Patrones

- Es una descripción de un problema y su solución que recibe un nombre y que puede puede aplicar en nuevos contextos.

- Los patrones tienen nombres sugerentes que ayudan a identificarlos y facilitan la comunicación.

- Una adecuada asignación de responsabilidades entre objetos contribuye al desarrollo de sistemas

Dependencias

- Modelo conceptual.
- Contratos de operación
- Casos de uso real(o esencial)
- Diagramas de colaboración.

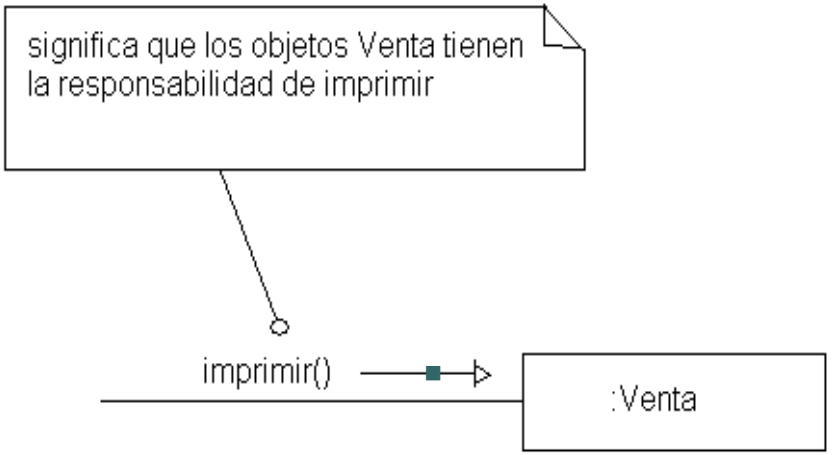
Notación

Nombre del Patrón: Experto

Solución: Asignar una responsabilidad a la clase que tiene la información necesaria.

Problema que resuelve: ¿cuál es el principio fundamental en virtud de lo cual asignaremos las responsabilidades a los objetos ?

Responsabilidad y Método



- Las responsabilidades se relacionan con las obligaciones de un objeto respecto a su comportamiento.

Patrones GRASP

- Experto
- Creador
- Alta Cohesión
- Bajo Acoplamiento
- Controlador

Experto

El experto es una clase que tiene toda la información necesaria para implementar una responsabilidad.

Experto

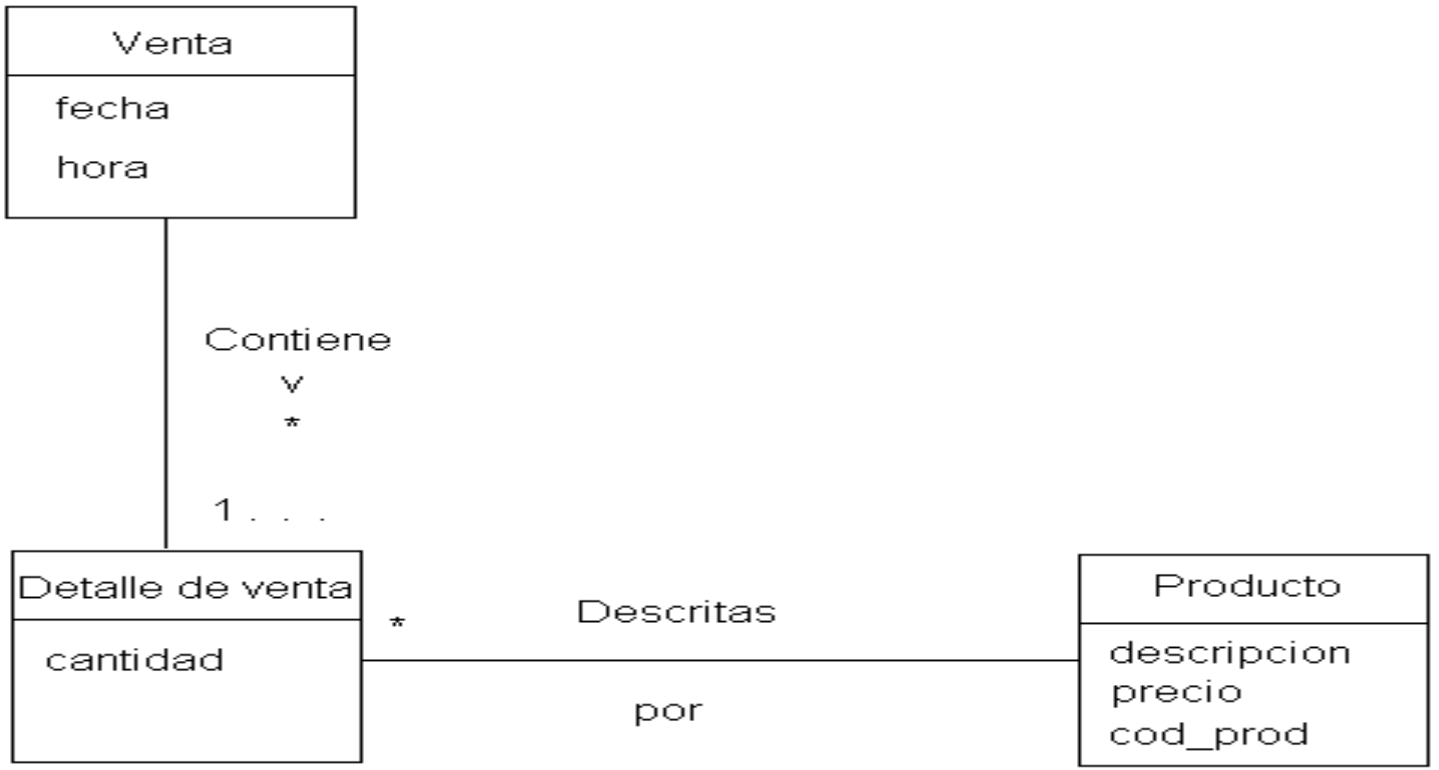
Problema: ¿cuál es el principio para asignar responsabilidades a los objetos?

