18

GRASP: PATRONES PARA ASIGNAR RESPONSABILIDADES

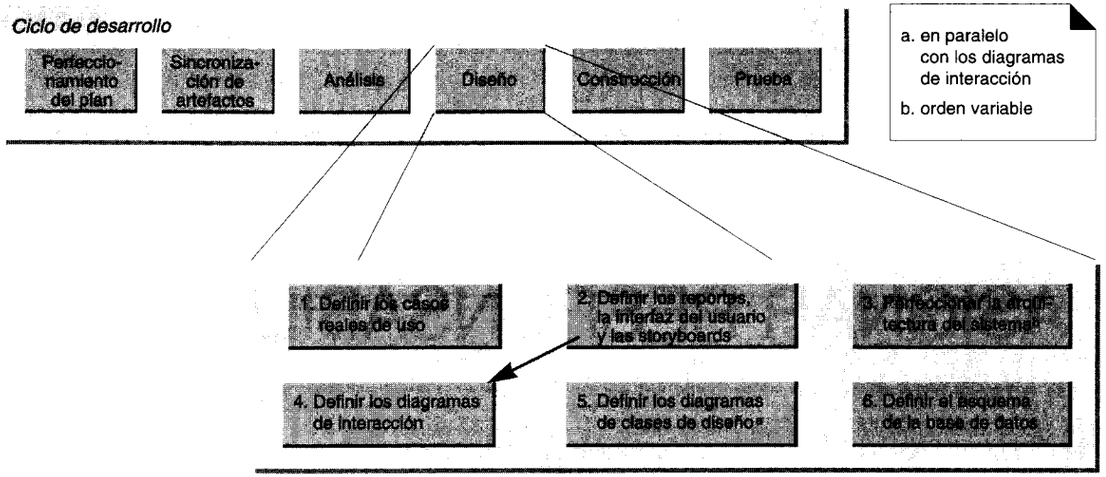
Objetivos

- Definir patrones.
- Aprender a aplicar cinco patrones GRASP.

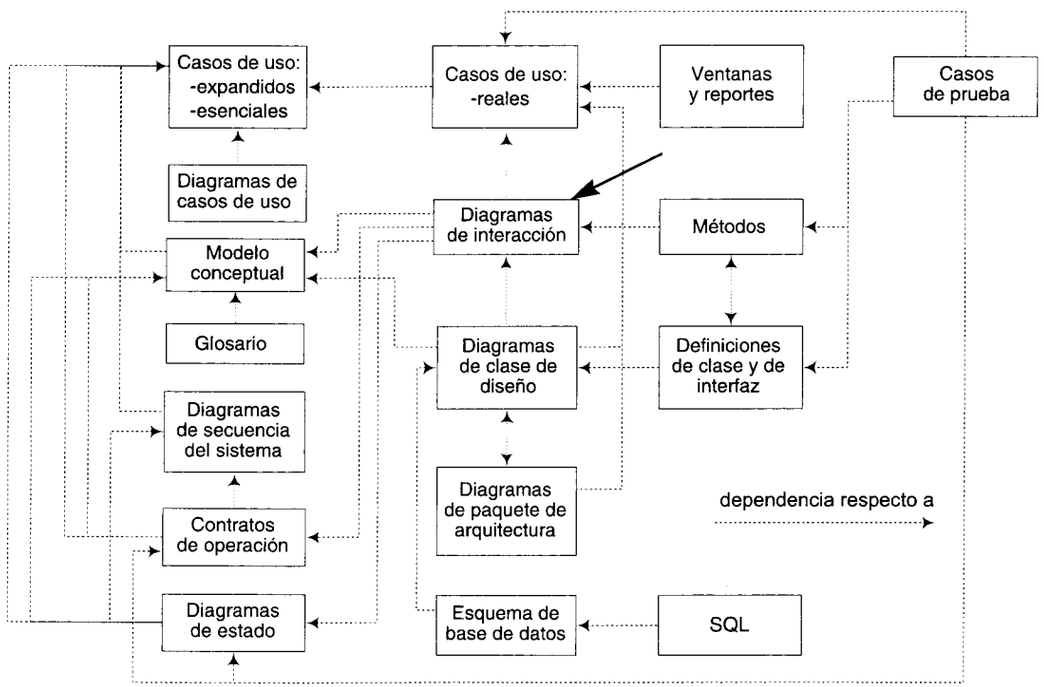
18.1 Introducción

Un sistema orientado a objetos se compone de objetos que envían mensajes a otros objetos para que lleven a cabo las operaciones. En los contratos se incluye una conjetura inicial óptima sobre las responsabilidades y las poscondiciones de las operaciones *inicio, introducirProducto, terminarVenta* y *efectuarPago*. Los diagramas de interacción describen gráficamente la solución —a partir de los objetos en interacción— que estas responsabilidades y poscondiciones satisfacen.

La calidad de diseño de la interacción de los objetos y la asignación de responsabilidades presentan gran variación. Las decisiones poco acertadas dan origen a sistemas y componentes frágiles y difíciles de mantener, entender, reutilizar o extender. Una implementación hábil se funda en los principios cardinales que rigen un buen diseño orientado a objetos. En los patrones GRASP se codifican algunos de ellos, que se aplican al preparar los diagramas de interacción, cuando se asignan las responsabilidades o durante ambas actividades.



Actividades de la fase de diseño dentro de un ciclo de desarrollo.



Dependencias de los artefactos durante la fase de construcción.

18.2 Actividades y dependencias

Los patrones a los que nos referimos se aplican durante la elaboración de los diagramas de interacción, al asignar las responsabilidades a los objetos y al diseñar la colaboración entre ellos.

18.3 Los diagramas de interacción bien diseñados son muy útiles

A continuación repetimos algunos puntos ya expuestos en el capítulo anterior cuando hablamos de los diagramas de colaboración:

- Los diagramas de interacción son algunos de los artefactos más importantes que se preparan en el análisis y diseño orientados a objetos.
- Es muy importante asignar acertadamente las responsabilidades al momento de elaborar los diagramas de interacción.
- El tiempo y el esfuerzo que se dedican a su elaboración, así como un examen riguroso de la asignación de responsabilidades, deberían absorber parte considerable de la fase de diseño de un proyecto.
- Los patrones, principios y expresiones especializadas codificados sirven para mejorar la calidad del diseño.

Los principios del diseño que se requieren para construir buenos diagramas de interacción pueden codificarse, explicarse y utilizarse en forma metódica. Esta manera de entender y usar los principios del diseño se funda en los *patrones con que se asignan las responsabilidades.*

18.4 Responsabilidades y métodos

Booch y Rumbaugh definen la **responsabilidad** como "un contrato u obligación de un tipo o clase" [BJR97]. Las responsabilidades se relacionan con las obligaciones de un objeto respecto a su comportamiento. Esas responsabilidades pertenecen, esencialmente, a las dos categorías siguientes:

1. conocer
2. hacer

Entre las responsabilidades de un objeto relacionadas con **hacer** se encuentran:

- hacer algo en uno mismo
- iniciar una acción en otros objetos

- controlar y coordinar actividades en otros objetos

Entre las responsabilidades de un objeto relacionadas con **conocer** se encuentran:

- estar enterado de los datos privados encapsulados
- estar enterado de la existencia de objetos conexos
- estar enterado de cosas que se puede derivar o calcular

Las responsabilidades se asignan a los objetos durante el diseño orientado a objetos. Por ejemplo, puede declararse que "una *Venta* es responsable de imprimirse ella misma" (un hacer) o que "una *Venta* tiene la obligación de conocer su fecha" (un conocer). Las responsabilidades relacionadas con "conocer" a menudo pueden inferirse del modelo conceptual por los atributos y asociaciones explicadas en él.

La granularidad de la responsabilidad influye en su traducción a clases y métodos. La responsabilidad de "brindar acceso a las bases relacionales de datos" puede incluir docenas de clases y cientos de métodos. En cambio, la de "imprimir una venta" tal vez no incluya más que un método o unos cuantos.

Responsabilidad no es lo mismo que método: los métodos se ponen en práctica para cumplir con las responsabilidades. Éstas se implementan usando métodos que operen solos o en colaboración con otros métodos y objetos. Así, la clase *Venta* podría definir uno o varios métodos que imprimen una instancia *Venta*; digamos el método *imprimir*. Para cumplir con esa responsabilidad, la *Venta* puede colaborar con otros objetos, entre ellos el envío de un mensaje a los objetos *VentasLineadeProducto*, pidiéndoles que se impriman ellos mismos.

18.5 Las responsabilidades y los diagramas de interacción

Este capítulo tiene por objeto ayudarle al lector a aplicar los principios fundamentales que rigen la asignación de responsabilidades a objetos. En los artefactos del UML, las responsabilidades (implementadas como métodos) suelen tenerse en cuenta al momento de preparar los diagramas de interacción, cuya notación ya examinamos en el capítulo anterior.

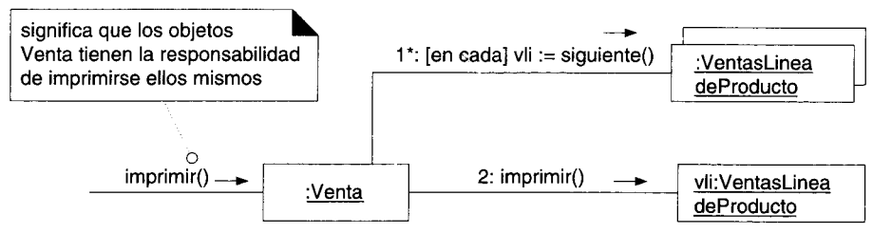


Figura 18.1 Las responsabilidades y los métodos están relacionados.

