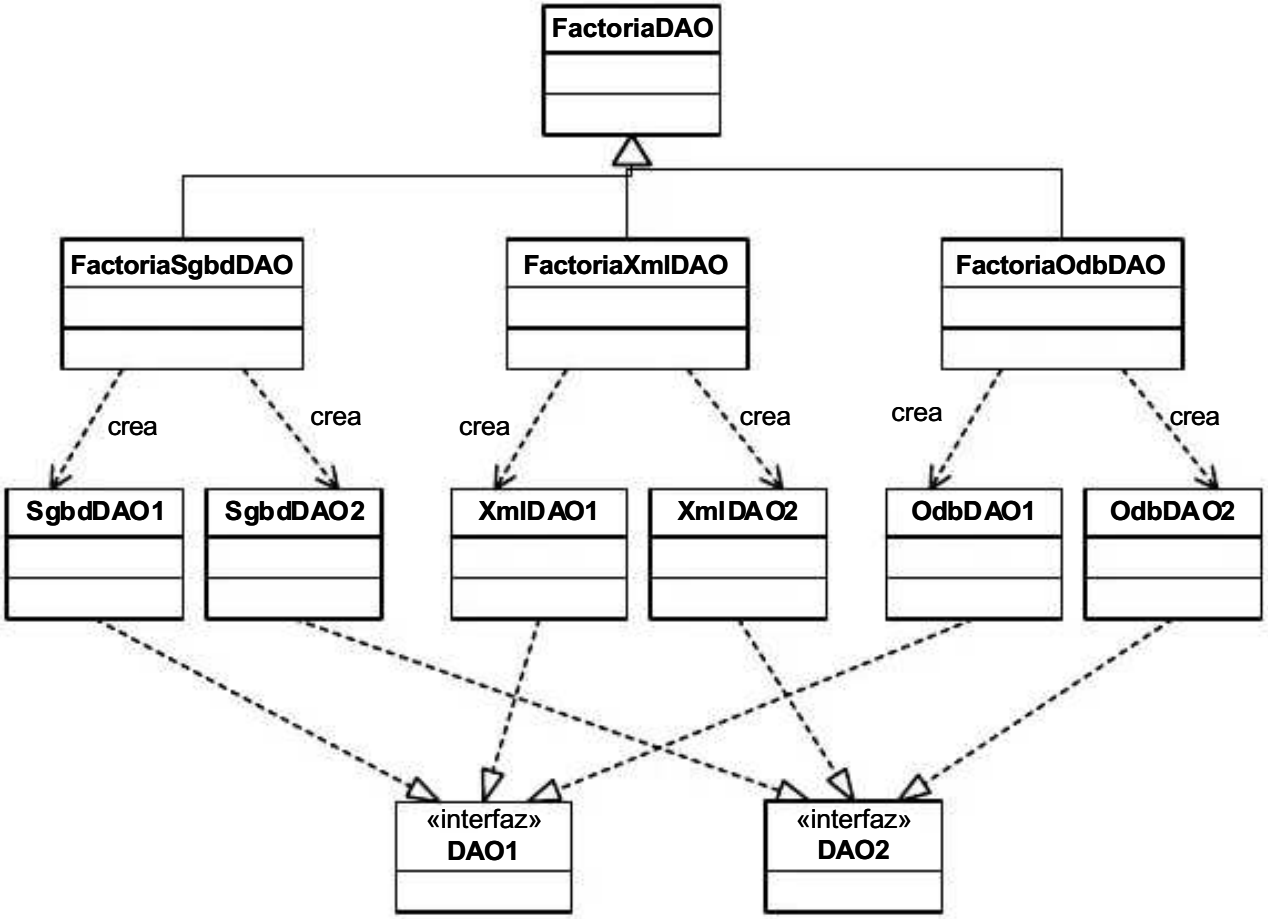
- Ejemplo de **Factory Method**
  - Se presenta un ejemplo donde se implementa una factoría DAO que produce varios DAOs para una única implementación de base de datos (por ejemplo MySQL)
  - La factoría produce DAOs como **CustomerDAO**, **AccountDAO**, **OrderDAO**…



