6 Implementación

6.1 Introducción

En este apartado vamos a analizar las funcionalidades aportadas y componentes involucrados en la implementación de los requerimientos expuestos en el capítulo de análisis de requisitos de diseño. Particularizaremos los elementos genéricos que componen la herramienta, dividiendo el estudio en las tres áreas del modelos de datos relacional.

Tanto la estructura de la aplicación, como la interacción entre sus componentes

y el modelo relacional son bastante complejas. Por tanto, en este apartado se realizará una descripción cualitativa de estos aspectos.

6.2 Estudio del modelo relacional de datos

En versiones anteriores de la aplicación, el modelo de datos contemplaba la división en las tres áreas propuestas. A pesar de esta clasificación mínima de los datos, no existía ninguna otra particularización u optimización. Tenemos así inicialmente tres tablas que poseen claras redundancias de datos y varias transitividades. Todos estos defectos del modelo natural, orientado a tupla, hacían que las lecturas y escrituras en base de datos se volvieran lentas conforme el proyecto crecía en volumen. Más importante aún, la recuperación de datos ante un fallo de la aplicación, ante una operación desatomizada, se hizo inviable.

En varios proyectos reales en los que se ha tenido la oportunidad de trabajar, se ha visto como, para modelos de datos altamente masivos, la redundancia no está prohibida. Se llega en estos casos a una solución de compromiso, dejando una mínima "cantidad de redundancia", que hace que el sistema de gestión de datos se comporte de forma más rápida. Quizás gestionar una nueva tabla y cruzarla con otras se hace más pesado que tener varias veces la misma información ocupando espacio en la base de datos.

En cualquier caso, en nuestro sistema se ha propuesto como objetivo de diseño cumplir con formas normales y canónicas en los modelos de datos. En concreto, nuestro sistema llegará a la forma de Boyce-Codd.

Por otra parte, en sus orígenes, el proyecto contemplaba la creación de diferentes usuarios de base de datos, con distintos roles y permisos. Pero debido a que el modelo resultante debe ser ejecutado en un sistema real, de un proveedor de hosting real, debemos limitar nuestro SQL y la edición de usuarios. Se contempló la edición de un *SCHEMA* por usuario, pero la realidad nos dijo que solo disponíamos de una base de datos, tablas y usuarios con perfiles simples.

En la figura 6.1 presentamos el modelo final de datos, resultado del estudio que a continuación se detalla.

