Catálogo de patrones

Jordi Pradel i Miquel
José Antonio Raya Martos

PID_00165658

Universitat Oberta de Catalunya
www.uoc.edu

Ninguna parte de esta publicación, incluido el diseño general y la cubierta, puede ser copiada, reproducida, almacenada o transmitida de ninguna forma, ni por ningún medio, sea éste eléctrico, químico, mecánico, óptico, grabación, fotocopia, o cualquier otro, sin la previa autorización escrita de los titulares del copyright.

Índice

Introducción..... 5

Objetivos..... 6

1. Tabla de referencia..... 7

2. Principios de diseño..... 9
2.1. Bajo acoplamiento (*Low coupling*) 9
2.2. Alta cohesión (*High cohesion*) 9
2.3. Abierto-cerrado (OCP, *Open-closed principle*) 10
2.3.1. Ley de Demeter (*Law of Demeter*) 10
2.4. No-repetición (DRY, *Don't repeat yourself*) 11
2.5. Sustitución de Liskov (LSP, *Liskov substitution principle*) 11
2.6. Segregación de interfaces (ISP, *Interface-segregation principle*) 11
2.7. Inversión de dependencias (DIP, *Dependency inversion principle*) 12

3. Patrones de análisis..... 13
3.1. Asociación histórica 13
3.2. Objeto compuesto (*Composite*) 17
3.3. Cantidad (*Quantity*) 18
3.4. Rango (*Range*) 21

4. Patrones de arquitectura..... 24
4.1. Arquitectura en capas (*Layers*) 24
4.1.1. Variantes 27
4.2. Inyección de dependencias (*Dependency injection*) 27
4.3. Modelo, vista y controlador (MVC, *Model-view-controller*) 29

5. Patrones de asignación de responsabilidades..... 33
5.1. Controlador (*Controller*) 33
5.2. Experto (*Expert*) 36
5.3. Fabricación pura (*Pure fabrication*) 37
5.4. Creador (*Creator*) 39

6. Patrones de diseño..... 41
6.1. Estado (*State*) 41
6.2. Fachada (*Facade*) 44
6.3. Iterador (*Iterator*) 47
6.4. Método factoría (*Factory method*) 50
6.5. Método plantilla (*Template method*) 52

6.6. Observador (*Observer*) 54
6.7. Orden (*Command*) 58
6.8. Otros patrones de diseño 60
6.8.1. Adaptador (*Adapter*) 60
6.8.2. Decorador (*Decorator*) 60
6.8.3. Estrategia (*Strategy*) 61
6.8.4. Instancia única (*Singleton*) 62
6.8.5. Objeto nulo (*Null object*) 63
6.8.6. Representante (*Proxy*) 64
6.8.7. Servidor abstracto (*Abstract server*) 66

Ejercicios de autoevaluación..... 69

Solucionario..... 70

Bibliografía..... 72

Introducción

Para la aplicación de los patrones a las diferentes etapas del ciclo de vida del desarrollo, habrá que disponer de un catálogo que nos permita buscar y hacer referencia de manera rápida a los diferentes patrones.

Este módulo contiene un catálogo organizado según las etapas del ciclo de vida en las que se pueden aplicar los patrones (análisis, arquitectura, asignación de responsabilidades y diseño). De cada etapa se han escogido aquellos patrones que se han considerado más representativos de su categoría y cuyo uso es más generalizado.

Según la estructura vista en el módulo anterior, "Introducción a los patrones", de cada patrón se explica la problemática que ataca, las soluciones propuestas y también las consecuencias (beneficios e inconvenientes) de su aplicación. También se indica el catálogo donde se publicó originalmente con el fin de dar la posibilidad de ampliar la información disponible tanto sobre el propio patrón, como sobre otros patrones del mismo catálogo que puedan estar relacionados.

Este módulo también se puede utilizar como referencia durante la realización de las actividades de la asignatura.

Objetivos

El objetivo principal de este módulo es mostrar un conjunto de patrones y facilitar su comprensión mediante ejemplos. También sirve como referencia donde encontrar los patrones que nos ayuden a mejorar el análisis y diseño de los sistemas de información a desarrollar. Los objetivos concretos que tiene que alcanzar el estudiante con este módulo didáctico son los siguientes:

1. Conocer una selección de patrones aplicables a las diferentes etapas del ciclo de vida del desarrollo.

2. Conocer los diferentes patrones, así como sus consecuencias.

1. Tabla de referencia

A continuación se incluye una tabla de referencia con un resumen de todos los patrones del catálogo. De cada patrón se indica su nombre, el tipo, el apartado del módulo donde se explica en detalle y un resumen; también indicamos algunas referencias bibliográficas.

Nombre	Tipo	Ref.	Ap.	Resumen
Adaptador	Diseño	GOF, LAR, MAR	6.8.1	Permite utilizar una clase usando un conjunto de operaciones diferentes al que ofrecía originalmente.
Arquitectura en capas	Arquitectura	POSA, LAR	4.1	Organiza un sistema haciendo que cada elemento trabaje a un único nivel de abstracción.
Asociación histórica	Análisis	FOW	3.1	Representa los valores de una asociación a lo largo del tiempo.
Controlador	Asig. resp.	LAR	5.1	Asigna la responsabilidad de tratar acontecimientos a un objeto controlador.
Creador	Asig. resp.	LAR	5.4	Asigna la responsabilidad de crear instancias a un objeto.
Decorador	Diseño	GOF, MAR	6.8.2	Añade responsabilidades a un objeto sin modificar su clase.
Estado	Diseño	GOF, LAR, MAR	6.1	Permite a un objeto variar su comportamiento dependiendo de su estado.
Estrategia	Diseño	GOF, LAR, MAR	6.8.3	Permite escoger entre una familia de algoritmos.
Experto	Asig. resp.	LAR	5.2	Asigna una responsabilidad al objeto que tiene la información para llevarla a cabo.
Fabricación pura	Asig. resp.	LAR	5.3	Crea una nueva clase para asignarle una responsabilidad independiente del dominio.
Fachada	Diseño	GOF, LAR, MAR	6.2	Reduce el acoplamiento entre subsistemas.
Inyección de dependencias	Arquitectura	FOWDI	4.2	Permite escoger y cambiar la implementación de los servicios sin tener que cambiar el código de los clientes.
Instancia única	Diseño	GOF, LAR, MAR	6.8.4	Asegura, para una clase, que sólo puede haber una instancia.
Iterador	Diseño	GOF	6.3	Permite recorrer los elementos de una colección.
Método factoría	Diseño	GOF, LAR	6.4	Crea instancias de una clase sin conocer la subclase exacta.
Método plantilla	Diseño	GOF, LAR, MAR	6.5	Implementa varios algoritmos parecidos reutilizando la parte común.
MVC	Arquitectura	POSA	4.3	Organiza las responsabilidades de interacción con el usuario.
Objeto compuesto	Análisis	GOF, LAR, MAR	3.2	Permite tratar una colección de objetos y sus elementos de la misma manera.
Objeto nulo	Diseño	MAR	6.8.5	Trata el valor nulo como un objeto.
Observador	Diseño	GOF, LAR, MAR	6.6	Notifica los cambios en el estado de un objeto a otros objetos.

Nombre	Tipo	Ref.	Ap.	Resumen
Orden	Diseño	GOF, LAR, MAR	6.7	Trata una invocación a una operación como un objeto.
Cantidad	Análisis	FOW	3.3	Representa una medida con su unidad.
Rango	Análisis	FOW	3.3	Representa un rango de valores.
Representante	Diseño	GOF, LAR, MAR	6.8.6	Controla el acceso a un objeto.
Servidor abstracto	Diseño	MAR	6.8.7	Reduce el acoplamiento entre clientes y servidores.

2. Principios de diseño

Para poder evaluar la calidad de un diseño, es necesario establecer ciertos principios que éste tiene que cumplir. De esta manera, diremos que un diseño es de calidad en la medida en que se ajuste a estos principios de diseño.

La finalidad de la aplicación de los patrones de diseño es mejorar la calidad del sistema desarrollado aprovechando la experiencia previa. Por lo tanto, a la hora de decidir sobre la aplicación de los patrones, lo tendremos que hacer a la luz de los principios mencionados antes. Por ello, en este módulo, se incluye un pequeño catálogo de principios de diseño.

A continuación se da una serie de principios de diseño ampliamente aceptados que, en algunos casos, nos pueden ayudar en otras etapas del ciclo de vida del desarrollo, como puede ser el análisis.

2.1. Bajo acoplamiento (*Low coupling*)

Tenemos que minimizar el acoplamiento.

El acoplamiento mide la dependencia entre elementos (como clases, paquetes, etc.), típicamente como resultado de la colaboración entre los elementos para proporcionar un servicio.

Cuando una clase tiene un acoplamiento alto con respecto a otras clases, nos encontramos con lo siguiente:

- Un cambio en las clases relacionadas puede provocar un cambio en nuestra clase.
- Es difícil entender nuestra clase de manera aislada.
- Es difícil reutilizar nuestra clase, ya que su uso requiere la presencia de las clases relacionadas.

2.2. Alta cohesión (*High cohesion*)

Tenemos que maximizar la cohesión.

La cohesión mide el grado de relación entre las diferentes responsabilidades de una clase. Cuanto más relacionadas estén entre ellas estas responsabilidades, más alta será la cohesión de una clase.

Una clase con cohesión baja tienen los problemas siguientes:

- Es difícil de entender.
- Es difícil de reutilizar.
- Es difícil de mantener.
- Es frágil con respecto a los cambios que se produzcan en el sistema.

Una versión más restrictiva de este principio es el conocido como principio de la responsabilidad única (SRP, *single-responsibility principle*): "Una clase sólo debería tener un motivo para cambiar".

Bibliografía complementaria

Podéis encontrar más información sobre el principio SRP en [MAR].

2.3. Abierto-cerrado (OCP, *Open-closed principle*)

Una entidad software (clase, módulo, función, etc.) tendría que estar abierta a la extensión pero cerrada a la modificación.

Bibliografía complementaria

Podéis encontrar más información sobre el principio abierto-cerrado en [MAR] y en [MEY]. También en [LAR] bajo el nombre de *Variaciones protegidas*.

En términos de diseño, este principio nos indica que, para añadir nuevas responsabilidades a nuestro sistema, lo tenemos que poder hacer añadiendo nuevas clases (extensión), pero sin modificar las que había.

Una manera de cumplir este principio es crear abstracciones en torno a los aspectos que prevemos que tienen que cambiar de manera que se pueda crear una interfaz estable con respecto a los cambios. Mediante esta solución, nuestro sistema estará abierto a la extensión (añadiendo nuevas implementaciones de las interfaces definidas) pero cerrado a la modificación (no tenemos que modificar las interfaces definidas).

Principio de Variaciones protegidas

Nuestro diseño debe minimizar el impacto en el sistema actual de los cambios que se produzcan en el futuro y que ya podemos anticipar.

2.3.1. Ley de Demeter (*Law of Demeter*)

La ley de Demeter, a veces resumida como "No hables con desconocidos[1]", es un heurístico que nos ayuda a cumplir el principio Abierto-cerrado:

(1)En inglés: *"Don't talk to strangers"*.

Una operación de un objeto sólo tendría que utilizar:

- Las operaciones del mismo objeto.
- Los objetos que tenga asociados el mismo objeto (o que sean atributos suyos).
- Los objetos que recibe como parámetro la operación.
- Los objetos que cree la operación.

Esta ley nos ayuda a reducir el acoplamiento con respecto a una estructura de clases concreta. Además, potencia la encapsulación, ya que, para acceder a los objetos asociados a un objeto O, lo tendremos que hacer por medio de las operaciones del objeto O, de manera que nos aseguremos de que éste se enterará del acceso o manipulación.

Ley de Demeter

A veces se usa el nombre de *Ley de Demeter* como principio de diseño en sustitución del principio Abierto-cerrado.

2.4. No-repetición (DRY, *Don't repeat yourself*)

Cada pieza de conocimiento debe tener una única e inambigua representación en el sistema.

Bibliografía complementaria

Podéis encontrar más información sobre el principio de No-repetición en [HUNT].

Este principio nos indica que tenemos que evitar, siempre que podamos, la duplicación de información y de responsabilidades. Si lo conseguimos, nuestro sistema será más sencillo y más fácil de mantener, ya que, ante un cambio o error, podremos identificar fácilmente cuál es el componente afectado.

2.5. Sustitución de Liskov (LSP, *Liskov substitution principle*)

Las instancias de una subclase C tienen que ser sustituibles por instancias de las superclases de C.

Bibliografía complementaria

Podéis encontrar más información sobre el principio de Sustitución de Liskov en [MAR].

Este principio nos indica que una buena jerarquía de herencia tiene que respetar el comportamiento de las superclases. Una clase o programa que utilizara instancias de una superclase S tendría que poder utilizar cualquier instancia de una subclase de S sin que su comportamiento se vea afectado negativamente.

2.6. Segregación de interfaces (ISP, *Interface-segregation principle*)

Los clientes no tendrían que depender de operaciones que no utilizan.

Bibliografía complementaria

Podéis encontrar más información sobre el principio de segregación de interfaces en [MAR].

Una clase puede ofrecer muchas operaciones. En estos casos, es normal que nos encontremos con que algunas de las clases cliente sólo utilizan un subconjunto de las operaciones. Este principio nos dice que, en estos casos, tenemos que separar la interfaz de la clase en subconjuntos de operaciones con el fin de evitar el acoplamiento de las clases cliente hacia operaciones que no utilizan.

2.7. Inversión de dependencias (DIP, *Dependency inversion principle*)

Los módulos o clases de alto nivel no tendrían que depender de los de bajo nivel, sino de una abstracción.

Las abstracciones no tendrían que depender de los detalles. Los detalles tendrían que depender de las abstracciones.

Bibliografía complementaria

Podéis encontrar más información sobre el principio de inversión de dependencias en [MAR].

Para poder maximizar la reutilización de nuestras clases, tenemos que evitar el acoplamiento con respecto a las clases de más bajo nivel. Eso nos permitirá limitar el impacto de un cambio al nivel de abstracción en el que se produzca. Por ejemplo, si cambiamos de servidor de base de datos, la capa de dominio no se debería ver afectada.

3. Patrones de análisis

Los patrones de análisis son aquellos patrones que documentan soluciones aplicables durante la realización del diagrama estático de análisis para resolver los problemas que surgen: nos proporcionan maneras probadas de representar conceptos generales del mundo real en un diagrama estático del análisis.

Ved también

Podéis encontrar más información sobre los patrones de análisis en el apartado 2.1 del módulo "Introducción a los patrones".

3.1. Asociación histórica

Contexto

Una clase A participa en una asociación de grado N.

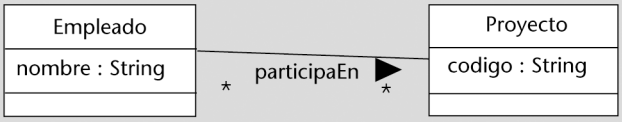
Problema

Queremos poder recuperar los valores que la asociación ha ido tomando a lo largo del tiempo.

En concreto:

- Hay que poder conocer el valor de la asociación en un momento determinado.
- Hay que poder consultar la lista de valores de la asociación en un periodo de tiempo.

Por ejemplo, supongamos que el sistema que estamos desarrollando necesita almacenar información sobre los proyectos en los que ha participado un empleado, por lo cual nos encontramos con el diagrama estático del análisis siguiente:

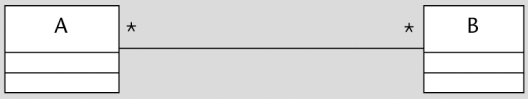


Esta solución, sin embargo, sólo nos sirve para saber, en el momento actual, en qué proyectos participa el empleado. ¿Cómo podemos saber en qué proyectos participó el empleado el año pasado? A esta información sobre el estado del sistema en el pasado la llamamos "información histórica" y es una situación bastante habitual en el desarrollo de sistemas de información.

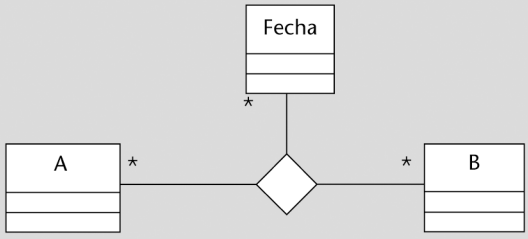
Solución

Añadir una dimensión a la asociación (convirtiendo la asociación *n*-aria en *n*+1-aria) que represente el tiempo.

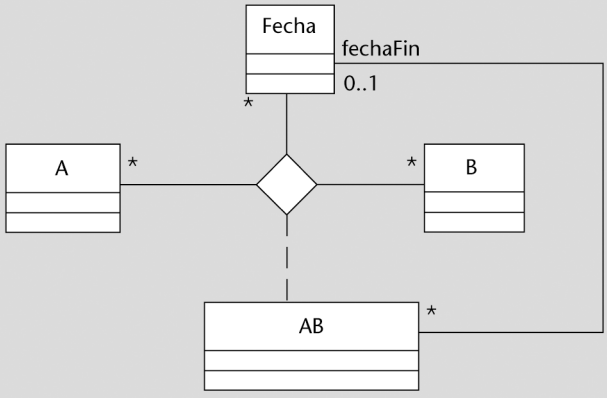
Para una asociación como la siguiente:



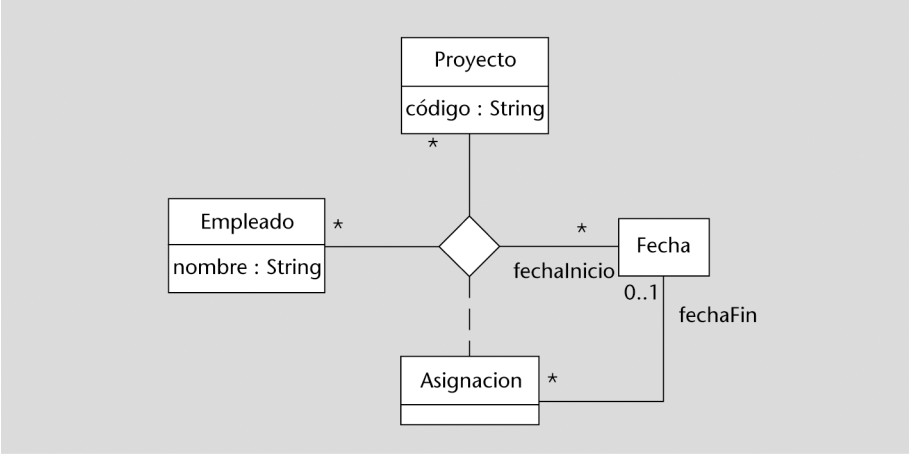
la solución sería ésta:



Si los valores que toma la asociación tienen un determinado periodo de validez, el momento final de este periodo se indicará como asociación de la clase asociativa:



Volviendo al caso de los proyectos de los empleados, la solución sería la siguiente:



Así, pues, ahora nuestro sistema sabe, para cada pareja Empleado-Proyecto, la fecha en la que se produjo esta asignación y la fecha en la que la asignación finalizó. Para saber cuál es la asignación actual, tendremos que mirar qué fecha es la más reciente o también podríamos considerar que es aquella que todavía no tenga fecha final.

