INGENIERÍA DEL SOFTWARE II

TEMA 6

PATRONES DE DISEÑO

PATRONES DE DISEÑO

- Introducción
- Definición
- Patrones GRASP
  - Experto, Creador, Alta Cohesión, Bajo Acoplamiento y Controlador
  - Polimorfismo, Fabricación Pura, Indirección y Variaciones Protegidas
- Patrones GoF
  - Adaptador, Factoría, *Factoría Abstracta*, Singleton, Estrategia, Composite, *Fachada*, y *Observador*

Introducción

- Durante el diseño OO es frecuente encontrarse repetidamente con ciertos tipos de problemas
- Para analizar, compartir y documentar el conocimiento sobre dichos tipos de problemas se han desarrollado los patrones de diseño
- Los patrones también se aplican en otras partes del desarrollo (análisis)
- Recomendable usarlos en cualquier momento
- Aprovechar el conocimiento y la experiencia existente, no partir de cero

Definición

- Un patrón es un conjunto de información que proporciona respuesta a un conjunto de problemas similares
- Para ello se aíslan los aspectos comunes y su solución, y se añaden cuantos comentarios y ejemplos sean oportunos
- Es una solución a un problema en un contexto
  - contexto: las situaciones recurrentes
  - problema: metas y restricciones en el contexto
  - solución: diseño para conseguir las metas dentro de las restricciones

Ejemplo

- Tengo una clase "Depuracion" especializada en guardar mensajes de error en un fichero "errores.log“
- Aunque esa clase será usada por varios componentes sólo necesitaré una instancia de ella (esto se debe a que si más de una instancia escribiese a la vez el fichero, éste quedaría inservible)
- El problema genérico subyacente es evitar que una clase se instancie más de una vez, el patrón que da respuesta a este problema se llama “Singleton”

Descripción

- Ayudan a capturar conocimiento y crear un vocabulario
- Hacen el diseño orientado a objetos más flexible, elegante y en algunos casos reusable
- Compuestos de:
  - Nombre: una o dos palabras, poder de abstracción, facilita la asimilación, la incorporación a la memoria y la comunicación
  - Problema: describe cuando aplicar el patrón y su contexto
  - Solución: detalla el mejor diseño
  - Consecuencias: ventajas y desventajas
  - Ejemplos, Patrones relacionados…

Patrones GRASP

- General Responsibility Assignment Software Patterns (GRASP)
- Una vez tenemos el Modelo de Dominio hay que decidir qué métodos añadir y dónde, es la parte esencial del desarrollo OO
- GRASP nos da unos principios generales para asignar responsabilidades
- Se utilizan sobre todo en la realización de diagramas de interacción

Patrones GRASP

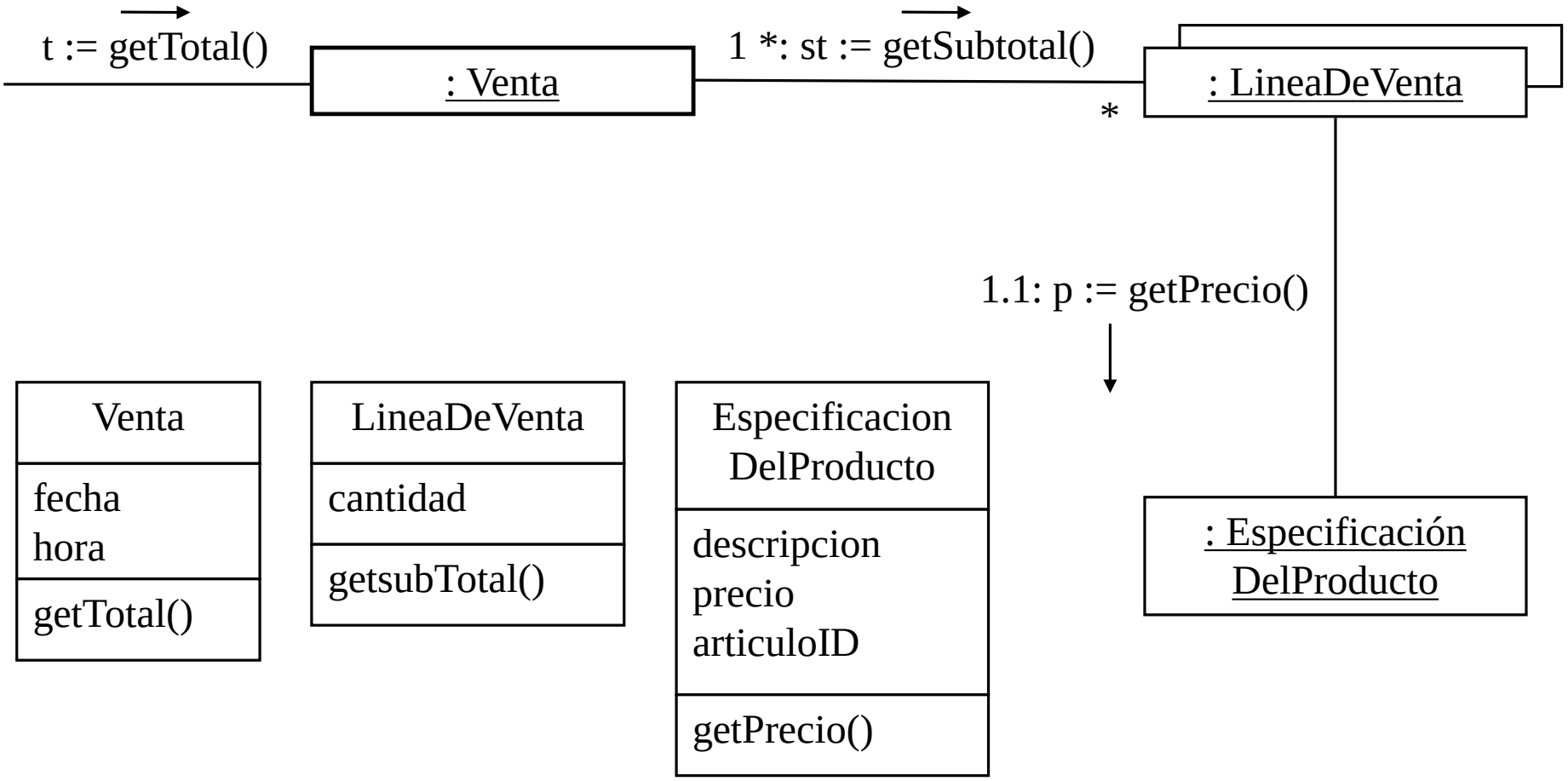
- Básicamente las responsabilidades de un objeto son conocer y hacer, implicarán más o menos métodos y clases
- Cinco principales
  - Experto (en Información)
  - Creador
  - Alta Cohesión
  - Bajo Acoplamiento
  - Controlador

Experto

- Problema: ¿Cuál es el principio más básico para añadir responsabilidades en una clase?
- Solución: Asignar responsabilidades al experto de la información, es decir, a la clase que tiene la información necesaria para llevar la tarea a cabo
- Consecuencias:
  - Encapsulamiento de la información, y por ende el bajo acoplamiento
  - El comportamiento distribuido entre las clases, clases más cohesivas

Experto

- Ejemplo:



Experto

- Contraindicaciones: En algunas ocasiones no es una guía deseable sobre todo por problemas de cohesión, acoplamiento y duplicación de código
  - si Venta debe ser almacenada en una base de datos, el Experto sugiere a Venta como responsable, pues contiene mucha de la información a almacenar
- Patrones relacionados: Bajo Acoplamiento y Alta Cohesión

Creador

- Problema: ¿Quién debe ser responsable en la creación de una nueva instancia de una clase?
- Solución: Una clase B tiene la responsabilidad para crear una instancia de la clase A si
  - B agrega objetos de A
  - B contiene objetos de A
  - B almacena objetos de A
  - B usa objetos de A
  - B tiene los datos necesarios para inicializar a A cuando este es creado
- Consecuencias: Soporta el bajo acoplamiento, éste no se incrementa

