Caso práctico de aplicación de patrones

Jordi Pradel i Miquel
José Antonio Raya Martos

PID_00165659

Universitat Oberta de Catalunya
www.uoc.edu

Índice

Introducción

Este módulo muestra un caso práctico de análisis y diseño orientados a objetos con patrones. Partiremos de un análisis previo realizado sin utilizar patrones e iremos aplicando los patrones a lo largo de todo el ciclo de vida del desarrollo. De esta manera podremos ver cómo el uso de los patrones nos facilita el desarrollo de las aplicaciones.

Primero refinaremos el análisis utilizando los patrones de análisis. Después definiremos la arquitectura del sistema mediante patrones de arquitectura. Finalmente, diseñaremos con detalle los diferentes subsistemas definidos en la arquitectura utilizando los patrones de asignación de responsabilidades y de diseño. Todos estos patrones se pueden encontrar explicados detalladamente en el módulo "Catálogo de patrones".

También se mostrarán algunos ejemplos de uso de *frameworks* para el diseño e implementación de algunos de los subsistemas detectados.

Objetivos

El objetivo principal del módulo es servir como ejemplo de un proceso de análisis y diseño orientados a objetos mediante patrones. De esta manera, se pueden encontrar en este módulo ejemplos adicionales de aplicación de los patrones del catálogo, así como una visión general del proceso de análisis y diseño y sobre el punto donde puede ser interesante utilizar los patrones. Los objetivos concretos que tiene que alcanzar el estudiante con este módulo didáctico son los siguientes:

1. Entender el concepto de patrón y saber aplicarlo a las diversas etapas del ciclo de vida del desarrollo.

2. Conocer un ejemplo de un método completo de desarrollo de aplicaciones orientado a objetos.

3. Conocer las ventajas del uso de *frameworks* para el desarrollo del software.

4. Comprender cómo los patrones y el método propuesto nos ayudan a obtener un diseño de calidad que sigue unos principios de diseño sólidos.

1. Requisitos y análisis preliminar

1.1. Requisitos

eMusica.biz es una empresa de venta de música por Internet que ha contratado nuestros servicios para hacer el desarrollo de una nueva versión de su tienda virtual. Los clientes de eMusica.biz tendrán la opción de comprar canciones (que se bajarán de los servidores de la empresa), valorar una canción (darle una puntuación con el fin de orientar al resto de clientes en su compra de música) y consultar los pedidos que hayan hecho con anterioridad. Por parte de eMusica están los administradores del sistema que se encargan de gestionar el catálogo de productos y los precios de éstos, y también los pedidos de los clientes.

Tenemos una versión casi completa del análisis. Lo único que tenemos que hacer es refinarlo y a partir de él haremos el diseño y la implementación del nuevo sistema.

1.2. Modelo de casos de uso

Nuestro modelo consta de dos actores (Cliente y Administrador) claramente diferenciados. Con respecto a los casos de uso, el cliente es el actor principal en cuatro de ellos; a saber: "comprar productos", "valorar producto", "registrarse" y "consultar pedidos".

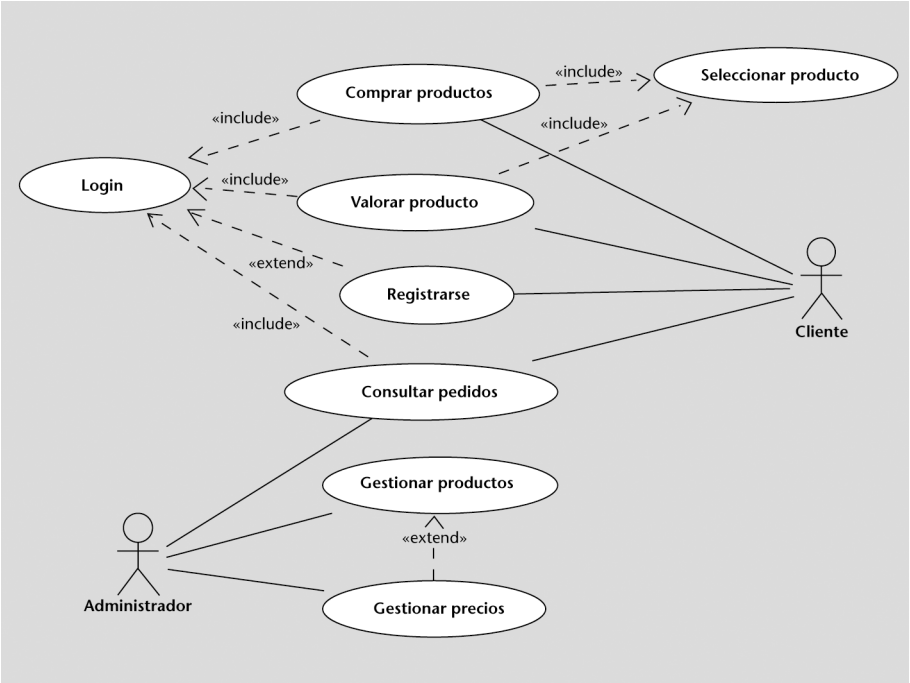
El administrador comparte con el cliente el caso de uso "Consultar pedidos", además de tener dos casos de uso propios: "Gestionar productos" y "Gestionar precios".

Tres de los casos de uso de cliente incluyen el caso de uso "Login" porque en algún punto de su secuencia de acontecimientos todos tienen que hacer un proceso de identificación del usuario. Para no documentar las tres veces este proceso, se ha creado un caso de uso que lo describa y se ha incluido desde los tres lugares en los que aparece el proceso. Algo similar sucede con los casos de uso "Comprar productos" y "Valorar producto": en los dos casos hay que seleccionar un producto, funcionalidad que se ha separado en un caso de uso incluido "Seleccionar producto". Por lo tanto, "Login" y "Seleccionar producto" son casos de uso que no tienen sentido por sí mismos ni tienen ningún actor que los inicie.

Por otra parte, entre los casos de uso del administrador, "Gestionar precios" extiende "Gestionar productos". Esta relación indica que en algún punto de la secuencia de pasos de "Gestionar productos", si se cumple cierta condición, puede ser que se haga también una gestión de precios. En este caso, sin em-

bargo, "Gestionar precios" es un caso de uso que puede ser iniciado como caso de uso independiente por el actor Administrador, aunque también pueda ser iniciado como extensión de "Gestionar productos". De manera análoga, "Registrarse" extiende el caso de uso "Login".

El diagrama de casos de uso que refleja este análisis es el siguiente:



Pero un modelo de casos de uso no es completo sólo con el diagrama. Éste sólo muestra los actores y casos de uso del sistema y las relaciones entre ellos. Para obtener un análisis completo de casos de uso, hace falta una especificación textual de cada caso de uso. A partir de la especificación textual, también añadiremos una recopilación de cuál es la información que introduce el usuario al sistema y cuál le muestra el sistema. Nos servirá para después definir la interfaz gráfica más fácilmente.

1.2.1. Caso de uso "Comprar productos"

Éste es el caso de uso que analizaremos con más detalle. Demos, pues, una especificación detallada del mismo.

Nombre: Comprar productos

Resumen de la funcionalidad: permite al usuario visualizar los productos, hacer un pedido y pagarlo

Flujo de acontecimientos principal:

1) El usuario selecciona un producto (caso de uso incluido "Seleccionar producto").

2) El usuario escoge añadir el producto a la cesta de la compra.
3) El sistema añade el producto a la cesta de la compra.
4) El sistema muestra la pantalla "Cesta".
5) El usuario selecciona pagar el pedido.
6) Si el usuario no ha sido identificado previamente, se identifica ante el sistema (caso de uso incluido "Login").
7) El sistema muestra el formulario "Datos de pago".
8) El usuario introduce los datos solicitados. Todos los datos son obligatorios.
9) El sistema muestra la pantalla "Confirmación de pago".
10) El usuario confirma su compra.
11) El sistema registra la compra y envía por correo electrónico al usuario los productos comprados.
12) El usuario selecciona "Continuar".
13) El caso de uso finaliza.

Flujo de acontecimientos alternativos:

5a) El usuario selecciona eliminar una línea de la cesta de la compra.
5a.1) El sistema elimina la línea de la cesta.
5a.2) Ir al punto 4.
5b) El usuario selecciona la opción "Continuar comprando".
5b.1) Ir al punto 1.

8a) El usuario cancela el pago.
8a.1) Ir al punto 4.
10a) El usuario cancela el pago.
10a.1) Ir al punto 4.

Pantallas:

- Cesta
  Datos mostrados:
  - Para cada producto añadido: Nombre del producto, Precio
  - Precio total del pedido

- Datos de pago
  Datos mostrados:
  - Los mismos que en la pantalla "Cesta"

  Datos solicitados:
  - Número de tarjeta de crédito
  - Tipo de tarjeta (Visa, Mastercard, etc.)
  - Fecha de caducidad (mes y año)

- Confirmación de pago
  Datos mostrados:
  - Los mismos que en la pantalla "Cesta"

  - Los mismos datos que solicita la pantalla "Datos de pago"

- Pedido realizado
  Datos mostrados:
  - Mensaje de pedido realizado

1.2.2. Caso de uso "**Seleccionar producto**"

Como "Seleccionar producto" está incluido en "Comprar productos", para especificar completamente "Comprar productos" en detalle, hace falta también una especificación detallada de "Seleccionar producto".

Nombre: Seleccionar producto

Resumen de la funcionalidad: permite al usuario visualizar los productos y seleccionar uno.

Flujo de acontecimientos principal:

1) El sistema muestra la pantalla "Lista de categorías".
2) El usuario repite los pasos 3 y 4 hasta que decida seleccionar un producto.
3) El usuario selecciona una categoría.
4) El sistema muestra la pantalla "Detalle de categoría".
5) El usuario selecciona un producto.

Pantallas:

- Lista de categorías
  Datos mostrados:
  - Árbol de categorías de producto (jerarquía entera de categorías). Para cada categoría, mostrar: Nombre, Imagen

- Detalle de categoría
  Datos mostrados:
  - El árbol de categorías (como la pantalla "Lista de categorías")
  - El nombre de la categoría seleccionada
  - Los productos de la categoría ordenados por número de unidades vendidas. Para cada producto se muestran: Nombre, Descripción, Imagen, Valoración media, Precio.

1.2.3. Caso de uso "Login"

Como pasaba con "Seleccionar producto", una especificación detallada de "Comprar producto" tiene que ir acompañada de la especificación detallada del caso de uso incluido "Login".

Nombre: Login

Resumen de la funcionalidad: permite al usuario identificarse ante el sistema.

Flujo de acontecimientos principal:

1) El sistema muestra la pantalla "Login".
2) El usuario introduce los datos de identificación (todos son obligatorios).
3) El sistema comprueba que los datos de identificación son correctos y registra la identificación del usuario en su sesión.

Flujo de acontecimientos alternativos:

2a) El usuario selecciona "Cancelar".
2a.1) El caso de uso finaliza.

3a) Los datos proporcionados por el usuario no son correctos.
3a.1) El sistema muestra la pantalla "Login" con un mensaje de error adecuado.
3a.2) Ir al punto 2.

Pantallas:

- Login
  Datos solicitados:
  - Nombre de usuario
  - Contraseña

1.2.4. Caso de uso "Registrarse"

Nombre: Registrarse

Resumen de la funcionalidad: permite a un usuario registrarse en el sistema para poder utilizar otros casos de uso.

Casos de uso relacionados

- Extiende el caso de uso "Login" el paso 2.

Flujo de acontecimientos principal:

1) El sistema muestra al usuario el formulario "Datos de registro de usuario".
2) El usuario introduce los datos solicitados (todos son obligatorios excepto el teléfono).
3) El sistema comprueba que los datos son correctos y registra al usuario.

Flujo de acontecimientos alternativos:

2a) El usuario selecciona "Cancelar".
2a.1) El caso de uso finaliza.
3a) Los datos introducidos por el usuario no son correctos por alguno de estos errores:

- La contraseña y su confirmación no coinciden.
- El nombre de usuario es de un usuario existente.

3a.1) El sistema muestra el formulario "Datos de registro de usuario" con un mensaje de error indicativo.
3a.2) Ir al paso 2.

Pantallas:

- Datos de registro de usuario
  Datos solicitados:
  - Nombre
  - Apellidos
  - Teléfono
  - Dirección de correo electrónico
  - Nombre de usuario
  - Contraseña
  - Confirmación de contraseña

1.2.5. Otros casos de uso

En el resto de casos de uso, aunque también nos hace falta una especificación textual, hemos optado por hacerla menos detallada.

Caso de uso "Valorar producto"

Resumen de la funcionalidad: permite al usuario valorar con una nota entre 0 y 10 un producto que haya comprado en una sesión anterior.

Flujo de acontecimientos principal: el usuario selecciona un producto (caso de uso incluido "Seleccionar producto") y lo valora entre 0 y 10.

Flujo de acontecimientos alternativos: si el usuario no ha sido identificado previamente, se identifica ante el sistema (caso de uso incluido "Login"). En caso de error del usuario, se le piden de nuevo los datos.

Pantallas: Valoración de producto

Caso de uso "Consultar pedidos"

Resumen de la funcionalidad: permite a administradores y clientes consultar los pedidos.

Flujo de acontecimientos principal: el usuario pide consultar sus pedidos y se le muestran los datos de los pedidos que ha realizado ordenados cronológicamente. En el caso del administrador, se le mostrarán los pedidos de todos los usuarios.

Flujo de acontecimientos alternativos: en caso de que el iniciador sea un cliente, si no ha sido identificado previamente, se identifica al sistema (caso de uso incluido "Login"). Si se selecciona un pedido, el sistema muestra el detalle.

Pantallas: Listado de pedidos, Detalle de pedido

Caso de uso "Gestionar productos"

Resumen de la funcionalidad: permite al administrador gestionar qué productos y categorías hay en el sistema.

Flujo de acontecimientos principal: el administrador puede crear, modificar o eliminar categorías y productos. Cada producto tiene que pertenecer a una o más categorías. Las categorías se organizan jerárquicamente: cada categoría tiene que tener una supercategoría, excepto la categoría raíz.

Pantallas: Gestión de categorías, Creación de categoría, Modificación de categoría, Eliminación de categoría, Creación de producto, Modificación de producto, Eliminación de producto

Caso de uso "Gestionar precios"

Resumen de la funcionalidad: permite al administrador establecer los precios de los productos.

Casos de uso relacionados: extiende el caso de uso "Gestionar productos" en cualquier punto donde el administrador esté visualizando un producto.

Flujo de acontecimientos principal: el administrador establece o modifica el precio de un producto por un intervalo de tiempo.

Pantallas: Gestión de precios, Establecimiento precio producto

1.3. Diagrama estático de análisis

Una vez elaborado el modelo de casos de uso, podemos pasar a elaborar el diagrama estático de análisis, que también podemos denominar *modelo conceptual* ya que representa los conceptos que el sistema realiza. Este diagrama tiene que reflejar lo que nuestro sistema conoce del mundo real desde un punto de vista de análisis y, por lo tanto, sin tener en cuenta, en ningún momento, la tecnología que se utilizará.

Para elaborar este diagrama, utilizaremos un diagrama de clases UML en el que aparecerá una clase para cada concepto del mundo real que nuestro sistema manipule y las relaciones entre éstos (de asociación y de generalización/especialización).

1.3.1. Productos y categorías

Para empezar hemos identificado los productos que se venderán en la tienda. Cada producto tendrá un nombre, una descripción, una imagen y un precio. Por otra parte, de cada producto queremos conocer las unidades que hemos vendido y la valoración media del producto, pero estos dos atributos se podrán derivar a partir de otros elementos del diagrama (y, por lo tanto, se indican como derivados con la barra ante el nombre).

Las categorías tendrán, también, un nombre y una imagen. Además, las categorías forman una jerarquía que hemos indicado como una agregación en la que cada categoría es agregada a una supercategoría o a ninguna, mientras que cada categoría es la agregación de cualquier número de subcategorías. El hecho de que ésta sea una agregación nos da el matiz de todo-parte entre categorías y sus subcategorías y documenta el hecho de que las categorías no pueden formar ciclos. Cada categoría es una agregación de cualquier número de productos, además, de manera tal que un producto puede estar agregado a una o más categorías.

1.3.2. Clientes

De los clientes que se registren, nuestro sistema tendrá que conocer el nombre, los apellidos, el teléfono (es opcional), el nombre de usuario, la contraseña y el correo electrónico. Por lo tanto, hemos identificado una clase Cliente con estos atributos.

1.3.3. Cesta y pedido

Los clientes pueden añadir productos a la cesta de la compra. Una cesta es una composición de líneas de cesta en la que, para cada línea, tenemos un producto y un número de línea que la identifica dentro de la compra, proporcionando un orden.

Por ejemplo:

1) "Concierto para piano y orquesta de Schumann"
2) "Requiem de Mozart"

Cuando una cesta es, finalmente, confirmada, pasa a ser un pedido. Los pedidos tendrán un número que los identifica y los datos de pago (número, tipo y fecha de caducidad de la tarjeta utilizada y fecha en que se efectuó el pago del pedido). Tanto para cestas como para pedidos, además, querremos conocer a qué cliente pertenecen.

Hemos resuelto este problema de análisis con una generalización: Cesta y Pedido son clases con muchos elementos en común (las líneas, el cliente, etc.), pero con elementos específicos. Podemos decir, pues, que las dos son subclases de una clase Compra que tendrá los elementos comunes.

Ésta será una generalización dinámica, ya que una misma instancia puede pasar de ser Cesta a ser Pedido. Con respecto al análisis, nada nos impide utilizar este modelo, aunque los lenguajes de programación orientados a objeto no soporten la generalización dinámica.

El precio de una Compra será un atributo derivado, ya que se puede calcular a partir del precio de cada producto que incluya.

