Tema 5. Patrones GRASP

Diseño de Sistemas Multimedia

Sumario

- Contexto. Necesidad de los patrones
- Definición de patrón software. Tipos
- Patrones GRASP
  - Experto
  - Creator
  - Bajo acoplamiento
  - Alta cohesión
  - Controlador
  - Polimorfismo
  - Fabricación Pura
  - Indirección

Necesidad de los Patrones

- "*El comienzo de la sabiduría de un programador está en reconocer la diferencia entre obtener un programa que funcione y uno que funcione correctamente*" (M.A. Jackson)

- Los **patrones** software nos dan la base para que un programa funcione correctamente (es decir, que tenga **Calidad** -> sea mantenible, escalable, usable, eficiente, etc.)

Necesidad del diseño dirigido por Patrones

- *"Los patrones te ayudan a aprender de los éxitos de otros en lugar de aprender de tus errores"* (Mark Johnson)

¿Qué son los patrones software?

- Uno de los mecanismos existentes para capturar conocimiento sobre **problemas** y **soluciones exitosas** en el desarrollo del software

- Permiten **reutilizar** la experiencia de otros diseñadores, y reducir el esfuerzo asociado a la producción de sistemas más **efectivos** y **flexibles** (más adaptables al cambio)

Definición de Patrón

- *"Es un documento que presenta una **solución probada** para un **problema** recurrente (no trivial) en un **contexto** concreto"*
- Especifica una **configuración espacial** de elementos y un **comportamiento asociado** a esta configuración
- Provee un **vocabulario común** y un concepto que permite la mejora del entendimiento entre expertos
- **Mejora la calidad** de la solución del problema

Representación del Patrón

- **Contexto**:
  - Sitúa el entorno bajo el cual el patrón existe
- **Problema**:
  - Describe que aspectos abordados por el desarrollador se realizan de forma insatisfactoria
- **Fuerzas**:
  - Lista de razones y motivaciones que afectan al problema y justifican la aplicación del patrón

Representación del Patrón (cont.)

- **Solución**:
  - Describe la solución adoptada brevemente y los elementos de la solución en 2 secciones:
  - **Estructura**:
    - Usa un diagrama para mostrar la estructura básica de la solución
  - **Estrategia**:
    - Describe una de las posibles implementaciones del patrón en la tecnología apropiada (p.e. .NET o Java)

Tipos de patrones de diseño

- **Patrones de diseño de objetos GRASP** [Larman]
- **Patrones de diseño detallado [Gamma et al.]**
- **Patrones de arquitectura** [Buchmann, **Fowler**, etc.]
- **Patrones de Mejores Practicas [.Net y J2EE]**

Patrones GRASP

- **G**eneral **R**esponsability **A**ssigment **S**oftware **P**atterns (GRASP)
- Patrones de Principios Generales para Asignar Responsabilidades
- "Describen los principios fundamentales de **diseño de objetos** y la asignación de responsabilidades expresados en forma de patrones" Craig Larman
- Constituye la base de **cómo** se diseñará el sistema
- Se aplica en los primeros momentos del diseño

Introducción: Usos del diagrama de clases

- UML incluye los **diagramas de clases** para ilustrar clases, interfaces y sus asociaciones.
- Éstos se utilizan para el **modelado estático de objetos**.
- Los diagramas de clases se pueden usar tanto desde una perspectiva conceptual (para el modelo del dominio) como desde una perspectiva software (para el modelado de la CAPA DE DOMINIO).
  - Es común hablar de **"Diagrama de clases de Análisis**" (DCA) para referirse al diagrama de clases que representa el modelo de dominio, y a **"Diagrama de Clases de Diseño" (DCD)** para hablar del diagrama de clases que representa a las clases software, es decir, a las clases que finalmente van a ser implementadas.

Diseño de Objetos

- **Tras haber identificado los requisitos, y haber creado un modelo de dominio (DCA), se añaden operaciones a las clases, y define las secuencias de mensajes entre objetos para cubrir los requisitos (en el DCD)**

- Estas recomendaciones son especialmente útiles, ya que hay **principios fundamentales de diseño, nada triviales,** que deben ser tenidos en cuenta a la hora de:
  - Decidir qué operaciones hay que asignar a qué clases
  - Cómo los objetos deberían interactuar para dar respuesta a los requisitos

Responsabilidad

- Una responsabilidad es “un contrato u obligación de una clase”
- Las responsabilidades están relacionadas con las obligaciones de una clase en cuanto a su comportamiento
- Básicamente las responsabilidades son de 2 tipos:
  - Conocer:
    - Conocer los datos privados encapsulados
    - Conocer los objetos relacionados
    - Conocer las cosas que puede derivar o calcular
      - Ejemplo: “Un barco es responsable de conocer cuándo está hundido”
  - Las responsabilidades relacionadas con ‘conocer’ son normalmente inferibles del Modelo de Dominio (debido a los atributos y asociaciones que éste ilustra)

Reponsabilidad

- Hacer:
  - Hacer algo él mismo, como crear un objeto o hacer un cálculo
  - Iniciar una acción en otros objetos
  - Controlar y coordinar las actividades en otros objetos
    - Ejemplo: Una factura es responsable de crear sus líneas de detalle”

Introducción: Responsabilidades vs. métodos

- La **complejidad del proceso de traducción de responsabilidades** a clases y operaciones (implementadas mediante métodos) está influenciada por la **granularidad de la responsabilidad**.
  - Ejemplo: la responsabilidad "proporciona acceso a BD relacional" puede involucrar docenas de clases y cientos de métodos, mientras que "crear Venta" podría involucrar sólo uno o un pequeño número de métodos.
- Una responsabilidad no es un método, pero los métodos se implementan para cubrir responsabilidades
- Los métodos pueden colaborar con otros métodos y objetos para cubrir una determinada responsabilidad

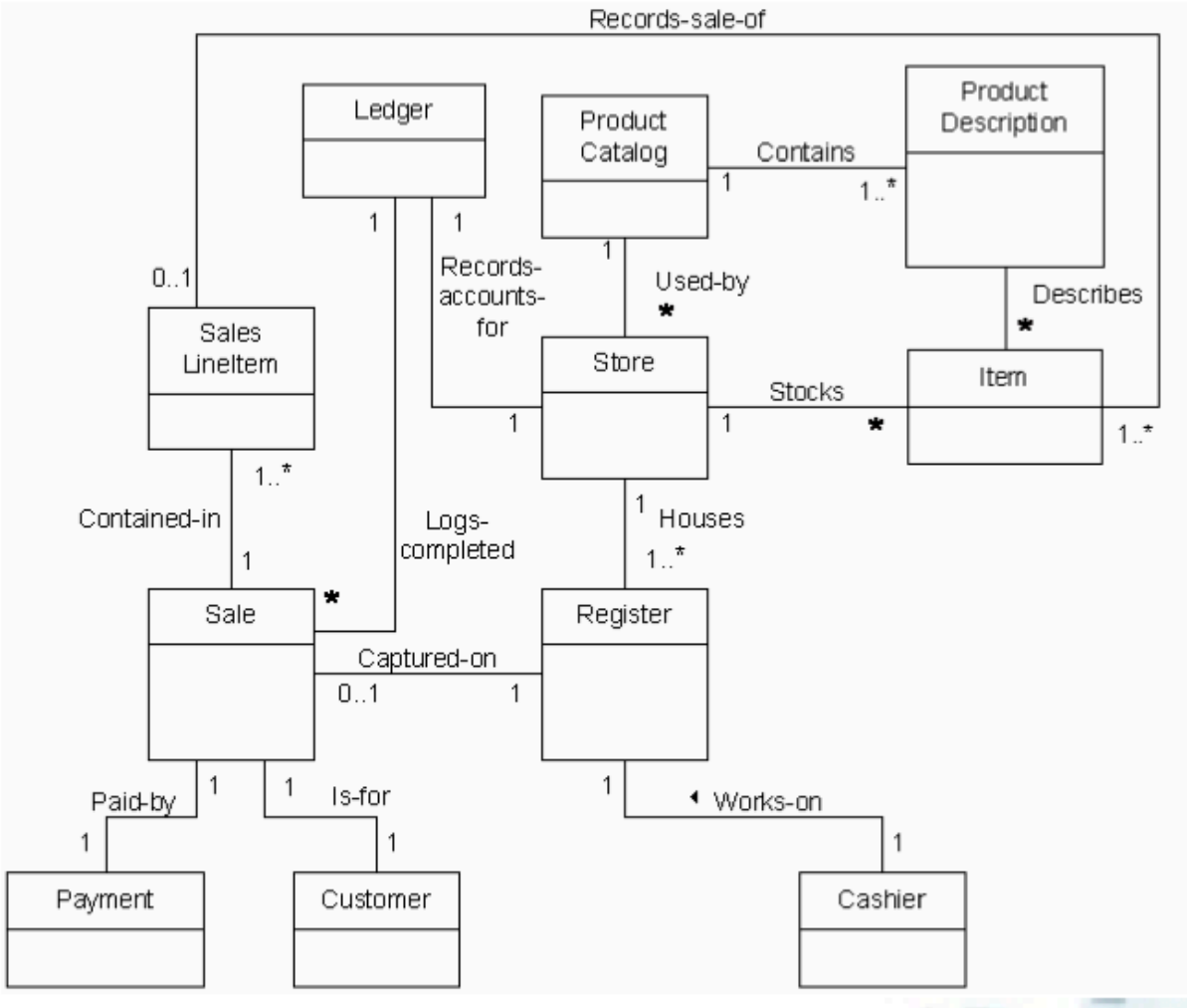
Patrones GRASP

Los patrones GRASP (Larman) describen **principios fundamentales del diseño de objetos** y de la **asignación de responsabilidades**, expresado en términos de patrones