Solución: Asignar una responsabilidad al experto de información a la clase que tiene la información necesaria para asignar la responsabilidad.

Experto: Ejemplo

Alguna clase necesita conocer el total de la venta.

¿Quién es el responsable de conocer el monto total de la venta?

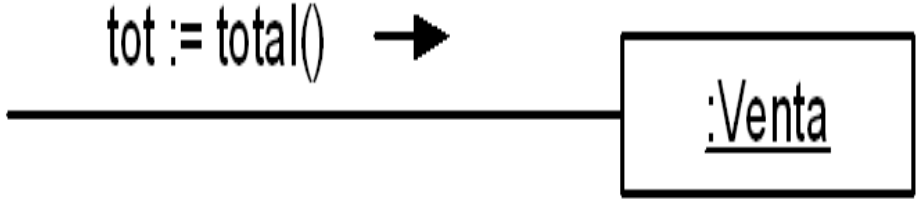


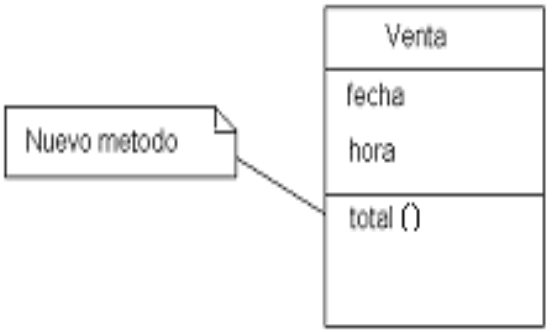
Experto: Ejemplo

- ¿Qué información necesito?

Todas las Detalle de Venta y sus respectivos subtotales

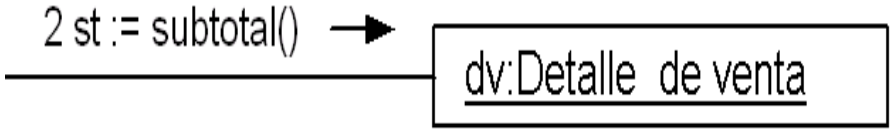
¿De qué clase del Diagrama de clases puedo obtener esa información?

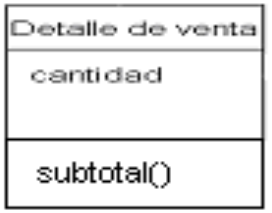




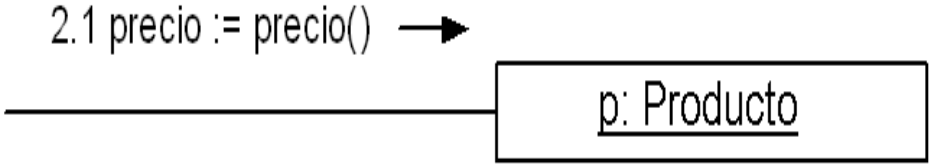
Experto: Ejemplo

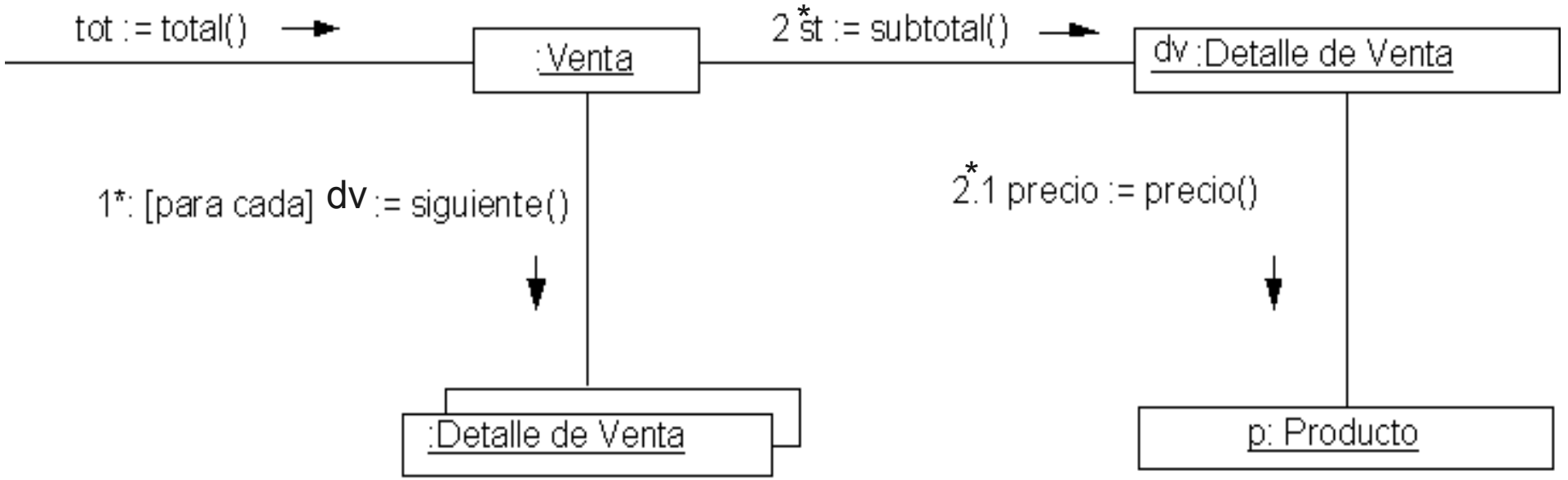
- ¿Y los respectivos subtotales (cantidad x precio) de cada Detalle de Venta ?





- Y el precio de cada Producto?





Venta
-fecha -hora
-total() : int

Detalle de Producto
-cantidad
-subtotal() : int

Producto
-descripción -precio -artID
-precio() : int

Experto :Beneficio

- Mantiene el encapsulamiento de la información
- Favorece el bajo acoplamiento.
- Son más fáciles de entender.