1.3.4. Valoración

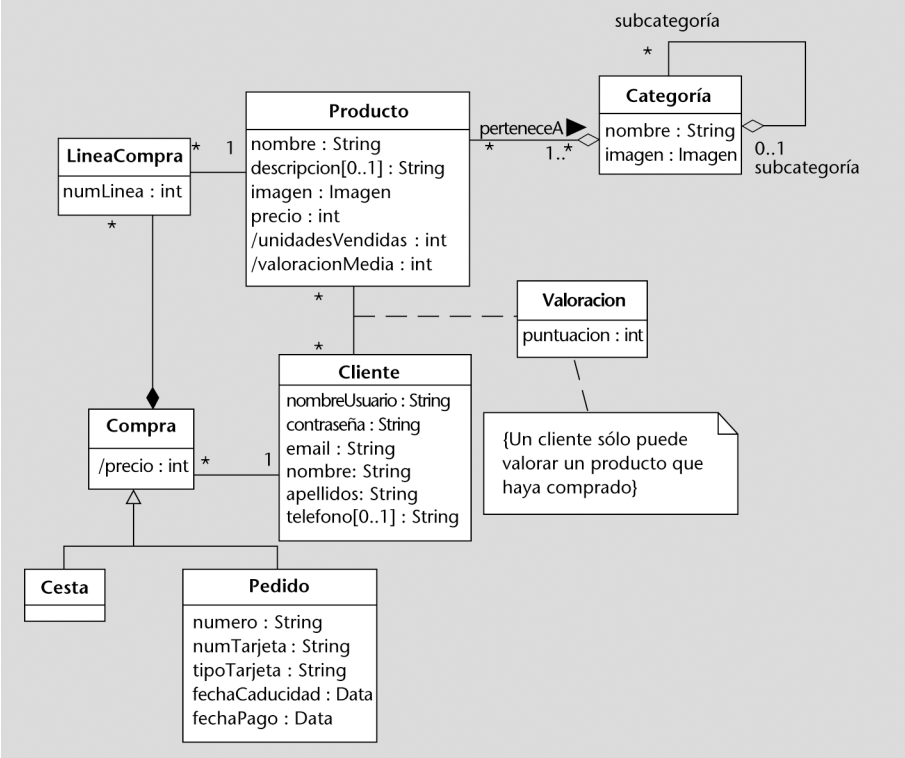
Los clientes pueden valorar productos de tal manera que un cliente puede valorar cualquier número de productos y un producto puede ser valorado por cualquier número de clientes. Pero hay que documentar claramente en el modelo que un mismo cliente no puede valorar dos veces el mismo producto.

Por lo tanto, la valoración será una asociación entre cliente y producto. Por definición, no puede pasar que el mismo cliente y el mismo producto estén asociados dos veces. Además, como de cada valoración habrá que guardar información (la puntuación que ha dado al cliente al producto), tenemos que crear una clase asociativa de la valoración.

También queremos indicar que un cliente no puede valorar un producto que no haya comprado. Como no hay elementos gráficos de UML que nos permitan representar este tipo de restricción, utilizamos una restricción textual: el comentario con el texto de la restricción entre los símbolos "{" y "}".

1.3.5. Modelo conceptual

El modelo conceptual resultante es el siguiente:



1.4. Diagramas de colaboración de los casos de uso

Para completar el análisis, tenemos que documentar el comportamiento del sistema para cada caso de uso. Se trata de analizar los mensajes que actores y sistema intercambian y, desde un punto de vista de análisis, identificar:

- Qué interacción deberá hacer el sistema con los usuarios: clases de frontera.

- Qué gestión deberá hacer el sistema de cada caso de uso: clases de control.

- Qué clases del modelo estático de análisis participarán para resolver cada caso de uso: clases de entidad.

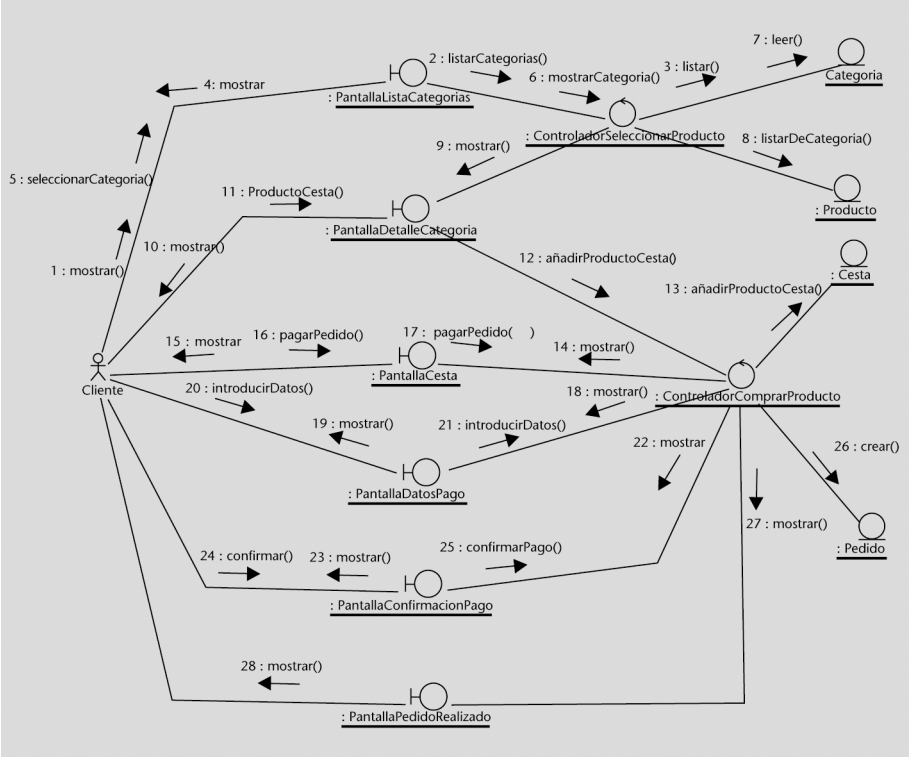
Para documentar este modelo del comportamiento, utilizaremos diagramas de colaboración de UML, ya que permiten representar de manera sencilla la interacción entre diversos objetos.

Sólo hemos incluido el diagrama correspondiente al caso de uso más importante: "Comprar productos". El resto se haría de una manera parecida.

1.4.1. Caso de uso "Comprar productos"

Para este caso de uso hemos hecho el diagrama de colaboración simplificado que documenta el escenario en que un usuario ya identificado quiere hacer la compra de un producto y todo el proceso se completa sin errores (escenario principal del caso de uso).

Denominaremos *PantallaX* a la clase de frontera correspondiente a la pantalla X. Así, por ejemplo, podremos distinguir la clase de frontera PantallaCesta que representa la pantalla Cesta de la clase de entidad Cesta.



Como en este escenario interviene, además del mismo caso de uso, el caso de uso incluido "Seleccionar producto", nuestro diagrama de colaboración tiene dos clases de control, una por caso de uso: ControladorSeleccionarProducto y ControladorComprarProducto.

El proceso empieza cuando el cliente pide la lista de categorías para seleccionar una (clase de frontera PantallaListaCategorias). La clase de frontera delega en el controlador para pedirle la lista de categorías que se va a mostrar, el cual, a su vez, delega en la clase de entidad ya identificada Categoría. Una vez mostrada la pantalla, el cliente selecciona una categoría (mensaje 5). En respuesta a eso, el controlador, después de consultar la categoría y los productos que contiene, muestra el detalle (a través de PantallaDetalleCategoria).

Cuando el usuario selecciona uno de los productos de la categoría y lo añade a la cesta, el control pasa al caso de uso correspondiente (clase Controlador-ComprarProducto), que lo añade a la cesta y muestra su contenido (Pantalla-Cesta).

El caso de uso continúa de una manera parecida. No hay que describir todo el detalle, pero sí que es interesante destacar algunos puntos:

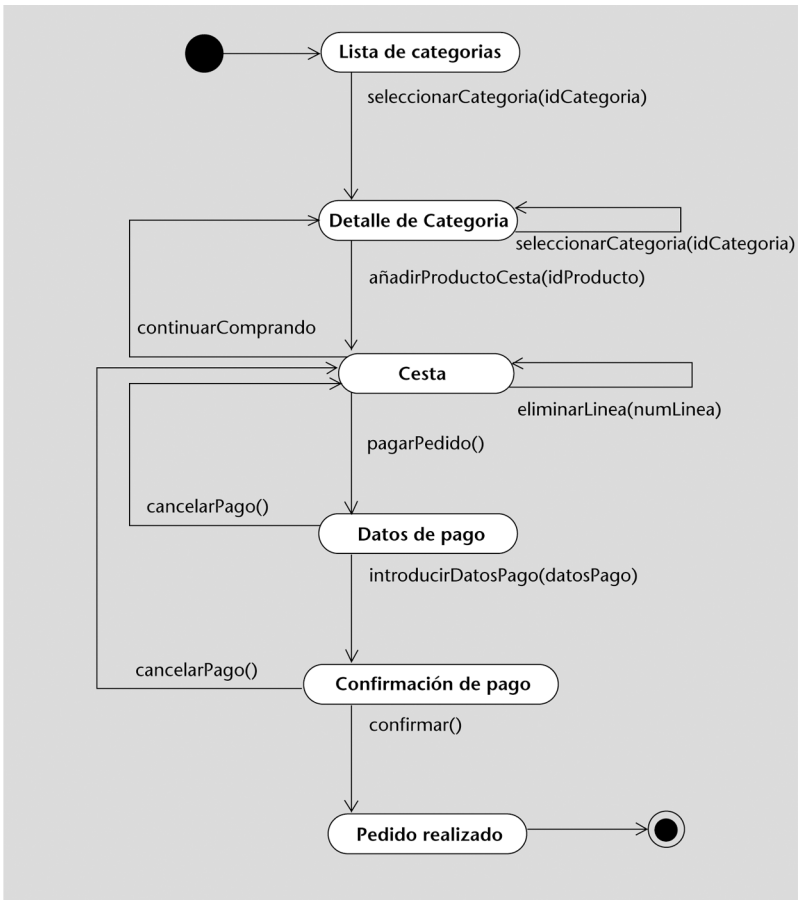
- Los actores sólo interactúan con clases de frontera.

- Las clases de frontera sólo interactúan con actores y clases de control, nunca entre sí ni con las clases de entidad.

- Cada clase de control representa un caso de uso. Recibe mensajes de las clases de frontera (nunca de los actores directamente) y habrá que identificar con qué clases de entidad (del diagrama estático de análisis previamente realizado) interactúa.

1.5. Interfaz gráfica de usuario

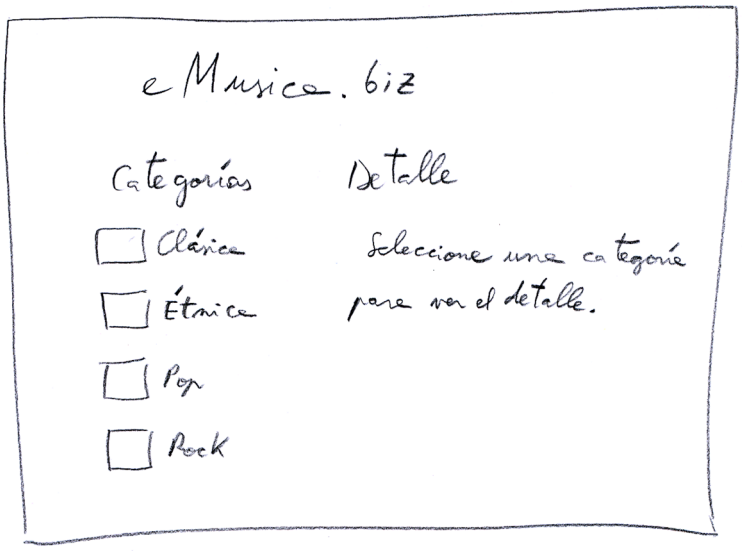
Para hacer el análisis de la interfaz gráfica de usuario, nos basaremos en los diagramas de colaboración simplificados de los casos de uso. A partir de éstos elaboraremos un diagrama de estados de UML por cada caso de uso, que represente las diferentes pantallas y transiciones que se pueden dar durante la ejecución del caso de uso. Este diagrama nos servirá como resumen de las transiciones entre pantallas que aparecen en los diagramas de colaboración que habremos hecho previamente. Para el caso de uso "Comprar producto", el diagrama resultante es el siguiente:



A partir de aquí ya podemos hacer un análisis de cada pantalla que interviene en el caso de uso. Para hacerlo, una de las maneras más rápidas y sencillas es realizar bocetos a mano alzada de las pantallas identificadas.

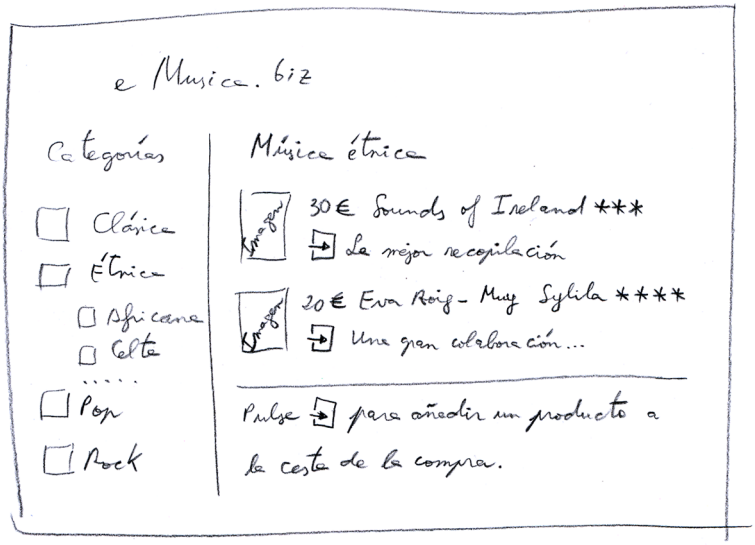
1.5.1. Lista de categorías

La lista de categorías tiene que mostrar las categorías principales de la tienda. Hay que mostrar los datos indicados en la especificación textual del caso de uso "Seleccionar producto":



1.5.2. Detalle de categoría

La pantalla de detalle de categoría tiene que mostrar todos los datos de una categoría. Éstos incluirán las subcategorías y los productos incluidos en la categoría. Para cada producto tiene que dar la opción de incluirlo en la cesta y, además, tiene que permitir continuar navegando hacia otras categorías:



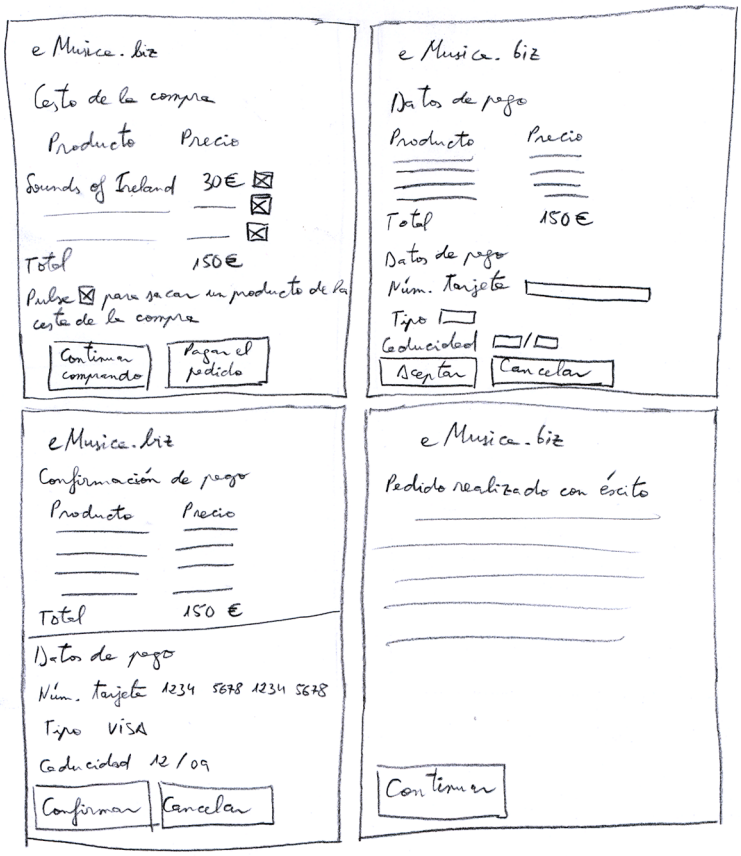
1.5.3. Otras pantallas

La pantalla "Cesta" tiene que mostrar los productos de la cesta (nombre y precio de cada uno) y dar la opción de continuar comprando, pagar el pedido y sacar productos de la cesta.

La pantalla "Datos de pago" tiene que mostrar los mismos datos que la pantalla anterior, pero además ha de pedir los datos de pago. Aquí ya tenemos que tomar una decisión de análisis como, por ejemplo, el tipo de tarjeta que se pedirá en un desplegable.

La pantalla "Confirmación de pago" tiene que mostrar todos los datos pedidos hasta el momento y dar las opciones de confirmar el pago o cancelar todo el proceso.

Finalmente, en la pantalla "Pedido realizado" sólo hace falta mostrar un mensaje. El mensaje concreto no es importante a estas alturas, pero sí que hay que mostrar que sólo hay una opción para el usuario: "Continuar".



2. Análisis con patrones

Una vez llegados a este punto, disponemos ya de una versión preliminar del análisis completo de nuestro sistema. El uso de patrones durante el análisis es una herramienta que, hasta ahora, no hemos utilizado; si lo hubiéramos hecho, habríamos obtenido una solución mejor, ya que hemos evitado algunos problemas que no hemos detectado durante la elaboración del análisis sin utilizar patrones.