GRASP = comprender. Larman sugiere que comprender estos principios es fundamental para diseñar con éxito

- Básicos:
  - Experto en información
  - Creador
  - Bajo Acoplamiento
  - Alta Cohesión
  - Controlador
- Avanzados:
  - Polimorfismo
  - Fabricación Pura
  - Indirección
  - Protección de Variaciones

Ejemplo de DC de Análisis

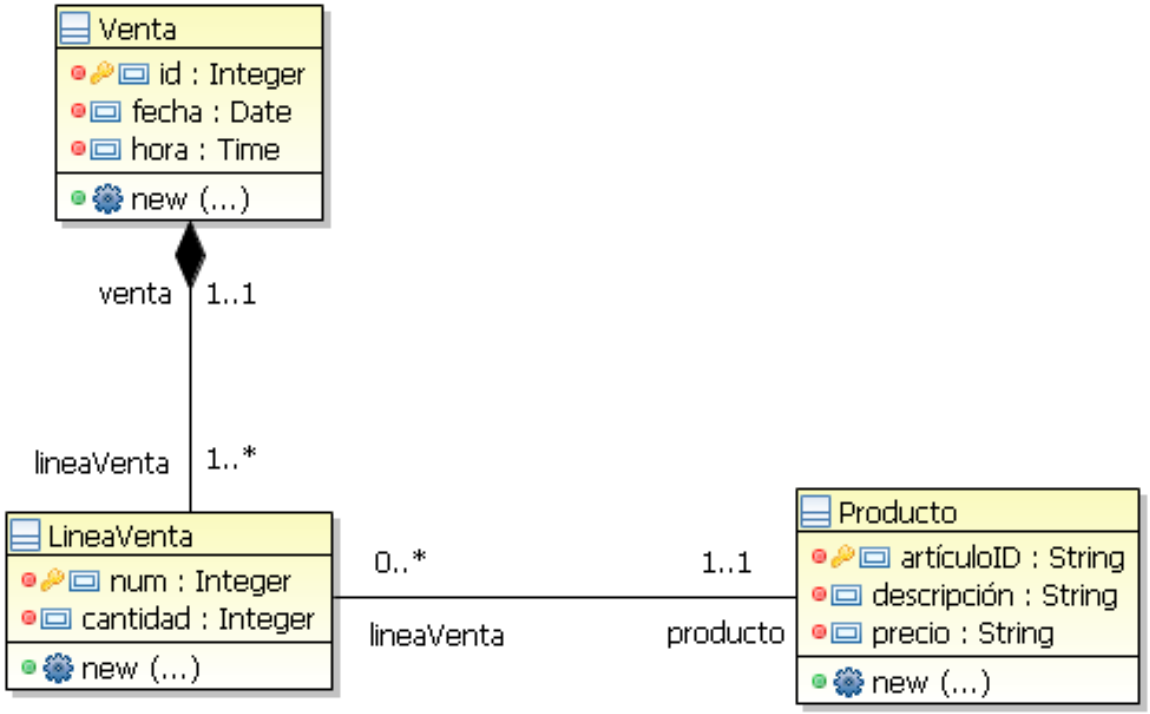


Experto en Información

- Problema
  - ¿Cuál es el principio general para asignar responsabilidades a las clases?
- Solución :
  - Asignar una responsabilidad al experto en información – la clase que tiene la *información* necesaria para realizar la responsabilidad.
  - **Un *Experto* es una clase que tiene toda la información necesaria para implementar una responsabilidad.**
  - Ventajas:
    - Encapsulamiento de la información
    - Distribución del comportamiento del manejo de la información

Experto en Información

- Ejemplo de Experto: ¿dónde situar las operaciones dameTotal(), dameSubtotal() y damePrecio()?

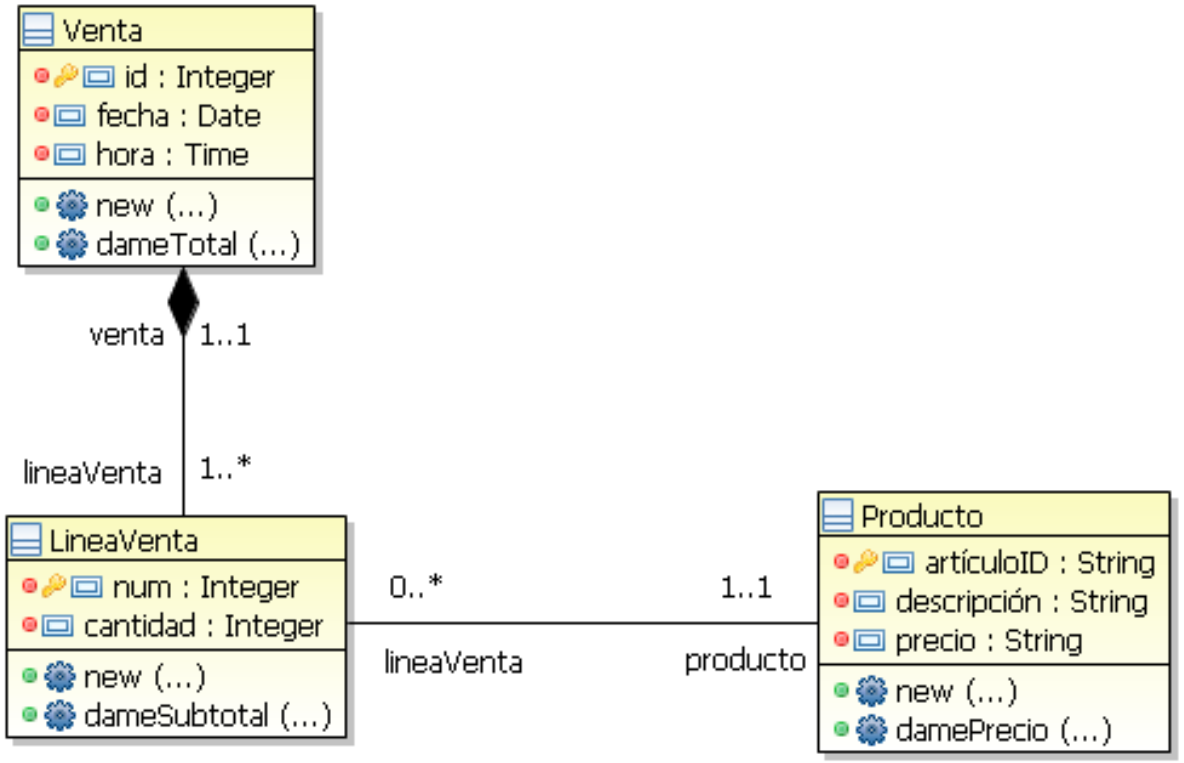


Experto en Información

- ¿Qué información se necesita para calcular el importe total?
  - Todas las instancias de LineaVenta, y la suma de sus subtotales
  - Ya que Venta contiene las instancias de LineaVenta, podría ser el experto adecuado
- ¿Qué información se necesita para calcular los subtotales?
  - LineaVenta.cantidad y Producto.precio
  - Ya que es la LineaVenta quien conoce la cantidad y el Producto, podría ser el experto adecuado
- ¿Quién se encarga de recuperar el precio del Producto?
  - Producto, ya que es quien contiene el precio

Experto en información

- La asignación quedaría como sigue:



Creador

- Problema:
  - ¿Quién debería ser el responsable de la creación de una nueva instancia de alguna clase?
- Solución:
  - B es un Creador de A si se asigna a la clase B la responsabilidad de crear una instancia de la clase A y si se cumple uno o más de los casos siguientes:
    - B agrega objetos de A;
    - B contiene objetos de A;
    - B registra instancias de objetos de A;
    - B utiliza más estrechamente objetos de A;
    - B tiene los datos de inicialización que se pasarán a un objeto de A cuando sea creado(por lo tanto, B es un Experto con respecto a la creación de A)