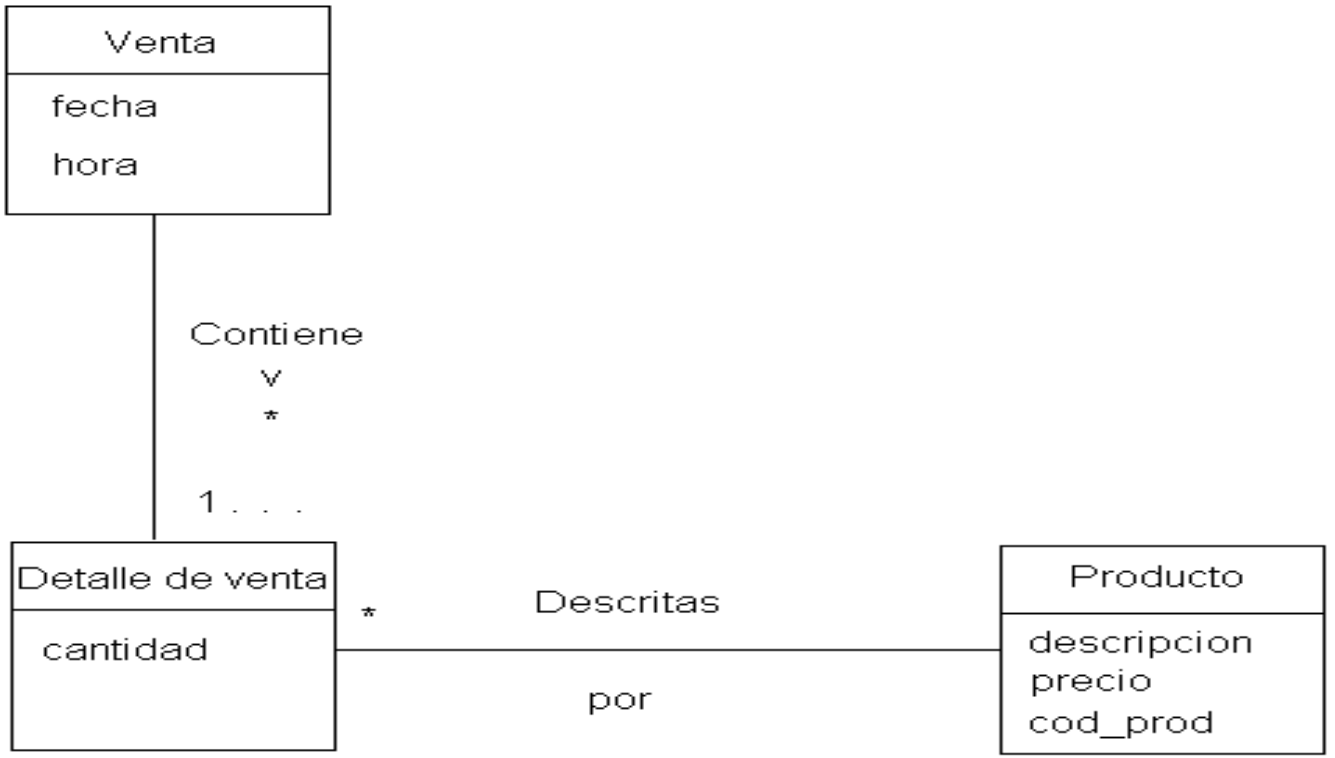
Creador

- El patrón creador nos ayuda a identificar quién debe ser el responsable de la creación (o instanciación) de nuevos objetos o clases. La nueva instancia deberá ser creada por la clase que:

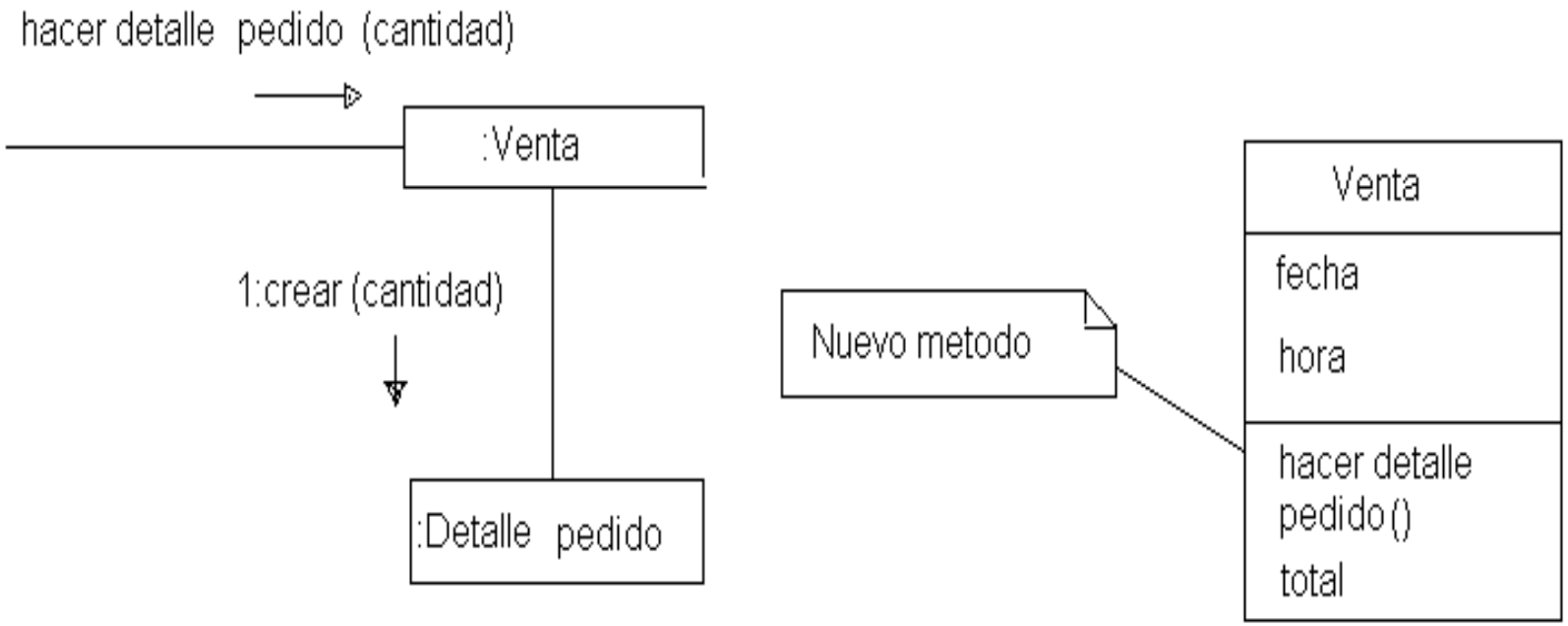
- Tiene la información necesaria para realizar la creación del objeto, o
- Usa directamente las instancias creadas del objeto, o
- Almacena o maneja varias instancias de la clase