Creador

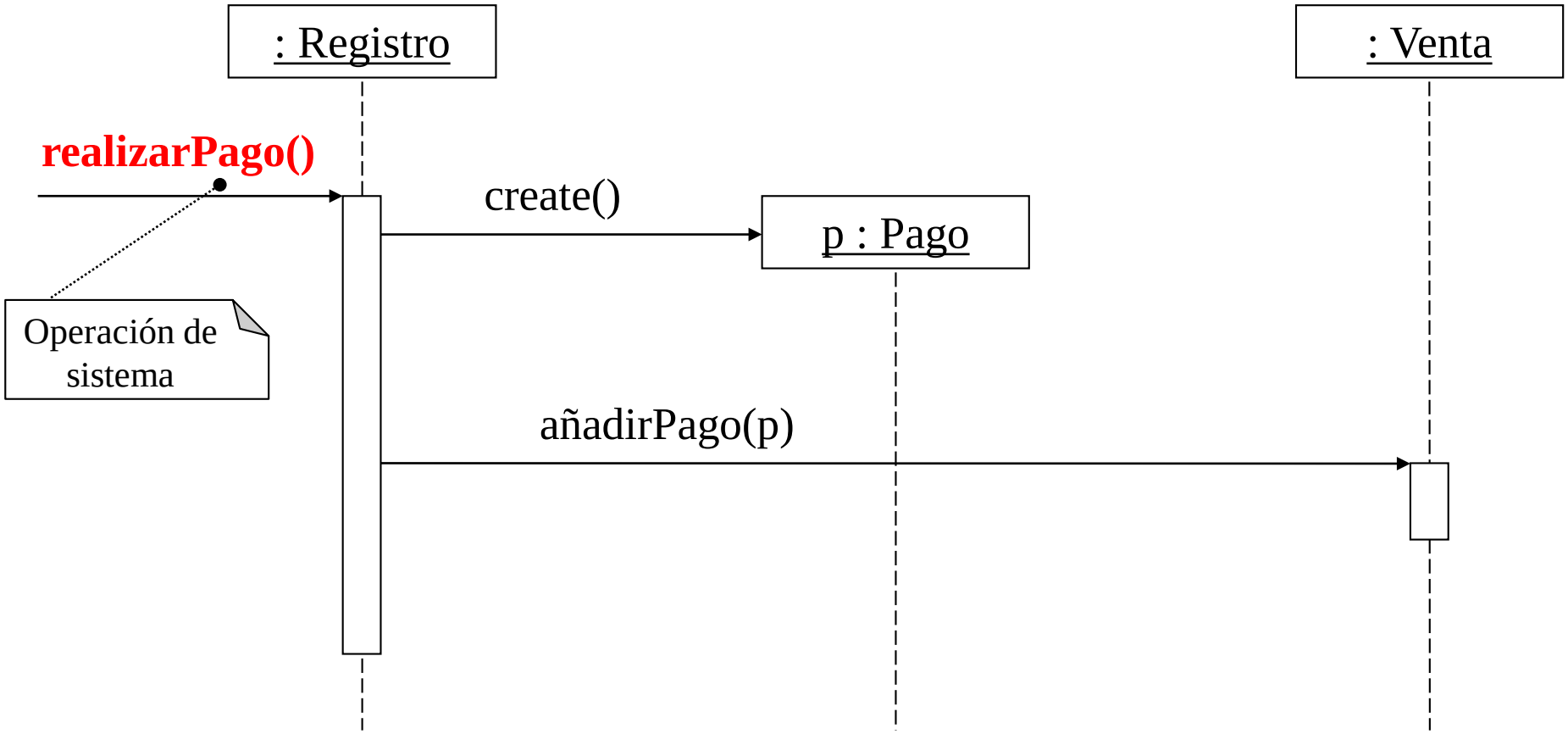
- Ejemplo: En la aplicación PDV, ¿quién debería crear una *LineaDeVenta*? *Venta* de hecho contiene (agrega) muchos objetos de *LineaDeVenta*, debe tener pues la responsabilidad de la creación de instancias de esta clase, debe contener un método *crearLineaDeVenta*
- Contraindicaciones: A menudo la creación de instancias es una tarea compleja, en estos casos es aconsejable delegar la creación a una clase auxiliar denominada *Factoría*
- Patrones relacionados: Bajo Acoplamiento y Factoría

Alta Cohesión

- Problema: ¿Cómo lograr que la complejidad sea lo más manejable posible?
  - Lo relacionadas que están las responsabilidades de una clase, una clase con responsabilidades altamente relacionadas y que no lleva a cabo gran cantidad de trabajo, tiene alta cohesión
- Solución: Asignar responsabilidades procurando que la cohesión sea lo más alta posible
- Consecuencias:
  - Se incrementa la claridad y facilita la comprensión, se simplifica el mantenimiento
  - Implica casi siempre bajo acoplamiento, incrementa reutilización

Alta Cohesión

- Ejemplo: (va contra el patrón Alta Cohesión)

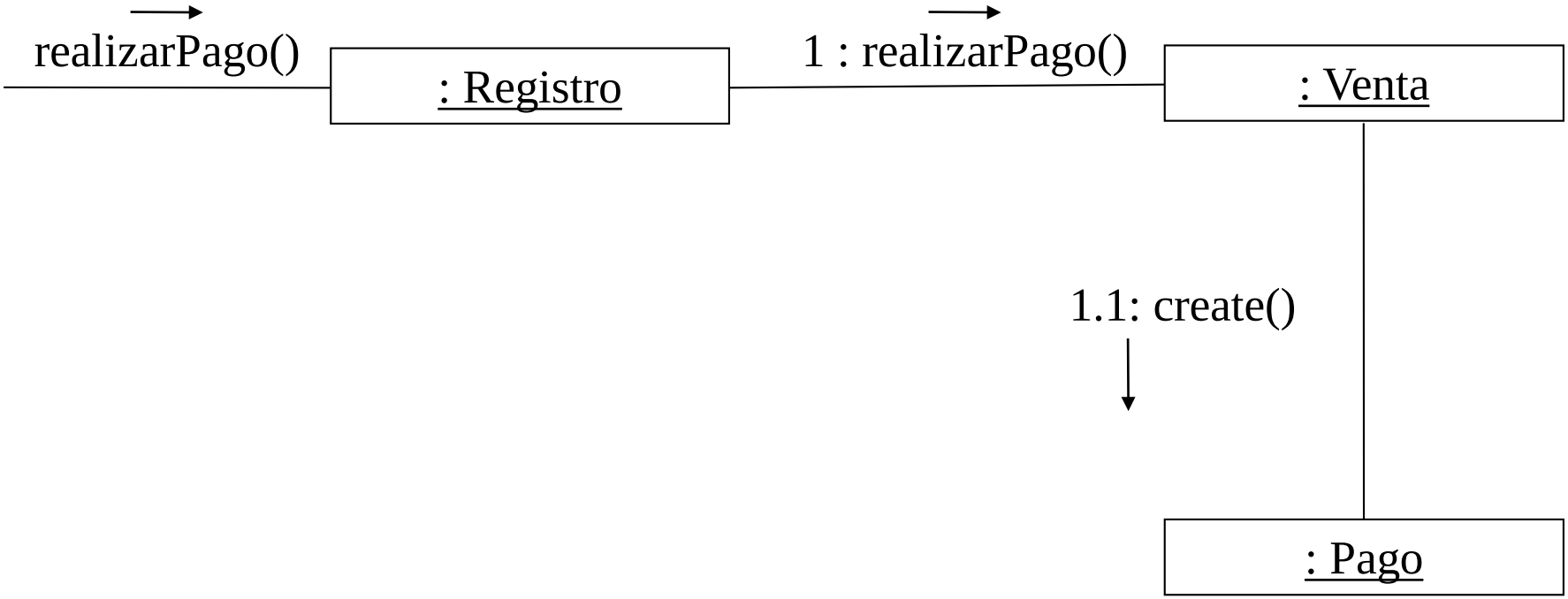


Bajo Acoplamiento

- Problema: ¿Cómo soportar baja dependencia e incrementar la reutilización?
  - Acoplamiento indica que tan fuertemente está conectada una clase con otra, tiene conocimiento de, o influye sobre otra clase, una clase con bajo acoplamiento no depende de otras clases
- Solución: Asignar responsabilidades de tal manera que el acoplamiento sea el menor posible
- Consecuencias:
  - Mejor comprensión de las clases aisladas
  - Convenientes para reutilizar
  - No afectan los cambios en otros componentes

Bajo Acoplamiento

- Ejemplo:



- Patrones relacionados:
  - Variaciones Protegidas

Controlador

- Problema: ¿Quién debe manejar eventos del sistema?
  - Controlador es un objeto responsable del manejo de los eventos del sistema, que no pertenece a la interfaz del usuario, el controlador recibe la solicitud del servicio desde la capa GUI, y coordina su realización delegando a otros objetos
- Solución: Responsabilidades para el manejo de mensajes de eventos del sistema a una clase que
  - Representa al conjunto del sistema o negocio (Controlador Fachada)
  - Representa algo del mundo real que está activo (Controlador de Roles)
  - Representa un administrador artificioso para todos los eventos del sistema (Controlador Caso de Uso)