La figura 18.1 indica que a los objetos *Venta* se les ha asignado la responsabilidad de imprimirse ellos mismos, la cual se llama con un mensaje *imprimir* y se cumple con el método correspondiente *imprimir*. Más aún, para atender esta responsabilidad hay que colaborar con los objetos *VentasLineadeProducto*, pidiéndoles que impriman.

En resumen, los diagramas de interacción muestran las decisiones referentes a la asignación de responsabilidades entre los objetos. Cuando se preparan, se toman decisiones sobre la asignación que se reflejan en los mensajes que son enviados a varias clases de objetos. En el presente capítulo expondremos ampliamente los principios fundamentales —expresados en los patrones GRASP— para guiar las decisiones sobre la asignación de responsabilidades. Esas decisiones se reflejarán después en los diagramas de interacción.

18.6 Patrones

Los diseñadores expertos en orientación a objetos (y también otros diseñadores de software) van formando un amplio repertorio de principios generales y de expresiones que los guían al crear software. A unos y a otras podemos asignarles el nombre de **patrones**, si se codifican en un formato estructurado que describe el problema y su solución, y si se les asigna un nombre. A continuación ofrecemos un ejemplo de patrón.

Nombre del patrón:	Experto.
Solución:	Asignar una responsabilidad a la clase que tiene la información necesaria para cumplirla.
Problema que resuelve:	¿Cuál es el principio fundamental en virtud del cual asignaremos las responsabilidades a los objetos?

En la terminología de objetos, el **patrón** es una descripción de un problema y su solución que recibe un nombre y que puede emplearse en otros contextos; en teoría, indica la manera de utilizarlo en circunstancias diversas.[1] Muchos patrones ofrecen orientación sobre cómo asignar las responsabilidades a los objetos ante determinada categoría de problemas.

> Expresado lo anterior con palabras más simples, el **patrón** es una pareja de problema/solución con un nombre y que es aplicable a otros contextos, con una sugerencia sobre la manera de usarlo en situaciones nuevas.

"El patrón de un individuo puede ser la estructura primitiva de otra persona", máxima con que en la tecnología de objetos se explica la vaguedad de los patrones [GHJV94]. En nuestro estudio de los patrones no abordaremos el tema de lo que conviene llamar patrón; nos centraremos en el valor pragmático de utilizar ese estilo como medio de presentar y recordar los principios tan útiles de la ingeniería del software.

[1] La notación formal de los patrones nació con los patrones arquitectónicos de Christopher Alexander [AIS77]. En los años ochenta, Kent Beck y Ward Cunningham hicieron su aplicación al software [Beck94, Coplien95].

18.6.1 Los patrones no suelen contener ideas nuevas

Los patrones no se proponen descubrir ni expresar nuevos principios de la ingeniería del software. Todo lo contrario: intentan codificar el conocimiento, las expresiones y los principios *ya existentes*: cuanto más trillados y generalizados, tanto mejor. En consecuencia, los patrones GRASP —que describiremos aquí— no introducen ideas novedosas; son una mera codificación de los principios básicos más usados.

18.6.2 Los patrones tienen nombre

En teoría, todos los patrones poseen nombres muy sugestivos. El asignar nombre a un patrón, a un método o a un principio ofrece las siguientes ventajas:

- Apoya el agrupamiento y la incorporación del concepto a nuestro sistema cognitivo y a la memoria.
- Facilita la comunicación.

Darle nombre a una idea compleja —digamos a un patrón— es un ejemplo de la fuerza de la abstracción: convierte una forma compleja en una forma simple con sólo eliminar los detalles. Por tanto, los patrones GRASP poseen nombres concisos como *Experto, Creador, Controlador.*

18.6.3 La asignación de nombre a los patrones mejora la comunicación

Cuando se le da nombre a un patrón, un simple nombre nos permite discutir con otros un principio en todos sus aspectos. Considere la siguiente conversación entre dos diseñadores de software que emplean una nomenclatura común de patrones (*Experto, Separacion Vista-Modelo* y otros términos afines) para tomar una decisión sobre un diseño:

Alfredo: "En tu opinión, ¿a qué objeto deberíamos asignar la responsabilidad de imprimir una *Venta*? Creo que convendría asignarla a *Separacion Vista-Modelo*. ¿Qué te parece una *VistadeReportedeVentas?*"

Teresa: "En mi opinión, *Experto* es una mejor solución por ser una impresión simple y porque Ventas tiene toda la información que se requiere en salida impresa. Creo que le asignaremos esta responsabilidad a la *Venta.*"

Alfredo: "Está bien, hagámoslo."

Agrupar las expresiones especializadas y los principios del diseño con nombres de uso común facilita la comunicación y le confiere a la búsqueda un nivel de abstracción más alto.

18.7 GRASP: patrones de los principios generales para asignar responsabilidades

Resumimos a continuación la introducción anterior:

- Asignar correctamente las responsabilidades es muy importante en el diseño orientado a objetos.
- La asignación de responsabilidades a menudo se asignan en el momento de preparar los diagramas de interacción.
- Los patrones son parejas de problema/solución con un nombre, que codifican buenos principios y sugerencias relacionados frecuentemente con la asignación de responsabilidades.

Dicho esto, podemos proceder a examinar los patrones GRASP.

Pregunta:	¿Qué son los patrones GRASP?
Respuesta:	Los patrones GRASP describen los principios fundamentales de la asignación de responsabilidades a objetos, expresados en forma de patrones.

Es importante entender y poder aplicar estos principios durante la preparación de un diagrama de interacción, pues un diseñador de software sin mucha experiencia en la tecnología de objetos debe dominarlos cuanto antes: constituyen el fundamento de cómo se diseñará el sistema.

GRASP es un acrónimo que significa **G**eneral **R**esponsibility **A**signment **S**oftware **P**atterns (patrones generales de software para asignar responsabilidades).[1] El nombre se eligió para indicar la importancia de *captar* (*grasping*) estos principios, si se quiere diseñar eficazmente el software orientado a objetos.

18.7.1 Cómo aplicar los patrones GRASP

En las siguientes secciones explicaremos los primeros cinco patrones de GRASP:

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

[1] Desde el punto de vista técnico, deberíamos escribir "Patrones GRAS" en vez de "Patrones GRASP" pero la segunda expresión suena mejor en inglés, idioma en que fue acuñado el término.

Hay otros patrones de los que nos ocuparemos en un capítulo posterior, pero vale la pena dominar los cinco anteriores porque se refieren a cuestiones y a aspectos fundamentales del diseño.

Por favor, estudie los patrones que siguen; observe cómo se emplean en los diagramas muestra de interacción y luego aplíquelos al preparar otros diagramas. Comience dominando *Experto, Creador, Controlador, Alta Cohesión* y *Bajo Acoplamiento*. Más tarde irá aprendiendo los patrones restantes.

18.8 La notación del UML para los diagramas de clase

Los diagramas de clase en UML presentan las clases de software en contraste con los conceptos de dominio. La casilla de clase consta de tres secciones; la tercera contiene los métodos de la clase, como se aprecia en la figura 18.2.

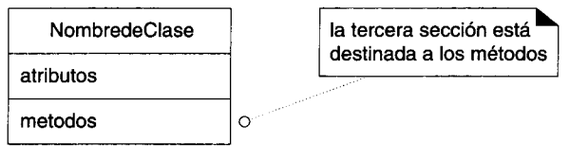


Figura 18.2 Las clases del software muestran los nombres de los métodos.

Los detalles de esta notación se abordan en un capítulo ulterior. En la siguiente exposición sobre los patrones, utilizaremos esporádicamente esta modalidad de la casilla de clase.

18.9 Experto

Solución **Asignar una responsabilidad al experto en información: la clase que cuenta con la información necesaria para cumplir la responsabilidad.**

Problema ¿Cuál es el principio fundamental en virtud del cual se asignan las responsabilidades en el diseño orientado a objetos?

Un modelo de clase puede definir docenas y hasta cientos de clases de software, y una aplicación tal vez requiera el cumplimiento de cientos o miles de responsabilidades. Durante el diseño orientado a objetos, cuando se definen las interacciones entre los objetos, tomamos decisiones sobre la asignación de responsabilidades a las clases. Si se hacen en forma adecuada, los sistemas tienden a ser más fáciles de entender, mantener y ampliar, y se nos presenta la oportunidad de reutilizar los componentes en futuras aplicaciones.

Ejemplo En la aplicación del punto de venta, alguna clase necesita conocer el gran total de la venta.

> Comience asignando las responsabilidades con una definición clara de ellas.

A partir de esta recomendación se plantea la pregunta:

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

Desde el punto de vista del patrón Experto, deberíamos buscar la clase de objetos que posee la información necesaria para calcular el total. Examinemos detenidamente el modelo conceptual parcial de la figura 18.3.

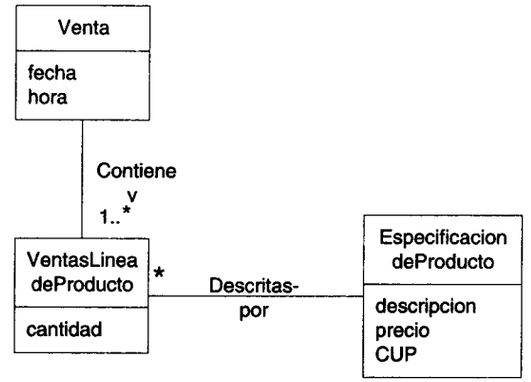


Figura 18.3 Asociaciones de la Venta.