Ejemplo Creador:

¿Quién debería encargarse de crear una nueva instancia?



Ejemplo creador



PROPÓSITO

El propósito fundamental de patrón creador es encontrar un creador que debemos conectar con el objeto producido en cualquier evento.

Condiciones

B es un Creador de A si se cumple uno o más de las siguientes condiciones:

- B agrega objetos de A
- B contiene objetos de A
- B registra objetos de A
- B utiliza intensivamente objetos de A
- B tiene los datos de inicialización de objetos de A (B es Experto en la creación de A)

Objetivos Creador

* La intención básica del patrón es encontrar un Creador que necesite conectarse al objeto creado en alguna situación
* Promueve el bajo acoplamiento, al hacer responsable a una clase de la creación de objetos que necesita referenciar

Bajo acoplamiento

El acoplamiento es una medida con la que una clase esta conectada a otras clases, con que las conoce y con que recurre a ellas.

Debe haber pocas dependencias entre las clases.
Para determinar el nivel de acoplamiento de clases, son muy buenos los diagramas de colaboración de UML.

Uno de los principales síntomas de un mal diseño y alto acoplamiento es una herencia muy profunda. Siempre hay que considerar las ventajas de la representación respecto de la herencia.

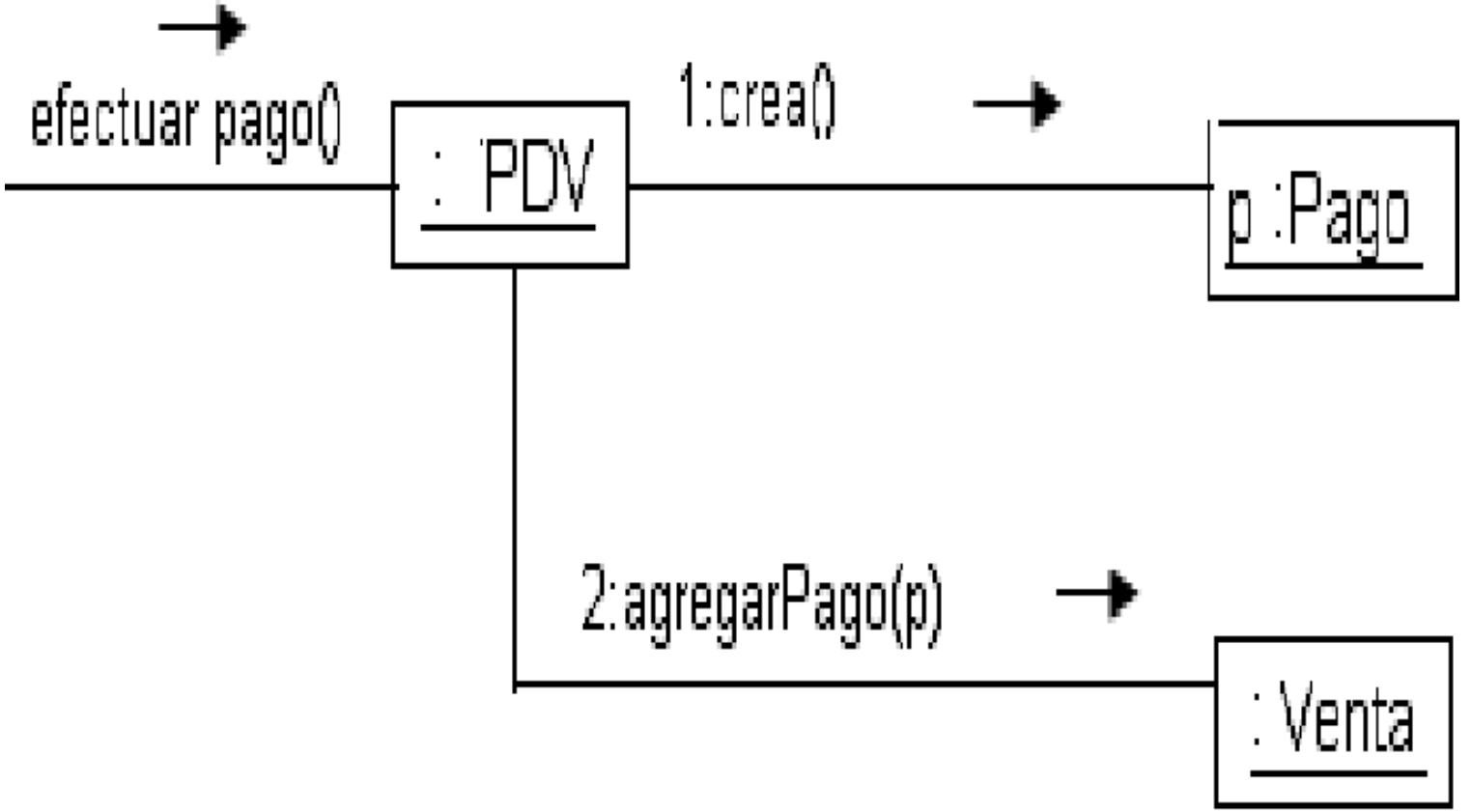
El patrón Bajo Acoplamiento es un principio a tener en mente en todas las decisiones de diseño; es un objetivo a tener en cuenta continuamente. Es un **principio evaluativo** que aplica un diseñador mientras evalúa todas las decisiones de diseño.