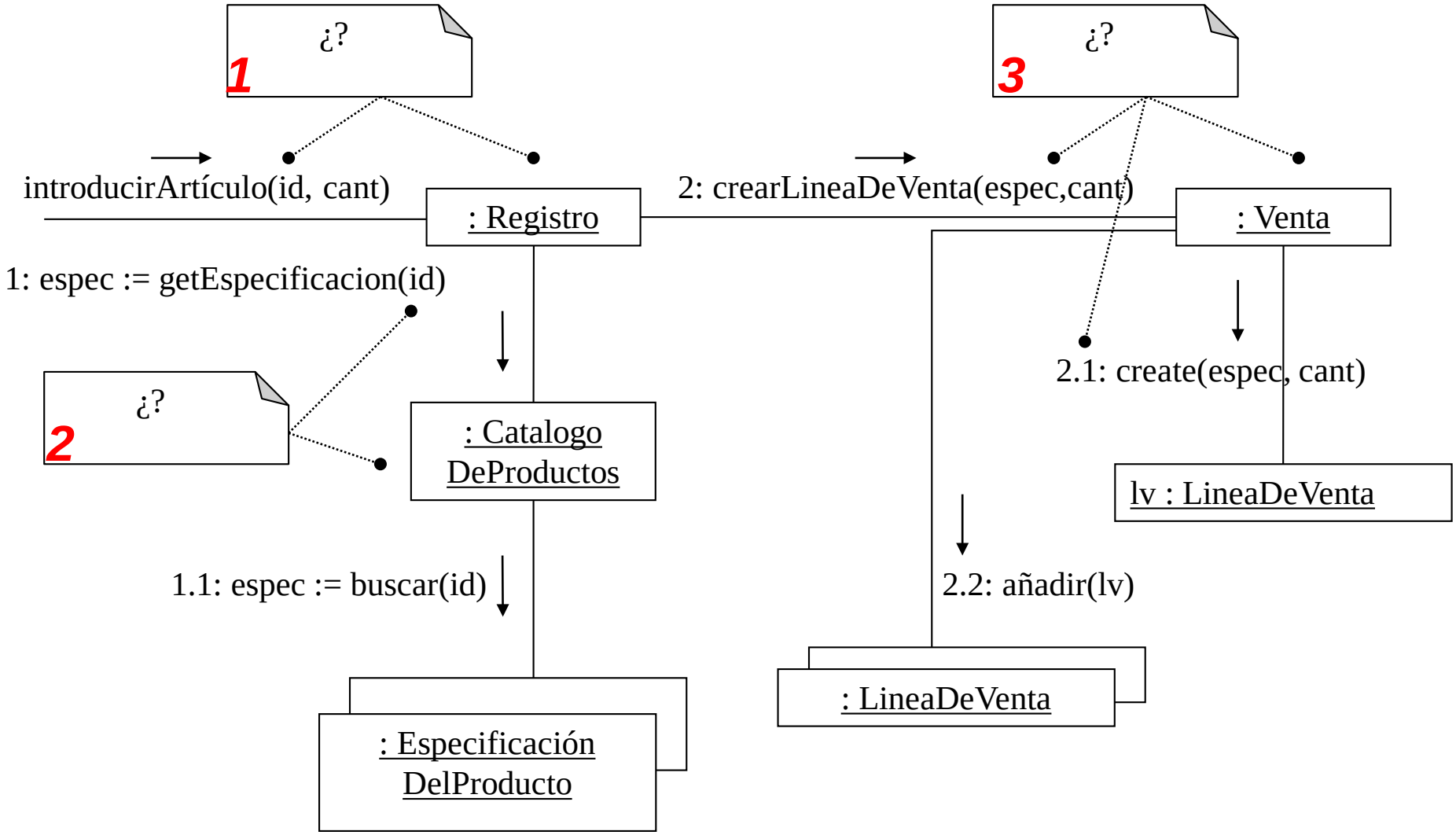
Controlador

- Ejemplo:
  - En el PDV hay varias operaciones de sistema como *finalizarVenta()*, *introducirArtículo()* o *introducirArtículoDevuelto()*
  - siguiendo el patrón Controlador podríamos asignar la responsabilidad de manejar estos eventos a una clase que represente al sistema como *Registro*, sería un controlador de Fachada
  - o si hay muchos eventos de sistema, también podríamos asignársela a controladores de los distintos casos de uso, como *ProcesarVentaControlador* o *GestionarDevolucionesControlador*
  - si el *Registro* asume demasiadas responsabilidades pierde la cohesión (compromiso)

Controlador

- Consecuencias:
  - Potencial para reutilizar interfaces conectables
  - Razonamiento sobre el funcionamiento de los casos de uso (valida la secuencia de operaciones, capturando información de estado)
- Patrones relacionados:
  - Command, Fachada, Capas, Fabricación Pura

¿Qué Patrones?



Más Patrones GRASP

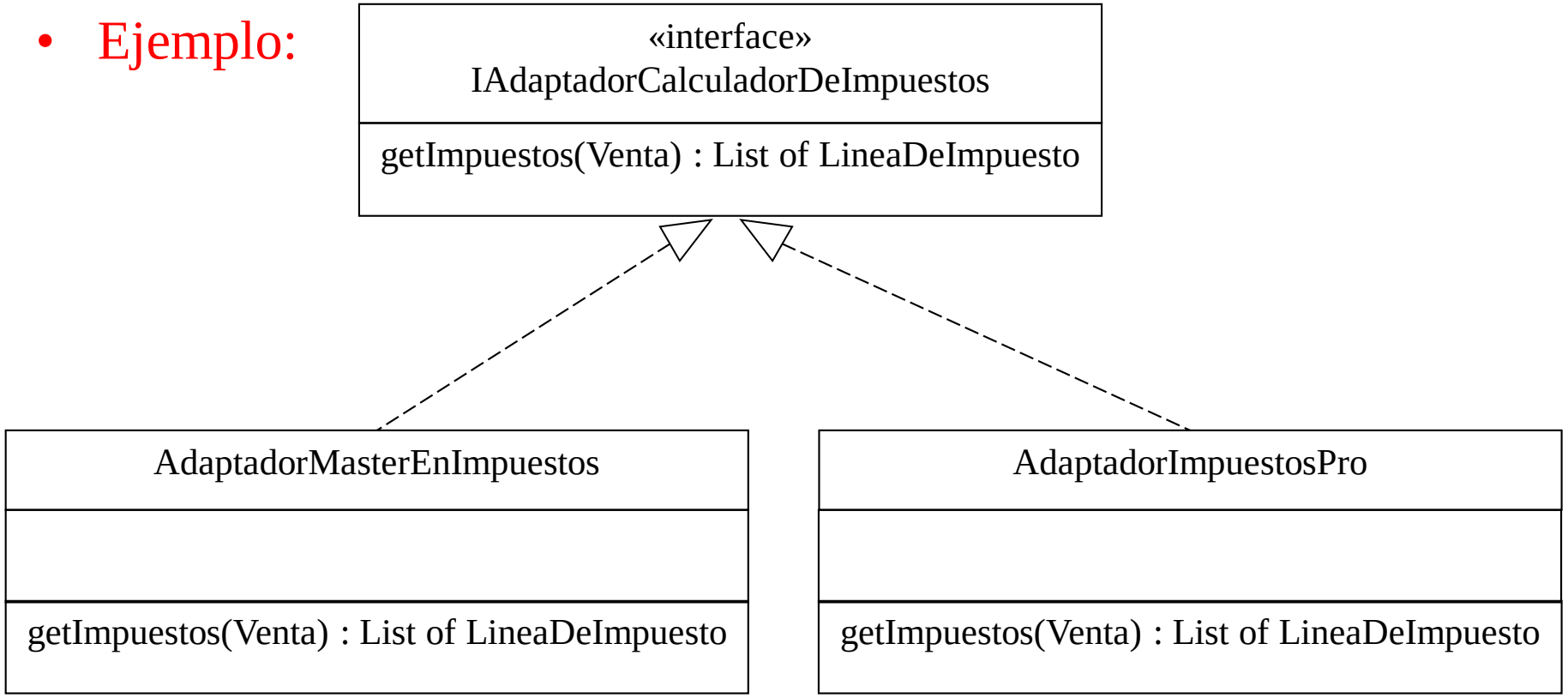
- Sigamos aumentado vocabulario…
  - Polimorfismo
  - Fabricación Pura
  - Indirección
  - Variaciones Protegidas

Polimorfismo

- Problema: ¿Cómo manejar alternativas basadas en el tipo? ¿Cómo crear componentes conectables?
- Solución: Cuando el comportamiento relacionado varía según el tipo (clase) asigne la responsabilidad para el comportamiento utilizando operaciones polimórficas a los tipos para los que varía el comportamiento
  - Corolario: Evítense las comprobaciones acerca del tipo de un objeto y procure no utilizar la lógica condicional
- Consecuencias:
  - Se añaden fácilmente extensiones necesarias para nuevas variaciones
  - Las nuevas implementaciones se introducen sin afectar a los clientes

Polimorfismo

- Ejemplo:



- Patrones relacionados: Variaciones Protegidas, varios de los patrones de diseño GoF como Adaptador, Command, Composite, Proxy, Estado y Estrategia

Fabricación Pura

- Problema: ¿Qué objetos deberían tener la responsabilidad cuando no se quiere violar los objetivos de alta cohesión y bajo acoplamiento?
- Solución: Asigne un conjunto de responsabilidades altamente cohesivas a una clase artificial que no representa un concepto del dominio
- Consecuencias:
  - Se soporta alta cohesión puesto que las responsabilidades se factorizan en una clase de grano fino
  - El potencial para reutilizar aumenta

Fabricación Pura

- Ejemplo:
  - Se necesita soporte para almacenar la *Venta* en una base de datos se podría suponer según el Experto que la propia clase debería tener la responsabilidad
  - La tarea implica un número amplio de operaciones de bases de datos no relacionadas con las ventas (baja cohesión)
  - Además tiene que acoplarse con el interfaz de la base de datos
  - Y por último la tarea de almacenar objetos puede ser necesaria para más objetos
  - Podríamos crear una clase *AlmacenamientoPersistente*

Fabricación Pura

- Contraindicaciones: llevar el principio al extremo
- Patrones relacionados: Bajo Acoplamiento, Alta Cohesión, los GoF (Adaptador, Command, Estrategia, etc.) y prácticamente el resto de patrones de diseño