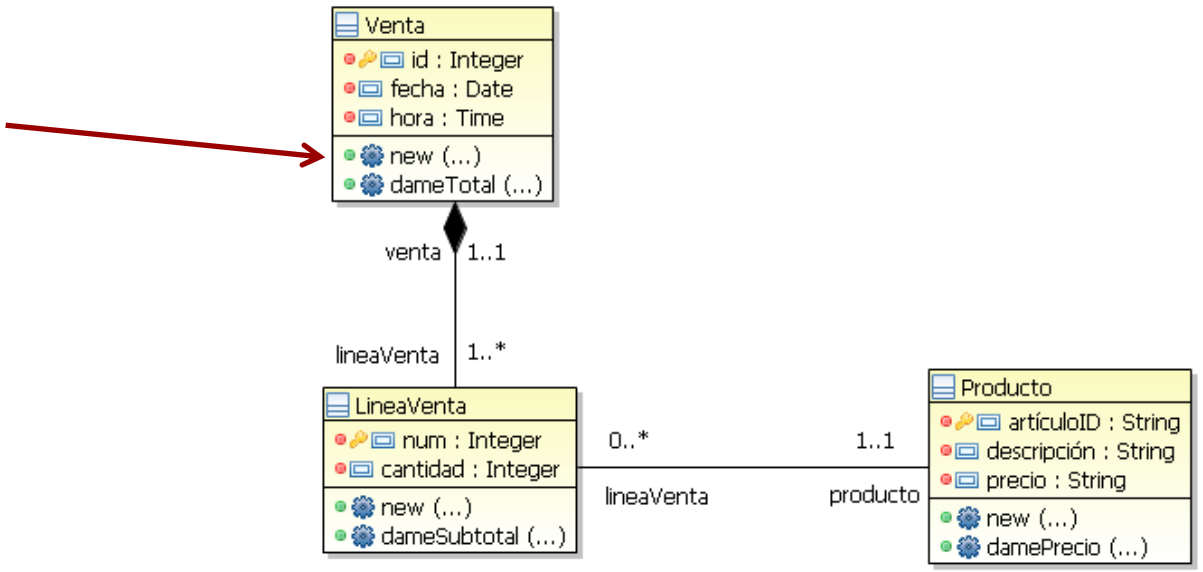
Creador

- El patrón *Creador* guía la asignación de responsabilidades relacionadas con la creación de objetos (una tarea muy común). La intención básica del patrón es encontrar **un creador que necesite conectarse al objeto creado** en alguna situación.
- Ventajas:
  - Bajo acoplamiento logrando mayor mantenibilidad y reutilización.
- Desventajas:
  - Puede ser muy compleja la operación de creación de instancia.

Ejemplo Creador

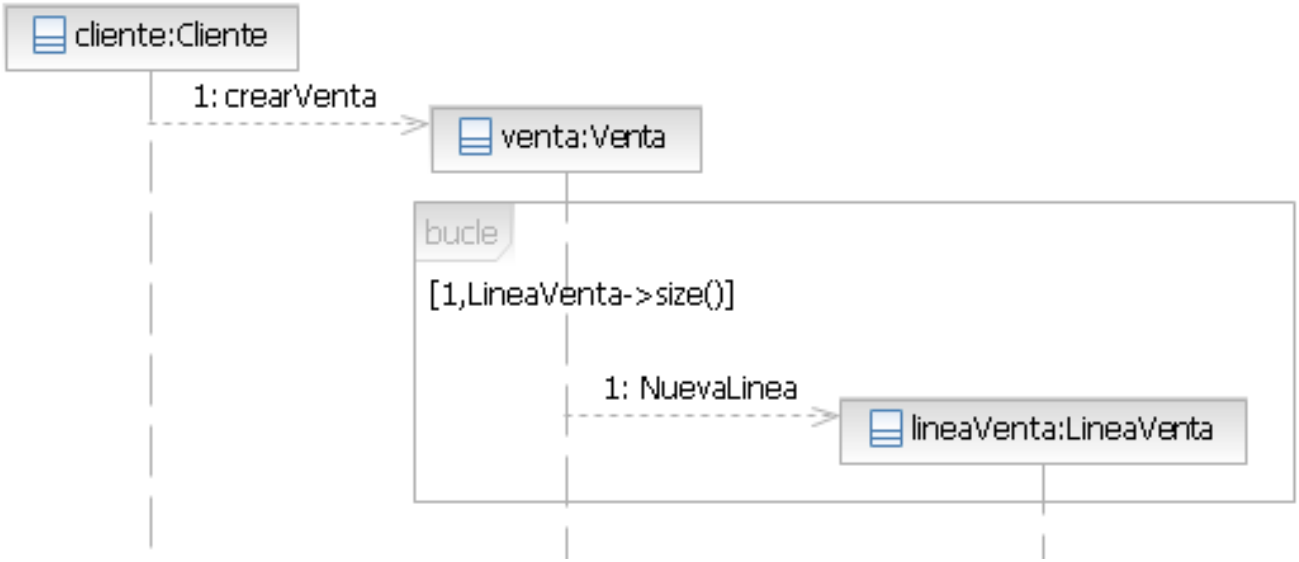
- En la aplicación anterior queremos asignar la responsabilidad de la operación crearLineaVenta
- ¿A qué clase le asignamos la responsabilidad?

El patrón creador nos sugiere que se lo asignemos a Venta (Venta agrega objetos de la clase LineaVenta)



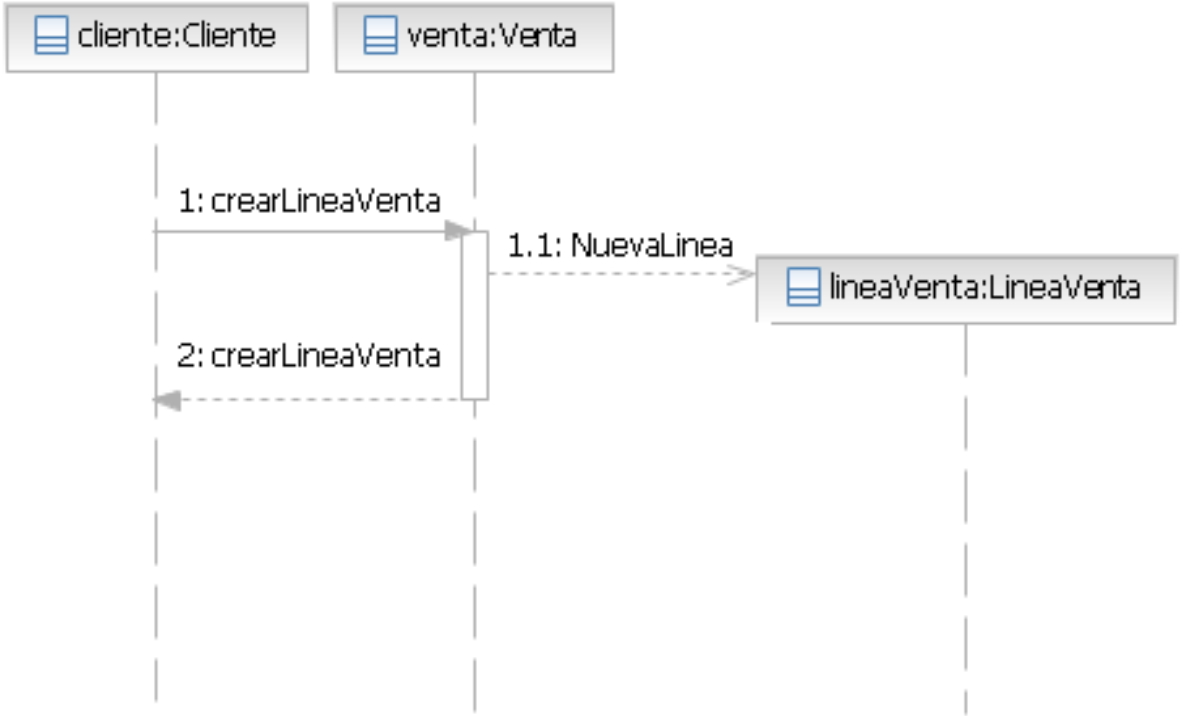
Ejemplo Creador

- Existen 2 alternativas:
  - 1er Caso: Que por la cardinalidad el objeto Venta (1..*) cree los objetos LineaVenta en la creación del propio objeto Venta (es decir, desde el new de Venta). Para ello, se recibirá los objetos de LineaVenta como parámetro de entrada.



