Diseño Orientado a Objetos en UML

- Análisis vs. Diseño en UML
- Modelo de casos de uso reales
- Diagramas de paquetes
- Diagramas de clase y normalización
- Diagramas de interacción
- Asignación de Responsabilidades: patrones GRASP
- Diseño de la capa de dominio
- Diseño de la capa de presentación
- Diseño de la capa de Gestión de Datos

Análisis vs. Diseño en UML (1)

- Se usan los mismos modelos en ambas fases
- La visión de los modelos es muy distinta:
  - Análisis: los modelos definen conceptos para modelar el mundo real (dominio del problema)
  - Diseño: los modelos definen conceptos para modelar una solución (dominio de la solución)
    - Debe contemplarse la solución tecnológica: eficiencia, etc.

Análisis vs. Diseño en UML (2)

- Análisis: qué hace el sistema?

- Resultado del análisis en UML
  - Modelo de Casos de Uso
    - Qué interacción hay entre los actores y el SI?
  - Modelo de Dominio
    - Cuáles son los conceptos relevantes a modelar?
  - Diagramas de secuencia del sistema
    - Qué operaciones debe tener el SI?
  - Contratos de las operaciones
    - Qué hacen las operaciones del SI?

Análisis vs. Diseño en UML (3)

- Diseño: cómo debe ser el SI?

- Resultado del diseño en UML
  - Modelo de Casos de Uso
    - Define la interacción real con una interfaz concreta
  - Diagrama de clases
    - Describe las clases y sus métodos (operaciones)
  - Diagramas de interacción (secuencia)
    - Define las interacciones entre los objetos para responder a un evento externo (operación del sistema)
  - Contratos de las operaciones
    - Definen qué hacen los métodos de las clases

Casos Reales de Uso

- Un caso real de uso describe el diseño concreto del caso de uso a partir de una implementación global y una tecnología particular de E/S.

- Si interviene una interfaz gráfica para el usuario, el caso de uso real incluirá diagramas de las ventanas en cuestión y una explicación de la interacción con los elementos de la interfaz

Ejemplo TPV: caso de uso real (1)

Caso de uso: **Comprar productos v1**

Actores: Cliente, Cajero (principal)

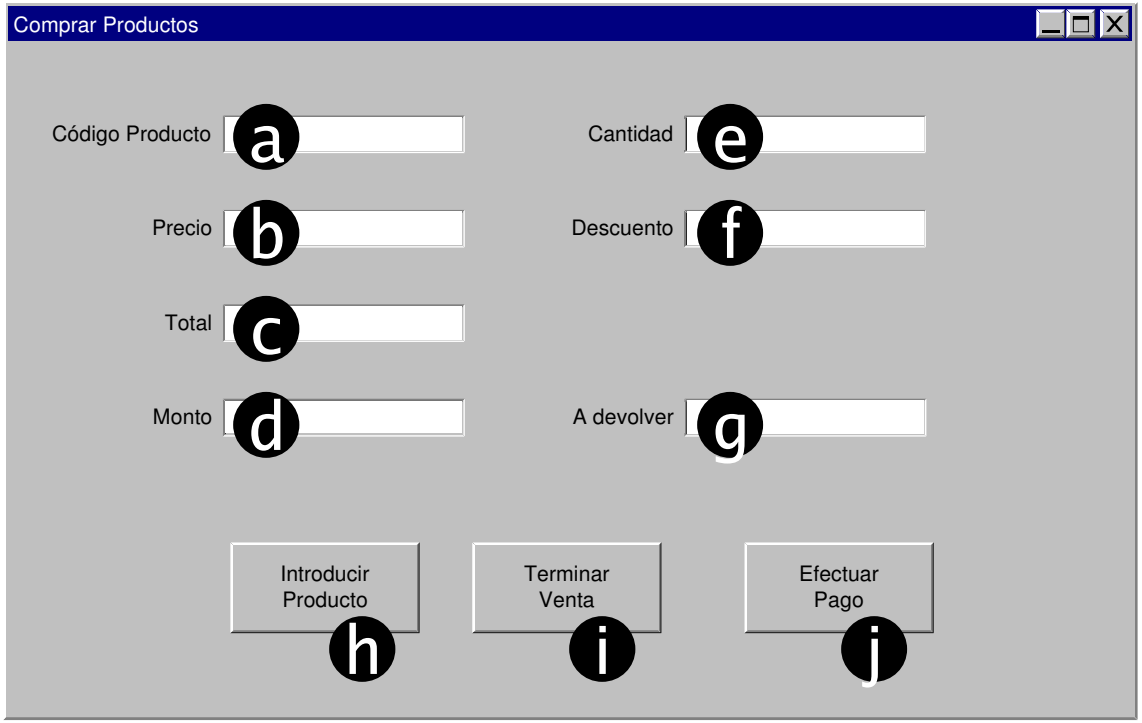
Resumen: Un Cliente llega a la caja registradora con los artículos que desea comprar. El Cajero registra los artículos y recibe un pago. Al terminar la operación, el Cliente se marcha con los productos comprados.

Precondiciones: El Cajero está identificado.

Postcondiciones: Se registra la venta completa, su importe y los impuestos. Se actualiza el inventario.

Referencias: R1.1, R1.2, R1.3, R1.4, R1.5, R1.7

Ejemplo TPV: caso de uso real (2)



Ejemplo TPV: caso de uso completo (2)

Escenario principal (o curso normal de los eventos):

1. **Cliente**: Llega a un TPV con productos que desea comprar.
2. **Cajero**: Comienza una nueva venta.
3. **Cajero**: Teclea el código de producto en **a** de la ventana-cp. Si hay más de un producto, es opcional añadir la cantidad en **e**. Se añade el producto con **h**.
4. **Sistema**: Registra la línea de la venta, y presenta el precio en **b** de la ventana-cp y la suma parcial en **c**.

El Cajero repite los pasos 3 a 4 hasta terminar los artículos del Cliente.

5. **Cajero**: Indica al TPV que se concluyó la captura de productos pulsando **i**.
6. **Sistema**: Calcula y presenta el total con impuestos de la venta en **c**.
7. **Cajero**: Le indica el total de la venta al Cliente.
8. **Cliente**: Efectúa un pago.
9. **Cajero**: Gestiona el pago.
10. **Sistema**: Registra la venta. Genera un recibo.
11. **Cajero**: Da al Cliente el recibo impreso.
12. **Cliente**: Se marcha con los artículos comprados.

Diagrama de paquetes (diseño arquitectónico)

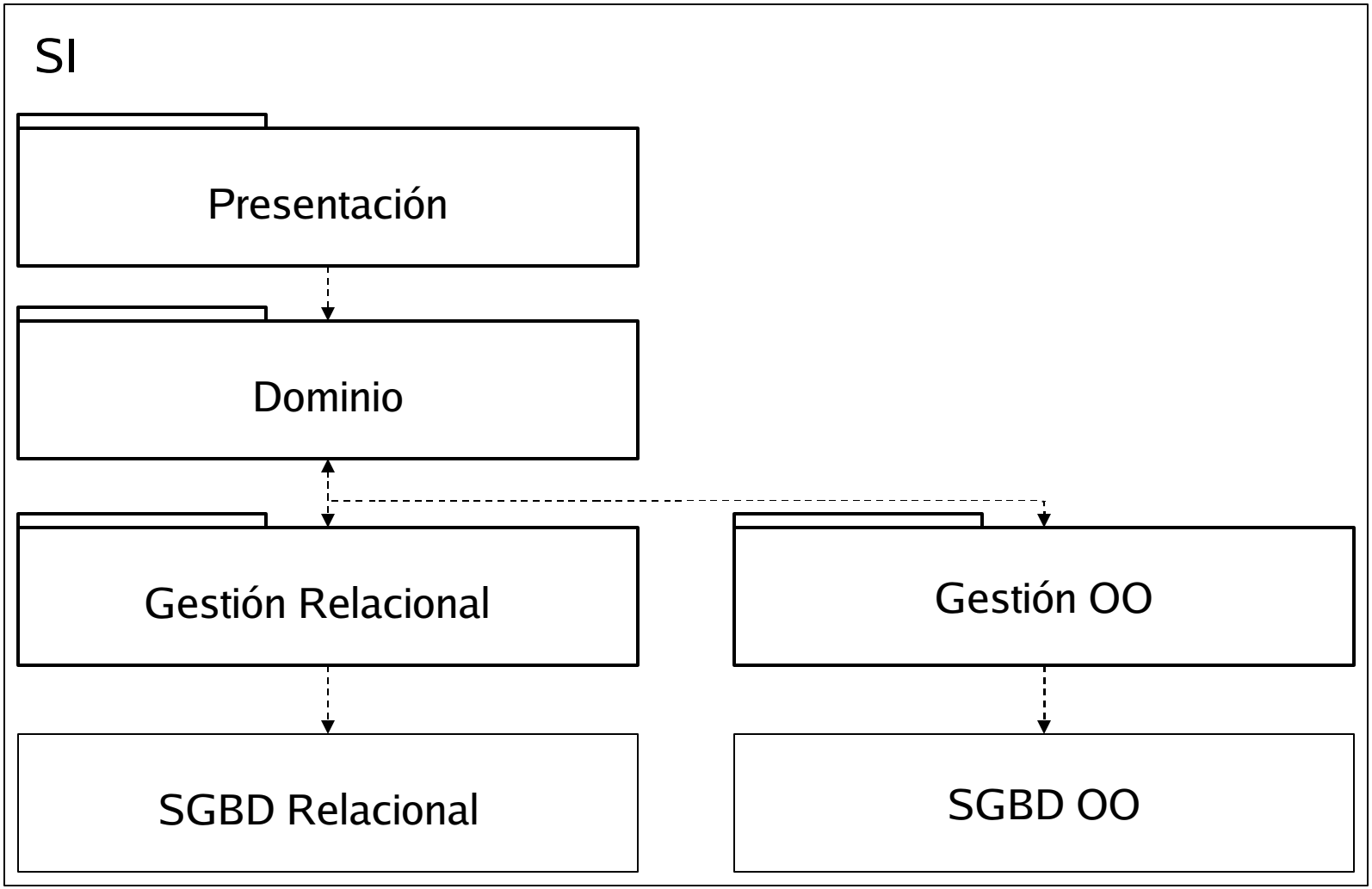


Diagrama de paquetes

- Diagrama de paquetes permite agrupar un grupo de elementos o subsistemas
- Un paquete es un conjunto de cualquier tipo de elementos de un modelo:
  - Clases
  - Casos de uso (por ejemplo: agrupados por actor)
  - Diagramas de interacción
  - ... u otros paquetes (anidados)
- El SI puede verse como un único paquete de alto nivel

Diagrama de clases

- Diagrama de clases: describe gráficamente la especificación software de un SI

- Clase de objetos: describe un conjunto de objetos que comparten atributos, asociaciones, operaciones y métodos
- Objeto: instancia de una clase que encapsula estado y comportamiento, distinguible del resto de objetos
- Atributo: propiedad compartida por los objetos de la clase
- Asociación: relación entre dos o más objetos
- Operación: funciones que pueden ejecutar los objetos
- Método: implementación concreta de una operación dentro de una clase

Diagrama de clases

- Atributos: sintaxis completa

  [visibilidad] nombre [multiplicidad][:tipo][=valor-inicial][{propiedades}]

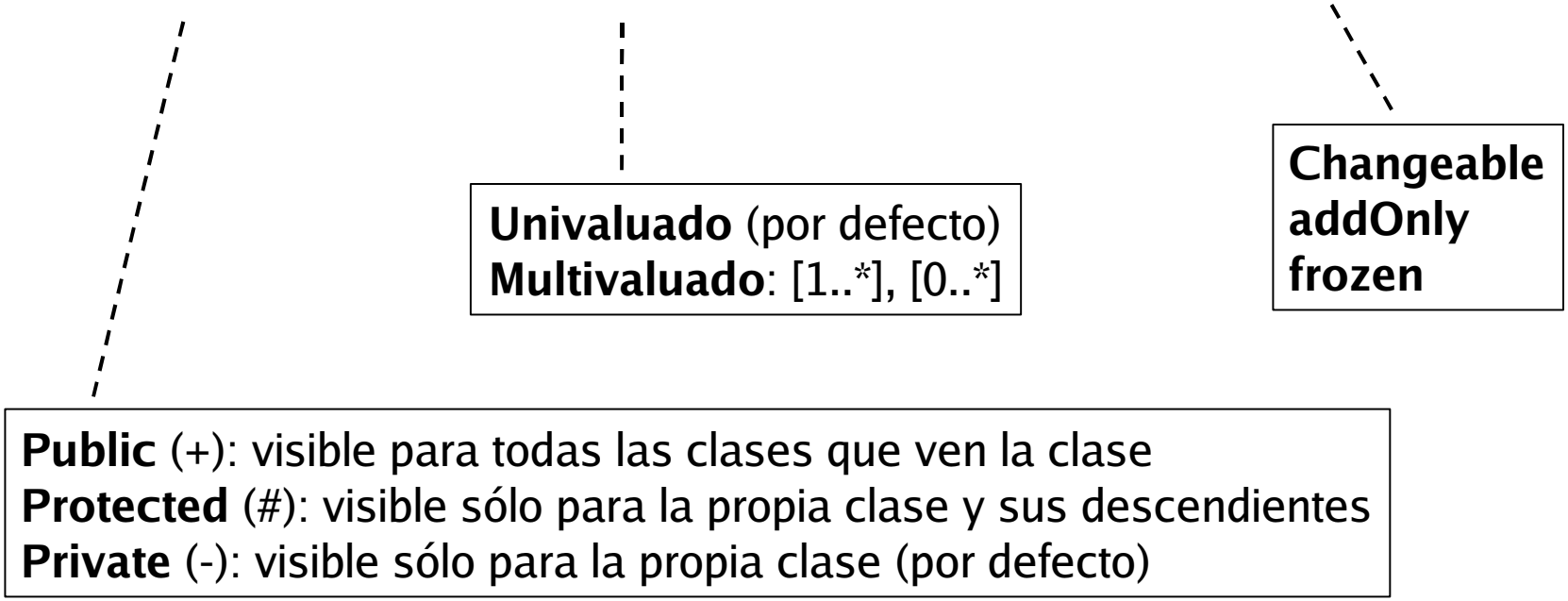


Diagrama de clases

- Atributos de instancia:
  - Propiedad aplicable a todos los objetos de una clase
  - Cada objeto puede tener un valor distinto
- Atributos de clase:
  - Propiedad aplicable a la clase de los objetos
  - Todos los objetos de la clase comparten el mismo valor

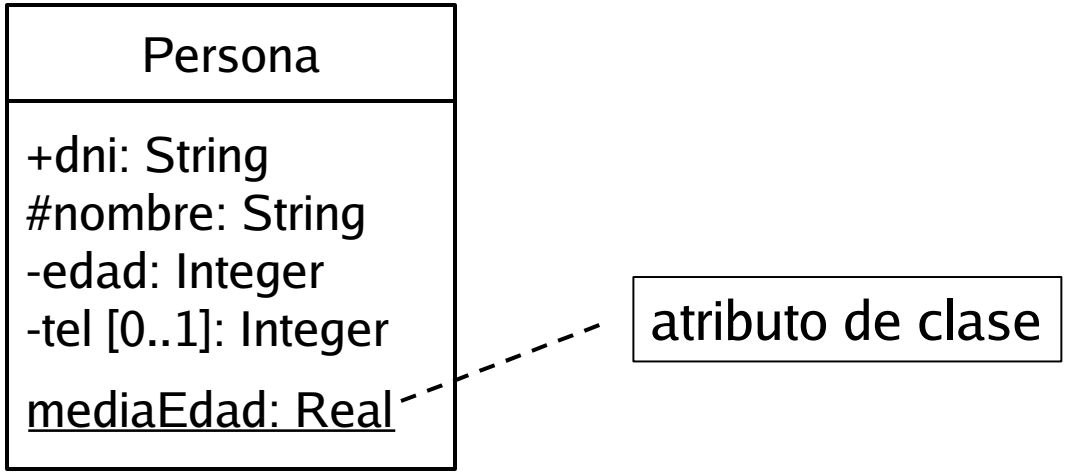


Diagrama de clases

- Operaciones: sintaxis completa

  [visibilidad] nombre [(lista-parámetros)][:tipo-retorno][{propiedades}]

- Visibilidad:
  - **Public** (+): puede ser invocada desde cualquier objeto (por defecto)
  - **Protected** (#): puede ser invocada sólo objetos de la propia clase y descendientes
  - **Private** (-): sólo puede ser invocada por la propia clase
- Parámetros:
  - [dirección] nombre : tipo [valor-por-defecto]
  - dirección : in, out, inout
- Propiedades:
  - Concurrencia: sequential, guarded, concurrent
  - Modificadores: abstract, leaf, root, query, static

Diagrama de clases

- Operaciones de instancia:
  - La operación se invoca sobre objetos individuales
- Operaciones de clase:
  - La operación se invoca sobre la clase. Ejemplo: por defecto, cada clase tiene una operación constructora, que permite dar de alta nuevas instancias de la clase

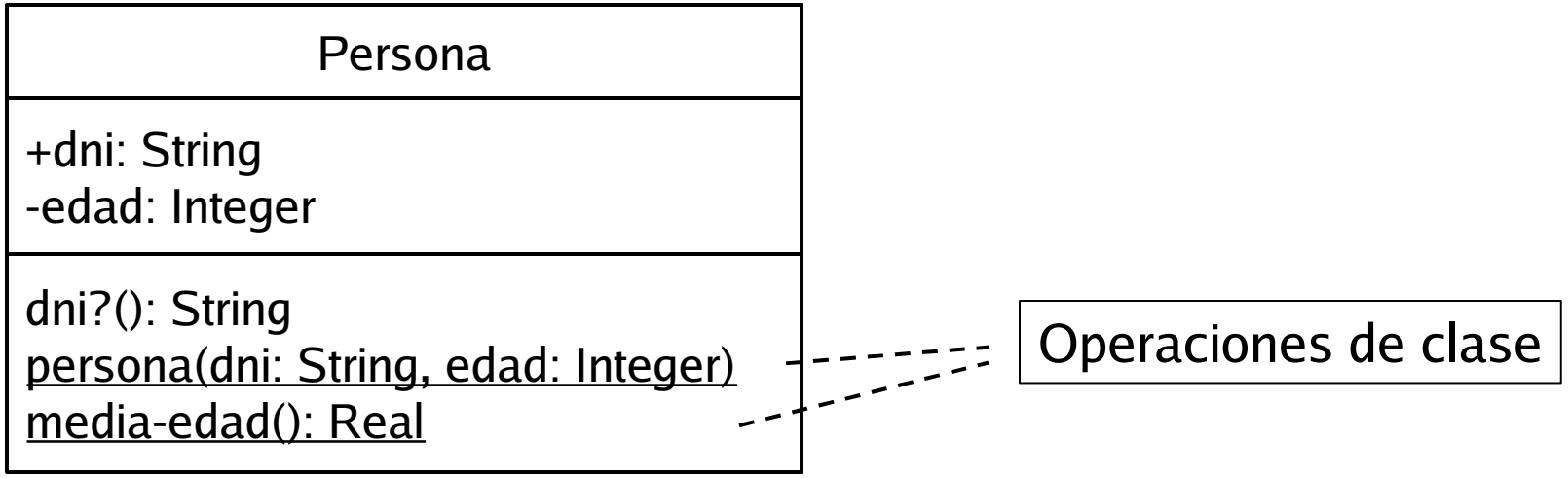
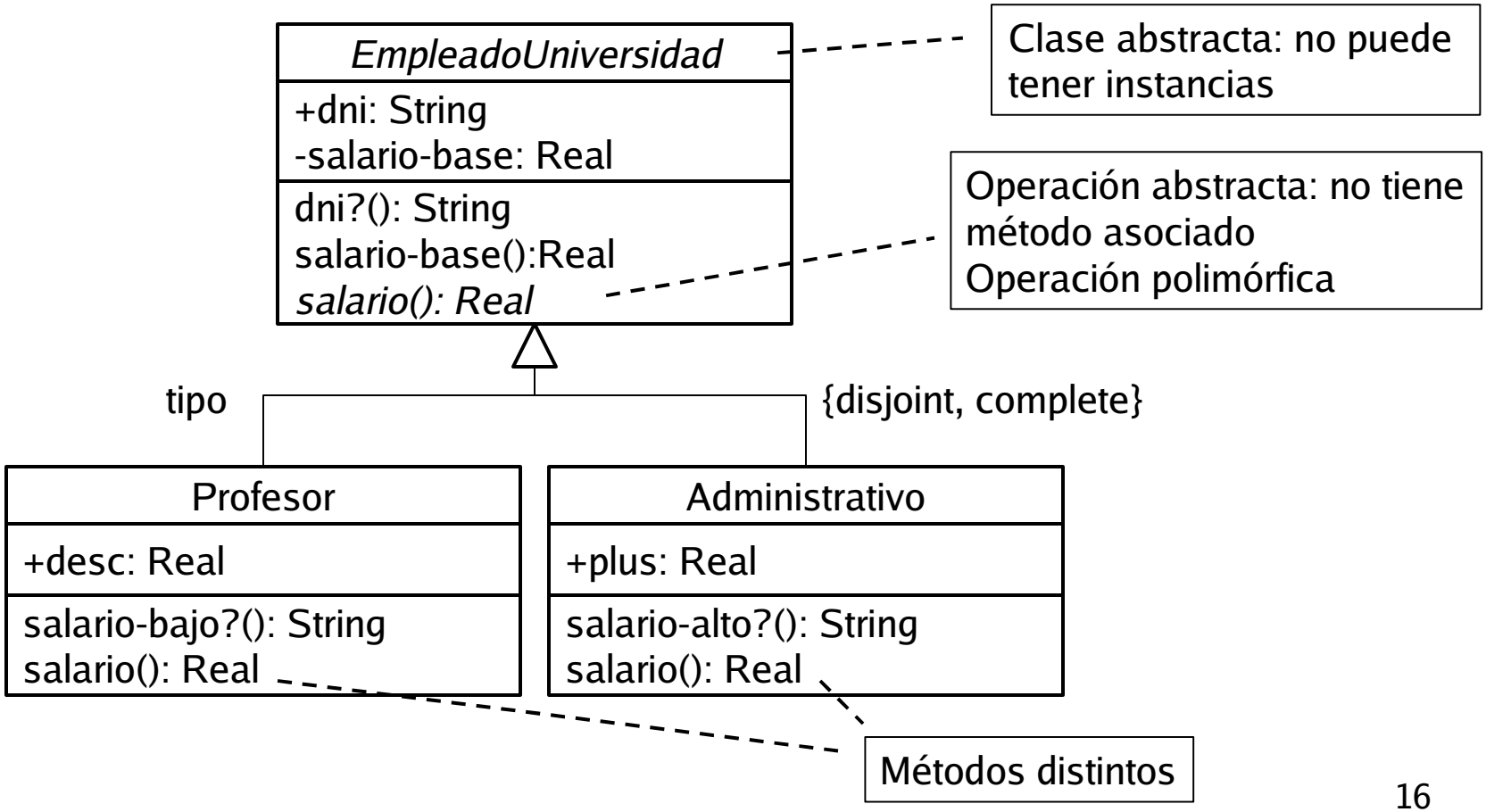