¿Qué información hace falta para calcular el gran total? Hay que conocer todas las instancias *VentasLineadeProducto* de una venta y la suma de sus subtotales. Y esto lo

conoce únicamente la instancia *Venta*; por tanto, desde el punto de vista del Experto, *Venta* es la clase correcta de objeto para asumir esta responsabilidad; es el *experto en información.*

Como se señaló con anterioridad, es dentro del contexto de la preparación de los diagramas de interacción (por ejemplo, los de colaboración) donde surgen estas cuestiones concernientes a la responsabilidad. Imagine que vamos a empezar a dibujar diagramas para asignar responsabilidades a los objetos. El diagrama parcial de colaboración y el de clases de la figura 18.4 describen gráficamente las decisiones que hemos adoptado hasta ahora.

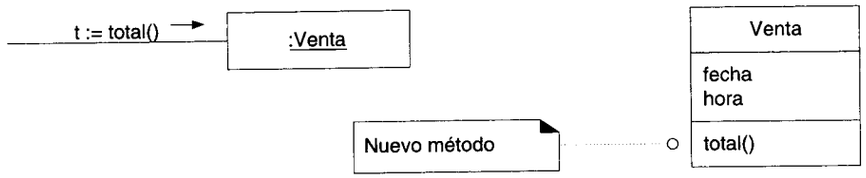


Figura 18.4 Diagrama parcial de colaboración.

Todavía no terminamos. ¿Qué información hace falta para determinar el subtotal de la línea de productos? Se necesitan *VentasLineadeProducto.cantidad* y *EspecificaciondeProducto.precio*. *VentasLineadeProducto* conoce su cantidad y su correspondiente *EspecificaciondeProducto*; por tanto, desde la perspectiva de patrón Experto, *VentasLineadeProducto* debería calcular el subtotal; es el *experto en información.*

Por lo que respecta a un diagrama de colaboración, lo anterior significa que la *Venta* necesita enviar mensajes de *subtotal* a cada *VentasLineadeProducto* y sumar los resultados; el diseño en cuestión se incluye en la figura 18.5.

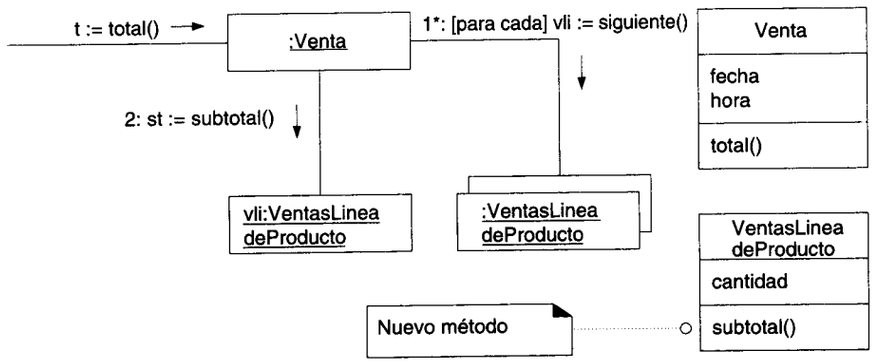


Figura 18.5 Cálculo del total de la venta.

VentasLineadeProducto no puede cumplir la responsabilidad de conocer y dar el subtotal, si no conoce el precio del producto. *EspecificaciondeProducto* es un Experto en información para contestar su precio; por tanto, habrá que enviarle un mensaje preguntándole el precio. El diseño correspondiente aparece en la figura 18.6.

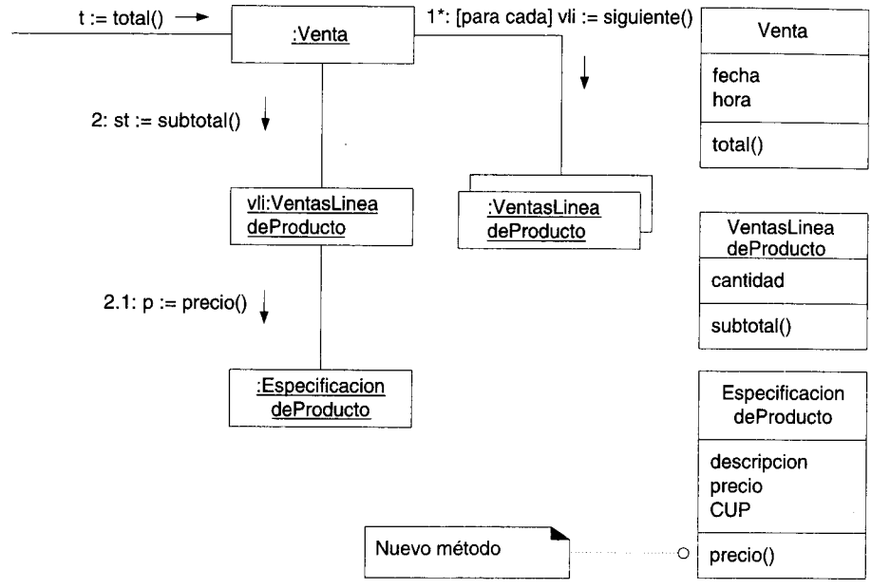


Figura 18.6 Cálculo del total de Venta.

En conclusión, para cumplir con la responsabilidad de conocer y dar el total de la venta, se asignaron tres responsabilidades a las tres clases de objeto así:

Clase	Responsabilidad
Venta	conoce el total de la venta
VentasLineadeProducto	conoce el subtotal de la línea de producto
EspecificaciondeProducto	conoce el precio del producto

El contexto donde las responsabilidades se consideraron y se decidieron fue el dibujo de un diagrama de colaboración. Después, la sección dedicada a métodos en un diagrama de clases puede resumir los métodos.

El principio en virtud del cual se asignaron las responsabilidades fue el patrón Experto: lo colocamos junto con el objeto que posee la información necesaria para cumplirlas.

Explicación Experto es un patrón que se usa más que cualquier otro al asignar responsabilidades; es un principio básico que suele utilizarse en el diseño orientado a objetos. Con él no se pretende designar una idea oscura ni extraña; expresa simplemente la "intuición" de que los objetos hacen cosas relacionadas con la información que poseen.

Nótese que el cumplimiento de una responsabilidad requiere a menudo información distribuida en varias clases de objetos. Ello significa que hay muchos expertos "parciales" que colaboraron en la tarea. Por ejemplo, el problema del total de la venta exigirá finalmente la colaboración de tres clases. Siempre que la información se encuentre esparcida en varios objetos, éstos habrán de interactuar a través de mensajes para compartir el trabajo.

El patrón Experto da origen a diseño donde el objeto de software realiza las operaciones que normalmente se aplican a la cosa real que representa: a esto Peter Coad lo llama estrategia de "Hacerlo yo mismo" [Coad95]. Por ejemplo, en el mundo real una venta no nos indica su total sin ayuda de aparatos electromecánicos; se trata de un concepto inanimado. Alguien calcula el total de la venta. Pero en el terreno del software orientado a objetos, todos los objetos del software están "vivos" o "animados", y pueden asumir responsabilidades y hacer cosas. Básicamente, hacen cosas relacionadas con la información de que disponen. A esto yo lo llamo principio de "Animación" en el diseño orientado a objetos; es como estar en una caricatura donde todo tiene vida.

El patrón Experto —como tantas otras cosas en la tecnología de objetos— ofrece una analogía con el mundo real. Acostumbramos asignar responsabilidad a individuos que disponen de la información necesaria para llevar a cabo una tarea. Por ejemplo, en una empresa ¿quién será el encargado de preparar un estado de pérdidas y ganancias? El empleado que tiene acceso a toda la información necesaria para elaborarlo: quizá el director financiero. Y del mismo modo que los objetos del software colaboran porque la información está esparcida, lo mismo sucede con el personal de una empresa. El director financiero podrá pedir a los encargados de cuentas por cobrar y de cuentas por pagar que generen reportes individuales sobre créditos y deudas.

Beneficios

- Se conserva el encapsulamiento, ya que los objetos se valen de su propia información para hacer lo que se les pide. Esto soporta un **bajo acoplamiento**, lo que favorece al hecho de tener sistemas más robustos y de fácil mantenimiento. (Bajo Acoplamiento es un patrón GRASP que examinaremos más adelante.)
- El comportamiento se distribuye entre las clases que cuentan con la información requerida, alentando con ello definiciones de clase "sencillas" y más cohesivas que son más fáciles de comprender y de mantener. Así se brinda soporte a una **alta cohesión** (patrón del que nos ocuparemos luego).

Otras formas de designar este patrón "Juntar responsabilidades y la información", "Lo hace el que conoce", "Animación", "Lo hago yo mismo", "Unir los servicios a los atributos sobre los que operan".

18.10 Creador

Solución **Asignarle a la clase B la responsabilidad de crear una instancia de clase A en uno de los siguientes casos:**

- **B *agrega* los objetos A.**
- **B *contiene* los objetos A.**
- **B *registra* las instancias de los objetos A.**
- **B *utiliza* específicamente los objetos A.**
- **B *tiene los datos de inicialización* que serán transmitidos a A cuando este objeto sea creado (así que B es un Experto respecto a la creación de A).**

B es un *creador* de los objetos A.

Si existe más de una opción, prefiera la clase B que *agregue* o *contenga* la clase A.

Problema ¿Quién debería ser responsable de crear una nueva instancia de alguna clase?

La creación de objetos es una de las actividades más frecuentes en un sistema orientado a objetos. En consecuencia, conviene contar con un principio general para asignar las responsabilidades concernientes a ella. El diseño, bien asignado, puede soportar un bajo acoplamiento, una mayor claridad, el encapsulamiento y la reutilizabilidad.