Ejemplo Creador

- 2do Caso: Que el objeto compuesto (Venta) cree los objetos mediante una operación que invoque al objeto componente (LineaVenta). En el ejemplo, crearLineaVenta en la clase Venta invoca a NuevaLinea de LineaVenta (este último privado)



Bajo Acoplamiento

- Problema
  - ¿Cómo podemos reducir el impacto de los cambios e incrementar la reutilización?
- Solución:
  - Asignar las responsabilidades a las clases de manera que el acoplamiento permanezca bajo
  - El acoplamiento es una medida de la fuerza con la que un objeto está conectado a, tiene conocimiento de, o confía en otros objetos
  - Un objeto con bajo (o débil) acoplamiento apenas depende de otros objetos

Bajo Acoplamiento

- El patrón Bajo Acoplamiento es un principio a tener en mente en todas las decisiones de diseño. Es un principio evaluativo que debe aplicar un diseñador mientras evalúa todas las decisiones de diseño
- El bajo acoplamiento genera clases más independientes
- Ventajas:
  - Se reducen los cambios en otras clases
  - Más fácil de entender de manera aislada
  - Es conveniente para reutilizar componentes (o clases)

Bajo Acoplamiento

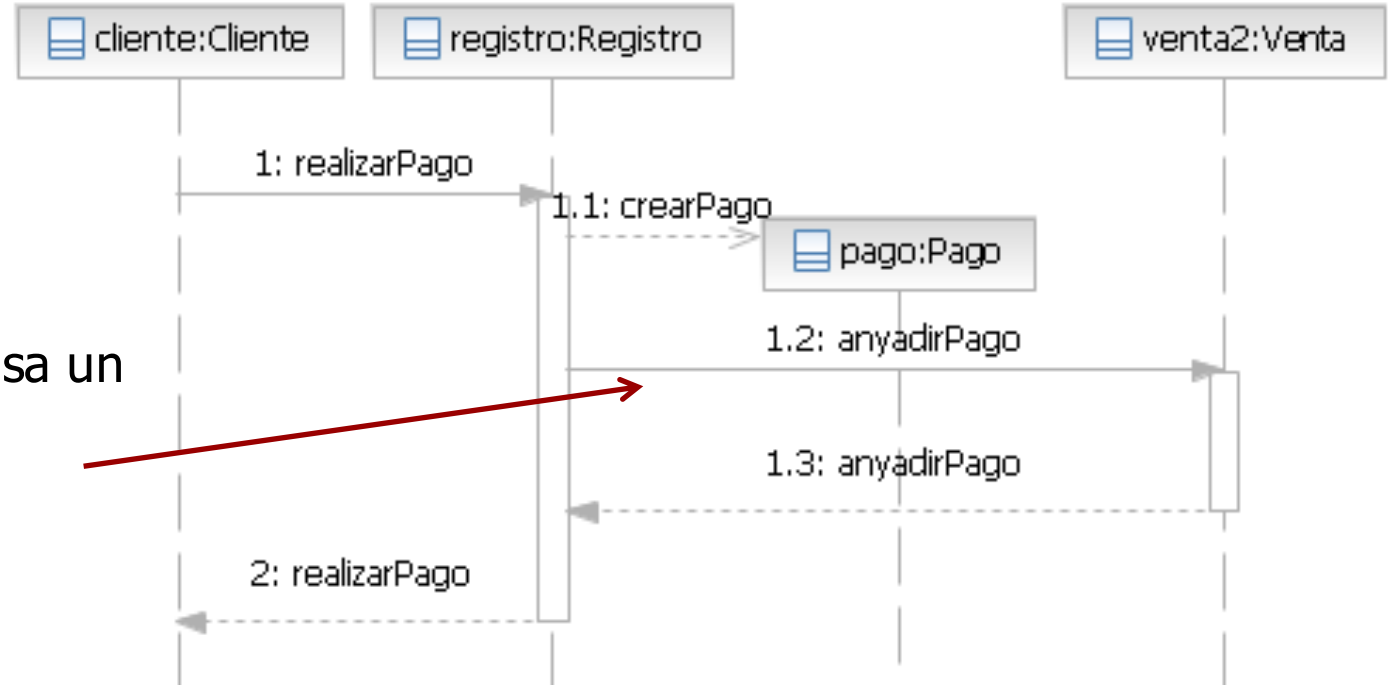
- Ejemplos de posibles acoplamientos entre una clase A y una clase B (de mayor a menor acoplamiento)
  - A es subclase directa o indirectamente de B
  - A tiene un atributo del tipo B (es decir, tiene una relación navegable de asociación, agregación o composición)
  - Un objeto de la clase A invoca a una operación de B
  - Una operación de A tiene un parámetro del tipo B
  - Una operación de A devuelve un valor de tipo B

Ejemplo Bajo Acoplamiento

- ¿Qué diseño, basado en la asignación de responsabilidades, soporta más Bajo Acomplamiento?
  - Se debe crear un objeto Pago y asignarse a una Venta.

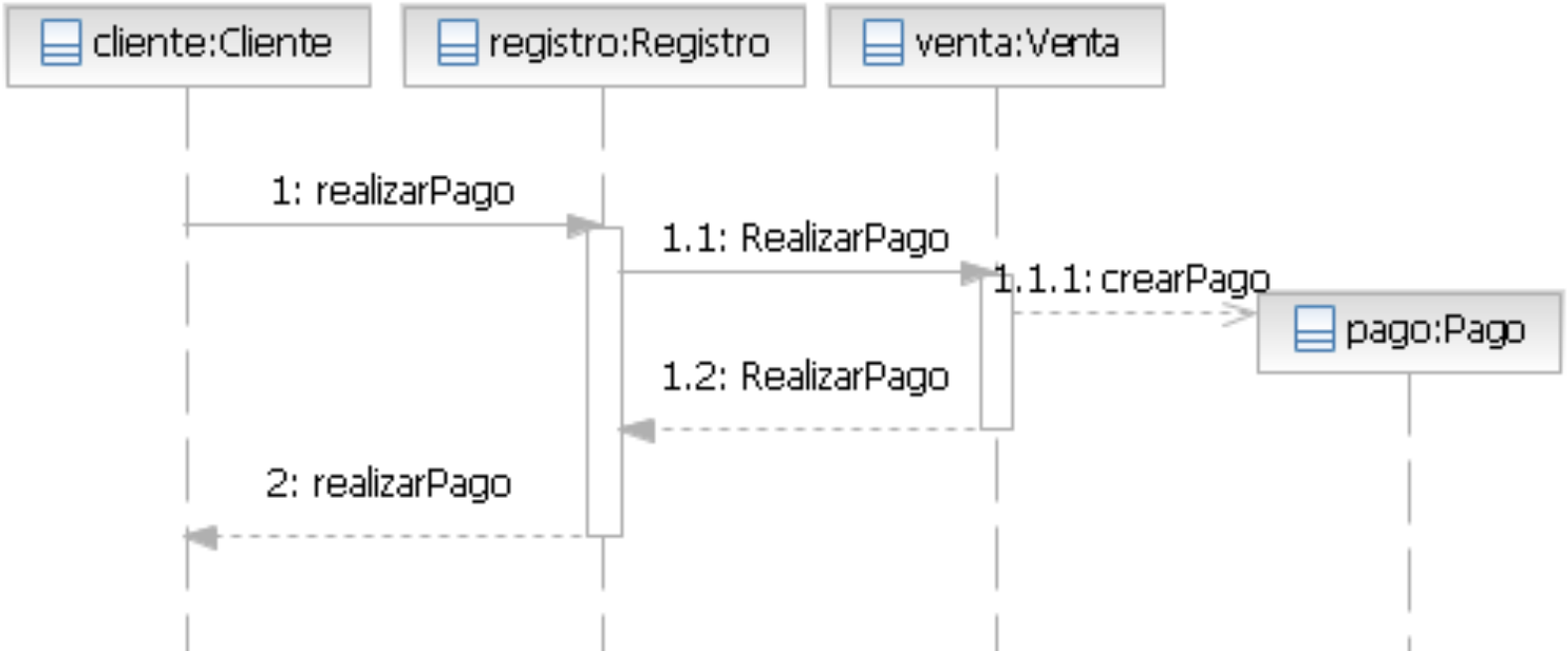
- **Caso A**

Recalcar: En **AnyadirPago** se pasa un objeto Pago como parámetro



Ejemplo de Bajo Acoplamiento

- **Caso B**: Es Venta quien crea y se vincula con el Pago



Ejemplo de Bajo Acoplamiento

- Tanto el caso A como el caso B, Venta y Pago están acoplados
- Pero en A también lo están Registro y Pago
- Mientras que en B no lo están
- Así, que desde el punto de vista del Bajo Acoplamiento es preferible la opción B