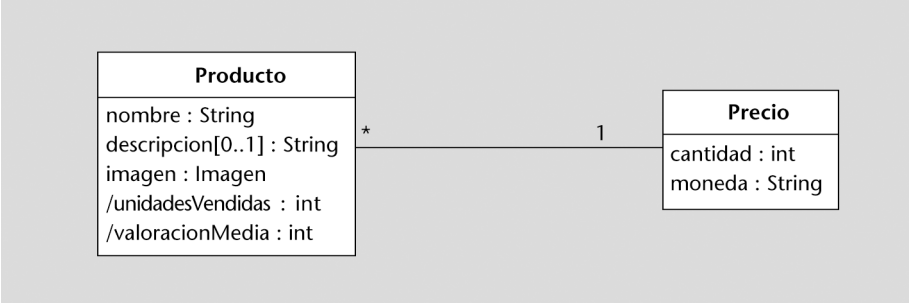
2.1. Precios internacionales

Hasta ahora hemos analizado el precio de los productos como un atributo de tipo entero. Refinando los requisitos, en entrevistas sucesivas se nos comunica que se quiere que el alcance de eMusica sea internacional y que, en determinados países, los precios se den en la moneda local. Además, el sistema puede utilizar varias divisas para registrar los precios de los productos (sobre todo el euro y el dólar); cada producto se quiere registrar en la divisa del proveedor del producto.

Por lo tanto, nos encontramos con que el precio no se puede reducir a un simple atributo de tipo numérico, aunque parecería bastante natural, ya que es una cantidad. Es decir, hemos de ser capaces de representar cantidades diferentes dependiendo de la divisa, y también de hacer conversiones entre estas cantidades para poder dar las equivalencias a la hora de proporcionar precios para mercados diferentes.

Consultando nuestro catálogo de patrones de análisis, encontramos que éste puede ser un buen ejemplo de aplicación del patrón Cantidad. En este caso nos encontramos con el contexto descrito en el patrón de tener que representar cantidades en unidades diferentes. Sólo habrá que tener cuidado para que la conversión entre unidades, en lugar de ser fijada como lo pueda ser entre unidades de longitud, dependa de la cotización de cada moneda en el momento de hacer la conversión.

De momento, sin embargo, lo que habrá que tener en cuenta es que el sistema tiene que conocer el precio del producto es una divisa y, más adelante, habrá que tener presente que ha de ser capaz de convertirlo dependiendo de la nacionalidad del cliente. Con respecto al análisis, nuestro diagrama de clases quedará refinado de manera tal que el precio de un producto se representará según este fragmento:



2.2. Historial de precios

Durante la iteración actual, estamos revisando el análisis del caso de uso "Gestionar precios" cuando nos damos cuenta de que este caso de uso permite modificar el precio de un producto. Cuando se modifica el precio de un producto, habrá que tener en cuenta que es posible que ya haya pedidos para este producto y que el importe de éstos no se tiene que ver modificado.

Según el análisis elaborado hasta ahora, este problema no queda resuelto, ya que el precio del pedido, actualmente, es un atributo que se deriva a partir del precio de los productos que incluye. Así, en la versión actual del análisis, un cambio en el precio de un producto cambiaría el precio del pedido y, por lo tanto, también el importe del pago que se hizo para aquel pedido. Eso es, evidentemente, un error que hay que corregir.

Una primera solución podría ser guardar en el pedido el importe total en el caso de uso "Comprar productos".

a) Beneficios: solución sencilla

b) Inconvenientes:

- Cuando recuperamos un pedido, no podemos dar el precio de las líneas de producto (sólo tenemos el total).

- Cuando modificamos un precio, puede ser que tengamos que modificar algunos de los pedidos que estén asociados a este precio (por ejemplo, si hay pedidos que todavía no han sido cobrados).

Visto eso, podríamos decidir que sea la línea de compra quien almacene este valor, de manera que el importe del pedido continúe siendo calculado, pero a partir del importe de las líneas de pedido y no de los productos.

a) Beneficios: soluciona el problema de la consulta de pedidos.

b) Inconvenientes: Esta solución no refleja el hecho de que el precio de una línea de pedido tiene que ser el precio del producto a la fecha en que se hace el pedido. Por ejemplo, no reflejamos que dos líneas de pedido de un mismo producto en una misma fecha tienen que tener el mismo precio.

Por este último inconveniente, nos planteamos una tercera solución que consiste en saber el precio de un producto en una fecha determinada, de manera que el sistema registre las variaciones en el precio a lo largo del tiempo. Así, en todo momento, podemos calcular el precio de una línea de pedido a partir del precio del producto en el momento en que se hizo el pedido.

a) Beneficios: Evita el problema anteriormente mencionado de posibles inconsistencias en el precio de diversas líneas con la misma fecha y el mismo producto.

b) Inconvenientes: solución más compleja.

Para elaborar esta solución, decidimos consultar nuestro catálogo de patrones de análisis para ver si hay algún patrón que sea aplicable. Una vez revisado el catálogo, encontramos el patrón Asociación histórica, que parece que nos podría ayudar.

En este caso, originalmente, nos encontrábamos con una variación descrita en el patrón, donde la información histórica que queremos guardar estaba en forma de atributo en lugar de ser una asociación. Pero al aplicar el patrón Cantidad, ya hemos transformado el atributo en una asociación, por lo cual podemos aplicar el patrón Asociación histórica en su versión principal. La asociación que menciona el patrón es la asociación binaria entre Producto y Precio. Queremos conocer el precio del producto (el valor de la asociación) cuando se hizo el pedido (el momento determinado del que habla el patrón). Otro problema que soluciona este patrón es el hecho de poder consultar la historia de los precios de un producto. Éste no es el problema que queremos solucionar y, por lo tanto, no nos afecta de cara a la elección del patrón.

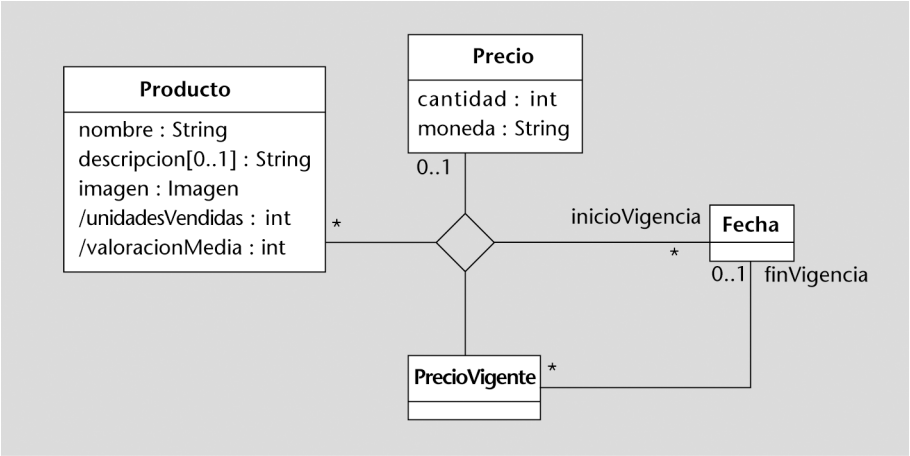
Así pues, parece que tenemos un buen candidato para solucionar nuestro problema con las modificaciones de los precios pero, antes de aplicarlo, hay que plantearse qué inconvenientes introduce:

- Solución más compleja: en nuestro caso, eso se traduce en un aumento de la complejidad de la gestión de los precios (ahora hay que tener en cuenta la información histórica). Así pues, habrá que revisar el caso de uso "Gestionar precios" para ver si hay que hacer cambios en la interacción con el usuario. Una vez revisado, vemos que se puede gestionar esta información de manera transparente para el usuario.

- No se puede trasladar la multiplicidad: en la asociación binaria original, un Producto sólo puede tener un Precio. Ahora, en cambio, podrá tener más

de uno (con fechas diferentes). Así pues, la restricción original tenemos que interpretarla en el sentido de que, en un momento dado, un Producto sólo puede tener un Precio, restricción que no podemos reflejar mediante la cardinalidad de la ternaria. Sí que podemos reflejar, sin embargo, que en una misma fecha un producto sólo puede sufrir un cambio de precio.

Finalmente, decidimos aplicar el patrón con el resultado siguiente:



2.3. Jerarquía de categorías

Revisando el caso de uso "Gestionar productos" vemos que tenemos una jerarquía de categorías que hemos representado como una agregación reflexiva de la clase Categoría.

a) Beneficios: solución sencilla.

b) Inconvenientes: no se han encontrado.

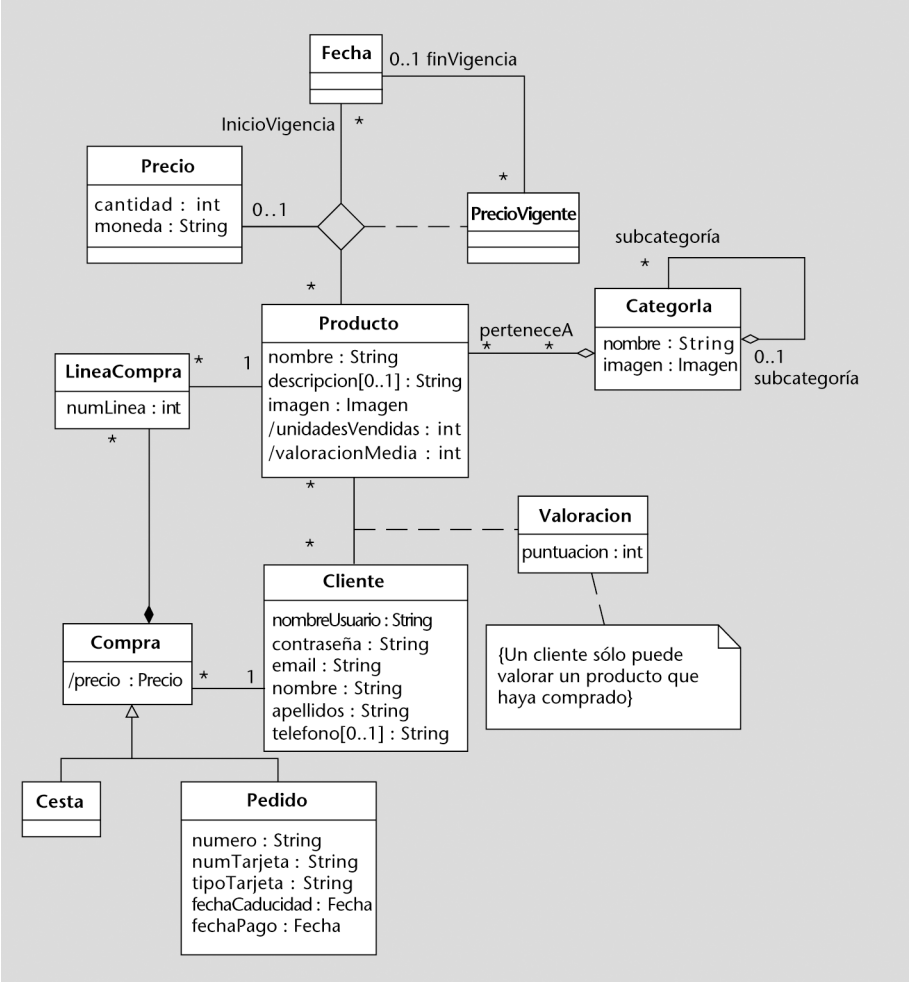
No obstante, hay un patrón en nuestro catálogo que parece que podría ser de aplicación: el patrón ObjetoCompuesto.

El contexto es el que describe el patrón: hay una asociación que agrupa los productos (elementos) en categorías (colecciones). El patrón soluciona el problema de tratar de manera uniforme productos y categorías. Revisando los casos de uso, vemos que no hay ninguno que haga un tratamiento de estas características.

Como aplicar este patrón no nos soluciona ningún problema pero sí que introduce inconvenientes, decidimos no aplicarlo y quedarnos con la solución que teníamos.

2.4. Modelo conceptual final

El modelo conceptual resultante de las modificaciones hechas al análisis es el siguiente:



3. Arquitectura con patrones

3.1. Arquitectura en tres capas

El primer paso del diseño consiste en definir la arquitectura del sistema (es decir, tomar las decisiones de diseño que afectan a la globalidad del sistema). Para la arquitectura general de nuestro sistema, aplicaremos el patrón de arquitectura en capas utilizando la división tradicional en tres capas: presentación, dominio y servicios técnicos (que, en nuestro caso, constará de un subsistema de persistencia y otro de mensajería electrónica). De esta manera conseguiremos el objetivo de poder trabajar en el diseño e implementación de cada capa situándonos, en cada caso, a un nivel de abstracción diferente.

Ved también

Podéis encontrar más información sobre el patrón Arquitectura en capas en el apartado 4.1 del módulo "Catálogo de patrones".

Aplicando el principio de inversión de dependencias, durante el diseño de cada capa tendremos que definir la abstracción que esta capa espera de la capa inferior. Así pues, durante el diseño de la capa de presentación definiremos la interfaz que esperamos de la capa de dominio, mientras que durante el diseño de la capa de dominio, definiremos la interfaz que esperamos de la capa de persistencia. Este enfoque nos llevará a diseñar las capas en orden descendente empezando, pues, por la capa de presentación.

Ved también

Podéis encontrar más información sobre el principio de inversión de dependencias en el catálogo de principios de diseño del módulo "Catálogo de patrones".

Además, durante el diseño de las diferentes capas iremos aplicando los patrones de diseño que consideramos adecuados para obtener un diseño final de calidad.

3.2. Arquitectura de la capa de presentación

Para la capa de presentación aplicaremos el patrón de arquitectura Modelo-Vista-Controlador en su variante adaptada para la arquitectura en capas. Este patrón se aplicará, por lo tanto, a todas las clases de frontera detectadas en el diagrama de colaboración del apartado 1.4.1, "Caso de uso 'Comprar productos'", de manera que dividiremos las clases de frontera en modelos, vistas y controladores.

Ved también

Podéis encontrar más información sobre el patrón Modelo-Vista-Controlador en el apartado 4.3 del módulo "Catálogo de patrones".

En el caso de eMusica, utilizaremos el *framework* Java Servlets para la interacción con el usuario. Este *framework* nos facilita la creación de aplicaciones basadas en web mediante clases Java especiales llamadas *servlets* (miniaplicaciones de servidor) para el tratamiento de peticiones HTTP, que nos servirán para implementar los controladores y clases Java para la generación de páginas HTML denominadas JSP (*Java server pages*), que nos servirán para implementar

Web recomendada

Podéis encontrar más información sobre el *framework* Java Servlets en: http://java.sun.com/products/servlet/

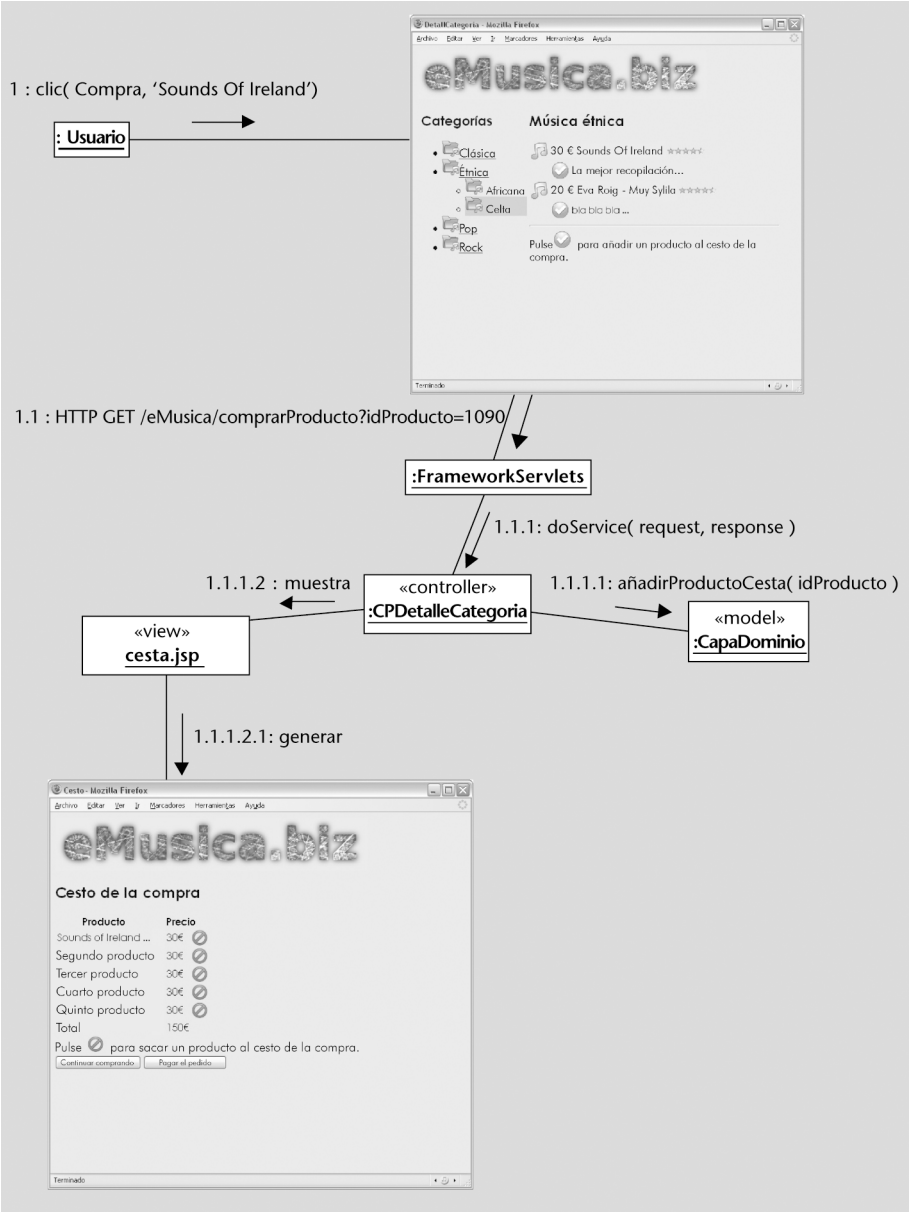
las vistas. Finalmente, el modelo será la capa de dominio y, por lo tanto, durante el diseño de la capa de presentación sólo definiremos las interfaces que necesitamos que nos implemente esta capa.

