PATRONES GRASP

GRASP es el acrónimo de *General Responsability Assignment Software Patterns*, y es una de las familias de patrones que propuso Craig Larman, el cual se centró principalmente en proponer los patrones como una codificación de principios básicos ampliamente utilizados para los expertos en objetos, para que parezcan elementales y familiares. Por ese motivo los patrones GRASP tienen nombres concisos como *Experto en Información*, *Creador*, *Variaciones protegidas*... para facilitar la comunicación.

Los patrones GRASP, también conocidos como Patrones de Principios Generales para Asignar Responsabilidades, se basan principalmente en que la asignación de responsabilidades es extremadamente importante en el diseño orientado a objetos, ya que es en esta fase de diseño donde nos encontramos con la tarea de crear las clases y las relaciones entre ellas. La decisión a cerca de la asignación de responsabilidades tiene lugar, casi siempre, durante la creación de los diagramas de interacción y durante la programación. Estos patrones GRASP codifican buenos consejos y principios relacionados con frecuencia con la asignación de responsabilidades.

Resumiendo, podemos decir que los patrones GRASP describen principios fundamentales del diseño de objetos y la asignación de responsabilidades, expresados como patrones.

Los patrones GRASP, que comentaremos más detalladamente en el apartado siguiente son:

- ***Bajo Acoplamiento***: debe haber pocas dependencias entre las clases.
- ***Alta Cohesión***: cada elemento de nuestro diseño debe realizar una labor única dentro del sistema.
- ***Experto o Experto en información***: la responsabilidad de realizar una labor es de la clase que tiene o puede tener los datos involucrados (atributos).
- ***Creador***: se asigna la responsabilidad de que una clase B cree un objeto de la clase A solamente cuando:
    1. B contiene a A
    2. B es una agregación (o composición) de A
    3. B almacena A
    4. B tiene los datos de inicialización de A (datos que requiere su constructor)
    5. B usa a A

- ***Controlador***: asignar la responsabilidad de controlar el flujo de eventos del sistema, a clases específicas. Esto facilita la centralización de actividades (validaciones, seguridad, etc.). El controlador no realiza estas actividades, las delega en otras clases con las que mantiene un modelo de alta cohesión.
- ***Polimorfismo***: cuando identificamos variaciones en el comportamiento, asignar la clase (interfaz) al comportamiento y utilizar el polimorfismo para implementar los comportamientos alternativos.
- ***Fabricación Pura***: cuando los problemas se complican, construir las clases que se encarguen de construir los objetos adecuados en cada momento (factorías).
- ***Indirección***: crear clases intermedias para desacoplar clientes de servicio y servicios.
- ***Variaciones protegidas***: identifica los puntos de variaciones previstas o de inestabilidad, y asigna las responsabilidades para crear una interface estable alrededor de ellos.

1. Patrón Bajo Acoplamiento.

Este patrón propone que el nivel de acoplamiento entre los objetos sea bajo, entendiendo por acoplamiento el número de elementos (clases, subclases, sistemas, etcétera) a los que un objeto está conectado a, tiene conocimiento de, confía en otros elementos.

Si todas las clases dependen de todas las clases, ¿cuánto software podemos extraer y reutilizarlo en otro proyecto? Además el que las clases tengan un fuerte acoplamiento nos conlleva los siguientes problemas:

- cambios en las clases relacionadas fuerzan cambios locales.
- son difíciles de entender de manera aislada.
- son difíciles de reutilizar ya que en necesaria la presencia adicional de las clases de las que depende.

Como vemos el nivel de acoplamiento es un aspecto importante a tener en cuenta si lo que queremos es diseñar sistemas robustos y reutilizables. En lenguajes Orientados a Objetos (C++, Java...), algunas de las formas más comunes de acoplamiento entre el Objeto-A y el Objeto-B son:

- El Objeto-A tiene un atributo o un método que hace referencia a una instancia del Objeto-B o al propio Objeto-B. Esto comprende un parámetro o variable global del Objeto-B, o que el objeto de retorno de un mensaje sea una instancia del Objeto-B
- Un Objeto-A invoca los servicios de un Objeto-B.
- El Objeto-A es una subclase, directa o indirecta, del Objeto-B.
- El Objeto- es una interfaz y el Objeto-A implementa esa interfaz.

El patrón Bajo Acoplamiento soporta el diseño de clases que son más independientes, lo que reduce el impacto del cambio y facilita la reutilización en otros sistemas. El Bajo Acoplamiento no puede considerarse como un principio aislado del diseño de aplicaciones, ya que junto con otros patrones como el Experto o el de Alta Cohesión, influyen en la elección al asignar responsabilidades.

No podemos decir exactamente cuando el acoplamiento entre dos elementos de nuestro diseño es demasiado alto, no existe una medida absoluta. Lo que el desarrollador o el diseñador tienen que tener en cuenta es que si aumenta el grado de acoplamiento actual de su sistema le causará problemas. En general, las clases que son por naturaleza muy genéricas y que se pueden reutilizar fácilmente, deben tener un grado de acoplamiento bajo.

No es bueno que exista un acoplamiento alto entre los elementos de nuestro sistema, pero tampoco es recomendable que no exista ningún acoplamiento en nuestro sistema, ya que la tecnología Orientada a Objetos es un sistema de objetos que se comunican entre sí mediante el intercambio de mensajes. Si llevamos el bajo acoplamiento al extremo lo que vamos a conseguir es un sistema compuesto de algunos objetos individuales y pasivos, que actuarían como simples repositorios de datos. Por tanto es necesario un grado de acoplamiento moderado para que en nuestro sistema orientado a objetos las tareas se realicen mediante la colaboración de los objetos conectados entre sí.

También hay que decir que no suele ser un problema un acoplamiento alto entre objetos estables y elementos generalizados, en una aplicación Java JEE, por ejemplo, puede acoplarse con seguridad las librerías de Java (*java.util*, etc.), porque son estables y extendidas.

Para disminuir el acoplamiento los diseñadores tiene que centrarse en los puntos en los que sea realista pensar en una inestabilidad o evolución alta, pero malgastarán sus esfuerzos si se preocupan en “futuras necesidades” o en disminuir el acoplamiento en algunos puntos donde de hecho no hay motivos realistas.

Ejemplo

Contamos un sistema para un supermercado, donde consideramos el siguiente diagrama de clases parcial:

Figura 1: *Diagrama de clases parcial*

Tenemos la necesidad de crear una instancia de *Pago* y asociarla con la *Venta*, y nos planteamos la siguiente pregunta: ¿Qué clase debería ser responsable de esto? Puesto que un *Registro* "registra" un *Pago* en el dominio del mundo real, el patrón Creador sugiere el *Registro* como candidata para la creación del *Pago*. La instancia de registro, pasaría entonces el mensaje de *añadirPago* a la *Venta*, pasando el nuevo *Pago* como parámetro, como vemos en el siguiente diagrama de interacción:

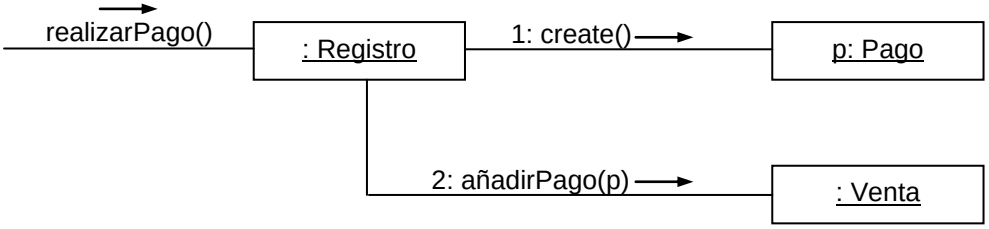


Figura 4.1.2: *Diagrama de interacción 1.*

Esta asignación de responsabilidades acopla la clase *Registro* con el conocimiento de la clase *Pago*. Una posible solución alternativa para crear el *Pago* y asociarlo con la *Venta* es la siguiente:

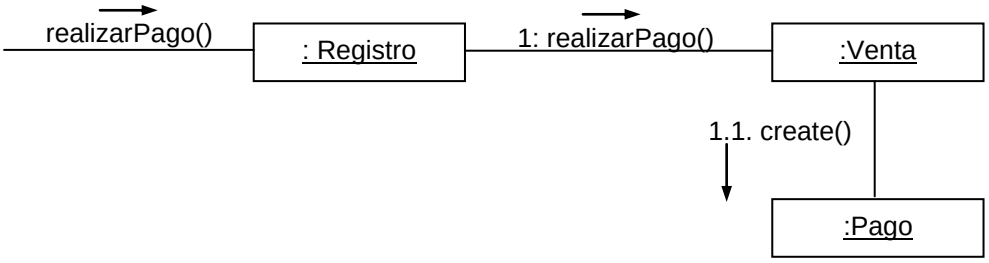


Figura 3: *Diagrama de interacción 2.*

Tenemos que tener en cuenta que, en ambos diseños, la *Venta* debe finalmente acoplarse con el conocimiento del *Pago*. En el primer diseño, en el que el *Pago* es creado por el *Registro*, se añade acoplamiento entre el *Registro* y el *Pago*, mientras que en el segundo diseño, en el que la *Venta* lleva a cabo la creación del *Pago*, no incrementa el acoplamiento. Teniendo en cuenta

únicamente el acoplamiento, es preferible el segundo diseño, ya que el nivel de acoplamiento es mucho más bajo.