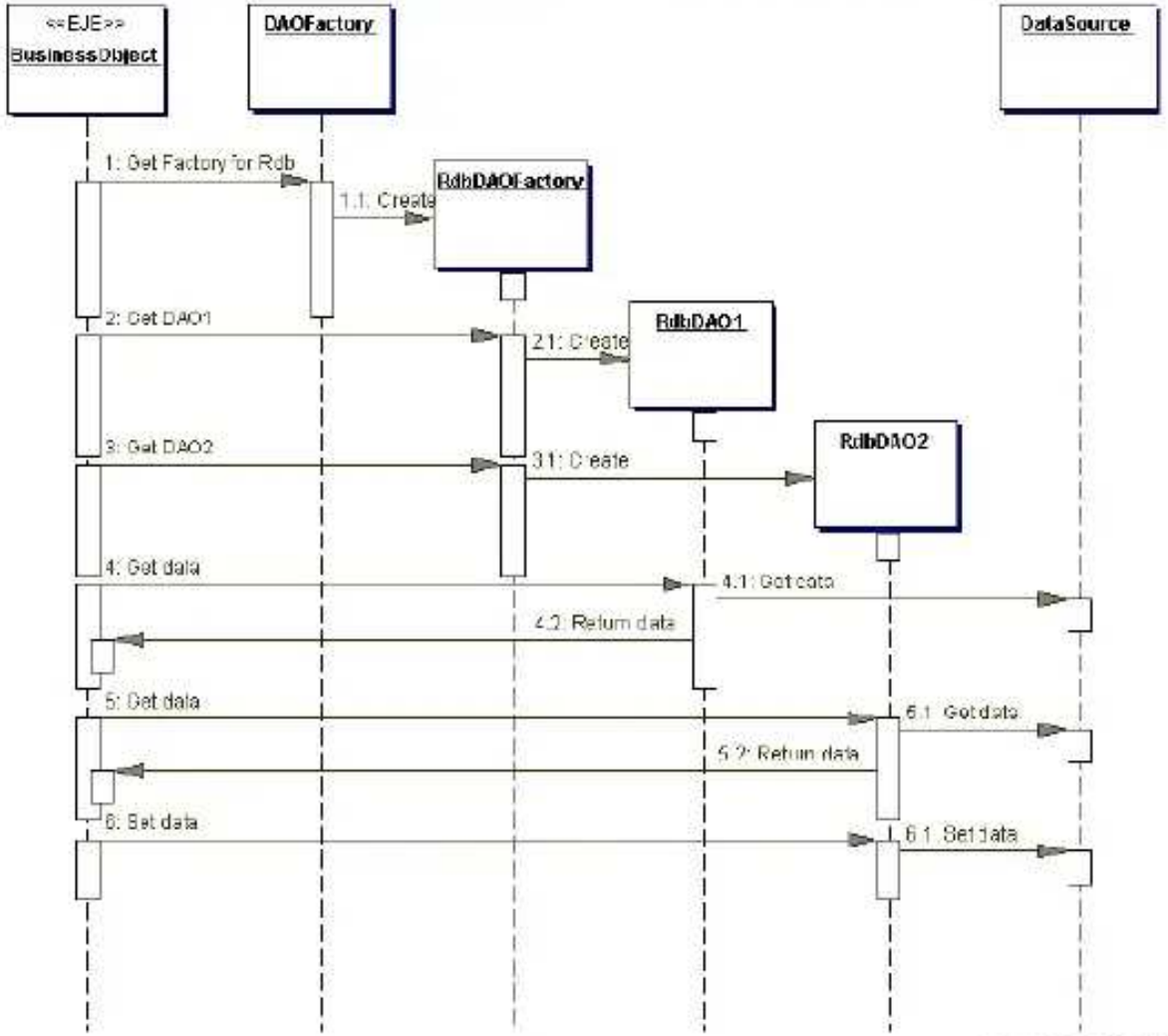
Patrón DAO (xiii)