A la hora de definir las nuevas cardinalidades, hay que ir con cuidado. En este caso, la cardinalidad al lado de Empleado nos indicará cuántas instancias de empleado pueden estar asociadas a un mismo proyecto con la misma fecha, pero no nos indica nada con respecto a otros momentos en el tiempo (por ejemplo, un empleado puede estar asignado más de una vez a un proyecto siempre que la fecha de asignación sea diferente).

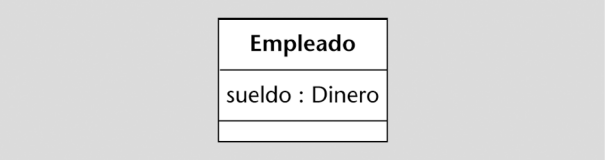
Consecuencias

- Permite reflejar la información histórica correctamente.
- El resultado es más complejo, ya que se ha aumentado el grado de la asociación.
- No se puede trasladar la multiplicidad de la asociación original a la asociación histórica.
- El diagrama no indica que los periodos de tiempo no se pueden solapar.

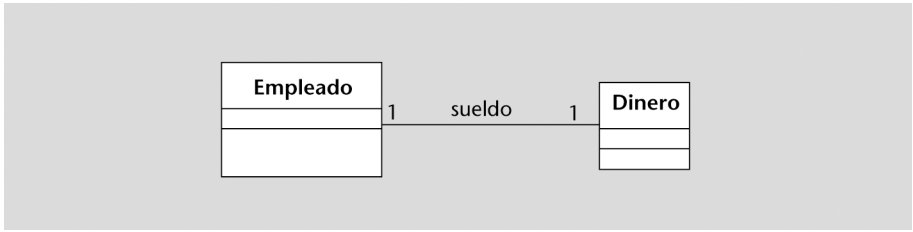
Variaciones

Si en lugar de una asociación lo que tenemos es un atributo del cual queremos conocer el historial de valores, hay que hacer una transformación previa: representamos el atributo como una asociación binaria entre la clase y el tipo de datos que corresponda y después aplicamos el patrón normalmente.

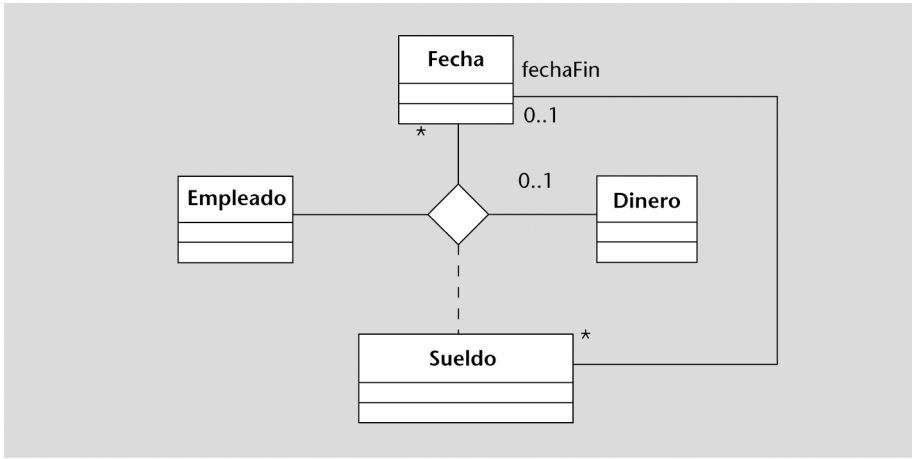
Por ejemplo, suponemos que el sistema que estamos desarrollando tiene empleados cuyo sueldo se quiere conocer históricamente:



Para aplicar este patrón, primero tenemos que convertir el atributo en una asociación binaria:



Una vez hecha la transformación, aplicamos el patrón de la manera que hemos visto anteriormente:



Algunos detalles sobre el ejemplo de aplicación del patrón:

- En este caso hemos supuesto que un mismo empleado no puede cambiar de sueldo más de una vez en una misma fecha (restricción que se refleja en el diagrama por la cardinalidad 0..1 en el lado Dinero). Si no fuera así, la cardinalidad al lado de Dinero sería *.

- El diagrama no refleja, sin embargo, que un empleado no puede tener dos sueldos en una misma fecha (es decir, que los periodos de tiempos por los cuales se define cada sueldo no se pueden solapar).

Variaciones

En [FOW] podemos encontrar un conjunto de patrones de análisis dedicados a aspectos más concretos relacionados con la problemática de representar el tiempo.

3.2. Objeto compuesto (*Composite*)

Contexto

Hay una asociación que agrupa elementos en colecciones.

Problema

Queremos tratar las colecciones de elementos y los elementos de manera uniforme.

Por ejemplo, supongamos que el sistema de información que estamos desarrollando utiliza archivos para almacenar información. De un usuario concreto conocemos los archivos que le pertenecen, y también el nombre y los permisos de cada archivo:

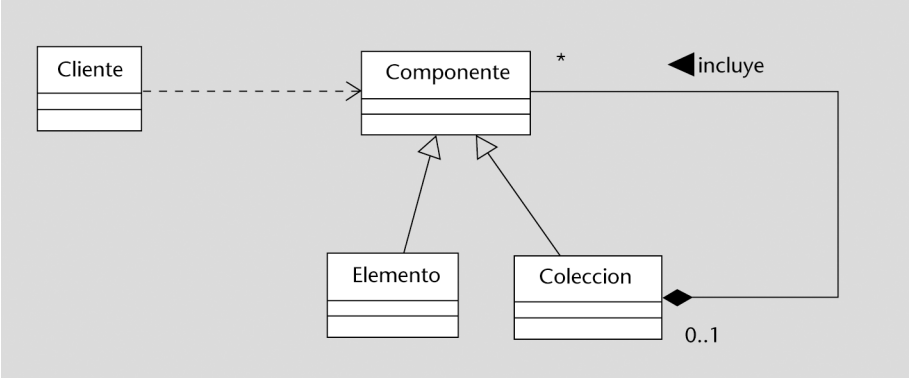
¿Patrón de análisis o de diseño?

Este patrón habitualmente se considera un patrón de diseño. Lo hemos incluido como patrón de análisis porque también puede ser muy útil en esta etapa del ciclo de vida del desarrollo de software.

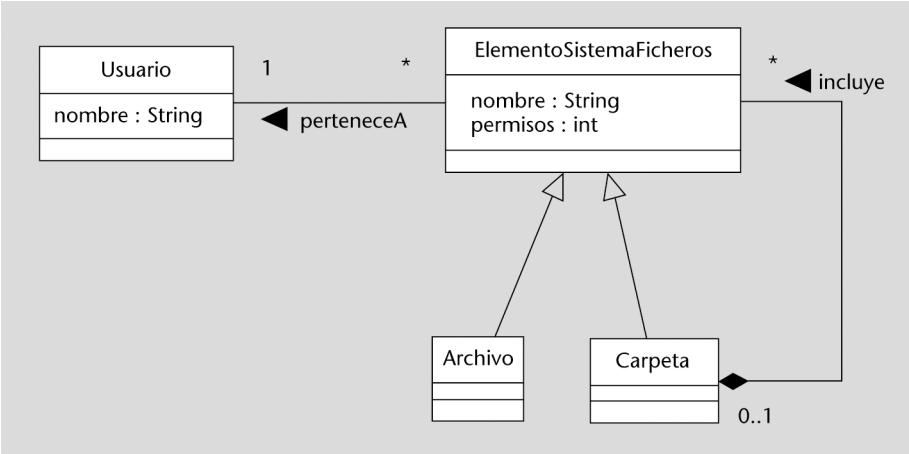
Ahora, sin embargo, queremos añadir la posibilidad de agrupar los archivos en carpetas. Sobre estas carpetas queremos aplicar el mismo tratamiento que sobre los archivos (es decir, también tendrán un nombre y unos permisos). ¿Cómo lo podemos hacer?

Solución

Crear, mediante una generalización, una superclase común a los elementos y a las colecciones y hacer que el resto del sistema no conozca la subclase concreta con la que está asociado.



En nuestro ejemplo, el resultado de aplicar el patrón sería el siguiente:



Consecuencias

- Elementos y colecciones son del mismo tipo, lo cual soluciona el problema del tratamiento uniforme.
- Solución más compleja: hemos hecho una generalización y, por lo tanto, se ha añadido una clase nueva.

3.3. Cantidad (*Quantity*)

Contexto

Se quiere registrar una cantidad.

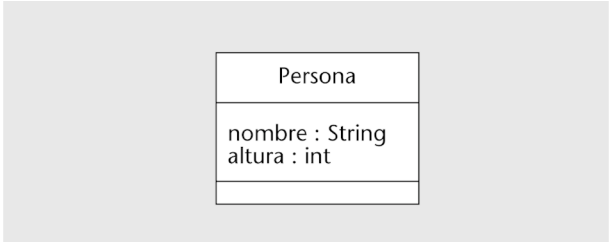
Problema

La representación de una cantidad mediante un valor numérico es poco adecuada, puede llevar a confusión o no es viable cuando pueden intervenir distintas unidades de medida.

En concreto, podemos querer:

- que la unidad de medida sea explícita;
- utilizar distintas unidades de medida;
- poder hacer la conversión entre unidades de manera sencilla.

Por ejemplo, supongamos que el sistema de información que estamos desarrollando tiene que gestionar información sobre personas, concretamente la altura, por lo cual añadimos un atributo de tipo entero que contiene la altura de la persona:



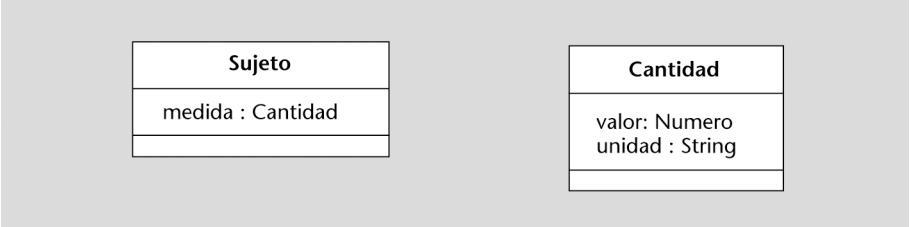
Esta solución, sin embargo, no es suficiente, ya que no nos indica en qué unidad está expresada la altura. ¿Está en metros? ¿En centímetros? ¿En milímetros? Una posible solución puede ser indicar esta unidad de medida como parte del nombre del atributo. Así, por ejemplo, podríamos cambiar el nombre del atributo por alturaEnCm para indicar que la altura se indica en centímetros. Pero entonces hay todavía algunos problemas que no se pueden resolver:

- La unidad de medida está implícita en el nombre del atributo y, por lo tanto, queda poco clara con respecto al análisis.
- Se fuerza siempre a utilizar la unidad de medida indicada en el atributo.
- Si la medida de que se dispone está en otra unidad de magnitud, habrá que hacer la conversión.
- El análisis no refleja cómo se puede hacer esta conversión.
- No se pueden registrar medidas diferentes en unidades diferentes.

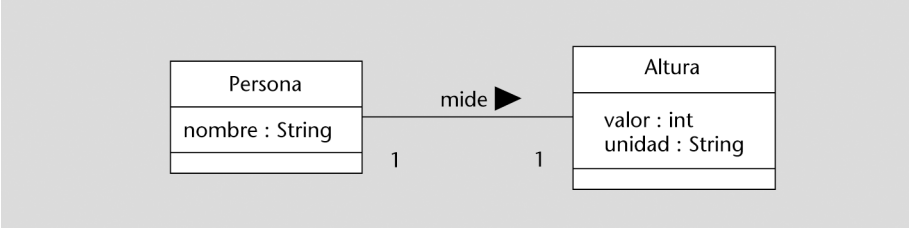
¿Qué solución se puede aplicar en una situación como ésta que nos permita solucionar todos los problemas mencionados antes?

Solución

Modelar la medida como un tipo de dato más complejo formado por los campos valor y unidad.



Podemos ver, pues, que de esta manera podemos solucionar los problemas que hemos mencionado antes con respecto al análisis y, más adelante, durante la etapa del diseño, podremos solucionar el problema de las conversiones entre unidades sin grandes dificultades. En nuestro caso, el resultado sería el siguiente:



Tenemos que tener en cuenta que, si dos personas tienen la misma altura (por ejemplo, 1,70 cm), cada una tendrá asociada una instancia diferente de Altura con los mismos valores en sus atributos. De esta manera, si se modifica la altura de una de las dos personas, no se modificará la de la otra. Por ello tenemos un 1 en la cardinalidad del lado de Persona.

Consecuencias

- Tenemos más clases, lo cual provoca que el modelo gane expresividad a cambio de perder simplicidad.

- Podemos tratar la cantidad como un tipo abstracto de datos, añadiendo el comportamiento que creamos necesario (conversiones, comparaciones, etc.) aumentando, de esta manera, la cohesión global del sistema modelado.

- La unidad de medida es explícita en el diagrama estático del análisis.

- Se pueden utilizar diferentes unidades de medida.

Variaciones

Otros patrones de análisis documentados en [FOW] (ved bibliografía) y que no veremos con detalle se relacionan con el patrón Cantidad, lo cual permite un modelo de análisis mucho más completo que prevea, entre otras cosas:

- Conversión entre unidades de una misma magnitud (como, por ejemplo, las ratios de conversión de centímetros a pulgadas).

- Unidades compuestas (como, por ejemplo, el m/s^2 para medir aceleraciones).

- Medidas y observaciones: tener conocimiento del momento y el mecanismo por el cual se hizo una observación.

Estos otros patrones de análisis, sin embargo, ya son mucho más específicos para dominios de ámbitos científicos y, especialmente, médicos.

3.4. Rango (*Range*)

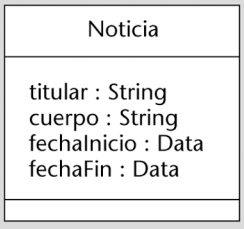
Contexto

Queremos representar un rango de valores.

Problema

Nuestro análisis debería reflejar la semántica propia de un rango, como saber si dos rangos se solapan o si un valor está dentro de un rango.

Por ejemplo, supongamos que el sistema de información que estamos desarrollando permite publicar noticias en Internet. Estas noticias tendrán un periodo de validez. Una solución podría ser añadir estas fechas a la clase Noticia:

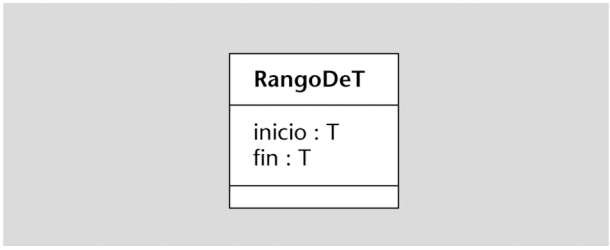


El problema de esta solución es que estamos añadiendo a la clase Noticia todas las responsabilidades asociadas a la gestión del rango de fechas (compararlas, gestionarlas, etc.). Además, si tenemos más clases con rangos de fechas, la solución se complica, ya que el conocimiento sobre el funcionamiento de los intervalos de fechas se encuentra disperso:

Solución

Crear una clase que representa un rango de valores y que se encargará de contener la semántica del rango.

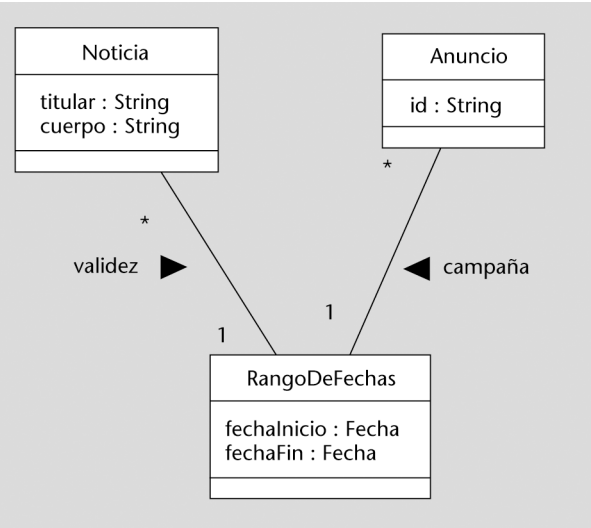
Esta clase tendrá un valor de inicio y un valor de final. Consideraremos que una instancia del rango contiene todos los valores situados entre el valor de inicio y el valor de final y que dos instancias se solapan si contienen algún valor común:



Ejemplo

Opcionalmente, también podríamos utilizar una clase parametrizada para representar el rango.

En el caso de las noticias y los anuncios, la solución sería la siguiente:



Consecuencias

- Sólo se puede aplicar con tipos que soporten la comparación (es decir, que tengan definido un orden por defecto), aunque se podría hacer una versión más sofisticada en la cual se pudiera indicar el criterio de ordenación.

- ¿Cómo tratamos los rangos abiertos (por ejemplo, mayor que seis)? Podemos utilizar un valor especial (el valor nulo, o el máximo representable, etc.) como indicador de la falta de límite y encapsular este hecho dentro de la clase de manera que los clientes no se vean afectados.

- Además de las operaciones que indican que un valor pertenece a un rango, podemos añadir operaciones que comparen rangos, que nos digan si se solapan, etc.

Variaciones

En lugar de tener un atributo principio y un atributo final, podemos tener un atributo principio y una longitud.

4. Patrones de arquitectura

Los patrones de arquitectura son aquellos que se aplican en la definición de la arquitectura de software y que, por lo tanto, resuelven problemas que afectarán al conjunto del diseño del sistema.

Ved también

Podéis encontrar más información sobre los patrones de arquitectura en el apartado 2.2 del módulo "Introducción a los patrones".

4.1. Arquitectura en capas (*Layers*)

Contexto

Queremos estructurar nuestro sistema de manera que cada componente de éste trabaje a un cierto nivel de abstracción.

Problema

- Nuestro sistema tiene que trabajar a diferentes niveles de abstracción.
- Los componentes de alto nivel no pueden utilizar directamente servicios con un nivel de abstracción mucho más bajo porque queremos minimizar la complejidad del sistema resultante. Por ejemplo, no queremos que las clases de interfaz gráfica accedan directamente a la base de datos.
- Se quiere diseñar una arquitectura que evite que los cambios en el código de una parte se propaguen por todo el sistema reduciendo el acoplamiento entre partes.
- Se quiere evitar que la lógica de la aplicación y la interfaz de usuario de ésta estén excesivamente acopladas para poder reutilizar, en caso de que sea necesario, la misma lógica con una interfaz diferente.
- Se quiere evitar que la lógica de negocio y los servicios técnicos de la aplicación estén excesivamente acoplados para poder reutilizar, distribuir o sustituir fácilmente un servicio técnico por una implementación diferente.

Por ejemplo, el sistema que estamos desarrollando gestiona información sobre las nóminas de los trabajadores de una empresa. El sistema trabajará a diferentes niveles de abstracción:

- Elementos de interfaz gráfica (botones, menús, etc.).
- Operaciones de entrada/salida (escrituras en disco, mensajes de red, etc.).
- Consultas y actualizaciones en una base de datos.