Alta Cohesión

- Problema:
  - ¿Cómo mantener una complejidad manejable?
- Solución:
  - Asignar las responsabilidades de manera que la cohesión permanezca alta
  - La cohesión es una medida de la fuerza con la que se mide el grado de focalización de las responsabilidades de un elemento

Alta Cohesión

- Una clase con baja cohesión hace muchas cosas que no están relacionadas, o hace demasiado trabajo:
  - Clases difíciles de entender
  - Difíciles de reutilizar
  - Difíciles de mantener
  - Delicadas, constantemente afectadas por los cambios de las clases de las que depende

Alta Cohesión

- Sin embargo, una clase con alta cohesión tiene:
  - Un número relativamente pequeño de operaciones, con funcionalidad altamente relacionada
  - No realiza mucho trabajo
  - Colabora con otros objetos para compartir el esfuerzo si la tarea es extensa

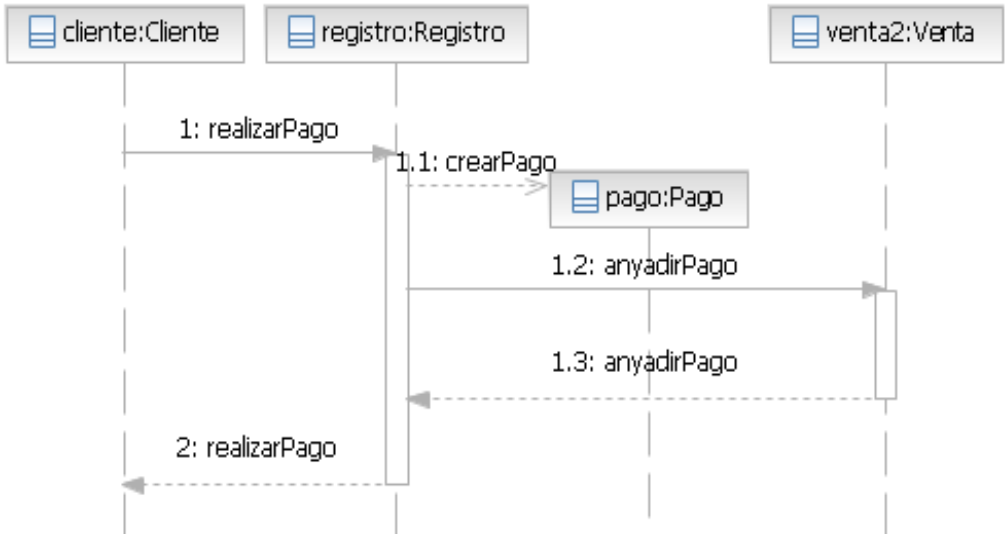
Alta Cohesión

- Ventajas:
  - Incrementa la claridad y facilita la comprensión del diseño
  - Simplifica el mantenimiento y las mejoras
  - Soporta a menudo el bajo acoplamiento
  - Incrementa la reutilización

Ejemplo de Alta Cohesión

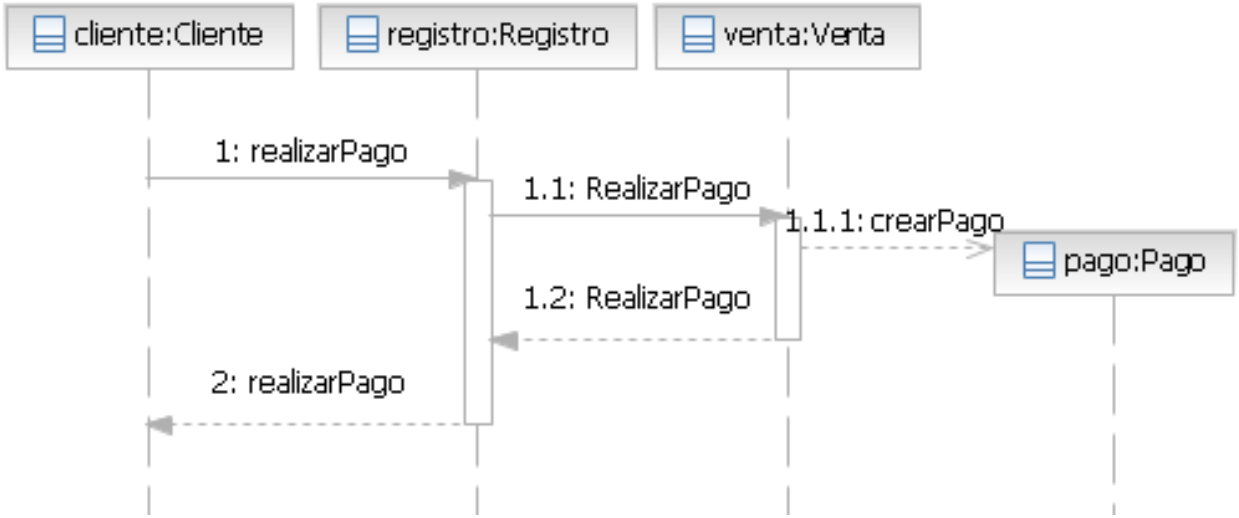
- Basándonos en el ejemplo anterior, evaluamos esta vez la cohesión de ambas soluciones:

- Caso A:
  - En este caso Registro mediante la operación realizarPago se encarga de crear el Pago y establecer la relación con Venta



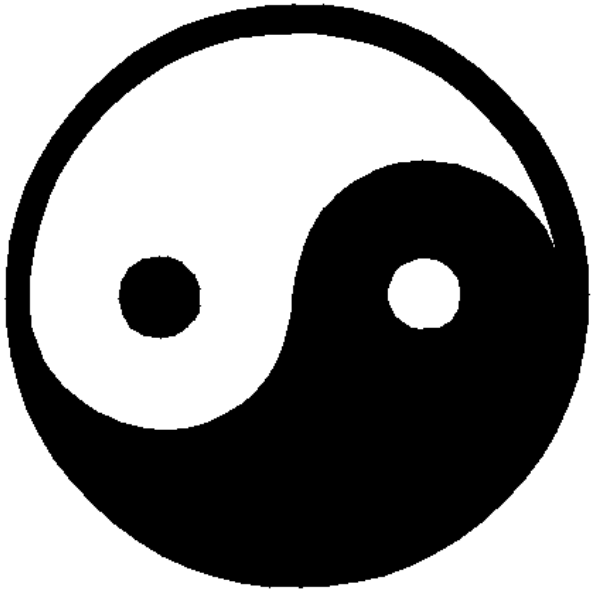
Ejemplo Alta Cohesión

- Caso B:
  - La responsabilidad de crearPago es asignada a la Venta
  - Quitamos de responsabilidades a Registro



Es la solución con mayor cohesión

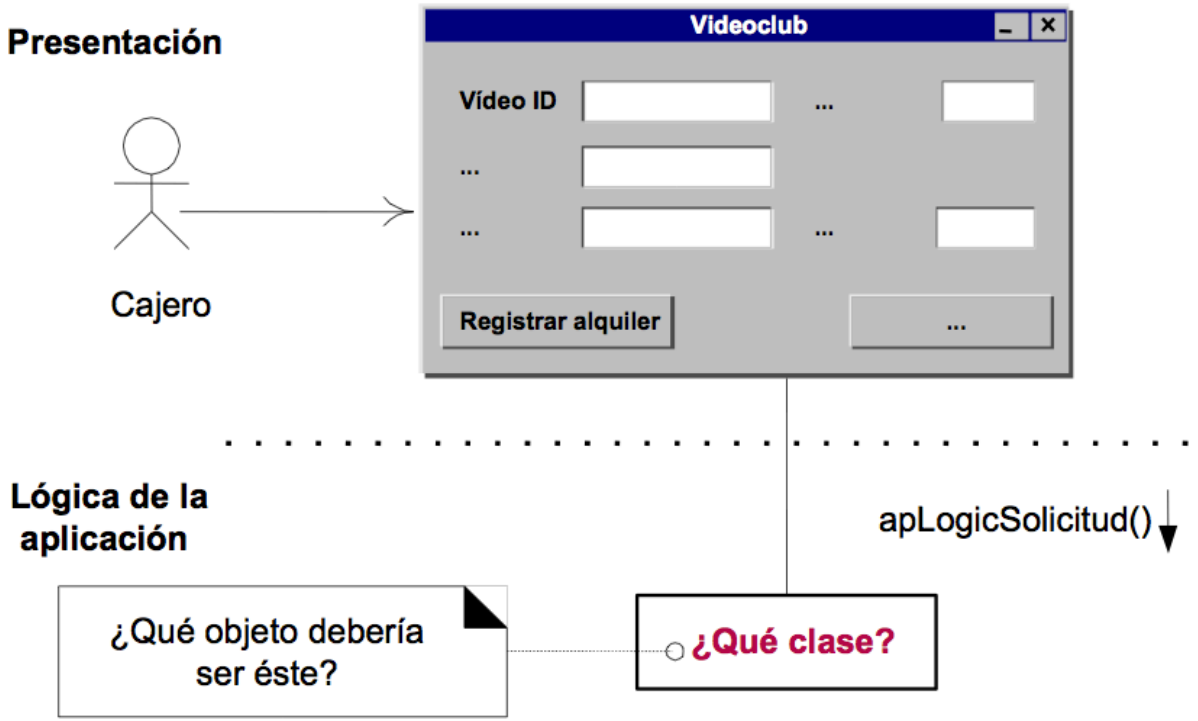
GRASP: Cohesión vs. acoplamiento



- **Cohesión y acoplamiento son el "yin and yang" de la ingeniería del software**, debido a su interdependencia e influencia.