Beneficios

- Los cambios que se realicen no afectan a otros componentes.
- Fácil de entender de manera aislada.
- Conveniente para reutilizar

2. Patrón Alta Cohesión.

Este patrón lo que busca es que en nuestro diseño cada elemento debe realizar una labor única dentro del sistema, y no desempeñada por el resto de los elementos. De forma que asigna una responsabilidad de manera que la cohesión permanezca alta, entendiendo por cohesión como una medida de la fuerza con la que se relacionan los objetos o de la cantidad de trabajo que realizan. De esta forma, una clase que tiene una cohesión baja, hace muchas cosas que no tiene relación entre sí; estas clases con baja cohesión suelen ser clases demasiado abstractas a las que se les han asignado demasiadas responsabilidades, y conllevan las siguientes dificultades:

- Son difíciles de entender.
- Son difíciles de reutilizar.
- Son difíciles de mantener.
- Se ven muy afectadas a la hora de realizar cambios en el sistema.

Una clase con alta cohesión tiene un número relativamente pequeños de métodos, con funcionalidad altamente relacionada, no realiza mucho trabajo, y colabora con otros objetos para compartir el esfuerzo si la tarea es demasiado grande. Este tipo de clases anulan todas las dificultades que se nos planteaban con una cohesión baja, ya que por su pequeño número de métodos resultan fáciles de mantener, de entender y de reutilizar.

Al igual que ocurría con el patrón Bajo Acoplamiento, durante toda la fase de diseño, hay que tener muy en cuenta el patrón de Cohesión para todas las decisiones que se tomen.

A continuación comentamos distintos niveles de cohesión:

1. *Muy Baja Cohesión*: donde una única clase es responsable de muchas cosas en áreas funcionales muy diferentes.
2. *Baja Cohesión*: una única clase tiene la responsabilidad de una tarea compleja en un área funcional.

3. *Alta Cohesión*: una clase tiene una responsabilidad moderada en un área funcional y colabora con otras clases para llevar a cabo las tareas.
4. *Moderada Cohesión*: una clase tiene responsabilidades ligeras y únicas en unas pocas áreas diferentes que están lógicamente relacionadas entre sí, con el concepto de la clase, pero no entre ellas.

Como podemos ver hasta ahora, la cohesión y el acoplamiento están fuertemente relacionados y, normalmente, una mala cohesión causa un mal acoplamiento, y viceversa.

Existen pocos casos en los que una baja cohesión sea aceptada justificadamente, pero existen casos en los que debido a implicaciones de costes o de rendimientos ocurra. Es el caso por ejemplo de los objetos servidores distribuidos en los que a veces es deseable crear menos objetos servidores, de mayor tamaño y con menos cohesión para que proporcionen una interfaz con muchas operaciones.

Ejemplo

Utilizamos el mismo ejemplo empleado en el apartado anterior. Se considera la necesidad de crear una instancia de *Pago* (en efectivo) y asociarla con una *Venta*, entonces se plantea qué clase debería ser la responsable de esto. Puesto que el *Registro* registra un *Pago* en el dominio del mundo real, el patrón *Creador* sugiere que *Registro* como candidato para la creación de *Pago*. La instancia de *Registro* podría entonces enviar un mensaje *añadirPago* a la *Venta*, pasando como parámetro el nuevo *Pago*, como podemos ver en la siguiente figura:

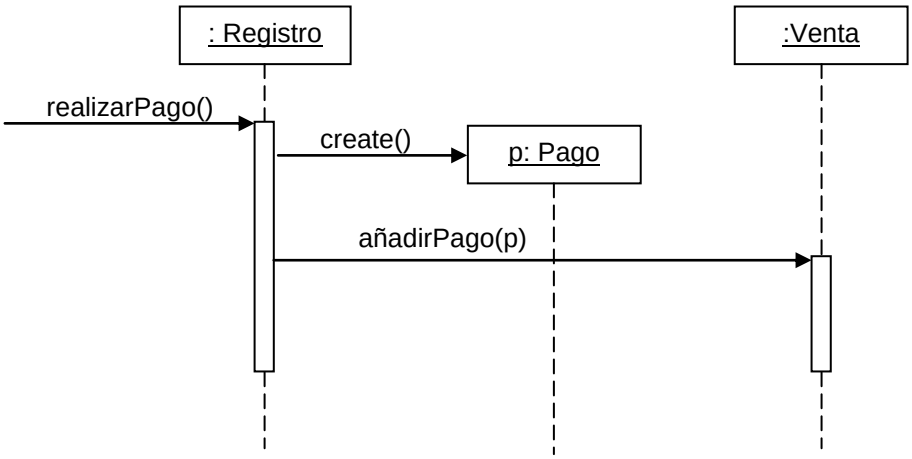


Figura 4: *Diagrama de secuencia 1*.

En esta asignación de responsabilidades, el *Registro* es el que toma la responsabilidad de realizar un *Pago* y de llevar a cabo la operación *añadirPago*.

Este ejemplo aislado, no plantea ningún problema, pero si la clase *Registro* sigue adquiriendo más responsabilidades y trabajo, la cohesión disminuirá. Por ejemplo supongamos que hubiera cincuenta operaciones del sistema, todas recibidas por *Registro*. Si hace trabajo relacionado con cada una de ellas, se convertirá en un objeto sin cohesión. En este caso se plantea el siguiente diseño como solución:

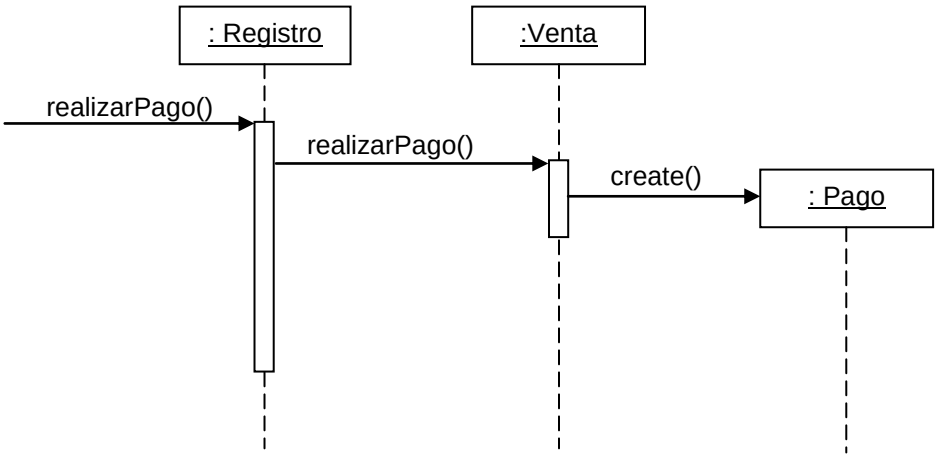


Figura 5: *Diagrama de secuencia 2.*

Beneficios

- Se incrementa la claridad y facilita la comprensión del diseño.
- Se simplifica el mantenimiento y las mejoras.
- Se soporta a menudo bajo acoplamiento.
- Se incrementa la reutilización, porque una clase con alta cohesión se puede utilizar para un propósito muy específico.

