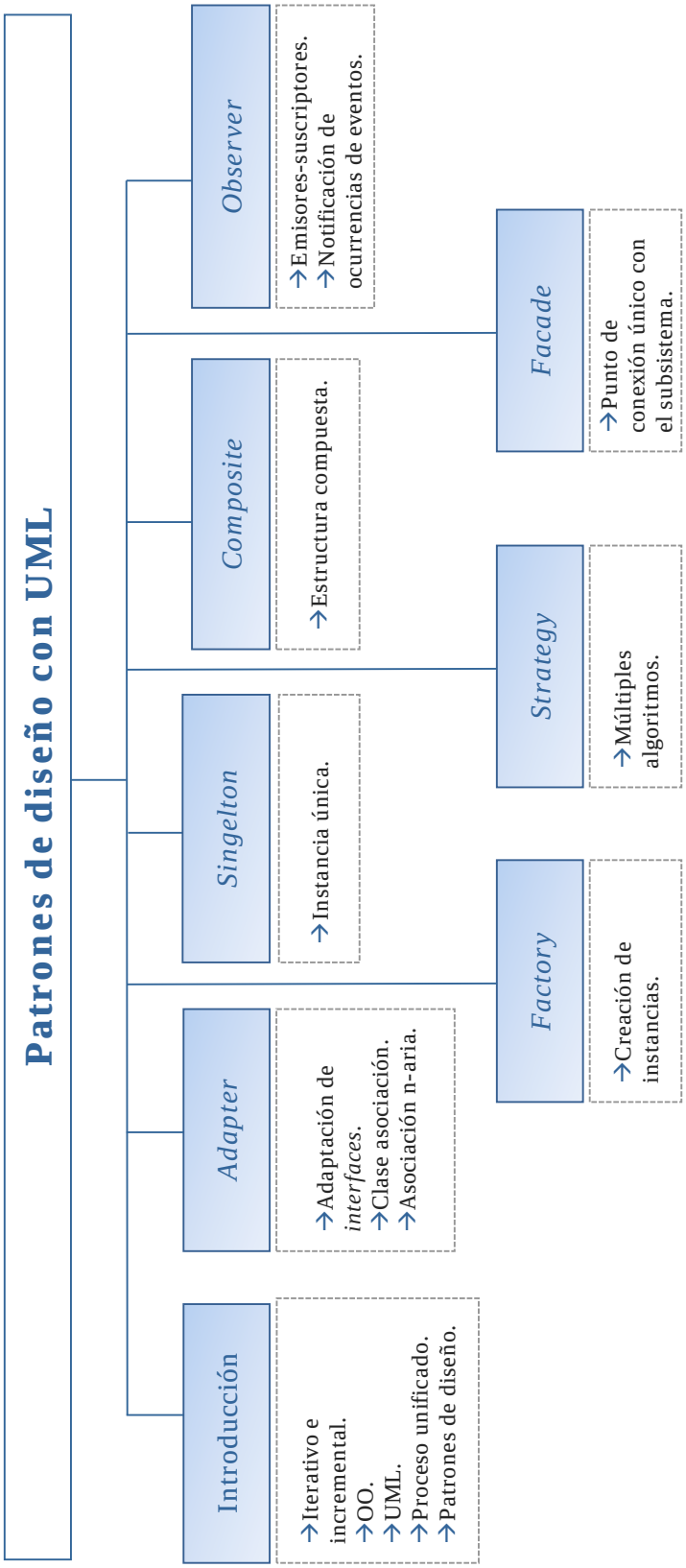
Patrones de diseño con UML

TEMA 6

Esquema



Ideas clave

6.1. ¿Cómo estudiar este tema?

Para estudiar este tema lee las **Ideas clave**, además del **capítulo 21 (páginas 299-304)**, disponible en el aula virtual bajo licencia CEDRO, del libro:
Larman, C. (2003). *UML y Patrones. Una introducción al análisis y diseño orientado a objetos y al proceso unificado. Segunda edición*. Pearson Prentice-Hall.

En este tema se analiza el uso de patrones de diseño «*gang-of-four*» (GoF) en combinación con modelos UML cuando se lleva a la práctica el análisis y diseño orientado a objetos (del inglés *Object-Oriented Analysis and Design*, OOA/D) dentro de un ciclo de vida iterativo e incremental en el desarrollo de sistemas *software* bajo el enfoque del proceso unificado.

6.2. Introducción

Desarrollar *software* en la época actual se ha convertido en una tarea más complicada si cabe, y eso a pesar de la gran cantidad de herramientas, técnicas, etc., que hay disponibles y que siguen apareciendo, todas ellas orientadas a facilitar el desarrollo de sistemas. El problema en gran medida es debido a que el cambio en las necesidades de los clientes crece a un ritmo vertiginoso y los sistemas *software* han de evolucionar y adaptarse a las nuevas necesidades lo más rápidamente posible si la organización que los da soporte no quiere quedarse sin mercado.

Es por ello, que si hace unos años ya se veía la inviabilidad del modelo en cascada para la gran mayoría de proyectos, hoy en día ya ni siquiera se considera como opción de modelo de proceso posible cuando se aborda un proyecto real. Los proyectos se inician y van refinándose y adaptándose a las necesidades del cliente (o incluso de otros muchos clientes potenciales), por lo que el **modelo iterativo e incremental** es el que mejor se adapta a la velocidad de cambio en las necesidades del cliente que se da en la actualidad.

Por otro lado, la **Orientación a Objetos** (OO) sigue siendo la tecnología más extendida a la hora de implementar sistemas, y **UML** sigue siendo la notación por excelencia a la

hora de modelar las diferentes vistas de un sistema *software*. Enfoques como el *desarrollo de software dirigido por modelos* (DSDM), o la ingeniería dirigida por modelos (del inglés *Model Driven Engineering*, MDE), precisamente, desde hace ya varios años, han ayudado mucho a mantener una línea de investigación en esta área bastante enriquecedora y prometedora.

En la misma línea, los patrones de diseño aplicados al *software* han ido consolidándose a lo largo de los años desde que ya apareciera el famoso libro sobre **patrones de diseño** de los conocidos como «*gang-of-four*» (Gamma, Helm, Johnson y Vlissides, 1995). Resulta fundamental el definir una correcta colaboración entre los objetos que participan en el sistema de cara a poder implementar un producto de calidad y los patrones de diseño guían al desarrollador para conseguir este objetivo.