Ejemplo En la aplicación del punto de venta, ¿quién debería encargarse de crear una instancia *VentasLineadeProducto*? Desde el punto de vista del patrón Creador, deberíamos buscar una clase que agregue, contenga y realice otras operaciones sobre este tipo de instancias. Examine atentamente el modelo conceptual parcial de la figura 18.7.

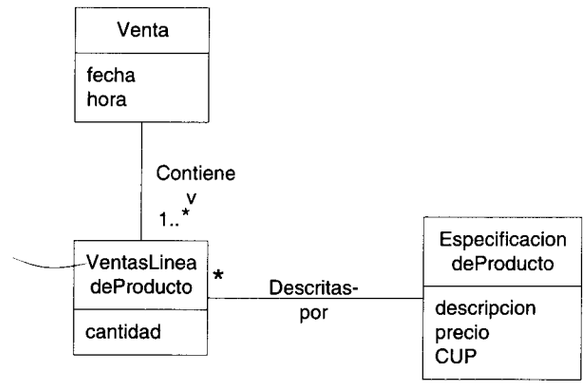


Figura 18.7 Modelo conceptual parcial.

Una *Venta* contiene (en realidad, agrega) muchos objetos *VentasLineadeProducto*; por ello, el patrón Creador sugiere que *Venta* es idónea para asumir la responsabilidad de crear las instancias VentasLineadeProducto.

Lo anterior nos permite diseñar las interacciones de los objetos en un diagrama de colaboración, según se observa en la figura 18.8.

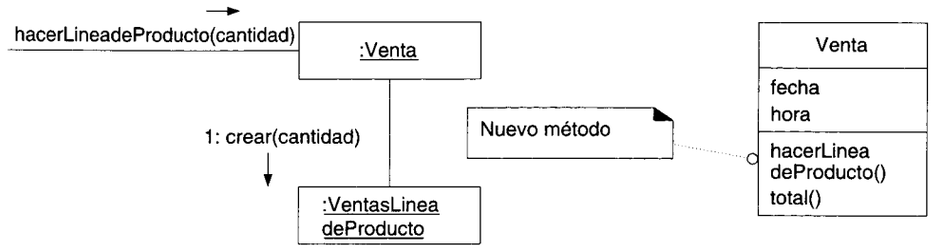


Figura 18.8 Creación de un objeto VentasLineadeProducto.

Esta asignación de responsabilidades requiere definir en *Venta* un método de *hacerLineadeProducto.*

Una vez más, el contexto donde se consideraron y se asignaron estas responsabilidades fue el momento de dibujar un diagrama de colaboración. Después, en la sección destinada a métodos en un diagrama de clases, podemos resumir los resultados de la asignación de responsabilidades, realizadas concretamente como métodos.

Explicación El patrón Creador guía la asignación de responsabilidades relacionadas con la creación de objetos, tarea muy frecuente en los sistemas orientados a objetos. El propósito fundamental de este patrón es encontrar un creador que debemos conectar con el objeto producido en cualquier evento. Al escogerlo como creador, se da soporte al bajo acoplamiento.

El Agregado *agrega* la Parte, el Contenedor *contiene* el contenido, el Registro *registra*. En un diagrama de clases se registran las relaciones muy frecuentes entre las clases. El patrón Creador indica que la clase incluyente del contenedor o registro es idónea para asumir la responsabilidad de crear la cosa contenida o registrada. Desde luego, se trata tan sólo de una directriz.

Nótese que el concepto de **agregación** se utilizó al examinar el patrón Creador. Es un tema que trataremos más ampliamente en un capítulo posterior; pero damos aquí una definición sucinta: la agregación incluye cosas que están en una sólida relación de parte-todo o de parte-estructura; por ejemplo, Cuerpo agrega Pierna y Párrafo agrega oración.

En ocasiones encontramos un patrón creador buscando la clase con los datos de inicialización que serán transferidos durante la creación. Éste es en realidad un ejemplo del patrón Experto. Los datos de inicialización se transmiten durante la creación a través de algún método de inicialización, como un constructor en Java que cuenta con parámetros.

Suponga, por ejemplo, que una instancia *Pago* al momento de ser creada necesita inicializarse con el total *Venta*. Puesto que *Venta* conoce el total, es un buen candidato para ser el parámetro creador de *Pago.*

Beneficios

- Se brinda soporte a un **bajo acoplamiento** (que describiremos más adelante), lo cual supone menos dependencias respecto al mantenimiento y mejores oportunidades de reutilización. Es probable que el acoplamiento no aumente, pues la clase *creada* tiende a ser visible a la clase *creador*, debido a las asociaciones actuales que nos llevaron a elegirla como el parámetro adecuado.

Patrones Conexos

- Bajo acoplamiento.
- Parte-todo [BMRSS96] describe un patrón para definir los objetos que soportan el encapsulamiento de sus componentes.

18.11 Bajo acoplamiento

Solución **Asignar una responsabilidad para mantener bajo acoplamiento.**

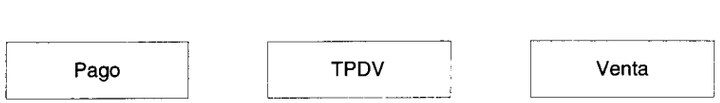
Problema ¿Cómo dar soporte a una dependencia escasa y a un aumento de la reutilización?

El **acoplamiento** es una medida de la fuerza con que una clase está conectada a otras clases, con que las conoce y con que recurre a ellas. Una clase con bajo (o débil) acoplamiento no depende de muchas otras; "muchas otras" depende del contexto, pero no lo estudiaremos aquí por el momento.

Una clase con alto (o fuerte) acoplamiento recurre a muchas otras. Este tipo de clases no es conveniente: presentan los siguientes problemas:

- Los cambios de las clases afines ocasionan cambios locales.
- Son más difíciles de entender cuando están aisladas.
- Son más difíciles de reutilizar porque se requiere la presencia de otras clases de las que dependen.

Ejemplo Examine con mucha atención el siguiente diagrama parcial de clases desde la perspectiva de la aplicación a la terminal del punto de venta:



Suponga que necesitamos crear una instancia *Pago* y asociarla a *Venta*. ¿Qué clase se encargará de hacer esto? Puesto que una instancia *TPDV* "registra" un *Pago* en el dominio del mundo real, el patrón Creador indica que *TPDV* es un buen candidato para producir el *Pago*. La instancia *TPDV* podría entonces enviarle a *Venta* el mensaje *agregarPago*, transmitiendo al mismo tiempo el nuevo *Pago* como parámetro. En la figura 18.9 se incluye un posible diagrama parcial de colaboración que representa gráficamente esto.

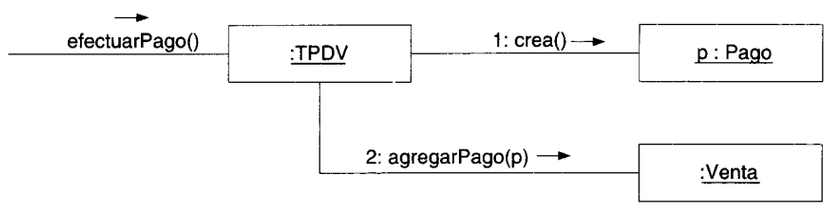


Figura 18.9 TPDV crea Pago.

Esta asignación de responsabilidades acopla la clase *TPDV* al conocimiento de la clase *Pago*. En la figura 18.10 se muestra una solución alterna para crear el *Pago* y asociarlo a la *Venta*.

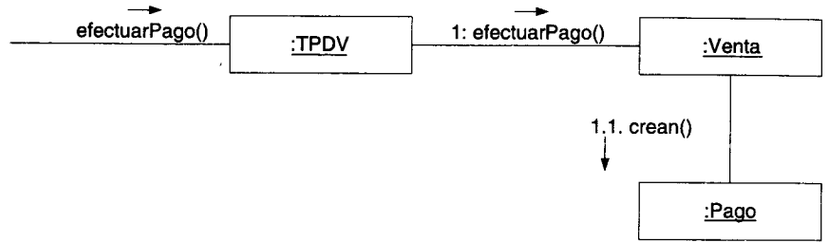


Figura 18.10 Venta crea Pago.

¿Qué diseño, basado en la asignación de responsabilidades, brinda soporte al patrón Bajo Acoplamiento? En ambos casos suponemos que *Venta* terminará acoplándose al conocimiento de un *Pago*. El diseño 1, donde la instancia *TPDV* crea *Pago*, incorpora el acoplamiento de *TPDV* a *Pago*; en cambio, el diseño 2, donde la *Venta* realiza la creación de un *Pago*, no incrementa el acoplamiento. Tomando como única perspectiva la del acoplamiento, el diseño 2 es preferible porque se conserva un menor acoplamiento global. Éste es un ejemplo donde dos patrones —Bajo Acoplamiento y Creador— pueden sugerir soluciones distintas. En la práctica, el grado de acoplamiento no puede considerarse aisladamente de otros principios como Experto y Alta Cohesión. Sin embargo, es un factor a considerar cuando se intente mejorar un diseño.

Explicación El Bajo Acoplamiento es un principio que debemos recordar durante las decisiones de diseño: es la meta principal que es preciso tener presente siempre. Es un *patrón evaluativo* que el diseñador aplica al juzgar sus decisiones de diseño.

En los lenguajes orientados a objetos como C++, Java y Smalltalk, las formas comunes de acoplamiento de *TipoX* a *TipoY* son las siguientes:

- *TipoX* posee un atributo (miembro de datos o variable de instancia) que se refiere a una instancia *TipoY* o al propio *TipoY*.
- *TipoX* tiene un método que a toda costa referencia una instancia de *TipoY* o incluso el propio *TipoY*. Suele incluirse un parámetro o una variable local de *TipoY* o bien el objeto devuelto de un mensaje es una instancia de *TipoY*.
- *TipoX* es una subclase directa o indirecta de *TipoY*.
- *TipoY* es una interfaz y *TipoX* la implementa.