3. Patrón Experto o Experto en Información.

Este patrón se centra en asignar la responsabilidad de realizar una tarea determinada, a aquel objeto que tiene la información (atributos) necesaria para ello. Este objeto expresa la "intuición" común de que los objetos hacen el trabajo relacionado con la información que tienen.

A menudo nos encontramos con que la información necesaria para realizar una tarea o responsabilidad se encuentra dispersa en diferentes clases de objetos. Cada vez que la

información se encuentre en varios objetos diferentes, necesitarán interactuar mediante el paso de mensajes para compartir el trabajo.

En algunas ocasiones la solución que sugiere el patrón Experto no es la más conveniente, porque normalmente ocasiona problemas de acoplamiento y de cohesión. Por lo que hay que tener en cuenta que el patrón Experto es aplicable mientras que estemos considerando los mismos aspectos del sistema:

- Lógica de negocio.
- Persistencia a la base de datos
- Interfaz de usuario.

No tiene sentido considerar que una clase se debe escribir a sí misma en la base de datos o formatearse para presentarse en una página HTML por el hecho de poseer datos. Estos son elementos estructuralmente distintos y deben considerarse desde una perspectiva distinta.

Ejemplo

Continuamos con el ejemplo que hemos estado manejando hasta ahora, en la que algunas clases necesitan conocer el total de una venta. Para comenzar a establecer las responsabilidades nos planteamos la siguiente pregunta: ¿Quién debería ser el responsable de conocer el total de la venta? Pues siguiendo al Patrón Experto, deberíamos buscar las clases de objetos que contienen la información necesaria para determinar el total.

Ahora llegamos a una pregunta clave: ¿Miramos en Modelo de Dominio o en el Modelo de Diseño para analizar las clases que tienen la información necesaria? El *Modelo de Dominio* representa clases conceptuales del dominio del mundo real; el *Modelo de Diseño* representa las clases software. Para responder tenemos en cuenta:

1. Si hay clases relevantes en el Modelo de Diseño, mire ahí primero
2. Si no, mire en el Modelo de Dominio, e intente utilizar (o ampliar) sus representaciones para inspirar la creación de las correspondientes clases de diseño.

Para nuestro ejemplo suponemos, que acabamos de comenzar el trabajo de diseño y no hay más que un Modelo de Diseño mínimo. Por tanto, miramos en el Modelo de Dominio, representado en la Figura 6, buscando expertos en información; quizás la *Venta* del mundo real es uno. Entonces añadimos una clase software al Modelo de Diseño denominada *Venta*, y le otorgamos la responsabilidad de conocer su total, representado con el método llamado *getTotal*.

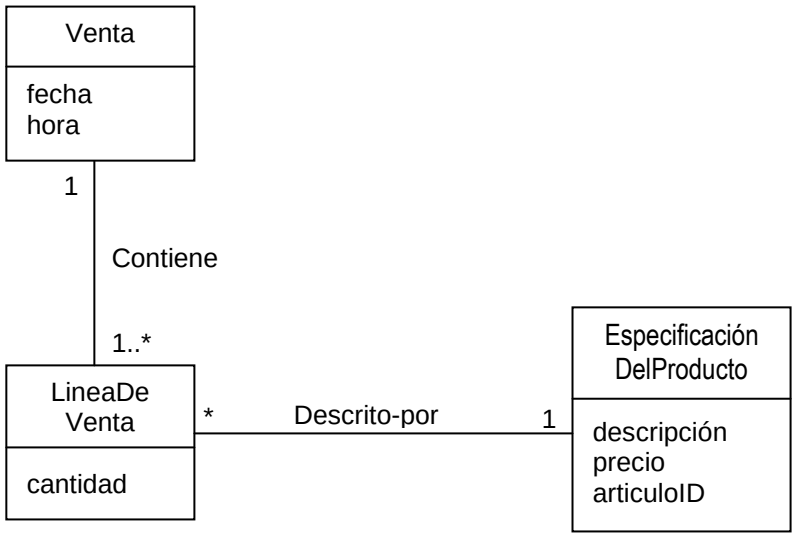


Figura 6: *Modelo de Dominio parcial.*

Para determinar el total es necesario conocer todas las instancias de *LineaDeVenta* de una venta y la suma de sus subtotales. Una instancia de *Venta* las contiene; por tanto según el Patrón Experto de Información, la clase de objeto *Venta* es adecuada para esta responsabilidad. Podemos representar algunas de estas responsabilidades en los siguientes diagramas de interacción:

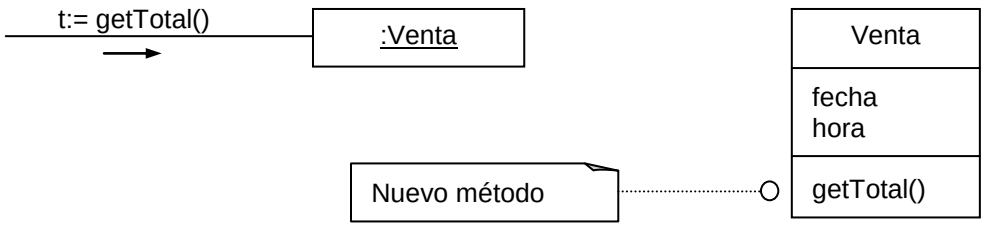


Figura 7: *Diagramas de interacción y de clases parciales.*

Pero para continuar se necesita saber el subtotal de la *LineaDeVenta*, para ello se necesitan *LineaDeVenta.cantidad* y *EspecificacionDelProducto.precio*. La *LineaDeVenta* conoce su cantidad y la *EspecificacionDelProducto* asociada; por lo tanto siguiendo el patrón Experto, la *LineaDeVenta* debería determinar el subtotal, por lo que la clase *Venta* enviaría un

mensaje *getSubtotal* a cada una de las *LineasDeVenta* y sumaría el resultado, como se muestra en el siguiente diagrama:

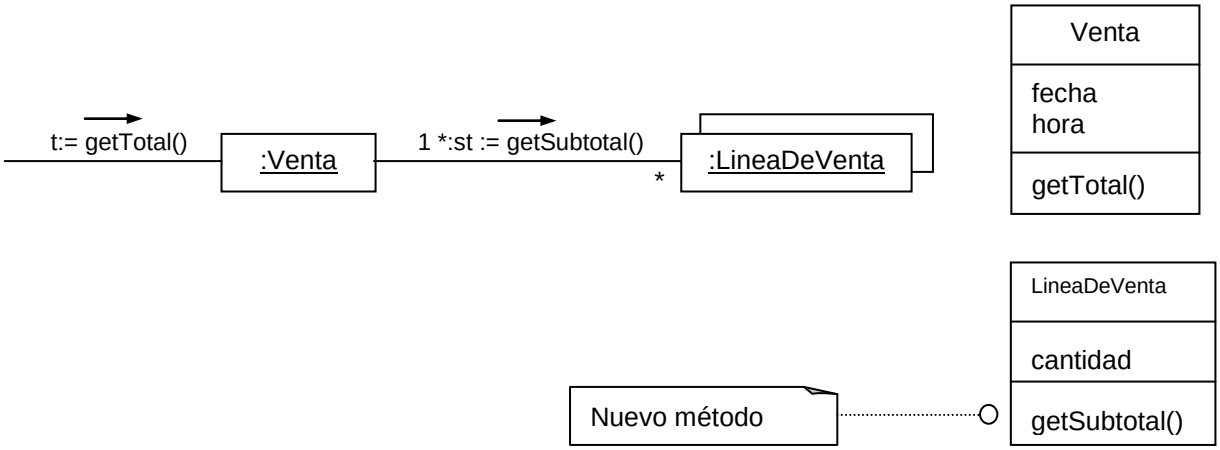


Figura 8: *Diagrama de interacción 1. Cálculo del total.*

La *EspecificacionDelProducto* será el responsable de proporcionar el precio del producto, por lo que se debe enviar un mensaje solicitando su precio, como se muestra en la siguiente figura:

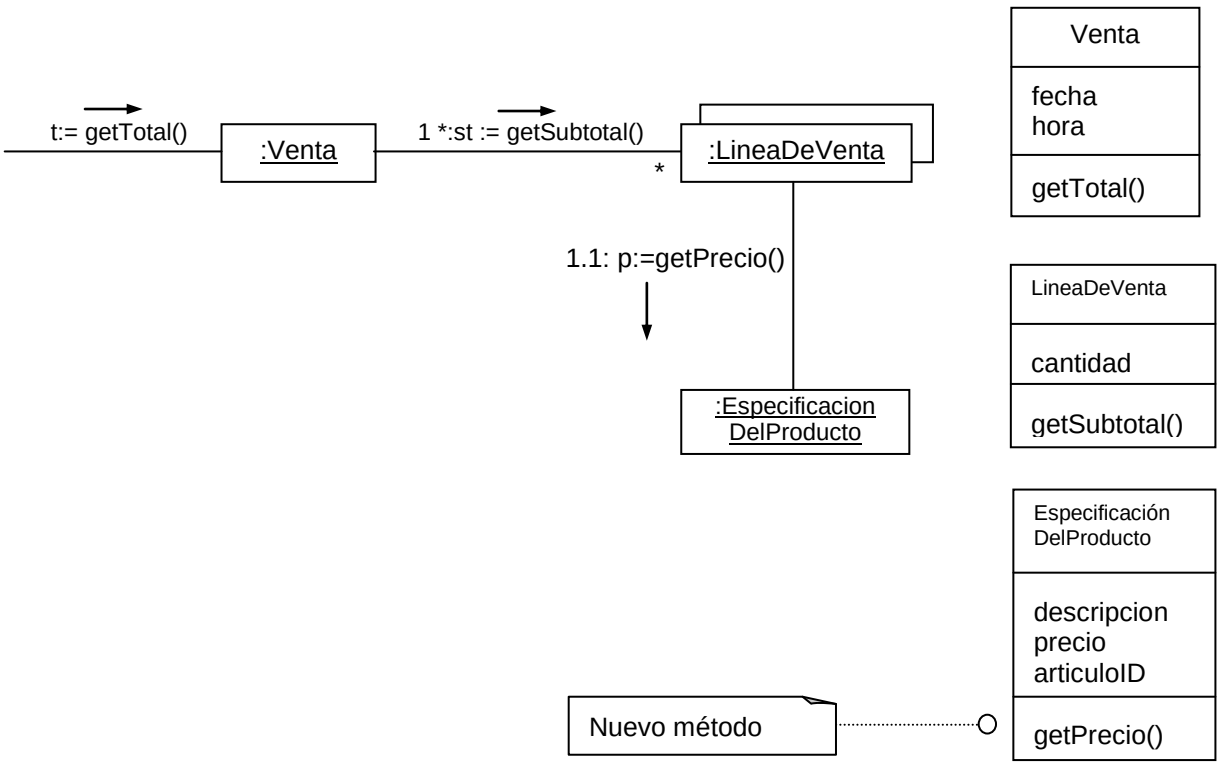


Figura 9: *Diagrama de interacción 2. Cálculo del total.*

En resumen, siguiendo el patrón Experto, para realizar la tarea de conocer y proporcionar el total de una venta se asignaron tres responsabilidades a tres clases de diseño de objetos, de la siguiente manera:

Clase de Diseño	*Responsabilidad*
Venta	Conocer el total de la venta.
LineaDeVenta	Conocer el subtotal de la línea de venta.
EspecificacionDelProducto	Conocer el precio del artículo.

Beneficios

- Se mantiene el encapsulamiento de la información, puesto con los objetos utilizan su propia información para llevar a cabo las tareas. Normalmente esto conlleva un Bajo Acoplamiento, lo que da lugar a sistemas más robustos y más fáciles de mantener.
- Se distribuye el comportamiento entre las clases que contienen la información requerida, por tanto, se estimula las definiciones de clases más cohesivas que son más fáciles de entender y mantener. Se soporta normalmente una Alta Cohesión.

4. Patrón Creador.

Este patrón asigna a la clase B la responsabilidad de crear una instancia de la clase A (B es *Creador* de los objetos A) si:

1. B *contiene* objetos de A
2. B *agrega* objetos de A
3. B *registra* instancias de objetos de A
4. B *tiene los datos de inicialización* de A (datos que requiere su constructor)
5. B *utiliza* más estrechamente datos de A

En el análisis y diseño orientado a objetos nos encontramos con el problema de asignar responsabilidades a la hora de crear instancias de objetos, ya que hay que salvaguardar el bajo acoplamiento de los sistemas, así como su claridad y encapsulación. El patrón creador guía la asignación de responsabilidades relacionadas con la creación de objetos, seleccionando como

objeto creador aquél que necesite conectarse al objeto creado en alguna situación de las que hemos mencionado anteriormente, para conseguir un bajo acoplamiento.

Según el patrón Creador, las relaciones *contiene* y *registra*, son las mejores candidatas, en principio, para asignarlas la responsabilidad de crear objetos. En cuanto a la relación *agrega*, podemos decir que involucra cosas que se encuentran en la relación Todo-Parte o Ensamblaje-Parte, como por ejemplo "Párrafo agrega Frase", entonces varias frases crean un párrafo, por lo que se podría aplicar también el patrón creador.

Ejemplo

Continuando con el ejemplo que estamos manejando, ¿quién debería ser el responsable de crear una instancia de *LineaDeVenta*? Para contestar a esta pregunta buscamos aquellos objetos que tengan la necesidad de conectarse en algún momento a *LineaDeVenta*, para ello consideramos el siguiente Modelo de Dominio:

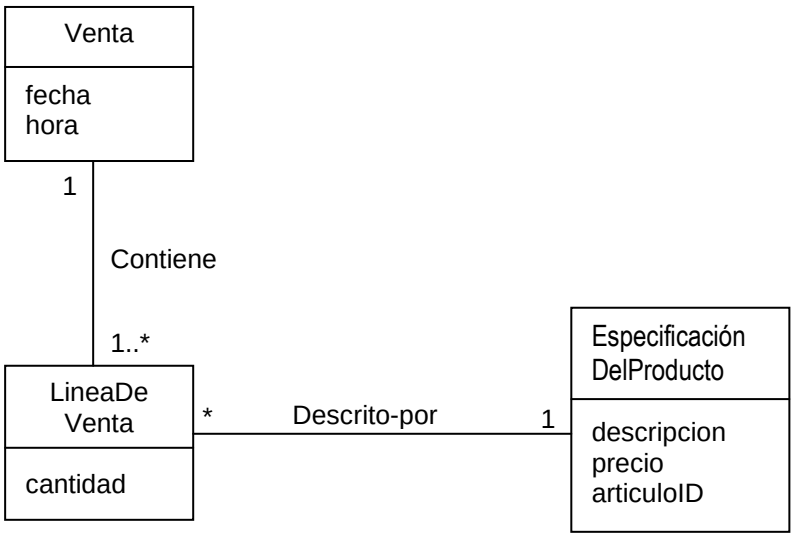


Figura 10: *Modelo del Dominio parcial.*

Vemos en el diagrama de dominio que entre *Venta* y *LineaDeVenta* se da la relación *Contiene*, por lo que *Venta* puede ser el creador de *LineaDeVenta*. Para esta asignación de responsabilidades se requiere que en la clase *Venta* se defina el método *crearLineaDeVenta*, como podemos ver en el diagrama de interacción 11.

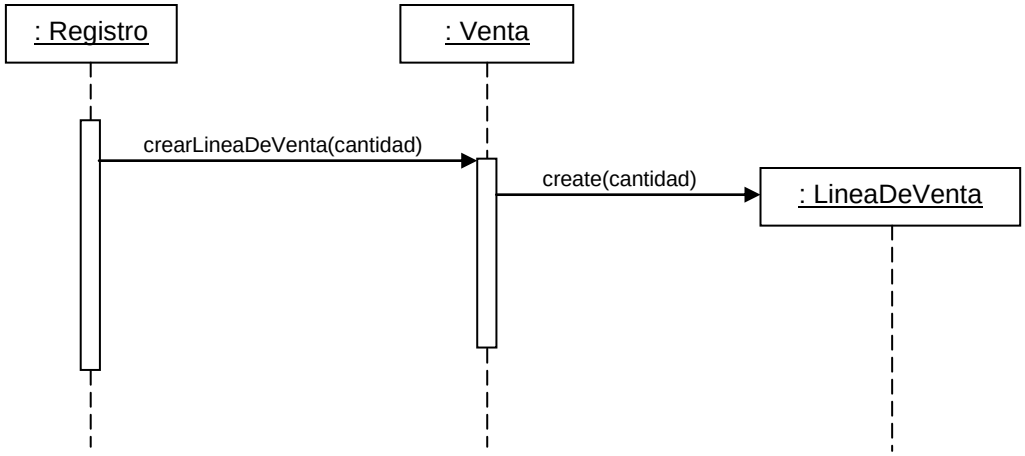


Figura 11: *D. Interacción. Creación de LineaDeVenta.*

Beneficios

- Favorece el Bajo Acoplamiento, lo que conlleva menos mantenimiento y más reutilización.