Los patrones de diseño fueron definidos como «la forma de describir un problema que se da una y otra vez en nuestro entorno, y que especifica el núcleo de la solución para ese problema, de tal forma que se pueda utilizar esa solución un millón de veces más, sin que se tenga hacer de la misma manera más de una vez» (Alexander et al., 1977), es decir, cada patrón expresa una relación entre un contexto específico, un problema y una solución.

Aunque esta definición fue dada en su inicio por Cristopher Alexander para su aplicación en la construcción de edificios y ciudades, los conocidos como «*gang-of-four*» supieron aplicarla con un éxito rotundo en el campo de la ingeniería de *software*, en concreto a la OO, haciendo de los patrones de diseño una herramienta muy útil para describir la comunicación que se da entre objetos y clases, que se personaliza a la hora de resolver un problema genérico de diseño para un dominio específico (Gamma, Helm, Johnson y Vlissides, 1995, 3).

Con lo cual, en el contexto de esta asignatura, los patrones de diseño van a estar constituidos por diagramas de objetos que dan forma abstracta a la solución a la que hacía referencia Cristopher Alexander, en el marco de la programación orientada a objetos y que son resultado de la experiencia de los programadores (Debrauwer, 2012, 17).

De manera resumida, los patrones de diseño estarán definidos a través de los siguientes elementos (Gamma, Helm, Johnson y Vlissides, 1995, 3).

- **Nombre.** Este elemento indica el nombre que se le da al patrón para describir el nombre del problema de diseño, sus soluciones y consecuencias, en una palabra o dos.
- **Problema.** Este elemento describe cuándo se debería aplicar el patrón. Este elemento explica el problema y su contexto.
- **Solución.** Este elemento describe a su vez los elementos que forman el diseño, sus relaciones, responsabilidades y colaboraciones.
- **Consecuencias.** Este elemento describe los resultados que se obtienen tras aplicar el patrón.

Como ya se ha comentado en temas anteriores, la comunicación entre clientes y contratistas constituye uno de los elementos más críticos dentro del proceso de desarrollo de *software*. En este sentido, los patrones de diseño pueden ayudar a mejorar esa comunicación de tal forma que, las necesidades del cliente podrán ser recogidas de manera más clara y concisa permitiendo un diseño de la solución más ajustado.

Es por ello, que en los siguientes apartados se describirán distintos patrones definidos por «*gang-of-four*» que se pueden utilizar para poder describir **las realizaciones de los casos de uso** que se han definido con notación UML durante la extracción de requisitos con el cliente, desde la perspectiva de uso en un ciclo de vida iterativo e incremental y con el enfoque del **proceso unificado** estudiado en el Tema 2.

La realización de los casos de uso debe indicar cómo se implementa un caso de uso concreto en el modelo de diseño a través de objetos que colaboran entre sí. Con la realización de los casos de uso, el diseñador describe el diseño de uno o más escenarios de un determinado caso. De esta manera, la realización del caso de uso va a establecer la conexión (**trazabilidad**) que se da entre los requisitos modelados como casos de uso y el modelo de diseño de objetos que satisface los requisitos (Larman, 2003, 232).

Siguiendo el enfoque del proceso unificado, en la iteración 2 de la fase de elaboración se pasa el foco de atención del análisis de requisitos y análisis del dominio al diseño de objetos con sus responsabilidades. En esa definición de objetos y responsabilidades es donde los patrones de diseño «*gang-of-four*» se pueden se pueden aplicar con el objetivo de satisfacer y refinar los requisitos modelados en la iteración 1.

6.3. *Adapter*

El patrón ***Adapter*** define lo siguiente (Larman, 2003, 322):

Problema:
¿Cómo resolver *interfaces* incompatibles, o proporcionar una *interfaz* estable para componentes parecidos con diferentes *interfaces*?

Solución:
Convertir la *interfaz* original de un componente en otra interfaz, mediante un objeto *Adapter* intermedio.

Si, por ejemplo, en el caso de uso a realizar se necesita que se soporten diferentes tipos de servicios externos de terceras partes (servicio de cálculo de impuestos, servicio de autorización de pagos, servicio de contabilidad, etc.), cada uno con su API independiente, entonces la solución podría estar en añadir un nivel de indirección con objetos que adapten las distintas interfaces externas de los distintos servicios a una interfaz consistente que será la que se utilice en la aplicación, tal y como se ilustra en la Figura 1.

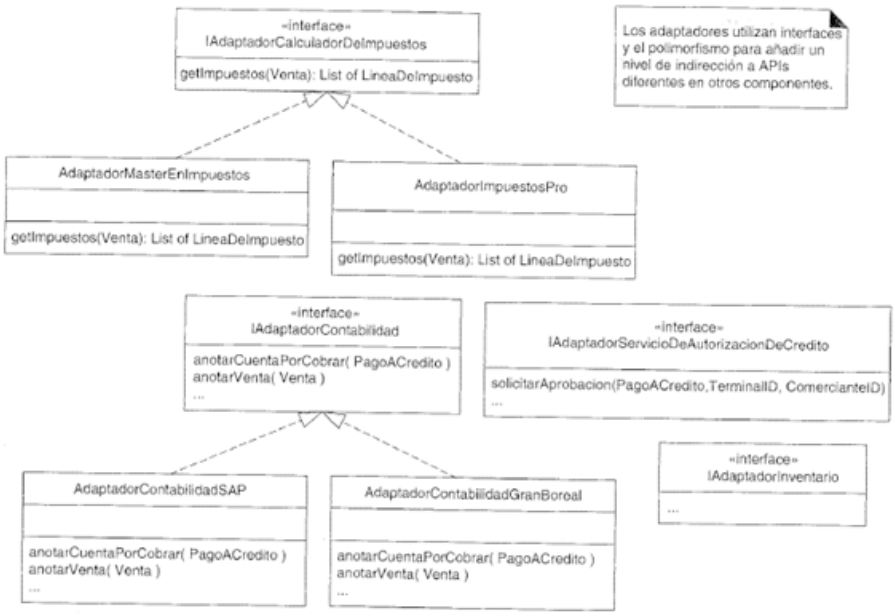


Figura 1. Ejemplo aplicación patrón *Adapter*. Fuente: Larman (2003, 323)

Es posible que al generar el nuevo diagrama de clases que permite realizar un caso de uso concreto generado durante la fase de análisis, el diseñador se dé cuenta de que necesita introducir nuevos conceptos del dominio como clases en el modelo de diseño.