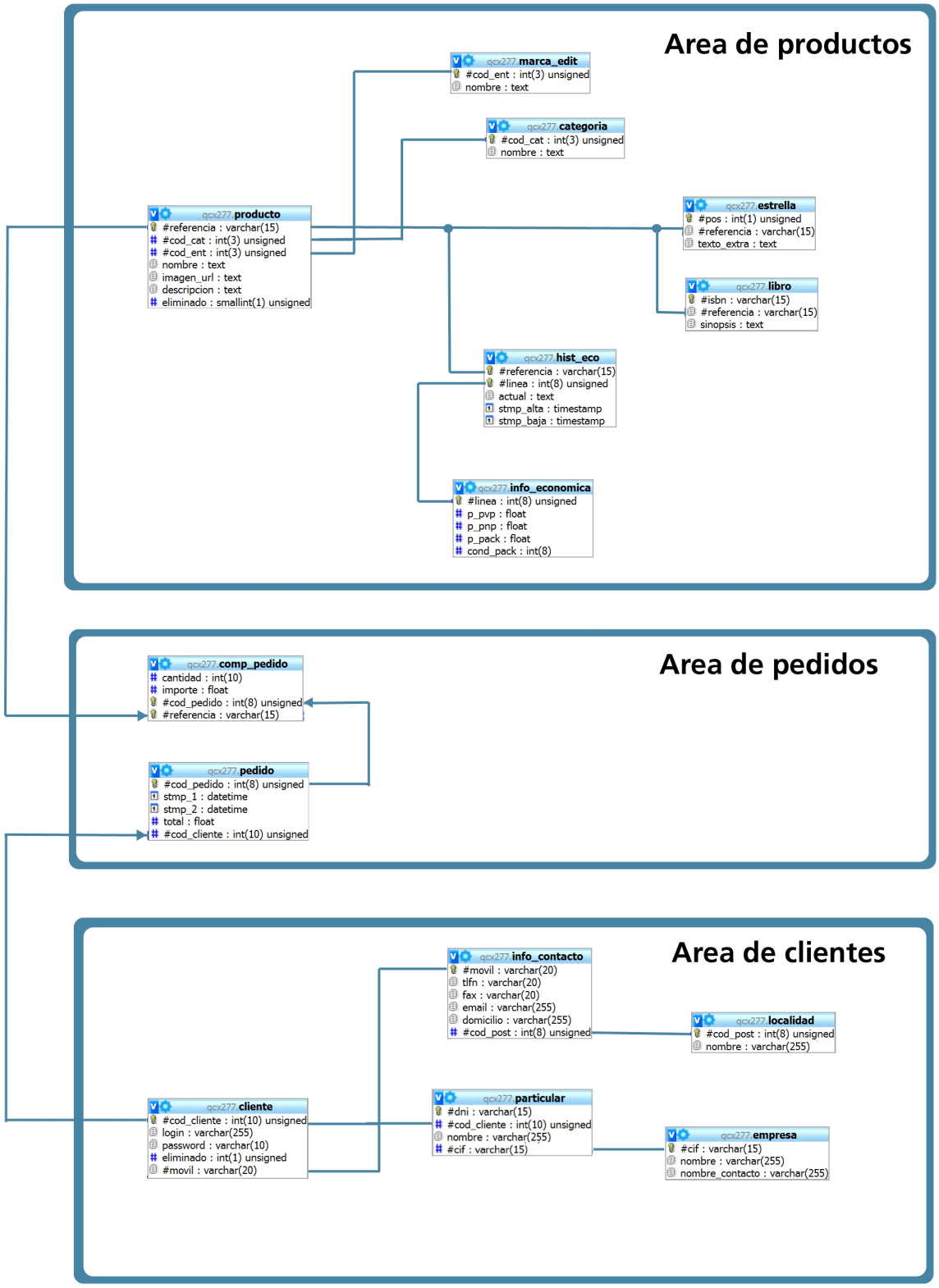


Figura 6.1: Modelo de datos relacional.

6.2.1 Modelado del área de productos

Presentamos a continuación el modelo entidad relación del área de productos, en el que se plasman los atributos de producto requeridos por el sistema para su completa definición y caracterización.