5. Patrón Controlador.

El patrón Controlador se encarga de asignar la responsabilidad de controlar el flujo de eventos del sistema a clases específicas. El objeto controlador no será el que realice estas actividades, sino que las delegará en otras clases con las que mantiene un modelo de alta cohesión. También podemos decir que el patrón Controlador asigna la responsabilidad de recibir o manejar un mensaje de evento del sistema a una clase que representa una de las siguientes opciones:

- Representa el sistema global, dispositivo o subsistema, en cuyo caso se denomina *controlador de fachada*, y son muy utilizados en aplicaciones Web, donde la lógica de presentación y la de negocio están separadas.
- Representa un escenario de caso de uso donde tiene lugar el evento del sistema, que a menudo recibe el nombre de: <NombreDelCasoDeUso>Manejador, <NombreDelCasoDeUso>Coordinador, <NombreDelCasoDeUso>Sesion (*Controlador de sesión o de caso de uso*). En este caso es conveniente seguir las siguientes recomendaciones:
    - ❖ Utilice la misma clase controlador para todos los eventos del sistema en el mismo escenario de caso de uso.

- ❖ Hay que considerar la sesión como una instancia de una conversación con un actor. Las sesiones pueden tener cualquier duración, pero se organizan frecuentemente en función de los casos de uso (sesiones de casos de uso).

Tenemos que tener cuidado con la asignación de responsabilidades relacionadas con los eventos, ya que hay clases como "ventana", "applet", "vista", "documento", etc. que no deberían abordar tareas asociadas con los eventos del sistema, ya que estas clases, normalmente, reciben los eventos y los delegan a un controlador. Los objetos interfaz y la capa de presentación no deberían ser responsables de llevar a cabo los eventos del sistema.

Pero todo esto nos plantea un problema a la hora de determinar quién será el responsable de un evento de entrada al sistema. Un evento de entrada al sistema es un evento generado por un actor externo, y se asocian con operaciones del sistema de igual forma que se relacionan los mensajes y los métodos. Por ejemplo: cuando un usuario utiliza un procesador de texto y pulsa el botón "comprobar ortografía", está generando un evento del sistema que indica que se "ejecute una comprobación de la ortografía".

Un *Controlador* es un objeto que no pertenece a la interfaz de usuario, que es el responsable de recibir o manejar un evento del sistema. Un controlador define el método para la operación del sistema. Además no debe de realizar mucho trabajo, sino que delega en otros objetos el trabajo que se necesita hacer, solamente coordina y controla la actividad.

A la hora de escoger algún manejador de eventos para un sistema orientado a objetos, el patrón Controlador nos proporciona una serie de guías a cerca de las opciones generalmente aceptadas:

- *Controlador de fachada:* representan al sistema global, dispositivo o subsistema. Se debe elegir algún nombre de clase que sugiera una cubierta, o fachada sobre las otras capas de la aplicación, y que será la encargada de proporcionar las llamadas a los servicios más importantes desde la capa de Interfaz de Usuario hacia las otras capas. Esta clase podría ser una abstracción de toda una unidad física, o de un sistema software completo (como por ejemplo una biblioteca Virtual: *clase BibliotecaVirtual*). Estos controladores de fachada resultan muy útiles cuando no hay demasiados eventos de sistema, o cuando no es posible que la Interfaz de Usuario envíe mensajes de eventos del sistema a controladores alternativos.
- *Controlador de Casos de Uso:* hay un controlador diferente para cada caso de uso, que será una especie de construcción artificial para dar soporte al sistema, en el ejemplo que estamos manejando contiene un caso de uso, por ejemplo, que es *Procesar Venta*, entonces podrá haber una clase controlador *ProcesarVentaManejador*. Este tipo de controlador resulta muy apropiado cuando

el uso de un controlador de fachada nos conlleva diseños con baja cohesión o alto acoplamiento, por la sobrecarga de responsabilidades que pueda tener asignadas el controlador de fachada. El controlador de casos de uso resulta muy eficaz cuando hay muchos eventos del sistema repartidos en diferentes procesos.

Por tanto, un Controlador la solicitud del servicio desde la capa de Interfaz de Usuario y coordina su realización, normalmente delegando en otros objetos.

Ejemplo

En el ejemplo que estamos viendo hasta ahora, hay varias operaciones del sistema asociadas a los eventos del sistema, como vemos en la siguiente figura:

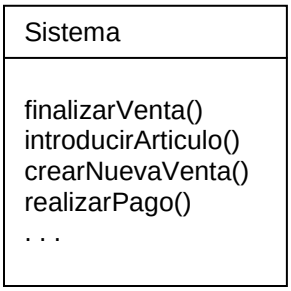


Figura 12: *Operaciones del sistema.*

Durante el análisis, las operaciones del sistema pueden asignarse a la clase *Sistema*, pero esto no significa que una clase software llamada *Sistema* las lleve a cabo durante el diseño, sino que se le asigna la responsabilidad de las operaciones a una clase Controlador desconocida por ahora, como podemos ver en la figura 13.

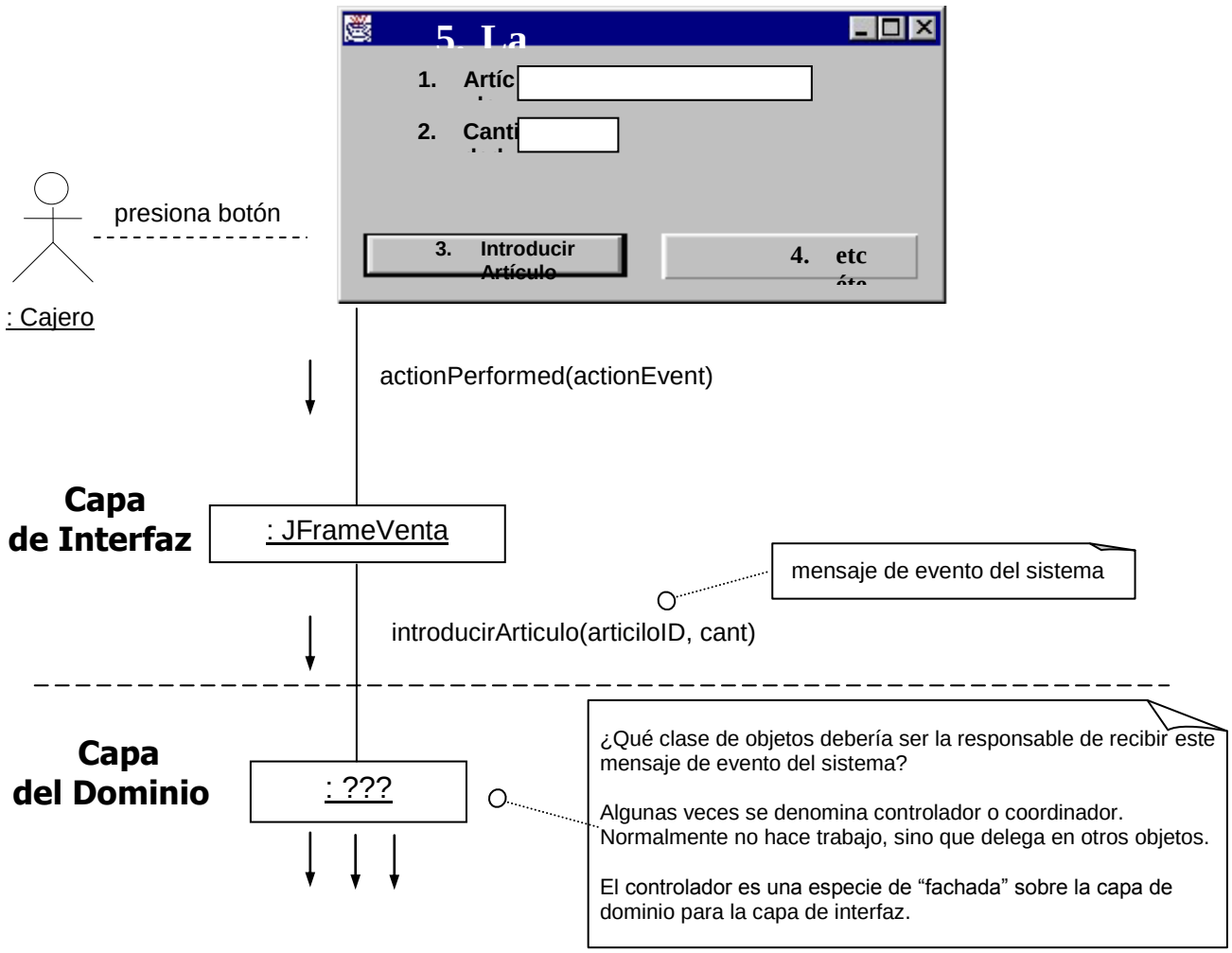


Figura 13: *¿Controlador para introducir Artículo?*

¿Quién debería ser el controlador de eventos del sistema para *introducirArticulo* y *finalizarVenta*? Siguiendo el patrón Controlador, a continuación presentamos algunas de las opciones:

Representa el "sistema" global, dispositivo o subsistema	*Registro, Sistema*
Representa un receptor o manejador de todos los eventos del sistema de un escenario de un caso de uso	*ProcesarVentaManejador* *ProcesarVentaSesion*

Podemos ver con los siguientes diagramas de interacción, cual de los ejemplos podría sernos de utilidad:

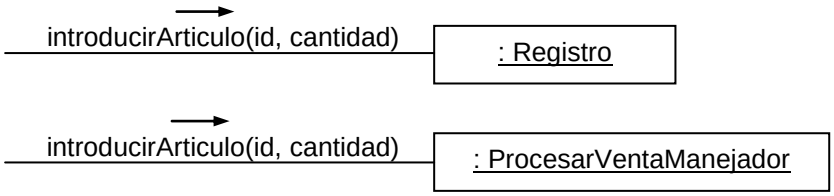


Figura 14: *Opciones de Controlador*

Teniendo en cuenta todos los factores que hemos estado viendo en este apartado para la elección del controlador más apropiado, la asignación de operaciones del sistema podría quedar como se muestra en la Figura 15.

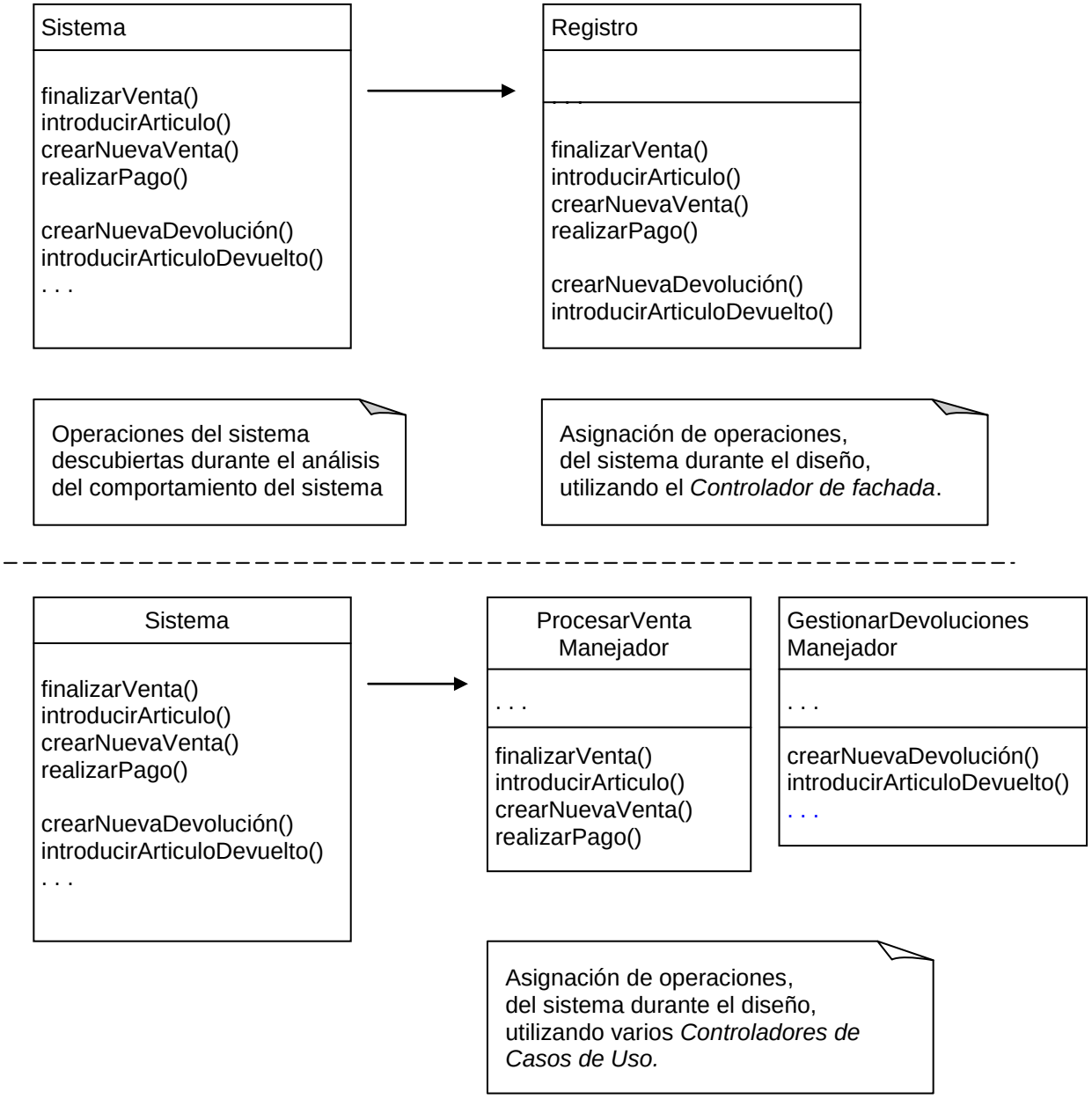


Figura 15: *Asignación de operaciones del sistema*.

Beneficios

- Aumenta el potencial para reutilizar las interfaces conectables, ya que el patrón controlador asegura que la lógica de la aplicación no se maneja en la capa de interfaz.
- Proporciona razonamiento sobre el estado de los casos de usos, ya que en muchas ocasiones es necesario que las operaciones tengan lugar en una secuencia válida, o ser capaces de razonar sobre el estado actual de la actividad y operaciones de casos de uso que están en marcha.

6. Patrón Polimorfismo.

El patrón Polimorfismo se basa en asignar la responsabilidad para el comportamiento, utilizando el polimorfismo[1], cuando las alternativas y comportamientos relacionados varíen según el tipo de clases. No es conveniente implementar comportamientos alternativos con sentencia IF-ELSE, para hacer comprobaciones acerca del tipo del objeto, ya que lo único que se consigue es limitar la reutilización y el crecimiento del sistema.

Llegados a este punto se nos plantean dos problemas: por un lado manejar las alternativas basadas en el tipo, y por otro cómo crear componentes software conectables:

- *Alternativas basadas en el tipo*: si diseñamos nuestros sistemas basándonos en las sentencias condicionales IF-THEN o CASE para comprobar el tipo de un objeto, cuando aparezcan variaciones o modificaciones, se requerirán modificaciones en la lógica de los casos. Por ejemplo, imaginemos que tenemos una aplicación que nos muestra unos determinados mensajes en distintos idiomas... con IF, al aumentar el número de idiomas, nos obligaría a añadir un nuevo IF; mientras que con el polimorfismo nos limitaríamos a crear un objeto de una clase polimórfica nueva.
- *Componentes software conectables*: cómo podemos hacer las modificaciones en relaciones cliente - servidor para que al sustituir un componente en el servidor por otro no afecte al cliente.

Según el patrón Polimorfismo, un diseño basado en la asignación de responsabilidades puede extenderse fácilmente para manejar nuevas variaciones.