Estas clases de dominio añadidas en diseño podrían ser útiles también en el modelo de análisis y por tanto, este debería actualizarse. Por ejemplo, la clase **LineaDeImpuesto** que aparece en la Figura 1 y que no formaba parte del análisis, podría incluirse en el diagrama de clases de análisis correspondiente tal y como se muestra en la Figura 2.

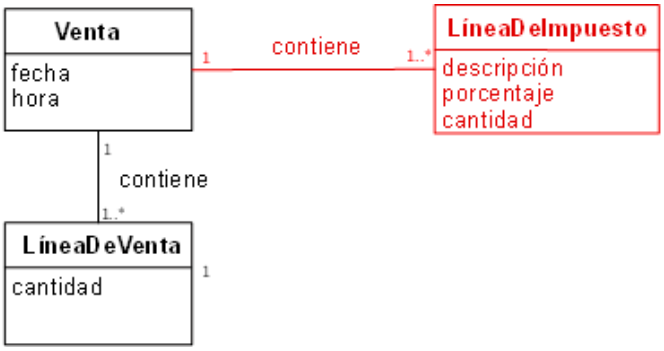


Figura 2. Diagrama de clases de análisis actualizado con la clase LíneaDeImpuesto, identificada en tiempo de diseño. Fuente: Larman (2003, 325)

6.4. *Factory*

El patrón ***Factory*** define lo siguiente (Larman, 2003, 327):

Problema:
¿Quién debe ser el responsable de la creación de los objetos cuando existen consideraciones especiales, como una lógica de creación compleja, el deseo de separar las responsabilidades de la creación para mejorar la cohesión, etc.?

Solución:
Crear un objeto de Fabricación Pura denominado *Factory* que se encargará de la creación.

Con el patrón *Adapter* visto en el apartado anterior, podría surgir el problema de que se pierda cohesión si se delega la creación de los adaptadores a un objeto del dominio (como por ejemplo, una clase **Registro**) que no distinga entre los distintos tipos de adaptadores que se pueden utilizar.

Para evitar este problema, se hará uso del patrón *Factory*, que será el encargado de separar las responsabilidades de los distintos adaptadores de una manera cohesiva y

ocultando la complejidad inherente que pueda tener la creación de ciertos objetos. La Figura 3 ilustra el patrón *Factory* siguiendo el ejemplo de la Figura 1.

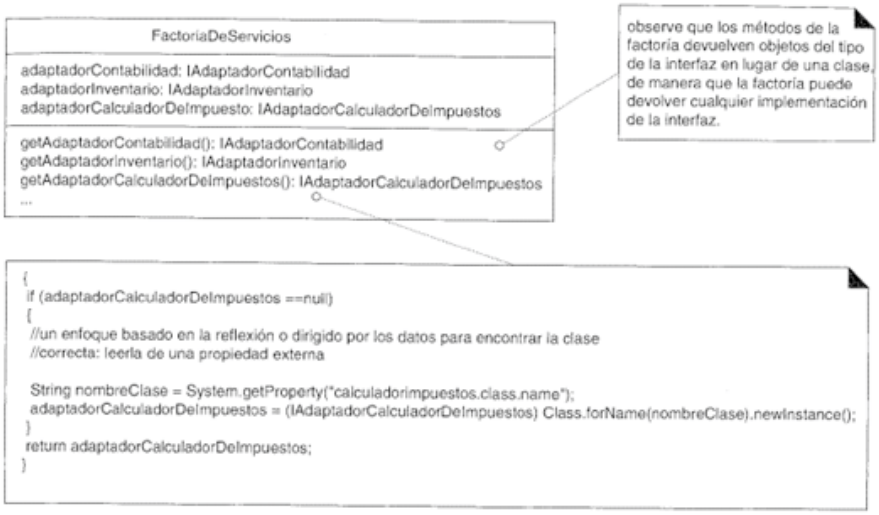


Figura 3. Ejemplo aplicación patrón *Adapter*. Fuente: Larman (2003, 327)

6.5. *Singleton*

El patrón *Singleton* define lo siguiente (Larman, 2003, 328):

Problema:
Se admite exactamente una instancia de una clase (es un *singleton*). Los objetos necesitan un único punto de acceso global.

Solución:
Definir un método estático de la clase que devolverá el *singleton*.

Con la aplicación del patrón *Factory* (clase **FactoríaDeServicios**) visto en el apartado anterior, vuelve a surgir otro problema y es que solo se necesita un único *Factory* en la aplicación. Para solucionarlo se puede aplicar el patrón *Singleton* de tal forma que, se define un método estático denominado ***getInstancia***, que será el método encargado de proporcionar la única instancia del *Factory*. De esta manera, el desarrollador tendrá la visibilidad global para poder acceder a esta única instancia allí donde lo necesite, tal y como se muestra en la Figura 4.