Diagrama de clases

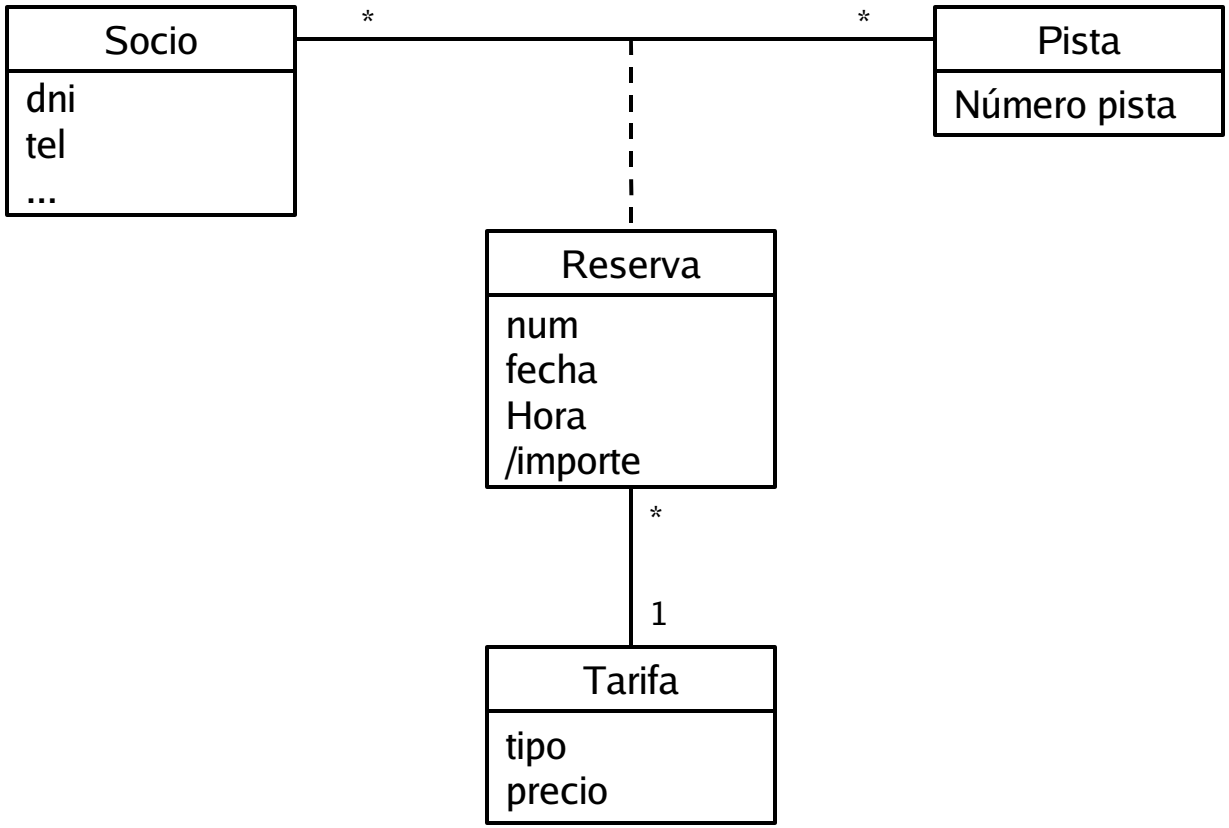
- Herencia: las subclases heredan la estructura y comportamiento de una superclase o varias (herencia múltiple)



Normalización del Modelo Conceptual

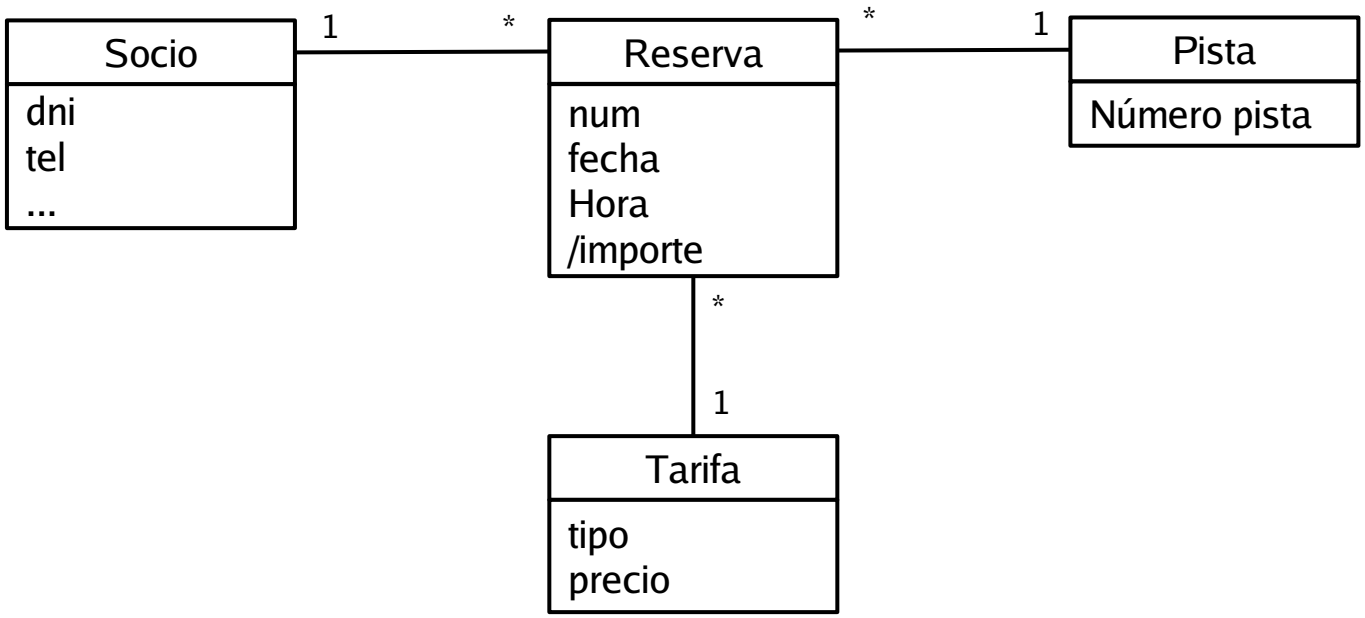
- En el diseño tenemos componentes software
- Limitación tecnológica de la OO: no permite implementar directamente todos los conceptos que hemos usado anteriormente
  - Asociaciones n-arias, con n>2
  - Clases asociativas
  - Información derivada
- Es necesario un proceso de normalización (binarización)
  - Eliminar asociaciones n-arias
  - Eliminar clases asociativas
  - Tratar la información derivada
- Todo ello modifica el diagrama de clases y los contratos

Normalización: Eliminación de una clase asociativa (1)



Normalización: Eliminación de una clase asociativa (2)

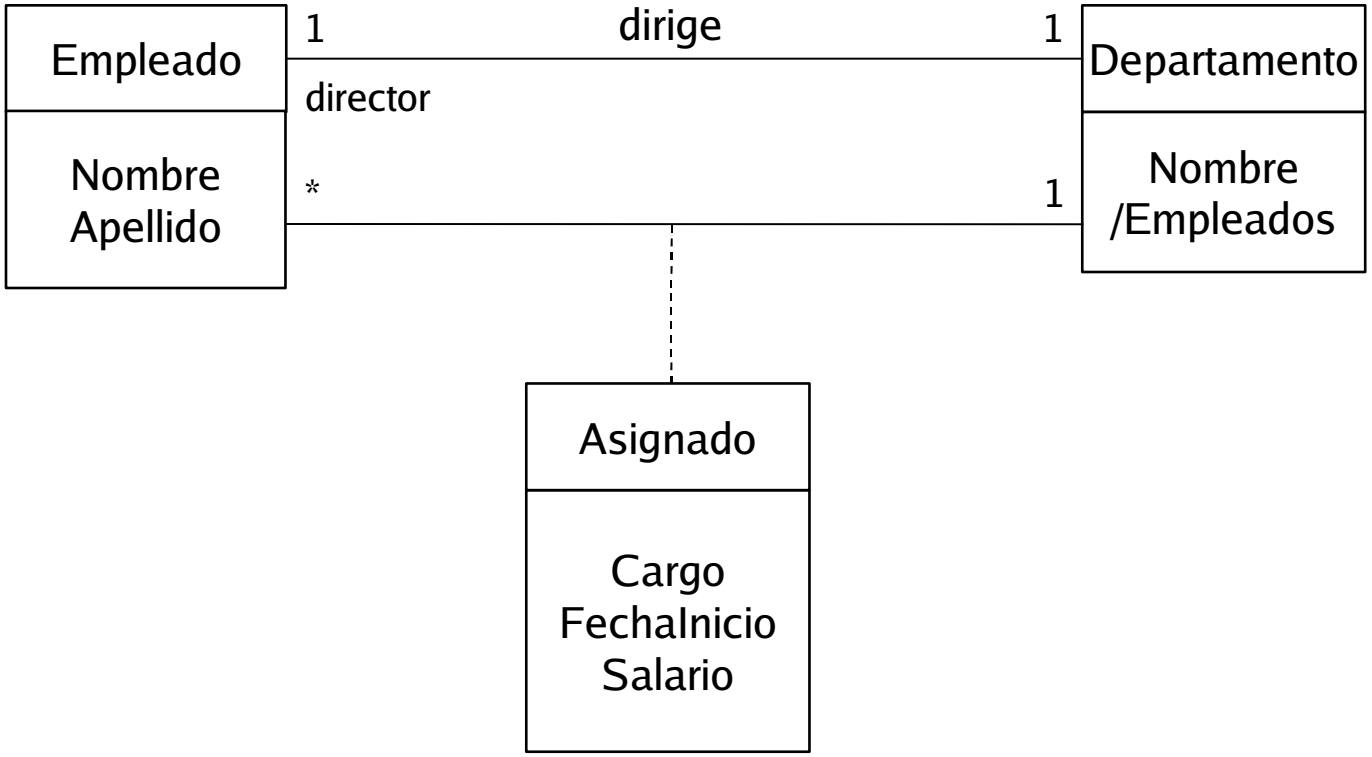
- El resultado debe tener la misma semántica que el Modelo conceptual de partida



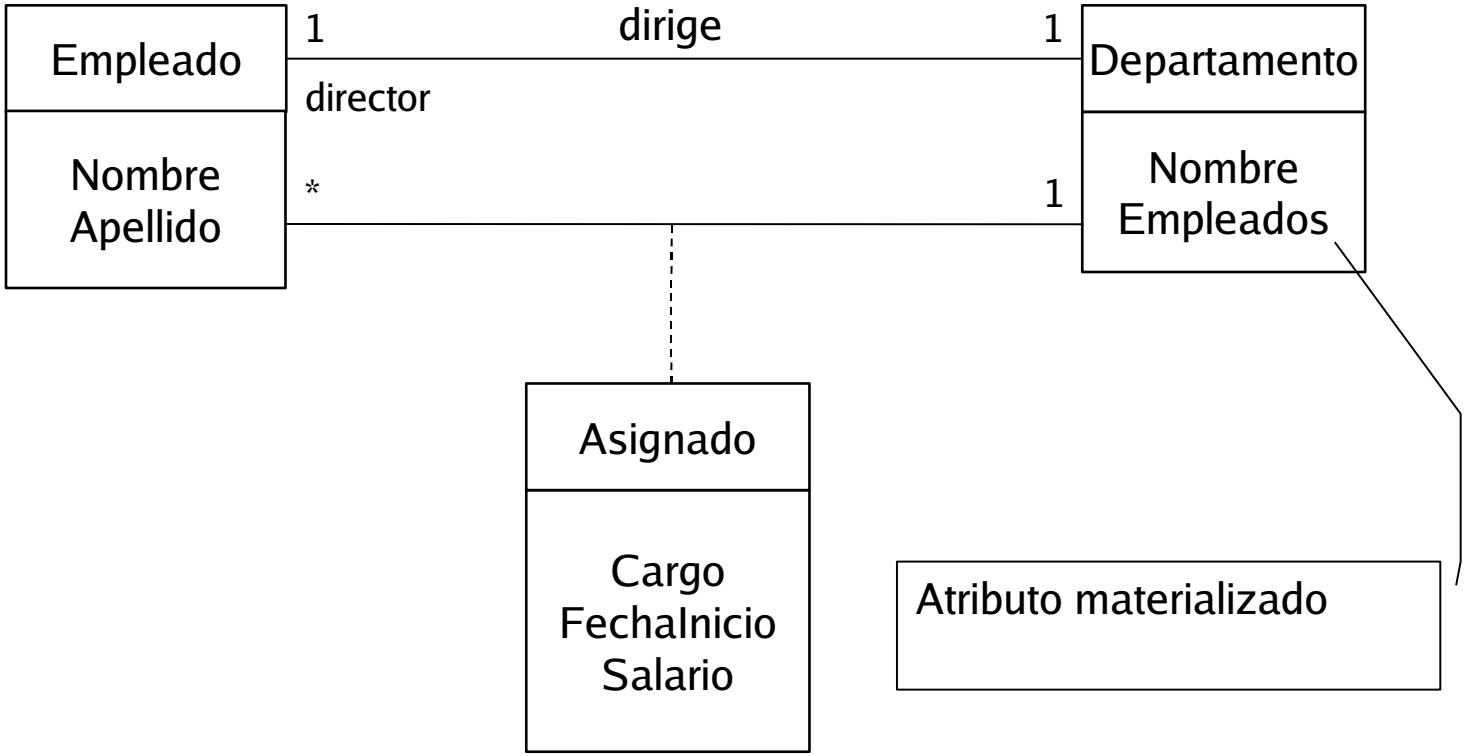
Normalización: Tratamiento de la información derivada (1)

- Los atributos y las asociaciones derivadas pueden ser:
  - Calculados o
  - Materializados
- Si se calculan:
  - Desaparece la información derivada (atributo o asociación)
  - Aparecen nuevos métodos que permiten obtener la información derivada
- Si se materializan:
  - Desaparece la indicación de que la información es derivada (atributo o asociación)
  - Se modifican adecuadamente los contratos de las operaciones ya existentes que pueden obtener la información derivada

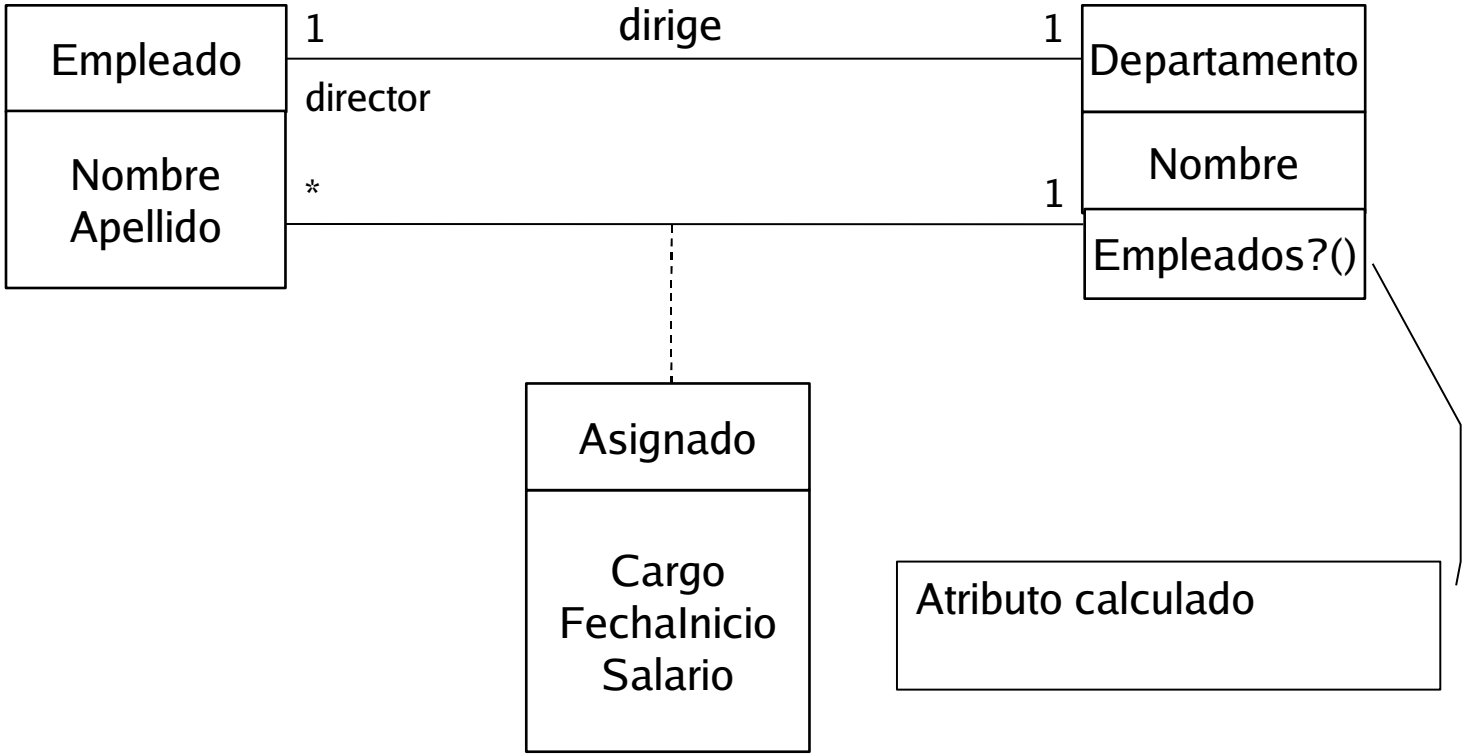
Normalización: Tratamiento de la información derivada (2)