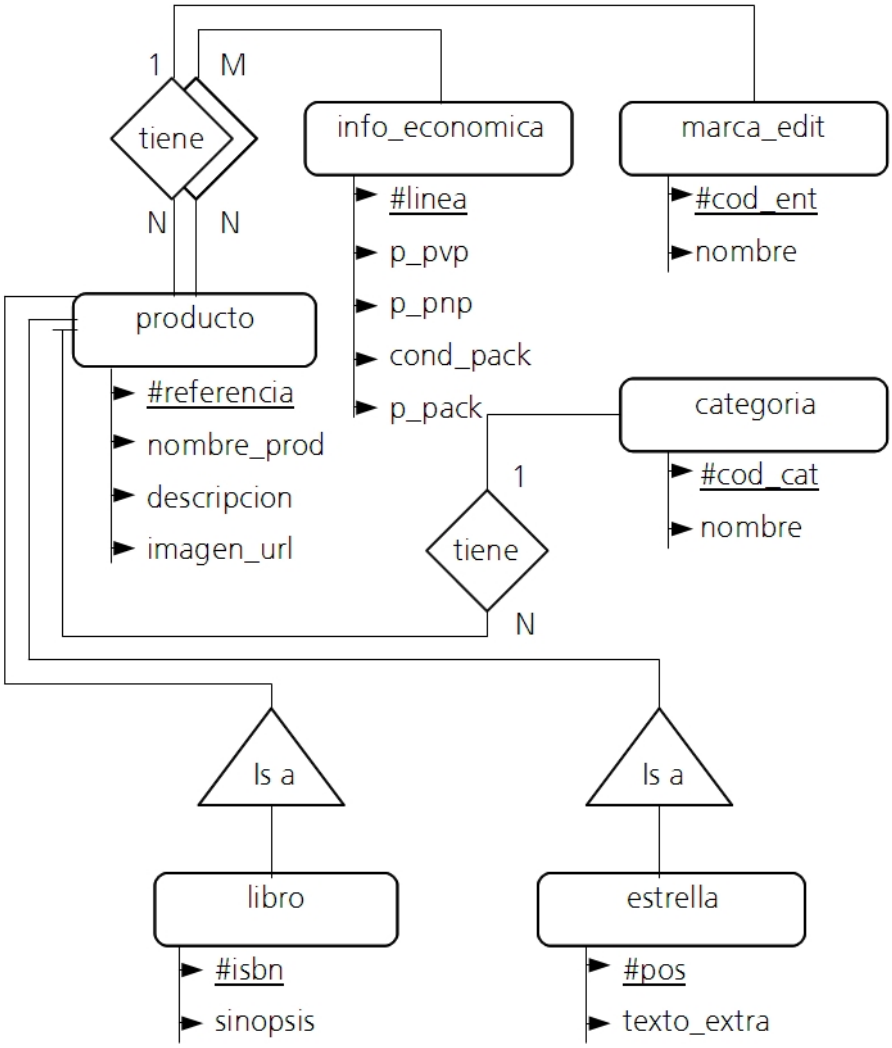


Figura 6.2: Diagrama entidad relación de producto.

Vamos a describir los atributos y entidades que no se obtienen de forma obvia y que son observables en el modelo anterior, con objeto de tener presente la semántica del sistema.

- *imagen_url*: La utilidad de este atributo es exclusiva de la aplicación web. En esta cadena de caracteres se almacenará la ruta y nombre del archivo de imagen usado en la página para presentar al producto. Sirve pues de puntero a la web. En futuras versiones de la aplicación KnoE, se podría contemplar el

uso de esta variable para realizar una vista previa de la imagen cargada. Reseñamos en este punto que tendrá un valor inicial por defecto, que apunta a una imagen del tipo "imagen no disponible", típica en toda web. La forma de uso de "*imagen_url*" es la siguiente: desde KnoE, en la inserción de producto, se especifica la imagen que será subida por un minicliente FTP. La cadena insertada en la aplicación será completada con la ruta de directorios usada en el servidor de hosting para almacenar dicho fichero. Posteriormente, la cadena está disponible al código PHP de la web, para ser usada de forma dinámica.

- *linea*: este parámetro será un "*auto_increment*" y por tanto, su valor, será transparente al usuario. Referencia una "linea de productos", concepto que se explica en el apartado dedicado al estudio de dependencias funcionales.
- *p_pvp*: precio de venta al público.
- *p_pnp*: precio de compra del producto por parte de la tienda.
- *p_pack*: precio de oferta del producto.
- *cond_pack*: número mínimo de unidades que se deben comprar para que se aplique el precio de oferta del producto. La aplicación web funcionará de la siguiente forma: "si el número de unidades solicitadas es mayor que *p_pack*, se aplicará *p_pack*; si no, se usará *p_pvp*".
- *marca_edit* es la marca o editorial asociada al producto.
- *categoria*: es una clasificación de productos, de forma que crea un catálogo. Los valores de este campo van del 1 al 66, y son inicializados en el proceso de instalación del paquete de aplicaciones, mediante una batería de sentencias SQL.
- *#cod_cat* y *#cod_ent* son "*auto_increment*" gestionadas totalmente por la aplicación KnoE y, por tanto, transparentes al usuario.
- *estrella*: es una característica añadida al producto. Su utilidad es señalar un producto como oferta en la web, de forma que será presentado en dicha web de forma especial. Se ha limitado a 3 el número de dichos "productos estrella". En versiones previas del programa, este concepto no existía. Su origen es exclusivamente de presentación web.
- *pos*: indica la posición que se usará en la web para presentar el producto estrella. Este parámetro solo puede tomar los valores 1, 2 o 3, valores que serán inicializados y constantes en el sistema.