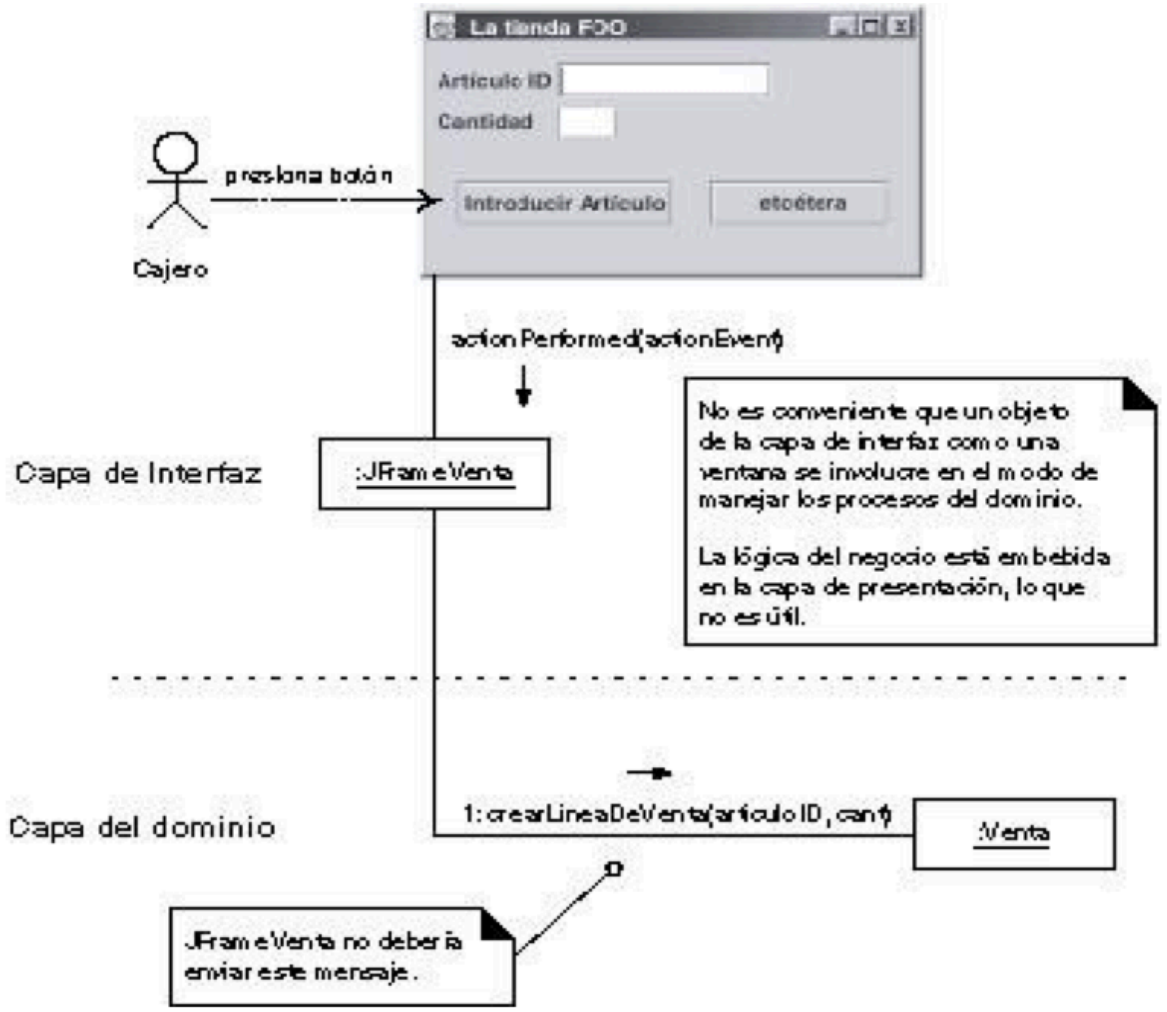
Controlador

- Problema:
  - ¿Qué clase de la lógica de negocio debe ser el responsable de gestionar un evento que procede de la capa de cliente?



Solución 1

- El responsable sería un componente de negocio que representa a las clases de dominio en la interfaz
- Sería un componente de IU donde residirían los procesos complejos



Solución 1

- No son buenas las clases o los componentes de negocio que representan al dominio en el interfaz
- Se crearían dependencias entre la interfaz y los componentes de lógica de negocio (aumenta el acoplamiento entre capas)
- No es conveniente que un objeto de interfaz maneje tanto los procesos de la lógica de negocio como de la interfaz (se reduce la cohesión)

Solución 2: Controlador

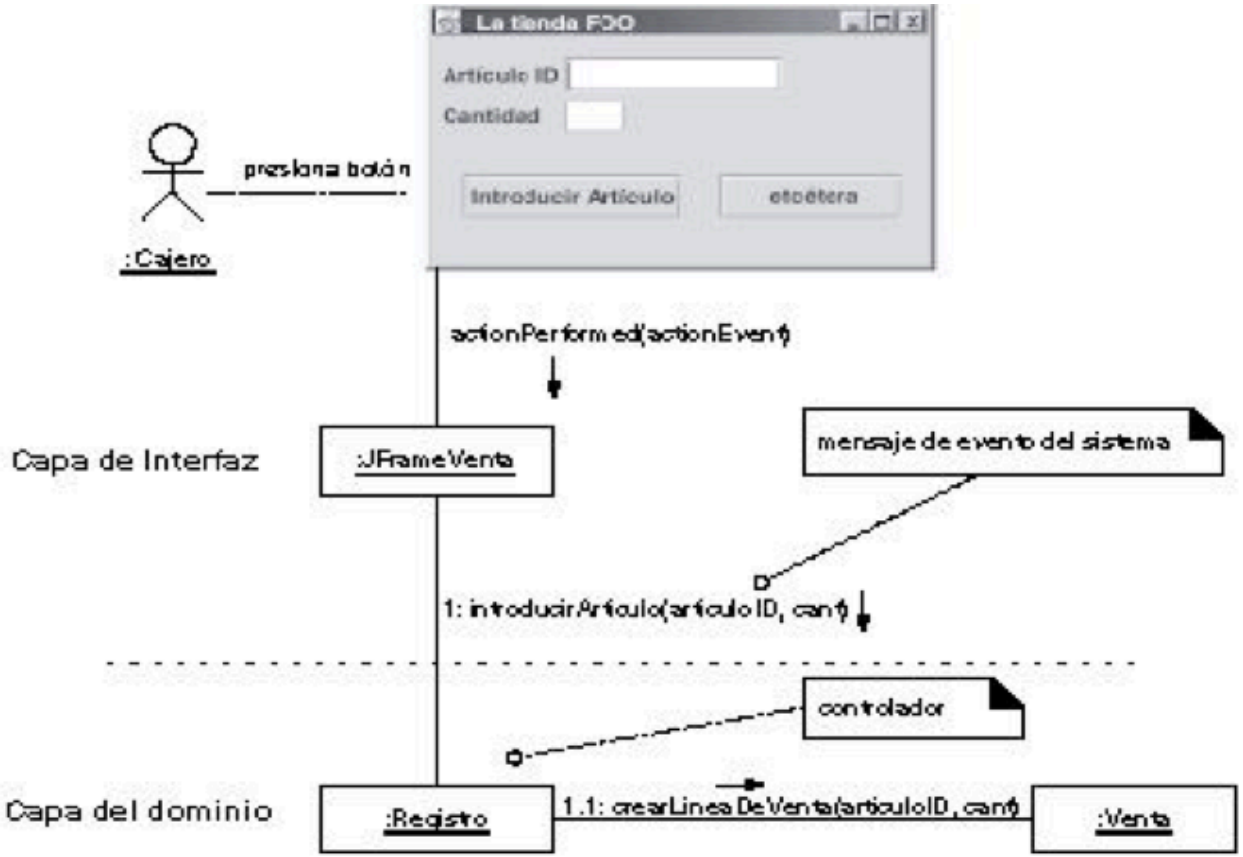
- Definimos una clase intermedia en la capa servidora llamada Controlador que únicamente se encarga de recibir la petición del interfaz y de reenviar las llamadas a los objetos de negocio
- En algunas ocasiones solamente reenviará la petición a la lógica (operaciones simples), en otras ocasiones se definirán coreografías o una secuencia de llamadas (operaciones complejas)