Asociaremos, en cada pantalla de la aplicación, un controlador (que será una miniaplicación de servidor) que gestionará todas las peticiones realizadas desde aquella pantalla y una vista (que será una JSP) que se encargará de generar, en cada caso, el código HTML necesario para mostrar la pantalla al usuario. Así pues, para cada clase de frontera del diagrama de colaboración tendremos una vista y un controlador.

Ejemplo

Por ejemplo, la clase de frontera PantallaDetalleCategoría, correspondiente a la pantalla "Detalle de categoría", tendrá una vista llamada *detalleCategoria.jsp* y un controlador llamado *CPDetalleCategoria*.

A continuación se puede ver el diagrama de colaboración asociado a la acción de seleccionar un producto para la compra por parte del usuario cuando se encuentra en la pantalla "Detalle de categoría":



En este caso, el usuario, que está visualizando el código HTML generado por la vista detalleCategoria.jsp, hace clic sobre el icono de compra del producto Sounds of Ireland y, como resultado de eso, se llama al controlador de la pantalla CPDetalleCategoría. Éste determina cuál de los posibles acontecimientos de la pantalla se ha producido, llama a las operaciones correspondientes de la capa de dominio, decide que la siguiente pantalla que hay que mostrar es "Cesta" y llama a la vista (cesta.jsp) para que genere el código HTML que se tiene que mostrar al navegador del usuario.

3.2.1. Comunicación entre la capa de presentación y la capa de dominio

Finalmente, nos queda la tarea de definir el mecanismo de comunicación con la capa de dominio. En este caso, aplicaremos el patrón de asignación de responsabilidades Controlador de manera que, desde la capa de presentación, accederemos a la capa de dominio a través de controladores que tratarán los acontecimientos asociados a las peticiones del usuario.

Ejemplo

Hay que distinguir entre el controlador de presentación (resultado de haber aplicado el patrón de arquitectura MVC) y el controlador de dominio (resultado de haber aplicado el patrón de asignación de responsabilidades Controlador).

Así pues, tendremos dos tipos de controladores:

a) el controlador de la capa de presentación, que recibirá acontecimientos de presentación (peticiones HTTP), y

b) el controlador de la capa de dominio, que recibirá acontecimientos de dominio (añadirProductoACesta, confirmarPago, etc.).

4. Diseño de la capa de presentación

Durante el diseño de la capa de presentación tendremos que diseñar, por una parte, el aspecto externo de nuestro sistema y, por la otra, las clases mediante las cuales implementaremos la capa de presentación.

En este caso, el diseño externo del sistema se limitará al diseño de las pantallas: se diseñará el aspecto gráfico del sistema, así como los controles de interfaz gráfica de usuario (desplegables, botones, etc.) que se utilizarán para mostrar la información a los usuarios y recoger las peticiones.

En el aspecto interno, se diseñarán las clases que implementarán las pantallas (modelos, vistas y controladores) y también las interacciones de estas clases con el resto del sistema.

4.1. Diseño externo

Para hacer el diseño de las pantallas, partiremos de los análisis siguientes:

- Análisis de casos de uso: datos mostrados y pedidos por cada caso de uso.
- Análisis de la interfaz de usuario por cada caso de uso: diagrama de estados de las pantallas y las transiciones posibles.
- Análisis de pantallas: boceto de las pantallas hecho durante el análisis.

A partir de estos análisis, tendremos que escoger, para cada pantalla, qué elementos utilizaremos para mostrar y pedir cada dato y qué controles para permitir cada posible interacción.

A continuación tenemos que establecer la correspondencia entre los acontecimientos que se producen en la capa de presentación (peticiones HTTP) y los acontecimientos del diagrama de estados de las pantallas.

El uso de tecnología web para la capa de presentación nos impone ciertas restricciones al diseño de pantallas que habrá que tener presentes:

- El cliente recibe las pantallas en formato HTML que su navegador mostrará. Por lo tanto, los controles de interfaz gráfica de usuario que podemos utilizar se limitan a los que ofrezca HTML.

- El cliente interactúa con el sistema haciendo peticiones HTTP. Cada petición HTTP puede tener parámetros que el usuario indicará rellenando formularios HTML.

4.1.1. Lista de categorías

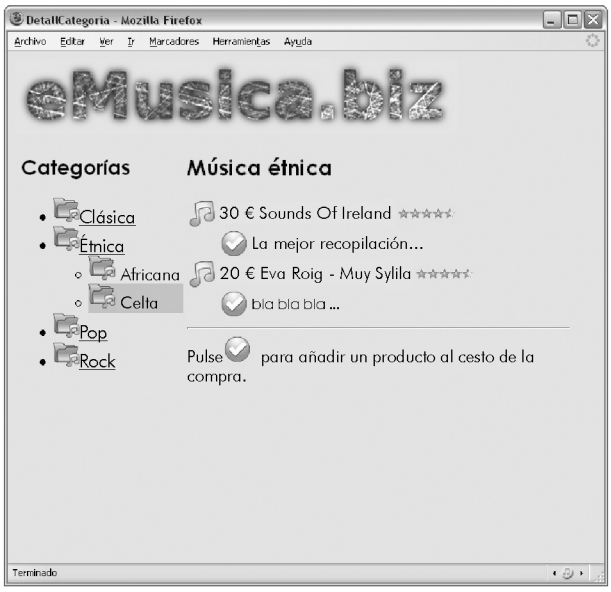
La lista de categorías tiene que mostrar un listado de todas las categorías para que el usuario seleccione una. La única transición que admite esta pantalla es la selección de una categoría.

Para mostrar las categorías se ha decidido utilizar una lista de HTML y para la selección de categoría se ha decidido hacer que cada nombre de categoría sea un enlace que permita seleccionar esa categoría. El resultado es el siguiente:

4.1.2. Detalle de categoría

El detalle de una categoría tiene que mostrar los mismos datos que el listado de categorías y, además, los datos de la categoría seleccionada.

De nuevo, se utilizará texto HTML para mostrar las categorías y los productos de la categoría seleccionada y se usarán enlaces para las diversas transiciones posibles: seleccionar una categoría o seleccionar un producto.



4.1.3. Otras pantallas

Para el resto de pantallas utilizaremos el mismo proceso:

4.2. Diseño interno

Para hacer el diseño interno de las pantallas, partiremos del diseño externo, el análisis realizado y de la arquitectura que ya hemos definido para obtener el diseño de las vistas, de los controladores y del modelo.

4.2.1. Diseño de las vistas

Las vistas, tal como se ha indicado anteriormente, serán páginas JSP (*Java server pages*) que generarán dinámicamente la página HTML que se enviará al navegador del usuario. No entraremos, sin embargo, en detalle en el diseño de cada página JSP, ya que su diseño es muy directo.

4.2.2. Diseño de los controladores

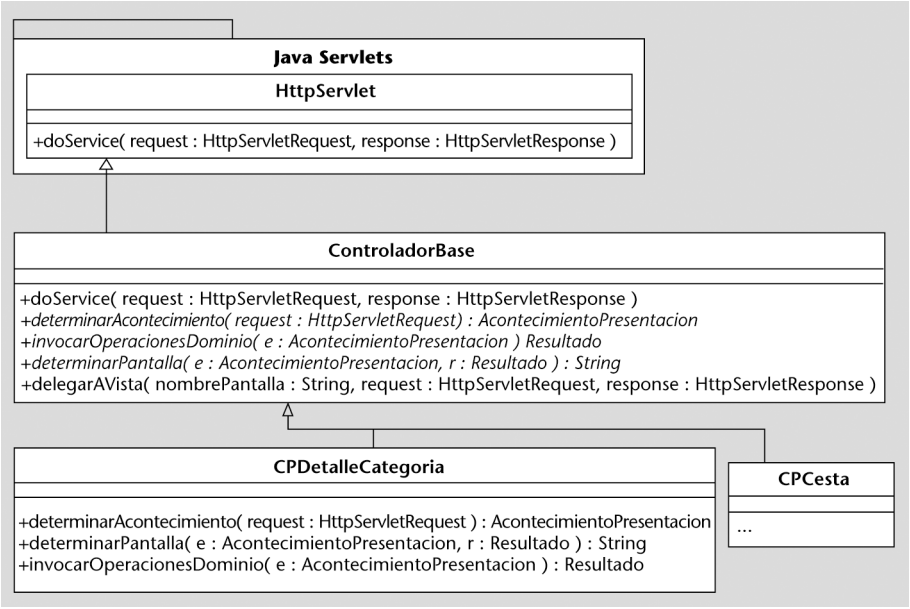
Siguiendo la arquitectura que hemos decidido, cada pantalla tendrá asociada un controlador, que será una miniaplicación de servidor que tendrá que tratar las peticiones HTTP que pueda originar. El controlador tendrá que determinar cuál de los posibles acontecimientos definidos durante el análisis se corresponde a aquella petición, tendrá que invocar las operaciones adecuadas de la capa de dominio, seleccionar la siguiente pantalla y delegar la generación de la página HTML de respuesta al usuario a la vista de la pantalla seleccionada.

Podemos ver, pues, que todos los controladores de presentación siguen una estructura muy similar. Como queremos asegurarnos de que todos los controladores respetan este algoritmo general, aplicaremos el patrón de diseño Método plantilla, de manera que tendremos una superclase común a todos los controladores que llamaremos *ControladorBase* y que implementará el algoritmo general que hemos descrito antes:

Ved también

Podéis encontrar más información sobre el patrón de diseño Método plantilla en el apartado 6.5 del módulo "Catálogo de patrones".

1) Determinar el acontecimiento que se ha producido.
2) Invocar las operaciones adecuadas de la capa de dominio.
3) Determinar la pantalla siguiente que se mostrará.
4) Delegar la generación de la página HTML a la vista asociada a la pantalla.



El código Java asociado a la clase ControladorBase sería el siguiente:

```
public abstract class <b>ControladorBase</b> extends HttpServlet {
  // Método heredado de HttpServlet que es llamado por cada petición HTTP
  public void <b>doService</b>(
  HttpServletRequest request,
  HttpServletResponse response) {
    AcontecimientoPresentacion e =
      determinarAcontecimientoPresentacion(request);
    Resultado r = invocarOperacionesDominio(e);
    String nombrePantalla = determinarPantalla(e,r);
    delegarAVista(nombrePantalla, request, response);
}
protected abstract AcontecimientoPresentacion
    <b>determinarAcontecimientoPresentacion</b>
    (HttpServletRequest request);
  protected abstract Resultado <b>invocarOperacionesDominio</b>
    (AcontecimientoPresentacion e);
  protected abstract String <b>determinarPantalla</b>(
    AcontecimientoPresentacion e, Resultado r);
  protected void <b>delegarAVista</b> (String nombrePantalla,
    HttpServletRequest request, HttpServletResponse response) {
    ... //Se redirige la petición a la vista
  }
}
```

4.2.3. Diseño del modelo

Durante la definición de la arquitectura decidimos aplicar el patrón de asignación de responsabilidades Controlador para la comunicación entre la capa de presentación y la capa de dominio. Por lo tanto, en este punto tenemos que decidir qué controladores formarán nuestro modelo. De entre las opciones que tenemos para la aplicación del patrón Controlador, una de las más adecuadas para este caso será un controlador de caso de uso. Esta opción hace que no tengamos ni tan pocos controladores como para que sean demasiados complejos de escribir y mantener, ni un número muy grande de controladores que serían difíciles de gestionar en conjunto:

Ved también

Podéis encontrar más información sobre el patrón de asignación de responsabilidades Controlador en el apartado 5.1 del módulo "Catálogo de patrones".

Caso de uso	Controlador
Comprar productos	ControladorComprarProductos
Valorar productos	ControladorValorarProductos
Registrarse	ControladorRegistrarse
Consultar pedidos	ControladorConsultarPedidos
Gestionar productos	ControladorGestionarProductos
Gestionar precios	ControladorGestionarPrecios

Para los dos casos de uso incluidos, también utilizaremos un controlador propio:

Caso de uso	Controlador
Seleccionar producto	ControladorSeleccionarProducto
Login	ControladorLogin

4.2.4. Caso de uso "Comprar productos"

El único caso de uso cuyo diseño estudiaremos en detalle será "Comprar productos".

Diseño de las vistas

Este caso de uso tiene asociadas varias pantallas. Para cada pantalla tendremos que tener una página JSP que generará el código HTML necesario para visualizar la pantalla en el navegador del usuario. Por ejemplo, para la pantalla "Cesta" tendremos la página JSP cesta.jsp.

Diseño de los controladores

Tomemos como ejemplo la pantalla "Detalle de categoría". Esta pantalla admite dos acontecimientos: seleccionarCategoria(idCategoria) y añadirProductoCesta(idProducto).

Por lo tanto, el controlador CPDetalleCategoria tiene que tratar estos dos acontecimientos.

1) SeleccionarCategoria: invocará una operación de dominio que dado el identificador de la categoría, nos dé todos los datos asociados a ésta y los muestre al usuario en la pantalla "Detalle de categoría".

2) AñadirProductoCesta: invocará una operación de dominio que, a partir de un identificador de producto, lo añade a la cesta del usuario y, una vez añadido, mostrará al usuario la pantalla "Cesta".

Así pues, ya hemos detectado dos operaciones que necesitaremos en la capa de dominio:

- consultarCategoria(idCategoria:String):Categoria
- añadirProductoCesta(idProducto:String):Compra

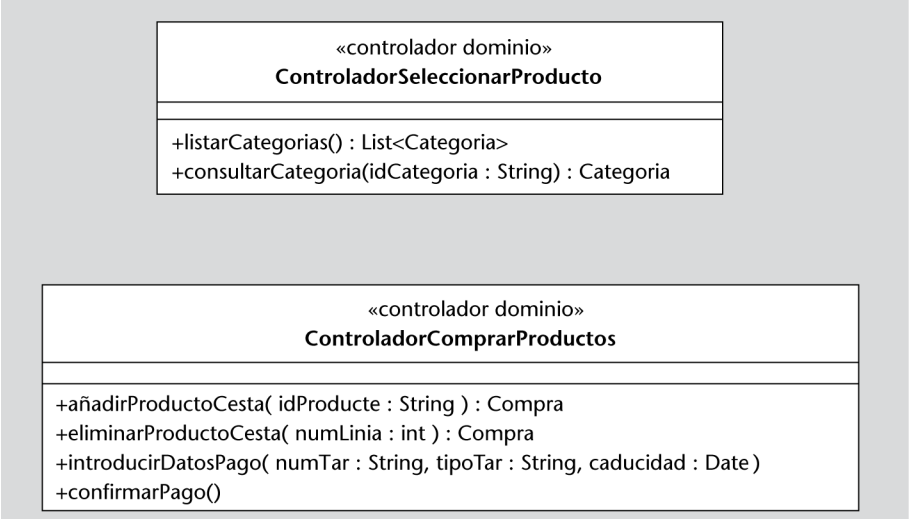
Aplicando el mismo proceso en el resto de pantallas, determinamos que las operaciones que necesitamos de la capa de dominio para los controladores asociados al caso de uso "Comprar productos" son las siguientes:

Pantalla	Acontecimiento	Operaciones	Siguiente pantalla
Lista de categorías	inicio	listarCategorias()	Lista de categorías
Lista de categorías	seleccionarCategoria	consultarCategoria(idCategoria)	Detalle de categoría
Detalle de categoría	seleccionarCategoria	consultarCategoria(idCategoria)	Detalle de categoría
Detalle de categoría	añadirProductoCesta	añadirProductoCesta(idProducto)	Cesta
Cesta	pagarPedido	-	Datos de pago
Cesta	eliminarLinea	eliminarProductoCesta(numLinea)	Cesta
Cesta	continuarComprando	-	Detalle de categoría
Datos de pago	introducirDatosPago	introducirDatosPago(numTarjeta, tipoTarjeta, caducidad)	Confirmación de pago
Datos de pago	cancelarPago	-	Cesta
Confirmación de pago	confirmar	confirmarPago()	Pedido realizado
Confirmación de pago	cancelarPago	-	Cesta

Diseño del modelo

Una vez determinadas todas las operaciones que necesitaremos de la capa de dominio, las asignaremos a los controladores de los casos de uso donde se llamen. En nuestro caso, asignaremos la operación consultarCategoria al controlador del caso de uso "Seleccionar productos" y añadirProductoCesta al controlador del caso de uso "Comprar producto".

De esta manera, asignando todas las operaciones detectadas para el caso de uso "Comprar productos", la interfaz esperada de los controladores de dominio será:



Hay un caso especial: la autenticación del usuario (caso de uso Login). En este caso, dejaremos que sea el *framework* Servlets quien controle cuándo se tiene que pedir la identificación del usuario y ejecutar este caso de uso.

Este mismo proceso se iría repitiendo para todos los casos de uso hasta determinar las interfaces esperadas para todos los controladores de dominio de nuestro sistema.

5. Diseño de la capa de dominio

El diseño de la capa de dominio será el núcleo del proceso de diseño de nuestro sistema. Para poder trabajar con esta capa, nos basaremos en los principios siguientes:

a) Habrá un conjunto de clases llamadas *controladores* que implementarán las operaciones que la capa de presentación necesita. Estas operaciones estarán definidas en términos de la capa de dominio y serán independientes de cualquier tecnología utilizada en la capa de presentación.

b) Habrá un conjunto de clases de software que representarán las clases conceptuales detectadas durante el análisis y que implementarán toda la lógica del sistema. No obstante, será preciso realizar cierto proceso de diseño para poder representar en forma de clases software conceptos que nuestro lenguaje orientado a objetos quizás no soporta, como las asociaciones ternarias o la herencia múltiple.

c) Finalmente, habrá que definir qué servicios esperamos que nos ofrezca la capa de servicios técnicos. Así, definiremos los servicios que esperamos antes de implementarlos y evitaremos la tendencia a que la capa de servicios técnicos tenga dependencias hacia la capa de dominio.

