Análisis y diseño orientados a objetos

ABRAHAM SÁNCHEZ LÓPEZ
GRUPO MOVIS
FCC-BUAP

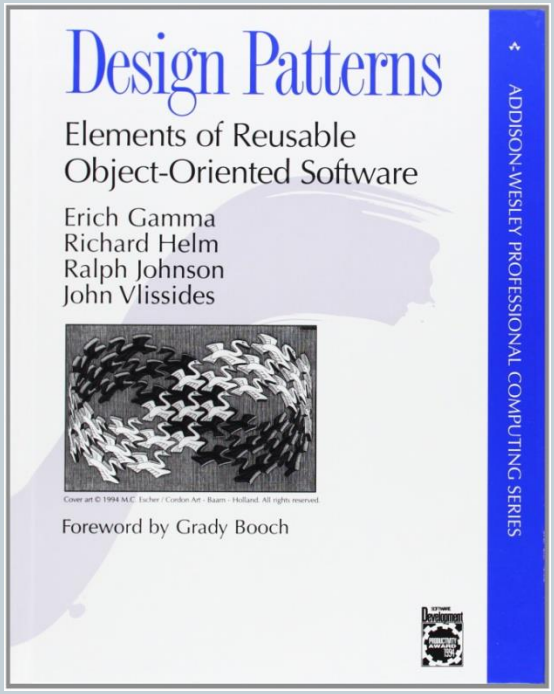
Introducción, I

- ¿Qué significa contar con un sistema orientado a objetos que esté bien diseñado?
- A lo largo del curso se propondrán habilidades prácticas en el análisis y diseño orientado a objetos.
- Estas destrezas son indispensables para crear sistemas de software bien diseñados, robustos y de fácil mantenimiento, utilizando la tecnología de objetos y los lenguajes de programación orientados a objetos (C++, Java, C#, PHP,…).
- El proverbio “el hábito no hace al monje” se aplica a la perfección en la tecnología de objetos.
- El hecho de conocer un lenguaje orientado a objetos (por ejemplo Java), es un primer paso necesario pero insuficiente para crear sistemas de objetos.
- Se requiere además analizar y diseñar un sistema desde la perspectiva de los objetos.
- A menudo nos planteamos la siguientes preguntas:
  - ¿Cómo deberían asignarse las responsabilidades a las clases de objetos?
  - ¿Cómo deberían interactuar estos?

Introducción, II

- ¿Qué papel debe destinársele a cada clase?

- Algunas soluciones ya probadas y eficaces de los problemas de diseño pueden expresarse (y se han plasmado) como un conjunto de principios, con heurística o patrones (fórmulas de solución de problemas que codifican los principios aceptados del diseño).



Asignación de responsabilidades

- Hay muchas posibles actividades y elementos en el análisis y en el diseño, así como gran cantidad de principios y directrices.
- Supongamos que tenemos que elegir una sola habilidad práctica entre todos los temas existentes en el análisis y diseño orientados a objetos.

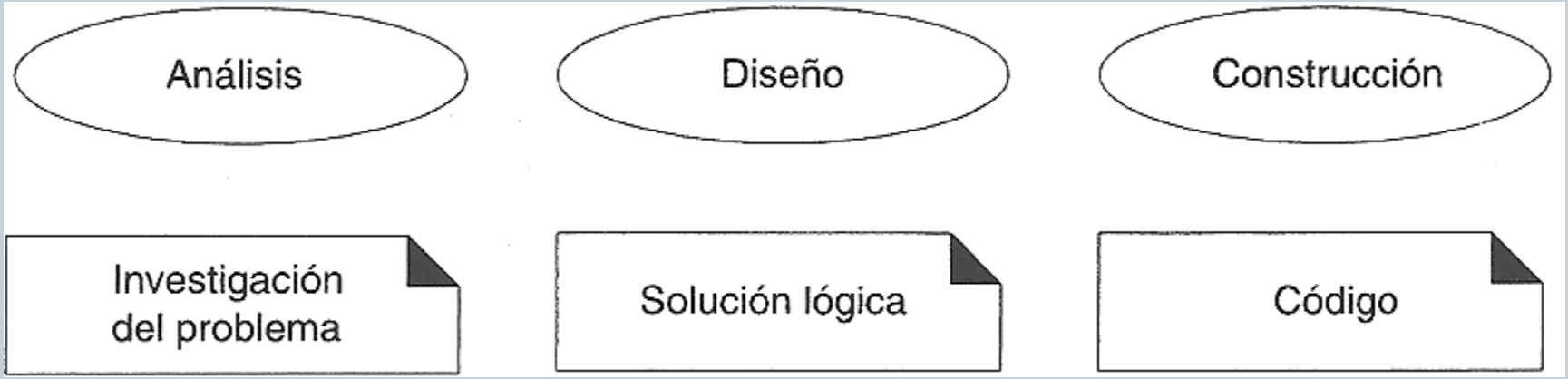
La habilidad más importante en el análisis y diseño orientados a objetos es asignar eficientemente las responsabilidades a los componentes de software.

- Es de hecho la actividad que debe efectuarse (es ineludible) y que influye más profundamente en la solidez, en la capacidad de mantenimiento y en la reusabilidad de los componentes de software.
- Igualmente como segunda actividad importante, aparece la obtención de los objetos o las abstracciones adecuadas.
- El proceso de desarrollo "con prisa por codificar" ➔ asignación de responsabilidades (inevitable!!).

Análisis y diseño orientados a objetos, I

- Para crear una aplicación de software hay que describir el problema y las necesidades o requerimientos.
- El **análisis** se centra en una *investigación* del problema, no en la manera de definir una solución.
- Para desarrollar una aplicación, también es necesario contar con descripciones detalladas y de alto nivel de la solución lógica y saber cómo satisface los requerimientos y las restricciones.
- El **diseño** enfatiza una *solución lógica*: cómo el sistema cumple con los requerimientos.
- Por lo tanto, la esencia del análisis y diseño orientados a objetos consiste en situar el dominio de un problema y su solución lógica dentro de la perspectiva de los objetos (cosas, conceptos o entidades), como se muestra en la siguiente figura.
- Durante el análisis orientado a objetos se procura ante todo identificar y describir los objetos – o conceptos – dentro del dominio del problema.