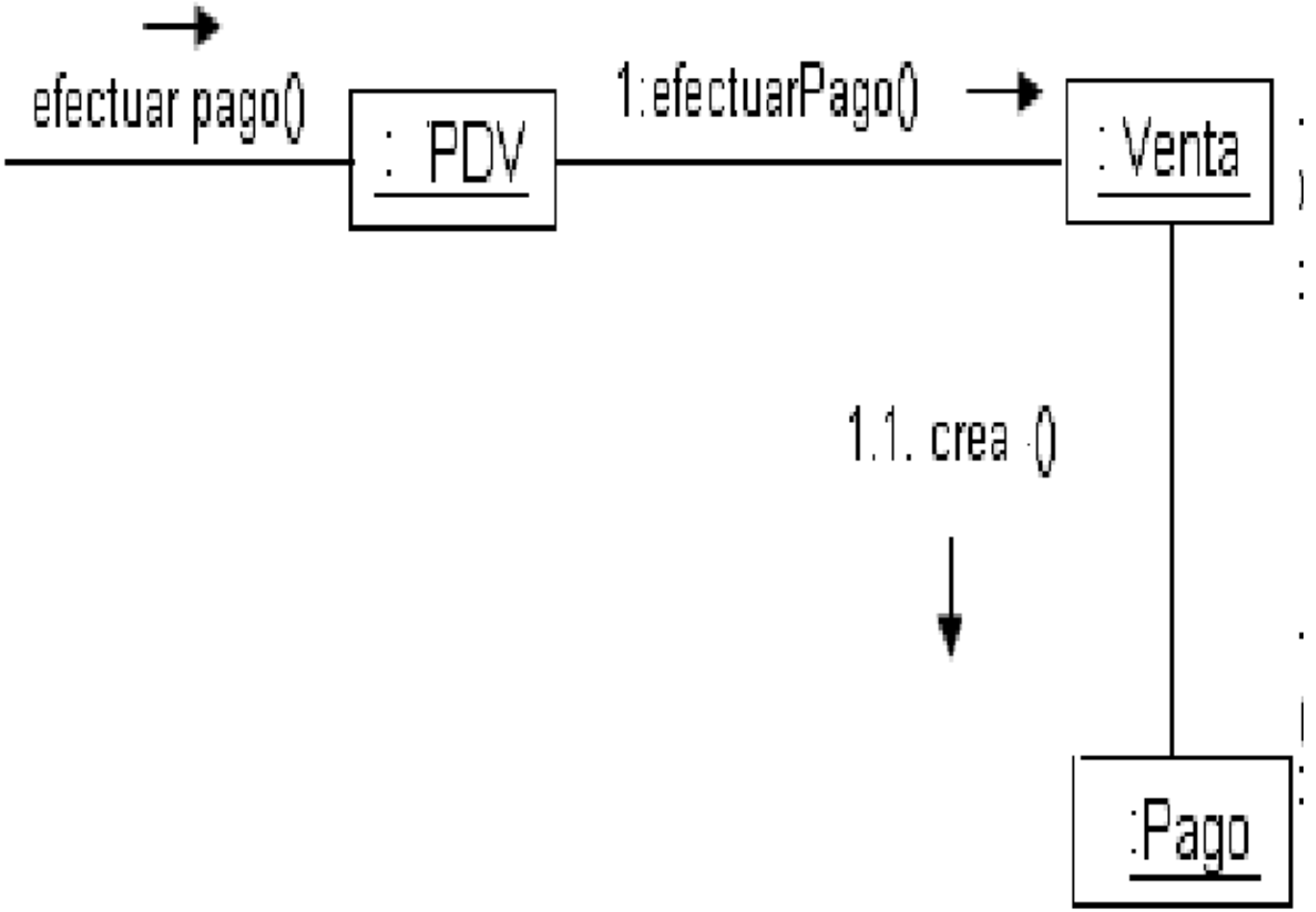
El patrón impulsa la asignación de responsabilidades de manera que su localización no incremente el acoplamiento hasta un nivel que lleve a los resultados negativos que puede producir un acoplamiento alto.

No puede considerarse en forma independiente de otros patrones cono Experto o Alta Cohesión, sino que más bien ha de incluirse como uno de los principios del diseño que influyen en la decisión de asignar responsabilidades.

El acoplamiento tal vez no sea tan importante, sino se busca la reutilización. Para apoyar una mejor reutilización de los componentes al hacerlo más independientes, el entero contexto de las metas de reúso ha de tenerse en cuenta antes de intentar reducir al mínimo el acoplamiento.

Ejemplo




efectuar pago()
: PDV
1:efectuarPago()
: Venta
1.1. crea ()
:Pago

Alta cohesión

La alta **cohesión** es una medida con la que se relacionan las clases y el grado de responsabilidades de un elemento.

Algunos escenarios que ejemplifican los diversos grados de la cohesión funcional:

Muy baja cohesión. Una clase es la única responsable de muchas cosas.

Baja Cohesión. Una clase tiene la responsabilidad exclusiva de una tarea compleja dentro de un área funcional.

Alta Cohesión. Una clase tiene responsabilidades moderadas en un area funcional y colabora con otras para realizar las tareas.

Cohesión Moderada. Una clase tiene responsabilidades exclusivas que están relacionadas lógicamente con el concepto de clase, pero no entre ellas.