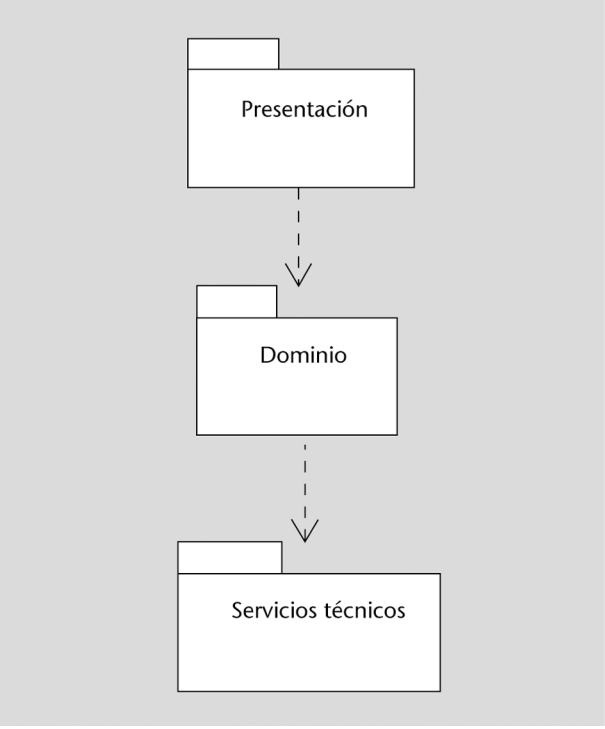
- Clases software que representan conceptos del dominio (empleados, nóminas, etc.).

Solución

Organizar la estructura lógica a gran escala del sistema en capas separadas con responsabilidades diferentes de tal manera que las capas más bajas son servicios generales de bajo nivel y las más altas son más específicas de la aplicación.

La colaboración y el acoplamiento se efectúa desde las capas más altas hacia las más bajas. Una capa de nivel N sólo estará acoplada a la capa de nivel N-1. De esta manera, cada clase tendrá una visión del sistema más coherente y más sencilla.

Una aplicación muy típica de este patrón es la arquitectura en tres capas:



- Presentación: su nivel de abstracción es el de los elementos de interfaz gráfica de usuario (pantallas, botones, etc.) y sus responsabilidades las de recoger las entradas de los usuarios, llamar a la capa de dominio y mostrar los resultados a los usuarios.

- Dominio: su nivel de abstracción es el de los conceptos del sistema y sus responsabilidades las de implementar las reglas de negocio. Las clases de la capa de dominio ven el sistema como un conjunto de clases de entidad (nómina, empleado, etc.) que contienen la información que gestiona el

sistema y una serie de casos de uso u operaciones de sistema que determinan la funcionalidad del mismo.

- Servicios técnicos: su nivel de abstracción es el de la plataforma (sistema operativo, ficheros, etc.) y sus responsabilidades las de aislar en la medida de lo posible la capa de dominio de los elementos de la plataforma.

En nuestro ejemplo, para calcular una nómina, las diferentes capas intervendrán de la manera siguiente:

1) Capa de presentación: sólo nos preocuparemos de saber que se ha pulsado cierto botón y que, como resultado de eso, se tiene que mostrar cierta pantalla con unos datos que recibiremos de la capa de dominio.

2) Capa de dominio: recibiremos una llamada a una operación de una clase que tiene que saber cómo se hace el cálculo a partir de sus parámetros. No nos tenemos que preocupar de qué botones se han pulsado ni de qué pantallas se muestran, como tampoco de en qué ficheros físicos radica la información que tenemos que utilizar.

3) Capa de servicios técnicos: la visión que tienen estas clases del sistema consta de una base de datos, una serie de ficheros auxiliares y, quizás, un servidor remoto al cual enviamos datos a través de la Red. No saben nada de nóminas, ni empleados, ni interfaces gráficas de usuario y sólo tienen que atender peticiones de lectura/escritura en disco, consultas en la base de datos, etc.

Consecuencias

- Reduce la complejidad del sistema, ya que cada componente trabaja a un único nivel de abstracción claramente definido por la capa que lo contiene.

- Permite el intercambio de una capa entera por otra en un sistema con un impacto mínimo.

- Reduce la eficiencia a causa del aumento potencial de niveles de indirección entre los objetos a la hora de ejecutar una tarea. Además, se puede dar el caso de que una capa haga tareas que las capas superiores no necesiten.

- Facilita la reutilización de capas enteras gracias a la definición de interfaces claras entre ellas. Esta reutilización puede ser, por lo tanto, de un volumen mayor de clases que la reutilización de componentes individuales.

4.1.1. Variantes

El número y la función de las capas puede variar de un sistema a otro y, de hecho, son dos de las variables a tener en cuenta en el momento de aplicar el patrón. Por ejemplo, podemos utilizar dos capas como en los sistemas cliente/servidor.

4.2. Inyección de dependencias (*Dependency injection*)

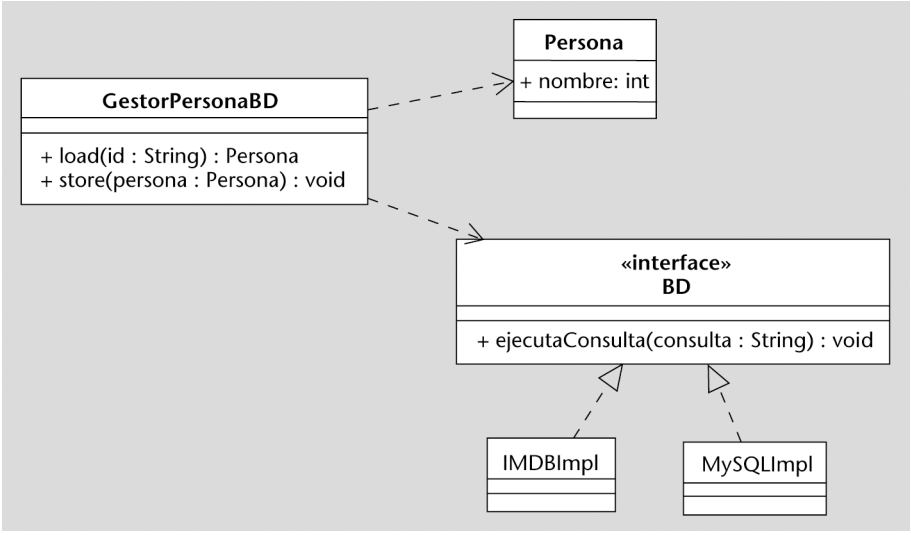
Contexto

En nuestro sistema tenemos definidos una serie de servicios (como el envío de mensajes de correo electrónico o un servicio de persistencia) para los cuales tenemos (o podemos tener en un futuro) varias implementaciones.

Problema

Queremos escoger y cambiar la implementación de cada servicio sin tener que modificar el código de nuestras clases, que no dependerán de ninguna implementación concreta, sino de la interfaz del servicio.

Supongamos que estamos desarrollando un sistema que tiene que acceder a una base de datos. Algunas de las clases necesitarán utilizar un componente de acceso a base de datos para llevar a cabo sus funciones:



En este caso, tenemos dos implementaciones de la misma interfaz. Se podría dar el caso de que, en el entorno de desarrollo, trabajáramos con una base de datos en memoria para acelerar la ejecución (IMDBImpl) y, en el entorno de

explotación, con un servidor de bases de datos (MySQLImpl). En este caso, necesitamos un mecanismo que nos permita escoger la implementación que utilizaremos de manera transparente en la clase GestorPersonaBD.

Solución

Ofrecer operaciones para poder inyectar en las clases usuarias de un servicio la implementación de éste. Esta inyección la llevará a cabo una tercera clase que, típicamente, formará parte de un *framework* especializado.

Esta clase o *framework* nos tiene que permitir configurar qué implementación queremos utilizar de manera transparente a nuestras clases (por ejemplo, mediante un fichero de configuración).

En nuestro caso, éste sería el resultado de aplicar la inyección de dependencias a la clase GestorPersonaBD:

```
public void setBD(BD implementacion) {
  this.bd = implementacion;
}
public void load(String id) {
  bd.ejecutaConsulta(SELECT...");
}
```

Consecuencias

- Las dependencias son explícitas, ya que aparecen como operaciones o atributos de la clase Cliente. Por ejemplo, setBD en la clase GestorPersonaBD hace explícita la dependencia de GestorPersonaBD hacia BD.

- Las dependencias se mezclan con las operaciones o atributos propios de la clase. En el ejemplo anterior, no queda claro si BD es una propiedad de la clase PersonaBD o una dependencia.

- Se facilita la reutilización al no depender de ninguna implementación concreta de inyección de dependencias (si queremos distribuir nuestras clases, no habrá que distribuir el servicio de inyección de dependencias).

- Es más fácil probar nuestra clase de manera aislada, ya que no necesitamos modificar el mecanismo de resolución de dependencias, sino, simplemente, configurar la instancia sobre la cual ejecutaremos las pruebas unitarias con las implementaciones que más nos convengan para hacer la prueba. Por ejemplo, para probar una instancia de GestorPersonaBD sólo le tene-

Bibliografía complementaria

Podéis encontrar más información sobre pruebas unitarias de clases software en [BECK].

mos que proporcionar una implementación de BD llamando a la operación setBD.

- Es fácil identificar cada servicio, ya que todos tienen un nombre (este nombre estará determinado por el nombre de la operación que utilizamos para inyectar la implementación del servicio).

- Tenemos que garantizar que toda instancia esté configurada correctamente en el momento de utilizarla.

Variaciones

- Inyección por constructor: las dependencias son parámetros del constructor de nuestro objeto. Eso nos asegura que es imposible tener una instancia que no esté configurada correctamente, ya que se configura en el mismo momento de su creación. Esta solución es problemática en los lenguajes de programación en los que los parámetros no tienen nombre si tenemos más de uno del mismo tipo, ya que no hay manera de diferenciar los dos servicios.

- Inyección por interfaces: se crea una interfaz que nuestra clase tiene que implementar. En la interfaz se definen las operaciones que se utilizarán para inyectar las dependencias.

Implementaciones

Algunos *frameworks* Java que facilitan la aplicación de este patrón son PicoContainer (www.picocontainer.org) y Spring Framework (www.springframework.org).

4.3. Modelo, vista y controlador (MVC, *Model-view-controller*)

Contexto

Estamos desarrollando un sistema con interfaz gráfica de usuario.

Problema

Queremos desacoplar la interfaz gráfica de nuestro sistema del resto del sistema.

En una arquitectura en capas, queremos definir la arquitectura de la capa de presentación.

A la hora de diseñar las clases de la interfaz gráfica de usuario de nuestra aplicación, tenemos que tener en cuenta que la interacción con el usuario es una tarea muy compleja y que si no hacemos una buena separación de responsa-

bilidades, el mantenimiento de esta parte de la aplicación puede devenir fácilmente muy costoso. Por otra parte, la interfaz de usuario de un sistema es, probablemente, la parte más propensa a cambios.

Tomemos como ejemplo un formulario de edición de los datos de un usuario que permite modificar el nombre. Una vez cambiado el nombre, el usuario tiene que pulsar el botón Aceptar.

Solución

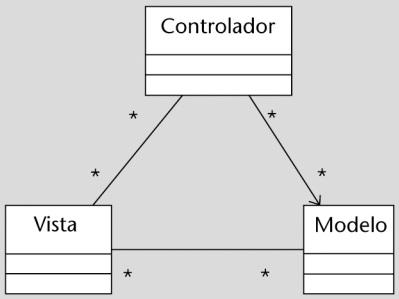
Dividir el sistema en tres tipos de componentes: modelos, vistas y controladores.

Los **modelos** encapsulan el estado del sistema. Su nivel de abstracción es el dominio del problema. Es donde se implementa la llamada *lógica de negocio*[2] . También notifican a las vistas los cambios en el estado del sistema.

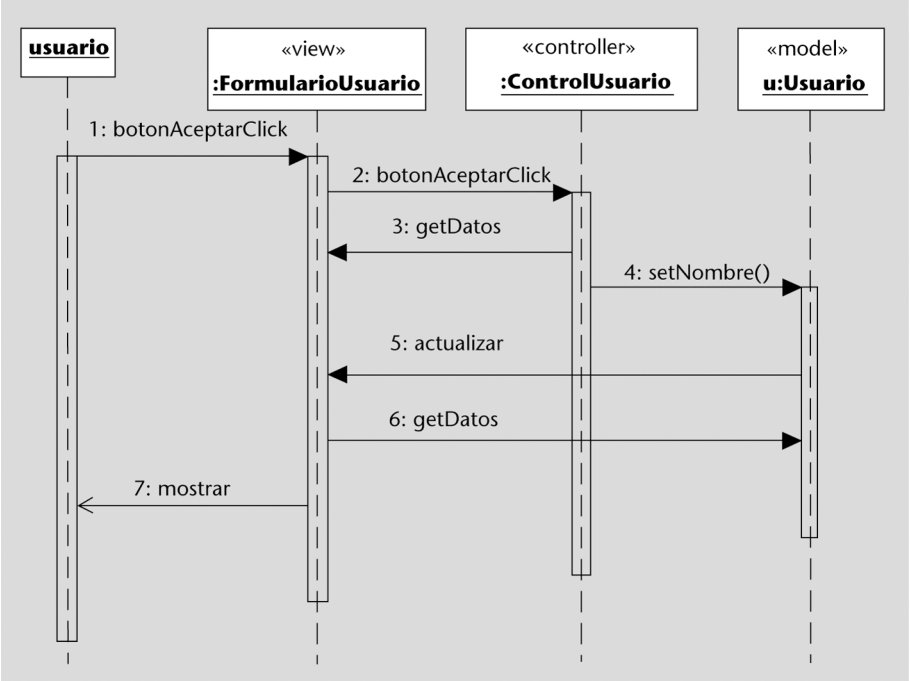
[(2)]Comportamiento del sistema en términos del dominio.

Las **vistas** presentan los datos a los usuarios y recogen las interacciones (por ejemplo, se ha pulsado un botón de una pantalla) para enviarlas a los controladores. Piden el estado del sistema al modelo.

Los **controladores** establecen la correspondencia entre las acciones del usuario y los acontecimientos del sistema. También deciden qué vistas se muestran a los usuarios:

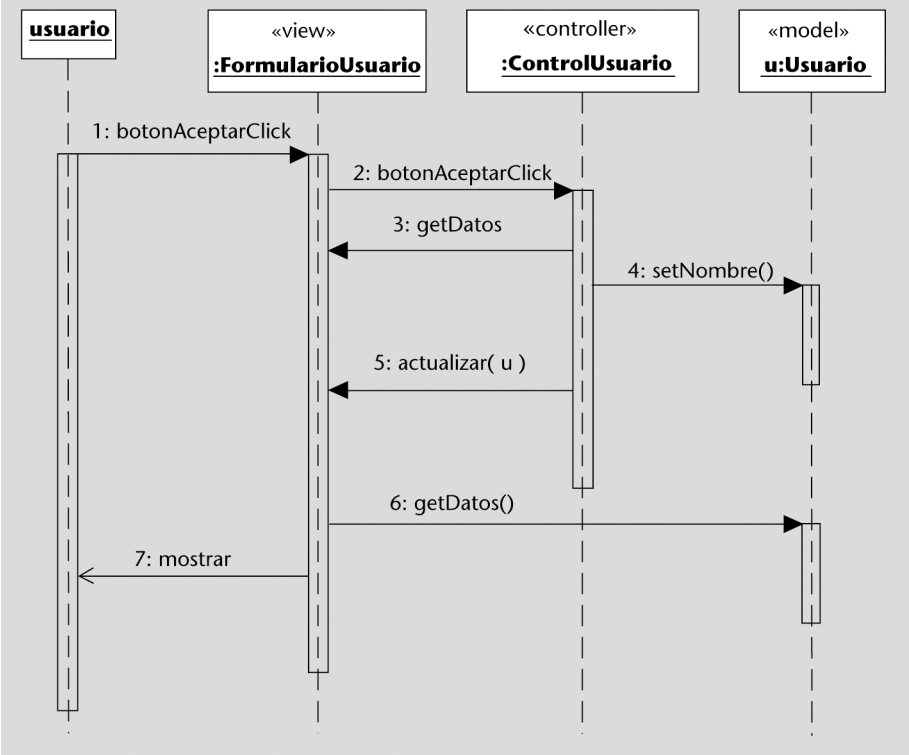


En nuestro ejemplo, cuando el usuario pulse el botón Aceptar, la vista llamará al controlador; éste traducirá el acontecimiento a una operación de sistema sobre el modelo (setNombre de un usuario). Como resultado del cambio en su estado, el usuario notificará a las vistas que se tengan que actualizar:



En una arquitectura en capas, el modelo estará formado por el conjunto de clases de la capa de dominio. Sin embargo, como la capa de dominio no puede estar acoplada a la capa de presentación, los modelos no actualizarán las vistas directamente. Una manera de solucionar este problema es que los controladores, que conocen el modelo y provocan los cambios, sean los encargados de notificar a las vistas. De esta manera, los controladores, después de modificar el modelo, avisan a las vistas a fin de que se actualicen.

En una arquitectura en capas, por lo tanto, nuestro ejemplo funcionaría de la manera siguiente:



Consecuencias

- La cohesión de las clases es más alta porque todas sus responsabilidades están relacionadas.

- Podemos soportar más de una tecnología de visualización (por ejemplo, interfaces web e interfaces "rich client").

- Los desarrolladores se pueden especializar en un dominio concreto, sea el dominio de la aplicación o el dominio del desarrollo de interfaces gráficas.

- Podemos actualizar la lógica de visualización independientemente de la lógica de negocio (o actualizar la lógica de negocio sin tener que modificar la lógica de visualización).

- Si queremos que las vistas se actualicen automáticamente, cada vez que haya un cambio en el modelo tendremos que establecer un mecanismo de notificación de estos cambios. Eso nos puede crear fácilmente una dependencia de la capa de dominio en la capa de presentación, cosa que no es deseable si estamos siguiendo una arquitectura en capas.

5. Patrones de asignación de responsabilidades

Los patrones de asignación de responsabilidades son un tipo especial de patrón de diseño que no se aplican para resolver un problema concreto de diseño, sino que se aplican, de manera general, para repartir las responsabilidades entre las diferentes clases del diagrama de clases. Así pues, su aplicación acostumbra a ser previa a la aplicación del resto de patrones de diseño.

Ved también

Podéis encontrar más información sobre los patrones GRASP en [LAR] o en el apartado 2.3 del módulo "Introducción a los patrones".

Para documentar estos patrones, modificaremos ligeramente la estructura de la explicación, ya que todos resuelven un mismo problema: cómo asignar las responsabilidades a nuestras clases a fin de que nuestro diseño respete los principios del diseño de calidad.

Ved también

Consultad el catálogo de principios de diseño en el apartado 2 de este módulo didáctico.

5.1. Controlador (*Controller*)

Contexto

Se producen eventos y hay que decidir quién será el responsable de gestionarlos.

Ejemplo

No se debe confundir este patrón con el controlador del patrón Modelo-Vista-Controlador visto en el apartado 4.3. "Modelo, vista y controlador" de este módulo.

En una arquitectura en capas, cuando la capa de presentación detecta que se ha producido un evento que hay que notificar a la capa de dominio (por ejemplo, el usuario quiere dar de alta a un cliente), necesitamos determinar qué clase de la capa de dominio recibirá este evento para tratarlo.

Solución

Crear una clase que denominaremos *Controlador* y asignarle la responsabilidad de gestionar el evento.

Existen distintos tipos de controlador:

- Controlador de fachada: representa el sistema global o un subsistema. Contiene tantas operaciones como eventos se pueden producir en el sistema o subsistema.