Normalización: Tratamiento de la información derivada (3)



Normalización: Tratamiento de la información derivada (4)

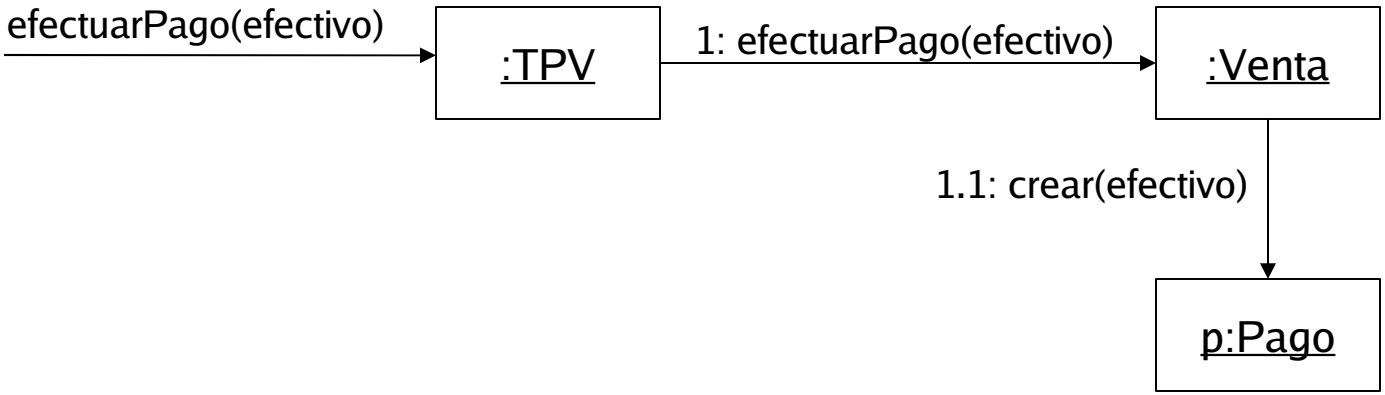


Diagramas de Interacción

- Los diagramas de interacción explican gráficamente cómo los objetos (instancias y clases) interactúan entre sí a través de mensajes
- Diagramas de interacción equivalentes
  - Diagramas de secuencia
  - Diagramas de colaboración
- Para elaborarlos se requiere
  - Modelo conceptual
  - Contratos de las operaciones del sistema
  - Casos de uso reales (o expandidos o esenciales)
- Se elabora un diagrama de interacción para cada operación del CU

Diagramas de Colaboración

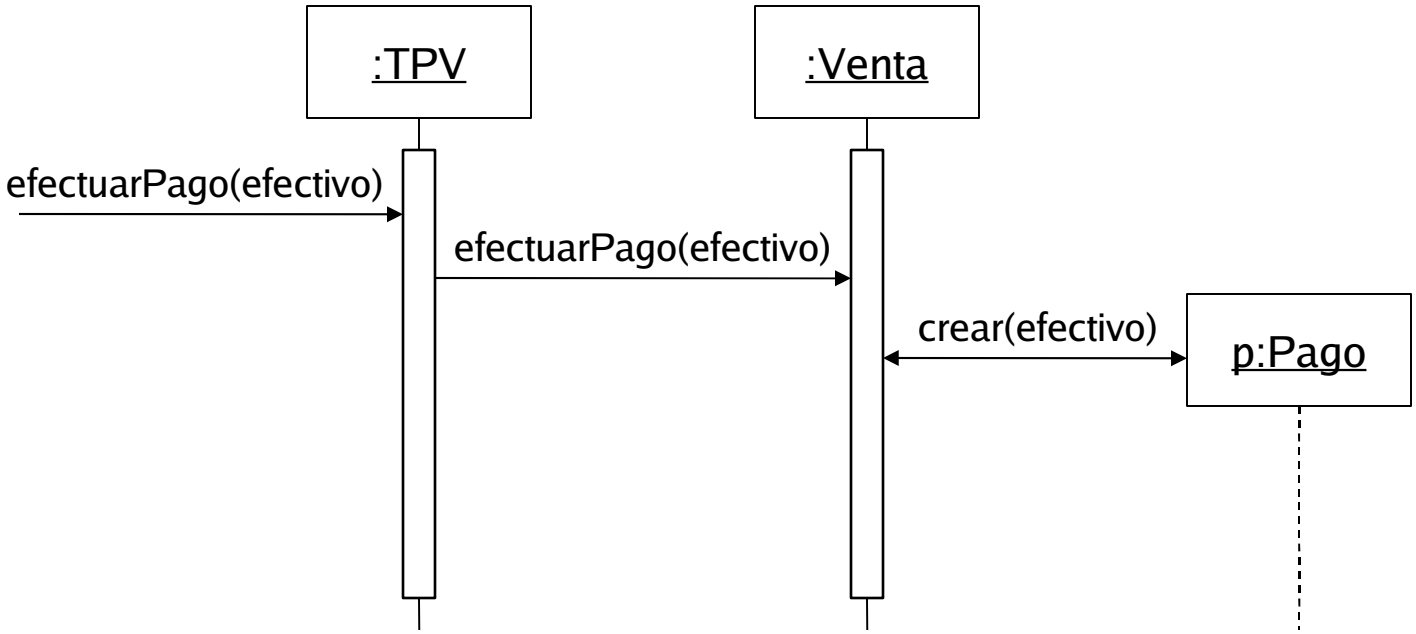
- Ejemplo : efectuarPago



- 1. El mensaje *efectuarPago* se envia a una instancia TPV
- 2. El objeto TPV envía un mensaje *efectuarPago* a la instancia Venta
- 3. El objeto Venta *crea* una instancia *p* de un Pago

Diagramas de Secuencia (1)

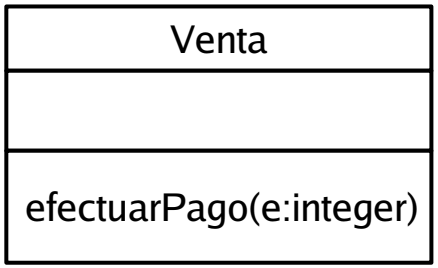
- Ejemplo : efectuarPago



- 1. El mensaje *efectuarPago* se envia a una instancia TPV
- 2. El objeto TPV envía un mensaje *efectuarPago* a la instancia Venta
- 3. El objeto Venta *crea* una instancia *p* de un Pago

Diagramas de Secuencia (2)

- Ejemplo : efectuarPago



- 1. La clase TVP debe proporcionar un método *efectuarPago*
- 2. La clase Venta debe proporcionar otro método *efectuarPago*
- 3. La clase Pago debe poder crearse (método constructor)

Así se diseñan las clases, identificando sus métodos a partir de las responsabilidades definidas en los contratos creados durante el análisis

Diagramas de Secuencia (3)

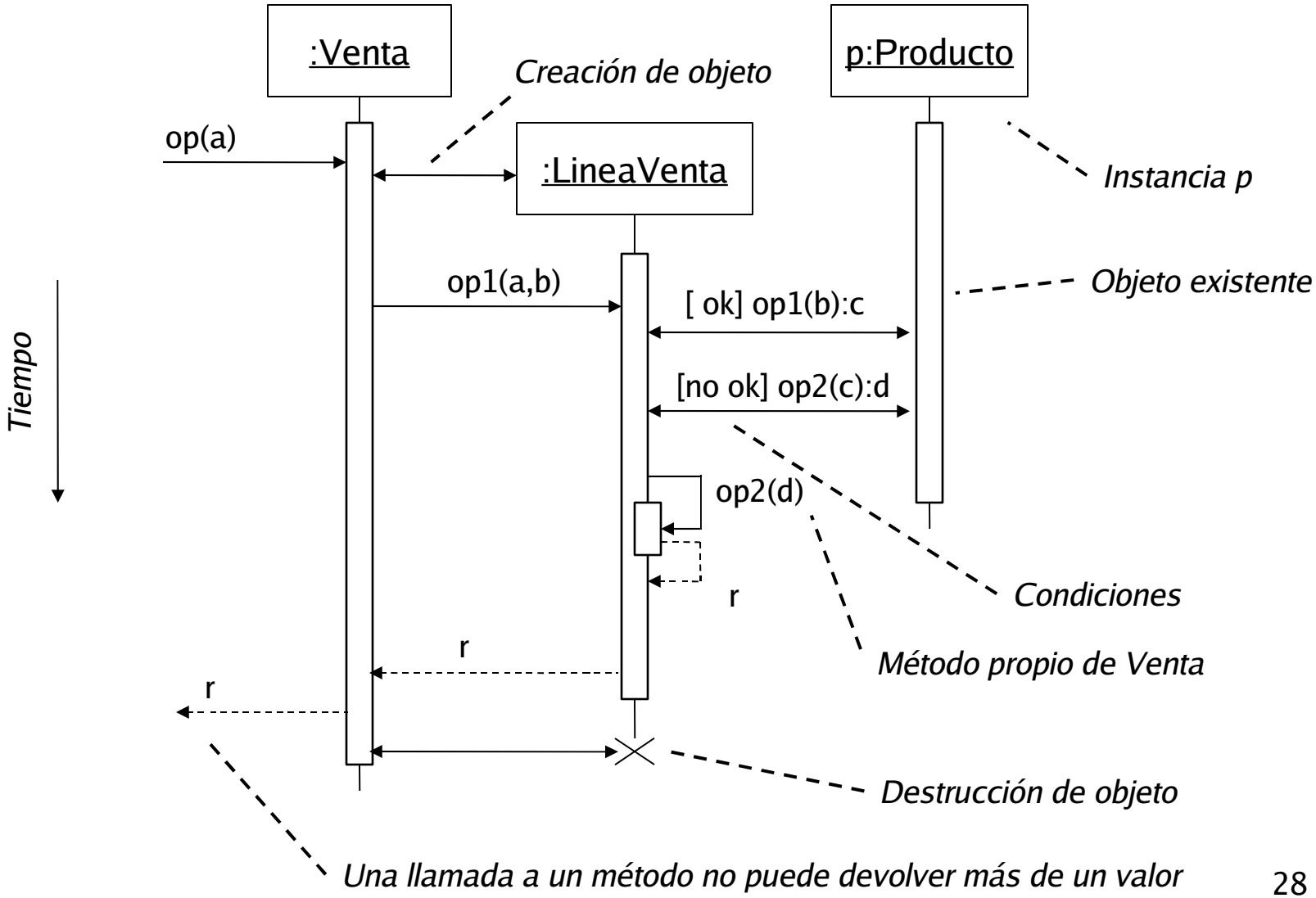
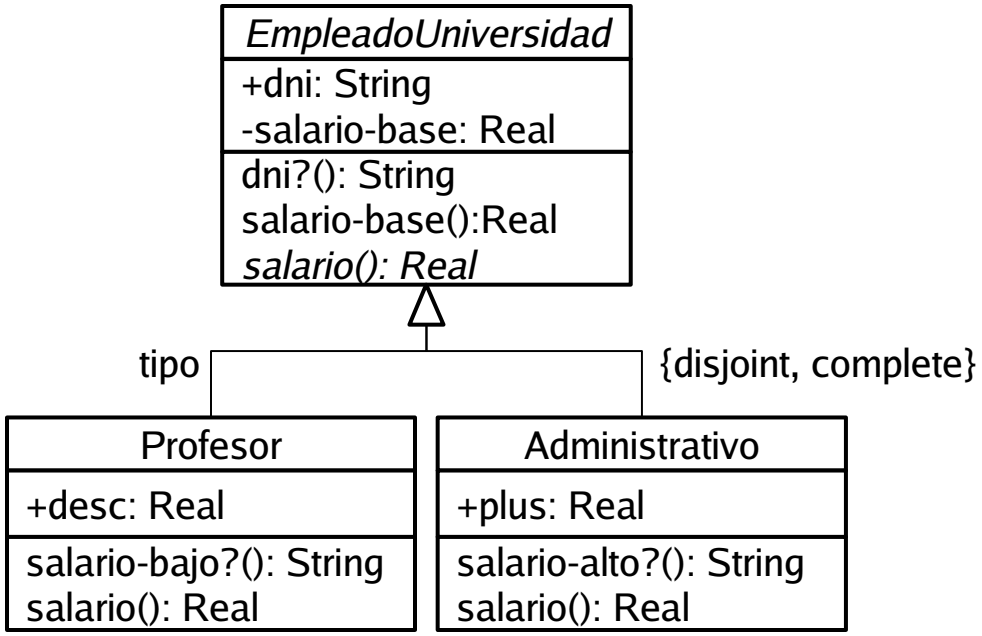
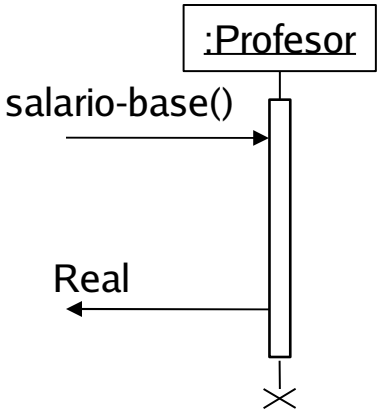
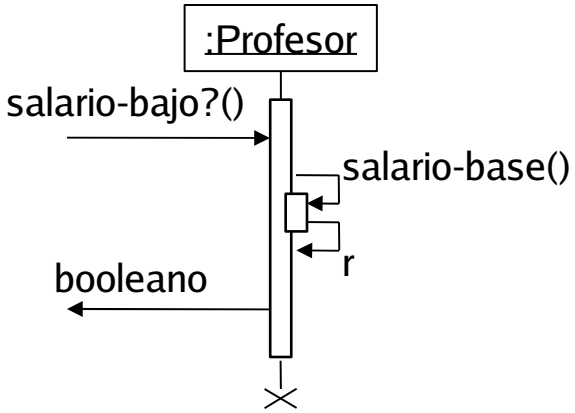


Diagrama de secuencia (4)







Asignación de responsabilidades: patrones GRASP

- La asignación de responsabilidades a objetos consiste en determinar cuáles son las obligaciones (responsabilidades) concretas de los objetos del diagrama de clases que permita dar respuesta a las funcionalidades del sistema
- La asignación de responsabilidades a objetos se realiza al definir y localizar las operaciones de cada clase de objetos
- Las responsabilidades de un objeto consisten en:
  - Saber:
    - Sobre los atributos de los objetos
    - Sobre los objetos asociados
    - Sobre los datos que puedan derivarse o calcularse
  - Hacer
    - Operaciones sobre el propio objeto
    - Coordinar y controlar las actividades de otros objetos
    - Llamar a métodos de otros objetos

Asignación de responsabilidades: patrones GRASP

- Los diagramas de interacción muestran las decisiones referentes a la asignación de responsabilidades entre los objetos (y en el diagrama de clases)
- Los patrones de diseño pueden verse como principios básicos que nos ayudan a tomar decisiones sobre la asignación de responsabilidades
- Un patrón es una descripción de un problema y su solución. Recibe un nombre y puede aplicarse en muchos contextos.
- Los patrones son más fáciles de reusar que el código, porque son menos concretos.
- Los patrones no expresan nuevos principios, todo lo contrario! Intentan encapsular la experiencia previa. Cuanto más trillados y generalizados, mejor!

Asignación de responsabilidades: patrones GRASP

- Los patrones de diseño no son diseños
- Un patrón de diseño nos proporciona una forma de diseñar nuestro SI, no es un diseño por sí mismo
- Dada una responsabilidad, debemos seleccionar aquel patrón (o patrones) que proporcionen una mejor solución, e instanciar el patrón al problema
- El patrón nos proporciona la organización abstracta para un diseño, aún queda mucho trabajo por hacer!

Asignación de responsabilidades: patrones GRASP

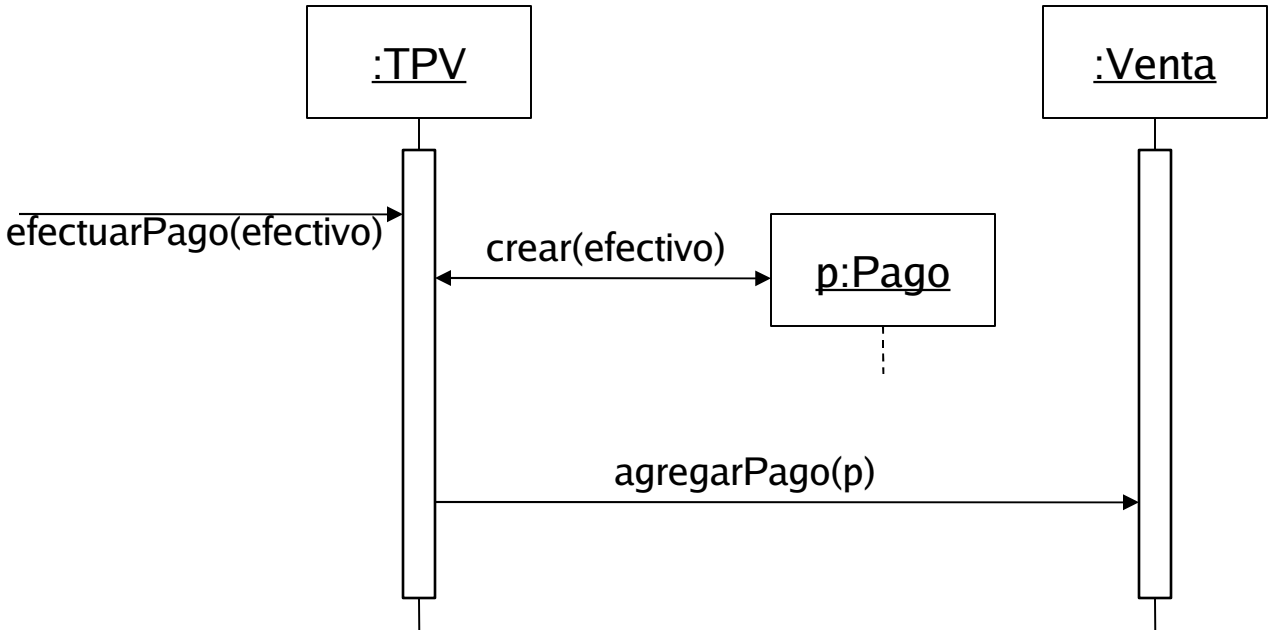
- GRASP: General Responsability Asignment Software Patterns
- Criterios para la asignación de responsabilidades:
  - Bajo acoplamiento
  - Alta Cohesión
- Patrones de asignación de responsabilidades:
  - Experto
  - Creador
  - Controlador
  - ...