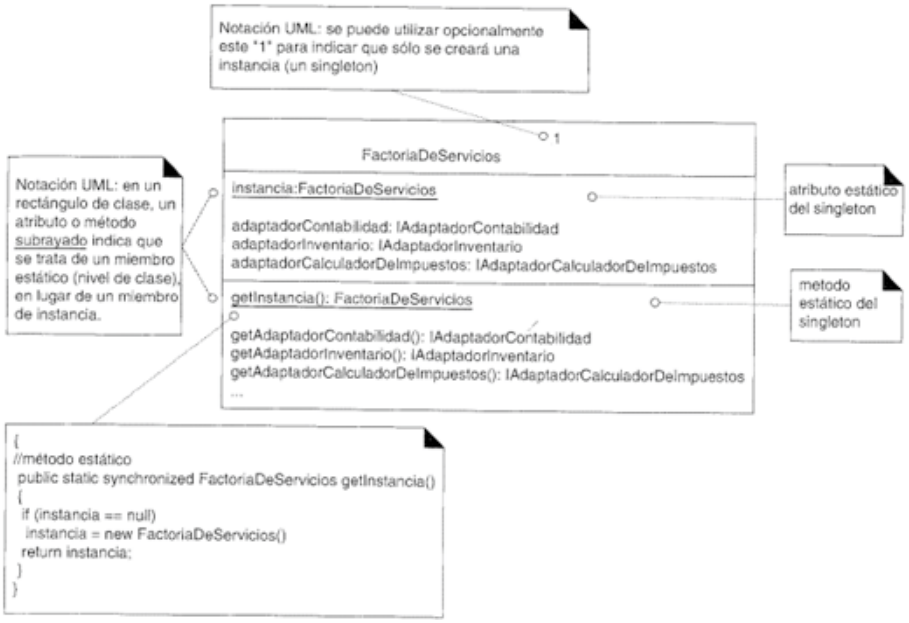


Figura 4. Ejemplo aplicación patrón *Singleton*. Fuente: Larman (2003, 329)

Además, resulta interesante destacar que cuando se utilice el método ***getInstancia*** en un diagrama de interacción UML, se podrá añadir el estereotipo <<*singleton*>> para indicar así que se trata de un mensaje implícito del patrón *Singleton* (ver Figura 5).

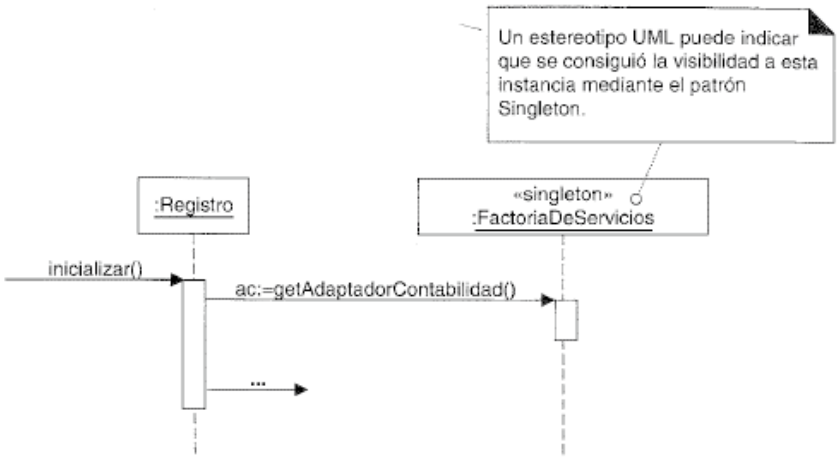


Figura 5. Mensaje implícito del patrón *Singleton* para obtener la visibilidad de una instancia (Fuente: Larman (2003, 331)

6.6. *Strategy*

El patrón *Strategy* define lo siguiente (Larman, 2003, 339):

Problema:
¿Cómo diseñar diversos algoritmos o políticas que están relacionadas? ¿Cómo diseñar que estos algoritmos o políticas puedan cambiar?

Solución:
Definir cada algoritmo o política en una clase independiente, con una interfaz común.

Con el patrón *Strategy* se podrá resolver el problema de diseño que surge cuando hay que proporcionar una lógica más compleja a la hora de, por ejemplo, fijar los precios en una tienda (descuentos para un día en todos los productos de la tienda, descuentos a estudiantes, descuentos a parados...).

Como el algoritmo será distinto en función de la estrategia que se siga, la solución consistirá en crear distintas clases de la **EstrategiaFijarPreciosVenta**, donde cada una tendrá el método polimórfico **getTotal**. Así cada clase aplicará el descuento como estime oportuno en cada caso. La Figura 6 ilustra esta situación.

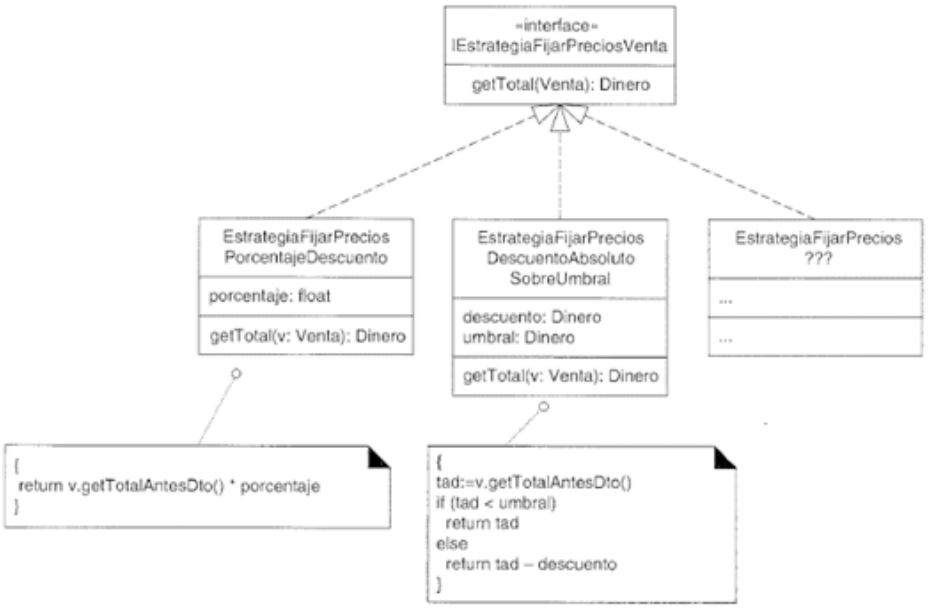


Figura 6. Ejemplo aplicación patrón *Strategy*. Fuente: Larman (2003, 333)

Para crear las distintas estrategias que fijan los precios de venta, se podrá volver a hacer uso del patrón *Factory* creando la clase **FactoríadeEstrategiasFijarPrecios**, que será la responsable de crear todas las estrategias que se puedan necesitar en la aplicación. Dado que la política de fijación de precios puede cambiar con frecuencia, no resulta conveniente almacenar la instancia de la estrategia creada en un campo de la **FactoríadeEstrategiasFijarPrecios**, sino crear una cada vez y leer la propiedad externa para poder obtener el nombre de la clase, e instanciar posteriormente la estrategia. Además, como la mayoría de aplicaciones del patrón *Factory*, la clase será un *singleton* a la que se accederá a través del patrón *Singleton*, tal y como se muestra en la Figura 7.