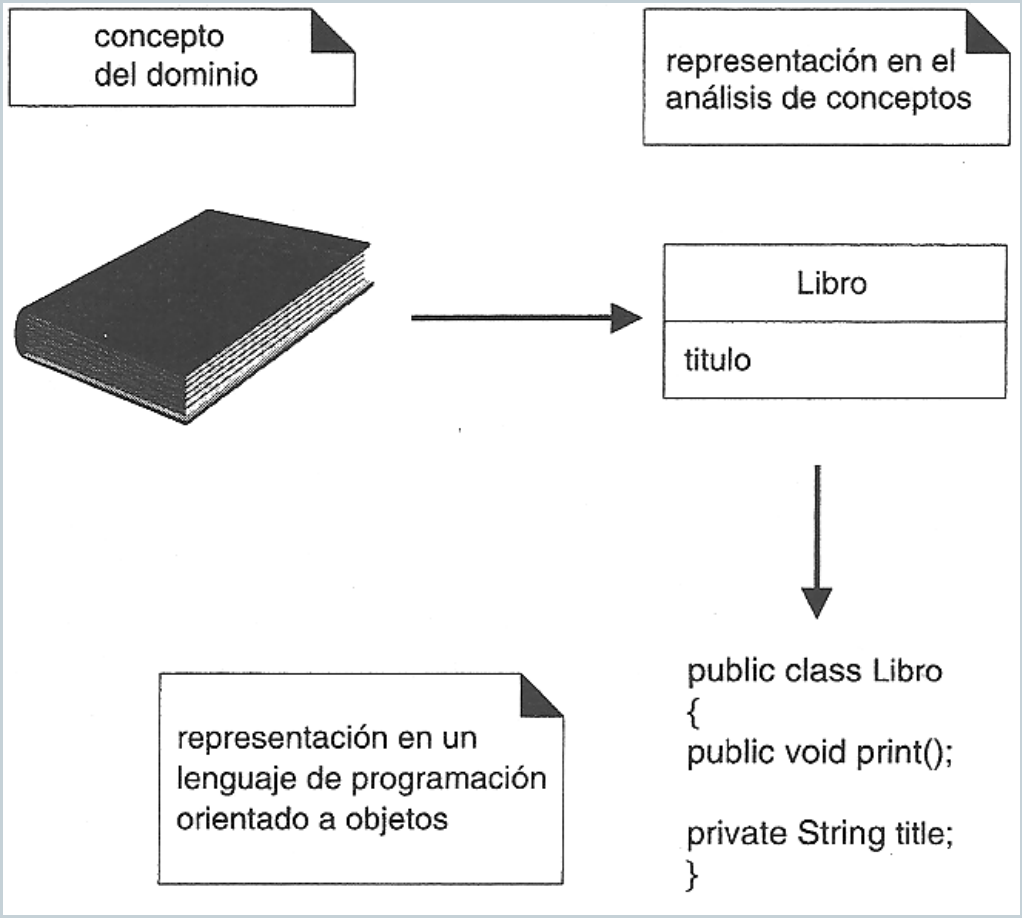
Análisis y diseño orientados a objetos, II



- Durante el diseño orientado a objetos, se procura definir los objetos lógicos del software que finalmente serán implementados en un lenguaje de programación orientado a objetos.
- Los métodos tienen atributos y métodos. Ver la siguiente figura para ilustrar con un ejemplo estos conceptos.
- Finalmente, durante la construcción o programación orientada a objetos, se implementan los componentes del diseño, como una clase (PHP, C++, Java, …).

Análisis y diseño orientados a objetos, III

- La orientación a objetos se centra en la representación de objetos.



Responsabilidades y métodos, I

- Booch y Rumbaugh definen la **responsabilidad** como un “contrato u obligación de un tipo o clase”.
- Las responsabilidades se relacionan con las obligaciones de un objeto respecto a su comportamiento.
- Esas responsabilidades pertenecen, esencialmente a las dos características siguientes: 1) conocer y 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 pueden derivar o calcular.

Responsabilidades y métodos, II

- Las responsabilidades se asignan a los objetos durante el diseño orientado a objetos.
- Ejemplo:

  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 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. Estas se implementan usando métodos que operen solos o en colaboración con otros métodos y objetos.

Patrones GRASP, I

- GRASP (General Responsability Asignment Software Patterns), Patrones Generales de Software Para Asignar Responsabilidades.
- Como sabemos, asignar 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 o de clases.
- Los patrones son parejas de problema/solución con un nombre, que codifican buenos principios y sugerencias relacionados frecuentemente con la asignación de responsabilidades.
- Es importante entender y poder aplicar estos principios durante la preparación de un diagrama, 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.
- A continuación, se explicaran los patrones GRASP.

Patrones GRASP, II

- Revisemos los primeros cinco patrones:
  1. Experto
  2. Creador
  3. Alta cohesión
  4. Bajo acoplamiento
  5. Controlador

Patrón Experto, I

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 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.

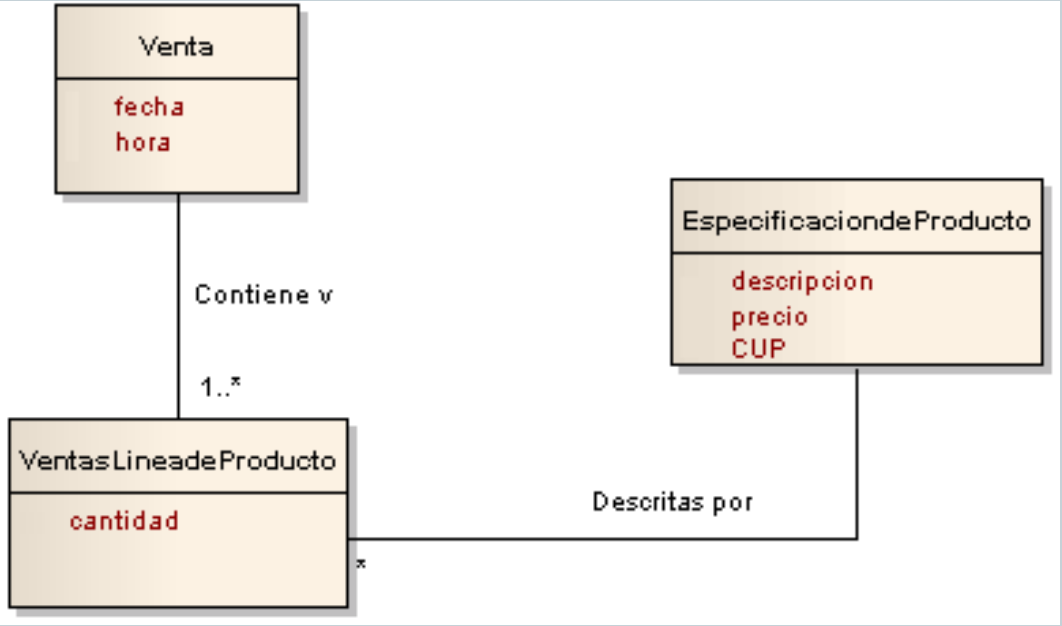
Si se hacen de forma adecuada los sistemas tienden a ser mas fáciles de entender, mantener y ampliar, y se nos presenta la oportunidad de reutilizar los componentes en futuras aplicaciones.