Hay que indicar que los desarrolladores y analistas no deben gastar esfuerzos innecesarios a la hora de diseñar sistemas con interfaces y polimorfismo para futuras

[1] Entendemos por **polimorfismo**, asignar el mismo nombre a servicios en diferentes objetos cuando los servicios son parecidos o están relacionados.

necesidades especulativas, a no ser que una evaluación crítica de esas necesidades futuras nos lleve a la conclusión de que sí es necesario un diseño con polimorfismo.

Ejemplo

En nuestro ejemplo de tienda virtual, se deben soportar diferentes sistemas externos para el cálculo de impuestos (como Master-En-Impuestos e Impuestos - Pro), por lo que el sistema necesita ser capaz de integrarse con varios de ellos. Cada calculador de impuestos tiene una interfaz diferente, por lo que hay un comportamiento parecido pero que varía para adaptarse a cada uno de estos interfaces externos o API´s. Un producto podría soportar un simple protocolo de sockets TCP, otro podría ofrecer un interfaz SOAP, y un tercero podría soportar una interfaz RMI en Java.

Para que el sistema soporte estos subsistemas externos, debemos averiguar a qué objetos debemos asignarles la responsabilidad de manejar estas interfaces diferentes de cálculo de impuestos. Puesto que el comportamiento para la adaptación del calculador varía según el tipo de calculador, según el patrón Polimorfismo, debemos asignar esta responsabilidad a los diferentes objetos calculadores, implementando una operación polimórfica *getImpuestos*.

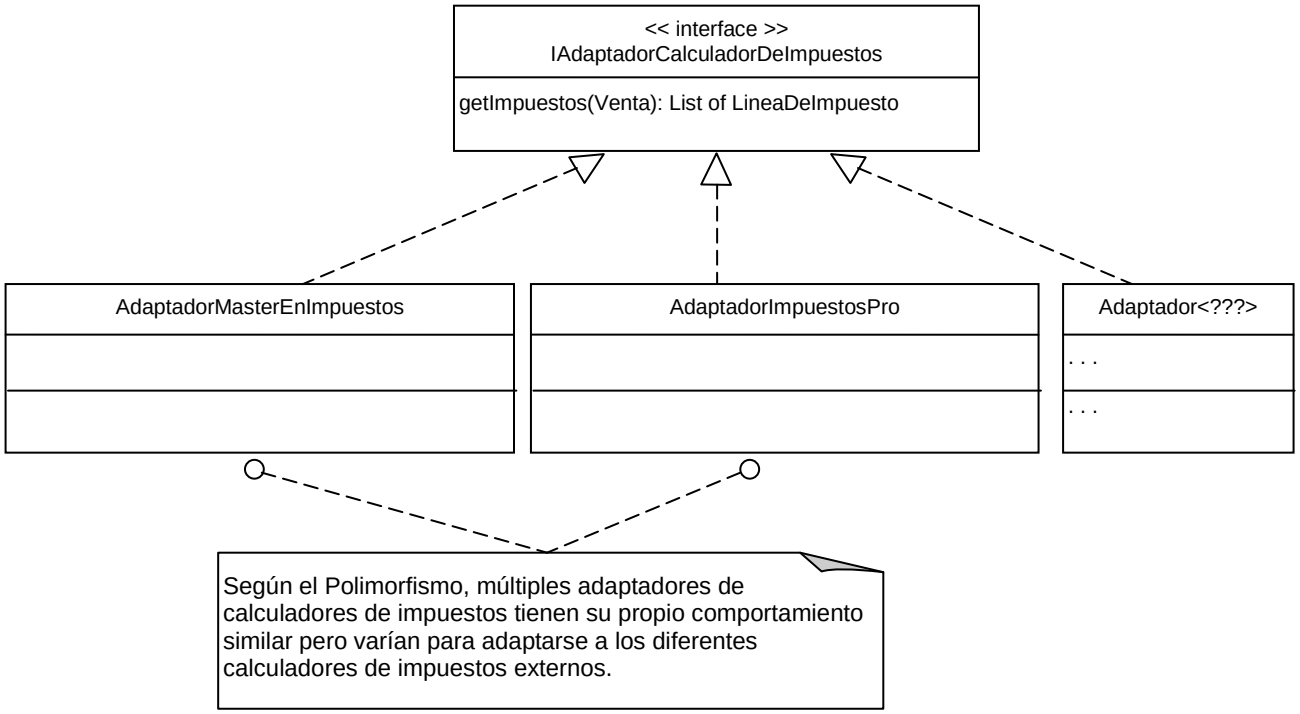


Figura 16: *Polimorfismo en adaptación a diferentes calculadores.*

Beneficios

- Se añaden fácilmente las extensiones para las futuras variaciones que se puedan llevar a cabo.
- Las nuevas implementaciones se pueden producir sin afectar a los clientes.

7. Patrón Fabricación Pura.

Este patrón es muy utilizado cuando los problemas se complican, ya que construye clases que se encargan de construir los objetos adecuados en cada momento (factorías). Estas clases *factorías* son clases artificiales que no representan ningún concepto del dominio del problema, tienen un conjunto de responsabilidades altamente cohesivo y tienen un bajo acoplamiento, con lo que se consigue un diseño limpio y puro, de ahí el nombre de *Fabricación Pura*.

El patrón de Fabricación Pura resulta muy útil en el diseño orientado a objetos, ya que en muchas ocasiones, representamos conceptos del mundo real con clases software, por ejemplo: una clase *Venta* y *Cliente*. Pero esta representación en algunas ocasiones provoca conflictos a la hora de asignar responsabilidades, ocasionando problemas de cohesión, acoplamiento y reutilización.

El diseño de objetos se puede dividir en general en dos grupos:

1. Los escogidos según una **descomposición de la representación**:

   Son aquellos objetos donde la clase software representa una cosa en el dominio del sistema, por ejemplo: la clase *Venta*.

2. Los escogidos según una **descomposición del comportamiento**:

   En algunas ocasiones deseamos asignar responsabilidades agrupando comportamientos o algoritmos, sin que interese crear una clase con un nombre que la relacione con el dominio del mundo real, por ejemplo: un desarrollador crea un objeto "algoritmo" llamado *GeneradorDeTablaDeContenidos*, cuyo objetivo es generar una tabla de contenidos, y la crea como clase de ayuda o de conveniencia para agrupar algunos comportamientos o métodos.

El patrón de Fabricación Pura[2] se inspira en una descomposición del comportamiento, y se basa en la creación de clases de conveniencia para agrupar algún comportamiento común necesario para el diseñador, son objetos centrados en la función o de comportamiento.

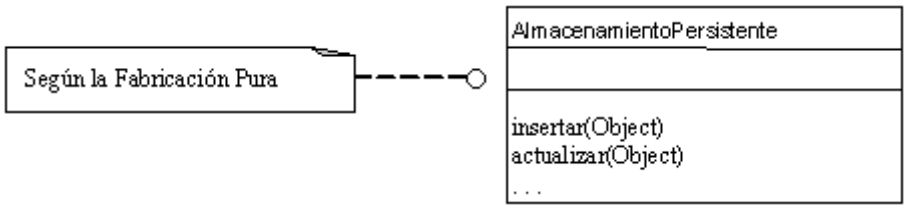
Ejemplo

En nuestro ejemplo de tienda virtual, suponemos que se necesita soporte para almacenar las instancias de *Venta* en una base de datos relacional. Si considerásemos el patrón Experto, podríamos asignar esta responsabilidad a la propia clase *Venta*, puesto que tiene los datos que necesitamos almacenar, pero aparecen los siguientes inconvenientes:

[2] Según el libro "UML y Patrones" prácticamente el resto de patrones de diseño son Fabricación Pura.

- La clase *Venta* deja de tener Cohesión: ya que la tarea requiere un amplio número de operaciones relacionadas con base de datos, que no tienen ninguna relación con el concepto de ventas.
- Aumenta el acoplamiento: la clase *Venta* tiene que acoplarse a la interface de la base de datos relacional, como por ejemplo JDBC en las tecnologías Java.
- Almacenar los objetos en una base de datos relacional es una tarea muy general, ya que otros objetos pueden necesitar este soporte también. Si asignamos esta responsabilidad a la clase *Venta* y por algún motivo necesitamos almacenar otros objetos tendremos que duplicar, con lo que la reutilización disminuye.

Por tanto, aunque la clase *Venta* es una buena candidata para asignar esta responsabilidad según el patrón Experto, vemos que conlleva algunos inconvenientes. Una solución razonable es recurrir al patrón de Fabricación Pura y crear una nueva clase llamada *AlmacenamientoPersistente* que sea responsable únicamente de almacenar los objetos en una base de datos relacional.