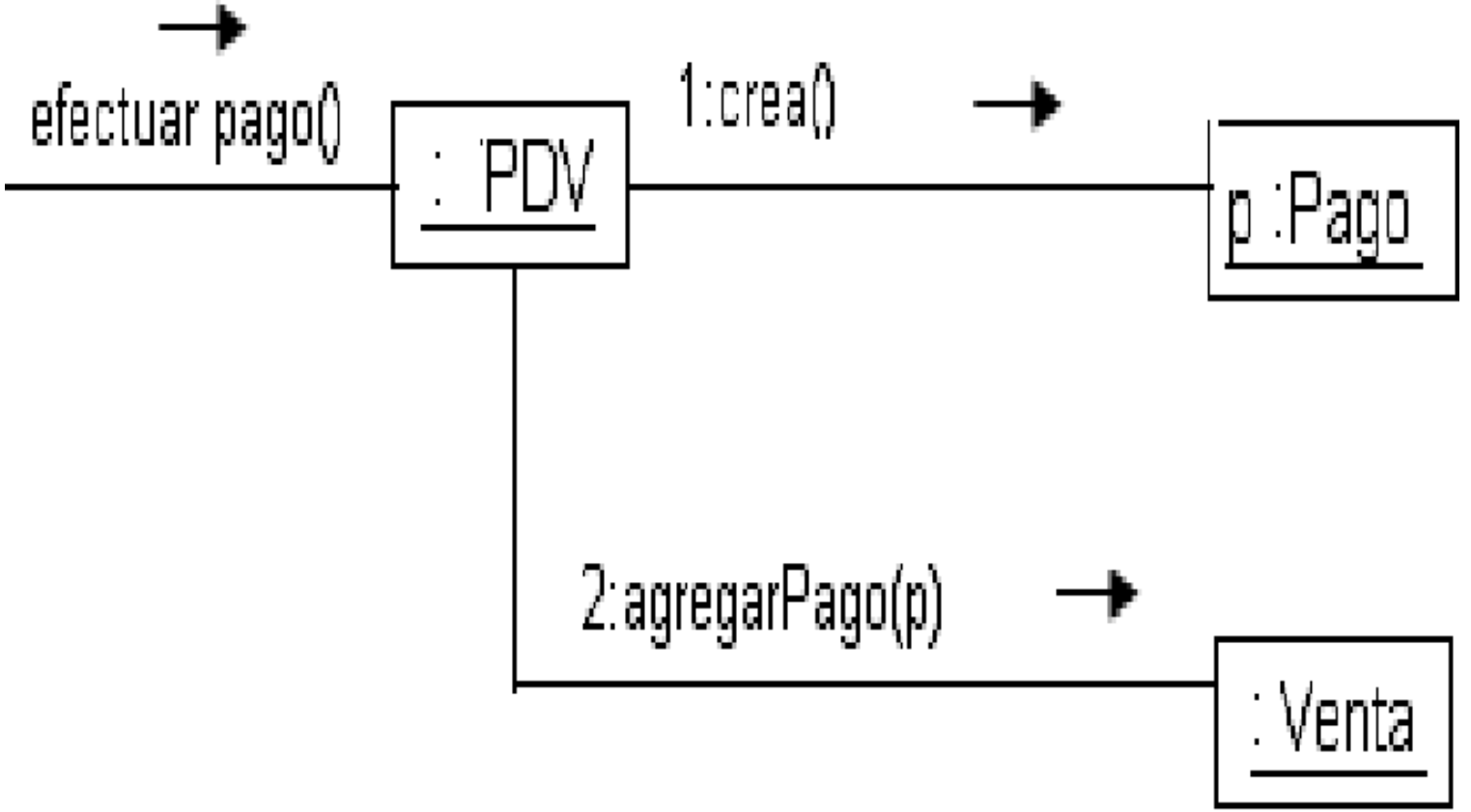
Una clase con baja cohesión hace muchas cosas, hace demasiado trabajo:

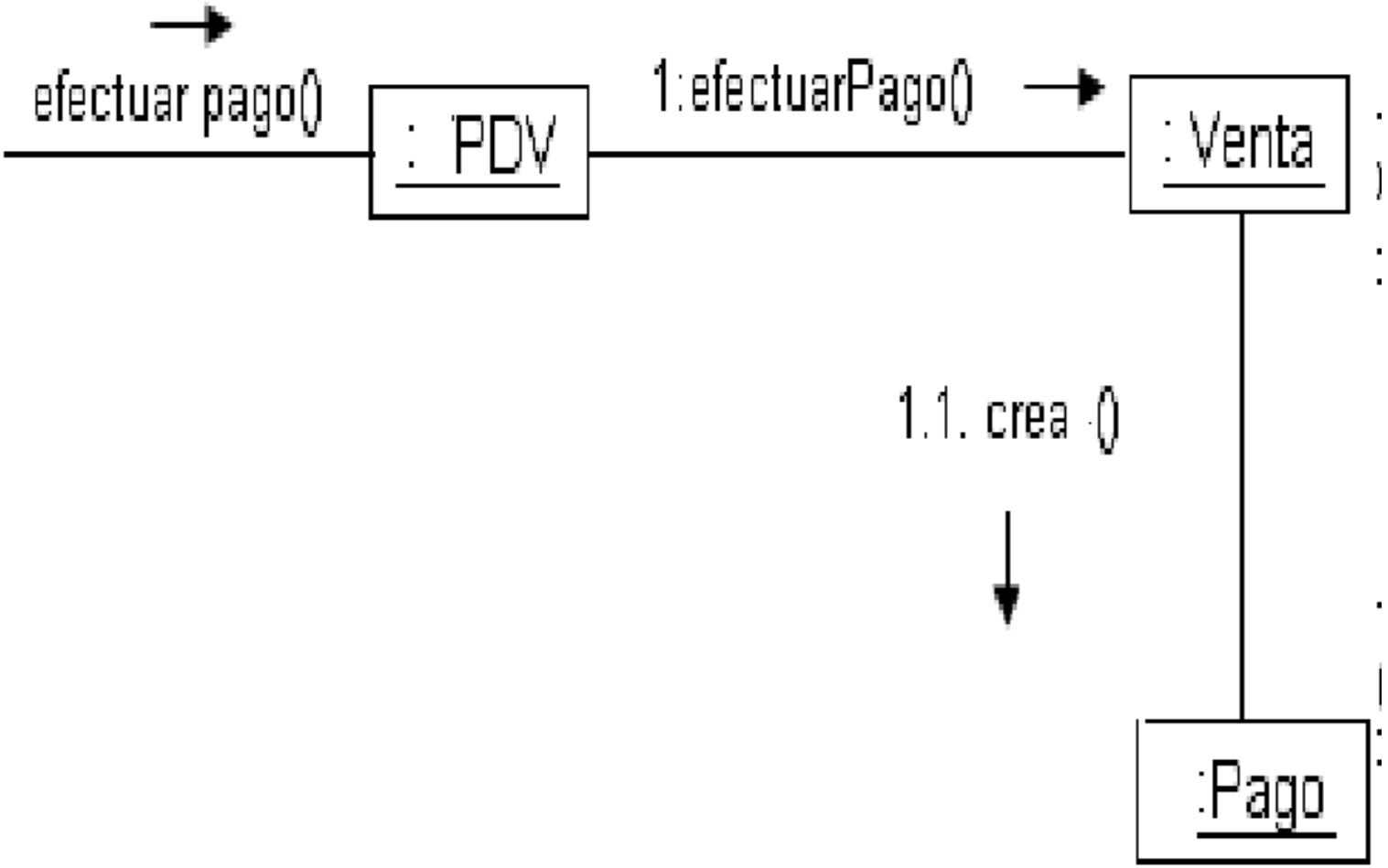
*Clases difíciles de entender;
*Difíciles de reutilizar;
*Difíciles de mantener;
*Delicadas, afectadas por los cambios.