Ejemplo:

En la aplicación de un punto de venta, alguna clase necesita conocer el gran total de la venta. Nos planteamos la siguiente pregunta: ¿Quién es el responsable de conocer el gran total de la venta ?

Patrón Experto, II

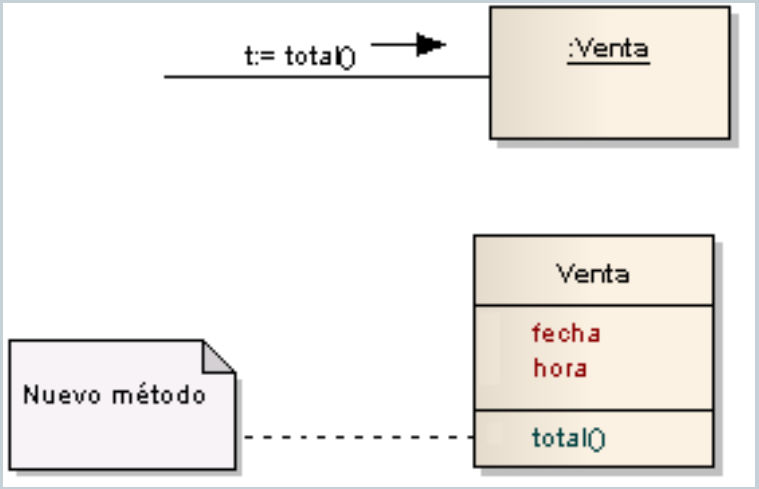
- Desde el punto de vista del patrón Experto, deberíamos buscar la clase de objetos que posee la información necesaria para calcular el total. Examinaremos con detalles, el modelo conceptual parcial de la siguiente figura.



- ¿Qué información falta para calcular el gran total? Hay de hecho, que conocer todas las instancias VentasLineadeProducto de una venta y la suma de los subtotales.

Patrón Experto, III

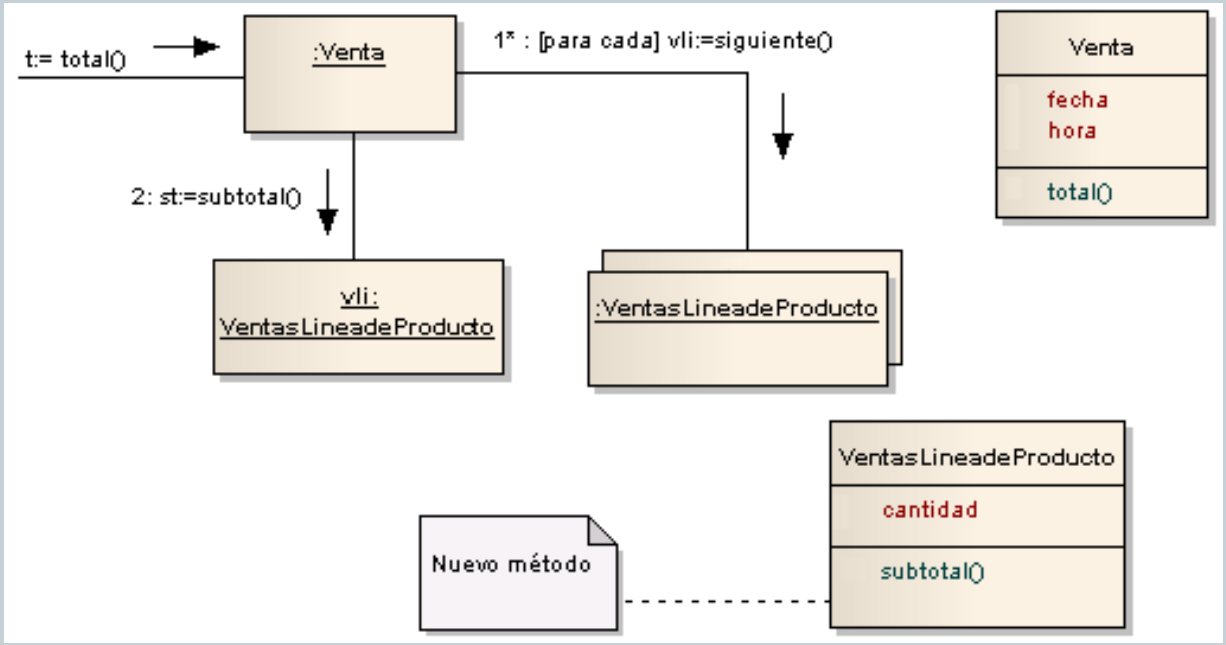
- Esto lo conoce únicamente la instancia Venta; por lo tanto, desde el punto de vista del Experto, Venta es la clase correcta de objeto para asumir esta responsabilidad; es el experto en información.
- El siguiente diagrama parcial de colaboración y de clases de la siguiente figura describen las decisiones que iremos adoptando.



- ¿Qué información hace falta para determinar el subtotal de la línea de productos?
- Veamos la siguiente figura para mostrar el cálculo del total de la venta.

Patrón Experto, IV

- Se necesitan VentasLineadeProducto.cantidad y EspecificaciondeProducto.precio. VentasLineadeProducto conoce su cantidad y su correspondiente EspecificaciondeProducto; por lo tanto, desde la perspectiva del patrón Experto.
- VentasLineadeProducto debería calcular el subtotal, es el experto en información.