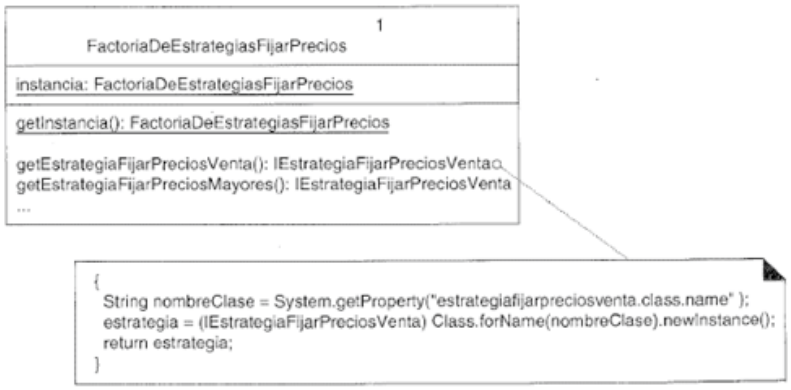


Figura 7. Ejemplo aplicación patrón Factory para la utilización del patrón *Strategy*. Fuente: Larman (2003, 335).

Además, cuando se crea una instancia de la clase **Venta**, se podría pedir a la instancia del *Factory* la estrategia que se desea utilizar tal y como se ilustra en la Figura 8.

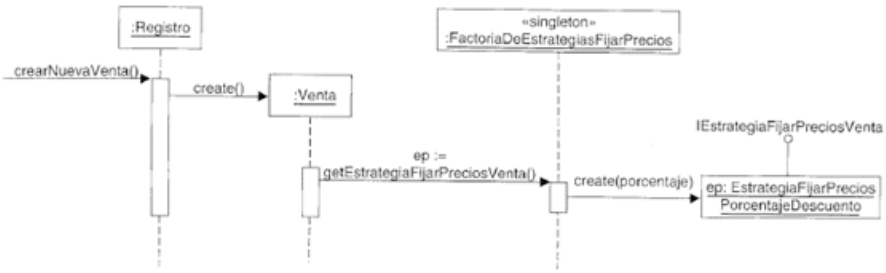


Figura 8. Creación de una estrategia concreta. Fuente: Larman (2003, 336).

6.7. *Composite*

El patrón ***Composite*** define lo siguiente (Larman, 2003, 339):

Problema:
¿Cómo tratar un grupo o una estructura compuesta del mismo modo (polimórficamente) que un objeto no compuesto (atómico)?

Solución:
Definir las clases para los objetos compuestos y atómicos de manera que implementarán la misma interfaz.

Con el ejemplo que se está siguiendo para mostrar cómo realizar los casos de uso aplicando patrones de diseño, se puede encontrar otro problema entre los requisitos y el diseño a la hora de plantearse la gestión de distintas políticas de fijación de precios que son contradictorias entre sí.

Este caso se daría si se tuviera una política de precios para parados y a la vez se diera otra política de precios si se hace una compra de un importe superior a una cantidad específica. En este caso, si una persona cumple los dos requisitos (es decir, está parado y su compra supera el importe establecido para acogerse a otra política de precios), ¿qué política de precios habría que aplicar?

Con el patrón *Composite* se puede conseguir que un objeto concreto (la clase **Venta**, según el ejemplo que se está siguiendo) desconozca si se está aplicando una o más estrategias. La Figura 9 ilustra esta situación.

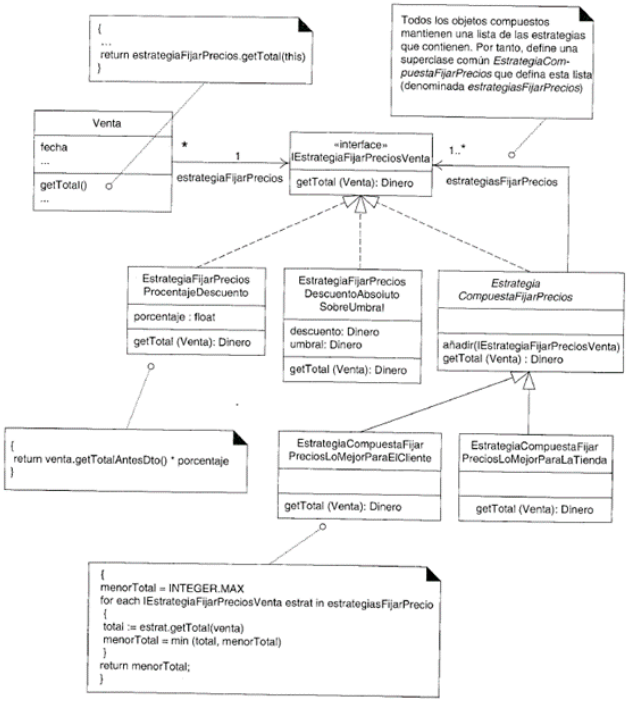


Figura 9. Ejemplo aplicación patrón *Composite*. Fuente: Larman (2003, 338).

De esta manera, tal y como se muestra en la Figura 9, la clase **EstrategiaCompuestaFijarPreciosLoMejorParaElCliente** implementaría la interfaz **IEstrategiaFijarPreciosVenta**, y la misma clase contendrá a su vez otros objetos del tipo **IEstrategiaFijarPreciosVenta**.

En este caso, la creación de la estrategia compuesta se realizaría tal y como se muestra en la Figura 10.