Con el patrón de Fabricación Pura conseguimos que:

- *Venta* permanece bien diseñada, con alta cohesión y bajo acoplamiento.
- *AlmacenamientoPersistente* es relativamente cohesiva, ya que tiene como única responsabilidad el almacenamiento e inserción de objetos en la base de datos.
- *AlmacenamientoPersistente* es un objeto muy genérico y reutilizable.

Beneficios

- Favorece la Alta Cohesión, las responsabilidades se centran en un conjunto muy específico de tareas relacionadas.
- Hay mayor potencial de reutilización, ya que las clases de Fabricación Pura, debido al conjunto tan específico de tareas que tiene asignadas, son muy reutilizables en otras aplicaciones.

8. Patrón Indirección.

Este patrón se basa en la creación de clases intermedias para desacoplar componentes o servicios, o asigna la responsabilidad a un objeto intermedio que medie entre dos componentes o servicios de manera que no se acoplen directamente.

En los diseños orientados a objetos se tiene en cuenta un viejo dicho que dice que:

"*la mayoría de los problemas en informática se pueden resolver mediante otro nivel de indirección*"

Muchos de los patrones existentes son especificaciones del patrón Indirección, además de que muchas Fabricaciones Puras se generan debido a la Indirección.

Ejemplo

Los objetos *AdaptadorCalculadorDeImpuestos*¸ que aparecían en nuestro ejemplo del apartado 6, actúan como intermediarios con los calculadores de impuestos externos. Mediante el polimorfismo, proporcionan una interface consistente para los objetos internos y ocultan las variaciones a las API´s externas. Añadiendo un nivel de indirección y el polimorfismo, los objetos adaptador protegen el diseño interno frente a las variaciones de las interfaces externas, como vemos en la siguiente figura:

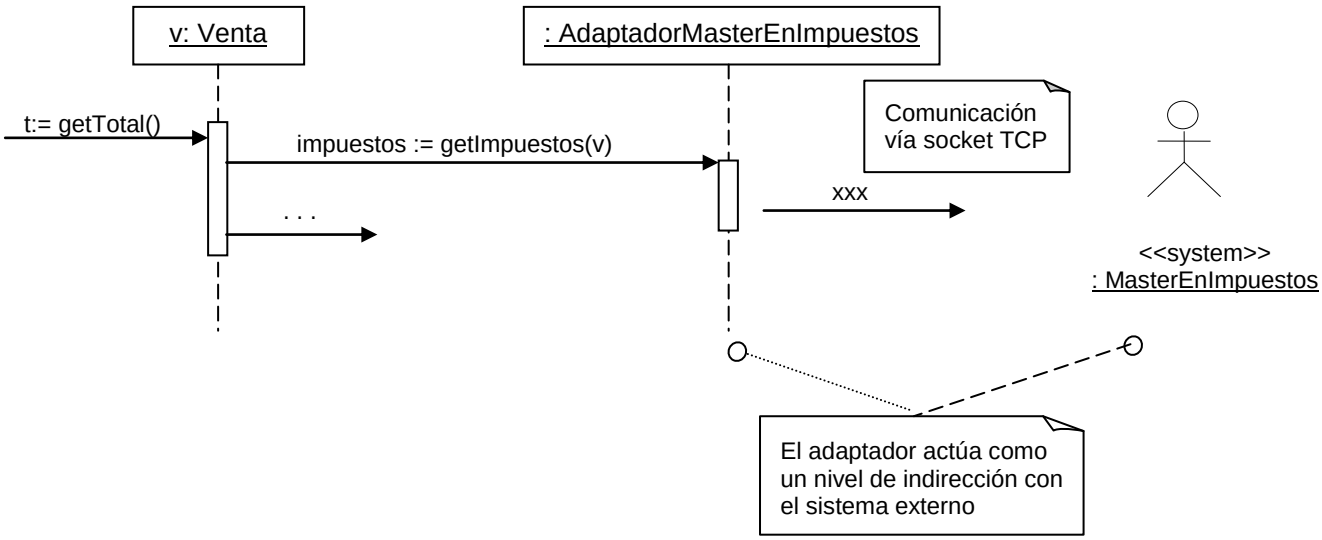


Figura 17: *Ejemplo de Indirección*.

En el ejemplo del apartado 7 tenemos otro ejemplo de Indirección, ya *AlmacenamientoPersistente* actúa como intermediario entre la *Venta* y la base de datos relacional.

Beneficios

- Bajo Acoplamiento
- Muchos intermediarios de Indirección son Fabricaciones Puras.

9. Patrón Variaciones Protegidas.

Este patrón identifica los puntos de variaciones previstas o de inestabilidad, y asigna las responsabilidades para crear una interfaz[3] estable alrededor de ellos.

El patrón Variaciones Protegidas es un importante concepto a tener en cuenta a la hora de proporcionar a un sistema de flexibilidad y protección frente a las variaciones. El patrón Variaciones Protegidas motiva la encapsulación de datos, interfaces, polimorfismo, indirección y estándares.

El patrón de Variaciones Protegidas se aplica tanto a puntos de variación como a puntos de evolución[4]. Pero hay que tener en cuenta que en algunas ocasiones, el coste de diseñar la protección en los puntos de evaluación es más alto que rehacer un diseño simple.

Las Variaciones Protegidas son aplicadas en multitud de diseños, y se aplican una serie de técnicas que resumimos en el siguiente cuadro:

Diseños	***Variaciones***	***Técnicas***
Dirigidos por los datos	Cubren una amplia familia de técnicas: lectura de códigos, valores, rutas de ficheros de clase, nombres de clase, procedentes de una fuente externa para cambiar el comportamiento del sistema en tiempo de ejecución.	Se protege el sistema de este tipo de variaciones registrando externamente la variante, leyéndola y razonando sobre ella.
Búsqueda de servicios	Incluye técnicas como: la utilización de servicios de nombres, intermediarios para obtener un servicio,...	Se protegen los clientes de las variaciones en la ubicación de los servicios, utilizando la interfaz estable de la búsqueda del servicio.
Dirigidos por un intérprete	Comprenden: intérpretes de reglas, intérpretes de script o del lenguaje que leen y ejecutan programas, máquinas virtuales, motores de redes neuronales, motores lógicos,...	Se protege al sistema del impacto de las variaciones lógicas registrando externamente la lógica, leyéndola y utilizando un intérprete.

[3] El término **interfaz** se utiliza en el sentido más general de una vista de acceso, no sólo significa literalmente como una interfaz Java o COM.
[4] **Puntos de variación** son variaciones en el sistema actual, existente o en los requisitos. **Puntos de evolución** son variaciones que podrán aparecer en el futuro, pero que no están presentes en los requisitos actuales.

Reflexivos o de meta-nivel	Comprende el impacto de la lógica o variaciones externas del código	Se protege al sistema mediante algoritmos reflexivos que utilizan introspección y servicios de metalenguage.
Acceso uniforme	Algunos lenguajes permiten expresar de la misma manera el acceso a un método y a un campo.	Como protección podemos cambiar de campos públicos a métodos de acceso, sin cambiar el código del cliente.

Ejemplo

Si retomamos el ejemplo anterior del calculador de impuestos externo y su solución con el Polimorfismo, ilustra las Variaciones Protegidas (Figura 16). El punto de inestabilidad o variación lo forman las diferentes interfaces o API´s de los calculadores de impuestos externos. Nuestro sistema de ventas necesita ser capaz de integrarse con muchos sistemas de cálculo de impuestos existentes, y también con futuros calculadores de terceras partes que no existen todavía.

Añadiendo un nivel de indirección, una interfaz, y utilizando el polimorfismo con varias implementaciones de *IAdaptadorCalculadorDeImpuestos*, se consigue proteger al sistema de variaciones en las API´s externas. Los objetos internos colaboran con una interfaz estable, y las distintas implementaciones del adaptador ocultan las variaciones de los sistemas externos.

Beneficios

- Se añaden fácilmente las extensiones que se necesitan para las nuevas variaciones.
- Se pueden introducir nuevas implementaciones sin afectar a los clientes.
- Se reduce el acoplamiento.
- Puede disminuirse el impacto o coste de los cambios.