Indirección

- Problema: ¿Dónde asignar una responsabilidad para evitar el acoplamiento directo entre dos objetos?
- Solución: Asigne la responsabilidad a un objeto intermedio que medie entre otros componentes para que no estén acoplados directamente, el intermediario crea una indirección
- Consecuencias:
  - Disminuye el acoplamiento entre componentes

Indirección

- Ejemplos:
  - El ejemplo de *AdaptadorCalculadorDeImpuestos* que se vio anteriormente actúa como intermediario con los calculadores externos de impuestos gracias al Polimorfismo
  - También *AlmacenamientoPersistente*, que es un ejemplo de Fabricación Pura, es un intermediario entre la *Venta* y la base de datos

Indirección

- Patrones relacionados:
  - Muchos de los patrones existentes son especializaciones de Indirección
  - Está relacionado con Variaciones Protegidas, Bajo Acoplamiento, muchos GoF como Adaptador, Puente, Fachada, Observador y Mediador

Variaciones Protegidas

- Problema: ¿Cómo diseñar componentes de manera que las variaciones en ellos no tengan repercusión en otros elementos?
- Solución: Asigne responsabilidades para crear una interfaz estable alrededor de los puntos de variaciones previstas o inestabilidad
- Consecuencias:
  - Se añaden fácilmente las extensiones para nuevas variaciones
  - Se pueden introducir nuevas implementaciones sin afectar a los clientes
  - Se reduce el acoplamiento

Variaciones Protegidas

- Ejemplo:
  - El ejemplo de *AdaptadorCalculadorDeImpuestos* también ilustra las Variaciones Protegidas
  - El punto de inestabilidad lo forman las interfaces de los diversos calculadores de impuestos
  - El sistema podrá necesitar integrarse con otros sistemas de cálculo actuales y futuros
  - Mediante la indirección de la interfaz y el polimorfismo se consigue una protección frente a las variaciones en las interfaces externas

Variaciones Protegidas

- Patrones relacionados:
  - La mayoría de los patrones son mecanismos de Variaciones Protegidas
  - Polimorfismo, Indirección y la mayor parte de los GoF

Patrones GoF

- Patrones publicados por Gamma, Helm, Johnson y Vlissides en 1995: patrones de la pandilla de los cuatro (*gang of four*)
  - ampliar lenguaje
  - aprender estilos de diseño
  - además se introducirá más notación UML

23 patrones
15 con frecuencia
8 en este tema

Patrones GoF

- Se clasifican según su propósito en patrones:
  - de creación (Factoría, Factoría Abstracta, Singleton)
    - creación de objetos
  - estructurales (Adaptador, Composite, Fachada)
    - composición de clases y objetos
  - de comportamiento (Observador, Estrategia)
    - modo en que las clases y los objetos interactúan y se reparten las responsabilidades

Adaptador-Wrapper (estructural)

- Problema: ¿Cómo resolver interfaces incompatibles, o proporcionar una interfaz estable para componentes parecidos con diferentes interfaces?
- Solución: Convierta la interfaz de una clase en otra interfaz que es la que esperan los clientes, permite que cooperen clases que de otra forma no podrían cooperar por tener interfaces incompatibles

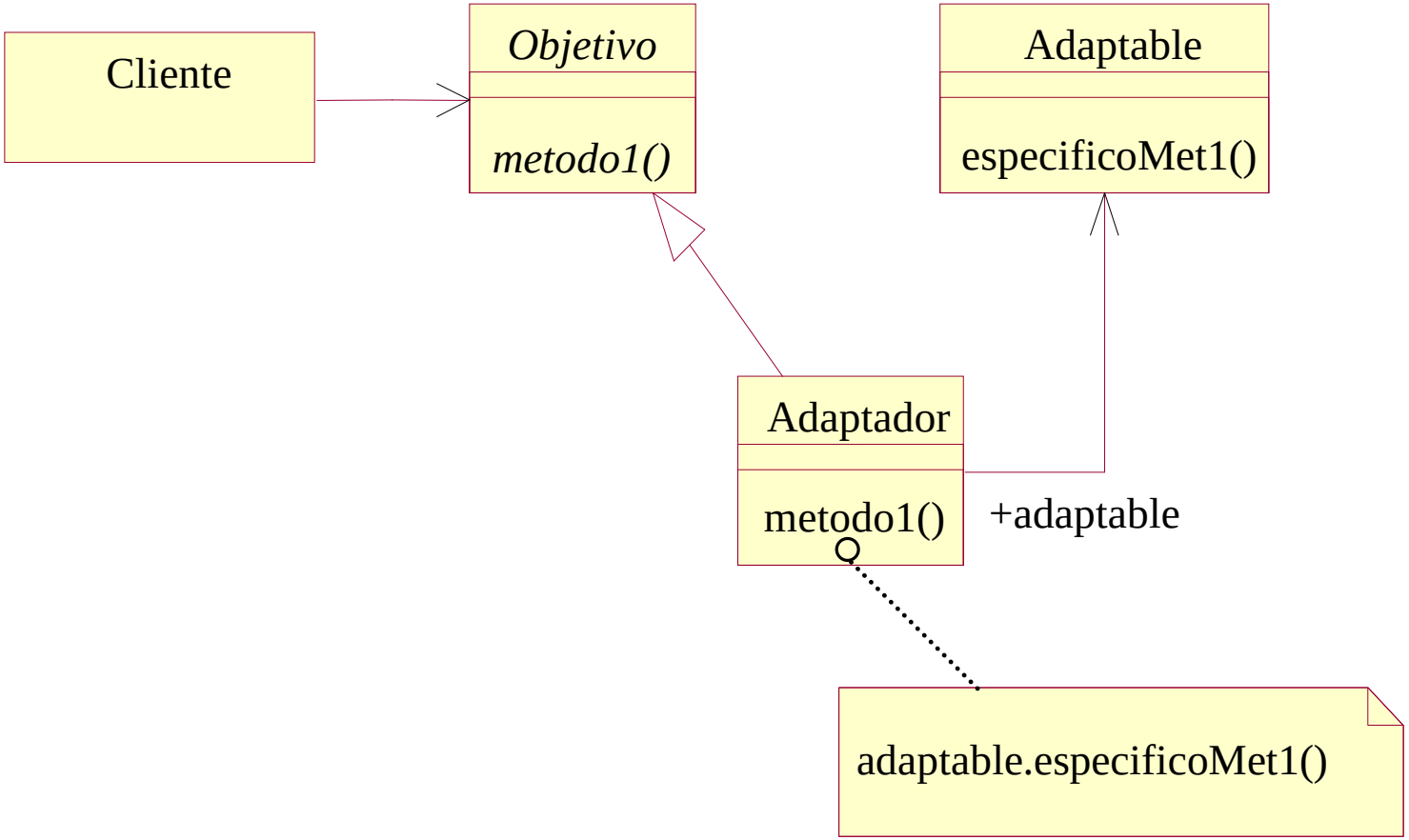
Adaptador

- Ejemplo: en el sistema PDV el ejemplo de Polimorfismo es más específicamente un Adaptador
  - para cada sistema de servicios externos se tiene una interfaz diferente que no puede ser cambiada
  - se añade un nivel de indirección que adapta las distintas interfaces a una compatible con PDV
  - para cada servicio externo se instanciará un adaptador particular (adaptador de impuestos, de contabilidad…)
- El Adaptador ofrece Variaciones Protegidas en paquetes de terceras partes mediante un objeto Indirección que aplica Interfaces y Polimorfismo

Adaptador

- Participantes:
  - Objetivo: Define la interfaz que utiliza el Cliente
  - Cliente: Colabora con objetos que se ajustan a la interfaz Objetivo
  - Adaptable: Define una interfaz existente que necesita ser adaptada
  - Adaptador: Adapta la interfaz de Adaptable a la interfaz Objetivo

Adaptador


Cliente
Objetivo
metodo1()
Adaptable
especificoMet1()
Adaptador
metodo1()
+adaptable
adaptable.especificoMet1()

Adaptador

- Consecuencias:
  - Posibilidad de reutilizar las clases que han sido pensadas para ese Objetivo pero que tenían interfaces no compatibles con el sistema en desarrollo
  - Un adaptador de objetos permite adaptar una clase y sus subclases, puede funcionar para varios adaptables

Factoría (creación)

- Problema: ¿Quién es el responsable de la creación de objetos cuando existen consideraciones especiales, como una lógica de creación compleja o el deseo de separar las responsabilidades de la creación para mejorar la cohesión?
- Solución: Cree un objeto Fabricación Pura denominado Factoría (Factoría Concreta) que resuelva la creación cuando existan consideraciones especiales

Factoría

- Ejemplo: ¿Quién crea Adaptador y cuál debería crearse?
  - si se encarga a algún objeto de dominio excedería la lógica de la aplicación o disminuiría su cohesión
  - mejor asignar a una *FactoriaDeServicios*, crea dinámicamente el Adaptador apropiado leyendo el nombre de la clase desde una fuente externa
  - fácilmente se podría cambiar de Adaptador sin cambiar código

Factoría

- Consecuencias:
  - Separar la responsabilidad de la creación compleja en objetos de apoyo
  - Ocultar la lógica de creación potencialmente compleja
  - Permitir introducir estrategias para mejorar el rendimiento de la gestión de memoria

Factoría Abstracta (creación)