- Estrategia de factoría para DAOs (ii)
  - Cuando el almacenamiento subyacente si está sujeto a cambios de una implementación a otra, esta estrategia se podría implementar usando el patrón **Abstract Factory**
  - Este patrón a su vez puede construir y utilizar la implementación **Factory Method**
  - En este caso, esta estrategia proporciona un objeto factoría abstracta de DAOs (**Abstract Factory**) que puede construir varios tipos de factorías concretas de DAOs, cada factoría soporta un tipo diferente de implementación del almacenamiento persistente
  - Una vez que se obtiene la factoría concreta de DAOs para una implementación específica, se utiliza para producir los DAOs soportados e implementados en esa implementación

Patrón DAO (xiv)

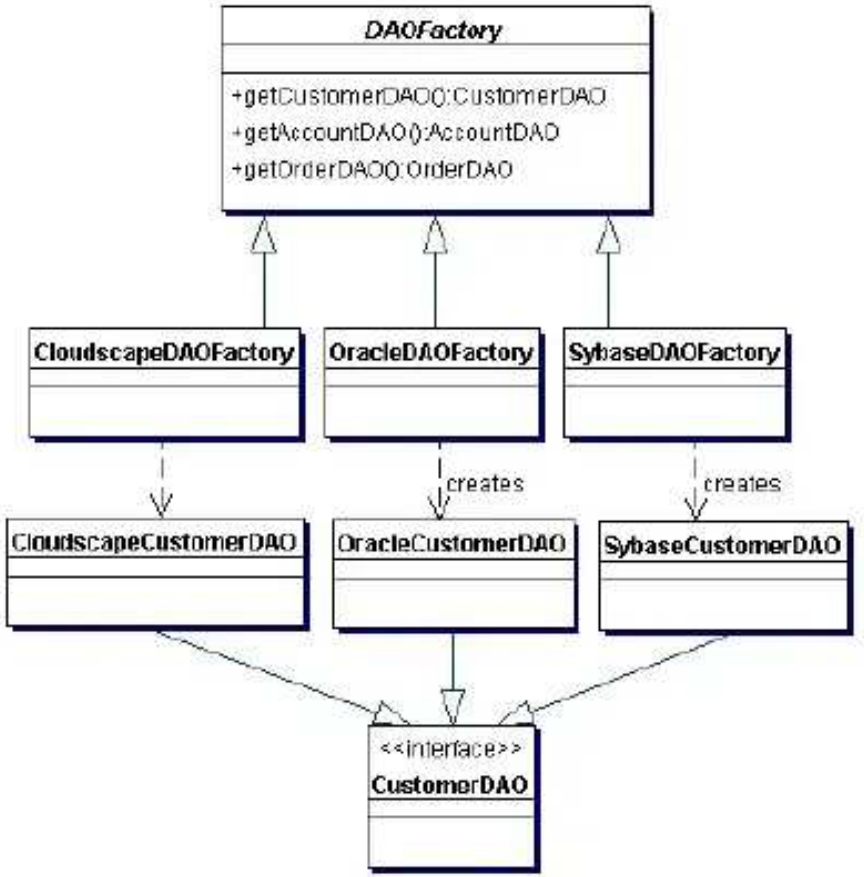


Patrón DAO (xv)



Patrón DAO (xvi)

- Ejemplo de **Abstract Factory**
  - Se presenta un ejemplo en el que se implementa este patrón para tres bases de datos distintas
  - Esta factoría produce DAOs tales como **CustomerDAO, AccountDAO, OrderDAO**…
  - Esta estrategía utiliza un **Factory Method** para las factorías producidas por la **Abstract Factory**



Patrón DAO (xvii)