Acoplamiento

- El acoplamiento de una clase es una medida del grado de conexión, conocimiento y dependencia respecto las otras clases
- Existe acoplamiento entre las clases A y la B si:
  - A tiene un atributo de tipo B
  - A tiene una asociación a B
  - Una operación de A referencia a un tipo B como parámetro o retorno
  - A es una subclase directa de B
- Bajo acoplamiento para:
  - Cambios en una clase no afecten a otras
  - Facilitar la reutilización
  - Facilitar su comprensión

Análisis del acoplamiento

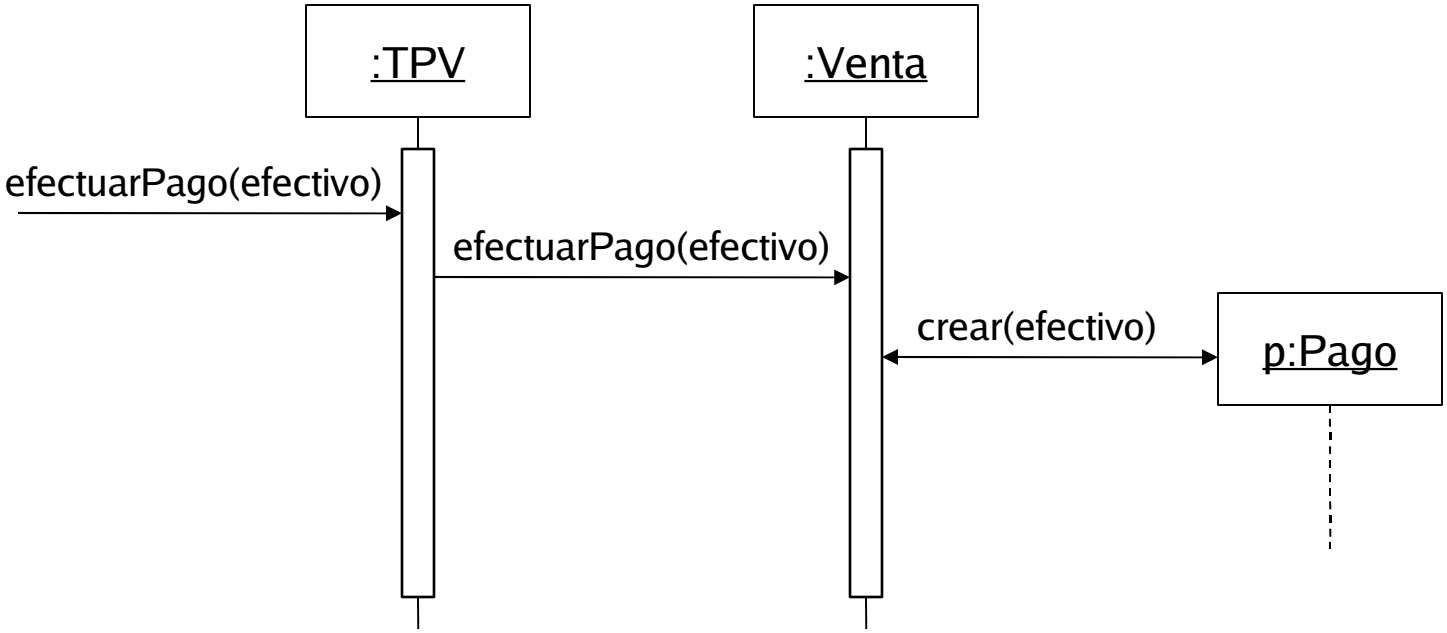
- Responsabilidad: Crear una instancia de *Pago* y asociarla a *Venta*



- *TPV* con *Pago*
- *Venta* con *Pago*
- *TPV* con *Venta*

Análisis del acoplamiento

- Responsabilidad: Crear una instancia de *Pago* y asociarla a *Venta*



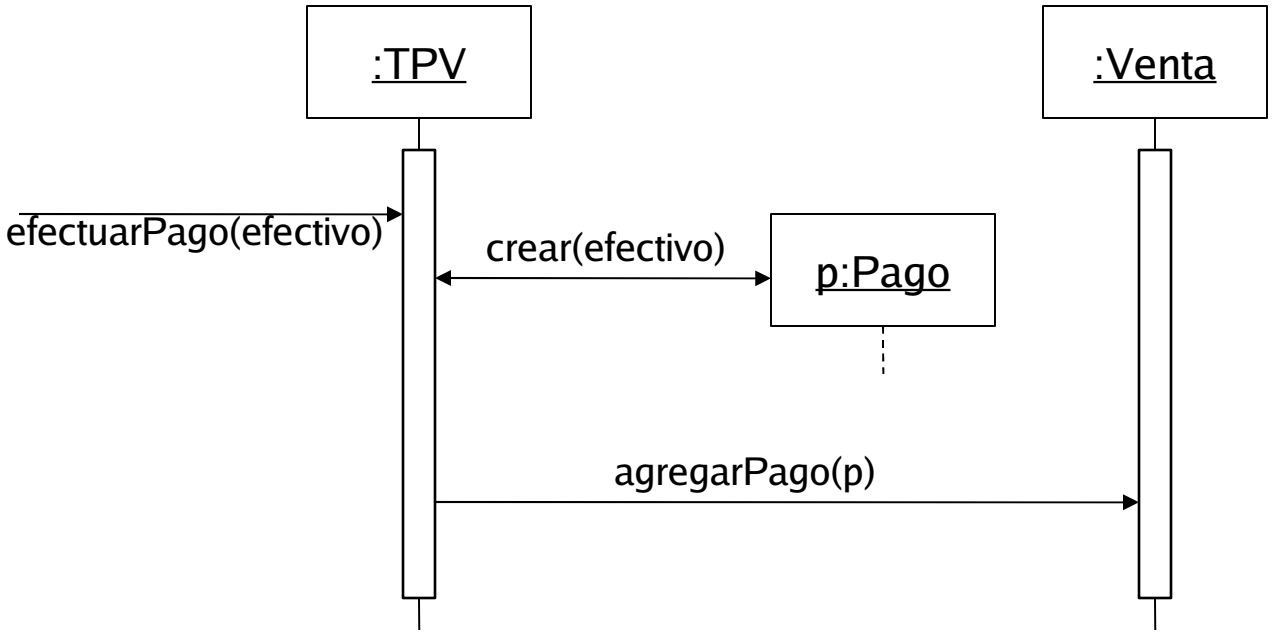
- *TPV* con *Venta*
- *Venta* con *Pago*

Cohesión

- La cohesión funcional de una clase es una medida del grado concentración de las responsabilidades de una clase
- Alta cohesión para:
  - Tener pocas responsabilidades y operaciones, pero muy relacionadas funcionalmente
  - Colaborar (delega) con otras clases
  - Facilitar la comprensión, mantenimiento, reutilización
- Una clase con baja cohesión:
  - Tiene muchas responsabilidades y operaciones, y muy poco relacionadas funcionalmente

Análisis de la cohesión

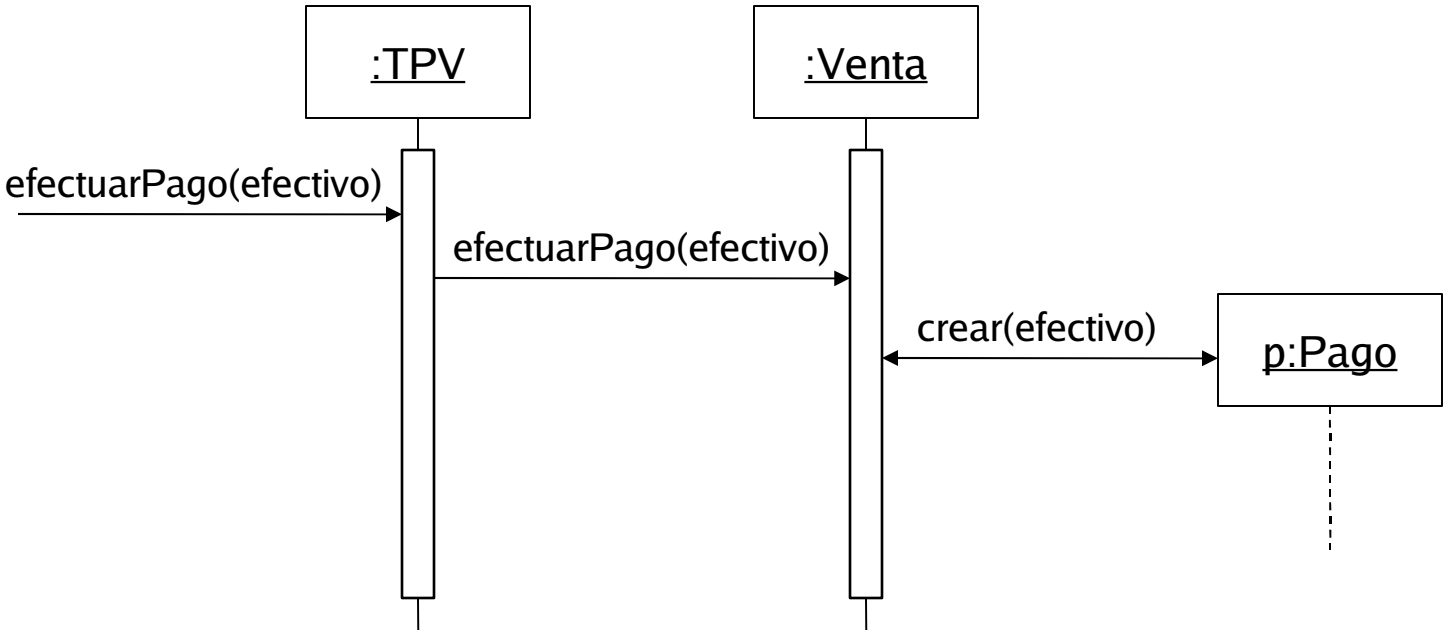
- Responsabilidad: Crear una instancia de *Pago* y asociarla a *Venta*



- *TPV* puede acabar asumiendo demasiadas responsabilidades, perdiendo su cohesión

Análisis de la cohesión

- Responsabilidad: Crear una instancia de *Pago* y asociarla a *Venta*



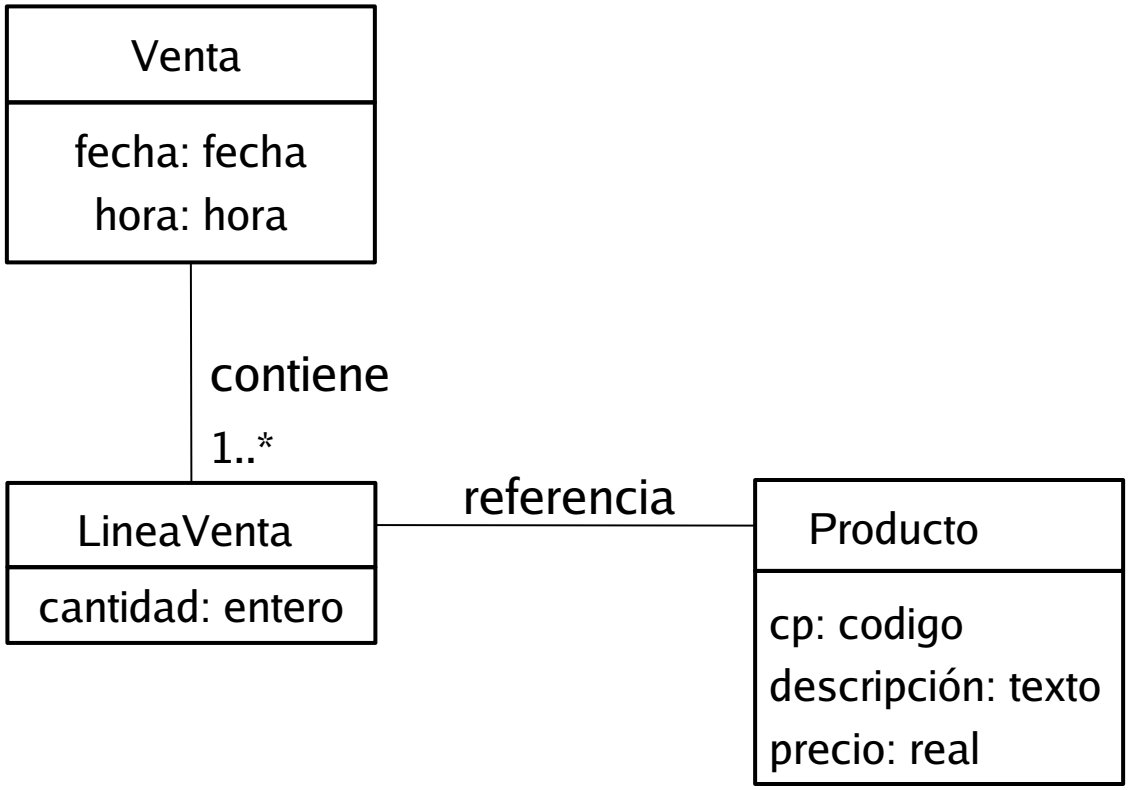
- Se delega en *Venta* la responsabilidad de crear el *Pago*, proporcionando mayor cohesión a *TPV*

Patrón Experto

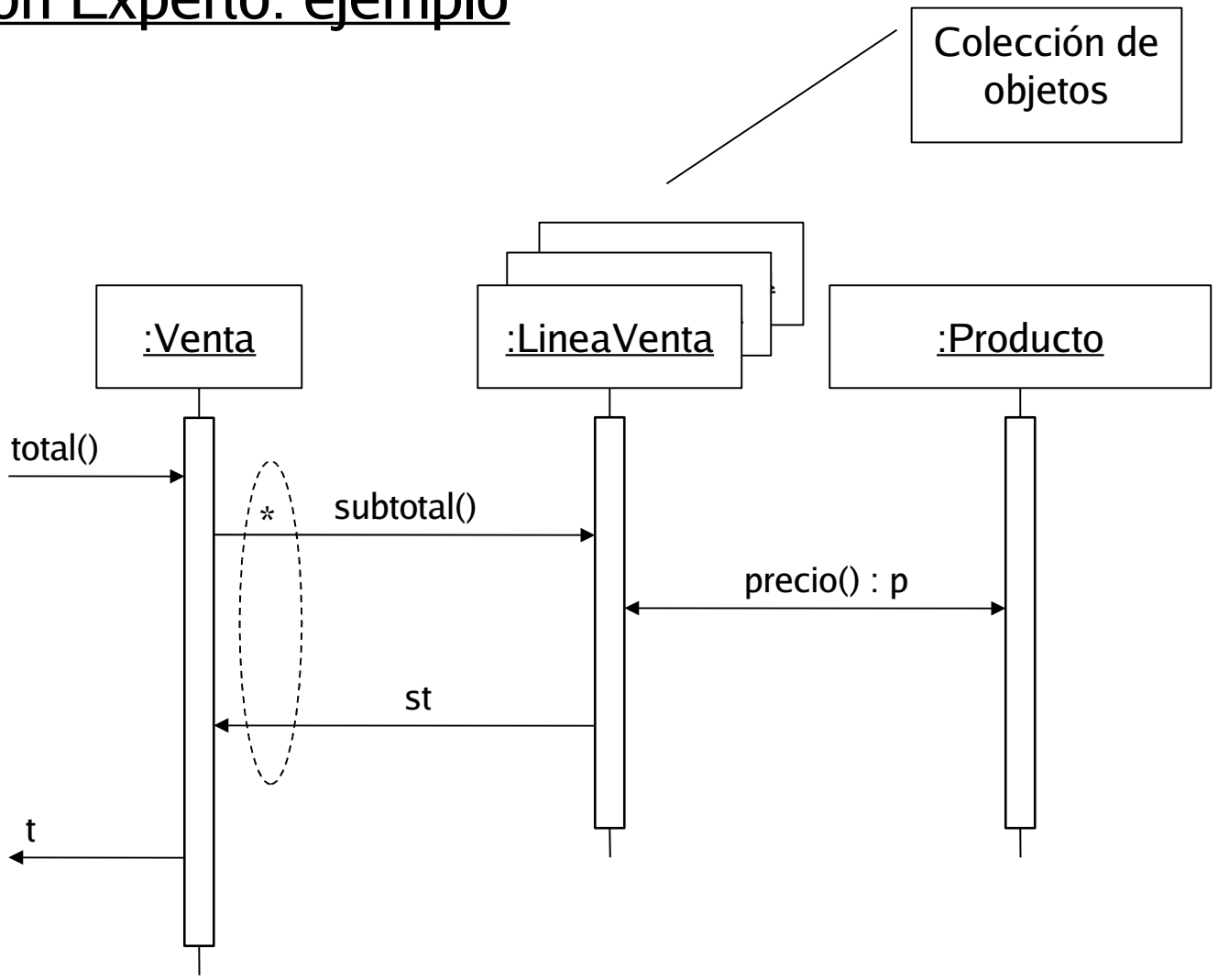
- Contexto:
  - Asignación de responsabilidades a objetos
- Problema:
  - Decidir a qué clase asignar una responsabilidad concreta
- Solución:
  - Asignarla a la clase que tiene la información necesaria para realizarla
- Consideraciones:
  - No siempre existe un único experto: deben colaborar
  - Requiere tener claramente definidas las responsabilidades que se quieren asignar (postcondiciones de las operaciones)
- Beneficios:
  - Responsabilidades distribuidas: alta cohesión
  - Se mantiene el encapsulamiento: bajo acoplamiento

Patrón Experto: ejemplo

- Responsabilidad: Obtener el total de la venta.
- Esquema conceptual

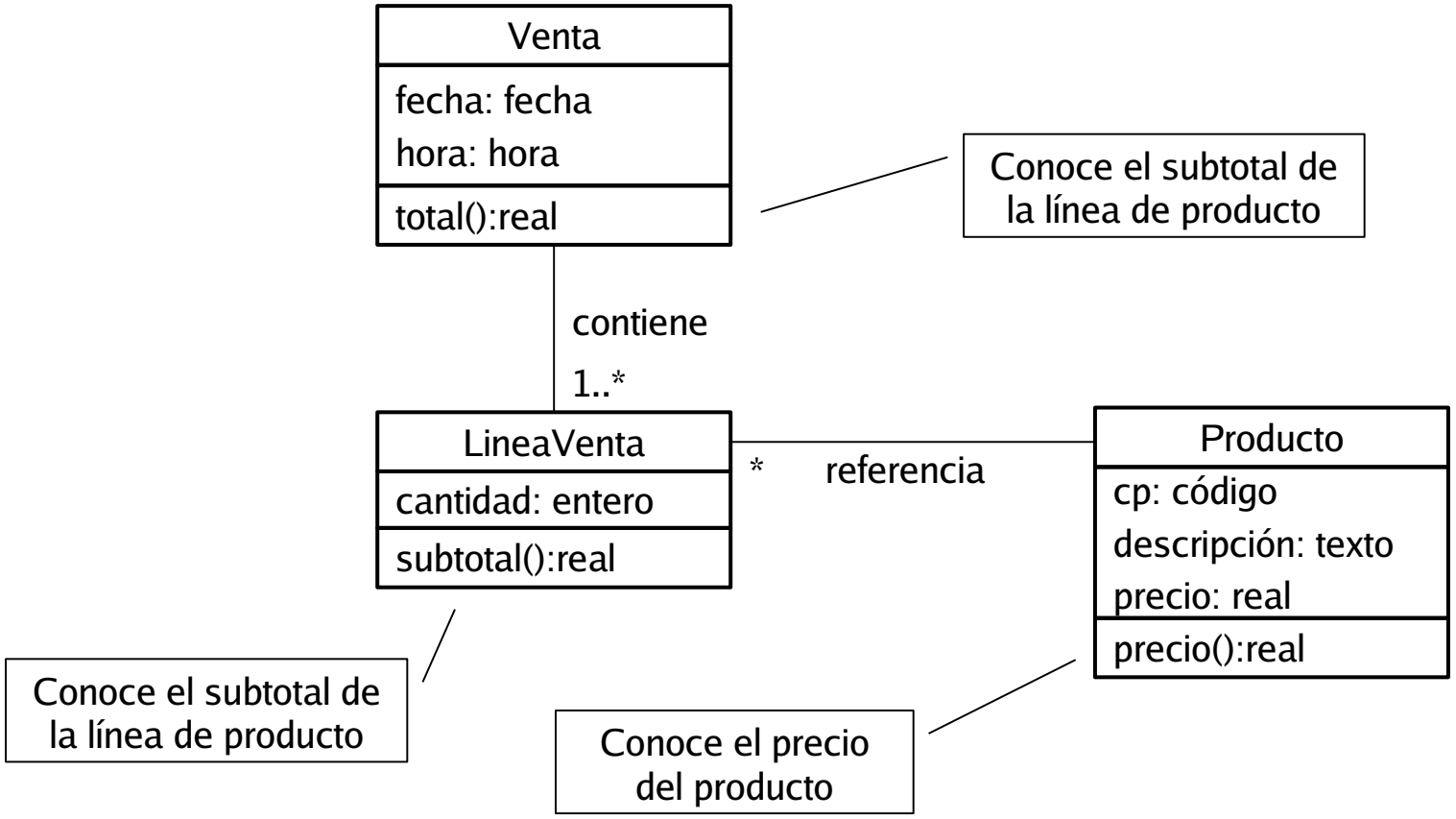


Patrón Experto: ejemplo



Patrón Experto: ejemplo

- Responsabilidad: Obtener el total de la venta.
- Vista parcial del esquema conceptual