- Problema: ¿Cómo crear familias de clases relacionadas que implementan una interfaz común?
- Solución: Proporcione una interfaz para crear familias de objetos relacionados o que dependen entre sí, sin especificar sus clases concretas

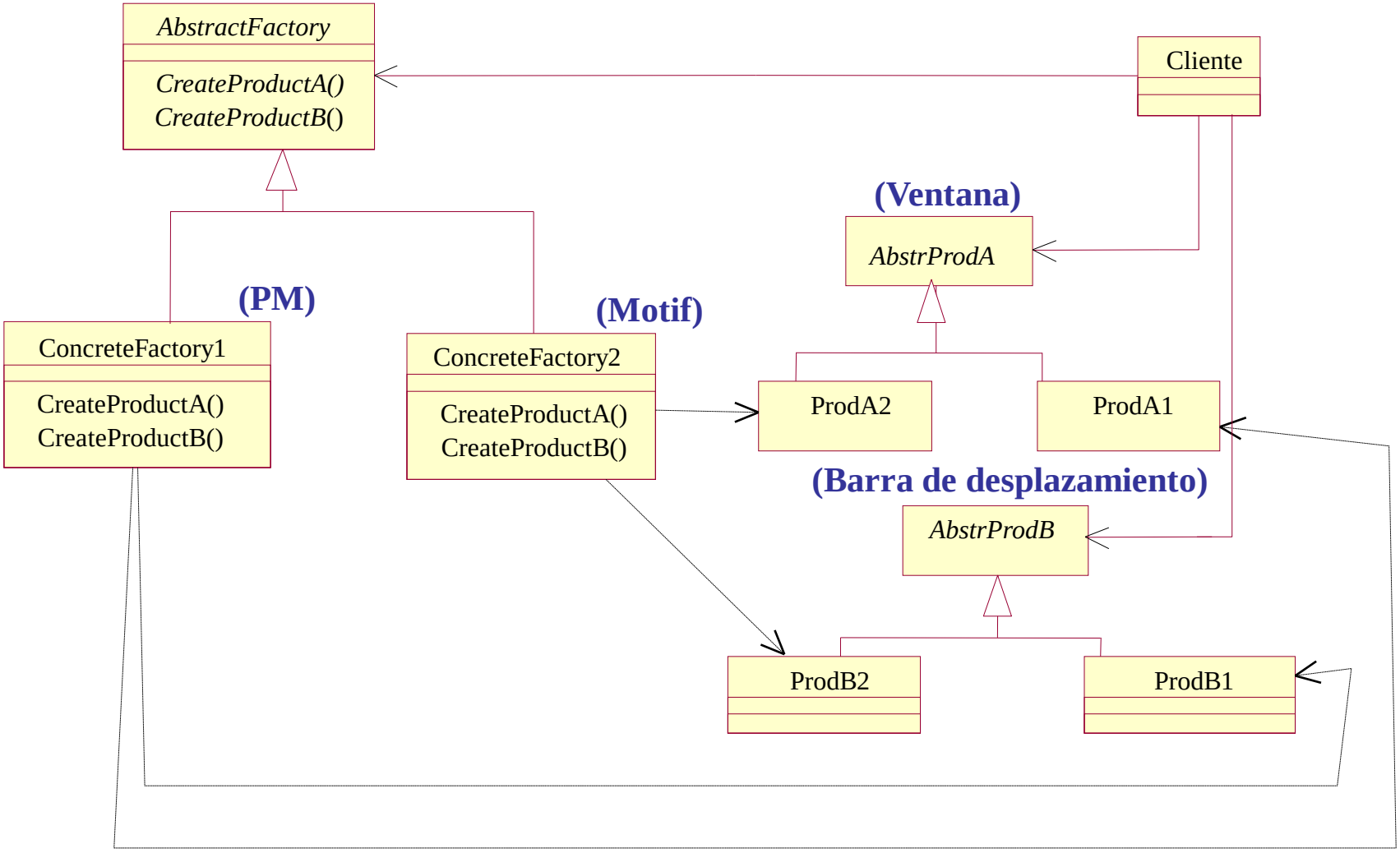
Factoría Abstracta

- Ejemplo: PDV necesario controladores hardware (como el del cajón de la caja u otros dispositivos) dependientes del fabricante, puede haber familias de clases y cada familia implementa la misma interfaz
- Otro Ejemplo: Aplicación para crear interfaces de usuario que admita múltiples estándares como Motif o Presentation Manager, la Factoría Abstracta define una interfaz para cada una de las utilidades básicas, cada subclase concreta define la utilidad apropiada para su estándar particular

Factoría Abstracta

- Participantes:
  - FabricaAbstracta: Declara la interfaz para operaciones que crean objetos producto abstractos
  - FabricaConcreta: Implementa las operaciones para crear objetos producto concretos
  - ProductoAbstracto: Declara una interfaz para un tipo de objeto producto
  - ProductoConcreto: Define un objeto producto para que sea creado por la fábrica correspondiente, implementa ProductoAbstracto
  - Cliente: Usa las interfaces declaradas por la clases FabricaAbstracta y ProductoAbstracto

Factoría Abstracta



Factoría Abstracta

- Aplicabilidad:
  - Un sistema es independiente de cómo sus productos son creados, compuestos y representados
  - Un sistema debe ser configurado como una familia de productos de entre varias
  - Una familia de objetos producto relacionados está diseñada para ser utilizada conjuntamente y se necesita hacer cumplir esta restricción *(botón y barra de desplazamiento Motif)*
  - Se quiere proporcionar una librería de productos y se quiere revelar sólo la interfaz, no la implementación

Factoría Abstracta

- Consecuencias:
  - Aísla a los clientes de las clases concretas de implementación
  - Favorece la consistencia entre productos
  - Pero es difícil soportar nuevas clases de productos

Singleton (creación)

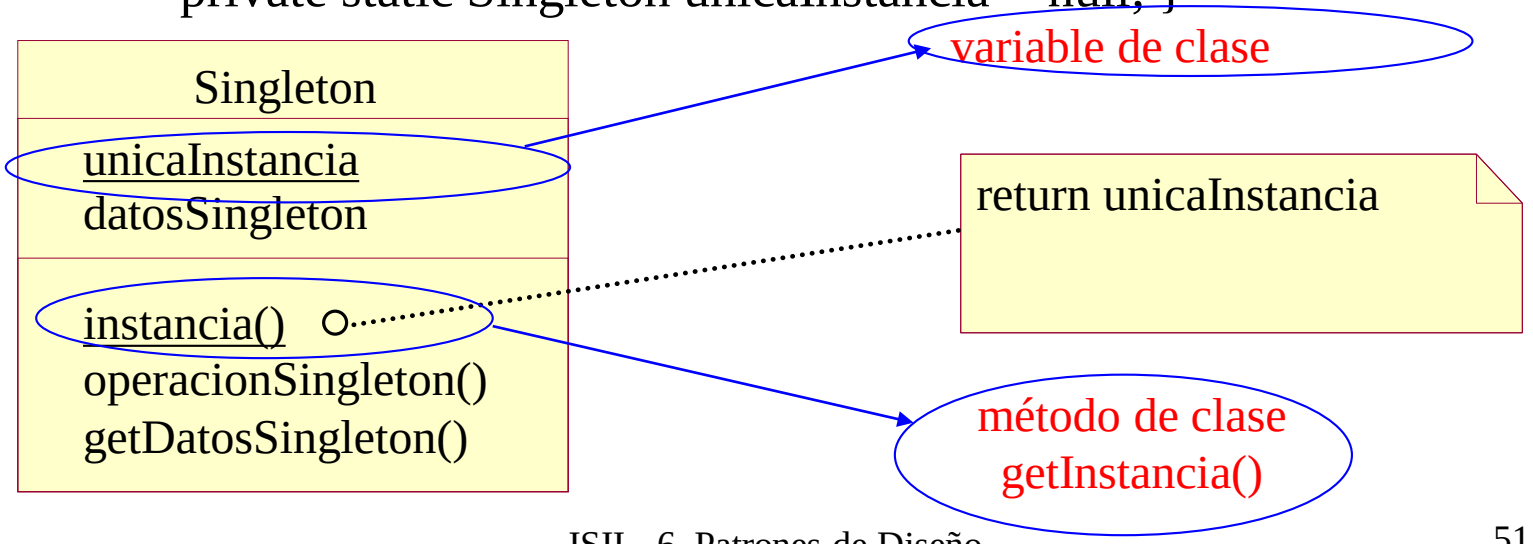
- Problema: ¿Cómo podemos asegurar que una clase tiene exactamente una única instancia y que ésta sea fácilmente accesible?
- Solución: Defina un método estático de la clase que devuelva el singleton