Solución 2: Controlador

- Tipos de Controladores:
  - Controlador de Fachada: representa una fachada global, o por dispositivo o por subsistema (controlador fachada)
  - Controlador de casos de uso:
    - Contendrá únicamente aquellos eventos que pertenecen a un caso de uso. Habiendo tantos controladores como casos de uso tenga el sistema
    - Construcción artificial para dar soporte al sistema
    - Se utilizan cuando los Controladores de Fachada conduce a diseños con baja cohesión (demasiadas operaciones diferentes)

Solución 2: Controlador

- Ejemplo donde la clase Registro es el Controlador de Fachada



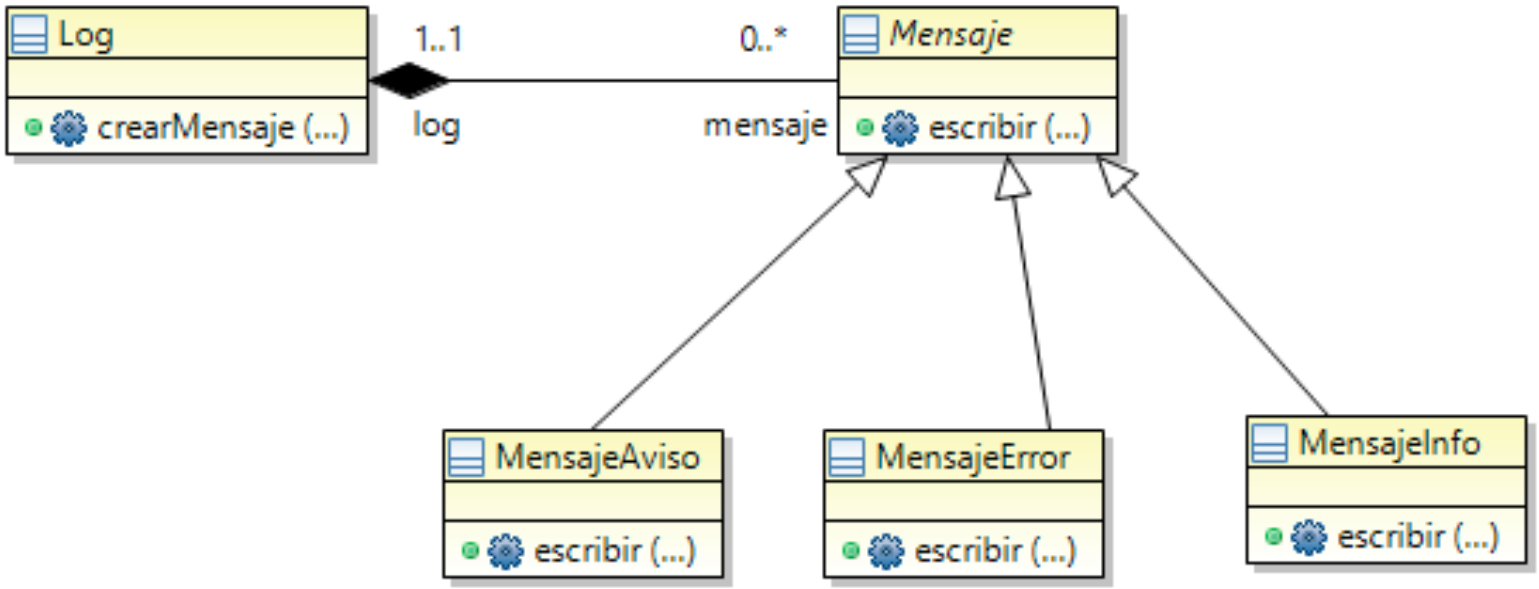
Patrones GRASP avanzados

Polimorfismo
Fabricación Pura
Indirección
Protección de variaciones

GRASP: Polimorfismo

- **Problemas:**
  - ¿Cómo manejar alternativas basadas en un tipo sin usar sentencias condicionales if-then o switch que requerirían modificación en el código?
- **Solución:**
  - Cuando alternativas o comportamientos relacionados varían por el tipo (clase), asigna la responsabilidad del comportamiento usando "operaciones polimórficas" a los tipos para los cuales el comportamiento varía
  - **Corolario:** No preguntes por el tipo del objeto usando lógica condicional para realizar las alternativas variantes basadas en el tipo.

Ejemplo de Polimorfismo



- Mensaje se puede definir como una clase abstracta o un interfaz

Polimorfismo

- Ventajas:
  - Se añaden fácilmente las extensiones necesarias para nuevas variaciones
  - Las nuevas implementaciones se pueden introducir sin afectar a los clientes
- Inconvenientes:
  - Algunas veces, los desarrolladores diseñan sistemas con interfaces y polimorfismo para futuras necesidades especulativas frente a posibles variaciones desconocidas
    - Es conveniente una evaluación crítica
- Patrones GOF relacionados: GoF (Adapter, Command, Composite, Proxy, State, Strategy)

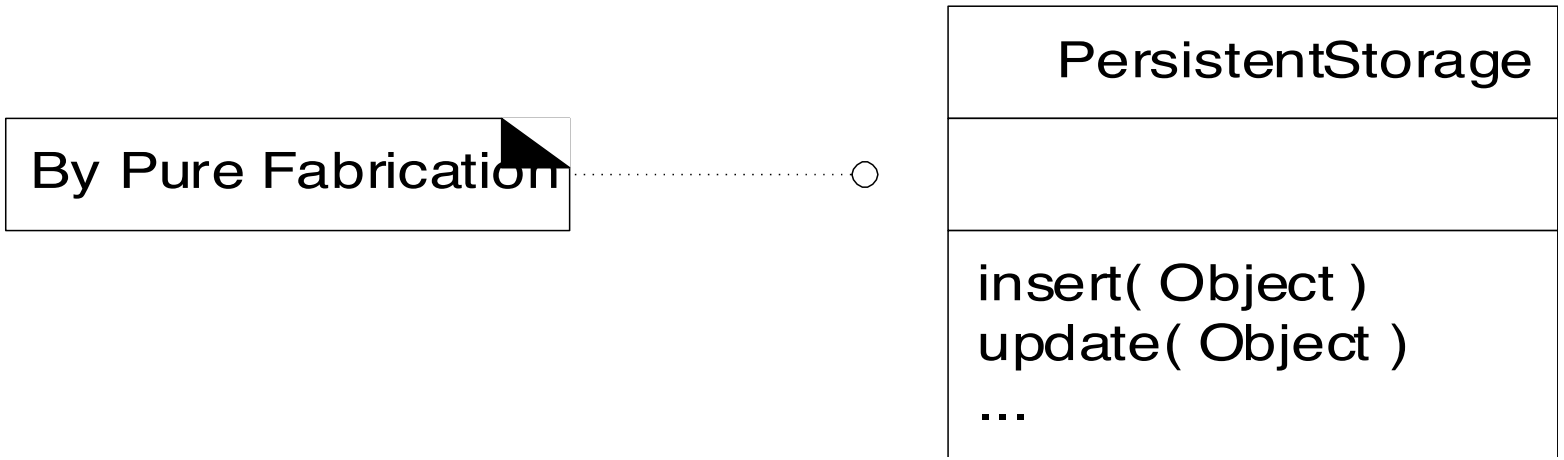
GRASP: Fabricación pura

- Es necesario guardar instancias de una clase VENTA en una base de datos relacional. ¿Quién debería tener esa responsabilidad?

- Por **Experto** la clase Venta debería tener esta responsabilidad, sin embargo:
  - La tarea requiere un número importante de operaciones de base de datos, ninguna relacionada con el concepto de Venta, resultaría **incohesiva**
  - El concepto Venta quedaría acoplado con la interface de la base de datos (ej.- ASP.NET en .NET, NHibernate, etc.) por lo que el **acoplamiento aumenta**.
  - Guardar objetos en una base de datos relacional es una tarea muy general para la cual se requiere que múltiples clases le den soporte. Colocar éstas con los objetos derivados de la lógica de negocio sugiere **pobre reuso**

GRASP: Fabricación pura

- **Solución:** Crear una clase (*PersistentStorage*) que sea responsable de guardar objetos de cualquier tipo un almacenamiento persistente (tal como una base de datos relacional).