Patrón Experto

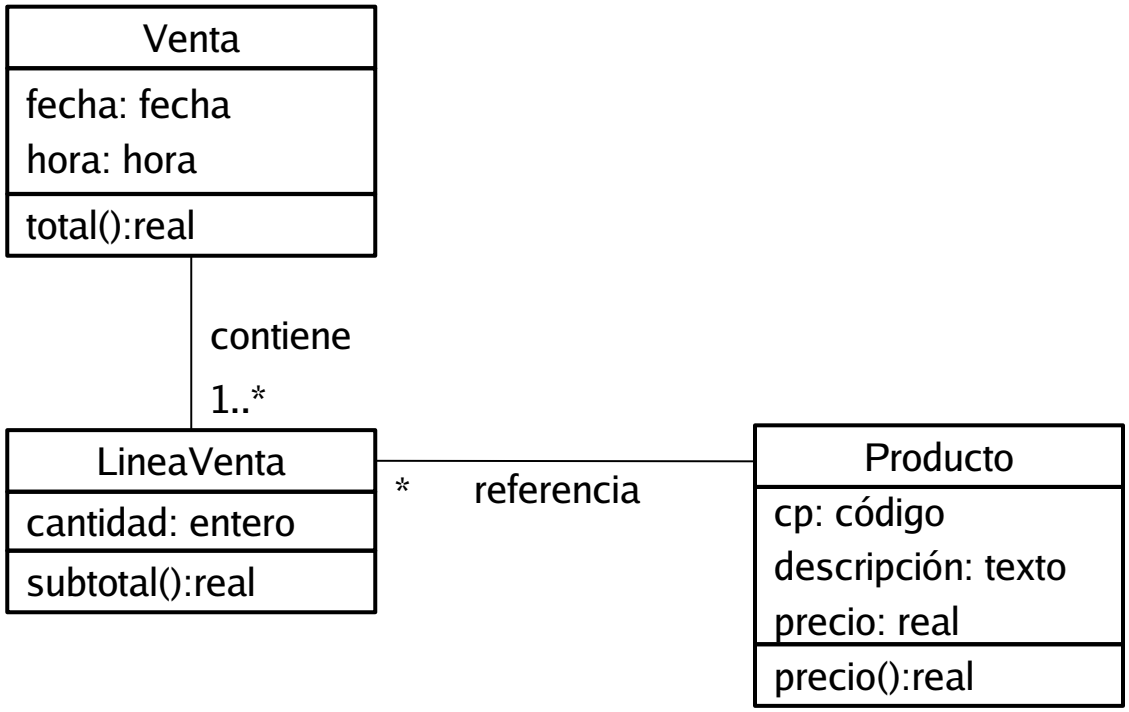
- Sin duda es el patrón más usado
- Principio básico que suele utilizarse en el diseño OO
- Expresa simplemente la “intuición” de que los objetos hacen cosas relacionadas con la información que poseen
- El cumplimiento de una responsabilidad requiere a menudo información distribuida en varias clases de objetos
- Beneficios:
  - Se conserva el encapsulamiento (los objetos se valen de su propia información): bajo acoplamiento
  - El comportamiento se distribuye entre las clases, alentando definiciones de clase “sencillas” que son más fáciles de comprender y mantener: alta cohesión

Patrón Creador

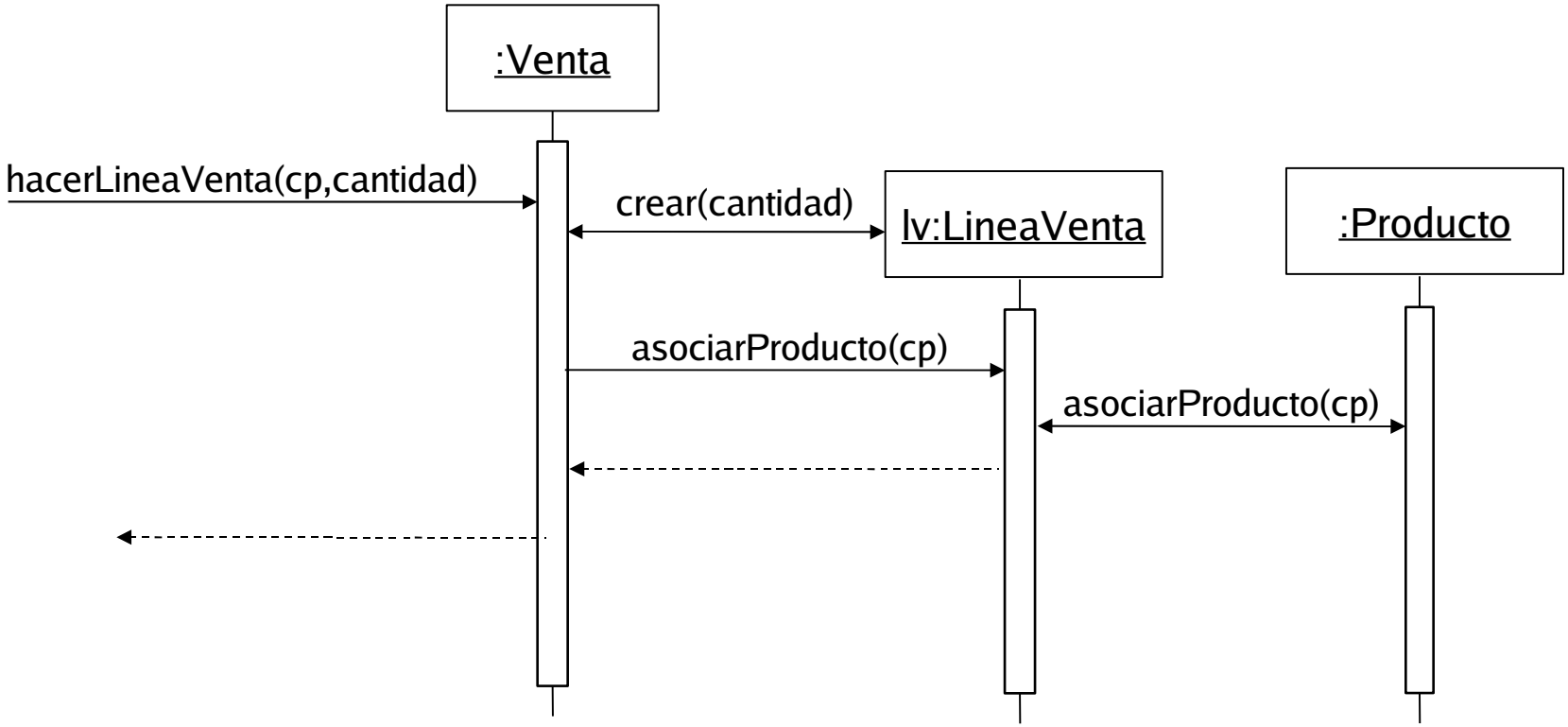
- Contexto:
  - Asignación de responsabilidades a objetos
- Problema:
  - Decidir quién debe ser responsable de crear una instancia concreta de una clase
- Solución:
  - Asignar a una clase B la responsabilidad de crear una instancia de una clase A si cumple una de las siguientes condiciones
    - B es un agregado de objetos de A
    - B contiene objetos de A
    - B usa a menudo objetos de A
    - B tiene la información necesaria para inicializar un objeto de A (B tiene los valores para construir A)
- Beneficios
  - No incrementamos el acoplamiento

Patrón Creador

- Responsabilidad: Crear una instancia de LíneaVenta.
- Vista parcial del esquema conceptual

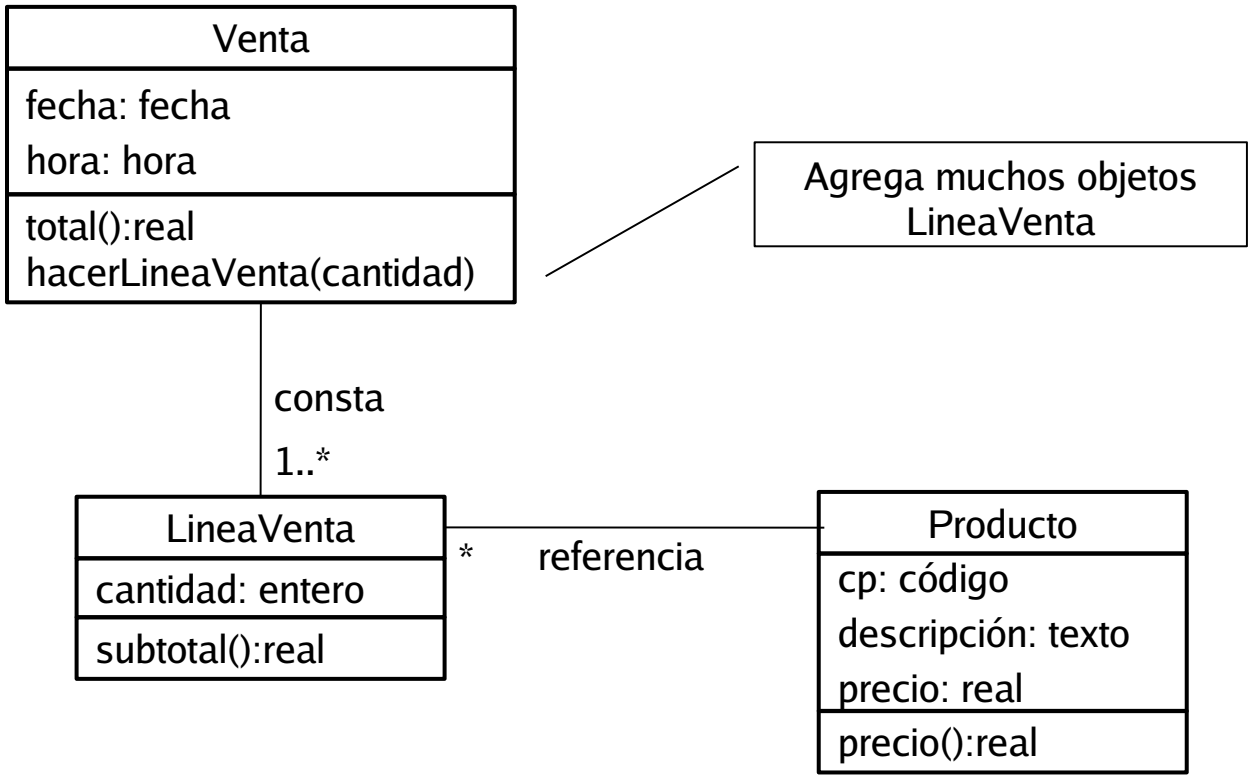


Patrón Creador



Patrón Creador

- Responsabilidad: Crear una instancia de LíneaVenta



Patrón Creador

- Patrón muy usado puesto que guía la asignación de responsabilidades relacionadas con el la creación de objetos

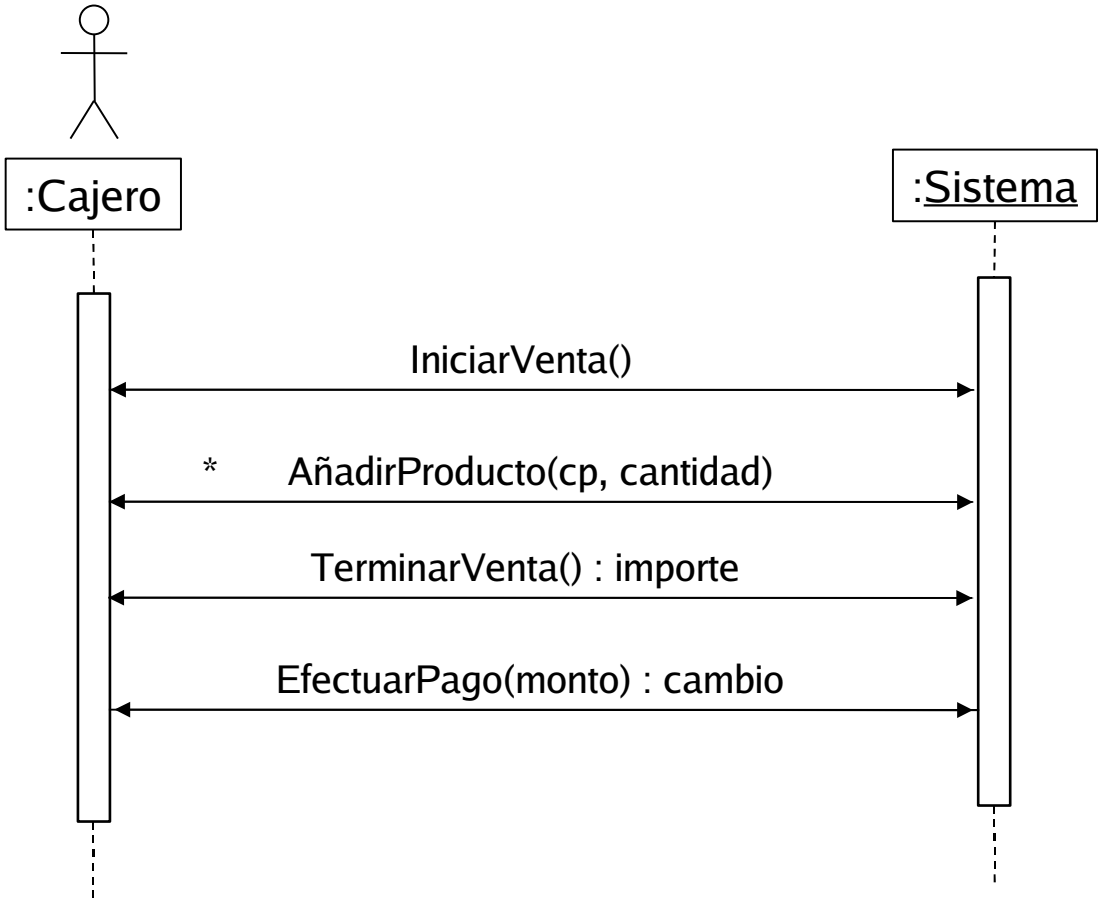
- Bajo acoplamiento: como la clase *creada* tiende a ser visible por la clase *creador*, es probable que el acoplamiento global no aumente

Patrón Controlador

- Contexto:
  - Los SI reciben eventos externos
  - Una vez interceptado un evento en la Capa de Presentación, algún objeto de la Capa de Dominio debe recibir este evento y procesarlo
- Problema:
  - Decidir quién debe ser responsable de tratar una evento externo
- Solución:
  - Asignar la responsabilidad a un controlador:
    - Fachada (sistema): una clase que representa el SI
    - Fachada (empresa): una clase que representa todo el dominio, empresa
    - Rol (actor): una clase que representa un rol activo en el evento
    - Caso de uso (gestor): una clase “artificial” para gestionar a todos los eventos de un caso de uso
    - Transacción (evento): una clase “artificial” que representa el evento externo

Ejemplo: Caso de uso Comprar productos

- Responsabilidad: Quién debe controlar estos eventos?



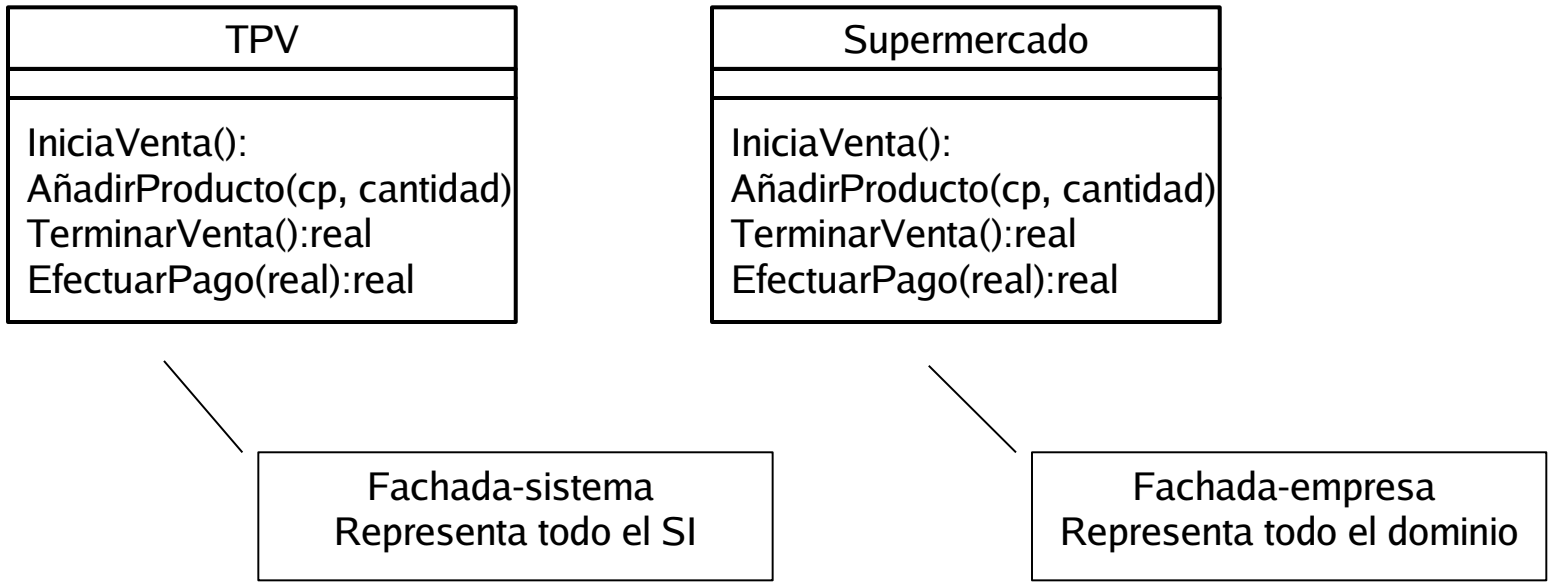
Ejemplo: Caso de uso Comprar productos

- Responsabilidad: Quién debe controlar estos eventos?