Singleton

- Implementación:

```
class Singleton {
    public static Singleton instancia() {
        if (unicaInstancia = = null)
            unicaInstancia = new Singleton();
        return unicaInstancia; }
    protected Singleton();
    private static Singleton unicaInstancia = null; }
```

para evitar que
se invoque al constructor

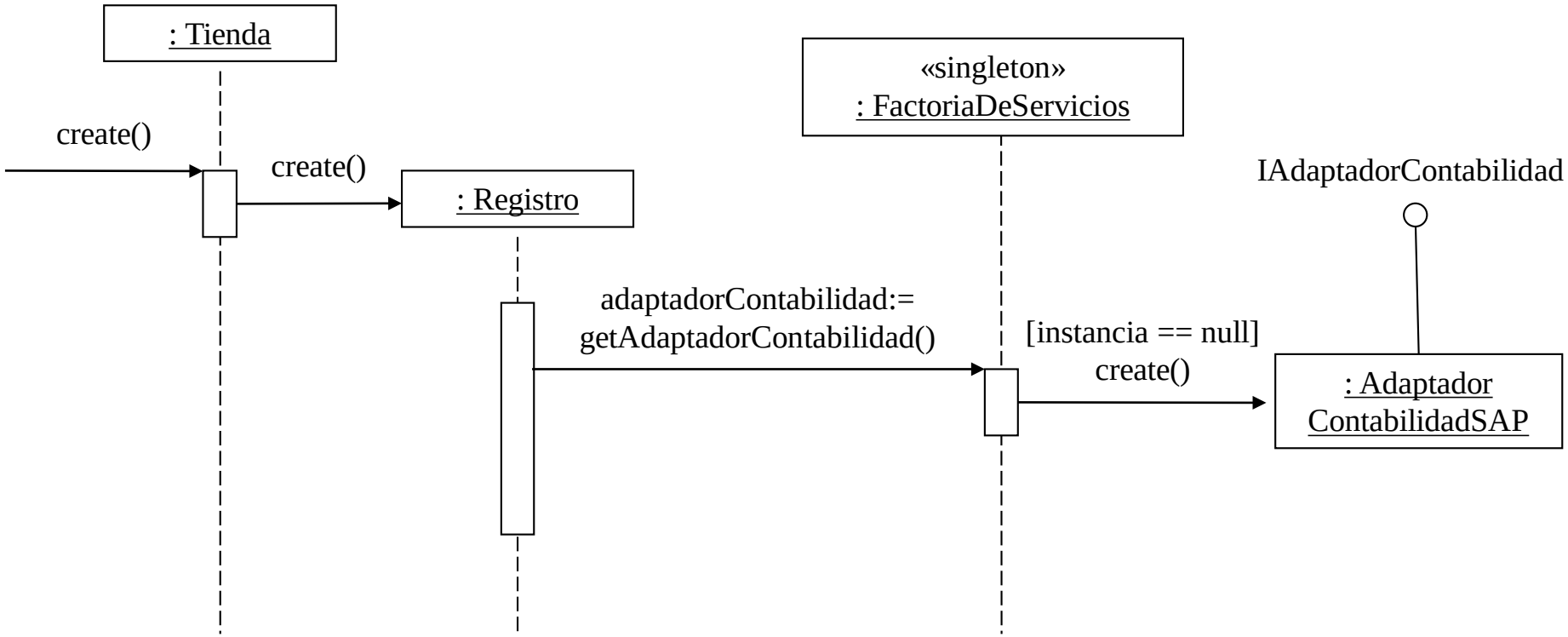


Singleton

- Ejemplo: En el caso PDV, ¿quién crea la *FactoriaDeServicios* y cómo se accede?
  - sólo debería haber una instancia aunque se invoquen sus métodos desde varios sitios
  - método estático que proporciona la única instancia del singleton
  - los Adaptadores son generados desde una Factoría Singleton

Singleton

- Inicializar sistema, ¿quién debe crear el adaptador?



Singleton

- Consecuencias:
  - Permite el manejo de objetos únicos y que sean accesibles a otros objetos
  - Acceso controlado a la única instancia
- Patrones relacionados:
  - Se utiliza a menudo para Factoría y Fachada

Estrategia (comportamiento)

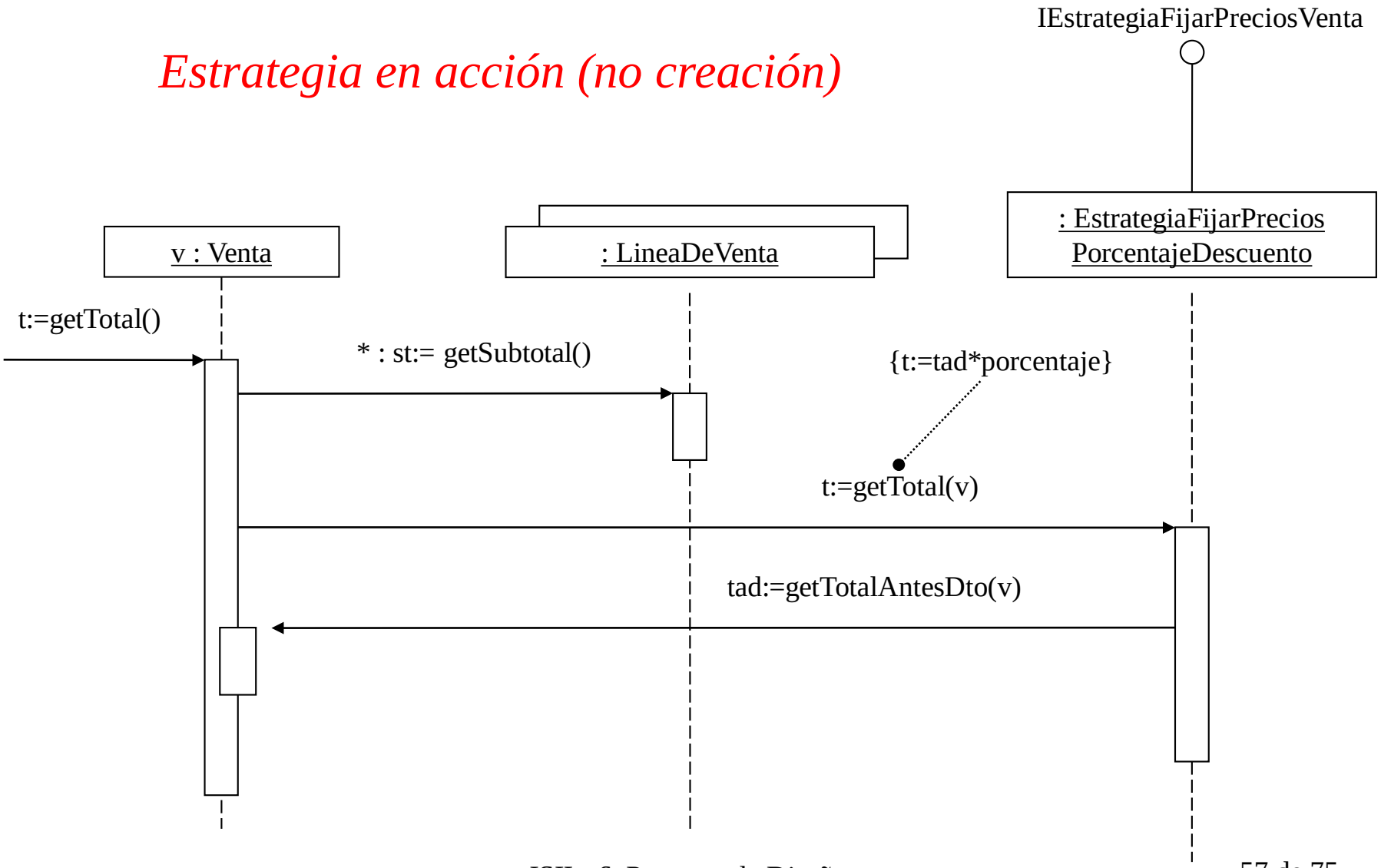
- Problema: ¿Cómo diseñar diversos algoritmos o políticas que estén relacionadas? ¿Cómo diseñar que éstos puedan cambiar?
- Solución: Defina cada algoritmo, política o estrategia en una clase independiente con una interfaz común

Estrategia

- Ejemplo: en el PDV una política de precios puede variar, p.e. un descuento puede ser un porcentaje sobre el total, o un fijo a partir de una cantidad
  - creamos múltiples clases estrategia cada una con un método polimórfico *getTotal*
  - un objeto estrategia se conecta a un objeto *Venta*
  - cuando se envía el mensaje *getTotal* a *Venta*, ésta delega parte del trabajo a su objeto estrategia, pasa una referencia a sí misma para que estrategia tenga visibilidad de ella