GRASP: Fabricación pura

- Problemas resueltos:
  - La clase Venta continua bien definida, con alta cohesión y bajo acoplamiento.
  - La clase PersistentStorage es, en sí misma, relativamente cohesiva, tiene un único propósito de almacenar o insertar objetos en un medio de almacenamiento persistente
  - La clase PersistentStorage es un objeto genérico y reusable

GRASP: Fabricación pura

- **Problema:** ¿Qué objeto debería tener la responsabilidad, cuando no se desean violar los principios de "Alta Cohesión" y "Bajo Acoplamiento" o algún otro objetivo, pero las soluciones que sugiere Experto no son apropiadas o cuando no es apropiado asignarlo a una clase software inspirada a partir de una clase conceptual?

- **Solución:** Asigne un conjunto "altamente cohesivo" de responsabilidades a una **clase artificial** conveniente que no represente un concepto del dominio del problema, algo producto de la "imaginación" para soportar alta cohesión, bajo acoplamiento y reuso.
  - Éste es precisamente el principio que se aplica cuando se introducen clases 'Helper' y clases 'Utility'

GRASP: Fabricación pura

- En sentido amplio, los objetos pueden dividirse en dos grupos:
  - Aquellos diseñados por/mediante **descomposición representacional**. (Ej.- *Venta* existe como concepto en el mundo real)
  - Aquellos diseñados por/mediante **descomposición conductual**. (Ej.- Para agrupar comportamientos o algoritmos; clases sin nombre ni propósito relacionado con el mundo real, e.g. *PersistenceStorage*). Este es el caso más común para objetos **Fabricación pura.**

GRASP: Fabricación pura

- El principio de *descomposición conductual* para objetos *Fabricación Pura* en ocasiones es sobreutilizado por novatos en diseño y tienden a dividir el software en términos de funciones.

  Exagerando: **las funciones se convierten en objetos**. (Clases "**functoides**").
  ¡Tened cuidado con esto si continuamente estáis pasando objetos como parámetros para que sean procesados por métodos!

GRASP: Indirección

- **Problema:**
  - ¿Dónde asignar una responsabilidad para **evitar acoplamiento directo entre dos o más clases**?
  - ¿Cómo desacoplar objetos de tal manera que se soporte el bajo acoplamiento y el reuso potencial se mantenga alto?
- **Solución:**
  - **Asignad la responsabilidad a un objeto intermedio** que medie entre otros componentes o servicios, de tal manera que los objetos no estén directamente acoplados
  - El objeto intermedio crea una **indirección** entre los componentes

GRASP: Indirección

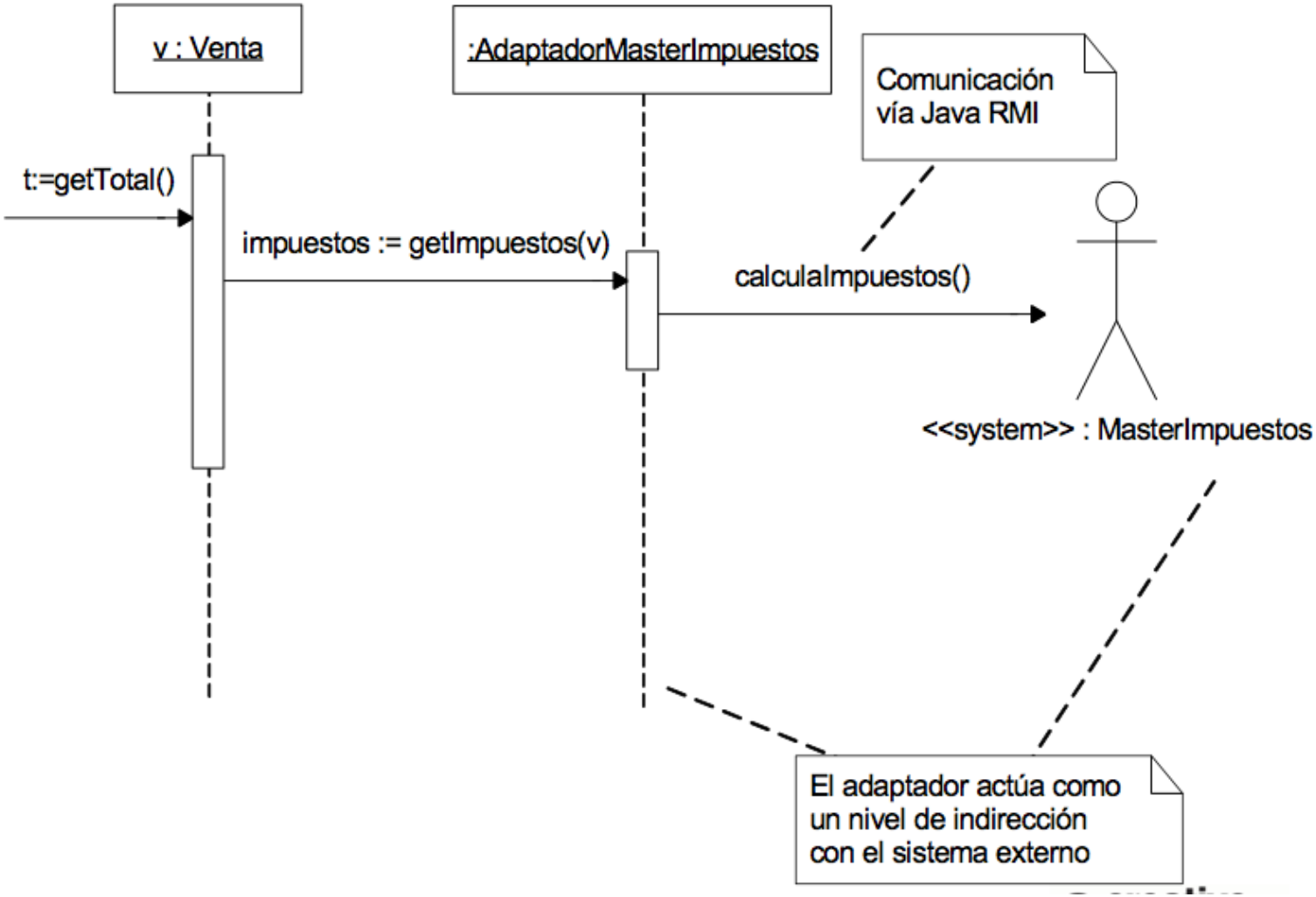
- ***"Muchos problemas en ciencias de la computación pueden resolverse mediante otro nivel de indirección"***
  es un viejo adagio con relevancia particular en diseño orientado a objetos (David Wheeler).

- Así como muchos patrones de diseño son especializaciones de **Fabricación Pura**, muchos otros también lo son de **Indirección** (Adapter, Facade, Observer, entre otros).

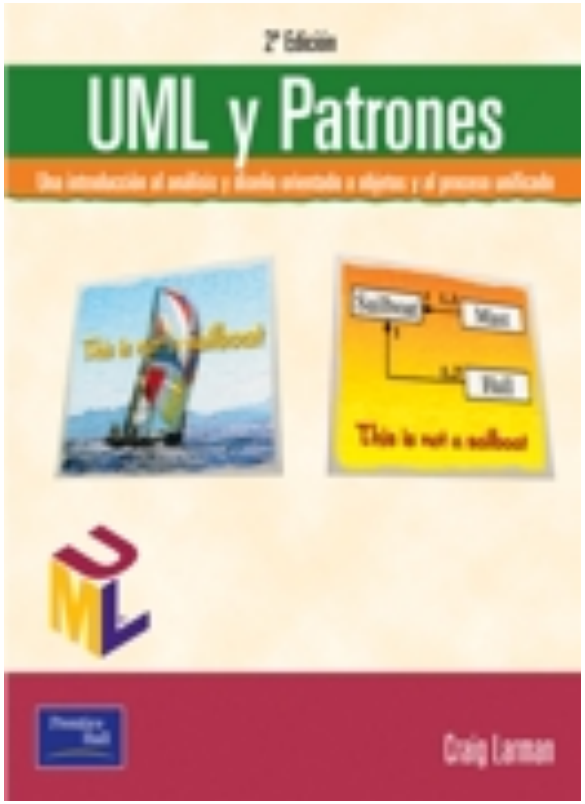
- Además muchas Fabricaciones Puras son generadas por causa de Indirección.

- **La motivación principal** es el **bajo acoplamiento**; por lo que un intermediario se agrega para desacoplar otros componentes o servicios

GRASP: Indirección



Bibliografía



UML y Patrones

Introducción al análisis y diseño orientado a objetos

Craig Larman

Prentice-Hall, 2002