- Controlador de caso de uso: representa el caso de uso en el cual se produce el evento. Contiene tantas operaciones como eventos se pueden producir a lo largo de la ejecución del caso de uso.

- Controlador de sesión: representa una sesión de un actor. Una sesión es una instancia de una conversación entre el actor y el sistema y puede tener

cualquier duración. Se pueden producir eventos correspondientes a todos los casos de uso en los que el actor pueda intervenir.

- Controlador de transacción: representa un único evento de sistema. Los parámetros del evento serán atributos de la clase. Sólo contiene una operación que llamaremos *ejecutar*.

Ejemplo

En una aplicación web, por ejemplo, la capa de presentación se encargará de interactuar con los usuarios por medio de su navegador. El navegador interactuará con el sistema enviando peticiones HTTP que el sistema responderá mediante el HTML que el usuario verá. Será la capa de presentación quien traduzca las peticiones HTTP a eventos del sistema en términos de la capa de dominio. Pero, entonces, ¿quién se encargará en la capa de dominio de gestionar estos eventos?

Supongamos que en un caso de uso de compra de productos de una tienda virtual tenemos una petición de consulta de una categoría de productos:

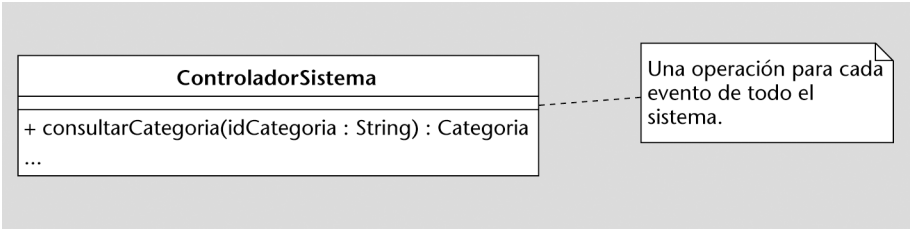
```
GET http://www.emusica.biz/consultarCategoria?idCategoria=321
HTTP/1.1
```

La capa de presentación la traducirá en un evento de sistema que corresponda a una operación de la capa de dominio con la firma siguiente:

```
public Categoria consultarCategoria(String idCategoria)
```

Pero ¿qué objeto de la capa de dominio será el encargado de atender esta petición? Según el patrón Controlador, lo hará un objeto controlador que podemos definir en diferentes niveles:

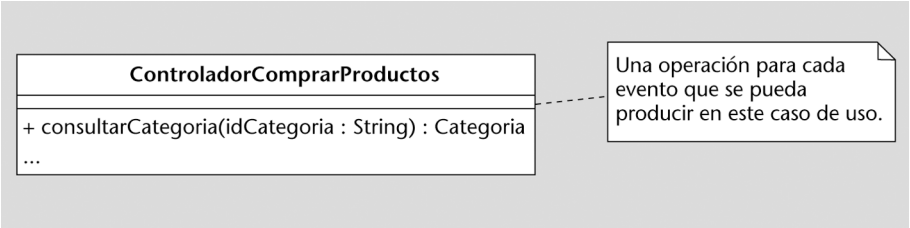
- Controlador de fachada:



Este tipo de controlador tiene la ventaja de ser una solución sencilla y elegante. Cuando lo utilizamos, hay que escoger un nombre que represente el conjunto del sistema y que proporcionará una abstracción de la capa de dominio de cara a otras capas. En caso de duda, ControladorSistema siempre es suficientemente indicativo.

El controlador de fachada será adecuado cuando no haya muchos eventos diferentes o bien cuando se quiera aligerar a la capa usuaria de los controladores de la tarea de ir cambiando de controlador y de escoger en cada caso el controlador más adecuado.

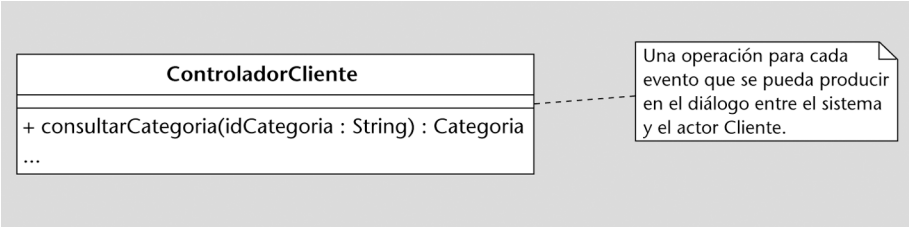
- Controlador de caso de uso:



Esta estrategia es interesante cuando hay muchos eventos diferentes, ya que nos permite agruparlos. En estos casos, el controlador de fachada se convierte en demasiado complicado y nos lleva a diseños con baja cohesión y/o alto acoplamiento.

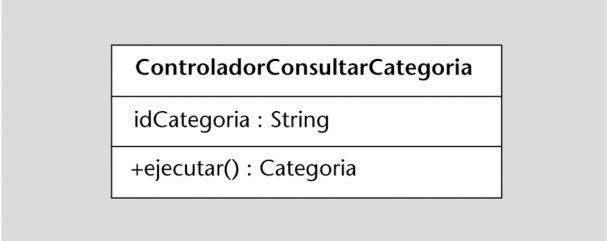
Por otra parte, tiene la ventaja de permitir controlar el estado del caso de uso de tal manera que se pueda gestionar y razonar sobre qué eventos son válidos en cada momento del caso de uso. Finalmente, el análisis basado en los estereotipos de clases de frontera, control y entidad se traduce en este tipo de controlador de una manera muy directa facilitando la tarea de diseño.

- Controlador de sesión:



Estos controladores comparten buena parte de las ventajas de los controladores de casos de uso. Normalmente, el número de controladores siguiendo esta estrategia es inferior y, por lo tanto, pueden ser adecuados en sistemas con un número de eventos medio.

- Controlador transacción:



Finalmente, los controladores transacción tienen la ventaja de que nos permiten tratar cada evento como un objeto. Además, nos permiten extender fácilmente el sistema añadiendo nuevas clases para los nuevos acontecimientos que queramos añadir.

Las cuatro opciones propuestas por el patrón controlador tienen sus ventajas e inconvenientes. Hay que escoger una modalidad que no genere un número excesivo de controladores demasiado simples, pero que no cree tampoco controladores demasiado sobrecargados y difíciles de entender y gestionar.

Corolario

Una conclusión importante de este patrón de diseño es que se tiene que evitar a toda costa que la capa de presentación y sus componentes (como botón, ventana, área de texto para aplicaciones de escritorio o miniaplicaciones de servidor[3] y JSP para aplicaciones web, etc.) sean responsables de gestionar los eventos del sistema. Es decir, los eventos de sistema tendrían que ser gestionados por la capa de dominio.

[(3)]en inglés, *servlet*.

5.2. Experto (*Expert*)

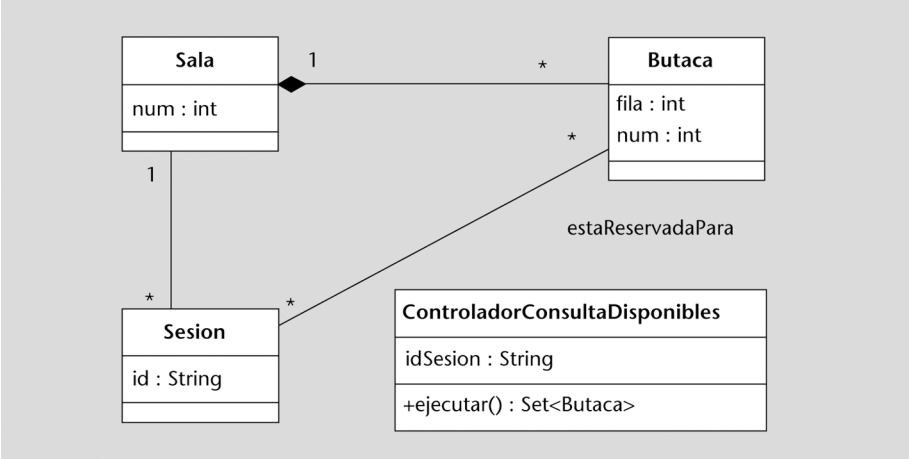
Contexto

Se tiene que hacer un tratamiento o cálculo sobre una información.

Solución

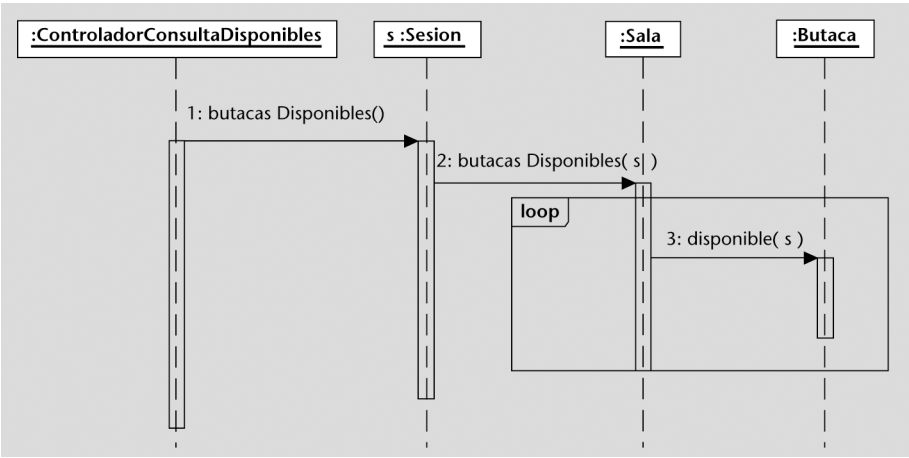
Asignar la responsabilidad de hacer el tratamiento o cálculo a la clase que contiene la información necesaria. Denominaremos a esta clase Experto (con respecto a este tratamiento o cálculo).

Por ejemplo, supongamos que estamos desarrollando un sistema de reserva de butacas para un cine. Queremos diseñar el tratamiento relativo a la consulta de la lista de butacas disponibles para una sesión:



Para diseñar la operación ejecutar() del controlador, partiremos del identificador de la sesión. Por lo tanto, el primer experto que hay que buscar es quien conoce al identificador de una sesión: la misma sesión. Una vez determinada la sesión, tendremos que consultar la lista de butacas de la sala donde se hace la sesión. El experto con respecto a esta lista es la sala.

Finalmente, hará falta saber, para cada butaca de la lista, si está o no reservada para aquella sesión. En este caso, el experto es la butaca. El diseño resultante, pues, es el siguiente:



5.3. Fabricación pura (*Pure fabrication*)

Contexto

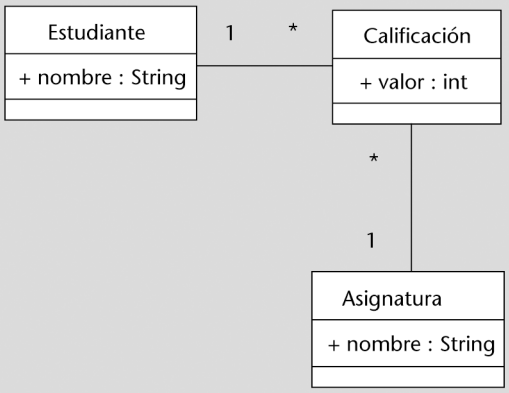
Se tiene que hacer un tratamiento o cálculo independiente de la lógica del dominio.

Solución

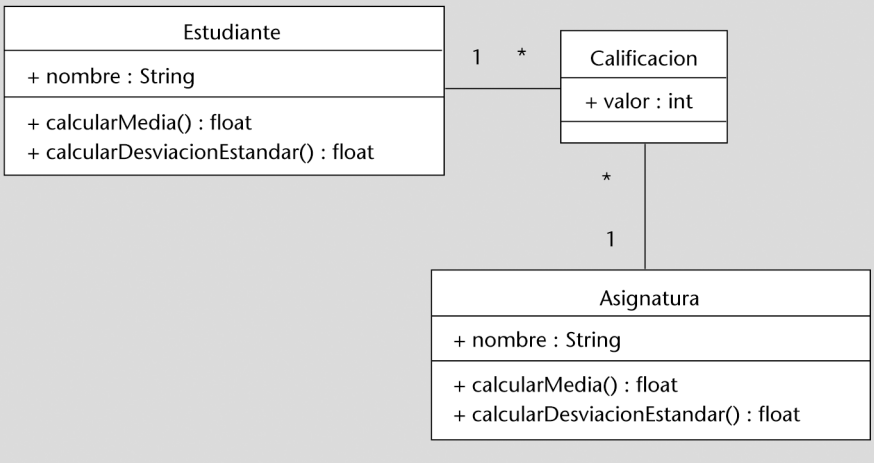
Asignar un conjunto de responsabilidades altamente cohesionadas entre ellas a una clase artificial o de conveniencia que no representa un concepto del dominio del problema; una clase inventada para soportar alta cohesión, bajo acoplamiento y reutilización.

Se denomina *Fabricación pura* porque la nueva clase es una *fabricación* de la imaginación (no existe en el mundo real) y, dado que sus responsabilidades están muy cohesionadas, su diseño es muy limpio, o *puro*.

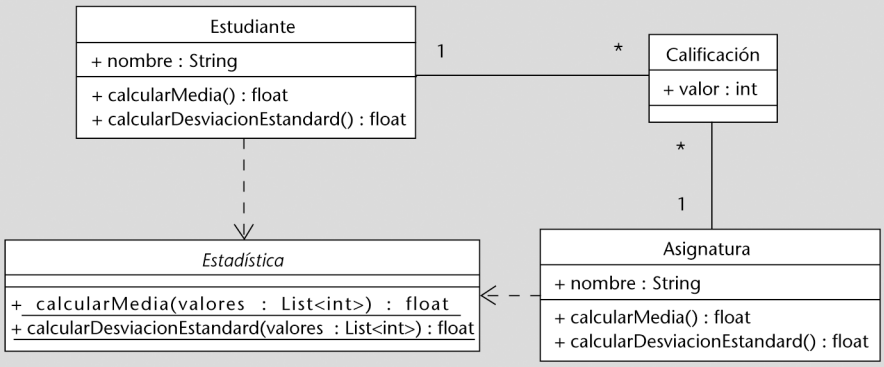
Por ejemplo, supongamos que el sistema que estamos desarrollando tiene que calcular la media y la desviación estándar de las calificaciones de un estudiante, y también las de una asignatura a partir del modelo conceptual siguiente:



Podemos asignar la responsabilidad de hacer el cálculo de la media y la desviación estándar de las calificaciones del estudiante (según el patrón Experto) a la clase Estudiante, y también a la clase Asignatura las de la asignatura, pero esta solución no nos permitirá reutilizar fácilmente el cálculo:



Aplicando el patrón Fabricación pura, podremos asignar la responsabilidad de hacer el cálculo de la media y la desviación estándar de manera que la podamos reutilizar y sólo lo tengamos que implementar una vez:



5.4. Creador (*Creator*)

Contexto

Tenemos que crear instancias de una clase.

Solución

Asignar a la clase B la responsabilidad de crear una instancia de la clase A si se cumplen uno o más de los casos siguientes:

B *agrega* objetos de A.

B *contiene* objetos de A.

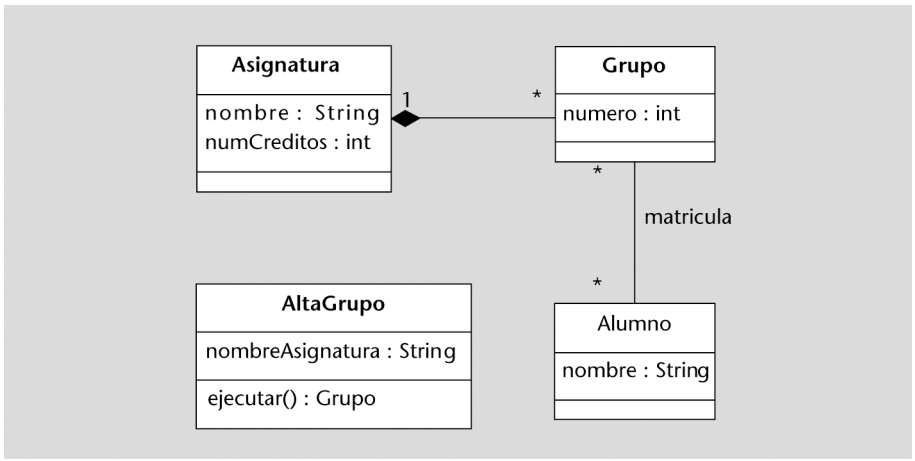
B *registra* objetos de A.

B *utiliza más estrechamente* objetos de A.

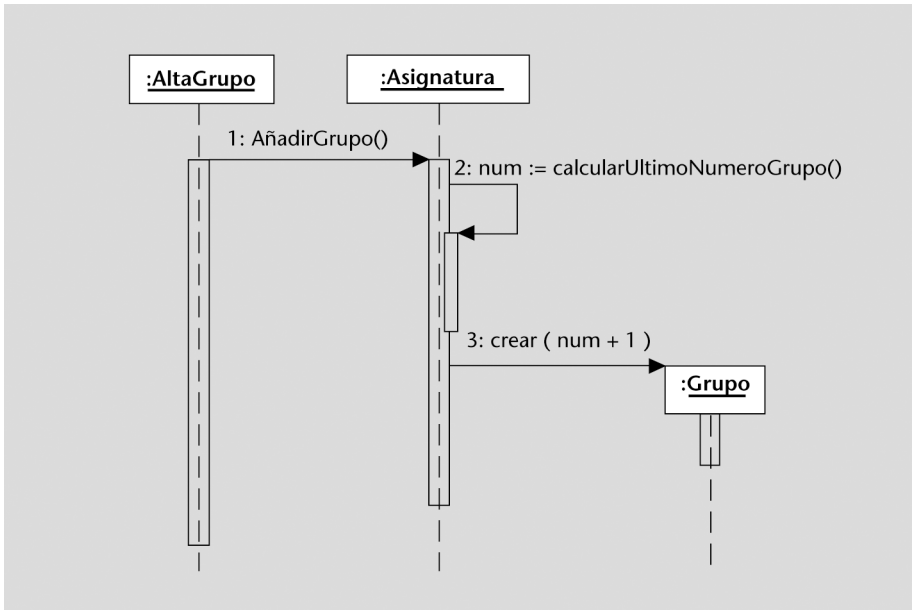
B *tiene los datos de inicialización* que se pasarán a un objeto de A cuando sea creado (y, por lo tanto, B es un Experto con respecto a la creación de A).

Si se puede aplicar más de una opción, es más recomendable inclinarse por una clase B que *agregue* o *contenga* objetos de A.

Por ejemplo, supongamos que estamos desarrollando un sistema de soporte a la matriculación de alumnos de una universidad:



¿Quién será el responsable de la creación de un nuevo grupo de una asignatura? Según el patrón creador, como la asignatura agrega instancias de grupo, será la responsable de la creación de las instancias de grupo. Por lo tanto, la solución será la siguiente:



6. Patrones de diseño

Los patrones de diseño son aquellos que se aplican para resolver problemas concretos de diseño que no afectarán al conjunto de la arquitectura del sistema.

Ved también

Podéis encontrar más información sobre los patrones de diseño en el apartado 2.4 del módulo "Introducción a los patrones".