5.1. Diseño de clases del modelo conceptual

El objetivo de esta tarea es obtener un conjunto de componentes software que representen los conceptos analizados en el modelo conceptual. Estas clases formarán el grueso de la capa de dominio y se encargarán de implementar la lógica de negocio de nuestro sistema.

La tecnología orientada a objetos nos permite construir componentes próximos a los conceptos, asociando una clase software a cada clase del modelo conceptual que se quiere representar. Ahora bien, esta tecnología no permite representar todos los conceptos del modelo conceptual y, por lo tanto, será necesario seguir un proceso de diseño que resuelva los puntos conflictivos que tenemos a continuación:

- Asociaciones
- Clases asociativas
- Herencia múltiple
- Generalización dinámica
- Información derivada
- Restricciones de integridad

Por lo tanto, el diseño de las clases del software consistirá en elaborar un diagrama de clases del diseño a partir del diagrama de clases del análisis que ya hemos visto.

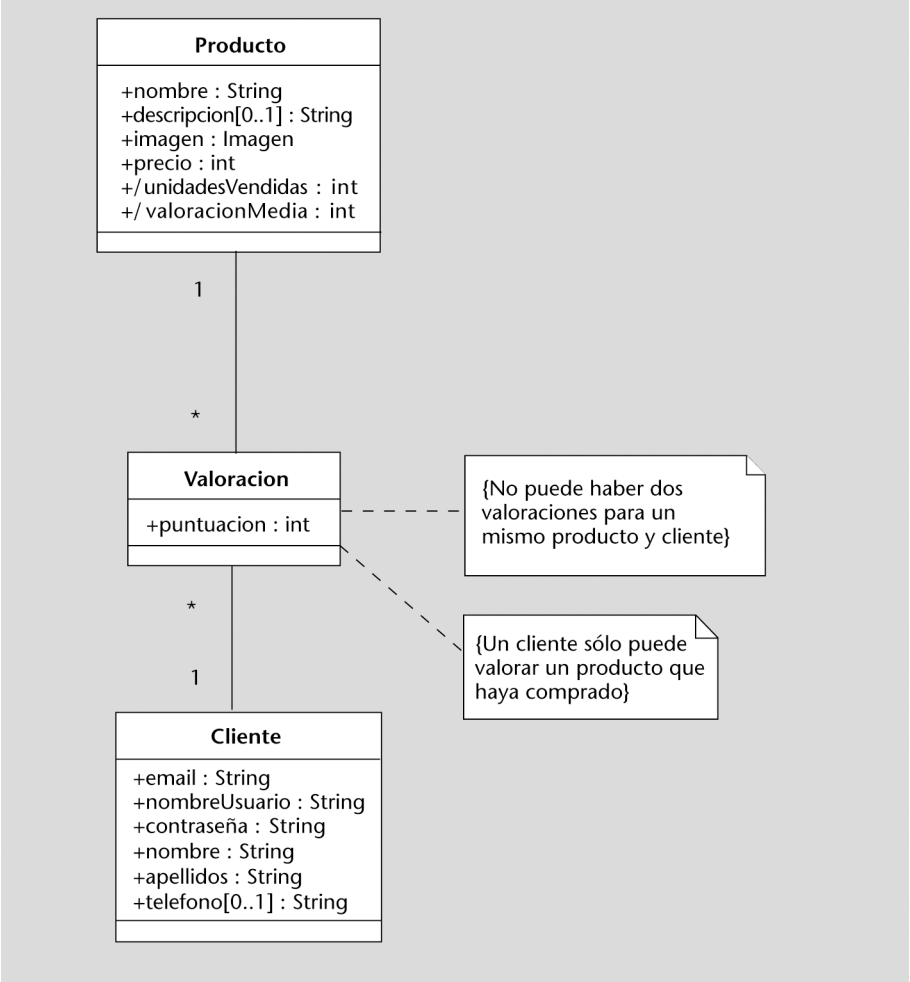
5.1.1. Asociaciones binarias

Las asociaciones binarias del modelo de análisis se representarán como atributos de las clases asociadas. En el caso de la Compra y sus líneas, por poner un ejemplo que muestre este tipo de asociaciones, el diseño detallado será el siguiente:

```
public class Compra{
   private List<LineaCompra> lineas;
   public Iterator<LineaCompra> getLineas() {
     return lineas.iterator();
   }
   public void addLinea(LineaCompra linea){
     lineas.add(linea);
   }
   public void removeLinea(int posicion){
     lineas.remove(posicion);
   }
  }
  public class LineaCompra{
    private Compra compra;
    public Compra getCompra(){
      return compra;
    }
```

5.1.2. Valoraciones de productos

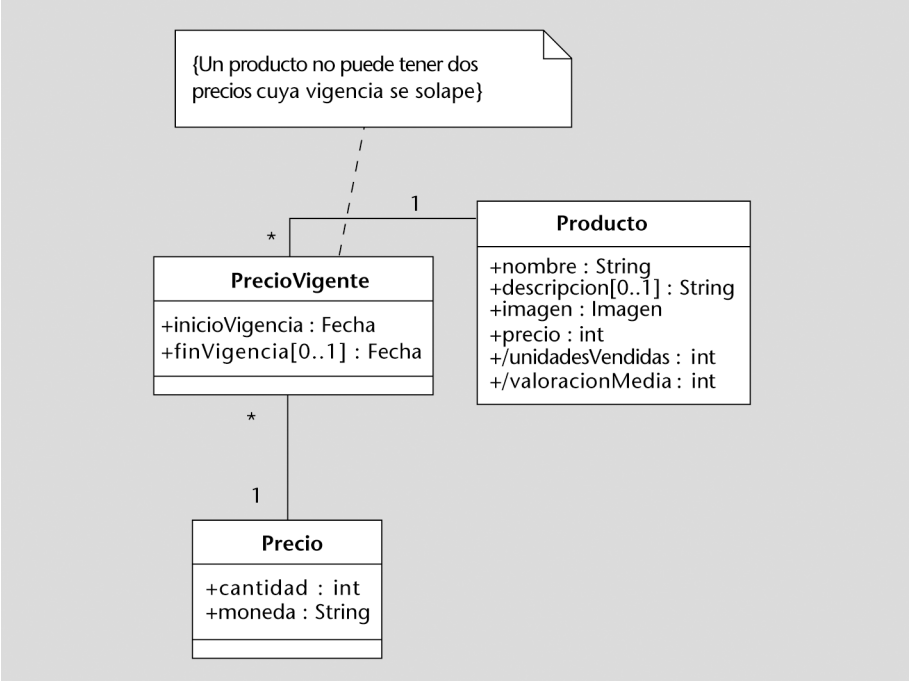
La clase Valoración es asociativa de una asociación binaria entre Producto y Cliente. Por lo tanto, habrá que transformarla en el diseño detallado que presentamos a continuación:



5.1.3. Precios de los productos

A causa de la inclusión de información histórica de precios, se ha introducido una clase PrecioVigente, que es una asociativa de la ternaria entre Producto, Precio y Fecha.

Como Fecha es un tipo de datos que Java ya ofrece, la transformación necesaria durante el diseño es la de una clase asociativa:



5.1.4. Compras, pedidos y cestas

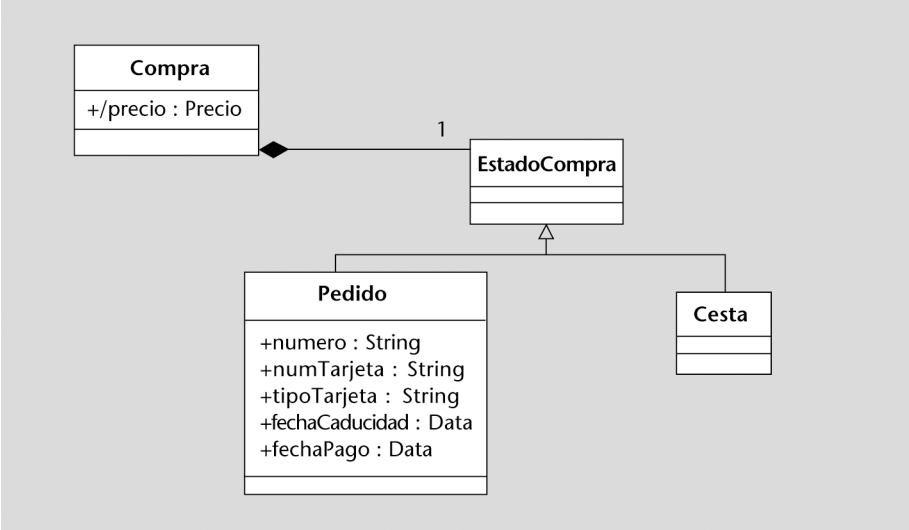
Otro caso que hay que adaptar es el de la generalización dinámica. En nuestro caso, una Compra inicialmente es de clase Cesta. Pero en el momento en que el cliente hace la compra, pasa a ser de clase Pedido y, por lo tanto, a tener un comportamiento, unos atributos y unas asociaciones diferentes. Tenemos, pues, una generalización dinámica.

Como en la mayoría de las generalizaciones dinámicas, este cambio de clase del objeto Compra se puede ver como un cambio de estado de la cesta, que en cierto momento pasa al estado Pedido. Así, pues, nuestro catálogo de patrones nos da el patrón que nos permitirá hacer el diseño de esta situación: patrón de diseño Estado.

Ved también

Podéis encontrar más información sobre el patrón de diseño Estado en el apartado 6.1 del módulo "Catálogo de patrones".

En nuestro caso, el Contexto será Compra, mientras que los posibles estados serán Cesta y Pedido. Por otra parte, en este caso concreto, los atributos de la Compra dependen del estado en que está; por este motivo, los objetos estado no pueden ser compartidos entre diferentes contextos. Así, la solución para este problema de diseño será la siguiente:

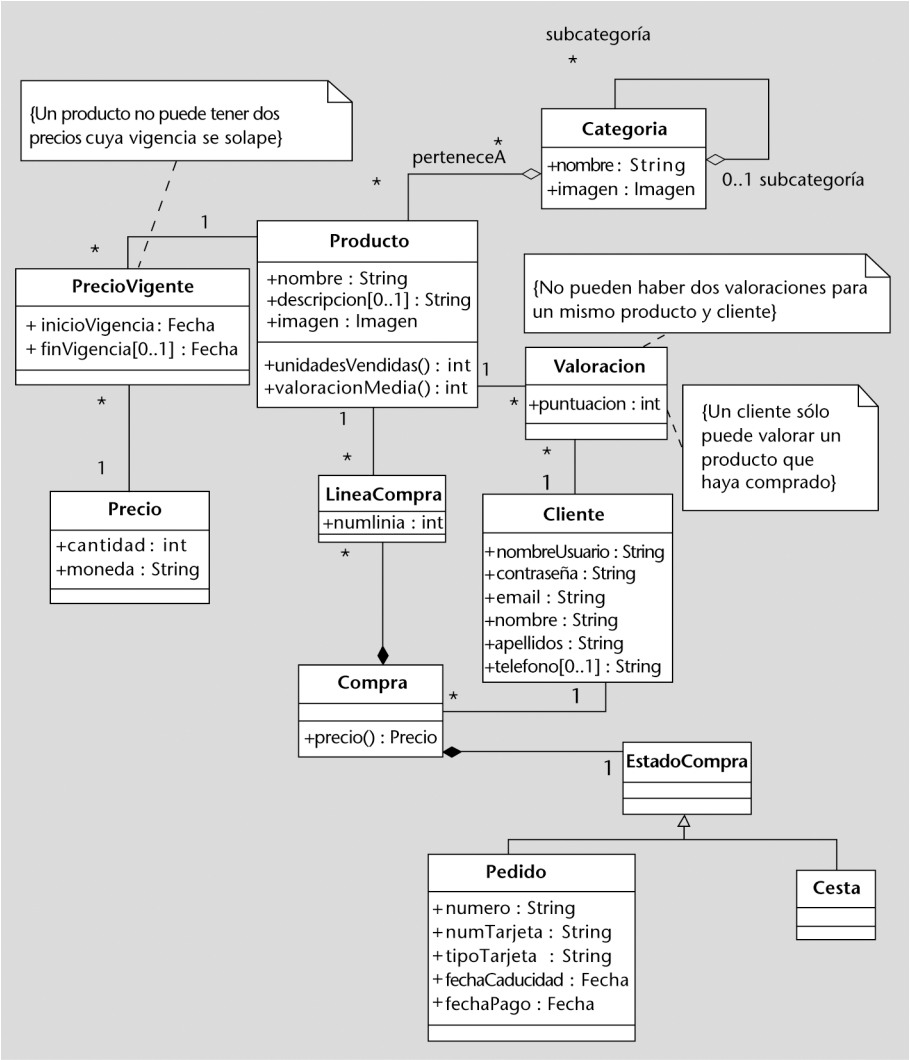


Hay que tener en cuenta que la aplicación del patrón Estado que se ha hecho hasta ahora no es completa, ya que sólo hemos resuelto la parte estática del problema. Más adelante, a la hora de asignar responsabilidades, habrá que volver a acudir al patrón Estado para decidir quién gestionará los cambios de estado de la Compra.

5.1.5. Información derivada

Nuestro modelo conceptual tiene tres atributos que son derivados: las unidades vendidas, la valoración media de un producto y el precio de una compra. Para representarlos en nuestro diseño, decidimos calcularlos cada vez, por lo cual los sustituiremos por operaciones.

5.1.6. Resultado del diseño del modelo conceptual



5.2. Asignación de responsabilidades con patrones GRASP

Según la arquitectura en capas escogida, la capa de presentación interactuará con el usuario y llamará a operaciones de la capa de dominio. Podríamos ver estas llamadas como acontecimientos de dominio. El siguiente paso que hay que dar para diseñar la capa de dominio es decidir cómo las clases que la formen se repartirán las responsabilidades a la hora de atender estos acontecimientos.

Para eso, asignaremos responsabilidades a las clases que ya hemos diseñado en el apartado anterior, a los controladores y a nuevas clases que crearemos si es preciso.

Los patrones de asignación de responsabilidad son patrones que nos ayudarán a resolver aspectos concretos de esta tarea ayudándonos a garantizar unos buenos principios de diseño.

Ved también

Podéis encontrar más información sobre los principios de diseño y los patrones de asignación de responsabilidades en los apartados 2 y 5 del módulo "Catálogo de patrones".

A continuación veremos el proceso de asignación de responsabilidades para las operaciones del caso de uso "Comprar productos" y su caso de uso incluido "Seleccionar productos". No veremos el caso de uso "Login", pero supondremos que, como resultado de la inclusión de este caso de uso, los dos controladores saben cuál es el cliente con quien interactúan (tienen un atributo denominado cliente de tipo *Cliente*).

5.2.1. Listar categorías

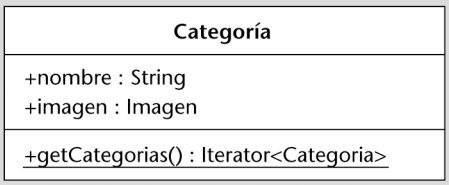
En esta operación del controlador sólo hay una responsabilidad: recuperar todas las categorías del sistema.

Acudiendo a nuestro catálogo de patrones de asignación de responsabilidades, encontramos que disponemos de los patrones Controlador, Experto, Fabricación pura y Creador.

El patrón Creador queda claramente descartado, ya que no tenemos la responsabilidad de crear ninguna instancia. El tratamiento del evento ya lo hemos resuelto mediante el Controlador y, por lo tanto, este patrón tampoco nos ayuda a saber quién tiene que recuperar estas categorías. Así pues, tendremos que aplicar o bien a Experto o bien a Fabricación pura.

Según Experto, tendríamos que pedir la responsabilidad a aquel objeto que tenga los datos necesarios para llevarla a cabo: una clase de dominio que tenga asociadas todas las categorías. Naturalmente, ésta es la clase Categoría (pero no lo es ninguna instancia de ésta).

El diseño resultante de aplicar este patrón es el siguiente:



Aunque ésta es una solución bastante buena, la cohesión de la clase Categoría no es la más alta posible, ya que sus responsabilidades, en estos momentos, son dos claramente diferenciadas:

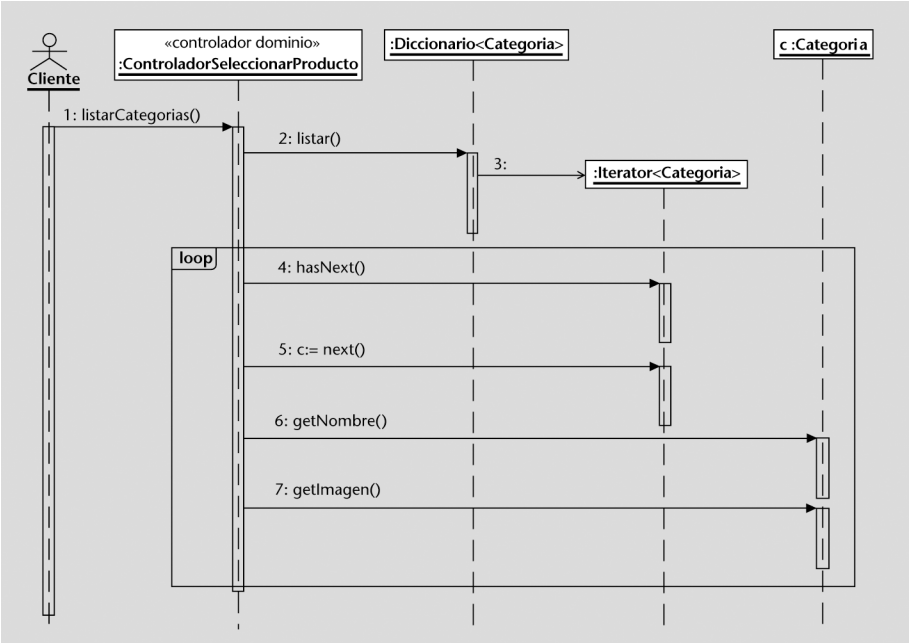
Ved también

Podéis ver más información sobre el principio de diseño alta cohesión en el apartado 2.2 del módulo "Catálogo de patrones".

a) Representar cada instancia de categoría y sus atributos.

b) Representar el conjunto de categorías y ofrecer operaciones estáticas para recuperarlas.