Estrategia

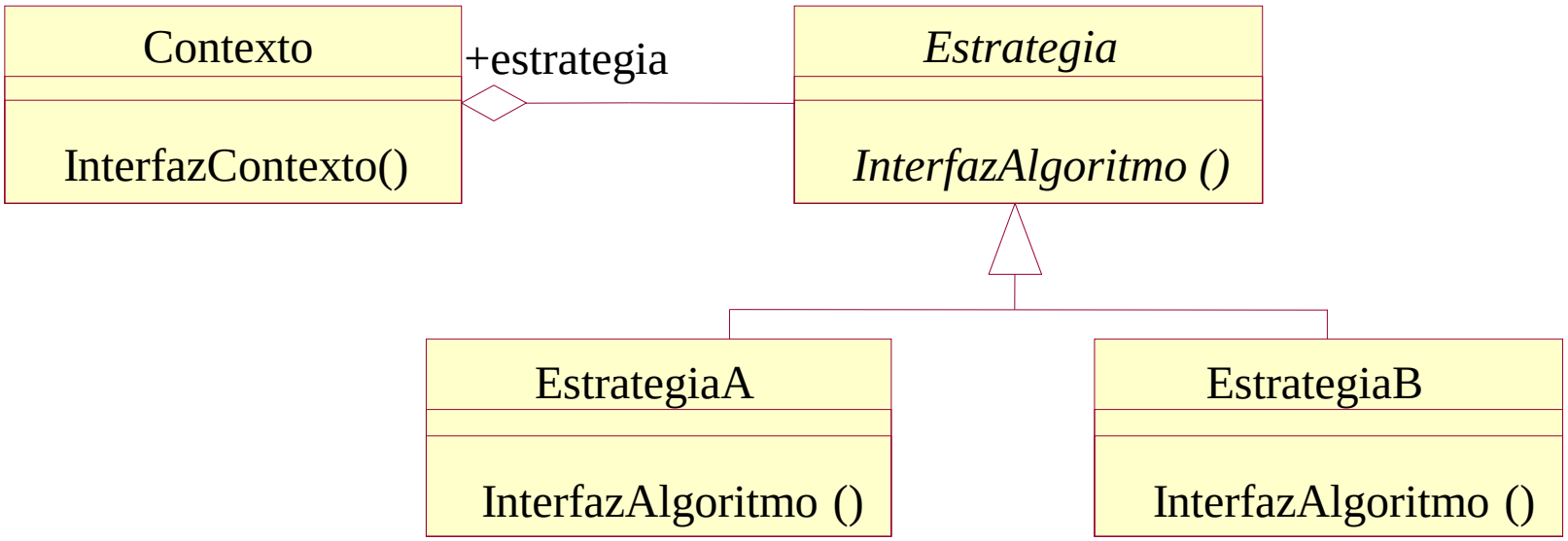
Estrategia en acción (no creación)



Estrategia

- Participantes:
  - Estrategia: Interfaz común a todos los algoritmos permitidos
  - Estrategia concreta: Implementa los algoritmos correspondientes, usando la interfaz Estrategia
  - Contexto: Mantiene una referencia al objeto Estrategia, se configura con un objeto Estrategia Concreta

Estructura Estrategia



Estrategia

- Consecuencias:
  - Configurar una clase con uno de varios comportamientos posibles
  - Se necesitan diferentes variantes de un algoritmo para un mismo comportamiento
  - Una clase define muchos comportamientos que aparecen como sentencias CASE en sus métodos, elimina sentencias CASE
  - El cliente puede elegir entre diferentes estrategias o implementaciones pero debe conocer los detalles que las diferencian
- Patrones relacionados:
  - Se crea mediante una Factoría que se accede como Singleton

Composite (estructural)

- Problema: ¿Cómo tratar un grupo o una estructura compuesta del mismo modo que un objeto atómico?
- Solución: Defina las clases para los objetos compuestos y atómicos de manera que implementen el mismo interfaz

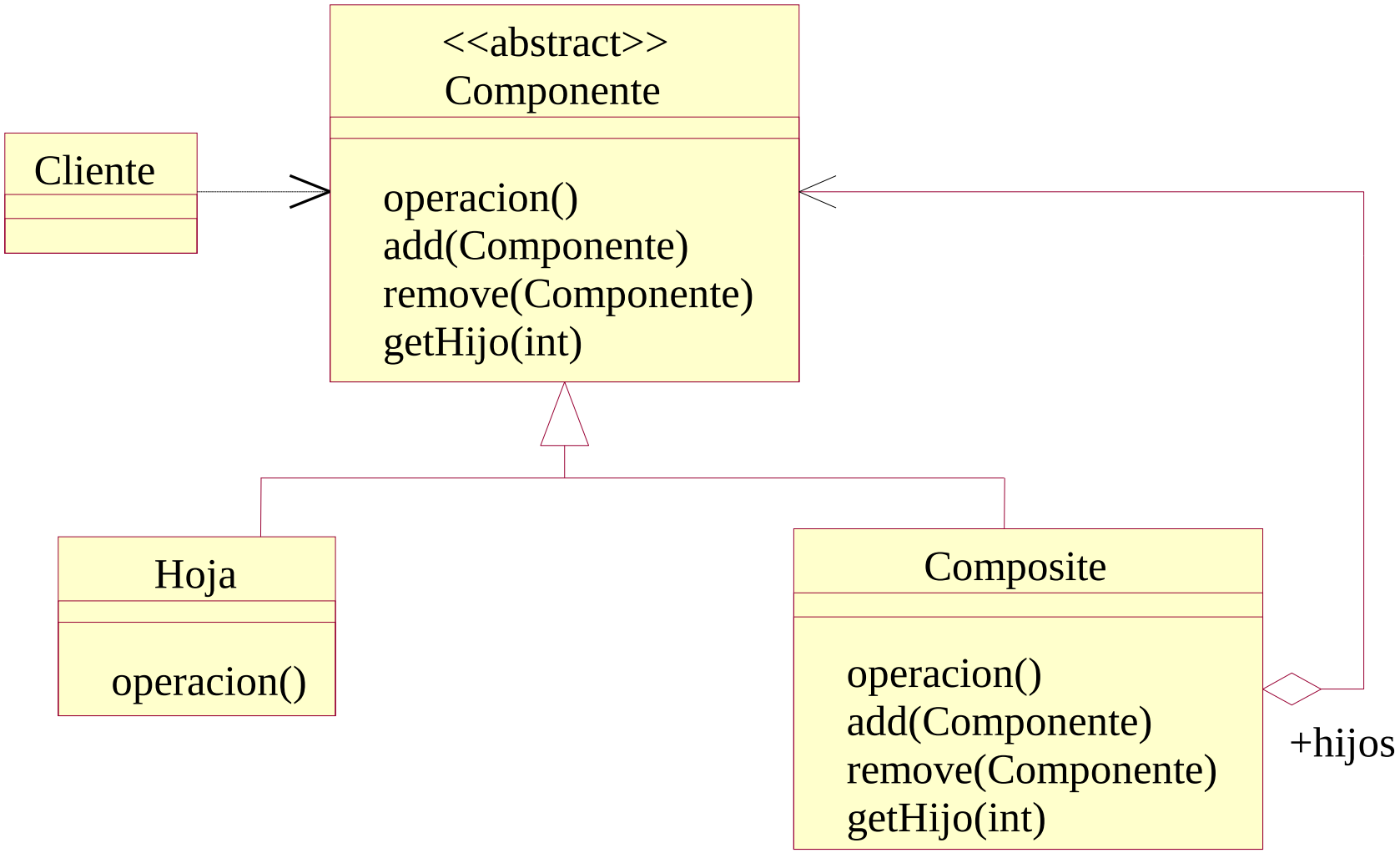
Composite

- Ejemplo: En el caso del PDV para establecer el descuento pueden coexistir varias estrategias a la vez
  - estrategia relacionada con el tipo de cliente o relacionada con el producto
  - se puede crear una estrategia que implemente la misma interfaz y que al mismo tiempo englobe varias estrategias, se van añadiendo en distintos puntos de la ejecución del caso de uso

Composite - Participantes

- Componente: Declara la interfaz para los objetos de la composición, e implementa también los métodos que determinan el comportamiento de las diferentes clases tanto compuestas como hijas y permite tener acceso a componentes en la estructura recursiva
- Hoja: Define el comportamiento de los objetos que no tienen hijos en la composición
- Compuesto: Maneja los hijos de los diferentes componentes compuestos y determina el comportamiento de aquellos componentes capaces de tener hijos
- Cliente: Manipula los objetos de la composición a través del componente

Composite


<<abstract>>
Componente
operacion()
add(Componente)
remove(Componente)
getHijo(int)
Cliente
Hoja
operacion()
Composite
operacion()
add(Componente)
remove(Componente)
getHijo(int)
+hijos

Composite

- Consecuencias:
  - Define jerarquías parte/todo
  - Los clientes ignoran la diferencia entre objetos compuestos y objetos individuales que los forman, pueden tratar objetos primitivos y compuestos de modo uniforme
  - Jerarquía con clases que modelan objetos primitivos y objetos compuestos, se permite composición recursiva
  - Es fácil añadir nuevos tipos de componentes

Fachada (estructural)

- Problema: Se requiere una interfaz común para un conjunto de implementaciones o interfaces dispares
- Solución: Defina un único punto de conexión de un subsistema, este objeto fachada presenta una única interfaz unificada y es responsable de colaborar con los clientes

(Controlador de fachada)

Fachada

- Ejemplo:
  - En el PDV supongamos que si se paga mediante un vale sólo se permite un único artículo
    - se quiere diseñar de manera que afecte poco a los componentes software existentes
    - ejemplo de regla de negocio conectable
  - Se define un objeto llamado *FachadaMotorReglasPDV* que será el encargado de evaluar un conjunto de reglas
  - Normalmente se accederá mediante el patrón Singleton, la complejidad se oculta tras la fachada

Fachada

- Consecuencias:
  - Oculta a los clientes los componentes del subsistema
  - Promueve acoplamiento débil entre el subsistema y los clientes
  - No impide que las aplicaciones usen las clases del subsistema

Observador (comportamiento)

- Problema: Diferentes tipos de objetos suscriptores están interesados en el cambio de estado o eventos de un emisor, y quieren reaccionar cada uno a su manera, además el emisor quiere mantener bajo acoplamiento con los suscriptores
- Solución: Defina una interfaz "suscriptor" u "oyente" (*listener*) para que objetos registrados como suscriptores de un emisor la implementen y se les pueda notificar cuando ocurren cambios de estado o eventos del emisor