6.1. Estado (*State*)

Contexto

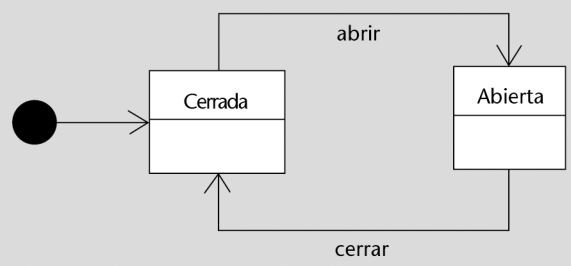
Los objetos de una clase varían su comportamiento dependiendo de su estado.

Tenemos operaciones con grandes bloques condicionales que dependen del estado del objeto. Las combinaciones de estado son comunes en más de un bloque y/o operación.

Problema

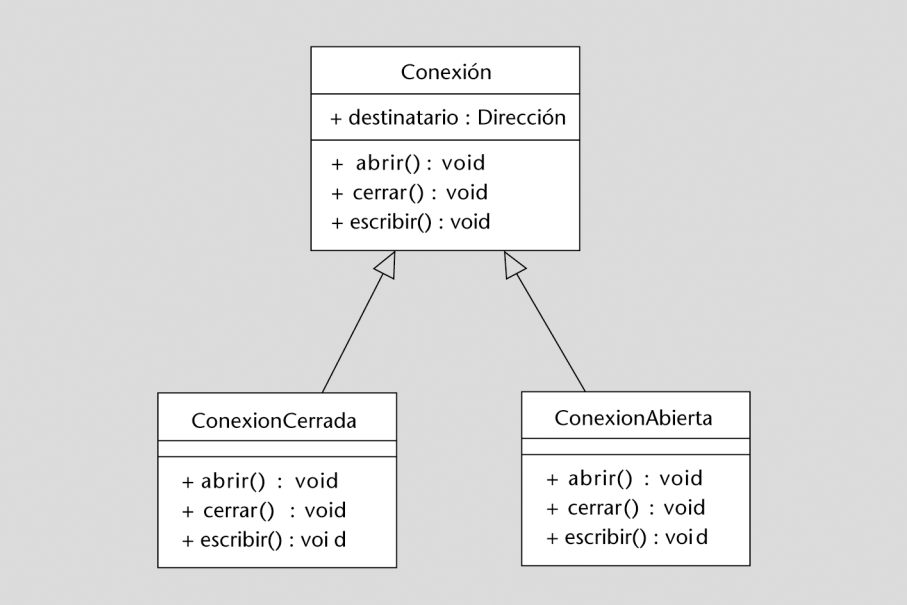
Queremos aprovechar el polimorfismo para escoger el comportamiento adecuado en cada momento sin necesidad de utilizar condicionales, pero no podemos utilizar la generalización dinámica para especificar una subclase diferente para cada estado.

Por ejemplo, imaginemos que durante el desarrollo de nuestro sistema, nos hemos encontrado con una clase denominada *Conexión* que representa una conexión a un servidor remoto. Su diagrama de estados es el siguiente:



Esta clase tiene tres operaciones: abrir, cerrar y escribir. El comportamiento será diferente según el estado: por ejemplo, cuando se invoca la operación cerrar sobre una conexión abierta, se envía un mensaje de final de conexión

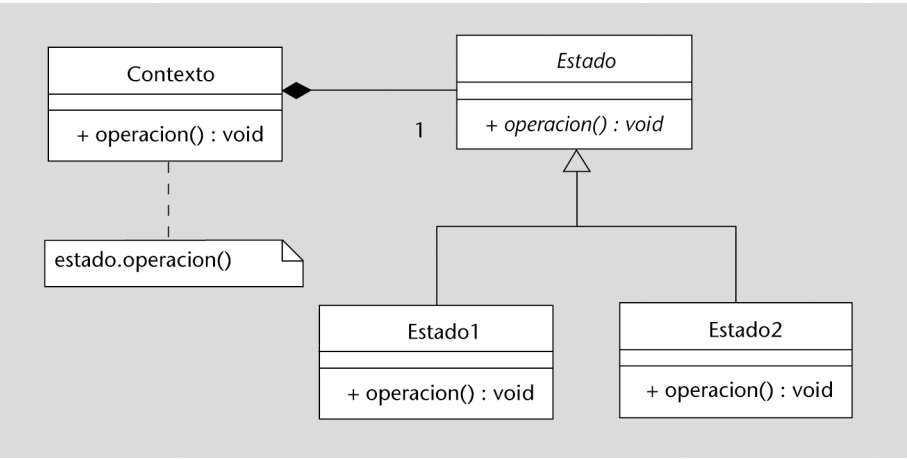
al servidor y se modifica el estado de la conexión (que pasa a estar cerrada), mientras que, cuando está cerrada, llamar a la operación cerrar produce un error. Se nos ocurre un posible diseño que solucionaría nuestro problema:



Esta solución, sin embargo, no nos es válida porque el lenguaje que utilizaremos no soporta la generalización dinámica (es decir, no nos permite que una instancia de conexión cambie de subclase durante la ejecución del sistema), de manera que necesitamos otra forma de resolver el problema.

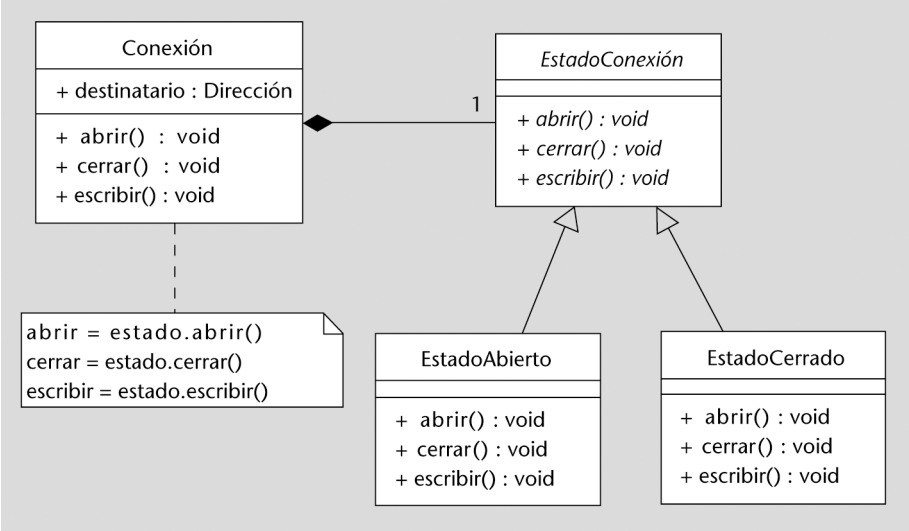
Solución

Separar la parte de la clase que varía dependiendo del estado del resto y tratar cada estado como una clase que implementará el comportamiento esperado cuando el objeto se encuentra en aquel estado.



El Contexto delega en la clase Estado las operaciones que varíen dependiendo de éste, y si es necesario se añade como parámetro.

Aplicado a nuestro caso, obtendríamos el resultado siguiente:



Ahora, para cambiar el estado de la conexión, en lugar de tener que cambiar de subclase, las instancias de conexión sólo tienen que modificar la instancia concreta de EstadoConexion que tienen asociada.

Consecuencias

- Se facilita la extensión del diseño añadiendo nuevos estados.
- Hay que decidir quién es el responsable de efectuar las transiciones entre estado (el contexto o el estado concreto). Esta decisión implicará un acoplamiento entre esta clase y los estados concretos (que afectará a la extensibilidad del diseño).
- Las transiciones de estado se convierten en explícitas.
- Los objetos Estado pueden ser compartidos entre diferentes contextos.
- Hay que determinar cuándo se crean y cuándo se destruyen los objetos Estado (su creación y eliminación puede no estar ligada a la creación y destrucción del contexto).

Variantes

En la solución, hemos supuesto que la asociación entre el contexto y el estado es uno a uno (es decir, para cada objeto contexto tenemos un objeto estado). Modificando esta cardinalidad, llegamos a dos situaciones interesantes: un objeto contexto que esté en más de un estado al mismo tiempo y/o un objeto estado que sirva para más de un contexto.

Para hacer que un objeto contexto esté en más de un estado al mismo tiempo, tendremos que modificar la cardinalidad de la asociación al lado del estado poniendo un asterisco (*). Eso nos puede ser útil cuando las diferencias entre

los estados son sólo relativas a los atributos y asociaciones que se pueden tener, pero será problemático si hay operaciones que estén implementadas de manera diferente en dos de los estados simultáneos.

Para hacer que un objeto estado esté asociado a más de un contexto, tenemos que modificar la cardinalidad de la asociación al lado del contexto poniendo un asterisco (*). En este caso, tenemos que poner en el objeto contexto todos los atributos de todos los estados, ya que si estuvieran en el objeto estado su valor sería compartido por todos los contextos. Esta solución es especialmente interesante si los objetos estado son costosos de crear y/o mantener en memoria.

6.2. Fachada (*Facade*)

Contexto

Nuestro sistema está estructurado en subsistemas o capas. Un subsistema (o capa) A utiliza otro subsistema (o capa) B:

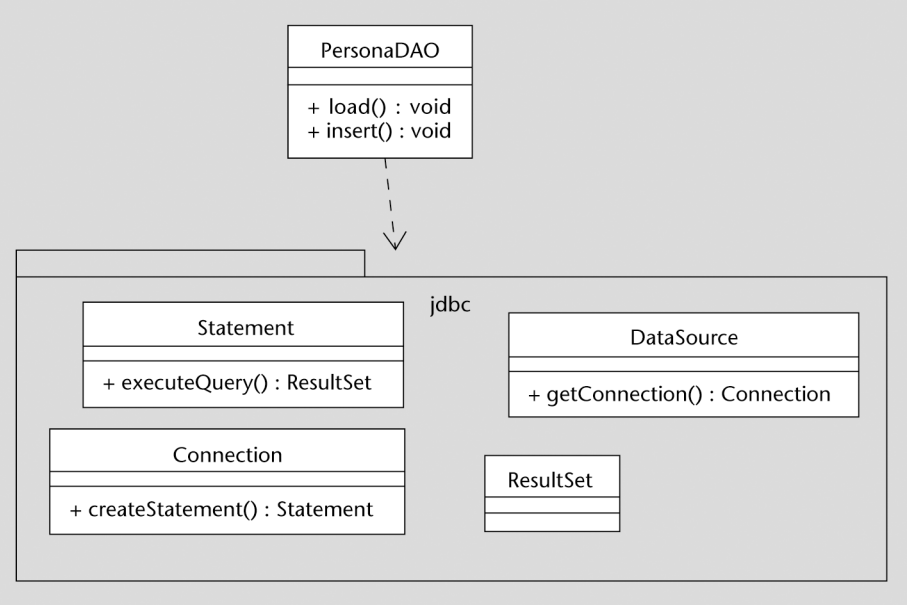
Problema

Queremos reducir el acoplamiento entre dos subsistemas.

Queremos acceder a una versión simplificada de un subsistema.

Queremos ofrecer un único punto de entrada a un subsistema.

Por ejemplo, supongamos que el sistema que estamos desarrollando tiene una clase denominada *PersonaDAO* que contiene las operaciones para acceder a la base de datos de personas. Esta clase tiene que acceder a las interfaces de programación de la base de datos, de manera que está acoplada con todas las clases del subsistema de bases de datos:



Este acoplamiento se repetirá para todas las clases que accedan a la base de datos. Por ejemplo, si añadimos una clase ClienteDAO para acceder a los datos de los clientes, también estará acoplada con todo el subsistema de base de datos.

¿Cómo podemos evitar que tantas clases dependan del subsistema de base de datos?

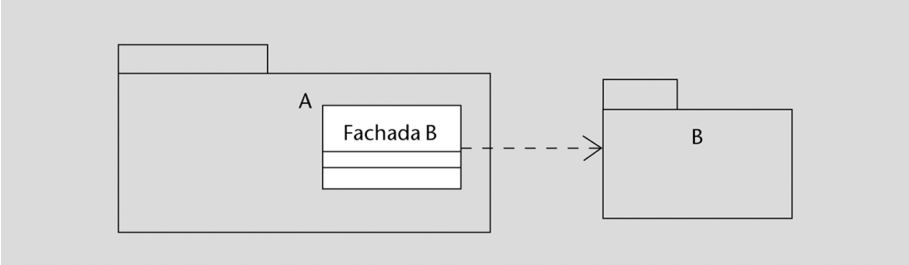
Solución

Crear una nueva clase Fachada que represente todo el subsistema o capa accedido.

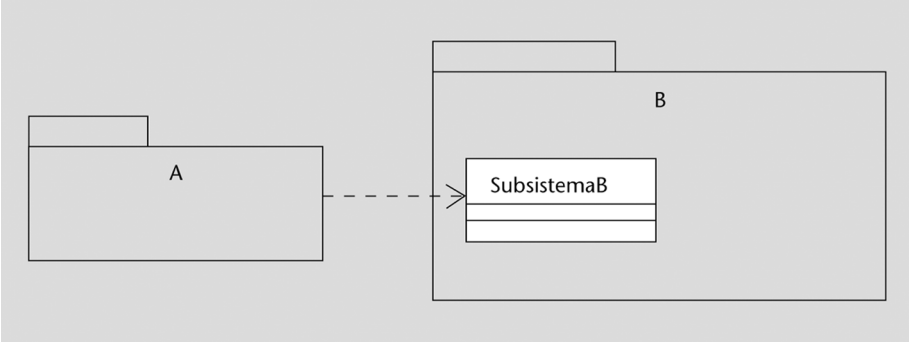
La Fachada concentra la dependencia en un único punto de manera que los cambios en el subsistema utilizado tendrán el mínimo impacto sobre las clases del subsistema cliente. Además, concentrará parte de la complejidad derivada del uso del subsistema que representa y ocultará la estructura de clases interna.

La Fachada puede pertenecer a uno u otro subsitema dependiendo de nuestras necesidades y posibilidades.

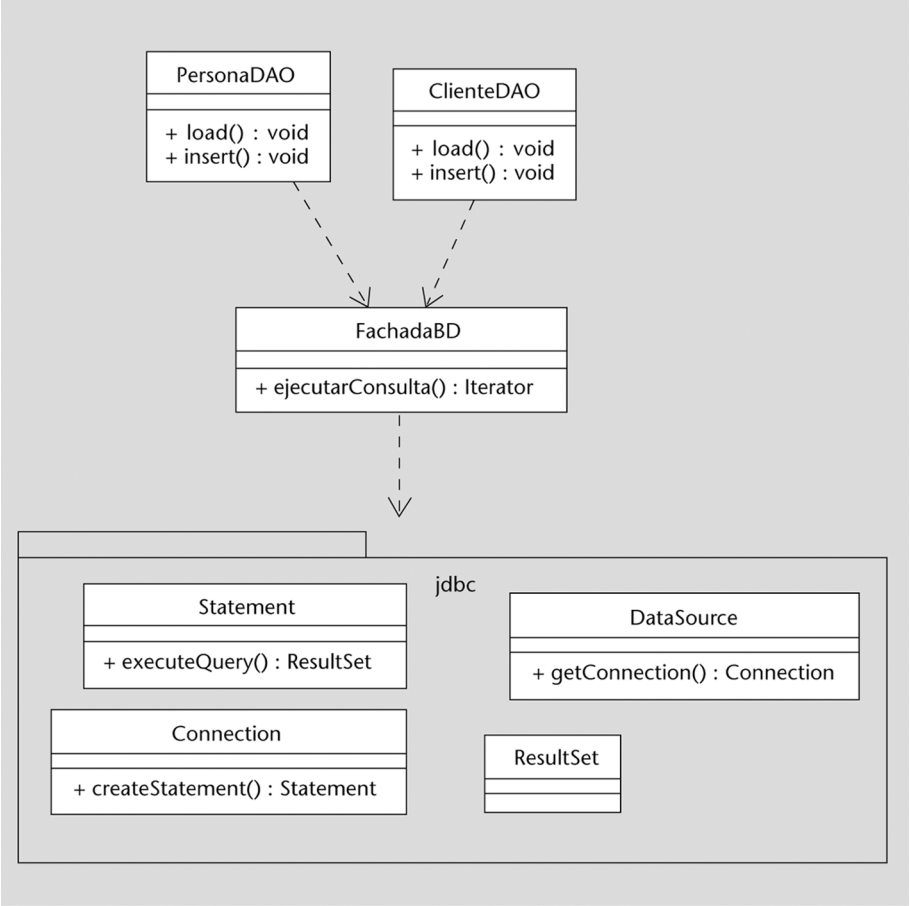
Si lo que queremos es reducir el acoplamiento entre A y B o acceder a una versión simplificada de B, añadiremos una Fachada al subsistema cliente (A) que concentre el acoplamiento y/o nos simplifique la visión que tenemos de B:



Pero si queremos ofrecer un único punto de entrada a B, añadiremos una Fachada a B que será la única clase pública del subsistema (el resto de clases pueden tener visibilidad de paquete) con el fin de asegurarnos de que no hay ninguna otra vía de entrada:



En nuestro ejemplo, queremos reducir el acoplamiento con respecto al subsistema de base de datos y no lo podemos modificar porque es un subsistema desarrollado por otra organización. Por lo tanto, lo que haremos será crear una fachada dentro del subsistema cliente:



Como se puede ver, en el nuevo diseño las dependencias están más concentradas. Cualquier cambio en el subsistema de base de datos afectará sólo a la clase FachadaBD.

Consecuencias

- Las dependencias están concentradas en una única clase y, por lo tanto, si se produce un cambio en el subsistema utilizado, éste afectará sólo a la fachada y no se propagará al resto de clases del subsistema cliente.
- La solución es más compleja. Si la interfaz que ofrece el subsistema B es lo suficientemente estable y sencilla, es posible que la fachada no añada bastante valor para justificar su uso.

6.3. Iterador (*Iterator*)

Contexto

Un objeto agregado quiere dar acceso a sus elementos.

Un objeto quiere dar acceso a los objetos que tiene asociados.

Problema

No queremos exponer la estructura de datos que utilizamos internamente para almacenar las referencias a los objetos asociados o agregados (la representación interna de la agregación o la asociación) a quienes accedan a la misma.

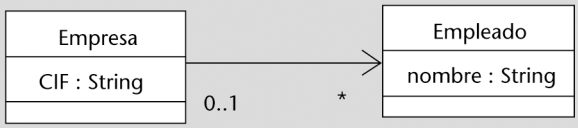
De esta manera queremos evitar el acoplamiento de nuestras clases cliente hacia esta estructura (hecho que nos dificultaría cambiarla por otra), así como posibles manipulaciones de esta estructura sin nuestro control.

Queremos proporcionar diferentes estrategias para acceder a los elementos (por ejemplo, con criterios de ordenación diferentes).

No queremos que la clase agregada tenga la responsabilidad de llevar el control de qué elementos ha mostrado ya a cada clase cliente que pueda acceder a ella.

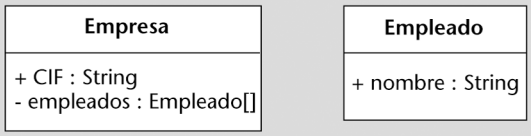
No queremos asignar la responsabilidad de saber cómo se recorre la estructura de datos interna a nuestras clases cliente.