La otra opción que tenemos es utilizar Fabricación pura para obtener un diseño alternativo; introduciremos una nueva clase que llamaremos *Diccionario* y que se encargará de esta función:



La ventaja de esta solución es que ahora la cohesión es máxima, ya que el diccionario tiene una única responsabilidad (representar el conjunto de todas las instancias de una clase) y la clase Categoría tiene una única diferente (representar cada instancia de una Categoría).

5.2.2. Diccionarios

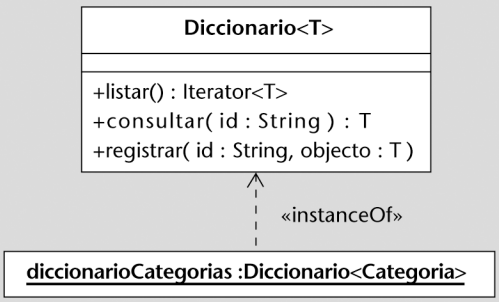
En el apartado anterior hemos introducido una Fabricación pura que hemos llamado *Diccionario* y que merece especial atención, ya que es bastante habitual en el diseño orientado a objetos.

Dado que el diccionario representa el conjunto de instancias de una clase, podemos prever que serán necesarias otras operaciones. En concreto, necesitaremos:

- Recuperar una instancia a partir de un identificador.

- Registrar nuevas instancias en el momento de crearlas.

Por lo tanto, la interfaz completa que utilizaremos para los diccionarios será la siguiente:



Hay múltiples implementaciones posibles para esta interfaz. En nuestro caso, sin embargo, como los datos representados por estos diccionarios serán persistentes, la implementación de los diccionarios formará parte del subsistema de persistencia. Así pues, ya hemos definido una de las interfaces que este subsistema tiene que ofrecer en la capa de dominio.

5.2.3. Consultar categoría

Las responsabilidades que hay que resolver en este caso son las siguientes:

1) Recuperar una categoría a partir de su identificador.

2) Consultar los datos de la categoría seleccionada (el nombre y la imagen).

3) Consultar las subcategorías de la categoría seleccionada.

4) Consultar los productos de la categoría seleccionada.

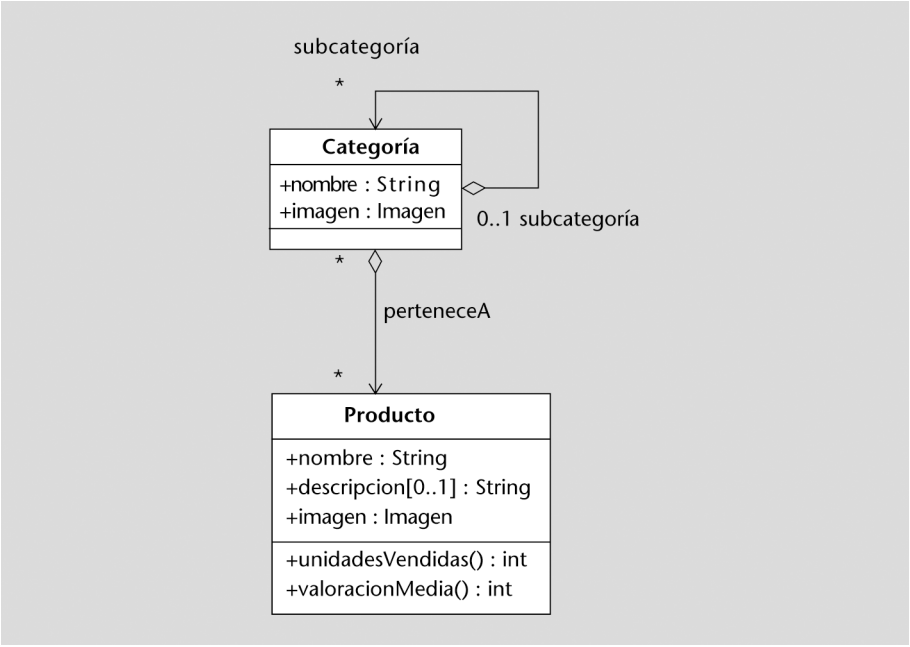
Para recuperar una categoría utilizaremos, de nuevo, la Fabricación pura, ya que nos basaremos en un diccionario de categorías.

Las otras tres responsabilidades están muy cohesionadas entre sí. Por lo tanto, si las juntamos en una única clase, ésta continuará teniendo una cohesión muy alta.

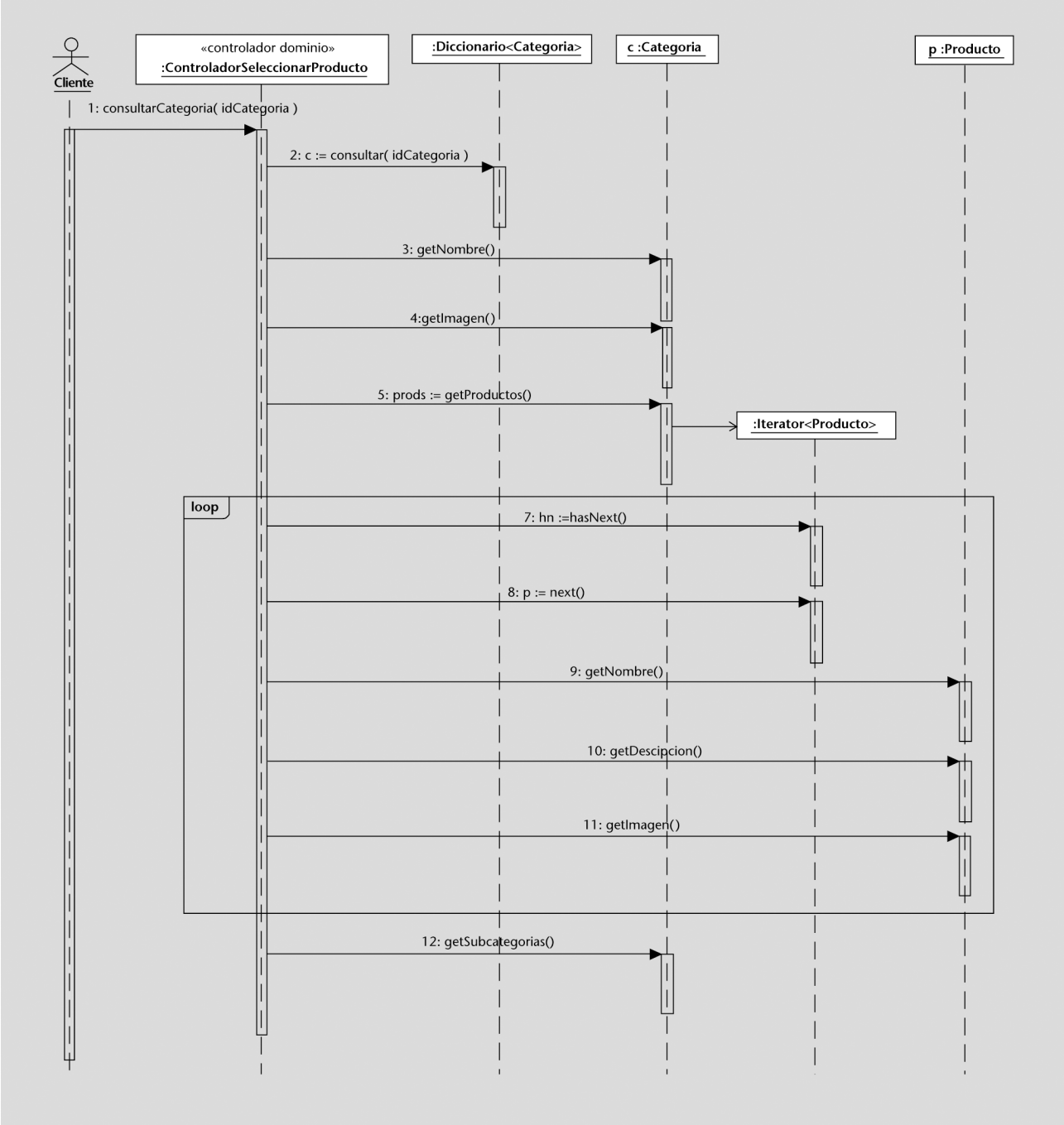
Según el patrón Experto, la clase que se responsabilice de eso tiene que ser la que tiene los datos necesarios. La clase que representa una instancia de categoría y que, por lo tanto, conoce sus datos, sus subcategorías y sus productos, será la clase Categoría. La clase Producto, sin embargo, será la experta en los datos específicos de cada producto.

Eso quiere decir que la asociación de una categoría con sus subcategorías tiene que ser navegable en este sentido; lo mismo pasa con la asociación de una categoría hacia sus productos. En el diseño más detallado, eso se traducirá

en un conjunto de operaciones que, hoy por hoy, representamos como una asociación; aunque esta asociación ya está presente en nuestro diagrama, hay que indicar ahora en qué sentido será navegable la asociación:



El diagrama de secuencias del diseño de esta operación resultará, pues:



El diagrama no muestra cómo se recorre la lista de subcategorías, pero se haría de una manera parecida al recorrido de productos.

5.2.4. Añadir producto

Al añadir un producto, habrá que añadir un nuevo producto a la cesta de la compra del usuario. Hará falta, por lo tanto:

- Crear una cesta si es que no hay una creada.
- Recuperar un Producto a partir de su identificador.

- Determinar el siguiente número de línea de compra.

- Crear una nueva línea de compra en la cesta.

Además, sin embargo, el controlador tiene que devolver los datos de la cesta de la compra. Por lo tanto, también se tendrán que tratar las responsabilidades siguientes:

- Mostrar los productos de la cesta (nombre y precio).

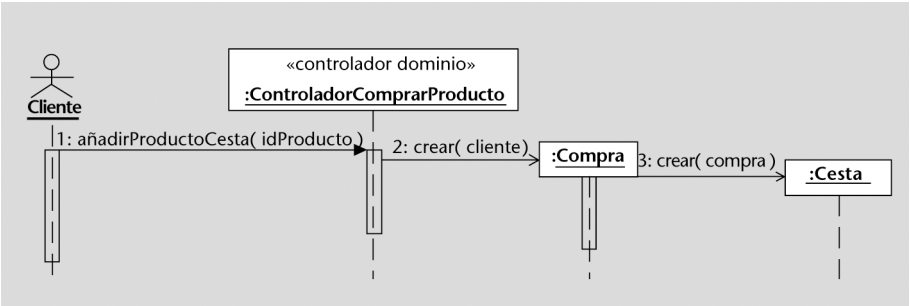
- Calcular el precio total de una cesta.

Entonces, nos tenemos que preguntar quién se encargará de la creación de la Compra. Según el Creador, el Cliente, como tiene asociadas las compras, es un candidato. La otra opción sería que fuera el controlador directamente, ya que es quien tiene los datos de inicialización necesarios.

El controlador tendrá que tener constancia de la cesta con la que está trabajando el caso de uso. Para eso, tendrá un atributo que represente esta cesta. Por este motivo decidimos que sea el controlador quien tenga la responsabilidad de crear la Compra.

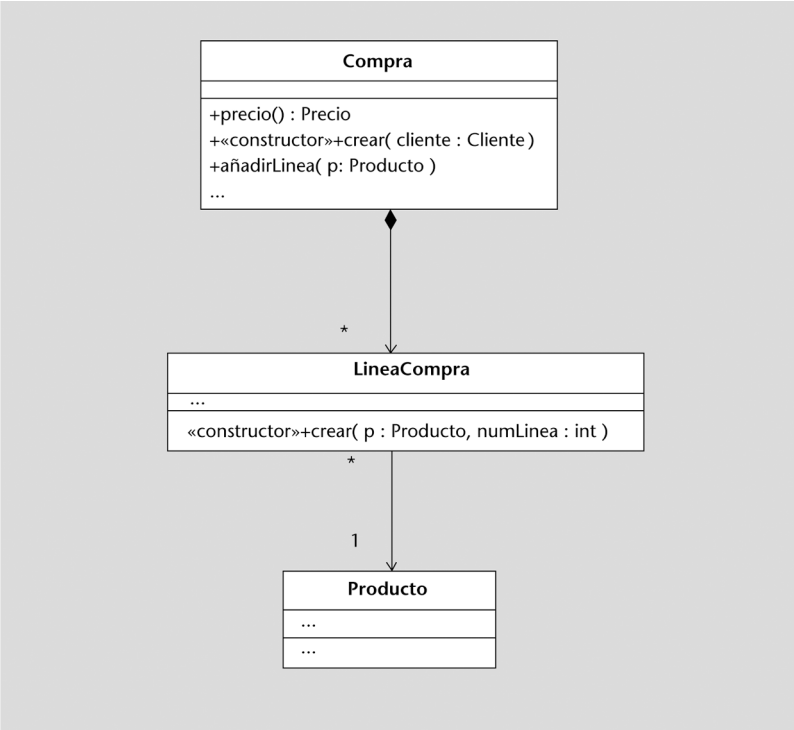
En el momento de crear una Compra según el patrón Estado, ésta tendrá que crear una instancia de su estado inicial por defecto. En este caso, por lo tanto, la Compra tendrá que crear una instancia de Cesta.

El diagrama de secuencias de este fragmento del caso de uso, bajo el supuesto de que el cliente no tiene ninguna cesta asociada (porque, por ejemplo, éste es el primer producto que quiere añadir a la cesta) es el siguiente:



Para la creación de una línea de cesta, podríamos optar por la sencilla solución de que fuera el controlador quien se encargara de ello. Eso, sin embargo, acoplaría el controlador con la estructura interna de la Compra; en particular, lo acoplaría con las clases Compra y LíneaCompra. El patrón de diseño Creador nos sugiere que sea la Compra quien se encargue de crear sus líneas, ya que

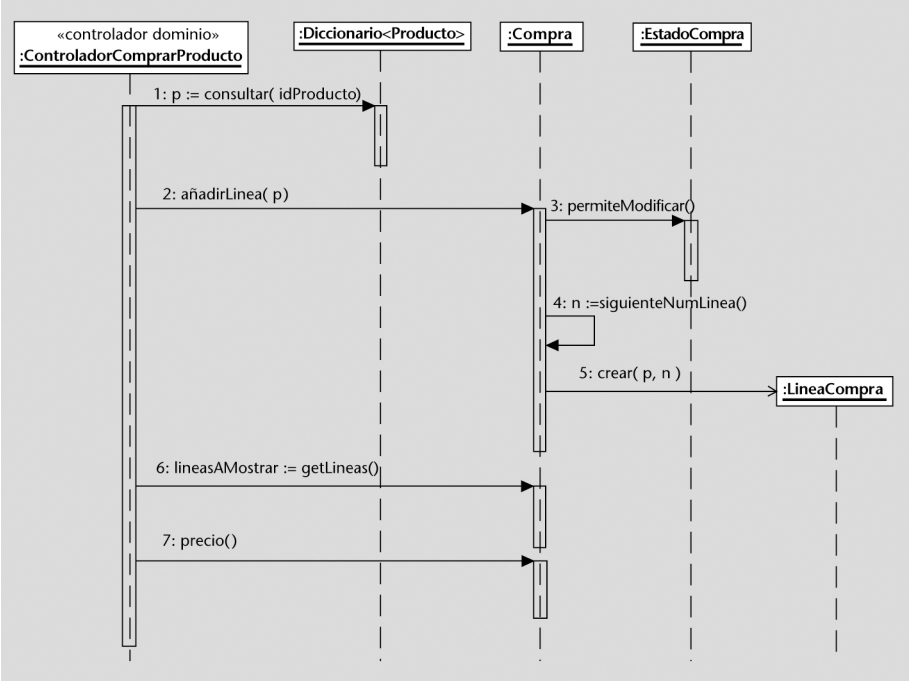
una Compra agrega líneas de compra. Según Experto, será la propia Compra la encargada de determinar el siguiente número de línea que se va a asignar y, por lo tanto, lo pasará al construir la LíneaCompra.



El comportamiento de la Compra, sin embargo, depende de su estado. Así, no queremos que un Pedido permita modificaciones como añadir o quitar líneas. Para eso, según el patrón Estado, la Compra tendrá que delegar en su estado la responsabilidad de determinar este comportamiento variable: si admitimos o no modificaciones.

Como para añadir una línea a una compra necesitaremos indicar el producto y sólo disponemos de su identificador, de nuevo, utilizaremos un diccionario para que el controlador pueda recuperar el Producto a partir de su identificador.

En cuanto a mostrar los productos de la cesta, según Experto, la misma Compra tendrá toda la información necesaria. Finalmente, el cálculo del precio total de la Compra, que ya habíamos modelado como operación al hacer el diseño del atributo derivado correspondiente, según Experto, tiene que ir donde lo hemos puesto: en la clase Compra.



El diagrama de secuencias que adjuntamos muestra la segunda parte de este caso de uso, pero no es completo. La parte de consulta de las líneas de una cesta es bastante parecida a los casos anteriores y sólo se muestra parcialmente. Entre los pasos 6 y 7 del diagrama habría que hacer el recorrido del iterador de líneas de compra lineasAMostrar.

5.2.5. Introducir datos de pago

A priori, al introducir los datos de pago tendríamos que cambiar el estado de la Compra a Pedido y asociar estos datos.

Pero la confirmación del pago que viene después nos plantea una duda: ¿en qué momento tenemos que hacer el cambio de estado a Pedido?