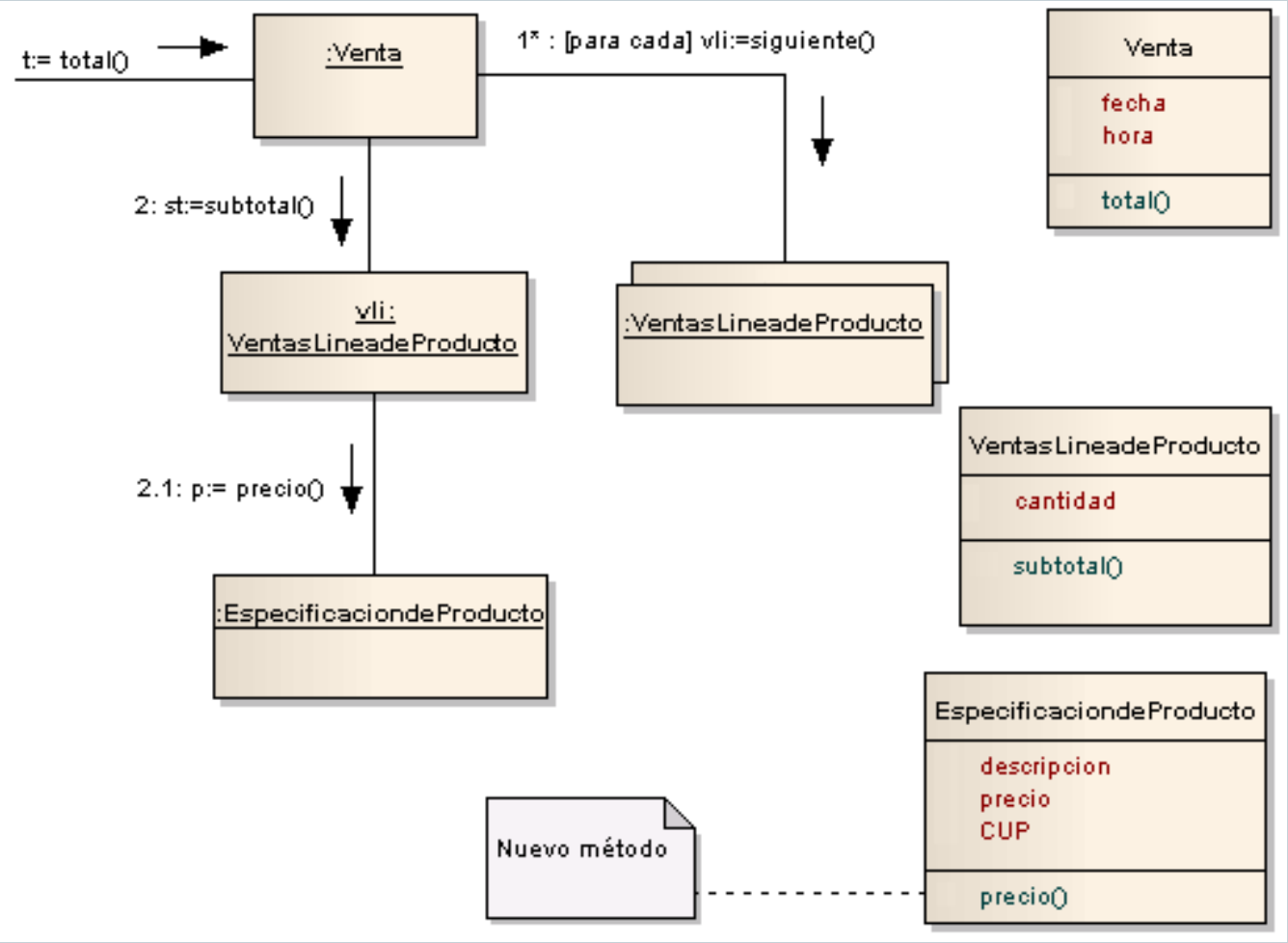


Patrón Experto, V

- De acuerdo a la figura, Venta necesita enviar mensajes de subtotal a cada VentasLineadeProducto y sumar los resultados.
- 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 lo tanto, habrá que enviarle un mensaje preguntándole el precio.
- El diseño correspondiente se muestra en la figura del acetato 17.
- 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.

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

Patrón Experto, VI



Explicación del patrón Experto

- 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, el cual expresa simplemente la intuición de que los objetos hacen cosas relacionadas con la información que poseen.
- Podemos notar que el cumplimiento de una responsabilidad requiere a menudo información distribuida en varias clases de objetos.
- Ello significa que hay varios expertos parciales que colaboran en la tarea, cuando la información se encuentra esparcida en varios objetos, estos habrán de interactuar a través de mensajes para compartir el trabajo.
- El patrón experto ofrece una analogía con el mundo real. Acostumbramos a asignar responsabilidad a individuos que disponen de la información necesaria para llevar a cabo una tarea.

Beneficios del patrón Experto

- 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.
- 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 mas fáciles de comprender y mantener.
- Así se brinda el soporte a una **alta cohesión**.

Patrón Creador, I

Solución: Asignar 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 a la clase B que contenga la clase A.

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

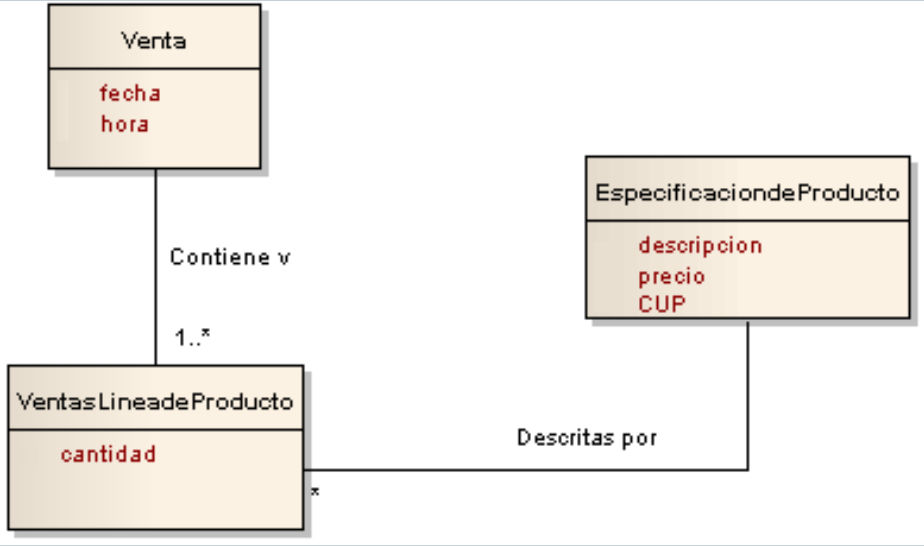
La creación de objetos es una de las actividades mas frecuentes en un sistema orientado a objetos.