TPV	Producto	Supermercado	Venta fecha: fecha Hora: hora
LineaVenta cantidad: entero	Cajero	Cliente	Director
Pago importe: cantidad	Catálogo	EspProducto Descripción: texto Precio: cantidad	

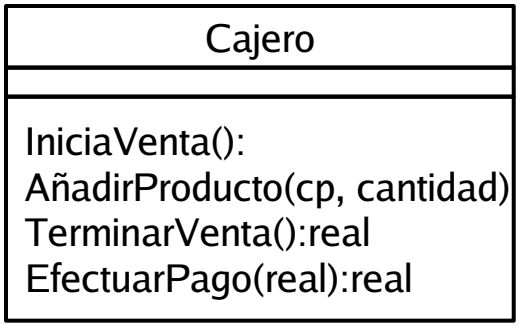
Controlador Fachada (sistema, empresa)

- Todos los eventos externos son procesados por un solo objeto
- Controladores “saturados” si hay muchos eventos externos



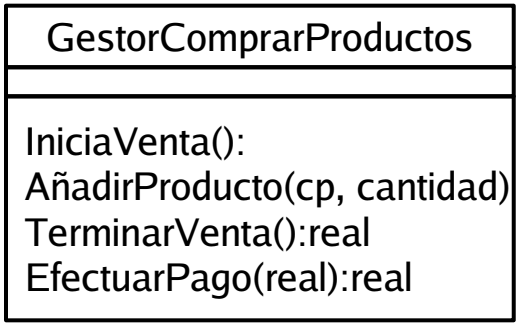
Controlador Rol (actor)

- Hay un controlador por actor
- Adecuado para encapsular y controlar las acciones de un actor
- Inadecuado para los eventos que puede ser generados por varios actores
- Pueden conducir a diseños deficientes (baja cohesión y alto acoplamiento) sino se delega adecuadamente



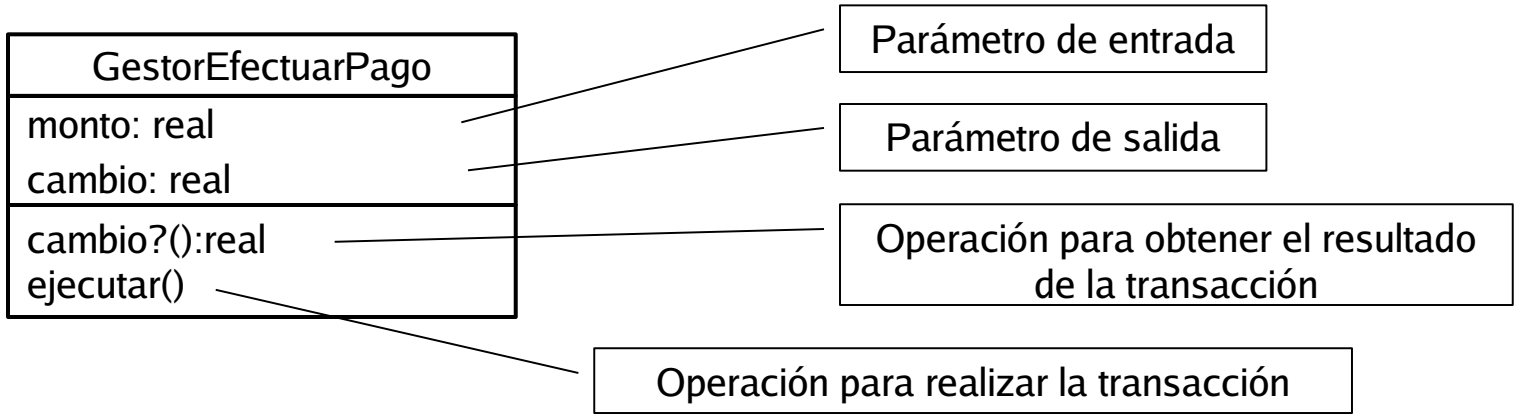
Controlador Caso de Uso (gestor)

- Hay tantos controladores como casos de uso
- Puede tener atributos definidos para controlar el caso de uso
- Adecuado cuando las otras opciones generan diseños deficientes: baja cohesión y alto acoplamiento
- Inadecuado para los eventos que puede ser generados por varios casos de uso



Controlador Transacción (evento)

- Basado en el patrón de diseño “command”
- Hay tantos controladores como eventos externos del SI
- Los parámetros de la operación asociadas al evento se corresponden con los atributos del objeto transacción
- La ejecución de la operación se realiza con la invocación a ejecutar(), y se definen las operaciones específicas para consultar los resultados
- Se acostumbra a destruir el objeto tras recuperar el resultado



Controlador Transacción (evento)

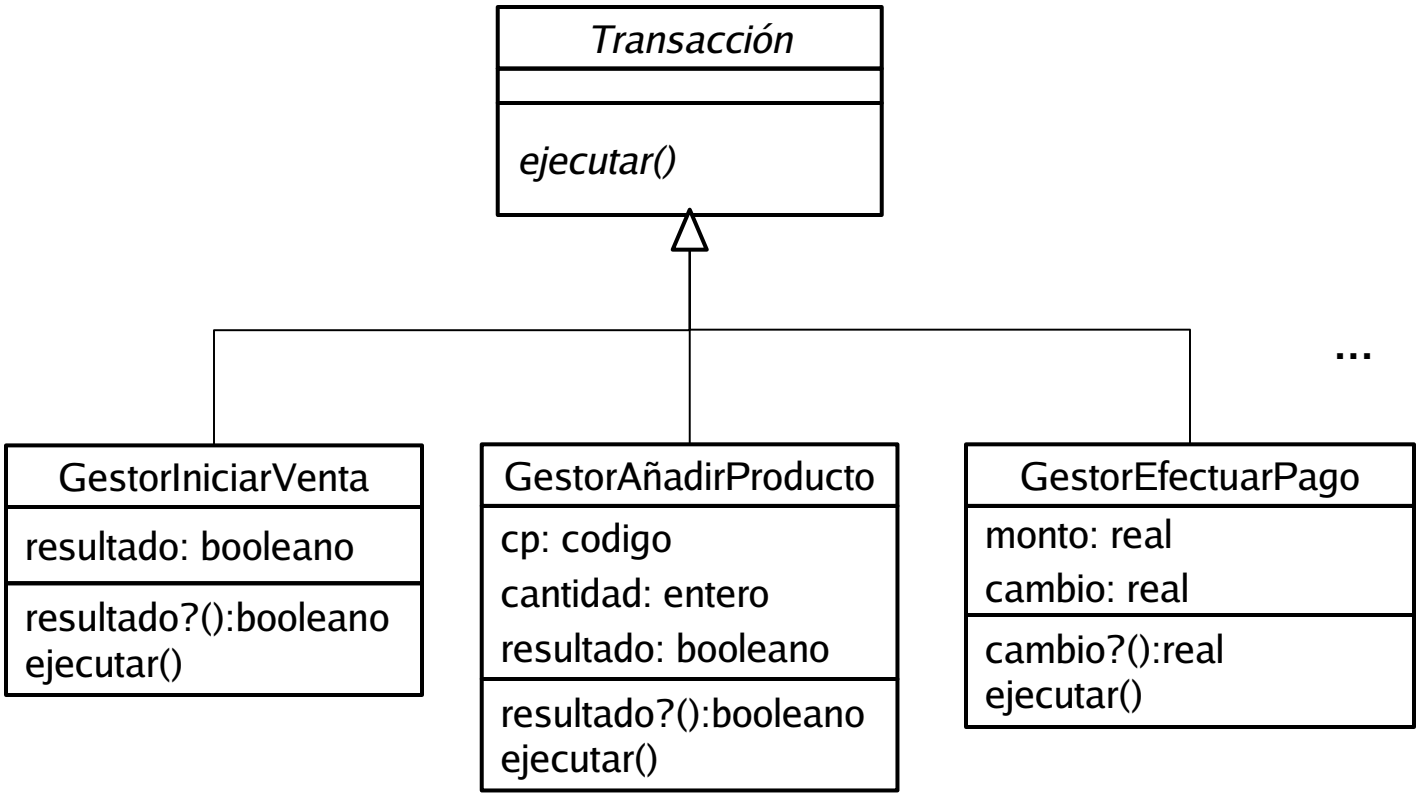
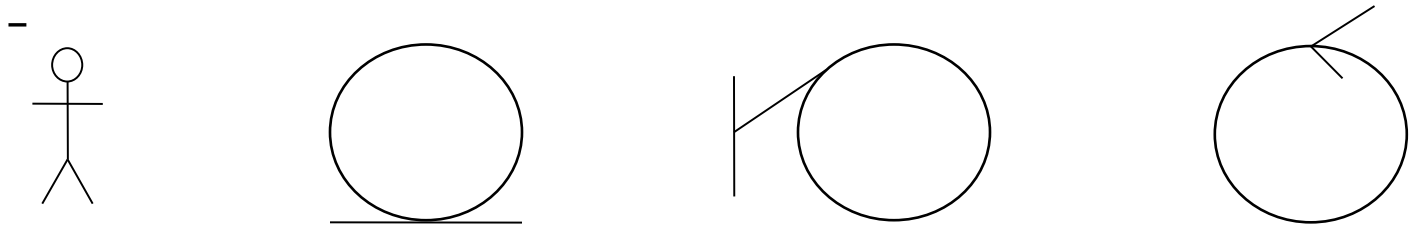


Diagrama de Jacobson y controladores CU y Transacción



ACTOR

Entidad

Interfaz

Control

Representa datos almacenados

Representa una interfaz del sistema

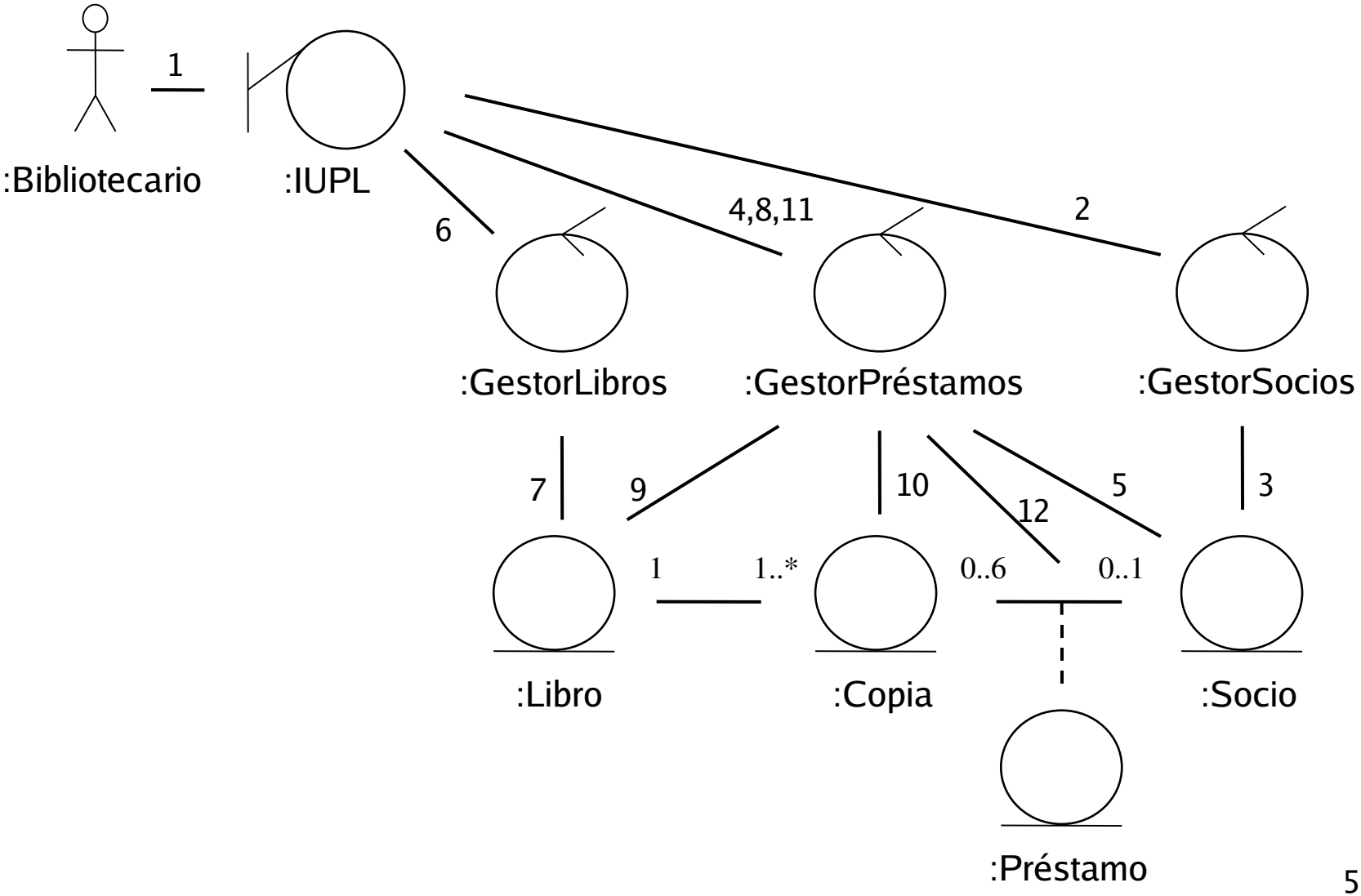
Representa transferencia de información

frontera

Patrón de diseño

“Modelo”	“vista”	“controlador”

Diagrama de Jacobson Pedir Libro



Ejemplo: Caso de uso Comprar productos

- Asignar la responsabilidad a un controlador:

Fachada (sistema):	TPV
Fachada (empresa):	Supermercado
Tarea (actor):	Cajero
Caso de uso (gestor):	GestorComprarProductos
Transacción (evento):	GestorIniciarVenta
	GestorAñadirVenta
	GestorTerminarVenta
	GestorEfectuarPago

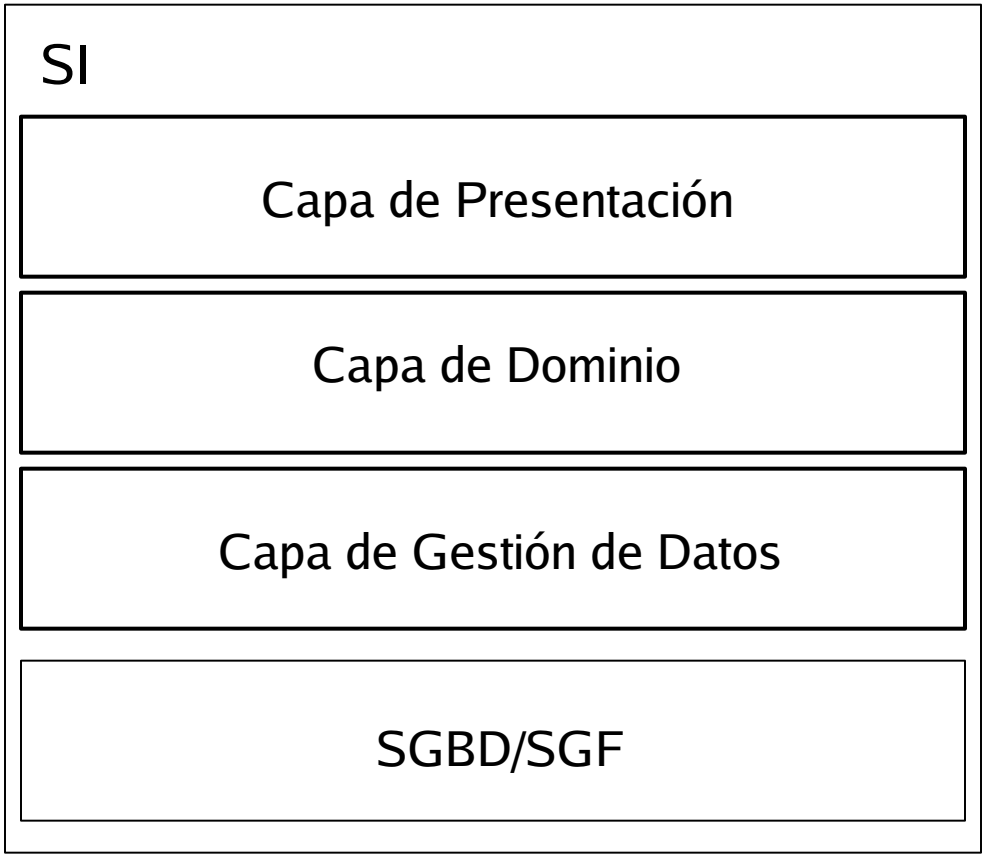
Para elaborar diagramas de secuencia

- Preparar un diagrama individual para cada contrato (que corresponde a una operación descrita en el diagrama de secuencia)
  - Para cada evento del sistema construir un diagrama de secuencia que lo incluya como mensaje inicial
- Si el diagrama se vuelve complejo, dividirlo en más pequeños
- Con las responsabilidades contractuales, las poscondiciones y la descripción del caso de uso real como punto de partida, diseñar un sistema de clases que interactúen para realizar la tarea. Aplicar los patrones GRASP para elaborar un buen diseño.

Diseño de la Capa de Presentación

- Arquitectura lógica en tres capas
- Diseño interno y externo de la capa de presentación
- Patrón Modelo – Vista – Controlador

Arquitectura lógica de un SI en tres capas



Responsable de la interacción con el usuario

Responsable de las funcionalidades del SI

Responsable de la interacción con el SGBD

Diseño de la Capa de Presentación

- Capa de presentación: componente del sistema software encargado de gestionar la interacción con el usuario
- Su diseño contempla:
  - Diseño externo: cómo el usuario interacciona con el SI (diseño externo)
  - Diseño interno: cómo la Capa de Presentación interacciona con la Capa de Dominio
- Su diseño requiere conocer:
  - Características tecnológicas de los periféricos de entrada (ratón, teclado, ...) y de salida (pantalla, impresora, ...)
  - Casos de uso reales

Diseño de la Capa de Presentación

- Normalmente es un proceso de diseño basado en prototipos

- Por un equipo de diseñadores
  - Conocimiento del dominio del sistema: usuario final
  - Conocimiento tecnológico: informáticos
  - Conocimiento en psicología, sociología, fisiología: psicólogos
  - Conocimiento en expresión gráfica: diseñadores gráficos

- El diseño que hagamos depende de la tecnología gráfica!

Diseño de la Capa de Presentación

- Diseño Externo: de los elementos (tangibles) que el usuario ve, oye, toca al interactuar con el sistema

- El diseño externo consiste en la definición de:
  - Mecanismos de interacción: formas con los que el usuario puede realizar peticiones
    - Escribir en la línea de comando, escoger opciones de desplegables, pantallas táctiles, peticiones orales, ...
  - Presentación de la información: formas con los que mostrar el resultado de las peticiones generadas por el usuario
    - Formatos gráficos, imágenes, listados (en pantalla, ...), ...

Diseño de la Capa de Presentación

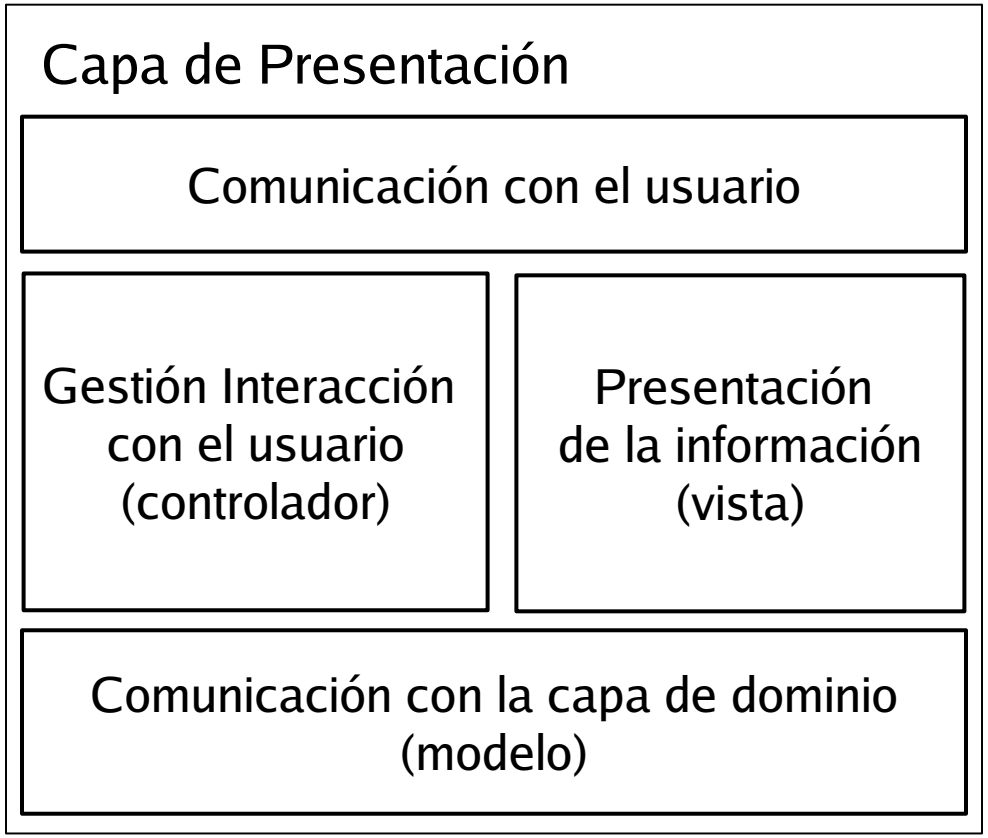
- Principios básicos para el diseño de la interacción con el sistema y la presentación de la información:
  - Integridad conceptual de la interacción (secuencias de acciones, abreviaturas, ayudas, ...) y en la presentación (formatos, colores, terminología, ...)
  - Control sobre las acciones a realizar (selección en vez de formato libre, evitar entrar datos redundantes, ...)
  - Hergonomía (que el usuario no deba recordar códigos complejos, información de otras pantallas, etc.)
  - Mensajes de error claros y explicativos
  - Barras de progreso, ...
  - Personalización del mecanismo de interacción y presentación según su perfil

Diseño de la Capa de Presentación

- Diseño Interno: los mecanismos que recogen, procesan y dan respuesta a las peticiones de los usuarios

- Comunicación con el usuario
  - Receptor de los eventos de presentación
  - Interfaces Gráficas de Usuario (GUI) basada en eventos (SO)
- Gestión de la interacción con el usuario
  - Controlar los eventos externos
- Presentación de la información
  - Recepción y presentación de los datos
- Comunicación con la capa de dominio
  - Enviar eventos externos a procesar
  - Recibir respuestas y trasmitirlas