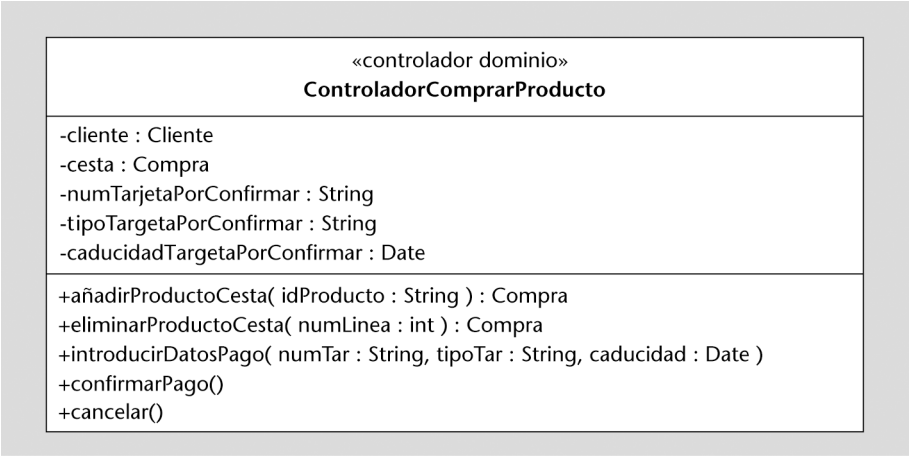
Hay dos soluciones posibles:

1) Cambiar el estado en el momento en que el cliente introduce los datos de pago y volver a cambiarlo si cancela.

2) Aplazar el cambio del estado de la Compra a Pedido para cuando el usuario haya confirmado el pago.

La primera solución tendría sentido, sobre todo, si la cancelación se pudiera producir independientemente de este caso de uso. Pero si la cancelación sólo se puede producir en este caso de uso, no tiene mucho sentido cambiar el estado de la Compra a Pedido hasta que el usuario lo confirme.

Ahora bien, cuando el usuario confirme su pago no volverá a indicar estos datos de pago. Por lo tanto, el controlador tendrá que guardarlos para su uso en la operación siguiente:



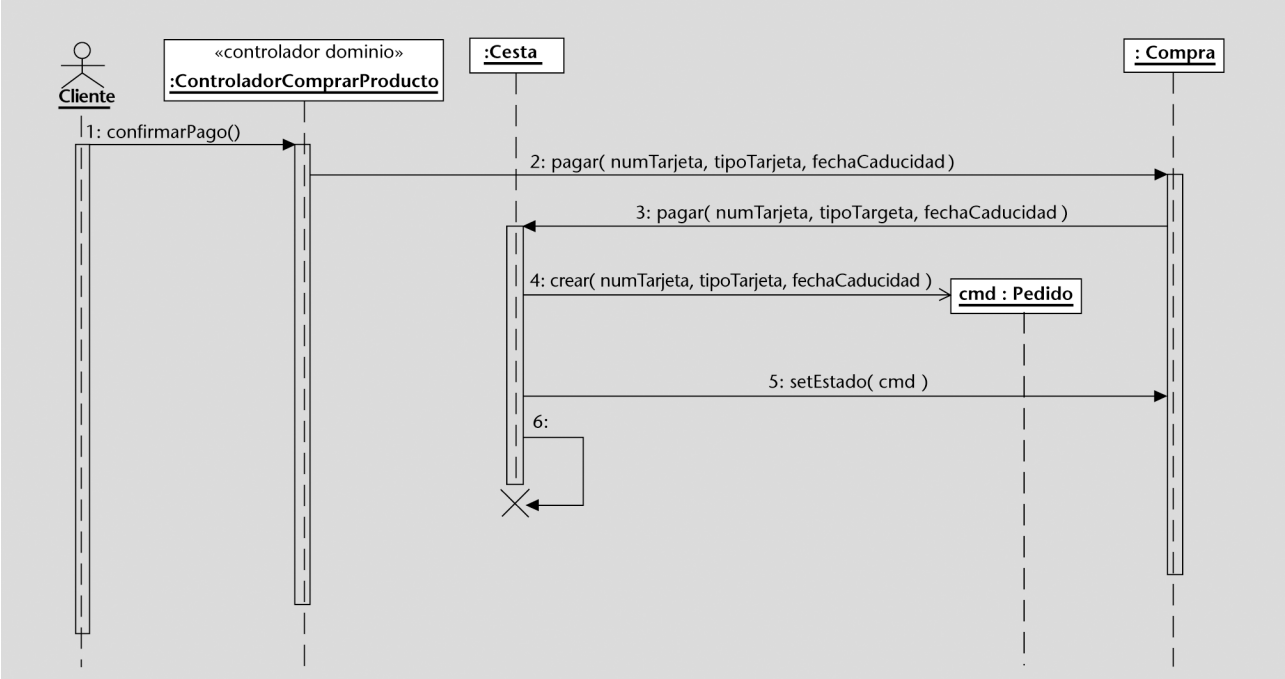
5.2.6. Confirmar pago

Las únicas responsabilidades que hay que resolver para esta operación son las siguientes:

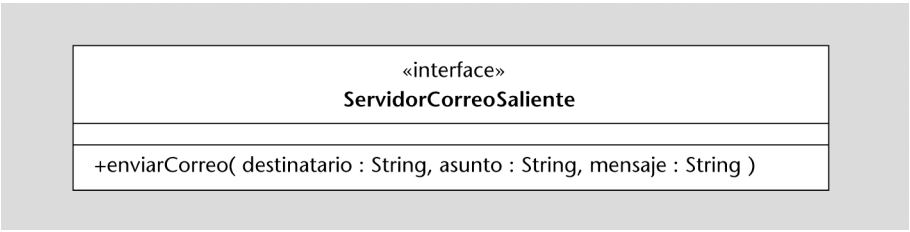
- Cambio de estado de la Compra al estado Pedido.

- Enviar un mensaje de correo electrónico.

Tal como habíamos avanzado, el patrón Estado ya nos define quién tiene que ser el responsable de los cambios de estado de un objeto: el estado o el contexto. Como la operación de pago sólo tiene sentido cuando el estado es Cesta, la responsabilidad la pondremos en la clase Cesta; desde el punto de vista del patrón Experto, la Cesta es, de hecho, quien tiene la información sobre si el estado es válido o no para hacer una compra. Así pues, la Compra delegará en su estado este cambio de estado.



Finalmente, hay que enviar un mensaje de correo electrónico al usuario con información del pedido y los productos comprados. Según el patrón Experto, el mismo Pedido es quien tiene la información necesaria para hacerlo, pero esta solución acoplaría el Pedido con el envío de correos electrónicos, que es una responsabilidad ortogonal respecto a las que tiene hasta ahora. Por ello, el patrón Fabricación pura nos sugiere que sea una nueva clase quien se encargue de la cuestión. Como esta clase no se implementará en la capa de dominio, aplicaremos el patrón Servidor abstracto y la convertiremos en una interfaz:



Ved también

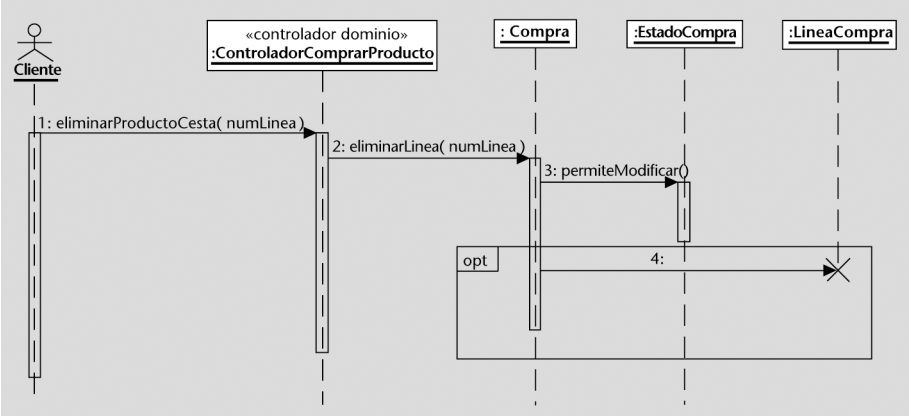
Podéis encontrar más información sobre el patrón de diseño Servidor abstracto en el apartado 6.8.7 del módulo "Catálogo de patrones".

Como el Pedido era el experto en la información necesaria para enviar este mensaje de correo, a él corresponderá utilizar esta fabricación pura para enviar el mensaje.

Tenemos que tener en cuenta que esta interfaz será implementada en la capa de servicios técnicos por el subsistema de mensajería electrónica.

5.2.7. Eliminar línea

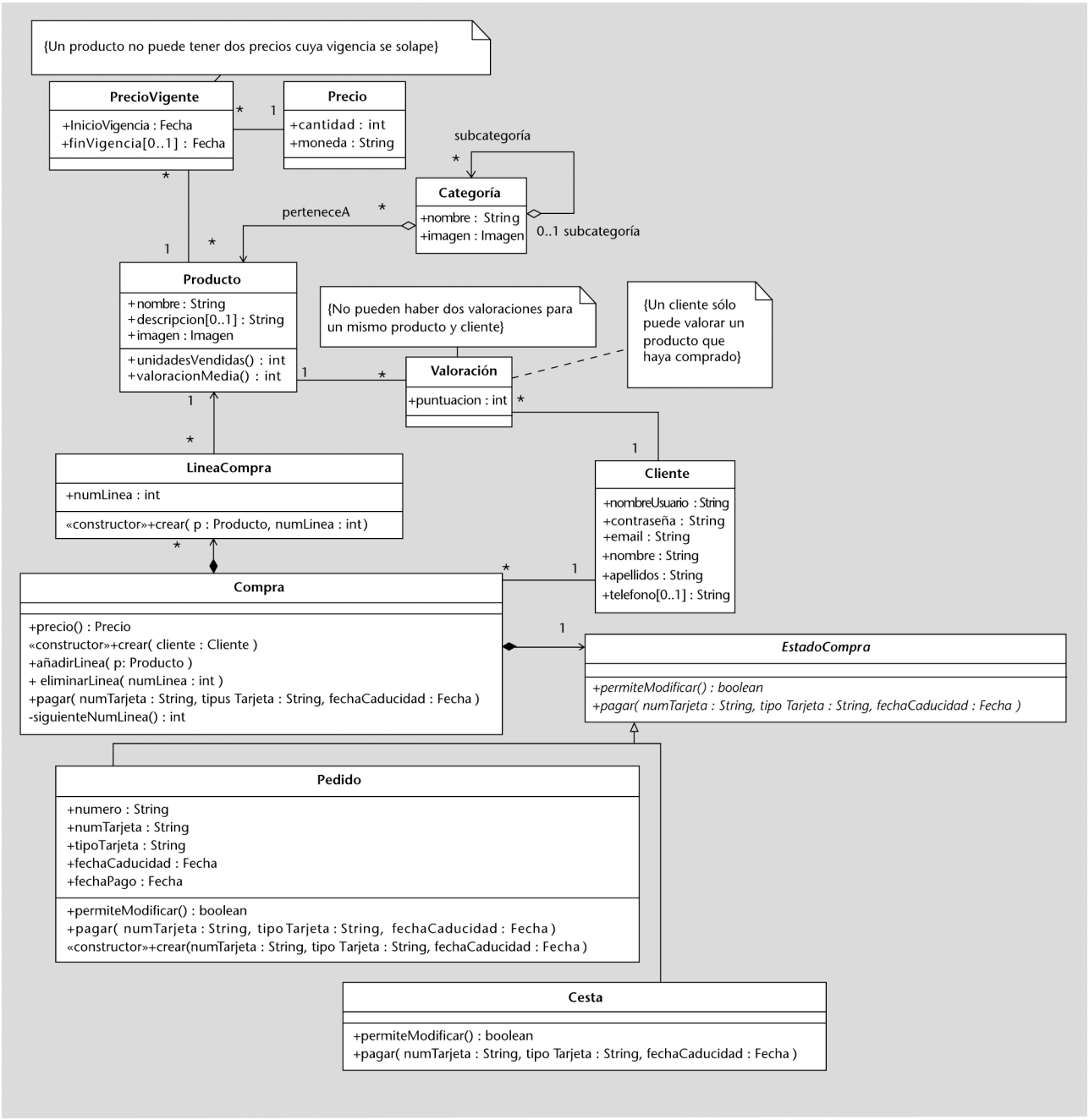
Finalmente, el usuario puede eliminar líneas de la cesta en ciertos puntos del caso de uso. De nuevo, según el patrón Experto, la Compra será la responsable de esta eliminación:



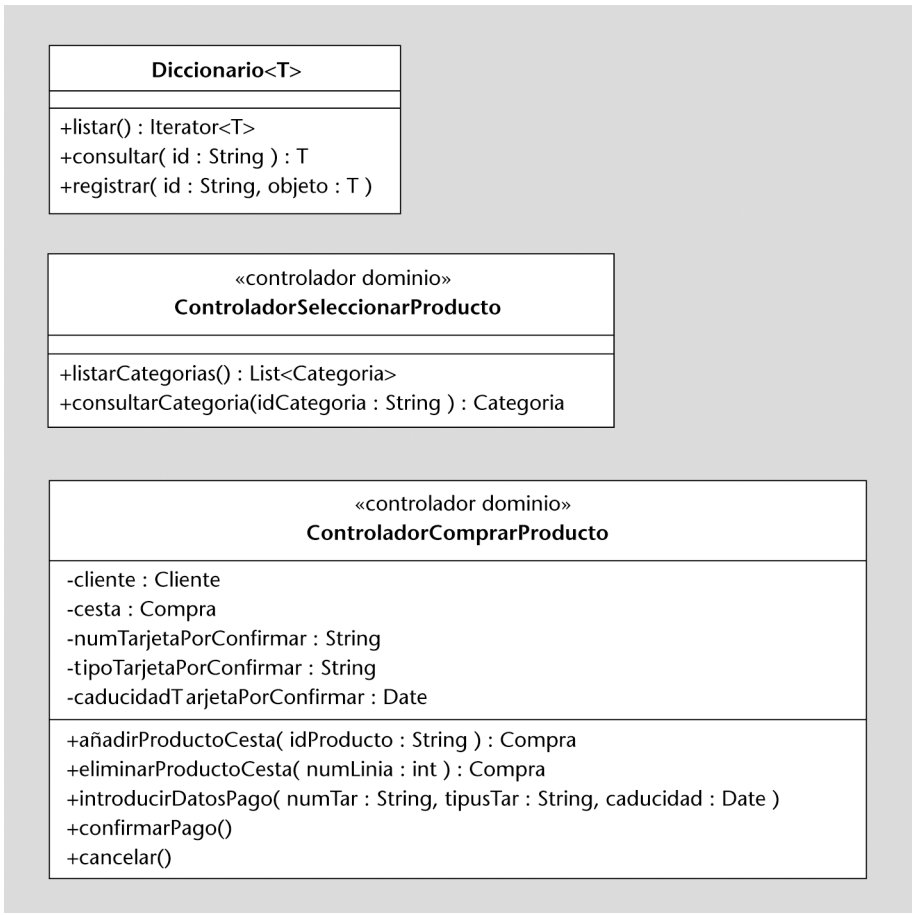
5.3. Diseño estático resultante

A medida que hemos ido haciendo el diseño del comportamiento para los casos de uso, hemos añadido responsabilidades a las clases software del modelo conceptual y nuevas clases como los controladores y las fabricaciones puras. El diagrama estático del diseño, por lo tanto, se tiene que actualizar con los cambios que se han ido introduciendo.

A continuación incluimos el diagrama estático del diseño resultante. Hay que tener en cuenta, sin embargo, que no es completo, ya que no hemos hecho el diseño de todos los casos de uso.



Por otra parte, las clases adicionales que hemos descubierto a lo largo del proceso de diseño son las siguientes:



5.4. Servicios técnicos

Durante el diseño de la capa de dominio hemos ido detectando ciertas responsabilidades que hemos querido delegar en la capa de servicios técnicos. Concretamente, las responsabilidades de persistencia y de mensajería electrónica.

5.4.1. Subsistema de persistencia

Para minimizar el acoplamiento entre las clases de la capa de dominio y el subsistema de persistencia, hemos decidido aplicar el patrón de diseño Fachada. Como resultado de eso crearemos una nueva clase en la capa de dominio que representará todo el subsistema de persistencia: BDEMusica.

Ved también

Podéis ver más información sobre el patrón de diseño Fachada en el apartado 6.2, "Fachada (*Facade*)", del módulo "Catálogo de patrones".

También aplicaremos el patrón Servidor abstracto de manera que BDEMusica no será una clase de la capa de dominio que llame a las clases del subsistema de persistencia sino una interfaz que este subsistema implementará para la capa de dominio. Así pues, los diccionarios y los controladores utilizarán las operaciones de BDEMusica para acceder al subsistema de persistencia.

Ved también

Podéis ver más información sobre el patrón de diseño Servidor abstracto en el apartado 6.8.7 del módulo "Catálogo de patrones".

Nos queda, sin embargo, una cuestión pendiente: ¿cómo reciben los diccionarios y controladores la instancia concreta de BDEMusica? Podemos aplicar el patrón de arquitectura Inyección de dependencias para solucionar este problema de manera global en toda la capa de dominio. De esta manera podríamos modificar la implementación de este subsistema sin modificar el código de los controladores y diccionarios.

Ved también

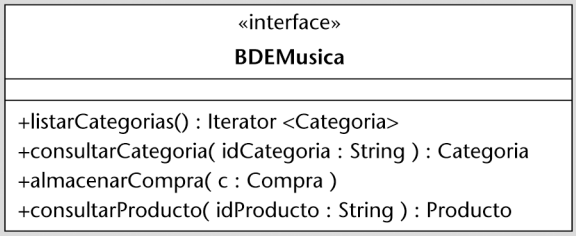
Podéis ver más información sobre el patrón de arquitectura Inyección de dependencias en el apartado 4.2 del módulo "Catálogo de patrones".