Ejemplos de una baja cohesión son clases que hacen demasiadas cosas.

Ejemplos de buen diseño se producen cuando se crean las clases agrupadas por funcionalidades que son reutilizables.

Ejemplo




efectuar pago()
: PDV
1:efectuarPago()
: Venta
1.1. crea ()
:Pago

Controlador

Es un objeto que no pertenece a la interfaz de usuario, responsable de recibir o manejar un evento del sistema. Un Controlador define el método para las operaciones del sistema.

Asignar la responsabilidad de controlar el flujo de eventos del sistema, a clases específicas.

Esto facilita la centralización de actividades (validaciones, seguridad, etc.) . El controlador no realiza estas actividades, las delega en otras clases con las que mantiene un modelo de alta cohesión.

Un error muy común es asignarle demasiada responsabilidad y alto nivel de acoplamiento con el resto de los componentes del sistema.

En el diseño de clases, cuando no tenemos claro a qué clase pertenece la responsabilidad de realizar una determinada tarea, crear una clase controlador que se llame igual a la tarea a desempeñar.

Tipos de controladores:

- Controlador de Fachada:

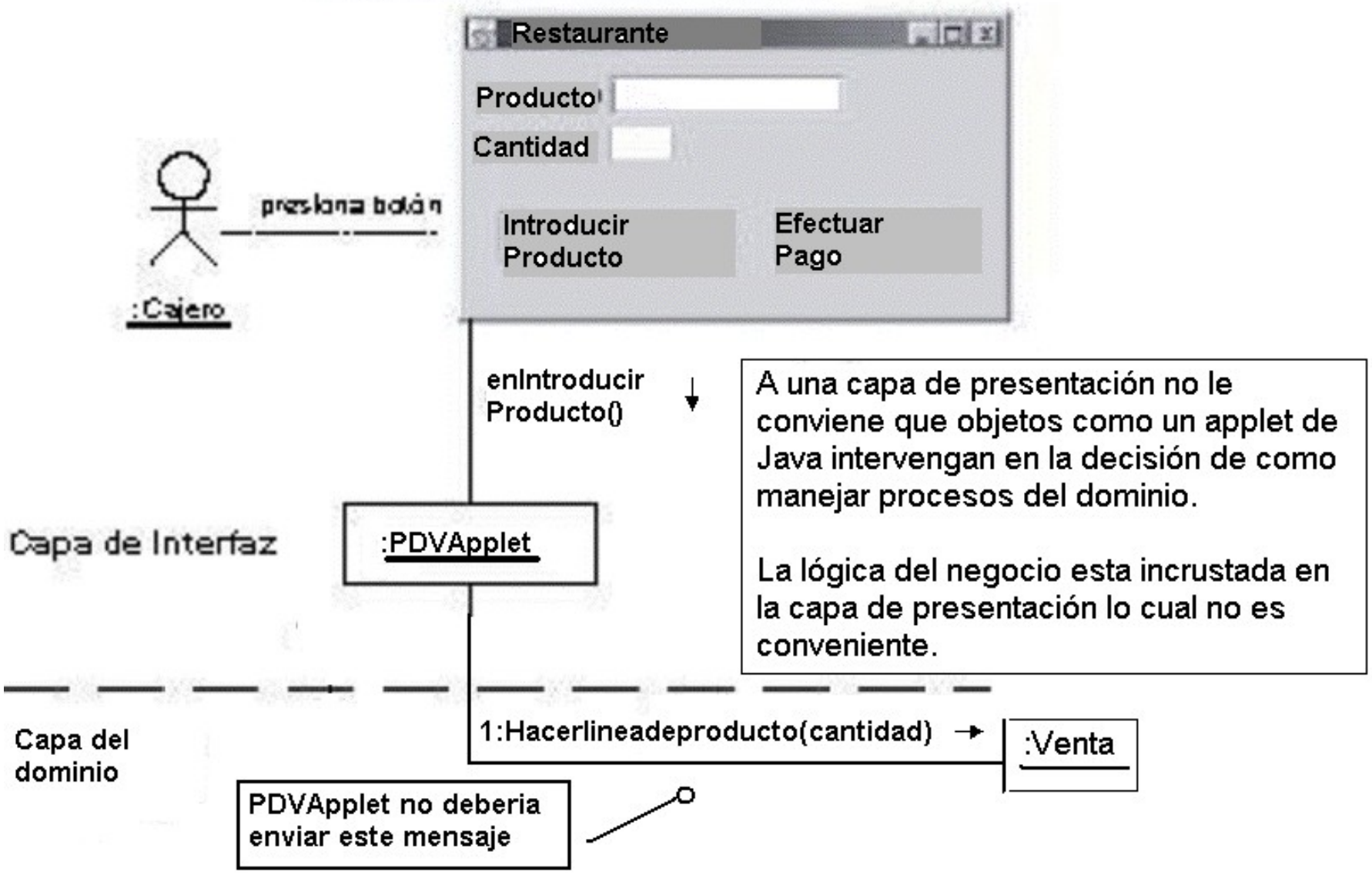
- Controlador de casos de uso:

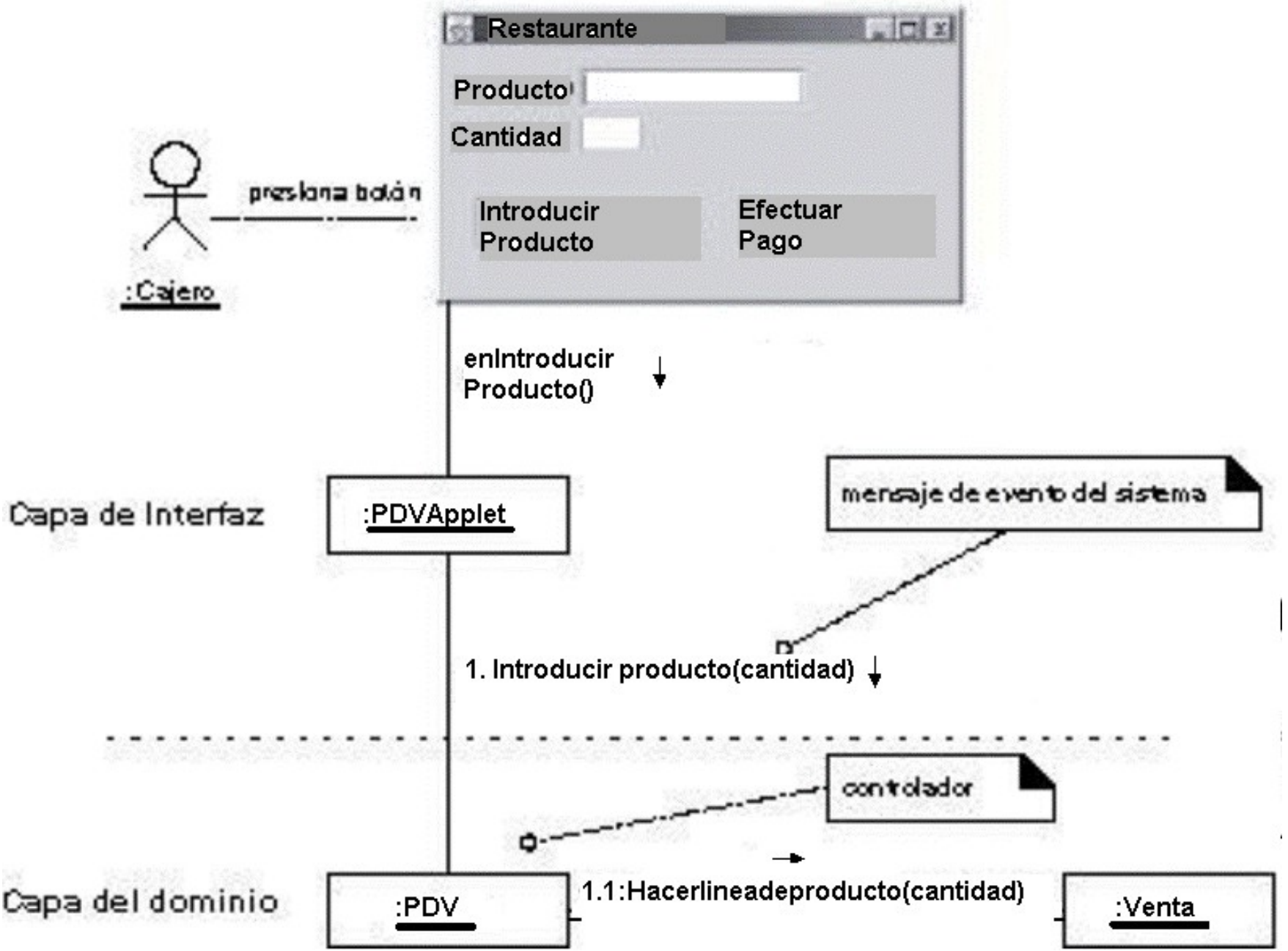
Ventajas:

Aumenta el potencial para reutilizar las interfaces.

Razonamiento sobre el estado (secuencia de pasos) de los casos de uso.

Ejemplo


Restaurante
Producto
Cantidad
Introducir Producto
Efectuar Pago
presiona botón
:Cajero
enIntroducir Producto()
A una capa de presentación no le conviene que objetos como un applet de Java intervengan en la decisión de como manejar procesos del dominio.
La lógica del negocio esta incrustada en la capa de presentación lo cual no es conveniente.
Capa de Interfaz
:PDVApplet
Capa del dominio
1:Hacerlineadeproducto(cantidad)
:Venta
PDVApplet no deberia enviar este mensaje


Restaurante
Producto
Cantidad
Introducir Producto
Efectuar Pago
presiona botón
:Cajero
enIntroducir Producto()
Capa de Interfaz
:PDVApplet
mensaje de evento del sistema
1. Introducir producto(cantidad)
controlador
Capa del dominio
:PDV
1.1:Hacerlineadeproducto(cantidad)
:Venta