El Bajo Acoplamiento estimula asignar una responsabilidad de modo que su colocación no incremente el acoplamiento tanto que produzca los resultados negativos propios de un alto acoplamiento.

El Bajo Acoplamiento soporta el diseño de clases más independientes, que reducen el impacto de los cambios, y también más reutilizables, que acrecientan la oportunidad de una mayor productividad. No puede considerarse en forma independiente

de otros patrones como 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, si no 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. Por ejemplo, a veces se dedica demasiado tiempo a la obtención de componentes reutizables en futuros proyectos "utópicos", a pesar de que no se sabe con certeza que se necesitarán. Con ello no queremos decir que se pierde tiempo al tratar de reutilizarlos, sólo que han de atenderse las consideraciones de costo-beneficio.

Una subclase está acoplada firmemente con su superclase. La decisión de derivarla de una superclase ha de ser estudiada con mucho cuidado, pues es una forma muy sólida de acoplamiento. Supongamos, por ejemplo, que es necesario guardar persistentemente los objetos en una base de datos relacional o de objetos. En este caso, está bastante generalizado (aunque no se recomienda) el diseño para crear una superclase abstracta denominada *ObjetoPersistente* del cual se derivan otras clases. Esta subclase tiene la desventaja de acoplar estrechamente objetos del dominio con un servicio en particular y la ventaja de una herencia automática del comportamiento de persistencia, una especie de "matrimonio por conveniencia". Y esto rara vez es una buena decisión en las relaciones.

No existe una medida absoluta de cuándo el acoplamiento es excesivo. Lo importante es que el diseñador pueda determinar el grado actual de acoplamiento y si surgirán problemas en caso de incrementarlo. En términos generales, han de tener escaso acoplamiento las clases muy genéricas y con grandes probabilidades de reutilización.

El caso extremo de Bajo Acoplamiento ocurre cuando existe poco o nulo acoplamiento entre las clases. Ello no conviene porque una metáfora esencial en la tecnología de objetos es un sistema de objetos conectados que se comunican entre sí a través de mensajes. Si el Bajo Acoplamiento se lleva a los extremos, dará origen a un diseño deficiente por producir objetos incoherentes, atiborrados y complejos que hacen todo el trabajo, con muchos otros objetos muy pasivos y de acoplamiento cero que funcionan como meros depósitos de datos. Un grado moderado de acoplamiento entre las clases es normal y necesario si quiere crearse un sistema orientado a objetos, donde las tareas se realicen por colaboración entre objetos conectados.

Beneficios

- no se afectan por cambios de otros componentes
- fáciles de entender por separado
- fáciles de reutilizar

18.12 Alta cohesión

Solución **Asignar una responsabilidad de modo que la cohesión siga siendo alta.**

Problema ¿Cómo mantener la complejidad dentro de límites manejables?

En la perspectiva del diseño orientado a objetos, la **cohesión** (o, más exactamente, la cohesión funcional) es una medida de cuán relacionadas y enfocadas están las responsabilidades de una clase. Una alta cohesión caracteriza a las clases con responsabilidades estrechamente relacionadas que no realicen un trabajo enorme.

Una clase con baja cohesión hace muchas cosas no afines o un trabajo excesivo. No conviene este tipo de clases pues presentan los siguientes problemas:

- son difíciles de comprender
- son difíciles de reutilizar
- son difíciles de conservar
- son delicadas: las afectan constantemente los cambios

Las clases con baja cohesión a menudo representan un alto grado de abstracción o han asumido responsabilidades que deberían haber delegado a otros objetos.

Ejemplo El mismo ejemplo que hemos utilizado en el patrón Bajo Acoplamiento puede analizarse en el caso de Alta Cohesión.

Suponga que necesitamos crear una instancia de *Pago* (en efectivo) y asociarla a la *Venta*. ¿Qué clase será la responsable de hacerlo? Como *TPDV* registra un *Pago* en el dominio del mundo real, el patrón Creador sugiere que *TPDV* es un buen candidato para que genere *Pago*. Entonces la instancia *TPDV* podría enviar a *Venta* un mensaje *agregarPago*, transmitiendo al mismo tiempo el nuevo *Pago* como parámetro, según se muestra en la figura 18.11.

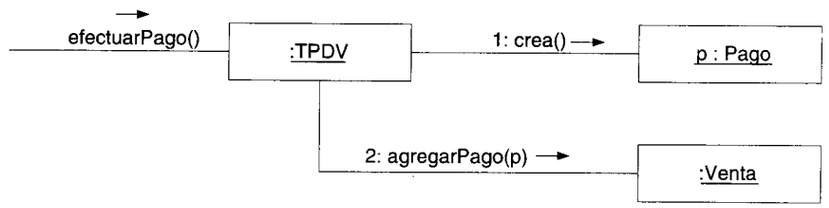


Figura 18.11 TPDV crea Pago.

Esta asignación de responsabilidades coloca en la clase *TPDV* la realización del pago. *TPDV* asume parte de la responsabilidad de realizar la operación del sistema *efectuarPago*.

En este ejemplo aislado, lo anterior es aceptable. Pero si seguimos haciendo que la clase *TPDV* se encargue de efectuar algún trabajo o la mayor parte del que se relaciona con un número creciente de operaciones del sistema, se irá saturando con tareas y terminará por perder la cohesión.

Imagine que el sistema tuviera 50 operaciones, todas ellas recibidas por la clase *TPDV*. Si lleva a cabo el trabajo relacionada con cada una, con el tiempo el sistema será una objeto "saturado" y sin cohesión. Lo importante no es que esta tarea individual de creación de *Pago* haga que *TPDV* pierda su cohesión, sino que sea parte de un panorama más general de la asignación global de responsabilidades y pueda indicar la tendencia a una menor cohesión.

En cambio, como se advierte en la figura 18.12, el segundo diseño delega a la *Venta* la responsabilidad de crear el pago, que soporta una mayor cohesión de *TPDV*. El segundo diseño brinda soporte a una alta cohesión y a un bajo acoplamiento; de ahí que sea conveniente.

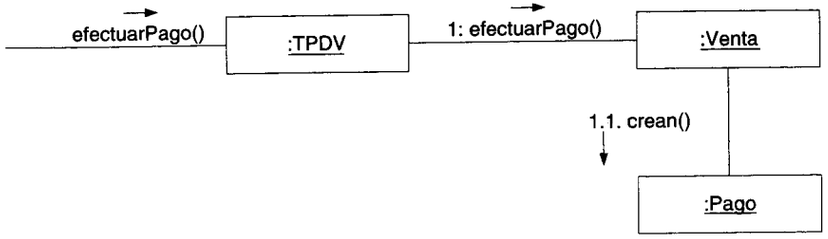


Figura 18.12 Venta crea Pago.

En la práctica, el nivel de cohesión no puede ser considerado independientemente de otras responsabilidades ni de otros principios como los patrones Experto y Bajo Acoplamiento.

Explicación Como el patrón Bajo Acoplamiento, también Alta Cohesión es un principio que debemos tener presente en todas las decisiones de diseño: es la meta principal que ha de buscarse en todo momento. Es un patrón evaluativo que el desarrollador aplica al valorar sus decisiones de diseño.

Grady Booch señala que se da una alta cohesión funcional cuando los elementos de un componente (clase, por ejemplo) "colaboran para producir algún comportamiento bien definido" [Booch94].

A continuación se mencionan algunos escenarios que ejemplifican los diversos grados de la cohesión funcional:

1. *Muy baja cohesión.* Una clase es la única responsable de muchas cosas en áreas funcionales muy heterogéneas.
    - Suponga que existe una clase llamada *RDB-RPC-Interfaz*, la única encargada de interactuar con las bases relacionales de datos y de atender las llamadas de procedimientos remotas. Éstas son dos áreas funcionales radicalmente distintas, y cada una requiere mucho código de soporte. Las responsabilidades deberían repartirse en una familia de clases relacionada con el acceso RDB y en otra familia relacionada con el soporte RPC.

2. *Baja cohesión.* Una clase tiene la responsabilidad exclusiva de una tarea compleja dentro de un área funcional.
    - Suponga que existe una clase llamada *RDBInterfaz*, la única encargada de interactuar con bases relacionales de datos. Los métodos de una clase están todos relacionados, pero hay muchos de ellos y una enorme cantidad de código de soporte; puede haber cientos o miles de métodos. La clase debería dividirse en una familia de clases ligeras que compartan el trabajo para ofrecer el acceso RDB.
3. *Alta cohesión.* Una clase tiene responsabilidades moderadas en un área funcional y colabora con las otras para llevar a cabo las tareas.
    - Suponga que existe una clase denominada *RDBInterfaz*, que se encarga de una parte de la interacción con bases relacionales de datos. Interactúa con una docena de clases afines mediante el acceso RDB para recuperar objetos y guardarlos.
4. *Cohesión moderada.* Una clase tiene un peso ligero y responsabilidades exclusivas en unas cuantas áreas que están relacionadas lógicamente con el concepto de clase, pero no entre ellas.
    - Suponga que existe una clase denominada *Compañía*, la única responsable de: a) conocer a sus empleados b) conocer la información financiera. Estas dos áreas no están estrechamente relacionadas entre ellas, aunque lógicamente lo están con el concepto de compañía. Además, el número total de métodos públicos es pequeño, lo mismo que el código de soporte.