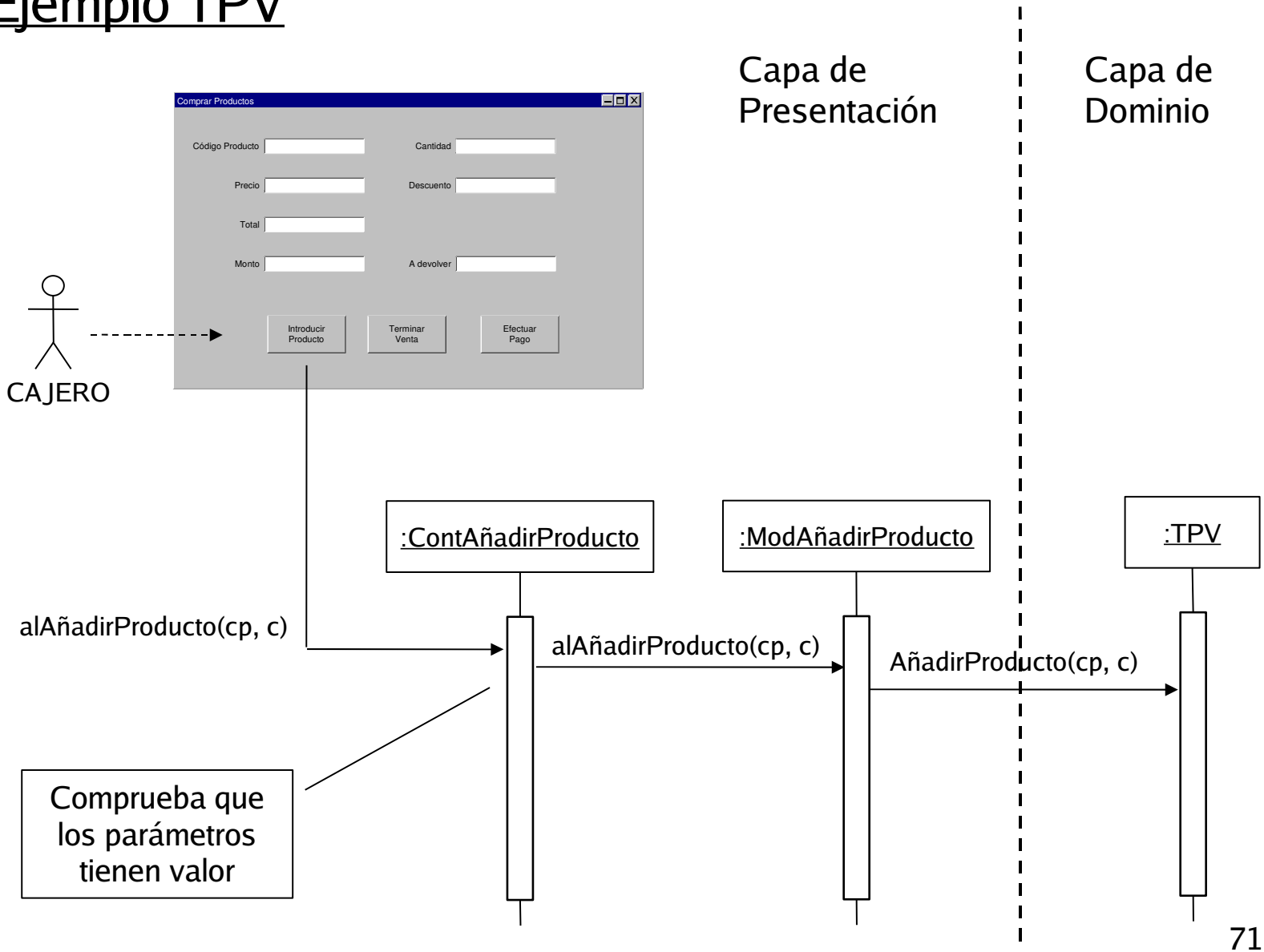
Diseño de la Capa de Presentación



Patrón arquitectónico Modelo – Vista – Controlador

- Contexto:
  - SI interactivos con interfaces de usuario flexibles
- Problema:
  - Independencia de la IGU (y su tecnología) respecto al núcleo del sistema (permitir mostrar diferentes vistas de la información)
- Solución:
  - Descomponer el sistema en tres componentes
    - Modelo (proceso): implementación de las funcionalidades y datos
    - Vista (salida): muestra la información al usuario
    - Controlador (entrada): gestiona la interacción con el usuario
- Consideraciones:
  - El modelo representa la Capa de Dominio (y gestión de datos) del SI
- Beneficios:
  - Responsabilidades distribuidas: alta cohesión

Ejemplo TPV


Capa de Presentación
Capa de Dominio
Comprar Productos
Código Producto
Cantidad
Precio
Descuento
Total
Monto
A devolver
Introducir Producto
Terminar Venta
Efectuar Pago
CAJERO
:ContAñadirProducto
:ModAñadirProducto
:TPV
alAñadirProducto(cp, c)
alAñadirProducto(cp, c)
AñadirProducto(cp, c)
Comprueba que los parámetros tienen valor

Diseño de la Capa de Gestión de Datos

- Persistencia
- Diseño automático y directo
- Persistencia en BDOO
- Persistencia en BD relacionales

Diseño de la Capa de Gestión de Datos

- Hasta el momento, todas las datos (clases, instancias y objetos) eran no permanentes (sólo existen mientras se ejecuta el SI)
- En la gran mayoría de aplicaciones es necesario guardar y recuperar la información en una BD
- Los objetos pueden hacerse persistentes en distintos SGBD:
  - Bases de Datos Orientadas a Objetos (BDOO)
  - Bases de Datos Relacionales
  - Otros: ficheros planos, etc.
- El diseño que hagamos depende del SGBD!
  - Generadores automáticos
  - Diseño directo

Diseño de la Capa de Gestión de Datos

- Generador automático de la persistencia
  - Sistema que proporciona una traducción automática para almacenar los datos en memoria externa
  - Transforma las actualizaciones hechas en memoria principal en actualizaciones en memoria externa
  - + La gestión de la persistencia es transparente al diseñador
  - - No puede aprovechar todas las potencialidades del SGBD
  - Ejemplos:
    - Top Link
    - JavaBlend (Sun Microsystems)

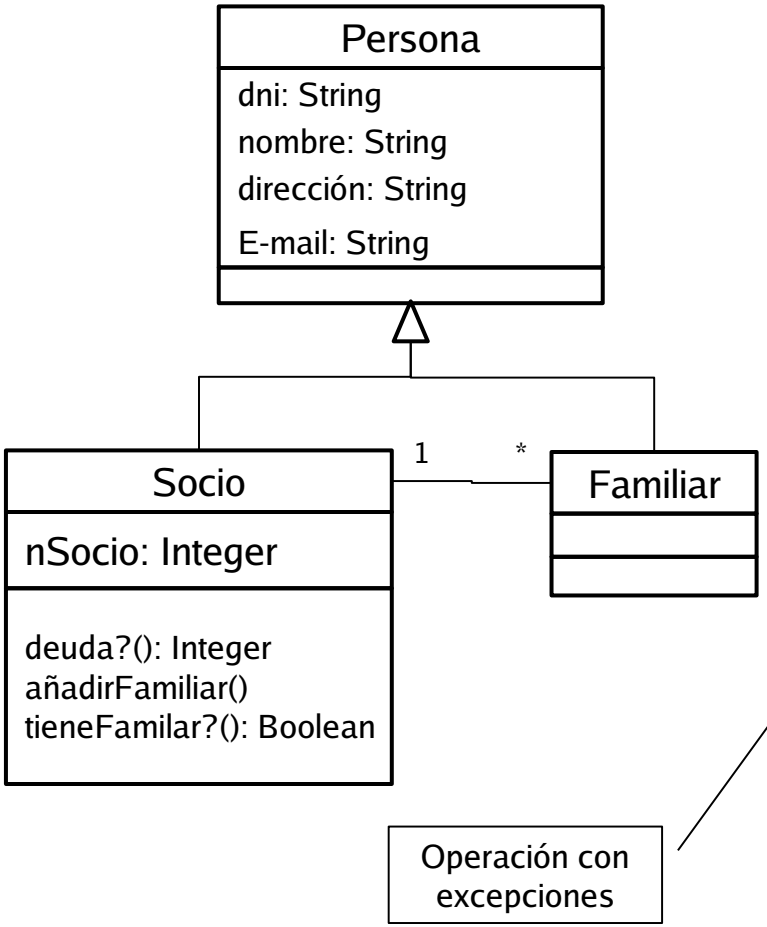
Diseño de la Capa de Gestión de Datos

- Diseño directo de la persistencia
  - El diseño contempla el SGBD que utilizará el SI
  - El diseño será distinto según el SGBD usado:
    - OO, relacional, ficheros, serialización de Java, etc.
  - El diseñador debe conocer el SGBD para diseñar la capa de Gestión de Datos
  - + Puede aprovechar completamente las funcionalidades del SGBD
  - + El diseño es muy similar para todos los SGBD (p.e. Oracle, MySQL, SQLServer, ...) de una misma familia (p.e. Relacional)

Persistencia en BDOO

- ODMG (Object Database Management Group)
  - Proponen un mecanismo estándar para trabajar con BDOO
- Transparencia de datos entre los objetos del SI y la BD
- La Capa de Gestión de Datos es transparente!
- Ejemplos:
  - ObjectStore
  - O2
  - Hibernate
- Componentes de un SGBDOO:
  - Modelo de objetos: objeto, tipo, estado, transacciones, etc.
  - ODL (lenguaje de especificación de objetos)
  - OQL (lenguaje de consulta de objetos)
  - OML (lenguaje de manipulación de objetos)
  - Enlace entre el LPOO y el SGBDOO (p.e. Java binding)

Persistencia en BDOO: ODL



Conjunto de Persona

Clave de Persona

```
class Persona (extent Personas key dni) {
        attribute string dni;
        attribute string nombre;
        attibute string dirección;
        attribute string e-mail;
        };
```

Herencia

```
class Socio extends Persona {
        attibute short nSocio;
        relationship set<Familiar> familiares
                        inverse Familiar::socio;
        short deuda?();
        void añadirFamiliar (in string nombre;
                        in string direccion, in string e-mail)
        raises
                        (familiar_ya_existe);
        boolean tieneFamiliar(in string nombre);
        }
```

Navegabilidad doble

Operación con excepciones

```
class Familiar extends Persona {
        relationship Socio socio inverse Socio::familiares
        };
```

Persistencia en BDOO: OQL

- Muy parecido a SQL92

```
typedef set<string> vectdir;
vectdir (select distinct direccion
                from Personas
                where nombre='Pepe')

typedef bag<string> socs;
socs (select distinct socio(a:nombre,b:direccion,c:e-mail)
                from Personas
                where nombre='Pepe')
```

Persistencia en BDOO: Java binding

- Modelo objeto de Java
  - Soporta todos los conceptos del modelo ODMG excepto relaciones, extents y claves
- Java ODL
  - Describe el esquema de BD como clases Java
- Java OML
  - Permite manipular objetos persistentes
- Java OQL
  - Permite ejecutar desde Java consultas con OQL

Persistencia en BDOO: Diseño

- Normalización del modelo conceptual:
  - No permite implementar directamente
    - asociaciones n-arias, con n>2
    - clases asociativas
  - Tratamiento de la información derivada

\+ Su modelo de datos es OO, pero ...

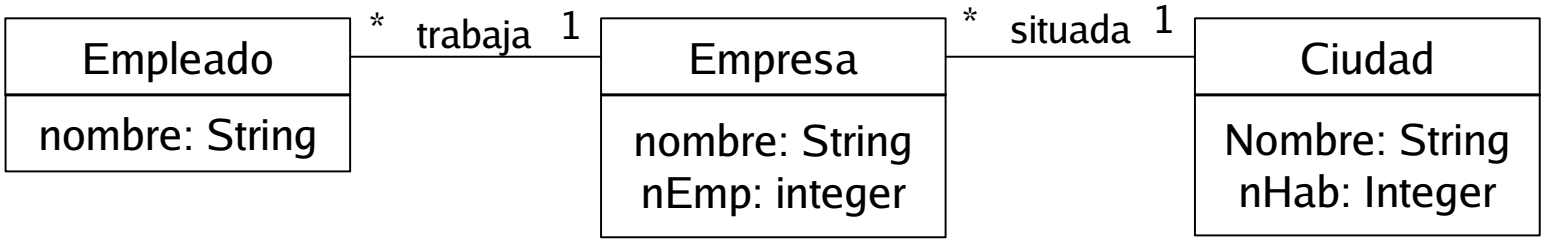
\- ... los SGBDOO proporcionan pocas funcionalidades

\+ Java Binding nos permite diseñar las operaciones como si éstas se ejecutaran en memoria principal

\+ Podemos usar los patrones de diseño OO

\- Excepto con el uso de OQL para diseñar las operaciones de consulta

Persistencia en BDOO: Diseño con OQL (1)



Nombre: consultaEmpleados() : ListaEmpleados

Responsabilidades: Obtener el nombre de los empleados que trabajan en empresas con más de 100 trabajadores en ciudades de más de 100.000 habitantes

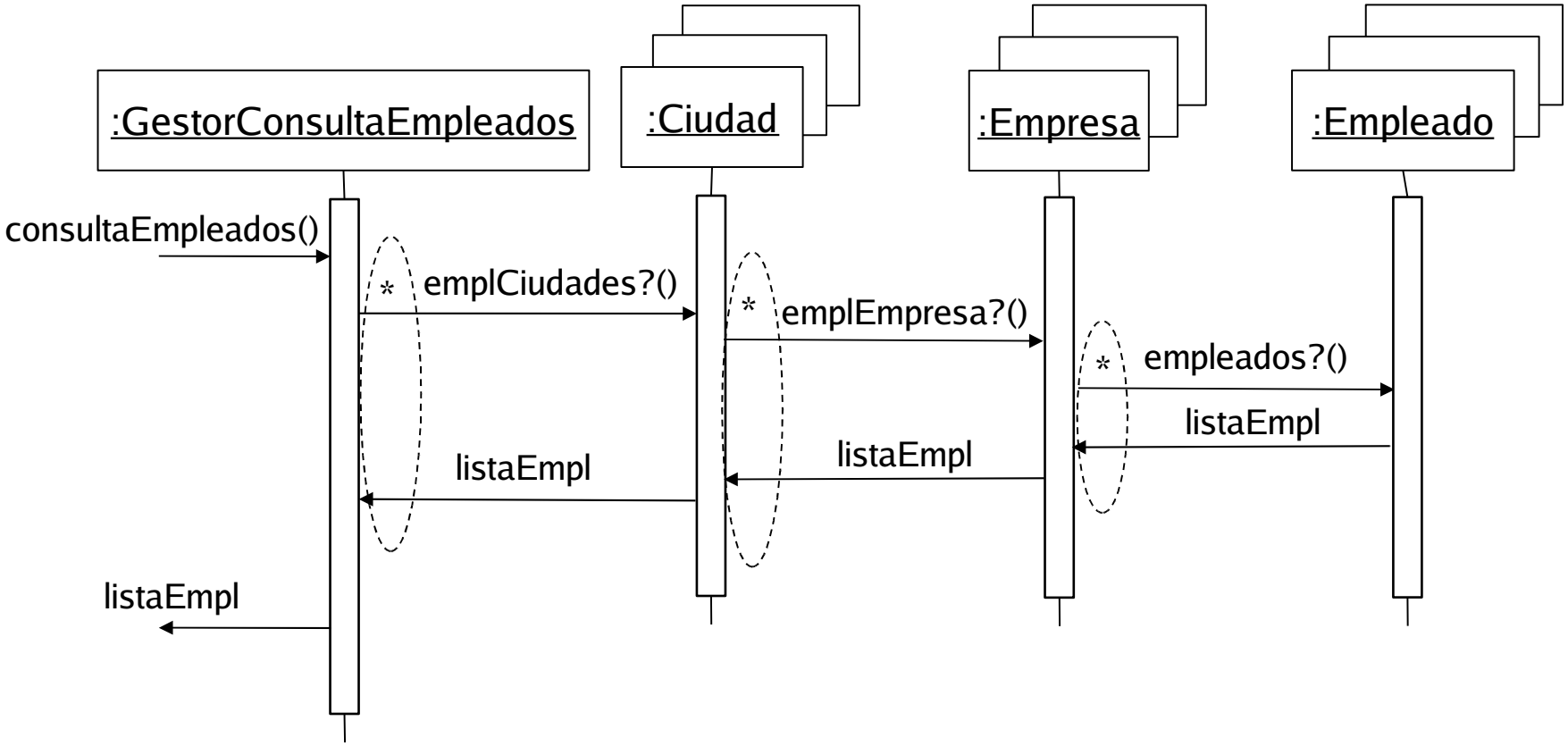
Precondiciones:

Postcondiciones:

Salida: Lista de los nombres de los empleados que trabajan en empresas con más de 100 trabajadores en ciudades de más de 100.000 habitantes

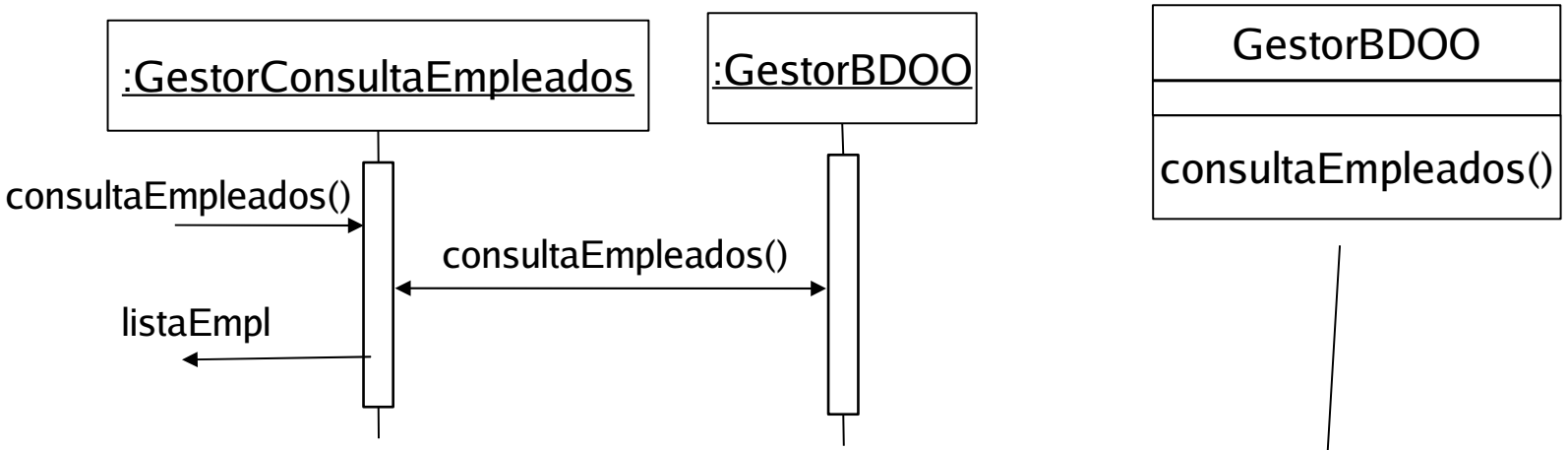
Persistencia en BDOO: Diseño con OQL (2)

- Solución OO pura



Persistencia en BDOO: Diseño con OQL (3)

- Solución usando OQL




```
SELECT DISTINCT e.nombre
FROM Empleados e, Empresas emp, Ciudades c
WHERE   e.empresa=emp AND emp.nEmp>100 AND
        emp.ciudad=c AND c.nHab=100000
```

Persistencia en DB relacionales

- Los SGBD relacionales son los más utilizados actualmente
- Su modelo de datos sigue una filosofía distinta a la OO!
- La Capa de Gestión de Datos es responsable de:
  - Desmaterializar: traducir los objetos a registros para guardarlos en la BD
  - Materializar: traducir los registros a objetos para recuperarlos de la BD
- Si realizamos un diseño directo sobre una BD relacional debemos:
  - Adaptar el Diagrama de Clases (normalización), los contratos, los diagramas de secuencia, etc.
  - Considerar las funcionalidades propias del SGBD: control de las Restricciones de integridad, lenguaje de acceso eficiente, concurrencia (transacciones), etc.