Para simplificar el estudio de dependencias funcionales y obtener un grafo de dependencias vamos a usar la notación propuesta en la tabla siguiente. Podemos observar que las especializaciones LIBRO y ESTRELLA quedan fuera de este estudio, ya que no añaden condiciones de transitividad o redundancia.

#referencia	A
nombre_prod	B
descripcion	C
image_url	D
linea	E
p_pvp	F
p_pnp	G
cond_pack	H
p_pack	I
#cod_ent	J
nombre_ent	K
#cod_cat	L
nombre_cat	M

Figura 6.3: Conversión nomenclatura.

Dependencias funcionales:

- A->ABCDEFGHIJKLM. A clave candidata, ya que la referencia de producto es un campo que se basta por sí solo para identificar unívocamente cualquier elemento de la relación.
- E->EFGHI. "linea": Existen series de productos, con distintas referencias, cuyas únicas diferencias radican en características tales como el color o diseño y, que tienen una misma información económica. De esta forma eliminamos la redundancia patente. Una vez que damos de alta una "línea de precios", esta disponible para dicha serie de productos. Como ejemplo, carpetas iguales de distintos colores, libros de una misma colección, bolígrafos de distintos colores, etc. Como inciso, la aplicación, a la hora de insertar un nuevo producto en la base de datos, deberá chequear antes de la ejecución de inserción la existencia o no de la línea económica propuesta por el usuario. Se hace, de esta forma, transparente al gestor.
- J->K. El código de entidad (marca o editorial) será suficiente para definir la marca del producto, e implica el nombre de la misma. Podría pensarse que el CIF de dicha empresa sería más adecuado. No obstante, el uso de dicho identificador no se hace necesario en este punto, y se simplifica la aplicación con la solución elegida, ya que el atributo *#cod_ent* será un "*auto_increment*", haciendo que su valor se convierta en transparente al gestor.
- L->M. El código de categoría, al igual que lo expuesto en el parrafo anterior, se hace suficiente para definir la categoría. La aplicación, al igual que en el caso anterior, será la encargada de gestionar estos valores.

En la siguiente figura se muestra el grafo de dependencias funcionales.