Por ejemplo, supongamos que el sistema que estamos desarrollando tiene las clases siguientes:



Como una empresa tiene que mantener un conjunto de referencias a los empleados que trabajan en ésta, decidimos utilizar un vector[4] para almacenar estas referencias:

(4)En inglés, *array*.



Queremos añadir una operación a Empresa que nos permita consultar el conjunto de los empleados de la empresa. Si devolvemos el vector directamente, estaremos exponiendo la estructura interna de la clase Empresa (con los problemas que eso comporta).

Solución

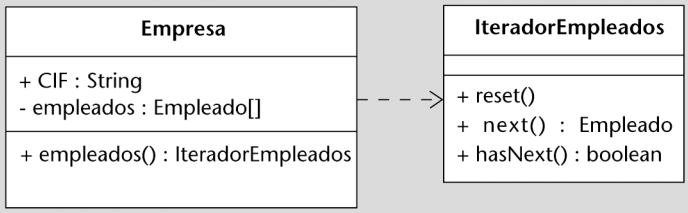
Crear una clase Iterador que conocerá la estructura interna, cómo se tiene que recorrer y cómo se tiene que cambiar (por ejemplo, notificándolo a la clase) y que encapsulará este conocimiento.

Podemos asociar una instancia de iterador a cada cliente de manera que sea el iterador quien tenga la responsabilidad de saber qué elementos ha mostrado y cuáles no a cada cliente.

La clase Iterador proporcionará una interfaz independiente de la representación interna con operaciones genéricas para recorrer los elementos de la estructura:

- reset() Nos permite reiniciar el recorrido.
- next() Avanza hacia el siguiente elemento y lo retorna.
- hasNext() Nos indica si todavía quedan elementos por recorrer.
- También podemos añadir otras operaciones (*remove*, *last*, etc.) según las necesidades concretas de cada caso.

En nuestro caso, la solución aplicando el patrón Iterador sería la siguiente:



En este caso, si quisiéramos hacer un recorrido de los trabajadores de una empresa haríamos:

```
IteradorEmpleados empleados = empresa.empleados();
while(empleados.hasNext()) {
  Empleado actual = empleados.next();
}
```

Una posible implementación de IteradorEmpleados sería la siguiente:

```
public class IteradorEmpleados {
  Empleado[] datos;
  int posicioActual;
  IteradorEmpleados(Empleado[] empleadosEmpresa) {
    datos = empleadosEmpresa;
    posicionActual = -1;
  }
```

```
    public void reset() {
      posicionActual = -1;
    }
    public boolean hasNext() {
      return posicionActual + 1 < datos.length;
    }
    public Empleado next() {
      posicionActual++;
      return datos[posicionActual];
    }
  }
```

Consecuencias

- Podemos modificar la representación interna sin tener que modificar las clases Cliente.
- Podemos variar la estrategia utilizada para acceder a los elementos (por ejemplo, modificando el criterio de ordenación).
- Podemos restringir la capacidad de modificación de los elementos agregados. Si exponemos la representación interna, los clientes la podrían modificar sin que la clase agregada se enterara. Con un Iterador, éste podría notificar la clase agregada.
- Se simplifica la interfaz del objeto agregado a causa de la reducción de responsabilidades (las delega al Iterador).

6.4. Método factoría (*Factory method*)

Contexto

Un objeto tiene que crear objetos de otra clase.

Problema

El objeto que tiene que crear otros objetos no puede anticipar la clase de éstos.

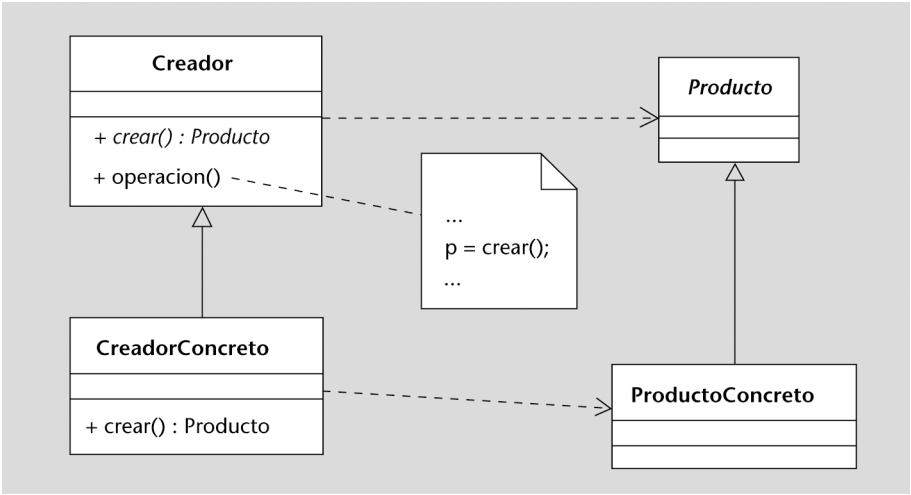
Queremos delegar la responsabilidad de especificar la clase de los objetos que se tienen que crear en nuestras subclases.

Una clase delega una responsabilidad a una o más clases de ayuda y queremos aislar la responsabilidad de decidir qué clase de ayuda utilizaremos.

Supongamos, por ejemplo, que estamos desarrollando un sistema que tiene que notificar ciertos acontecimientos a los usuarios. La clase encargada de gestionar las notificaciones tiene que crear un mensaje y enviarlo, pero queremos delegar en las subclases la selección del tipo de mensaje a enviar (por ejemplo, de correo electrónico o un mensaje MMS a un teléfono móvil).

Solución

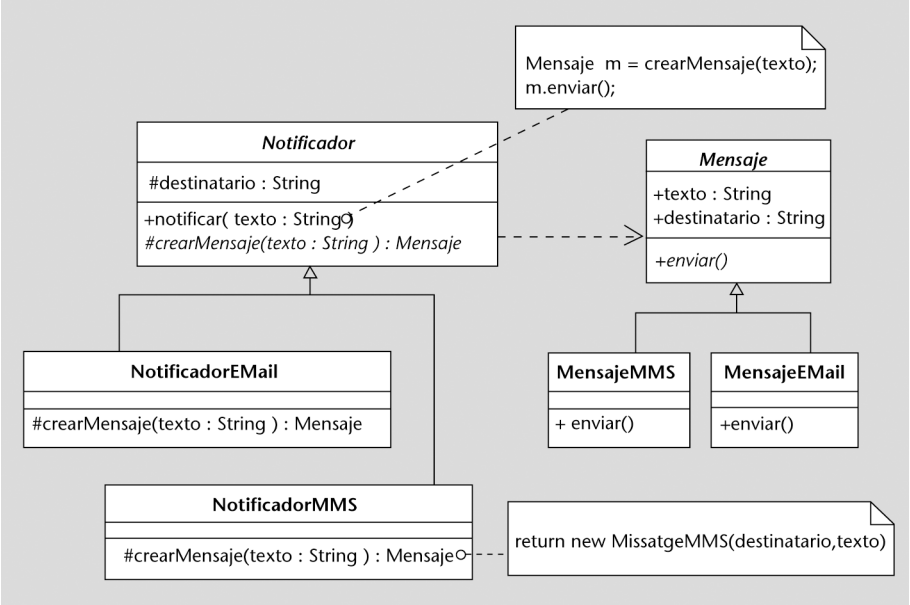
Definir, en la clase abstracta Creador (la que crea los objetos), un método "crear" encargado de la creación y que denominaremos *método factoría*. Proporcionar, a cada subclase de Creador, una implementación del método factoría que cree un tipo concreto de objeto.



En nuestro ejemplo, aplicaremos el método factoría haciendo que la decisión del tipo de mensaje que se creará sea de la subclase. De esta forma, los usuarios que trabajen con un notificador de tipo NotificadorEMail recibirán los mensajes por correo electrónico y los que utilicen un NotificadorMMS los recibirán en su teléfono móvil. Además, si queremos añadir nuevos mecanismos de notificación a nuestro sistema, sólo tenemos que añadir nuevas subclases de Notificador y de Mensaje respetando, así, el principio abierto-cerrado:

Ved también

Podéis ver más información sobre el principio de diseño abierto-cerrado en el apartado 2.3 de este módulo.



6.5. Método plantilla (*Template method*)

Contexto

Tenemos un algoritmo del cual queremos que algunas subclases puedan redefinir algunas partes.

Problema

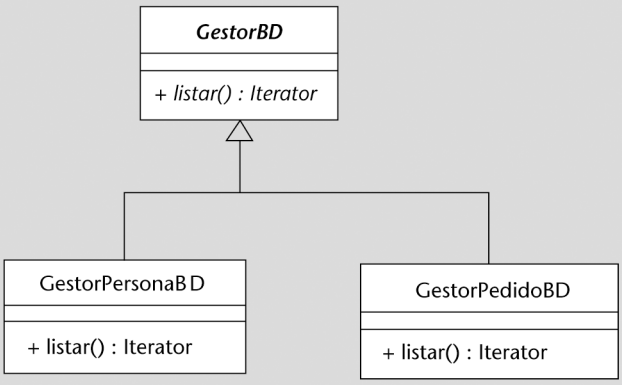
Queremos que las subclases sólo tengan que redefinir la parte que les es específica y evitar tener que rescribir las partes comunes, lo cual violaría el principio de no repetición.

Ved también

Podéis ver más información sobre el principio de no-repetición en el apartado 2.4 de este módulo.

Queremos restringir el conjunto de modificaciones posibles fijando una estructura general y unos puntos de extensión.

Por ejemplo, supongamos que en el sistema que estamos desarrollando tenemos una serie de clases que gestionan el acceso a la base de datos:



Las implementaciones de listar() en GestorPersonaBD y GestorPedidoBD siguen el mismo algoritmo general:

1) Obtener una conexión a la base de datos.

2) Construir una consulta.

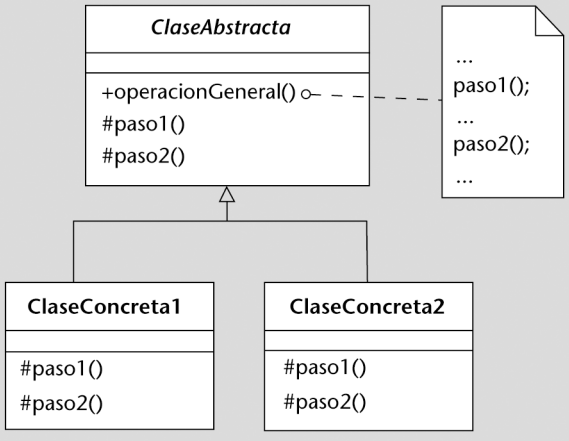
3) Ejecutar la consulta.

4) Tratar el resultado.

5) Cerrar la conexión a la base de datos.

Solución

Definir una clase abstracta que implementa el algoritmo general invocando operaciones abstractas que las subclases pueden redefinir.



En nuestro ejemplo, definiremos una operación listar() que implementará el algoritmo general. Las partes específicas de cada subclase las dejaremos a operaciones abstractas que las subclases tendrán que implementar:

```
public abstract class GestorBD {
  public Iterator listar() {
    Connexio c = obtenerConexion();
    String consulta = construirConsulta();
    Resultado r = c.ejecutar(consulta);
    List result = tratarResultado(r);
    c.cerrar();
    return result.iterator();
  }
  protected abstract String obtenerConsulta();
  protected abstract List tratarResultado(Resultado r);
  (...)
}
```

De esta manera, cada subclase tiene que definir las partes del algoritmo que le son propias y podrá redefinir otras partes del algoritmo si la superclase lo permite.

Consecuencias

- Este patrón es muy utilizado para mejorar las posibilidades de reutilización de código, ya que permite reutilizar no sólo funcionalidades enteras, sino partes de éstas.

- La superclase puede ofrecer implementaciones por defecto de algunos pasos del algoritmo. Por lo tanto, hay que distinguir entre las operaciones que pueden ser redefinidas de las que lo tienen que ser a la fuerza (y del resto de operaciones utilizadas); en muchos lenguajes, eso se puede hacer mediante operaciones abstractas.

- Si se tienen operaciones con implementaciones por defecto, la subclase puede extender esta implementación por defecto en lugar de sustituirla completamente.

6.6. Observador (*Observer*)

Contexto

Un cambio en un objeto implica cambios en otros objetos.

Problema

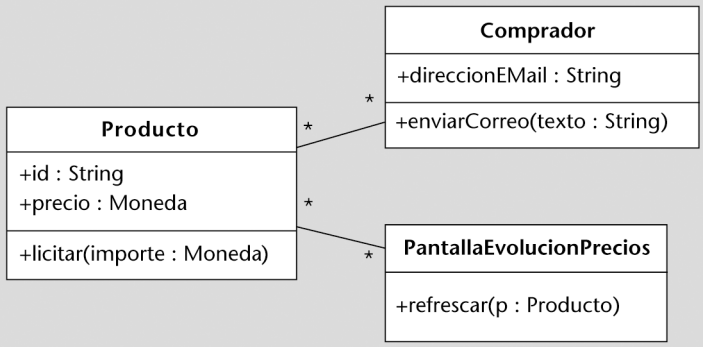
Queremos evitar el acoplamiento entre la clase del objeto en el cual se produce un cambio y la de los objetos que tienen que recibir la notificación.

Queremos que nuestro sistema esté abierto a la extensión para permitir que nuevos objetos sean notificados sin tener que modificar la clase del objeto en el cual se produce el cambio.

Ved también

Podéis ver más información sobre el principio de diseño abierto-cerrado en el apartado 2.3 de este módulo.

Por ejemplo, supongamos que estamos desarrollando un sistema de subastas electrónicas y tenemos las clases siguientes:

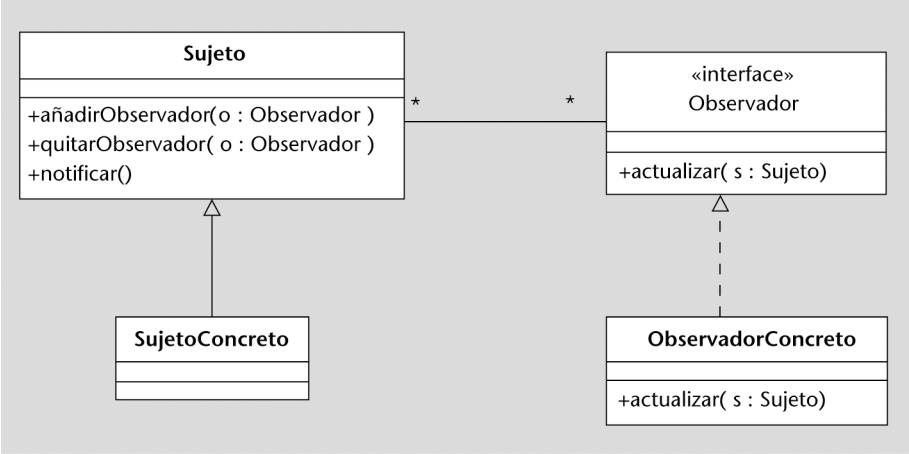


Cada vez que un usuario hace una licitación por un producto, el sistema tiene que notificar este cambio a los compradores mediante el envío de un correo electrónico y tiene que refrescar unas pantallas que muestran las últimas licitaciones que se han producido en el sistema.

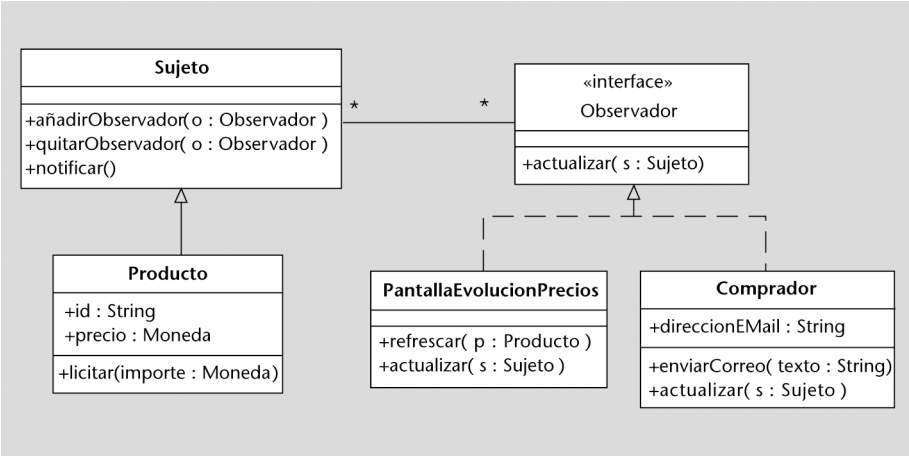
Queremos evitar que la clase Producto esté acoplada con respecto a Compradores y Pantallas.

Solución

Crear una nueva abstracción para el mecanismo de notificación y utilizar la herencia y el polimorfismo para reducir el acoplamiento.



Cuando se produce un cambio en el estado del sujeto concreto, éste invoca la operación Notificar de su superclase que se encargará de recorrer la lista de observadores e invocar, para cada uno, la operación Actualizar. En el caso de las subastas, el resultado de aplicar el patrón Observador sería el siguiente:



Así pues, la implementación de la operación licitar quedaría de la manera siguiente:

```
public void licitar(Moneda importe) {
   this.precio = importe;
   notificar();
 }
```

La implementación del sujeto será independiente de las clases concretas:

```
public void notificar() {
   foreach(o: Observador in observadores) {
     o.actualizar(this);
   }
}
```

Finalmente, cada observador implementará la operación Actualizar de manera diferente:

```
public class PantallaEvolucionPrecios {
  public void actualizar(Sujeto s) {
    refrescar((Producto)s);
  }
  (...)
}
public class Comprador {
  public void actualizar (Sujeto s) {
      String mensaje = generarMensaje((Producto)s);
      enviarCorreo(mensaje);
  }
  (...)
}
```

Consecuencias

- El acoplamiento se produce a un nivel de abstracción superior. El sujeto concreto sólo está acoplado con respecto a una interfaz muy sencilla.

- El sujeto concreto no se tiene que preocupar de cuáles ni cuántos objetos exactamente están interesados en la notificación. Eso nos permite añadir y quitar observadores dinámicamente.

- Si los observadores pueden provocar cambios en el sujeto concreto que provoquen nuevas actualizaciones, tendremos que tratar el caso de forma especial para evitar un bucle infinito de actualizaciones.

- El protocolo de notificación no nos dice qué ha cambiado exactamente.

Variaciones

En la solución propuesta, el sujeto se envía a sí mismo como parámetro de la operación Notificar. Una posible variación sería enviar otra información con la notificación (por ejemplo, los cambios concretos en el estado que se hayan producido).

Otra opción es que el sujeto no envíe ninguna información y que sea el observador quien se encargue de pedir el nuevo estado a todos los sujetos que observe. A este modelo se le denomina *modelo pull* en contraposición al *modelo push*, que hemos visto anteriormente.

6.7. Orden (*Command*)

Contexto

Queremos tratar las llamadas a operaciones como objetos, por ejemplo, para lo siguiente:

- Poder deshacer las operaciones.
- Ejecutarlas en otro entorno (en otro proceso o nodo de la red) o en otro momento.
- Parametrizar un objeto (por ejemplo, qué acción tiene que ejecutar un elemento de un menú de usuario).
- Tener un historial de cambios con el fin de poder volver a aplicarlos si cae el sistema.
- Estructurar el sistema en torno a operaciones de alto nivel (por ejemplo, tener una clase para cada operación de sistema).