Diseño de la interfaz del subsistema de persistencia

En nuestro caso, las necesidades de persistencia que se han detectado son las siguientes:

Controlador	Operación	Necesidad
SeleccionarProducto	listarCategorias	Obtener un listado con todas las categorías del sistema por medio del diccionario.
SeleccionarProducto	consultarCategoria	Consultar una categoría a partir de su identificador. Obtener los productos de una categoría. Obtener las subcategorías de una categoría.
ComprarProducto	añadirProducto	Almacenar la Compra con las modificaciones hechas (creación de la LineaCompra asociada al producto). Consultar un producto a partir de su identificador.
ComprarProducto	introducirDatosPago	-
ComprarProducto	confirmarPago	Almacenar los cambios producidos en la compra (destrucción de estado Cesta y creación del estado Pedido).
ComprarProducto	eliminarProductoCesta	Almacenar los cambios producidos en la Compra (eliminación de una LineaCompra).

Por lo tanto, las operaciones que necesitamos del subsistema de persistencia son las siguientes (a veces diferentes necesidades se pueden resolver con una misma operación):



6. Diseño de la capa de servicios técnicos

6.1. Diseño del modelo lógico de la base de datos

El diseño del modelo lógico y físico de la base de datos se hará a partir del análisis que ya se ha realizado de los datos almacenados por nuestro sistema. Por lo tanto, partiremos del diagrama estático de análisis que ya hemos utilizado anteriormente para el diseño de la capa de dominio.

Nuestra aplicación tendrá que almacenar los datos persistentes en una base de datos. El paradigma de base de datos más utilizado actualmente, y el que utilizará eMusica, es el de las bases de datos relacionales. Pero una base de datos relacional no permite representar clases, asociaciones, generalizaciones y otros conceptos del modelo conceptual. En lugar de estos elementos, disponemos del concepto de relación.

Por lo tanto, en este punto, es necesario hacer un diseño del modelo relacional a partir del modelo conceptual. Decidiremos, pues, cómo se estructurará, en la base de datos, la información que tenemos que guardar. Es lo que en el mundo de las bases de datos se conoce como *modelo lógico de la base de datos*.

Hemos decidido que todas las clases tienen que ser persistentes y que para todas ellas (excepto las clases asociativas) utilizaremos sustitutos de la clave primaria. Hay que tener especial cuidado con los atributos que son opcionales y los que no lo son según se haya definido en el modelo conceptual.

A continuación se indica la estructura de tablas relacionales resultante de este proceso. Utilizaremos el subrayado para indicar las claves primarias y la negrita para indicar campos obligatorios (NOT NULL).

La clase categoría tiene una asociación reflexiva que se transformará en una clave foránea:

```
Categoria(id, nombre, imagen, supercategoria)
{supercategoria} es clave foránea a Categoria(id)
```

El producto tiene una asociación con Categoría que se convertirá en una tabla intermedia:

```
Producto(id, nombre, descripcion, imagen)
{cast} es clave foránea a Categoria(id)
PerteneceA(idProducto, idCategoria)
{idProducto} es clave foránea a Producto(id)
{idCategoria} es clave foránea a Categoria(id)
```

La restricción de que un producto tiene que pertenecer, como mínimo, a una categoría no queda reflejada en este modelo lógico y, por tanto, habrá que controlarla desde la capa de dominio.

La clase Cliente tiene asociaciones que se representarán en otras relaciones:

```
Cliente(id, email, nombreUsuario, contraseña, nombre, apelli-
dos, telefono)
```

La valoración es una clase asociativa que, por lo tanto, se diseñará de la manera siguiente:

```
Valoracion(producto, cliente, puntuación)
{producto} es clave foránea a Producto(id)
{cliente} es clave foránea a Cliente(id)
```

La restricción identificada en el análisis no queda reflejada en este modelo lógico y, por tanto, también habrá que controlarla desde la capa de dominio.

La compra y sus subclases se transformarán utilizando una tabla para cada clase de la jerarquía:

```
Compra(id, cliente)
{cliente} es clave foránea a Cliente
Pedido(id, numero, numTarj, tipoTarj, cadTarj, fechaPago)
{id} es clave foránea a Compra
Cesta(id)
{id} es clave foránea a Compra
```

Las líneas de compra tienen una asociación con la Compra y una con el Producto; las dos se transformarán en claves foráneas:

```
LineaCompra(compra, numLinea, producto)
{compra} es clave foránea a Compra(id)
{producto} es clave foránea a Producto(id)
```

En este caso, como hay una composición, hemos utilizado una entidad débil en vez del substituto de la llave primaria.

El precio no tendrá claves añadidas:

```
Precio(id, cantidad, moneda)
```

El historial de precios es una ternaria. En ésta, la fecha no es una clase sino un tipo y, por lo tanto, no le corresponderá ninguna tabla de la base de datos. El diseño resultante será el siguiente:

```
PrecioVigente(producto, fechaInicio, precio, fechaFin)
{producto} es clave foránea a Producto(id)
{precio} es clave foránea a Precio(id)
```

6.2. Diseño del subsistema de persistencia

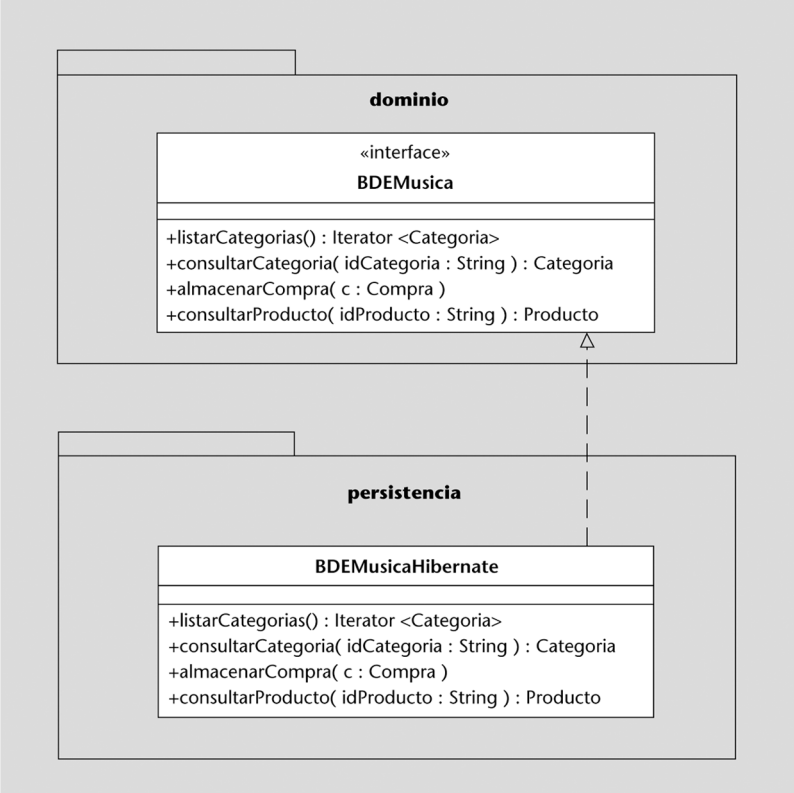
Para diseñar las clases de persistencia tenemos que diseñar una implementación de la interfaz BDEMusica que ofrezca en la capa de dominio los servicios que ésta necesita de nuestro subsistema.

En nuestro caso, hemos decidido basar la persistencia en el *framework* Hibernate. Eso nos facilitará enormemente la tarea del diseño, ya que será el *framework* quien se encargue de conectarse a la base de datos, crear las consultas SQL, detectar los cambios que se han producido en memoria, etc.

Ved también

Podéis encontrar más información sobre Hibernate en el apartado 3.4. "Un ejemplo de *framework*: Hibernate" del módulo "Introducción a los patrones".

Como la interfaz a implementar es muy sencilla y buena parte de la tarea de persistencia la implementará Hibernate por nosotros, hemos decidido que la capa de persistencia sólo constará de una clase que llamaremos *BDEMusicaHibernate* y que implementará la interfaz previamente definida.



Mostraremos el diseño detallado de esta capa mediante código Java[1]:

(1)Se ha omitido el tratamiento de excepciones.

```
public class BDEmusicaHibernate extends BDEmusica {
  public Iterator <Categoria> listarCategorias() {
    Session session = HibernateUtil.currentSession();
    Transaction tx = session.beginTransaction();
    Query query = session.createQuery("from Categoria");
List<Categoria> resultado = new LinkedList<Categoria>();
    for (Iterator it = query.iterate(); it.hasNext();) {
      Categoria c = (Categoria) it.next();
      resultado.add(c);
    }
    tx.commit();
    HibernateUtil.closeSession();
    return resultado.iterator();
  }
public Categoria consultarCategoria(String id) {
    Session session = HibernateUtil.currentSession();
    Transaction tx = session.beginTransaction();
    Categoria result =
      (Categoria) session.load(Categoria.class, id);
    tx.commit();
    HibernateUtil.closeSession();
    return result;

  }
```

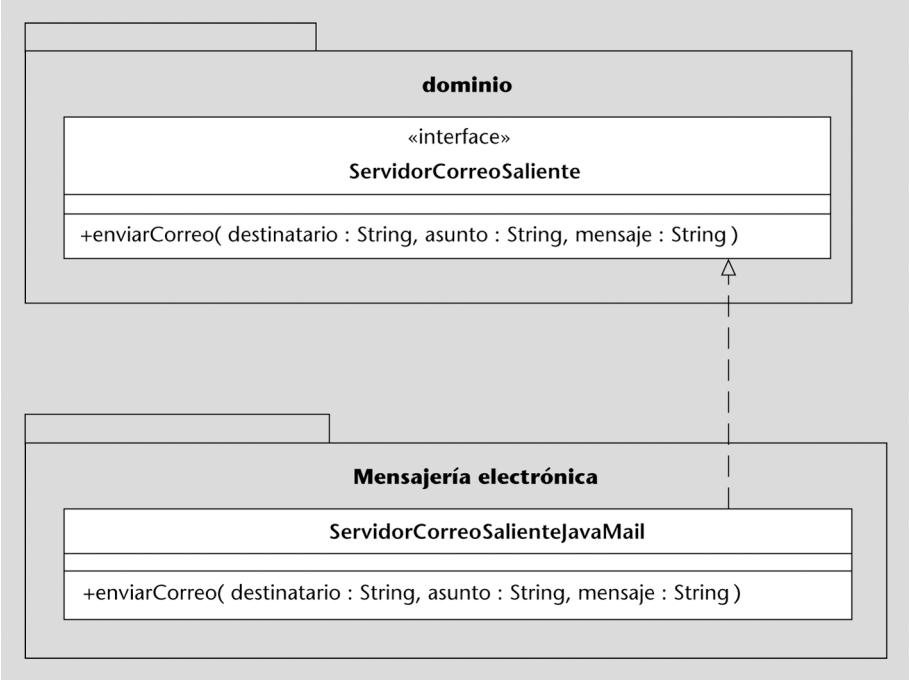
```
  public void almacenarCompra(Compra c) {
    Session session = HibernateUtil.currentSession();
    Transaction tx = session.beginTransaction();
    session.update(c);
    tx.commit();
    HibernateUtil.closeSession();
  }

public Producto consultarProducto(String id){
   Session session = HibernateUtil.currentSession();
   Transaction tx = session.beginTransaction();
   Producto result =
     (Producto) session.load(Producto.class, id);
   tx.commit();
   HibernateUtil.closeSession();
   return result;
  }
}
```

Para tener un diseño completo, ahora, habría que definir la correspondencia objeto-relacional entre las clases de nuestro modelo del dominio que tienen que ser persistentes y las tablas de la base de datos. Esta correspondencia se definiría mediante un descriptor de Hibernate en XML.

6.3. Diseño del subsistema de mensajería electrónica

En el apartado 5.2.6, "Confirmar pago", de este módulo, hemos visto que, en el momento de confirmar un pago, se tenía que enviar un mensaje electrónico al usuario. Para enviarlo hemos introducido la interfaz ServidorCorreoSaliente. El subsistema de mensajería electrónica sólo tiene que implementar esta interfaz:


dominio
«interface»
ServidorCorreoSaliente
+enviarCorreo(destinatario : String, asunto : String, mensaje : String)
Mensajería electrónica
ServidorCorreoSalienteJavaMail
+enviarCorreo(destinatario : String, asunto : String, mensaje : String)

Resumen

A continuación se incluye una tabla con los patrones aplicados durante el análisis y diseño del caso práctico:

Patrón	Apartado	Aplicación
Cantidad	2.1	Precio de los productos
Asociación histórica	2.2	Precio de los productos
Arquitectura en capas	3.1	Arquitectura del sistema
Modelo-vista-controlador	3.2	Arquitectura de la capa de presentación
Controlador	3.2.1	Comunicación entre la capa de presentación y la capa de dominio
Método plantilla	4.2.2, 4.2.3	Diseño de los controladores
Estado	5.1.4, 5.2.4	Especialización dinámica de Compra
Fabricación pura	5.2	Diccionarios
Experto	5.2	Diseño de las clases de la capa de dominio
Creador	5.2.4	Creación de LineaCompra
Fabricación pura	5.2.6	Envío de mensajes de correo
Fachada	5.4.1	Comunicación entre la capa de dominio y el subsistema de persistencia
Servidor abstracto	5.4.1, 5.2.6	Fachada del subsistema de persistencia Servidor de correo saliente
Inyección de dependencias	5.4.1	Obtención de la implementación de la fachada de persistencia

También se incluye un resumen de los *frameworks* utilizadas durante el diseño de eMusica:

Framework	Apartado	Aplicación
Java Servlets	3.2	Capa de presentación
Hibernate	6.2	Subsistema de persistencia

Ejercicios de autoevaluación

1) En el diseño actual, la operación invocarOperacionesDominio debe tratar todos los acontecimientos que pueda recibir este controlador. Nos preocupa que el código de las implementaciones de esta operación crezca excesivamente y queremos poder tratar cada uno de estos acontecimientos de presentación como un objeto, de modo que el controlador sólo tenga que decidir qué objeto llamar dependiendo del acontecimiento que reciba. ¿Existe algún patrón que podamos aplicar para solucionar este problema?

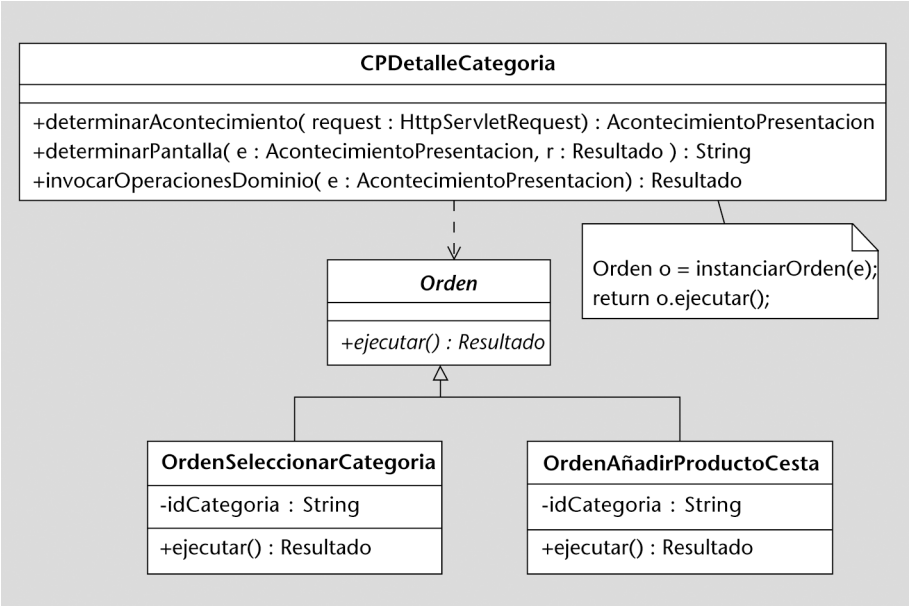
2) Durante el diseño de nuestro sistema no hemos tenido en cuenta la validación de los datos proporcionados por el usuario. Indicad a qué clase asignaríais la responsabilidad de validar la sintaxis del número de tarjeta de crédito a la hora de hacer un pago.

3) Hemos visto que la utilización de diccionarios como fabricación pura es una solución muy habitual a toda una familia de problemas. ¿Podríais describir esta utilización como un patrón? ¿Cómo?

Solucionario

1) El patrón de diseño Orden nos puede ayudar a solucionar este problema. Para aplicarlo definiremos una clase para cada posible evento que haya que tratar. Las instancias de estas clases representarán las ocurrencias de estos eventos. De ese modo, cuando el número de eventos que haya que tratar crezca, el controlador sólo tendrá que añadir nuevas clases de Orden para tratar los nuevos eventos.

Pongamos como ejemplo el controlador CPDetalleCategoria:



También podríamos considerar solucionar el problema a un nivel más general modificando la clase ControladorBase.

2) Lo primero que hay que decidir es qué capa de nuestra arquitectura ha de tratar este caso. *A priori*, tendría que ser la capa de dominio. En este caso, según el patrón Experto, tenemos cuatro candidatos: ControladorComprarProducto, Compra, Cesta y Pedido. Compra queda descartado porque en realidad lo que hace es delegar la operación compra a su estado. Para mantener la cohesión del controlador en el nivel más alto posible, asignaríamos esta responsabilidad a Cesta o a Pedido.

Por cuestiones de eficiencia, sin embargo, si tenemos en cuenta que ésta es una validación estrictamente sintáctica (y, por lo tanto, no depende del resto de datos del dominio) podríamos asignar esta responsabilidad a la capa de presentación. En este caso, para mantener la cohesión de nuestro sistema, tendría que ser el controlador de presentación quien tuviera esta responsabilidad.

3) Suponiendo que hayamos aplicado anteriormente con éxito esta solución y que hayamos constatado las consecuencias de su aplicación, sí que podríamos considerarla un patrón de diseño. En este caso, tendríamos que describir los elementos estándar de un patrón de diseño:

- Nombre
- Contexto
- Problema
- Solución
- Consecuencias

Bibliografía

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

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

[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. ed.). Madrid: Prentice Hall.

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