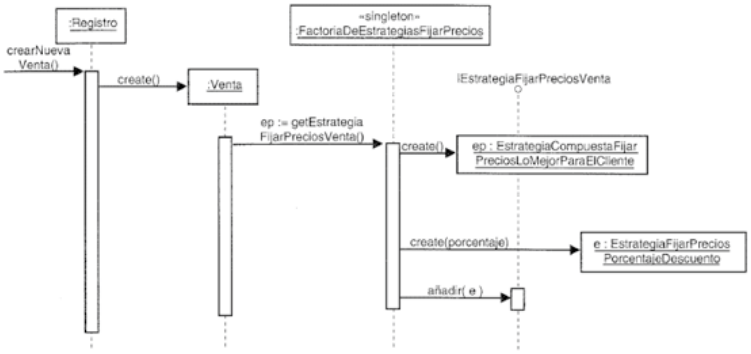


Figura 10. Creación de una estrategia compuesta. Fuente: Larman (2003, 342)

6.8. *Facade*

El patrón *Facade* define lo siguiente (Larman, 2003, 346):

Problema:
¿Qué se debería hacer cuando se requiere una interfaz común, unificada para un conjunto de implementaciones o *interfaces* dispares, como en un subsistema, en la que podría no ser conveniente acoplarla con muchas cosas del subsistema, o en la que podría cambiar la implementación del subsistema?

Solución:
Definir un único punto de conexión con el subsistema, lo que sería un objeto *Facade* que envuelve al subsistema. Este objeto *Facade* presentará una interfaz única y será el responsable de colaborar con los componentes del subsistema.

Otro requisito que podría estar definido en esta iteración dentro del proceso unificado, sería el soporte a reglas de negocio conectables. Es decir, siguiendo con el ejemplo anterior, se daría en ciertos puntos predecibles de los escenarios, como cuando se dan las acciones (operaciones**) crearNuevaVenta** e **introducirArtículo** dentro del caso de uso **Procesar Venta**. En este caso, el patrón *Facade* ayuda a resolver este problema, ya que un objeto *Facade* constituirá un único punto de entrada para los servicios de un subsistema, donde la implementación y otros componentes del sistema permanecerán privados y ocultos para otros componentes externos.

De acuerdo al ejemplo anterior, se podría definir un subsistema denominado «motor de reglas» que será el responsable de evaluar, de manera transparente y oculta, un conjunto de reglas cuando se dé una operación, y así podrá indicar entonces si alguna de las reglas que se han de cumplir invalida la operación. Normalmente el acceso al objeto *Facade* se realizará a través de un *singleton*. Para este ejemplo, el objeto *Facade* del subsistema «motor de reglas» se denominará **FachadaMotorReglasPDV** y el código que se podría introducir para dar soporte a las reglas a la hora de crear una línea de venta, sería el que se muestra a continuación:


```
public class Venta {

	public void crearLineaDeVenta (EspecificacionesDelProducto espec, int cantidad) {

		LineaDeVenta ldv = new LineaDeVenta(espec, cantidad);

		//En este punto se realiza la llamada a Facade
		if (FachadaMotorReglasPDV.getInstancia().esInvalido(ldv, this))
			return;

		lineasDeVenta.add(ldv);
	}

	//...
}//end class Venta
```

6.9. *Observer*

El patrón *Observer* define lo siguiente (Larman, 2003, 350):

Problema:
¿Qué se debería hacer cuando diferentes tipos de objetos están interesados en eventos de un objeto concreto, y quieren reaccionar cada uno a su manera ante la ocurrencia de dichos eventos manteniendo un bajo acoplamiento entre los objetos involucrados?

Solución:
Definir una interfaz «suscriptor» u «oyente» y los objetos interesados (es decir, los suscriptores) implementarán dicha interfaz. El objeto observado, es decir, el emisor, podrá registrar suscriptores que estén interesados en un evento, de tal forma que se les notificará cuando ocurra.

Con el patrón *Observer* se podrá gestionar de manera efectiva el cambio de datos para que queden reflejados en la ventana GUI manteniendo el mínimo acoplamiento posible. Siguiendo el ejemplo anterior, se deberá actualizar la información de la GUI cuando un objeto de la clase **Venta** cambia el coste total de la misma. Los objetos que no pertenecen a la parte de la interfaz de usuario no conocerán a los objetos que sí forman parte de ella, los cuales son los que deben enterarse de los cambios de información en algunos de esos objetos que no forman parte de la GUI para poder actualizar la ventana con los últimos datos. Así se consigue que el modelo de sistema y la vista del sistema queden separadas y protegidas ante cambios en la interfaz de usuario. La Figura 11 ilustra esta situación.

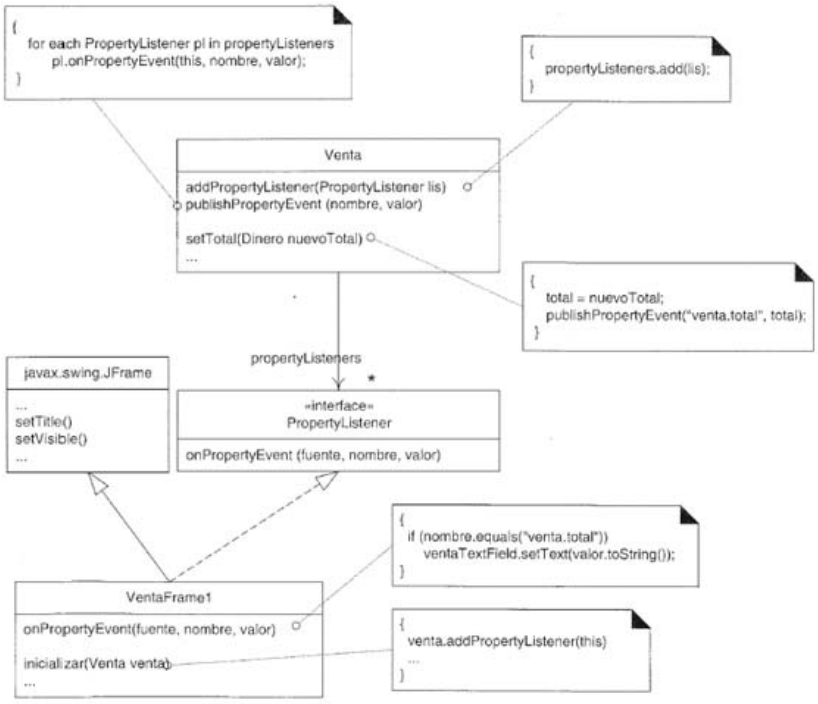


Figura 11. Ejemplo aplicación patrón *Observer*. Fuente: Larman (2003, 350)