En consecuencia, es conveniente 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 reusabilidad.

Patrón Creador, II

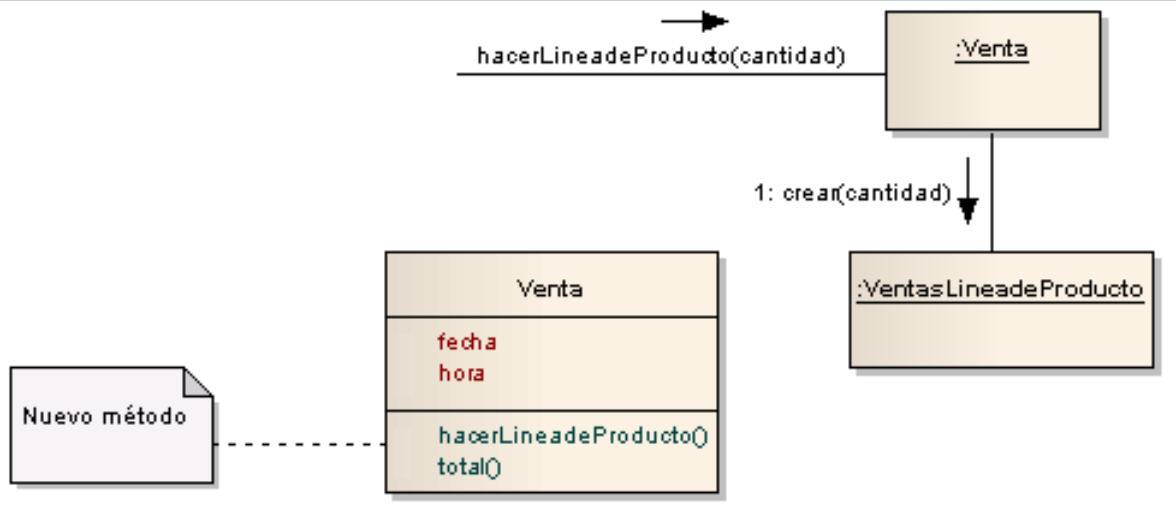
Ejemplo:

- 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.
- Examinemos a continuación el modelo conceptual parcial de la siguiente figura.



Patrón Creador, III

- Una venta contiene (en realidad, agrega) muchos objetos VentasLineadePoducto; 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, como el mostrado a continuación.



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

Explicación del patrón Creador, I

- 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 debe dar 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 solo de una directriz.
- El concepto de agregación se utiliza al examinar el patrón Creador. La agregación no es un tema que se trata en estas notas, pero podríamos proporcionar una definición adecuada.

Explicación del patrón Creador, II

- La agregación incluye cosas que están en una sólida relación de parte-todo o de parte-estructura; por ejemplo, Cuerpo agrega pierna, o Párrafo agrega oración.
- En ocasiones se puede encontrar un patrón creador buscando la clase con los datos de inicialización que serán transferidos durante la creación. Este 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.

Beneficios del patrón Creador

- Se brinda soporte a un **bajo acoplamiento**, 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.
- Entre los patrones conexos, tenemos a los siguientes:

  Bajo acoplamiento.

  Parte-todo describe un patrón para definir los objetos que soportan el encapsulamiento de sus componentes.

Patrón Bajo Acoplamiento, I

Solución: Asignar una responsabilidad para mantener bajo acoplamiento.

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

El acoplamiento es una medida de la fuerza con que la clase esta 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.

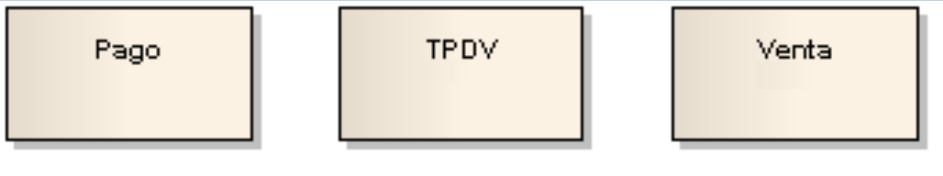
Una clase con alto (o fuerte) acoplamiento recurre a muchas otras. Este tipo de clases no es conveniente, pues presentan los siguientes problemas:

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

Ejemplo:

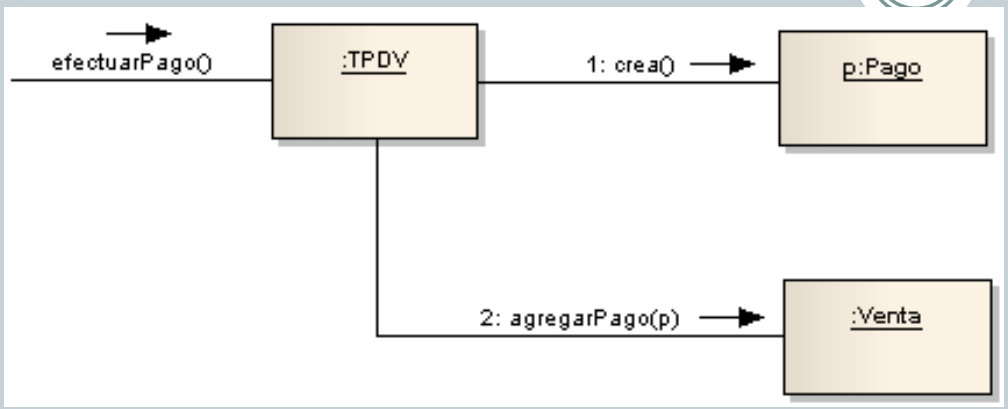
- Revisemos atentamente el siguiente diagrama parcial de clases, desde la perspectiva de la aplicación de la terminal de punto de venta (TPDV).