Persistencia en DB relacionales

- Traducción del Modelo Conceptual OO al modelo relacional
  - Normalización del Modelo Conceptual
  - Diseño lógico de la Base de Datos
  - Tratamiento de la información derivada
  - Tratamiento de las Restricciones de Integridad

- Diseño de las operaciones
  - Aprovechamiento de la potencia de SQL
  - Uso de procedimientos almacenados
  - Asignación de postcondiciones al SGBD
  - Establecer los límites de las transacciones

Normalización de BD relacionales

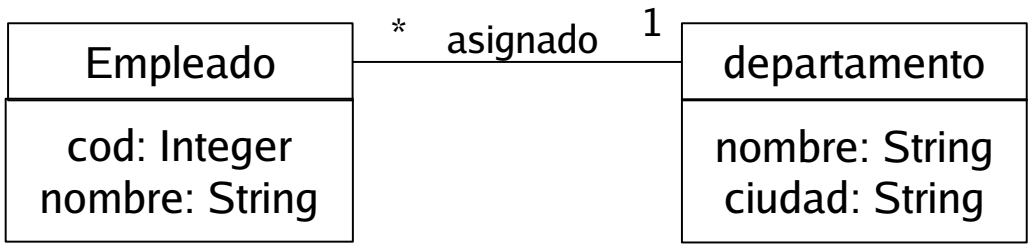
- Limitación tecnológica de la OO: no permite implementar directamente todos los conceptos que hemos usado anteriormente
  - Asociaciones n-arias, con n>2 y binarias N:M
  - Clases asociativas
  - Jerarquías de especialización
  - Información derivada
- Es necesario un proceso de normalización (binarización)
  - Eliminar asociaciones n-arias, con n>2 y binarias N:M
  - Eliminar clases asociativas
  - Eliminar las jerarquías de especialización
  - Tratar la información derivada
- Todo ello modifica el diagrama de clases y los contratos

Diseño lógico de la Base de Datos

- Básicamente, hay que traducir los elementos del modelo conceptual a componentes que puedan implementarse en los SGBD relacionales:
  - Objetos a tablas
  - Asociaciones entre clases de objetos
  - Todos los aspectos no contemplados por las BD relacionales:
    - Jerarquías de especialización (herencia)
    - Clases sin identificadores internos
    - Operaciones asociadas a las clases de objetos

Diseño lógico de la Base de Datos

- Objetos a tablas
- Asociaciones binarias 1:N

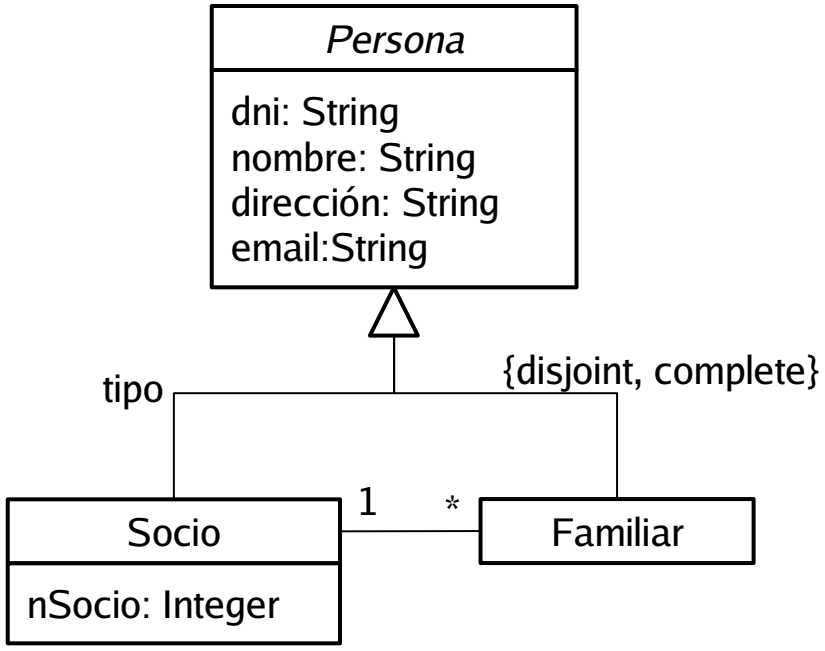


a) departamento(<u>nombre-dep</u>,ciudad)
   empleado(<u>cod</u>,nombre-emp,*nombre-dep*)

b) departamento(<u>nombre-dep</u>,ciudad)
   empleado(<u>cod</u>,nombre-emp)
   asignación(<u>cod,nombre-dep</u>)

Diseño lógico de la Base de Datos

- Jerarquías de especialización
  - Desaparecen las clases abstractas (no existen en SGBD relacionales)
  - La traducción depende del nivel al que se quiere colapsar la jerarquía



a) Persona(dni,nombre,dirección,email)
   Socio(nSocio,*dni*)
   Familiar(*dni*,*nSocio*)

b) Persona(dni,nombre,dirección,email,
   nSocio, *dni-socio-fam*)

admiten valores nulos

Tratamiento de la información derivada

- Los atributos y las asociaciones derivadas pueden ser:
  - Calculados
  - Materializados

- Si se calculan, desaparece la información derivada (atributo o asociación)
  - Aparecen nuevos métodos o vistas que cuando se consultan permiten obtener la información derivada de forma automática
- Si se materializan, desaparece la indicación de que la información es derivada (atributo o asociación)
  - Se modifica adecuadamente la capa de dominio
  - Se modifica adecuadamente la capa de gestión de datos usando el SGBD (triggers –disparadores-)

Tratamiento de las Restricciones de Integridad

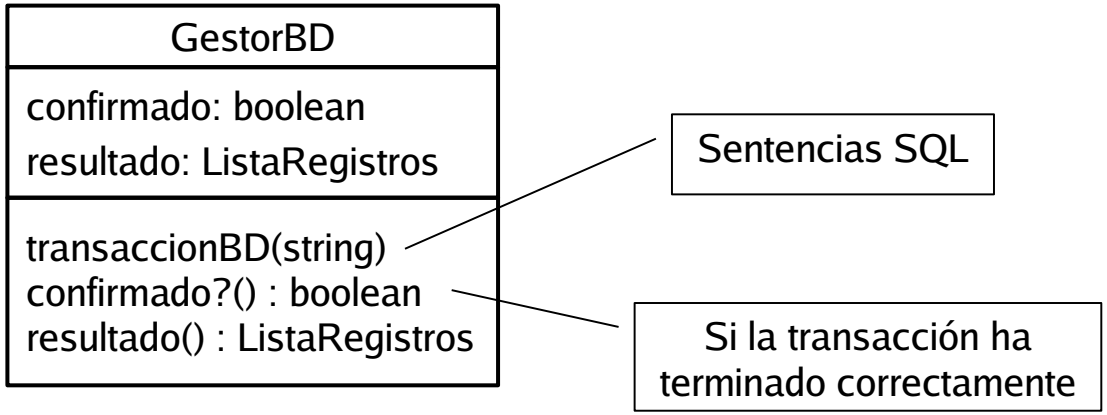
- Los SGBD relacionales proporcionan diversas funcionalidades para tratar las restricciones de integridad
  - Restricciones de columna
    - Not null
    - Distinct
    - Unique
    - Primary key
    - Foreign key
    - Check
  - Disparadores
  - Etc.

Diseño de las operaciones del sistema

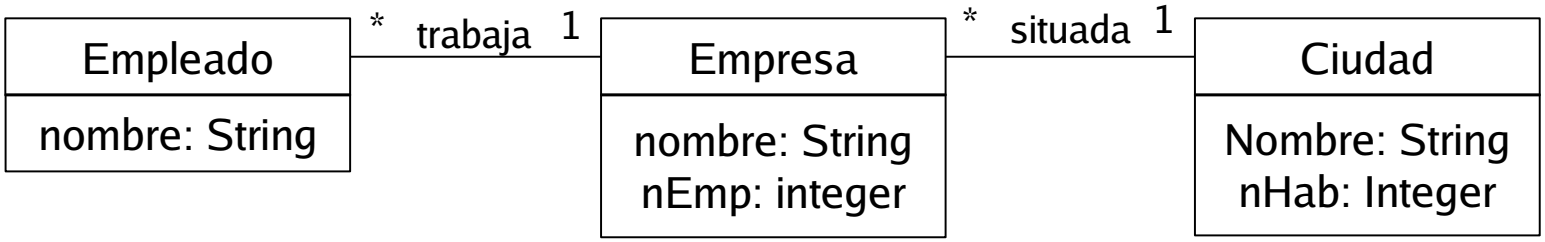
- Las operaciones pueden aprovechar la potencia del SGBD
- Las operaciones que se deben diseñar pueden residir en:
  - Memoria principal (programas, clases, etc.)
  - El propio SGBD (procedimientos almacenados)
    - Simplifica el desarrollo de aplicaciones
    - Mejora el rendimiento de la BD
    - Controla las operaciones que los usuarios realizan contra la BD
- En algunos casos, el propio SGBD puede tener la responsabilidad de las postcondiciones de una operación (p.e. on delete cascade)
- Debemos conocer cuáles son las transacciones, cuál es su inicio (begin) y cuál su final (commit o rollback)

Persistencia en DB relacionales

- El diagrama de clases (o parte de ella) debe ser convertido en un esquema lógico de una BD relacional (DBD)
- Se usará SQL para acceder a la BD. Tendremos un controlador (o varios) que nos permitan ejecutar transacciones (FBD, DBD y ABD)
- Las tablas relacionales no pueden tener métodos!



Persistencia en BD Relacionales: Diseño con SQL (1)



Nombre: consultaEmpleados() : ListaEmpleados

Responsabilidades: Obtener el nombre de los empleados que trabajan en empresas con más de 100 trabajadores en ciudades de más de 100.000 habitantes

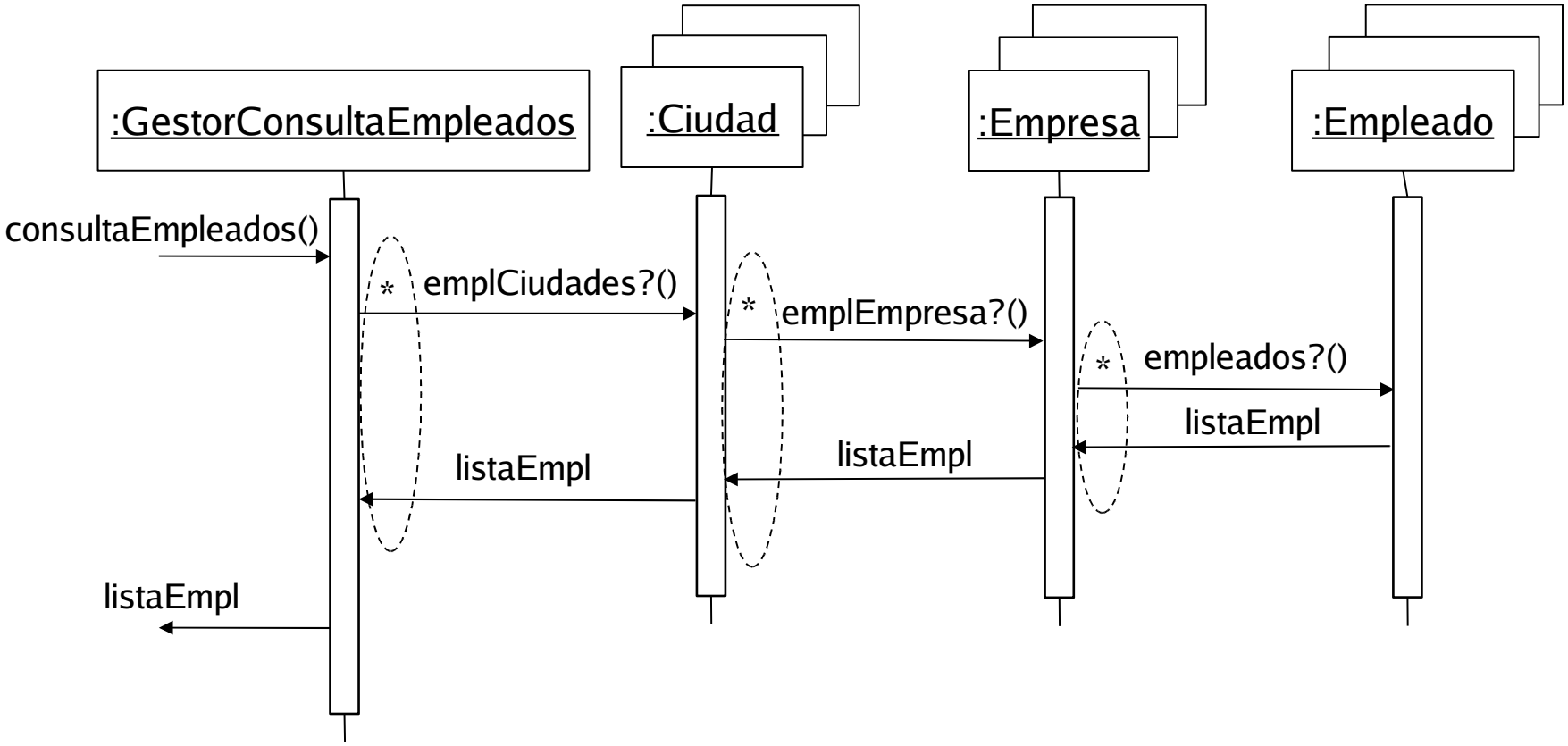
Precondiciones:

Postcondiciones:

Salida: Lista de los nombres de los empleados que trabajan en empresas con más de 100 trabajadores en ciudades de más de 100.000 habitantes

Persistencia en BD Relacionales: Diseño con SQL (2)

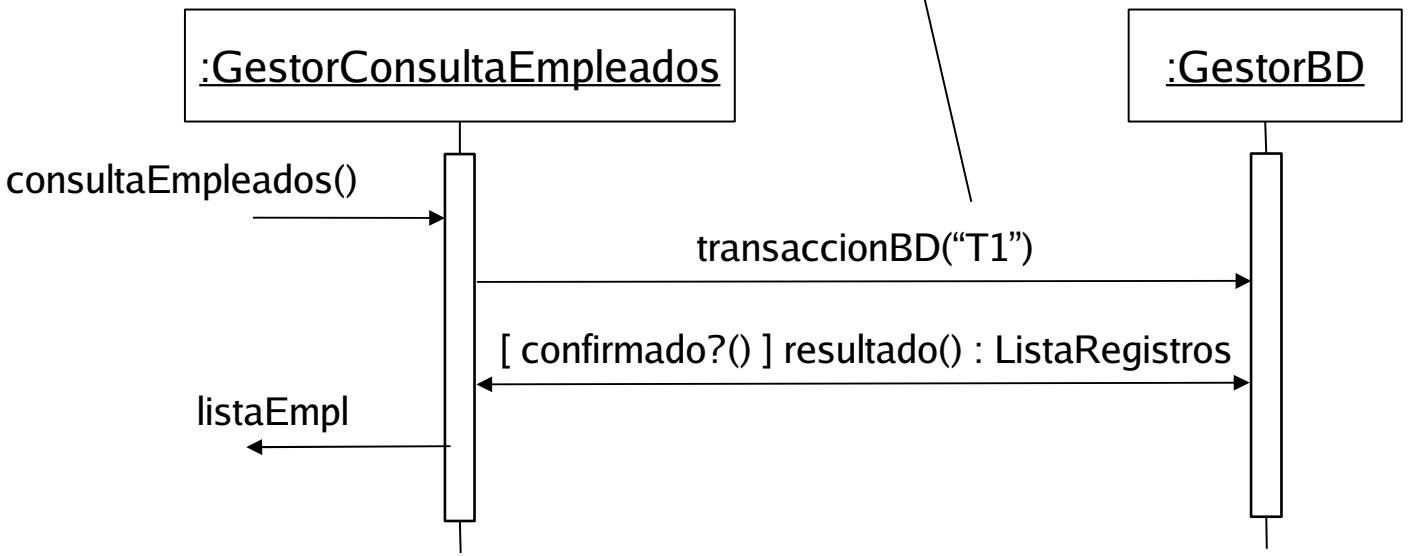
- Solución OO pura



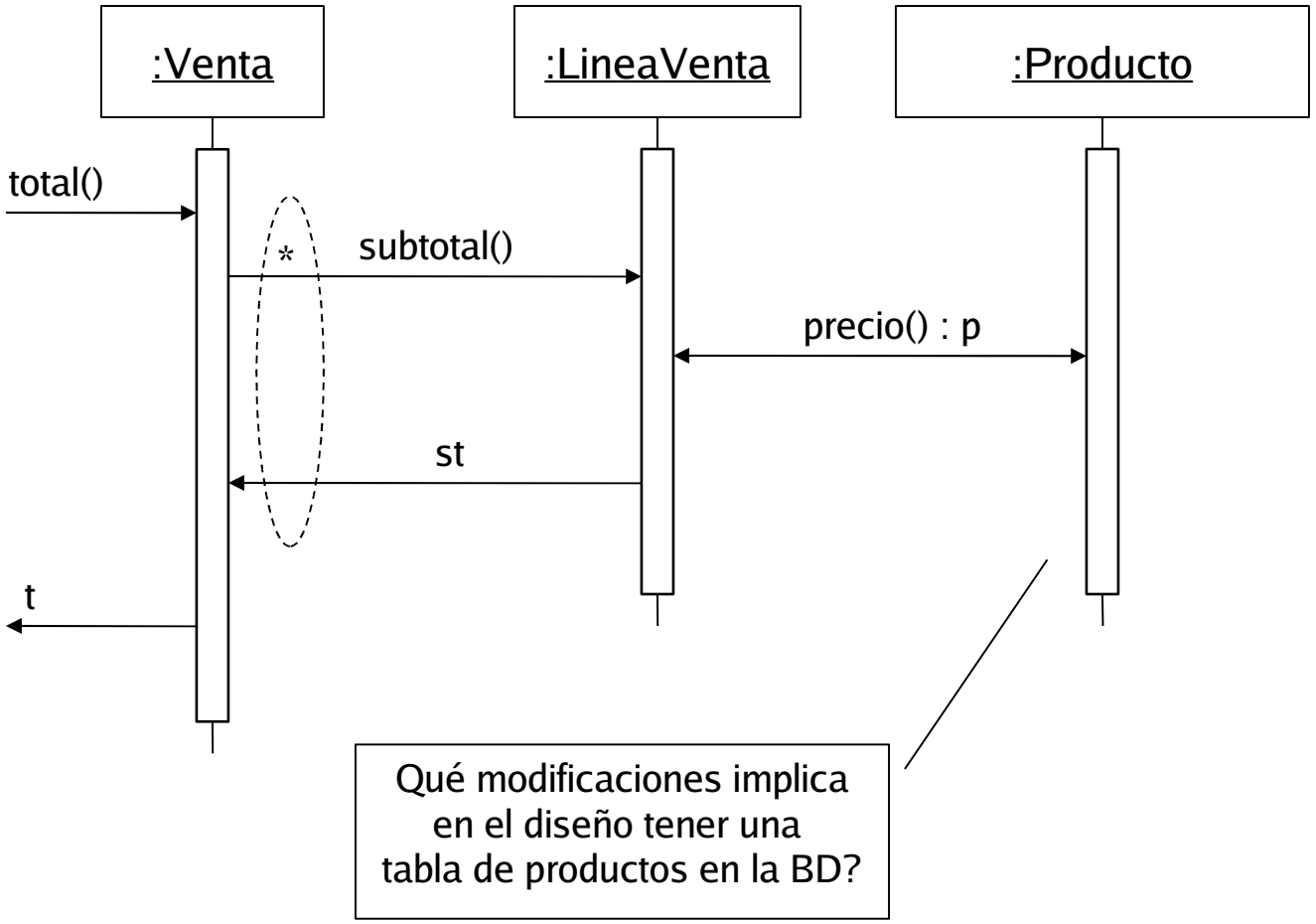
Persistencia en BD Relacionales: Diseño con SQL (3)

- Solución usando SQL

```
T1: SELECT DISTINCT e.nombre
    FROM Empleados e, Empresas emp, Ciudades c
    WHERE e.empresa=emp AND emp.nEmp>100 AND
          emp.ciudad=c AND c.nHab=100000
```



Persistencia de las BD relacionales



Persistencia en DB relacionales

- El diseño de la Capa de Gestión de datos debe considerar (entre otros) los siguientes factores:
  - Qué objetos deben ser persistentes? Todos, algunos, cuales?
  - Quien es el responsable de materializar/desmaterializar objetos?
  - Quien garantiza la consistencia con la BD de los objetos materializados por un SI? Otro SI puede estar modificando en la BD los objetos materializados ... uso de cachés ...
  - Qué granularidad debe tener la interacción con la BD? Instrucción SQL, transacción?
- Una arquitectura de dos capas puede ser adecuada para un SI que use BD relacionales