Una regla práctica es la siguiente: una clase de alta cohesión posee un número relativamente pequeño, con una importante funcionalidad relacionada y poco trabajo por hacer. Colabora con otros objetos para compartir el esfuerzo si la tarea es grande.

Una clase con mucha cohesión es útil porque es bastante fácil darle mantenimiento, entenderla y reutilizarla. Su alto grado de funcionalidad, combinada con una reducida cantidad de operaciones, también simplifica el mantenimiento y los mejoramientos. La ventaja que significa una gran funcionalidad también soporta un aumento de la capacidad de reutilización.

El patrón Alta Cohesión —como tantas otras cosas en la tecnología de objetos— presenta semejanzas con el mundo real. Todos sabemos que, si alguien asume demasiadas responsabilidades —sobre todo las que debería delegar—, no será eficiente. Esto se observa en algunos gerentes que no han aprendido a delegar. Muestran baja cohesión; prácticamente ya están "desligados".

Beneficios

- Mejoran la claridad y la facilidad con que se entiende el diseño.
- Se simplifican el mantenimiento y las mejoras en funcionalidad.
- A menudo se genera un bajo acoplamiento.
- La ventaja de una gran funcionalidad soporta una mayor capacidad de reutilización, porque una clase muy cohesiva puede destinarse a un propósito muy específico.

18.13 Controlador

Solución **Asignar la responsabilidad del manejo de un mensaje de los eventos de un sistema a una clase que represente una de las siguientes opciones:**

- **el "sistema" global (*controlador de fachada*).**
- **la empresa u organización global (*controlador de fachada*).**
- **algo en el mundo real que es activo (por ejemplo, el papel de una persona) y que pueda participar en la tarea (*controlador de tareas*).**
- **un manejador artificial de todos los eventos del sistema de un caso de uso, generalmente denominados "Manejador<NombreCasodeUso>" (*controlador de casos de uso*).**

Utilice la misma clase de controlador con todos los eventos del sistema en el mismo caso de uso.

Corolario: Nótese que en esta lista no figuran las clases "ventana", "aplicación", "vista" ni "documento". Estas clases *no* deberían ejecutar las tareas asociadas a los eventos del sistema; generalmente las reciben y las delegan al controlador.

Problema ¿Quién debería encargarse de atender un evento del sistema?

Un **evento del sistema** es un evento de alto nivel generado por un actor externo; es un evento de entrada externa. Se asocia a **operaciones del sistema**: las que emite en respuesta a los eventos del sistema. Por ejemplo, cuando un cajero que usa un sistema de terminal en el punto de venta oprime el botón "Terminar Venta", está generando un evento sistémico que indica que "la venta ha terminado". De modo similar, cuando un escritor que usa un procesador de palabras pulsa el botón "revisar ortografía", está produciendo un evento que indica "realizar una revisión ortográfica".

Un **Controlador** es un objeto de interfaz no destinada al usuario que se encarga de manejar un evento del sistema. Define además el método de su operación.

Ejemplo En la aplicación del punto de venta se dan varias operaciones del sistema, como se advierte en la figura 18.13.

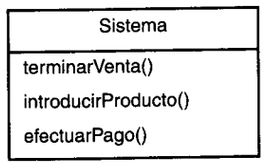


Figura 18.13 Operaciones del sistema asociadas a los eventos sistémicos.

Durante el análisis del comportamiento del sistema, sus operaciones son asignadas al tipo *Sistema*, para indicar que son operaciones del sistema. Pero ello *no* significa que una clase llamada *Sistema* las ejecute durante el diseño.

Más bien, durante el diseño, a la clase Controlador se le asigna la responsabilidad de las operaciones del sistema (figura 18.14).

¿Quién debería ser el controlador de eventos sistémicos como *introducirProducto* y *terminarVenta*?

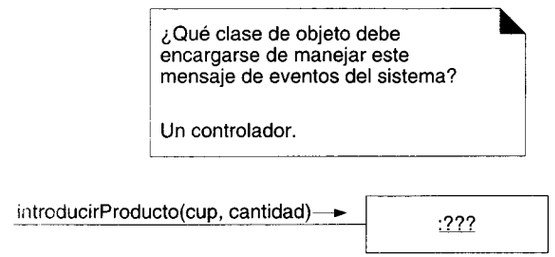


Figura 18.14 ¿Controlador de introducirProducto?

De acuerdo con el patrón Controlador, disponemos de las siguientes opciones:

representa el "sistema" global	*TPDV*
representa la empresa u organización global	*Tienda*
representa algo en el mundo real que está activo (por ejemplo, el papel de una persona) y que puede intervenir en la tarea	*Cajero*
representa un manejador artificial de todas las operaciones del sistema de un caso de uso.	*ManejadordeComprar-Productos*

Por lo que respecta a los diagramas de colaboración, lo anterior significa que se usará uno de los ejemplos de la figura 18.15.

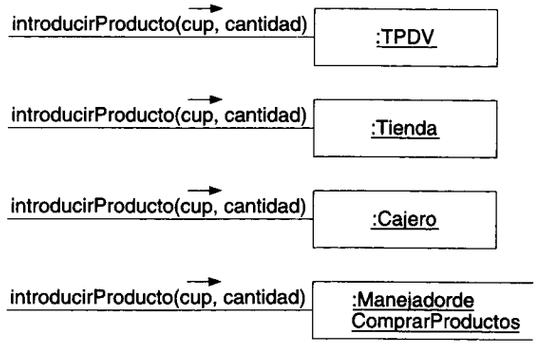


Figura 18.15 Decisiones del controlador.

En la decisión de cuál de las cuatro clases es el controlador más apropiado influyen también otros factores como la cohesión y el acoplamiento. La sección de la explicación profundiza este aspecto.

Durante el diseño, las operaciones del sistema detectadas en el análisis de su comportamiento se asignan a una o más clases de controladores como *TPDV*, según se observa en la figura 18.16.

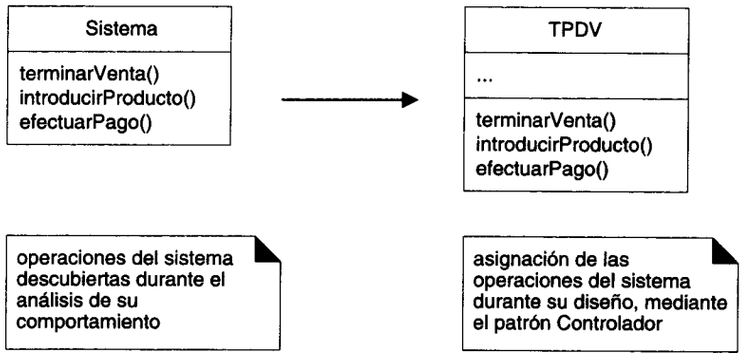


Figura 18.16 Asignación de las operaciones del sistema.

Explicación La mayor parte de los sistemas reciben eventos de entrada externa, los cuales generalmente incluyen una interfaz gráfica para el usuario (IGU) operado por una persona. Otros medios de entrada son los mensajes externos —entre ellos un conmutador de telecomunicaciones para procesar llamadas— o las señales procedentes de sensores como sucede en los sistemas de control de procesos.

En todos los casos, si se recurre a un diseño orientado a objetos, hay que elegir los controladores que manejen esos eventos de entrada. Este patrón ofrece una guía para tomar decisiones apropiadas que generalmente se aceptan.

La misma clase controlador debería utilizarse con todos los eventos sistémicos de un caso de uso, de modo que podamos conservar la información referente al estado del caso. Esta información será útil —por ejemplo— para identificar los eventos del sistema fuera de secuencia (entre ellos, una operación *efectuarPago* antes de *terminarVenta*). Puede emplearse varios controladores en los casos de uso.

Un defecto frecuente al diseñar controladores consiste en asignarles demasiada responsabilidad. Normalmente un controlador debería delegar a otros objetos el trabajo que ha de realizarse mientras coordina la actividad. Favor de consultar la sección *Problemas y soluciones* donde se dan más detalles al respecto.

La primera categoría de controlador es un controlador de fachada que representa al "sistema" global. Es una clase que, para el diseñador, representa de alguna manera al sistema entero. Podría tratarse de una unidad física, como una clase *TPDV, InterruptordeTelecomunicaciones* o *Robot*; una clase que representa todo el sistema de software, como *SistemadeInformacionAlmenudeo, SistemadeManejodeMensajes* o *ControladordeRobot*, o cualquier otro concepto que, por decisión del diseñador, denote el sistema global.

Los controladores de fachada son adecuados cuando el sistema sólo tiene unos cuantos eventos o cuando es imposible redirigir los mensajes de los eventos del sistema a otros controladores, como sucede en un sistema de procesamiento de mensajes.

Si se recurre a la cuarta categoría de controlador —un "manejador artificial de casos de uso"—, habrá entonces un controlador para cada caso. Adviértase que éste no es un objeto del dominio; es un concepto artificial cuyo fin es dar soporte al sistema (una *fabricación pura*, en términos de los patrones de GRASP). Por ejemplo, si la aplicación del punto de venta contiene casos de uso como "Comprar Productos" y "Devolver Productos", habrá una clase *ManejadordeComprarProductos* y una clase *ManejadordeDevolverProductos*.

¿Cuándo deberíamos escoger un controlador de casos de uso? Es una alternativa que debe tenerse en cuenta si el hecho de asignar las responsabilidades en cualquiera de las otras opciones de controlador genera diseños de baja cohesión o alto acoplamiento. Esto ocurre generalmente cuando un controlador empieza a "saturarse" con demasiadas responsabilidades. Un controlador de casos de uso constituye una buena alternativa cuando hay muchos eventos de sistema entre varios procesos: asigna su manejo a clases individuales controlables, además de que ofrece una base para reflexionar sobre el estado del proceso actual.