Patrón Bajo Acoplamiento, II



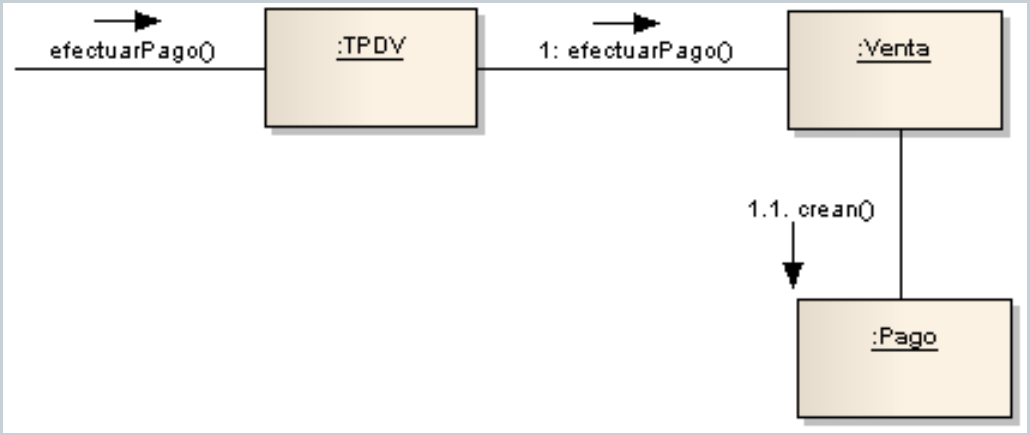
- Podemos suponer que se necesita crear una instancia Pago y asociarla a Venta. ¿Qué clase se encargará de esto?
- Dado que una instancia de TPDV "registra" un Pago en el dominio del mundo real, el patrón creador indica que TPDV es un buen candidato para producir el Pago.
- La instancia TPDV podría entonces enviarle a Venta el mensaje agregarPago, transmitiendo al mismo tiempo el nuevo Pago como parámetro.
- En la siguiente figura, se incluye un posible diagrama parcial de colaboración que muestra gráficamente esto.
- Esta asignación de responsabilidades acopla la clase TPDV al conocimiento de la clase Pago. En la figura del acetato 28 (parte de abajo) se muestra la solución alterna para crear el Pago y asociarlo a la Venta.

Patrón Bajo Acoplamiento, III



TPDV crea Pago

Venta crea Pago



- ¿Qué diseño, basado en la asignación de responsabilidades, brinda soporta al patrón bajo acoplamiento?

Patrón Bajo Acoplamiento, IV

- En ambos casos, suponemos que Venta terminará acoplándose al conocimiento de un Pago.
- En el primer diseño, donde la instancia TPDV crea Pago, incorpora el acoplamiento de TPDV a Pago.
- En cambio, en el segundo diseño, donde la Venta realiza la creación de un Pago, no incrementa el acoplamiento.
- Tomando como única perspectiva el acoplamiento, el segundo diseño es preferible porque se conserva un menor acoplamiento global.
- En la práctica, el grado de acoplamiento no puede considerarse aisladamente de otros principios como Experto y Alta cohesión.

Explicación del patrón Bajo Acoplamiento

- El bajo acoplamiento es un principio que se debe recordar durante las decisiones de diseño: es la meta principal que es preciso tener presente siempre.
- Es un patrón evaluativo que un diseñador debe aplicar al juzgar sus decisiones en el diseño.
- 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 mas 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 reusabilidad.

Formas comunes de acoplamiento

- 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.

- Entre los beneficios del patrón de bajo acoplamiento, tenemos a los siguientes:
  - No se afectan por cambios de otros componentes.
  - Fáciles de entender por separado.
  - Fáciles de reutilizar.

Patrón Alta Cohesión, I

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 es una medida de cuan 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.

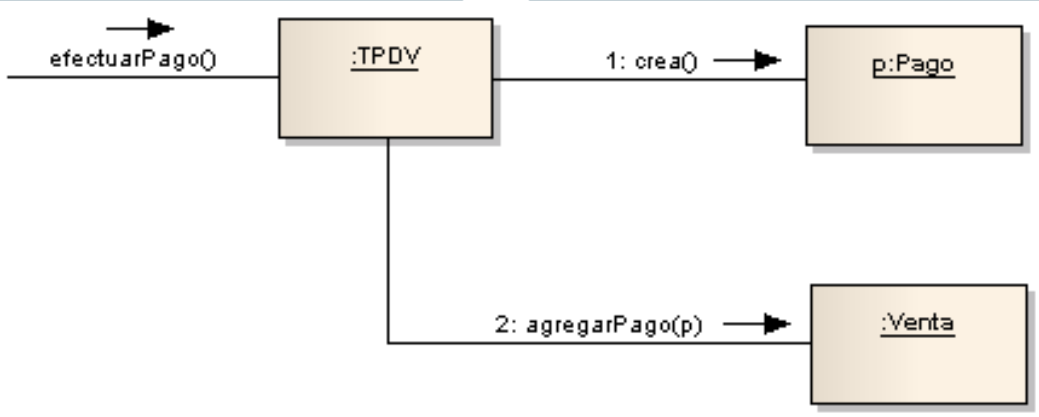
Las clases con baja cohesión a menudo representan un alto grado de abstracción o han asumido responsabilidades que deberían haber delegado a otros objetos.

Patrón Alta Cohesión, II

Ejemplo:

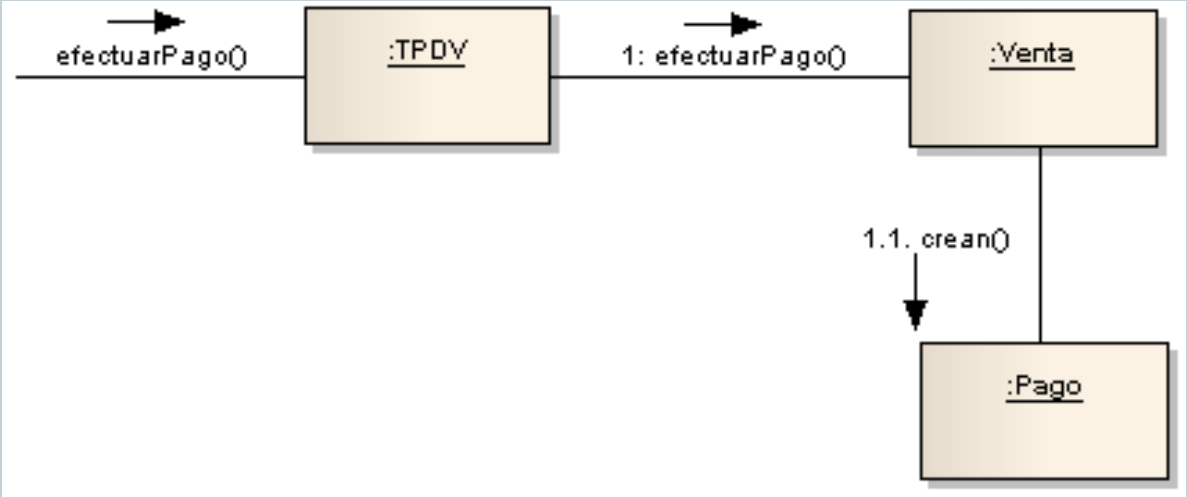
- Podemos retomar el ejemplo que se ha utilizado como ejemplo del patrón bajo acoplamiento.
- Supongamos que necesitamos crear una instancia de Pago (en efectivo) y asociarla a la Venta. ¿Qué clase será la responsable de hacerlo?
- Como TPDV registra un Pago en el dominio del mundo real, el patrón Creador sugiere que TPDV es un buen candidato para que genera Pago.
- Entonces la instancia de TPDV podría enviar a Venta un mensaje agregarPago, transmitiendo al mismo tiempo el nuevo Pago como parámetro, según se puede observar en la siguiente figura (acetato 34).
- 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.