Problema

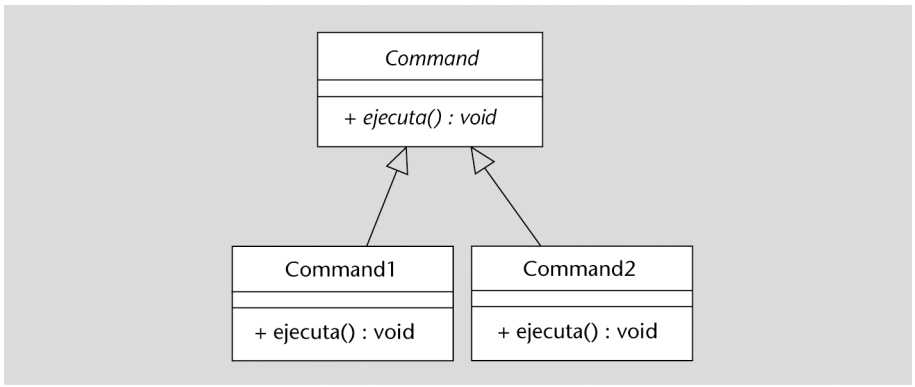
Necesitamos un mecanismo para poder tratar una operación como un objeto y nuestro lenguaje de programación no tiene esta capacidad.

Por ejemplo, queremos que el sistema que estamos desarrollando permita al usuario deshacer las acciones llevadas a cabo en el sistema (por ejemplo, si ha modificado unos datos, deshacer la modificación). Si pudiéramos tratar cada acción del usuario como un objeto, podríamos guardar una lista de las acciones efectuadas para poder deshacerlas; pero nuestro lenguaje no nos permite tratar las llamadas a operaciones como objetos.

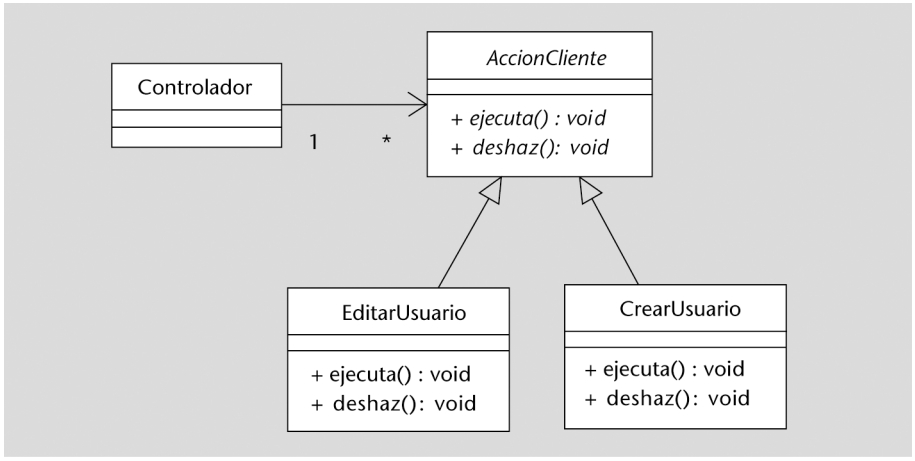
Solución

Crear una clase cuyas instancias representan invocaciones de una operación.

Podemos convertir los parámetros y el resultado en atributos de la orden. También podemos aprovechar para establecer una jerarquía de herencia entre las diferentes operaciones:



Esta solución se puede aplicar al ejemplo anterior, tratando cada acción del usuario como un objeto y convirtiendo el conjunto de acciones en una lista de objetos que recuerdan lo que han hecho y son capaces de deshacerlo:



Consecuencias

- El objeto que invoca la operación está desacoplado con respecto al que la implementa.

- Podemos manipular y extender las órdenes como cualquier otra clase del sistema (por ejemplo, podemos aplicar el patrón Método plantilla).

- Podemos crear composiciones de órdenes que permitan la ejecución conjunta de una serie de órdenes (por ejemplo, para registrar macros).

- Es fácil extender nuestro sistema añadiendo nuevas operaciones sin tener que modificar las existentes.

6.8. Otros patrones de diseño

A continuación se comentan, brevemente, otros patrones de diseño que, aunque quizás su aplicabilidad es menos general, también se han considerado importantes.

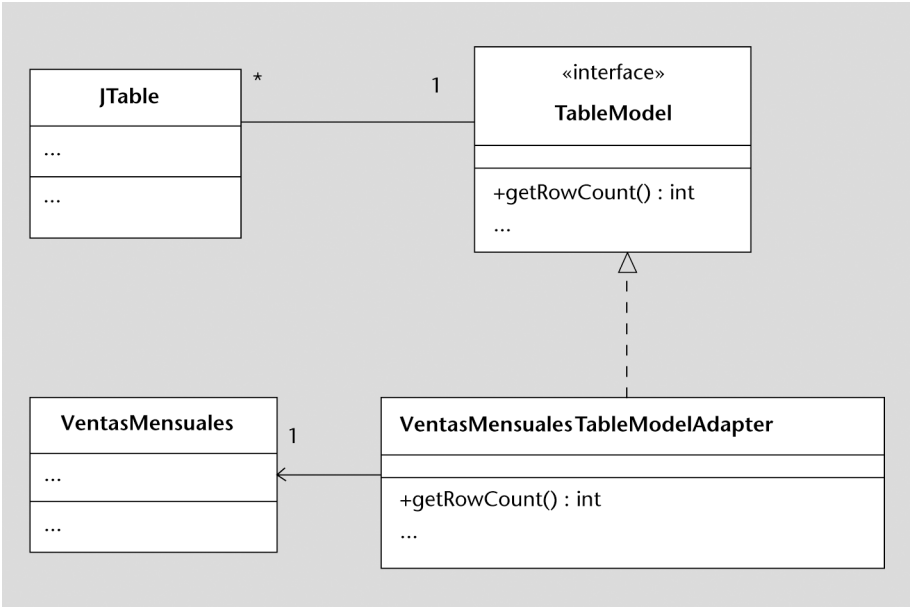
6.8.1. Adaptador (*Adapter*)

Permite utilizar una clase utilizando un conjunto de operaciones diferente al que ofrece originalmente.

Por ejemplo, en la estructura de aplicaciones gráficas Swing de Java, el componente JTable utiliza la interfaz TableModel para obtener los datos que tiene que visualizar. Si queremos mostrar los datos de una estructura de datos propia, podemos escribir un adaptador para poder ofrecer al componente JTable la interfaz que espera sin tener que modificar nuestra clase:

Ved también

Podéis encontrar una explicación detallada con ejemplos sobre adaptadores en el apartado 1.6.1 del módulo "Introducción a los patrones".



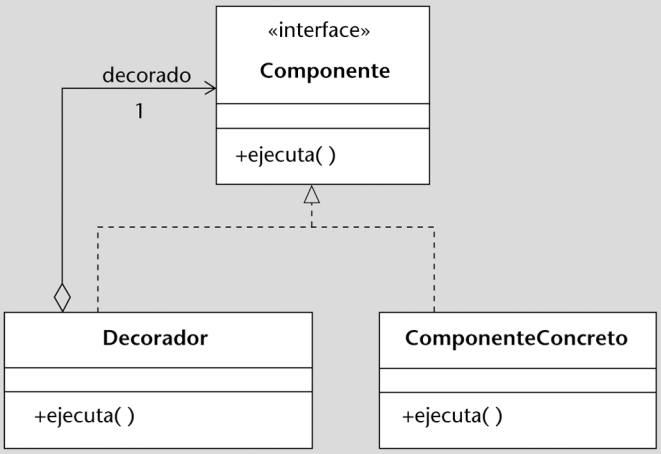
6.8.2. Decorador (*Decorator*)

Permite añadir responsabilidades a un objeto dinámicamente.

El Decorador tiene la misma interfaz que el objeto que decora, implementa las responsabilidades adicionales y delega el resto de responsabilidades en el objeto decorado:

Ved también

El Decorador está muy relacionado con el Representante, que podéis encontrar descrito en el apartado 6.8.6 de este módulo.



Por ejemplo, supongamos que estamos desarrollando un sistema y queremos hacer un estudio de rendimiento. Queremos poder añadir a las Órdenes que implementan nuestras operaciones de negocio la responsabilidad de controlar cuánto tiempo tardan en ejecutarse. Como queremos activar y desactivar esta capacidad en tiempo de ejecución, en vez de modificar las clases que tenemos añadimos una nueva clase: DecoradorRendimiento.

Ved también

Podéis ver más información sobre el patrón Orden en el apartado 6.7 de este módulo.

```
public class DecoradorRendimiento implements Orden {
  Orden decorado;
  long tiempoEmpleado;
  (...)
  public void ejecuta() {
    long inicio = System.currentTimeMillis();
    decorado.ejecuta();
    tiempoEmpleado=System.currentTimeMillis() - inicio;
  }
  (...)
}
```

6.8.3. Estrategia (*Strategy*)

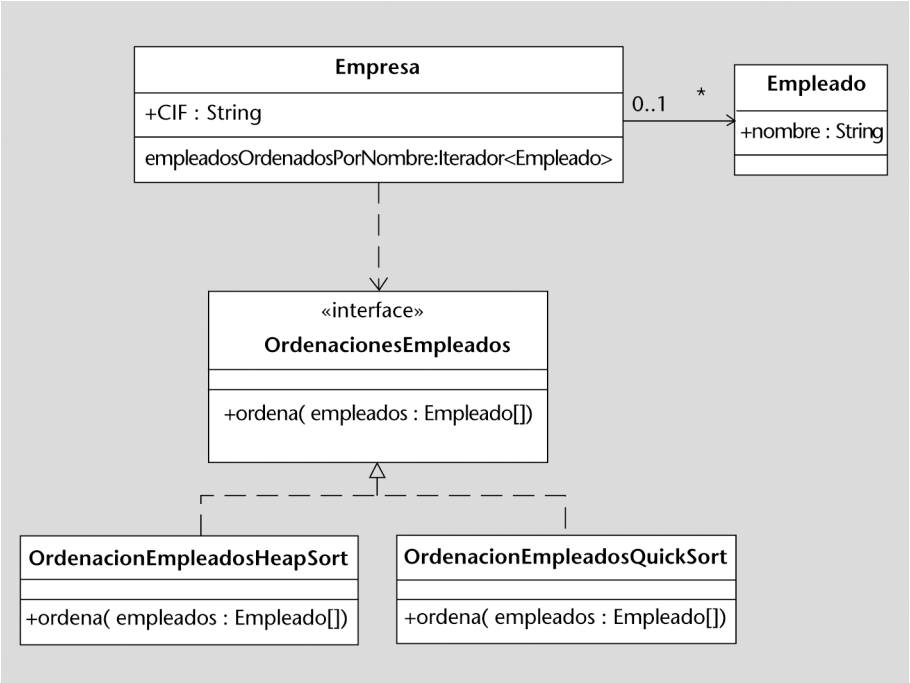
Permite definir una familia de algoritmos y hacerlos intercambiables los unos con los otros. De esta manera, permite escoger uno dinámicamente.

Estrategia y Método plantilla

El patrón de diseño Estrategia nos permite solucionar problemas muy similares a los problemas que soluciona el patrón Método plantilla que podéis encontrar en el apartado 6.5 de este módulo. A menudo podremos utilizar uno u otro indistintamente. La principal diferencia radica en el hecho de que el patrón Estrategia utiliza delegación en lugar de herencia.

La Estrategia permite separar un algoritmo general de los detalles concretos de cada posible implementación, facilitando, pues, la reutilización del algoritmo general de manera independiente de los detalles. Además, permite reutilizar las implementaciones de los detalles para más de un algoritmo general dado que no utiliza herencia sino delegación.

Supongamos que en el sistema que estamos desarrollando tenemos que ordenar una lista y queremos poder escoger qué algoritmo de ordenación utilizamos:



En el diseño resultante, la operación empleadosOrdenadosPorNombre de una instancia de empresa se comportará de una manera u otra dependiendo de la implementación de OrdenacionEmpleados que utilice. Además, este comportamiento podría cambiar dinámicamente si sustituimos la instancia de OrdenacionEmpleado utilizada dentro de la clase Empresa por una de otra subclase.

6.8.4. Instancia única (*Singleton*)

Permite asegurar que, de una determinada clase, sólo hay una instancia en todo el sistema.

La implementación de una clase de Instancia única no tendrá operaciones de construcción accesibles desde fuera de la misma clase y, por lo tanto, sólo podrá ser instanciada por ella misma.

```
public class Singleton {
  private static Singleton instance = null;
private Singleton() {}
public static Singleton getInstance() {
    if (instance == null) {
     instance = new Singleton();
    }
    return instance;
```

```
    }
  }
```

Supongamos que queremos que todas las clases que accedan a la base de datos en nuestra aplicación lo hagan por medio de la misma instancia, de tal manera que podamos controlar fácilmente el número de conexiones establecidas contra la base de datos. Para conseguirlo, tenemos que diseñar la clase de acceso a la base de datos como una clase de instancia única.

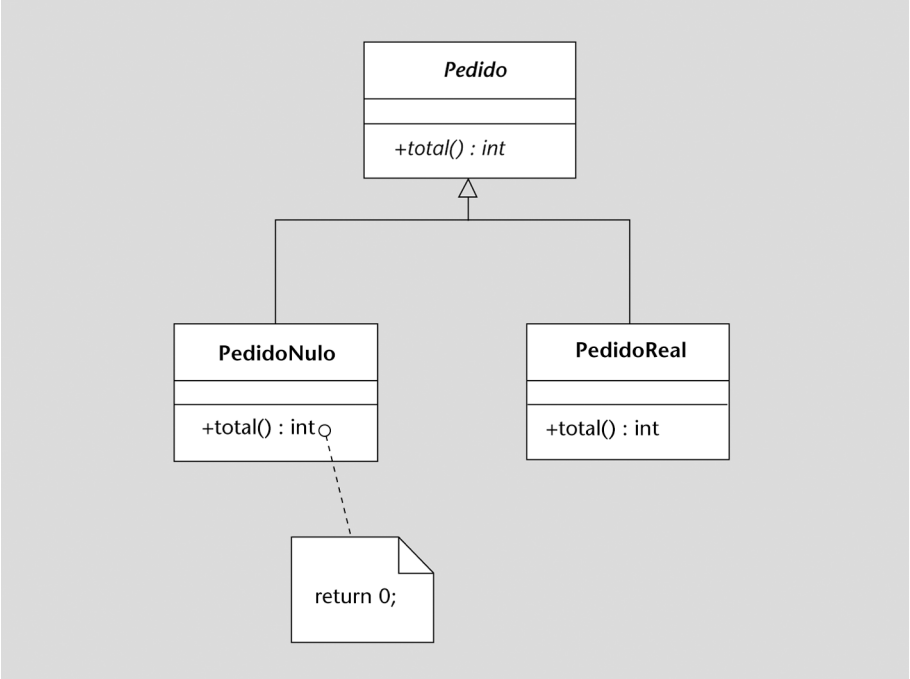
6.8.5. Objeto nulo (*Null object*)

Permite tratar el caso del valor nulo como si fuera una instancia válida.

Por ejemplo, consideramos este fragmento de código:

```
int total;
Pedido c = DB.cargaPedido(W2132");
if (c != null) {
  total = c.total();
} else {
  total = 0;
}
```

El tratamiento especial que tenemos que dar al objeto nulo aumenta considerablemente la complejidad del código. Por esta razón nos planteamos tratar de manera uniforme los dos casos. Para hacerlo, el patrón Objeto Nulo propone crear una subclase de Pedido que representa el caso especial en el que un Pedido está vacío:



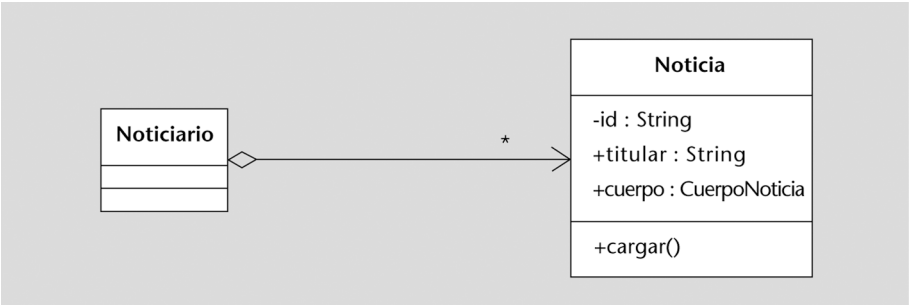
Nuestro fragmento de código quedaría de la manera siguiente:

```
int total;
Pedido c = DB.cargaPedido(W2132");
total = c.total();
```

6.8.6. Representante (*Proxy*)

Permite controlar el acceso a un objeto desde otro de manera transparente.

Supongamos, por ejemplo, que estamos desarrollando un sistema que gestiona noticias:



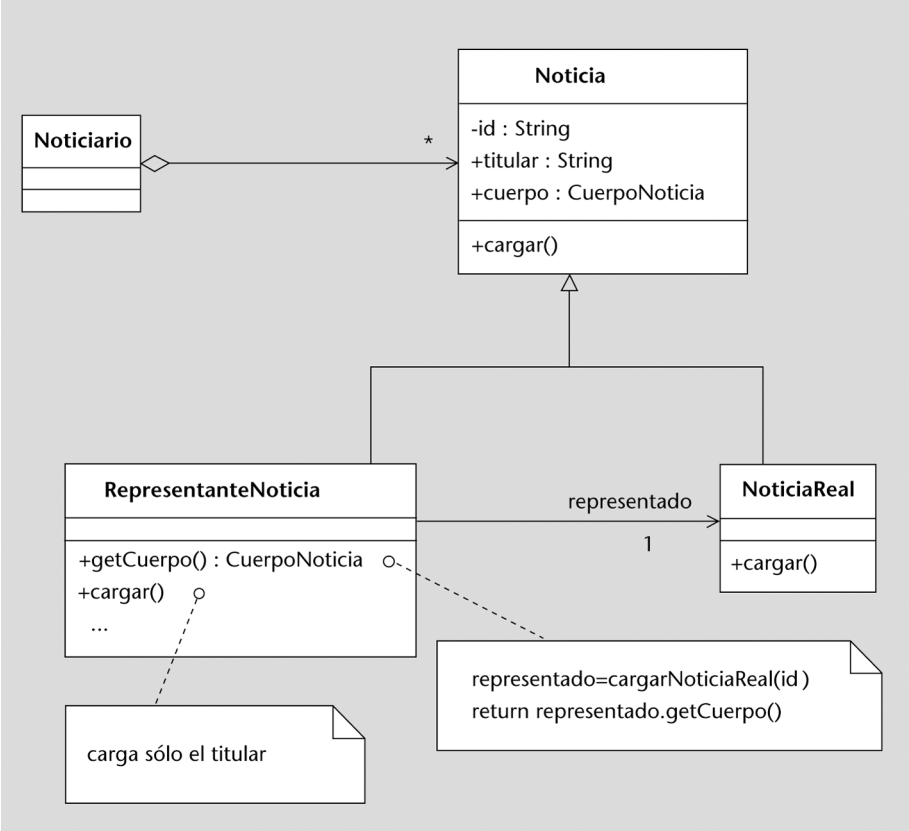
Las noticias se almacenarán en un repositorio y supondremos que cargar su cuerpo es una operación costosa, ya que puede implicar cargar grandes cantidades de texto y, seguramente, algunas imágenes. El noticiario, normalmente, mostrará los titulares de las noticias y, cuando el usuario seleccione una, mostrará su cuerpo.