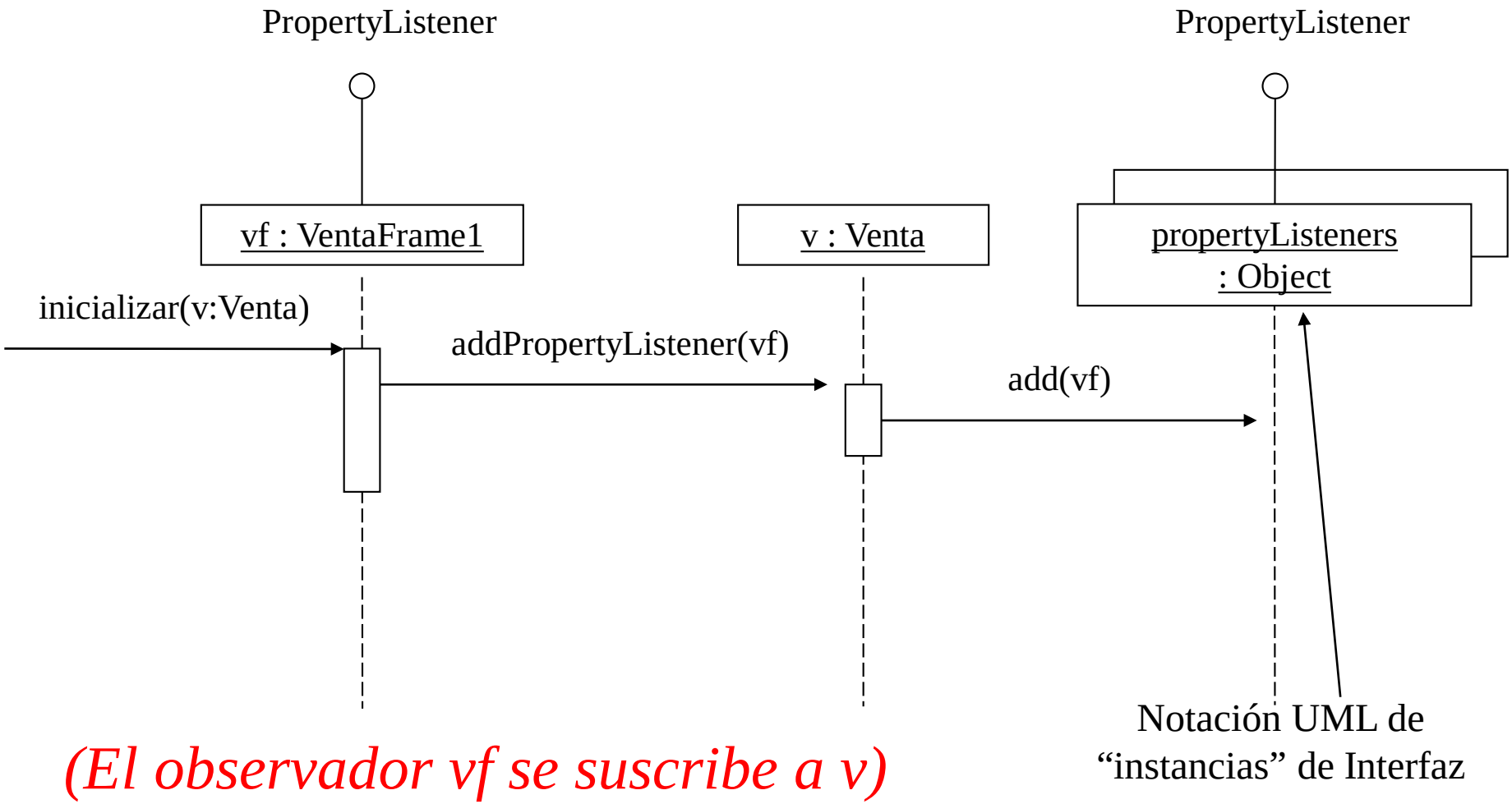
Observador

- Ejemplo:
  - Una ventana GUI debe actualizar la información que muestra sobre el total de la venta cuando éste cambia
  - ¿debería *Venta* enviar mensaje a la ventana?
  - según el principio de separación Modelo-Vista ¡no!

Observador

- Pasos a seguir:
  - Se define una interfaz con la operación para actualizar los objetos ante cambios en el emisor
  - Se define la ventana(s) que implementa la interfaz, y el método anterior
  - Cuando se inicializa la ventana se le pasa la instancia *Venta* de la que está mostrando su total
  - La ventana se registra o suscribe a la instancia *Venta* por medio de la operación de suscripción definida en la clase *Venta*

Observador

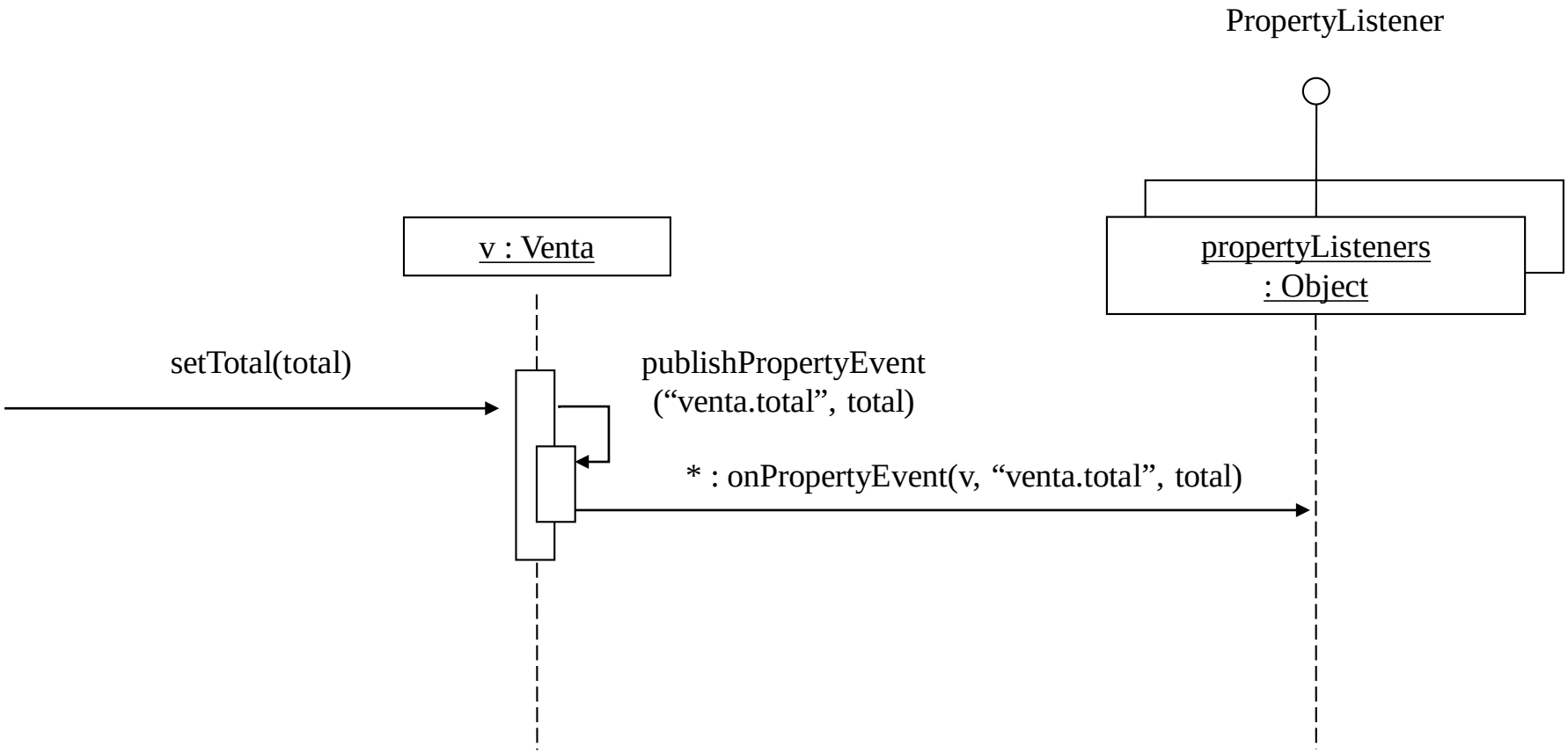


(El observador vf se suscribe a v)

Observador

- Pasos a seguir (continuación):
  - *Venta* no conoce a los objetos ventana, sólo se acopla con una interfaz
  - Como emisor, *Venta* cuando cambia el total itera sobre todos los objetos registrados y se lo notifica a cada uno

Observador



(v publica un evento a todos los suscriptores)

Observador

- Consecuencias:
  - Proporciona un modo de acoplar débilmente los objetos en términos de comunicación
  - Los emisores conocen a los suscriptores sólo a través de una interfaz y desconoce incluso el número de objetos con que se comunica
  - Los suscriptores pueden registrarse o darse de baja dinámicamente

Bibliografía

- *Larman*, *"UML y Patrones: una Introducción al Análisis y Diseño Orientado a Objetos y al Proceso Unificado," Pearson, 2003*