Patrón Alta Cohesión, III



- 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.
- En el segundo diseño, se delega a la Venta la responsabilidad de crear el pago, que soporta una mayo cohesión de TPDV. Este segundo diseño brinda soporte a una alta cohesión y a un bajo acoplamiento, de ahí que sea conveniente.
- 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.

Patrón Alta Cohesión, IV



- Como el patrón Bajo Acoplamiento, también Alta Cohesión es un principio que se debería tener presente en todas las decisiones de diseño.
- Es la meta principal que ha de buscarse en todo momento. Es un patrón evaluativo que un 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” .

Explicación del patrón Alta Cohesión, I

- Enseguida 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.
2. *Baja cohesión*: Una clase tiene la responsabilidad exclusiva de una tarea compleja dentro de un área funcional.
3. *Alta cohesión*: Una clase tiene responsabilidades moderadas en un área funcional y colabora con las otras para llevar a cabo las tareas.
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.

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

Explicación del patrón Alta Cohesión, II

- 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, 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 del patrón de Alta Cohesión

- Entre los beneficios, tenemos a los siguientes:
  - 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, por que una clase muy cohesiva puede destinarse a un propósito muy especifico.

Patrón Controlador, I

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<nombre caso de uso>” (controlador de casos de uso).

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

En la lista no figuran las clases ventana, aplicación, vista ni documento, estas clases no deben 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?

Patrón Controlador, II

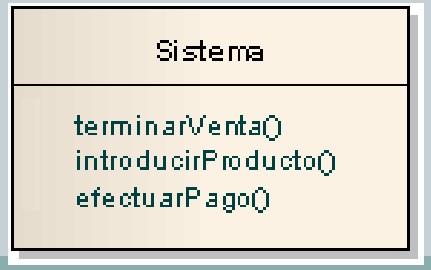
Un **evento del sistema** es un evento de alto nivel generado por un actor externo; es un evento de entrada externa. Se asocia a **operaciones del sistema**: las que emite en respuesta a los eventos del sistema.

Por ejemplo cuando un cajero que usa un sistema de Terminal en el punto de venta oprime el botón "Terminar Venta", esta 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", esta 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.

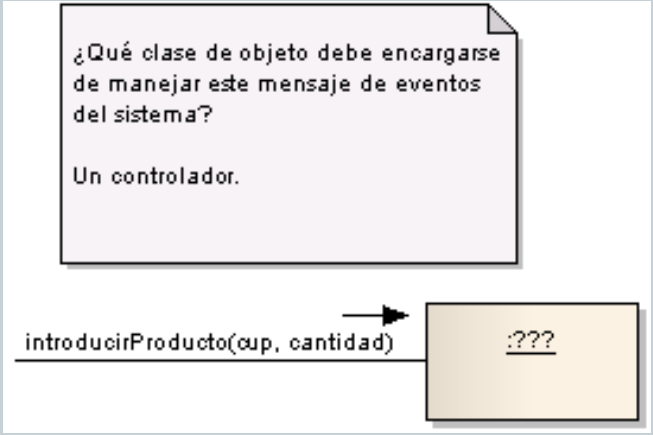
Ejemplo:

- En la aplicación del punto de venta, se tienen varias operaciones del sistema, como se muestra a continuación.



Patrón Controlador, III

- 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.
- Mas bien durante el diseño, a la clase Controlador se la asigna la responsabilidad de las operaciones del sistema, ver la siguiente figura.



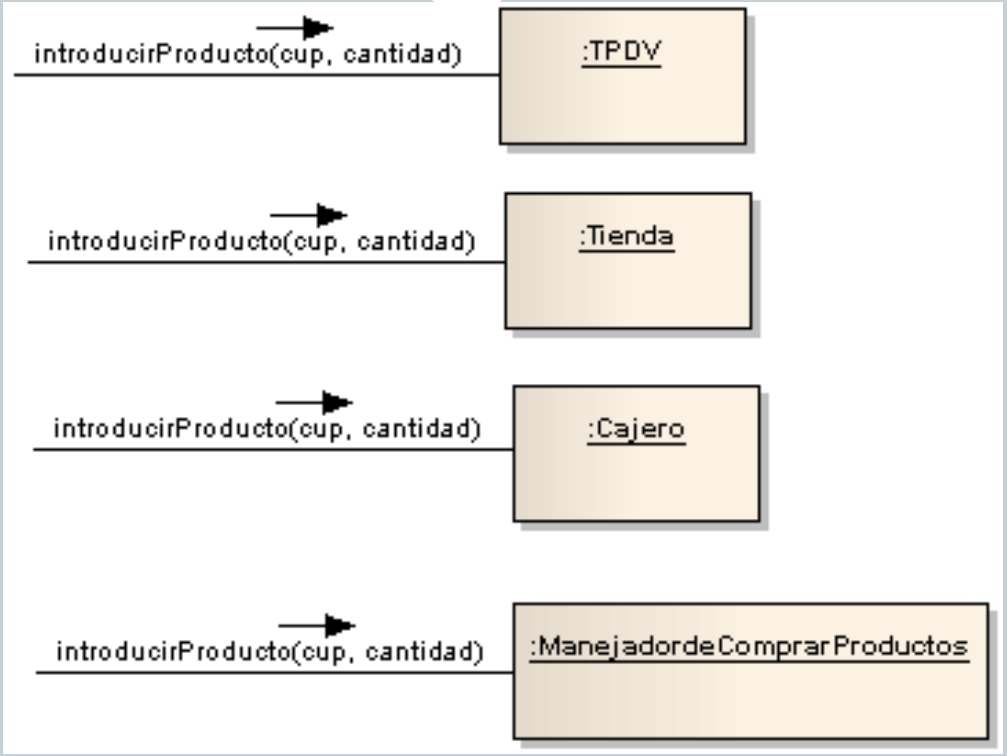
- ¿Quién debería ser el controlador de eventos sistémicos como **introducirProducto** y **terminarVenta**?
- De acuerdo con el patrón Controlador, disponemos de la siguientes opciones:

Patrón Controlador, IV

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	ManejadordeComprarProductos

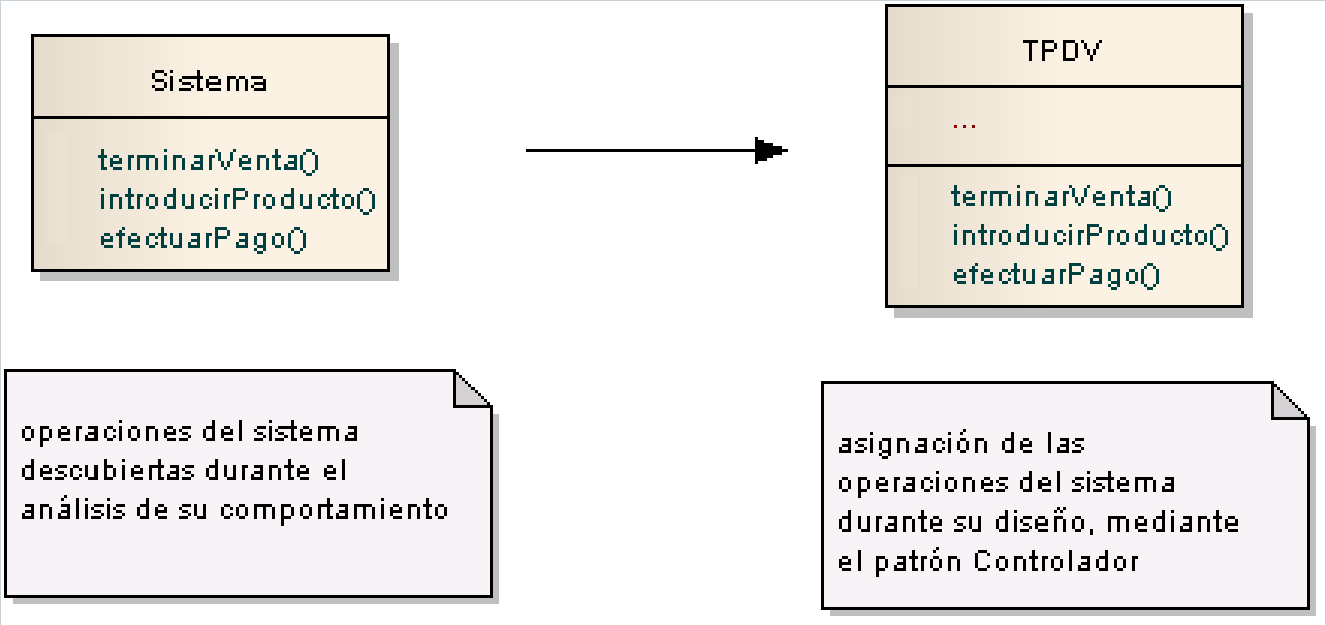
- Por lo que respecta a los diagramas de colaboración, lo anterior significa que se usará uno de los ejemplos de la siguiente figura (acetato 43).
- En la decisión de cuál de las cuatro clases es el controlador más apropiado influyen también otros factores como la cohesión y el acoplamiento.
- Durante 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 del acetato 44.

Patrón Controlador, V



- La mayor parte de los sistemas reciben eventos de entrada externa, los cuales generalmente incluyen una interfaz grafica para el usuario (GUI) operado por una persona. Otros medios de entrada son los mensajes externos o las señales procedentes de sensores, como sucede en los sistemas de control de procesos..

Explicación del patrón Controlador, I



- En todos los casos, si se recurre a un diseño orientado a objetos, hay que elegir los controladores que manejan esos eventos de entrada. Este patrón ofrece una guía para tomar decisiones apropiadas que generalmente se aceptan.
- La misma clase controlador debería usarse con todos los eventos sistemáticos de un caso de uso, de modo que podamos conservar la información referente al estado del caso.

Explicación del patrón Controlador, II

- Esta información será útil para identificar los eventos del sistema fuera de presencia (entre ellas por ejemplo una operación efectuarPago, antes de terminarVenta).
- 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.
- 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 Caja, InterruptordeComunicaciones o Robot; una clase que representa todo el sistema de software, como SistemaInformacionalMenudeo, 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 del sistema a otros controladores, como sucede en un sistema de procesamiento de mensajes.

Explicación del patrón Controlador, III

- Si se recurre a la cuarta categoría de controlador (manejador artificial de casos de uso), habrá entonces un controlador para cada caso. Se advierte que este 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).
- ¿Cuándo deberíamos escoger un controlador de casos de uso?
- Es una alternativa que debe de tenerse en cuenta si el hecho de asignar las responsabilices en cualquiera de las otras opciones de controlador genera diseños de baja cohesión o alto acoplamiento.
- Esto ocurre generalmente cuando un controlador comienza 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 un manejo a clases individuales controlables, además de que ofrece una base para reflexionar sobre el estado del proceso actual.

Beneficios del patrón Controlador ,I

- **Mayor potencial de los componentes reutilizables**.

  Garantiza que la empresa o los procesos del 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 ventana) 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.

Beneficios del patrón Controlador ,II

- **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; en el controlador es una buena opción, sobre todo si se emplea a lo largo de todo el caso (cosa que se recomienda).

Controladores saturados, I

- Una clase controlador mal diseñada presentara baja cohesión: esta dispersa y tiene demasiadas áreas de responsabilidad; recibe el nombre de **controlador saturado**.
- Entre los signos de saturación se pueden mencionar los siguientes:
  - Hay una sola clase de controlador que recibe todos los eventos del sistema y estos son excesivos. Tal situación suele ocurrir si se escoge un controlador de papeles o de fachada.
  - El controlador realiza el 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.

Controladores saturados, II

Además, de los controladores de fachada, se recomienda emplear los controladores de papeles o los casos de uso.

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