Un corolario importante del patrón Controlador es que los objetos externos de conexión (por ejemplo, los objetos ventana, los applets —o pequeñas aplicaciones—) y la capa de presentación no deberían tener la responsabilidad de llevar a cabo los eventos del sistema.

Dicho con otras palabras, las operaciones del sistema —las cuales reflejan los procesos de la empresa o el dominio— deberían manejarse en la capa de dominio de los objetos y no en las de interfaz, presentación o aplicación. Véase un ejemplo en la siguiente sección titulada *Problemas y Soluciones*.

Beneficios

- *Mayor potencial de los componentes reutilizables.* Garantiza que la empresa o los procesos de dominio sean manejados por la capa de los objetos del dominio y no por la de la interfaz. Desde el punto de vista técnico, las responsabilidades del controlador podrían cumplirse en un objeto de interfaz, pero esto supone que el código del programa y la lógica relacionada con la realización de los procesos del dominio puro quedarían incrustados en los objetos interfaz o ventana. Un diseño de interfaz-como-controlador reduce la posibilidad de reutilizar la lógica de los procesos del dominio en aplicaciones futuras, por estar ligada a una interfaz determinada (por ejemplo, un objeto similar a una ventaja) que rara vez puede utilizarse en otras aplicaciones. En cambio, el hecho de delegar a un controlador la responsabilidad de la operación de un sistema entre las clases del dominio soporta la reutilización de la lógica para manejar los procesos afines del negocio en aplicaciones futuras.
- *Reflexionar sobre el estado del caso de uso.* A veces es necesario asegurarse de que las operaciones del sistema sigan una secuencia legal o poder razonar sobre el estado actual de la actividad y las operaciones en el caso de uso subyacente. Por ejemplo, tal vez deba garantizarse que la operación *efectuarPago* no ocurra mientras no se concluya la operación *terminarVenta*. De ser así, esta información sobre el estado ha de capturarse en alguna parte; el controlador es una buena opción, sobre todo si se emplea a lo largo de todo el caso (cosa que recomendamos).

Problemas y soluciones

Controladores saturados

Una clase controlador mal diseñada presentará baja cohesión: está dispersa y tiene demasiadas áreas de responsabilidad; recibe el nombre de **controlador saturado**. Entre los signos de saturación podemos mencionar los siguientes:

- Hay una *sola* clase controlador que recibe *todos* los eventos del sistema y éstos son excesivos. Tal situación suele ocurrir si se escoge un controlador de papeles o de fachada.
- El controlador realiza él mismo muchas tareas necesarias para cumplir el evento del sistema, sin que delegue trabajo. Esto suele constituir una violación de los patrones Experto y Alta Cohesión.
- Un controlador posee muchos atributos y conserva información importante sobre el sistema o dominio —información que debería haber sido redistribuida entre otros objetos— y también duplica la información en otra parte.

Hay varias formas de resolver el problema de un controlador saturado, entre las que se encuentran las siguientes:

1. Agregar más controladores: un sistema no necesariamente debe tener uno solamente. Además de los controladores de fachada, recomendamos emplear los controladores de papeles o los de casos de uso. Considere, por ejemplo, una aplicación con

muchos eventos sistémicos como podría serlo el sistema de reservaciones de una línea aérea. Puede incluir los siguientes controladores:

Controladores de papeles	Controladores de casos de uso
AgentedeReservaciones	ManejadordeHacerReservacion
ProgramadordeVuelos	ManejadorAdministraciondePro-gramacion
AnalistadeTarifas	ManejadordeAdministraciondeTarifas

2. Diseñe el controlador de modo que delegue fundamentalmente a otros objetos el desempeño de las responsabilidades de la operación del sistema.

Advertencia: los controladores de papeles pueden conducir a la obtención de diseños deficientes

Asignar una responsabilidad a un objeto de papeles humanos en una forma que imite lo que ese papel realiza en el mundo real (por ejemplo, el objeto de software *Cajero* que maneja *efectuarPago*) es aceptable, a condición de que el diseñador conozca perfectamente los posibles riesgos y los evite. En particular, se corre el peligro de generar un controlador poco coherente de papeles que no delegue. Un buen diseño orientado a objetos "les da vida", al asignarles responsabilidades, aunque representen cosas inanimadas del mundo real. Si escoge un controlador de papeles, no caiga en la trampa de diseñar objetos de tipo humano que realicen todo el trabajo; opte más bien por delegar.

En términos generales, aconsejamos usar poco los controladores de papeles.

La capa de presentación no maneja los eventos del sistema

Un corolario importante —lo repetimos— del patrón Controlador es que los objetos de la interfaz (por ejemplo, objetos de ventanas, applets) y la capa de presentación no deberían encargarse de manejar los eventos del sistema. En otras palabras, las operaciones de éste —que reflejan los procesos de la empresa o del dominio— han de realizarse en la capa del dominio de objeto y no en la de interfaz, la de presentación ni la de aplicación.

Un ejemplo es un diseño en Java que se sirve de un applet para mostrar la información.

Suponga que la aplicación del punto de venta tiene una ventana que muestra la información de ventas y que captura las operaciones del cajero. Con un patrón Controlador, la figura 18.17 describe gráficamente una relación aceptable entre el applet, el Controlador y otros objetos de la parte del sistema correspondiente al punto de venta (con simplificaciones).

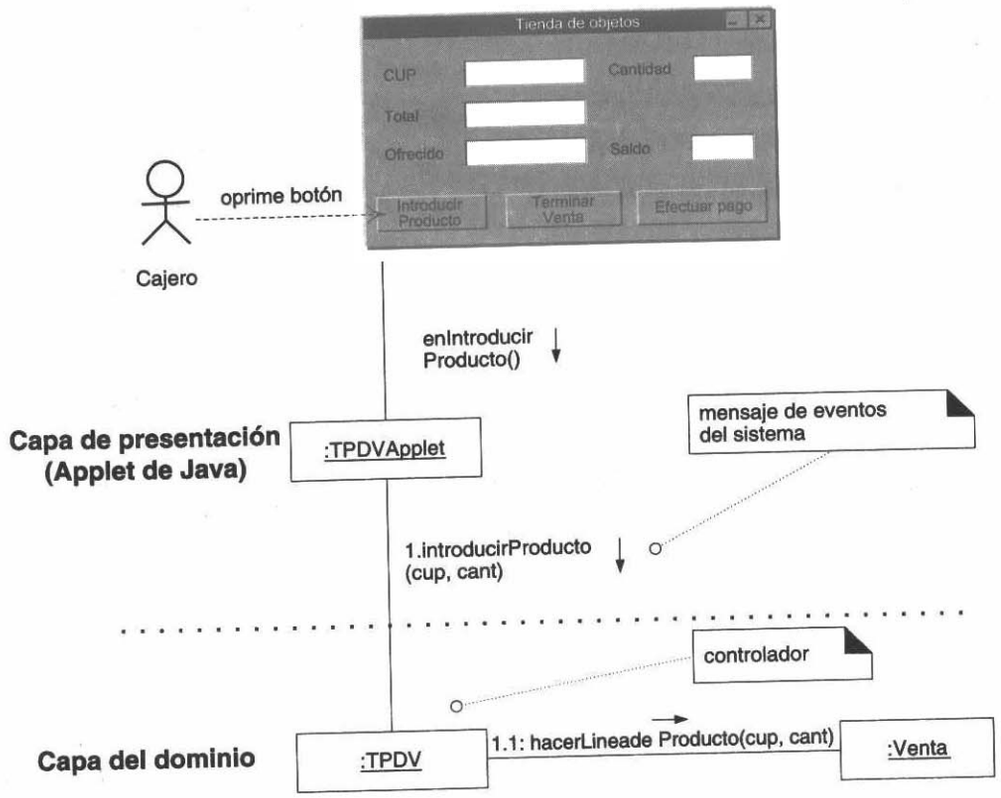


Figura 18.17 Acoplamiento adecuado de la capa de presentación a la capa del dominio.

Nótese que la clase *TPDVApplet* —parte de la capa de presentación— transmite un mensaje ***introducirProducto*** al objeto *TPDV*. No intervino en el procesamiento de la operación ni la decisión de cómo manejarla, el applet se limitó a delegarla a la capa del dominio.

El potencial de reutilización aumenta al asignar, mediante el patrón Controlador, la responsabilidad de las operaciones del sistema a los objetos situados en la capa del dominio y no en los soportes de la capa de presentación. Si un objeto capa de interfaz (como *TPDVApplet* manejó una operación del sistema —que representa parte de un proceso de negocios—, la lógica de la operación quedará contenida en una interfaz (de tipo ventana, por ejemplo), que tiene pocas posibilidades de reutilización por su acoplamiento con una interfaz y aplicaciones específicas.

En consecuencia, no recomendamos el diseño de la figura 18.18.