- Consecuencias
  - Beneficios
    - Flexibilidad en la instalación/configuración de una aplicación
    - Independencia del vendedor de la fuente de datos
    - Independencia del recurso (BD relacional, BD orientada a objetos, ficheros planos, servidor LDAP...)
    - Realmente esto sólo lo conseguiremos con EJB, donde los DAOs no necesitan recibir la conexión en sus métodos
    - Extensibilidad
    - Reduce la complejidad de la implementación de la lógica de negocio
  - Riesgos
    - Más complejidad

5. Aportaciones principales del tema

Aportaciones principales

- Los objetos de un diseño orientado a objetos están relacionados con la solución del problema a resolver
- Los objetos del dominio de la solución incluyen: objetos de interfaz, objetos de aplicación y objetos base o de utilidad
  - Éstos no forman parte directamente de los objetos del dominio problema, pero representan la vista del usuario de los objetos semánticos
- La descripción de la arquitectura del software se hace desde una doble vertiente
  - Modelo de diseño
  - Modelo de despliegue
- Los subsistemas de un diseño arquitectónico se organizan siguiendo un patrón de capas
  - Se sigue la máxima de que los subsistemas de una capa sólo pueden referenciar subsistemas de un nivel igual o inferior
- La experiencia en OO se acumula en un repertorio tanto de principios generales como de soluciones basadas en aplicar ciertos estilos en la creación de software
  - Estos principios y estilos cuando se codifican con un formato estructurado, que describe el problema y la solución, y se le asigna un nombre, se denominan **patrones**

6. Cuestiones y ejercicios

Cuestiones y ejercicios

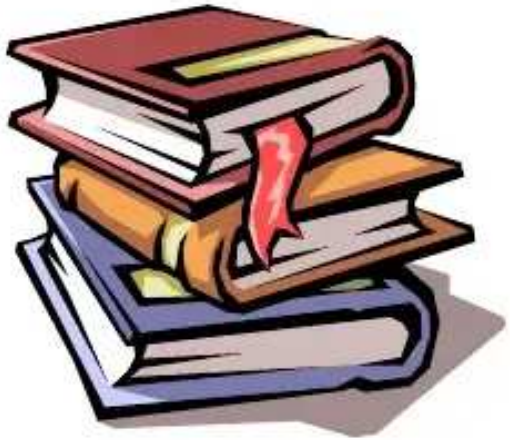
- Leer y estudiar los patrones que se recogen en [Gamma et al., 1995]
- ¿En qué se diferencian el diseño orientado a objetos y el diseño estructurado?
- Refinar los modelos de análisis construidos en los ejercicios del tema 4

7. Lecturas complementarias

Lecturas complementarias

- **Ambler, S. W.** "*The Design of a Robust Persistence Layer for Relational Databases*". White paper. Ronin International. http://www.ambysoft.com/persistenceLayer.pdf. November 2000
  - Se presenta el diseño de una capa de persistencia para aplicaciones orientadas a objetos
- **Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P., Stal, M.** "*Pattern Oriented Software Architecture: A System of Patterns*". John Wiley & Sons, 1996
  - Libro imprescindible para profundizar en los patrones de diseño
- **Martin, R. C.** "*Design Principles and Design Patterns*". http://www.objectmentor.com/resources/articles/Principles_and_Patterns.PDF, 2000
  - Buena guía para introducirse en los principios del diseño orientado a objetos
- **Ministerio de Administraciones Públicas.** "*MÉTRICA 3.0*", Volúmenes 1-3. Ministerio de Administraciones Públicas, 2001
  - Es interesante destacar la parte de diseño orientado a objetos de esta metodología
- **Page-Jones, M.** "*Fundamentals of Object-Oriented Desing in UML*". Addison-Wesley, 2000
  - Un libro muy interesante en lo referente al diseño orientado a objetos
- **Shaw, M., Garlan, D.** "*Software Architecture: Perspectives on an Emerging Discipline*". Prentice-Hall, 1996
  - Libro dedicado a las arquitecturas software

8. Referencias



Referencias (i)