Para mejorar el rendimiento del sistema, queremos evitar que el cuerpo de las noticias se cargue hasta que sea necesario. Pero querríamos hacerlo de manera transparente para el Noticiario y para la Noticia que ya tenemos diseñados.

El patrón Representante propone añadir un objeto (el Representante) que controlará el acceso a la noticia. Cuando este representante detecte que se quiere acceder al cuerpo de la noticia (y sólo en este caso), lo cargará:

Ejemplo

Hemos supuesto que el acceso al atributo público cuerpo se implementa con un método getCuerpo en nuestro lenguaje de programación.



Queda por resolver el problema de sustituir el objeto original por su representante, condición necesaria para que la aplicación del patrón sea realmente transparente a la clase Cliente. La manera de solucionarlo será diferente en cada caso y queda fuera del ámbito de este patrón.

El Representante puede tener otros usos interesantes como un representante caché[5] (que nos servirá como memoria caché ahorrando invocaciones a la clase representada), un representante remoto (que nos servirá para invocar ope-

(5)En inglés, *cache*.

raciones a un objeto que se encuentra en otro nodo de la red), un representante de seguridad (que nos permitirá bloquear las peticiones que no cumplan los requisitos de seguridad), etc.

Este patrón, en su estructura, es muy parecido al patrón Decorador. La principal diferencia, sin embargo, radica en su finalidad: un decorador añade responsabilidades a un objeto modificando el comportamiento, mientras que un representante controla el acceso.

Ved también

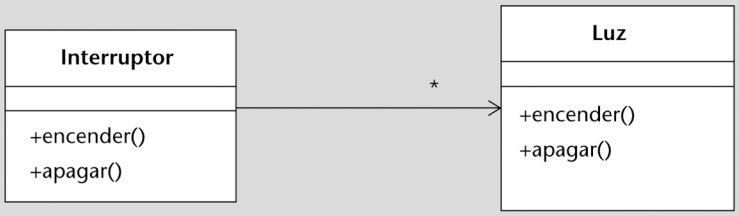
Podéis ver más información sobre el patrón Decorador en el apartado 6.8.2 de este módulo didáctico.

6.8.7. Servidor abstracto (*Abstract server*)

Permite desacoplar una clase cliente respecto de una clase que utiliza y que denominaremos *servidor*.

Para hacerlo, introduce una abstracción que representa el uso que el cliente hace del servidor.

Por ejemplo, supongamos que tenemos las clases siguientes:

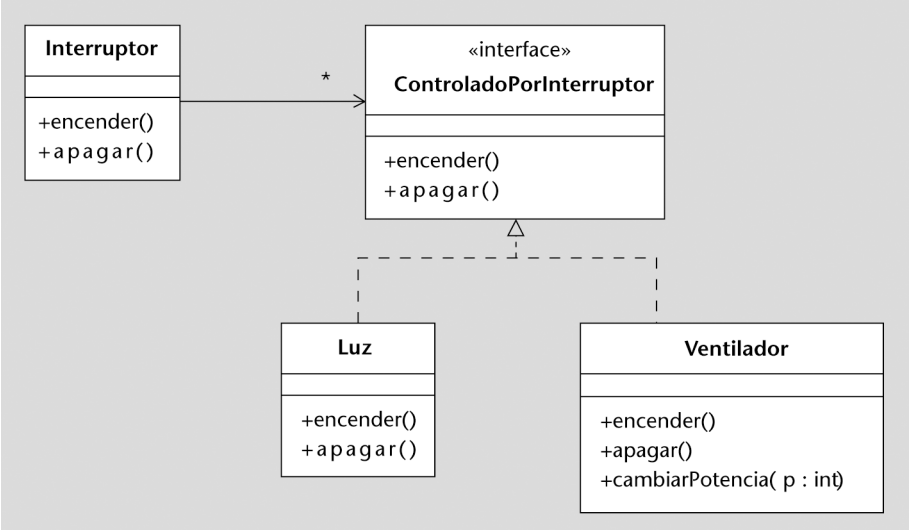


Este diseño acopla la clase Interruptor con respecto a la clase Luz. Además, se incumple el principio abierto-cerrado: ¿Qué pasa si queremos un interruptor para un ventilador?

Ved también

Podéis ver más información sobre el principio de diseño abierto-cerrado en el apartado 2.3 de este módulo didáctico.

El patrón Servidor abstracto nos dice que introduzcamos una abstracción entre Interruptor y Luz que represente el uso que hace el interruptor de luces, ventiladores y cualquier otra clase que se quiera controlar más adelante:



Hay que destacar el detalle del nombre de la interfaz. La interfaz con las operaciones de Luz se llama *ControladoPorInterruptor* y no *Luz*. Eso es debido al hecho de que las operaciones de la interfaz son las que sabe controlar el Interruptor y no necesariamente todas las de Luz. Por eso decimos que la interfaz pertenece a Interruptor y no a Luz. Con este diseño, cualquier aparato que implemente ControladoPorInterruptor podrá ser controlado por una instancia de Interruptor.

Si los objetos que tienen que implementar la nueva interfaz ya existen, seguramente tendremos que adaptar la interfaz actual a la que introduce el Servidor abstracto. Para hacerlo, nos podemos basar en el patrón Adaptador.

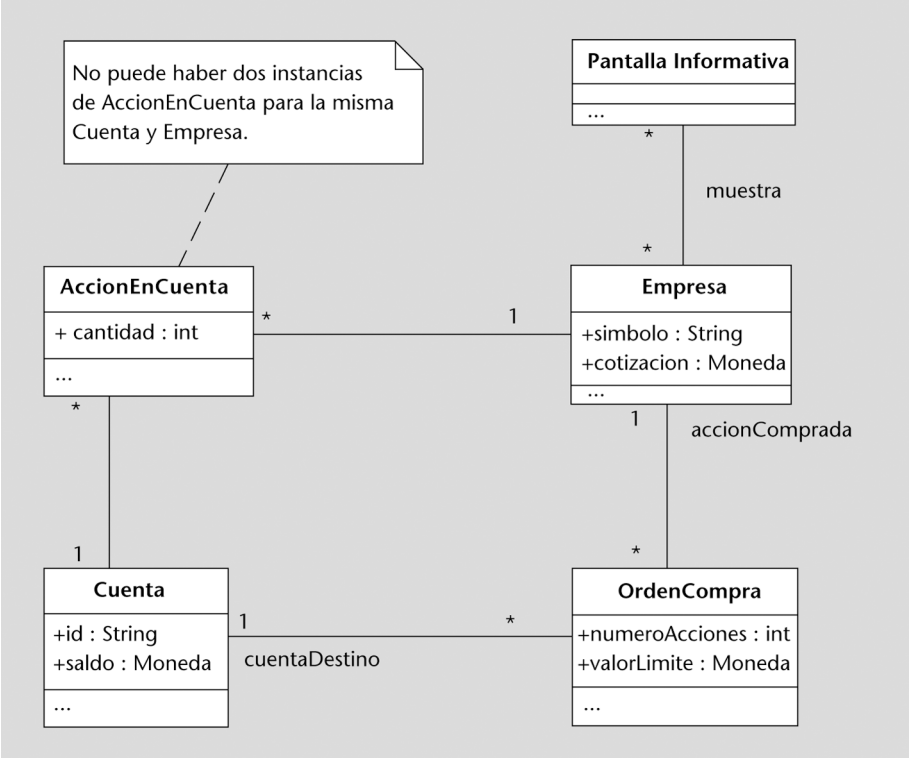
Ved también

Podéis ver más información sobre el patrón Adaptador en el apartado 6.8.1 de este módulo.

Ejercicios de autoevaluación

El sistema que estamos desarrollando gestiona información sobre cotización de acciones, órdenes de compra, y también unas pantallas que muestran la cotización de las diferentes empresas.

Los usuarios pueden programar órdenes de compra de manera que cuando la cotización de las acciones de una empresa baja por debajo de cierto valor límite (indicado en el orden), el sistema compre automáticamente el número de acciones que se ha programado en la orden y las añada a la cuenta del usuario:



Nota

La clase AccionEnCuenta es el resultado de normalizar la clase asociativa de la asociación "enCartera" entre Cuenta y Empresa.

1) Queremos diseñar una operación que denominaremos ejecutarOrdenCompra que, dada una orden de compra, calcula el valor de las acciones compradas, verifica el saldo de la cuenta y, si el saldo es suficiente, añade la cantidad de acciones compradas a la cartera de la cuenta y descuenta el precio del saldo. Aplica los patrones de asignación de responsabilidades para asignar las diversas responsabilidades asociadas a esta funcionalidad.

2) En realidad, la ejecución de la operación de la tarea anterior se debería hacer automáticamente cuando se produzca un cambio en la cotización de las acciones de una empresa que haga que alguna orden de compra se convierta en ejecutable. Además, queremos que las pantallas informativas se actualicen automáticamente cada vez que se modifique el valor de cotización de cualquiera de las acciones mostradas en la pantalla. Haced un diseño para este comportamiento utilizando un patrón de diseño que os ayude a evitar el acoplamiento de la clase Empresa a pantallas y órdenes de compra.

Solucionario

1) Para esta operación haremos la asignación de responsabilidades siguiente:

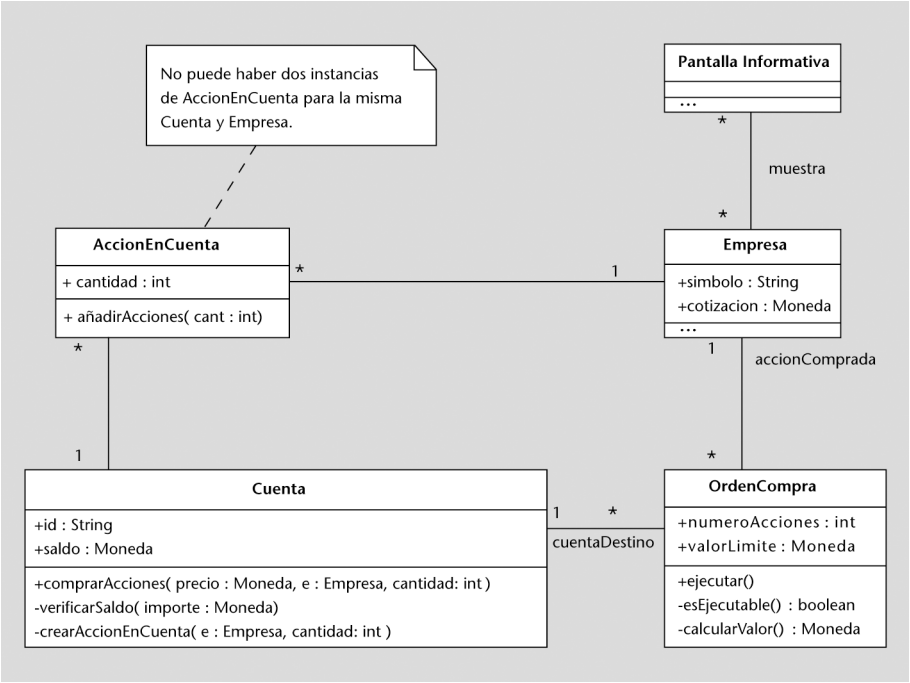
Verificar si se tiene que ejecutar la orden de Compra: para hacerlo, habrá que consultar el valor de cotización (es Empresa quien lo conoce) y compararlo con el valor límite de la orden (es OrdenCompra quien lo conoce). En este caso, por cohesión asignaremos esta responsabilidad a la Orden de compra de tal manera que sea la única que conozca cómo se decide si una orden de compra es ejecutable.

Si la orden es ejecutable, habrá que calcular el importe de compra de las acciones, verificar el saldo de la cuenta y, si hay suficiente saldo, añadir el número de acciones indicado en la orden de compra a la cuenta de destino. En este caso, la Orden es el experto para calcular el importe de compra, mientras que la Cuenta es quien conoce el saldo y las acciones.

El experto que sabe cuántas acciones de una determinada empresa tiene una cuenta es la clase AccionEnCuenta.

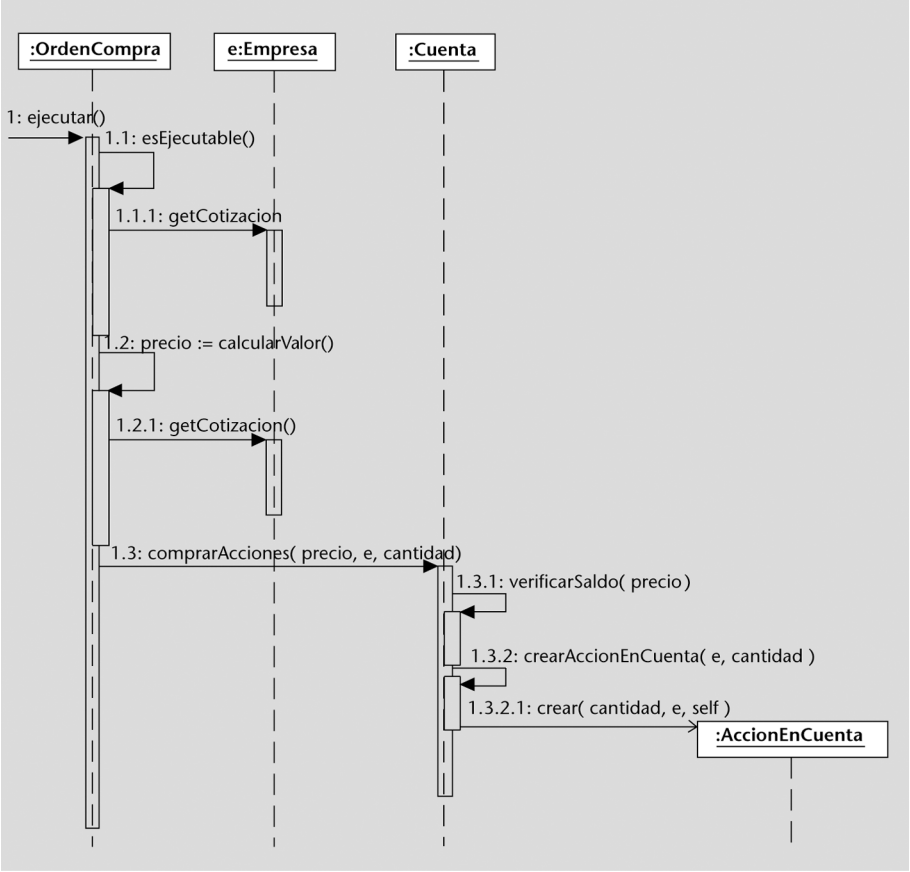
Puede pasar, sin embargo, que la Cuenta no tenga previamente ninguna Acción de la misma empresa y que, por lo tanto, haya que crear una nueva instancia de AccionEnCuenta. En este caso, según el patrón Creador, será la Cuenta quien se encargue de crear esta instancia, ya que la instancia de Cuenta contiene sus instancias de AccionesEnCuenta.

Siguiendo este criterio, pues, obtenemos el diagrama siguiente:



Si esta compra se quisiera efectuar como resultado de un evento generado por el usuario, además habría una clase controladora que se encargaría de recibir este acontecimiento y de invocar la operación creada de la clase OrdenCompra.

En el escenario donde la compra se acaba efectuando (el valor de cotización está por debajo del valor límite y hay suficiente saldo) y donde la cuenta no tenía ninguna acción de la empresa previamente, el diagrama de secuencias resultante de la asignación de responsabilidades hecha anteriormente es el siguiente:

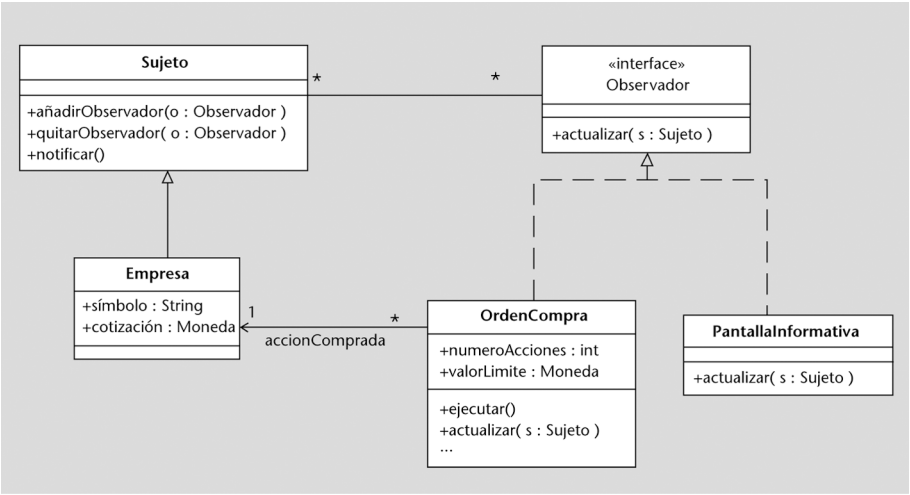


2) El patrón Observador nos permite notificar los cambios que se producen en una empresa a aquellos objetos que estén interesados (en nuestro caso, pantallas y órdenes de compra) con un mínimo acoplamiento.

Nota

La Empresa llamará la operación Notificar cuando su atributo cotizacion sea modificado. La operación Actualizar de OrdenCompra invocará la operación Ejecutar ya diseñada.

La estructura de la solución será la siguiente:



La asociación "muestra" entre PantallaInformativa y Empresa ya no es necesaria, dado que sólo se utilizaba a efectos de notificación (que ahora resuelve el patrón). De manera parecida, se ha suprimido la navegabilidad de Empresa hacia OrdenCompra, que ya no es necesaria (aunque se ha mantenido el otro sentido de navegación porque es necesario más allá de la notificación). De esta manera, hemos reducido el acoplamiento entre las clases.

Bibliografía

[FOW] Fowler, M. (1997). *Analysis Patterns: Reusable Object Models*. Massachusetts: Addison Wesley Professional.

[GOF] Gamma E.; Helm R.; Johnson R.; Vlissides J. (1995). *Design Patterns: Elements of Reusable Object-Oriented Software*. Massachusetts: Addison Wesley Professional.

[LAR] Larman, C. (2003). UML Y PATRONES. *Una introducción al análisis y diseño orientado a objetos y al proceso unificado* (2.ª ed.). Madrid: Prentice Hall.

[MAR] Martin, R. C. (2003). *Agile Software Development: Principles, Patterns, and Practices*. Upper Saddle River, Nueva Jersey: Prentice Hall.

Bibliografía adicional

[BECK] Beck, K. (2002). *Test Driven Development: By example.* Massachusetts: Addison Wesley Professional.

[FOWDI] Fowler, M. (2004). http://www.martinfowler.com/articles/injection.html

[HUNT] Hunt, A.; Thomas, D. (2000). *The Pragmatic Programmer.* Massachusetts: Addison Wesley.

[MEY] Meyer, B. (1999). *Construcción de software orientado a objetos* (2. ed.). Madrid: Prentice Hall.

[POSA] Buschmann, F.; Meunier, R.; Rohnert, H.; Sommerlad, P.; Stal, M. (1996). *Pattern-Oriented Software Architecture: A System Of Patterns*. West Sussex, Inglaterra: John Wiley & Sons Ltd.