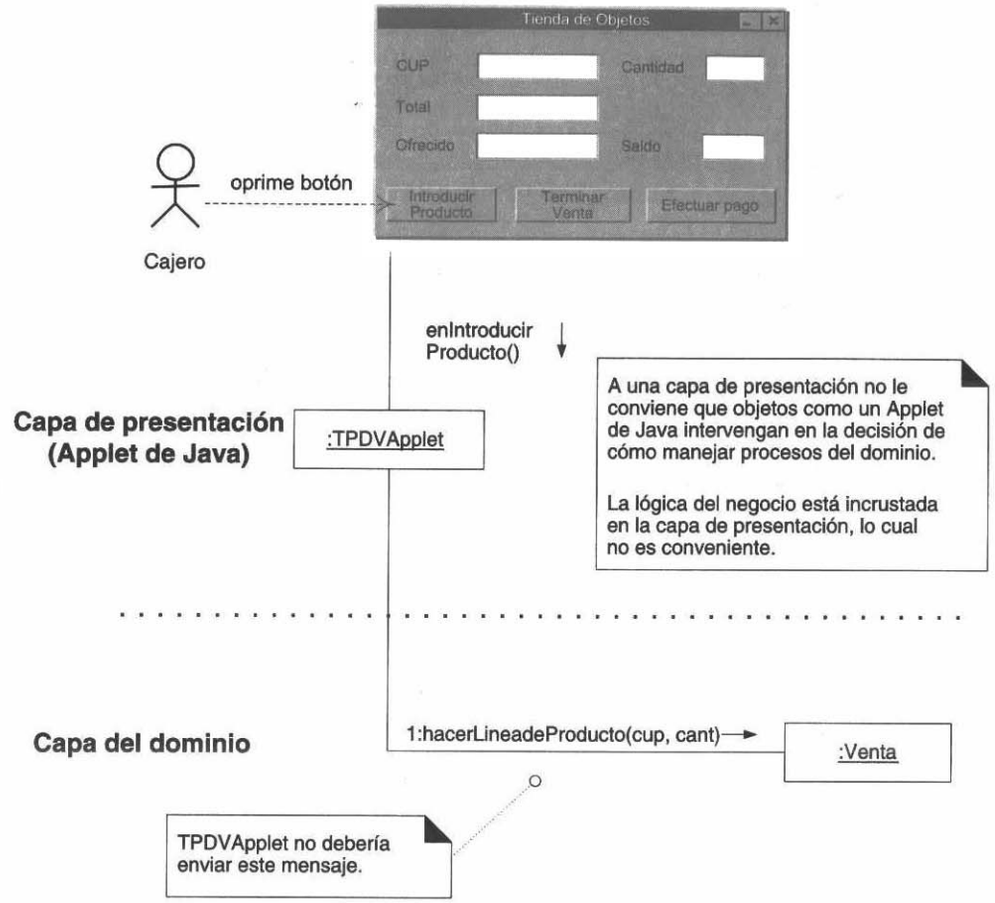


Figura 18.18 Acoplamiento inadecuado de la capa de presentación a la capa del dominio.

Asignar la responsabilidad de la operación del sistema a un controlador del objeto dominio facilita volver a utilizar la lógica del programa que soporte el proceso afín del negocio en aplicaciones futuras. Facilita asimismo desconectar la capa de interfaz y usar otro esquema u otra tecnología de interfaz o bien ejecutar el sistema en un modo "lotes" fuera de línea.

Sistemas del manejo de mensajes y el patrón Comando

Muchas aplicaciones no cuentan con una interfaz para el usuario y, en cambio, reciben mensajes provenientes de algún otro sistema externo; son sistemas de manejo de mensajes. Un conmutador de telecomunicaciones es un ejemplo muy conocido. El mensaje puede estar codificado en un registro, en una secuencia de bytes sin procesar o en un mensaje "real" dirigido a un objeto, si se usa un mecanismo de comunicación interprocesos orientado a objetos como CORBA.

En una aplicación con una interfaz para usuario (una ventana, por ejemplo), la ventana puede decidir cuál será el objeto controlador. Varias ventanas pueden colaborar con los controladores, sobre todo si se recurre a un controlador de casos de uso.

En cambio, el diseño de la interfaz y del controlador es diferente en una aplicación de manejo de mensajes. Tratándose de un caso simple, recomendamos el siguiente diseño. Si el lector desea un diseño más complejo y de muchas características, consulte el patrón Emisor-Receptor en [BMRSS96].

Para manejar los mensajes de eventos del sistema en un sistema de este tipo:

1. Defina un solo controlador para todos los mensajes de eventos; puede ser un controlador de fachada o un controlador individual de casos de uso, con un nombre como *ManejadordeMensajes*.
2. Use el patrón Comando [GHJV95] para contestar la consulta.

El patrón **Comando** especifica la definición de una clase en cada mensaje o comando, todas ellas con el método *ejecutar*. El controlador creará una instancia Comando correspondiente al mensaje del evento del sistema y le enviará un mensaje *ejecutar*. Las clases comando cuentan con un método único *ejecutar* que especifica las acciones de él, como se aprecia en la figura 18.19.[1]

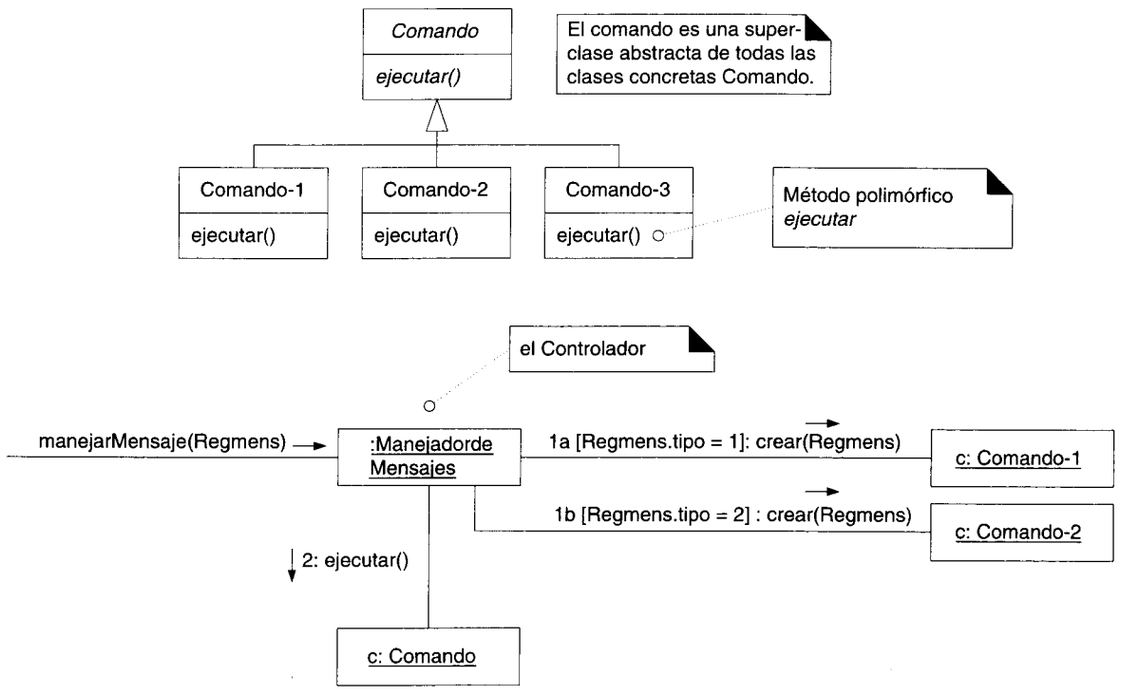


Figura 18.19 El controlador y el uso del patrón Comando.

[1] También es posible un Comando Abstracto *interfaz* en contraste con una superclase abstracta.

Patrones relacionados

- **Comando**: en un sistema de manejo de mensajes; un objeto aislado Comando puede representar y manejar los mensajes [GHJV95].
- **Fachada**: la selección de un objeto que represente un sistema u organización global para que sea controlador lo hace un tipo Fachada [GHJV95].
- **Emisor-Receptor**: éste es un patrón de Siemens [BMRSS96] útil en los sistemas de manejo de mensajes.
- **Capas**: éste es un patrón Siemens [BRMSS96]. Colocar la lógica del dominio en su capa y no en la capa de presentación es parte del patrón Capas.
- **Artefacto puro**: es otro patrón de GRASP. El artefacto puro es una clase artificial, no un concepto del dominio. Un controlador de casos de uso es un tipo de Artefacto Puro.

18.14 Responsabilidades, representación de papeles y las tarjetas CRC

Aunque no sean una parte formal del lenguaje UML, las **tarjetas CRC** [BC89] (siglas en inglés de *Class-Responsibility-Collaborator* Clase-Responsabilidad-Colaborador) son otra herramienta con la cual a veces se asignan las responsabilidades y se indica una colaboración con otros objetos. Fueron inventadas por Kent Beck y Ward Cunningham, principales promotores para que los diseñadores de objetos piensen en términos más abstractos sobre la asignación de responsabilidades y de las colaboraciones.

Las tarjetas CRC son tarjetas de índices, una por cada clase, sobre las cuales se abrevian las responsabilidades de la clase y se anota una lista de los objetos con los que colaboran para desempeñarlas. Suelen elaborarse en una sesión de grupos pequeños donde los participantes **representan papeles** de las diversas clases. Cada grupo tiene las tarjetas CRC correspondientes a las clases cuyo papel está desempeñando.

La representación de papeles hace divertido aprender a pensar en función de objetos y de responsabilidades. Pero las tarjetas basadas en texto tienen una capacidad limitada para registrar las colaboraciones en forma exhaustiva. Los diagramas de colaboración y los de clase describen mejor las conexiones entre los objetos y el contexto general.

Recomendamos por su utilidad las actividades grupales de diseño dirigidas a asignar responsabilidades y a representar papeles; los resultados pueden anotarse después en las tarjetas CRC o en diagramas. Las tarjetas son un método de registrar los resultados de la asignación de responsabilidades y de las colaboraciones. El registro es más satisfactorio si se utiliza este tipo de diagramas. El valor real no está constituido por las tarjetas, sino por el análisis de la asignación de responsabilidades y por la actividad relacionada con la representación de papeles.