[Aklecha, 1999] Aklecha, V. "*Object-Oriented Frameworks Using C++ and CORBA Gold Book*". Conolis Technology Press, 1999

[Alexander, 1979] Alexander, C. "*The Timeless Way of Building*". The Oxford University Press, 1979

[Appleton, 2000] Appleton, B. "*Patterns and Software: Essential Concepts and Terminology*". http://www.cmcrossroads.com/bradapp/docs/patterns-intro.html [Última vez visitado, 14-12-2007]. February 2000

[Booch, 1994] Booch, G. "*Object Oriented Analysis and Design with Applications*". 2nd Edition. The Benjamin/Cummings Publishing Company, 1994

[Buschmann et al., 1996] Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P., Stal, M. "*Pattern Oriented Software Architecture: A System of Patterns*". John Wiley & Sons, 1996

[Coplien, 2007] Coplien, J. O. "*A Pattern Definition - Software Patterns*". http://hillside.net/patterns/definition.html [Última vez visitado, 14-12-2007]. 2007

[Fowler, 1996] Fowler, M. "*Analysis Patterns: Reusable Object Models*". Object Technology Series. Addison-Wesley, 1996

[Gamma et al., 1995] Gamma, E., Helm, R., Johnson, R., Vlissides, J. "*Design Patterns. Elements of Reusable Object-Oriented Software*". Addison-Wesley, 1995

Referencias (ii)

[Jacobson et al., 1997] Jacobson, I., Griss, M., Jonsson, P. "*Software Reuse. Architecture, Process and Organization for Business Success*". Addison Wesley, 1997

[Jacobson et al., 1999] Jacobson, I., Booch, G., Rumbaugh, J. "*The Unified Software Development Process*". Object Technology Series. Addison-Wesley, 1999

[Larman, 2002] Larman, C. "*Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and the Unified Process*". 2nd Ed. Prentice Hall, 2002

[Liskov, 1987] Liskov, B. "*Data Abstraction and Hierarchy*". In *Addedum to Proceedings of OOPSLA'87*. Pages 17-35. ACM Press, 1987

[Martin, 1996] Martin, R. C. "*The Dependency Inversion Principle*". The C++ Report, Vol. 8, May 1996

[Meyer, 1997] Meyer, B. "*Object Oriented Software Construction*". 2nd Edition. Prentice Hall, 1997

[Monarchi y Puhr, 1992] Monarchi, D. E., Puhr, G. I. "*A Research Typology for Object-Oriented Analysis and Design*". Communications of the ACM, 35(9):35-47. September 1992

[OMG, 2003] OMG. "*OMG Unified Modeling Language Specification. Version 1.5*". Object Management Group Inc. Document formal/03-03-01. March 2003. http://www.omg.org/docs/formal/03-03-01.pdf [Última vez visitado, 16-12-2007]

[Reenskaug et al., 1996] Reenskaug, T., Wold, P., Lehne, O. A. "*Working with Objects. The OOram Software Engineering Method*". Manning Publications Co./Prentice Hall, 1996

Referencias (iii)

[Shaw y Garlan, 1996] Shaw, M., Garlan, D. "*Software Architecture: Perspectives on an Emerging Discipline*". Prentice-Hall, 1996

[Sommerville, 2005] Sommerville, I. "*Ingeniería del Software*". 7ª Edición, Addison-Wesley. 2005

[Sun, 2002] Sun Microsystems. "*Core J2EE Patterns - Data Access Object*". Sun Developer Network, 2002. http://java.sun.com/blueprints/corej2eepatterns/Patterns/DataAccessObject.html. [Última vez visitado, 14-12-2007]

Ingeniería del Software

Tema 6: Diseño orientado a objetos

Dr. Francisco José García Peñalvo
(fgarcia@usal.es)
Miguel Ángel Conde González
(mconde@usal.es)
Sergio Bravo Martín
(ser@usal.es)

3º I.T.I.S.
Fecha de última modificación: 16-10-2008

Universidad de Salamanca – Departamento de Informática y Automática