La interpretación de acuerdo al diagrama de clases mostrado en la Figura 11 sería la que se indica a continuación:

- Se define la interfaz **PropertyListener** con la operación **OnPropertyEvent**.
- Se define la ventana **VentaFrame1** que implementa la interfaz.
- Cuando se inicializa **VentaFrame1** se le pasa la instancia de **Venta** de la que está mostrando el total en la ventana.
- **VentaFrame1** se suscribe a la instancia de **Venta** que se le pasó como parámetro en su creación para que se le notifique acerca de los cambios en los eventos de una determinada propiedad a través del mensaje **addPropertyListener** (por ejemplo, la propiedad a la que se suscribe podría ser la correspondiente al importe total de la Venta, para que se le notifique cada vez que cambie).
- La instancia de **Venta** al que está suscrito VentaFrame1 no conoce los objetos de VentaFrame1, solamente los objetos que implementan la interfaz **PropertyListener**, lo que hará que disminuya el acoplamiento entre las clases Venta y VentaFrame1 (la clase VentaFrame1 podría cambiar o se podrían añadir nuevas clases para la interfaz de usuario y no le afectaría para nada a la clase Venta).
- La instancia de **Venta** es el objeto emisor que, cuando cambia el importe total de la venta, itera sobre todos los objetos **PropertyListener** que están suscritos para notificarles del cambio.

6.10. Referencias bibliográficas

Alexander, C., Ishikawa, S., Silverstein, M, Jacobson, M, Fiksdahl-King, I. y Angel, S. (1977). *A Pattern Language*. Oxford University Press, New York.

Debrauwer, L. (2012). *Patrones de diseño para C#*. Ediciones ENI.

Gamma, E., Helm, R., Johnson, R. y Vlissides, J. (1995). *Design Patterns. Elements of Reusable Object-Oriented Software*. Addison-Wesley.

Larman, C. (2003). *UML y Patrones. Una introducción al análisis y diseño orientado a objetos y al proceso unificado. Segunda edición*. Pearson Prentice-Hall.

Lo + recomendado

No dejes de leer...

Patrones de comunicación en sistemas tutores inteligentes

Ruddeck, G., Maciuszek, D., Weicht, M. & Martens, A. (2011). Patrones de comunicación en sistemas tutores inteligentes basados en componentes. *Novática: Revista de la Asociación de Técnicos de Informática*, 37(210), 52-56.

En este artículo se presenta la investigación realizada en la utilización de patrones adicionales para el desarrollo de sistemas para *e-learning*, en concreto, para el desarrollo de sistemas de tutores inteligentes (del inglés *Intelligent Tutoring Systems*, ITS).

Una introducción al análisis y diseño orientado a objetos

Larman, C. (2003). *UML y Patrones. Una introducción al análisis y diseño orientado a objetos y al proceso unificado. Segunda edición*. Pearson Prentice-Hall.

Libro que proporciona una introducción práctica al análisis y diseño orientado a objetos (OOA/D), a la aplicación de patrones de diseño y a aspectos relacionados de desarrollo iterativo. Para profundizar en lo estudiado en el tema se recomienda la lectura de capítulo 23: *Diseño de las realizaciones de casos de uso con los patrones de diseño GoF* (páginas 321-356) de este libro.

Disponible en el aula virtual bajo licencia CEDRO

+ Información

A fondo

Design patterns: magic or myth?

Budgen, D. (2013). *Design Patterns: Magic or Myth?* IEEE Software, 30(2), 87-90.

En este artículo el autor hace una revisión del uso de los patrones de diseño para analizar las consecuencias de su aplicación en los sistemas *software*.

Disponible en el aula virtual bajo licencia CEDRO

Guía de construcción de *software* en java con patrones de diseño

Martínez, F.J. (2002). Guía de construcción de *software* en java con patrones de diseño. *Proyecto Fin de Carrera, Escuela Universitaria de Ingeniería Técnica en Informática de Oviedo, Universidad de Oviedo.*

En este proyecto fin de carrera el autor muestra, de manera didáctica, la forma de aplicar patrones de diseño a la hora de desarrollar una aplicación en Java.

Enlaces relacionados

UML

El enlace *Unified Modeling Language* (UML) *Resource Page* forma parte del OMG y contiene toda la información asociada con el lenguaje de modelado UML.

Accede a la página desde el aula virtual o a través de la siguiente dirección web:
http://www.uml.org/

Bibliografía

Gamma, E., Helm, R., Johnson, R. y Vlissides, J. (1995). *Design Patterns. Elements of Reusable Object-Oriented Software*. Addison-Wesley.

Larman, C. (2003). *UML y Patrones. Una introducción al análisis y diseño orientado a objetos y al proceso unificado. Segunda edición*. Pearson Prentice-Hall.

OMG (2011). OMG Unified Modeling Language (OMG UML), Superstructure. Version 2.4.1. *Document formal/2011-08-06*. Recuperado de http://www.omg.org/spec/UML/2.4.1/Superstructure/PDF/

Recursos externos

Visual paradigm

Es una herramienta de modelado con UML con carácter gratuito en su versión Community.

Accede a la página desde el aula virtual o a través de la siguiente dirección web:

http://www.visual-paradigm.com/download/community.jsp

Tutoriales (*online*):

http://www.visual-paradigm.com/tutorials/?category=usecasemodeling

http://www.visual-paradigm.com/tutorials/?category=umlmodeling

Manual de usuario (*online*):

http://www.visual-paradigm.com/support/documents/vpuserguide.jsp

Manual de usuario (pdf):

http://images.visual-paradigm.com/docs/vp_user_guide/VP_Users_Guide.pdf

Actividades

Caso práctico: Diagrama de casos de uso y diagrama de clases

Para esta actividad, a partir del enunciado del primer caso práctico (gestionar de manera automatizada la tramitación y realización de los exámenes de Kárate para cinturón negro, en cualquier de sus grados) y seleccionando una de las herramientas de modelado con la que se hizo la comparativa, se pide:

- Diagramas de casos de uso.
- Descripción textual breve de los casos de uso.
- Diagrama de clases.

Para el apartado de descripción textual breve de los casos de uso, se recomienda usar la siguiente tabla para cada caso de uso:

Identificador	Código identificativo del caso de uso	
Nombre	Nombre del requisito funcional	
Descripción	Descripción de la funcionalidad del caso de uso	
Precondición	Precondición del caso de uso	
Postcondición	Postcondición del caso de uso	
Actores		
Secuencia Normal	**Paso**	**Acción**
	1	
	2	
	...	
Secuencia alternativa	**Paso**	**Acción**
	1	
	2	
	...	
Secuencia de error	**Paso**	**Acción**
	1	
	2	
	...	
Importancia	Sin importancia, importante, vital	
Urgencia	Puede esperar, hay presión, inmediatamente	
Observaciones	Comentarios adicionales	

Entregar la solución en un documento Word (Georgia 11 e interlineado 1,5) incluyendo las imágenes de los diagramas realizados con la herramienta utilizada. Enviar también los ficheros originales de los modelos de la herramienta.

Por si no sabes qué herramientas escoger, en el apartado Recursos externos del tema encontrarás la referencia a una herramienta de modelado con UML gratuita bastante utilizada.

Competencias

CB6. Poseer y comprender conocimientos que aporten una base u oportunidad de ser originales en el desarrollo y/o aplicación de ideas, a menudo en un contexto de investigación.
CB7. Que los estudiantes sepan aplicar los conocimientos adquiridos y su capacidad de resolución de problemas en entornos nuevos o poco conocidos dentro de contextos más amplios (o multidisciplinares) relacionados con su área de estudio.
CB8. Que los estudiantes sean capaces de integrar conocimientos y enfrentarse a la complejidad de formular juicios a partir de una información que, siendo incompleta o limitada, incluya reflexiones sobre las responsabilidades sociales y éticas vinculadas a la aplicación de sus conocimientos y juicios.
CB9. Que los estudiantes sepan comunicar sus conclusiones y los conocimientos y razones últimas que las sustentan a públicos especializados y no especializados de un modo claro y sin ambigüedades.
CB10. Que los estudiantes posean las habilidades de aprendizaje que les permitan continuar estudiando de un modo que habrá de ser en gran medida autodirigido o autónomo.
CG1. Capacidad para proyectar, calcular y diseñar productos, procesos e instalaciones en el ámbito de la Ingeniería de Software.
CE1. Capacidad para modelar, diseñar, definir la arquitectura, implantar, gestionar, operar, administrar y mantener aplicaciones, sistemas, servicios y contenidos informáticos.
CE2. Capacidad para utilizar y desarrollar metodologías, métodos, técnicas, programas de uso específico, normas y estándares de Ingeniería de Software.
CE3. Capacidad para analizar las necesidades de información que se plantean en un entorno y llevar a cabo en todas sus etapas el proceso de construcción de un sistema de información.

CE4. Capacidad para crear y diseñar sistemas software que resuelvan problemas del mundo real.

CE5. Capacidad para evaluar y utilizar entornos de Ingeniería de Software avanzados, métodos de diseño, plataformas de desarrollo y lenguajes de programación.

CT1. Analizar de forma reflexiva y crítica las cuestiones más relevantes de la sociedad actual para una toma de decisiones coherente.

CT2. Identificar las nuevas tecnologías como herramientas didácticas para el intercambio comunicacional en el desarrollo de procesos de indagación y de aprendizaje grupal.

CT3. Aplicar los conocimientos y capacidades aportados por los estudios a casos reales y en un entorno de grupos de trabajo en empresas u organizaciones.

CT4. Adquirir la capacidad de trabajo independiente, impulsando la organización y favoreciendo el aprendizaje autónomo.

Test

1. ¿Qué es un patrón de diseño?

A. Un patrón de diseño expresa una relación entre un determinado contexto, un problema y una solución.

B. Un patrón de diseño expresa una relación entre un determinado contexto, un problema pero no da una solución concreta.

C. Un patrón de diseño es algo abstracto que ofrece soluciones de diseño para el desarrollo de sistemas orientados a objetos.

D. Un patrón de diseño da solución a un problema pero no establece su contexto.

2. ¿Qué es la realización de un caso de uso?

A. La ejecución de un caso de uso concreto en la aplicación.

B. La trazabilidad que va de la definición del caso de uso a cada uno de sus escenarios.

C. La descripción en análisis de uno o más escenarios del caso de uso.

D. La implementación del caso de uso en el modelo de diseño a través de objetos que colaboran entre sí.

3. La fase de elaboración del proceso unificado:

A. Se centra en el análisis del sistema.

B. Se centra en el análisis y diseño del sistema.

C. Se centra en el diseño de objetos.

D. No admite la aplicación de patrones de diseño.

4. ¿Cuáles son los elementos más comunes de un patrón de diseño?

A. Nombre y solución.

B. Nombre, problema, solución y consecuencias.

C. Nombre, problema y solución.

D. Problema y solución.

5. ¿Qué patrón se debería utilizar cuando se necesite resolver interfaces incompatibles, o proporcionar una interfaz estable para componentes parecidos con diferentes interfaces?

A. *Adapter*.

B. *Strategy*.

C. *Singleton*.

D. *Observer*.

6. ¿Qué patrón se debería utilizar cuando un objeto necesite enterarse de cambios en otro objeto?

A. *Adapter*.

B. *Observer*.

C. *Strategy*.

D. *Composite*.

7. ¿Qué patrón se debería utilizar cuando se necesite dar soporte a reglas de negocio para poder validar una operación?

A. *Adapter*.

B. *Facade*.

C. *Singleton*.

D. *Factory*.

8. ¿Qué patrón se debería utilizar cuando se quiera utilizar una estructura compuesta de la misma manera que si fuera una estructura simple?

A. *Factory*.

B. *Singleton*.

C. *Facade*.

D. *Composite*.

9. ¿Qué patrón se debería utilizar cuando se requiera un punto de conexión único con un subsistema concreto?

A. *Adapter*.

B. *Facade*.

C. *Singleton*.

D. *Composite*.

10. ¿Qué patrón se debería utilizar cuando se deseen diseñar distintos algoritmos que están relacionados?

A. *Composite.*

B. *Factory*.

C. *Facade.*

D. *Strategy*.