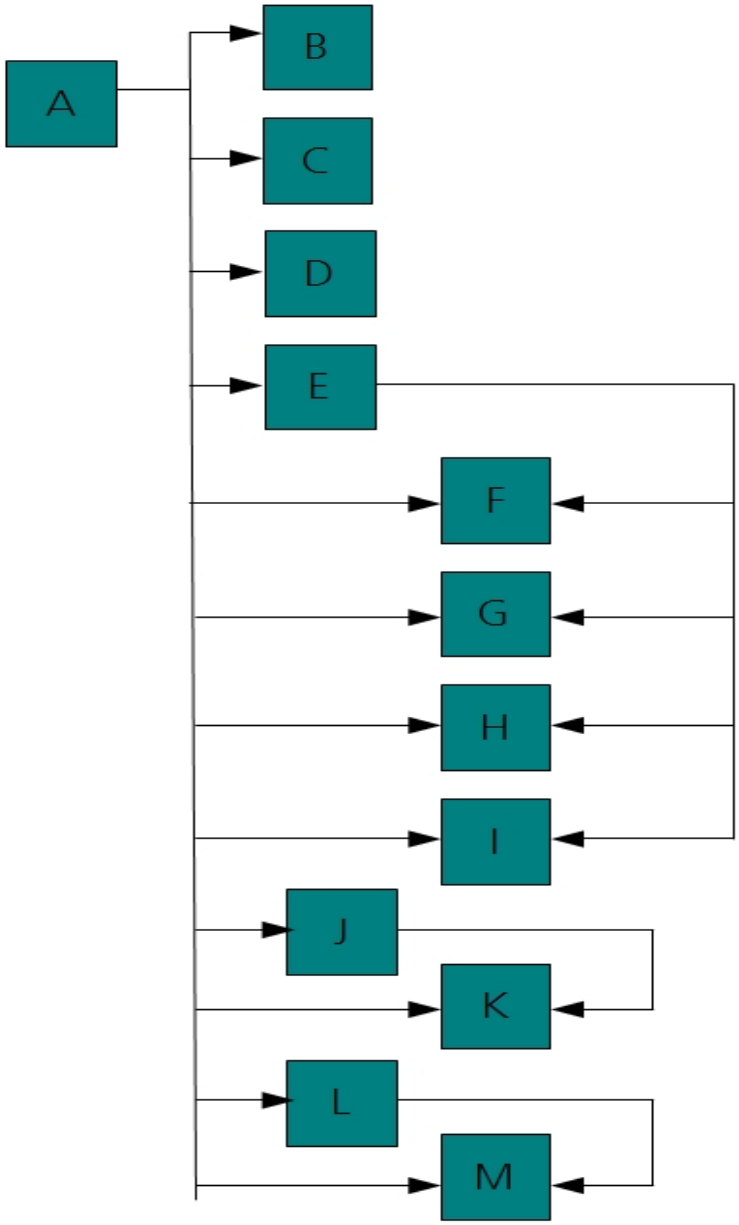


Figura 6.4: Grafo de dependencias inicial (producto).

Observamos que el sistema está en FN2. En el grafo siguiente se muestra la estructura final en FNBC.

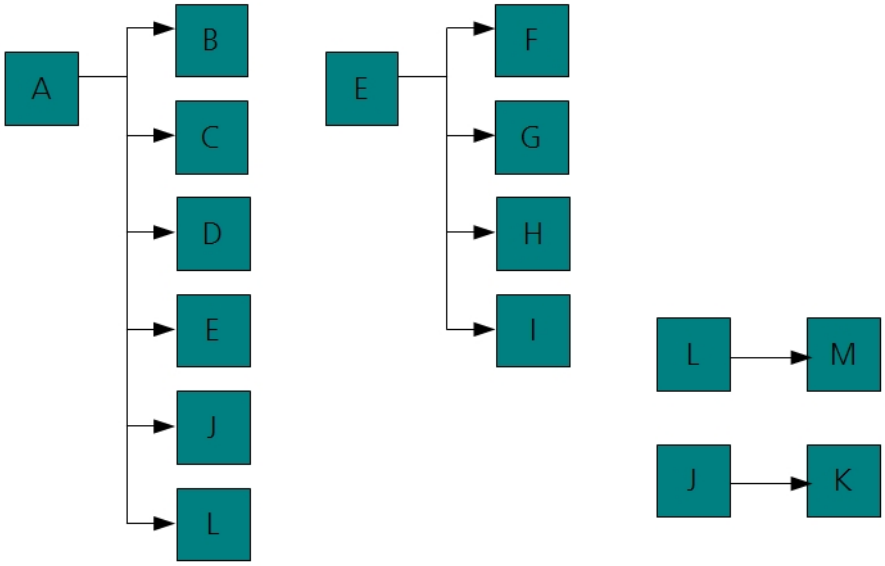


Figura 6.5: Grafo de dependencias final(producto).

En la figura siguiente se muestra el modelo del área de productos final, resultado del proceso de síntesis anterior.

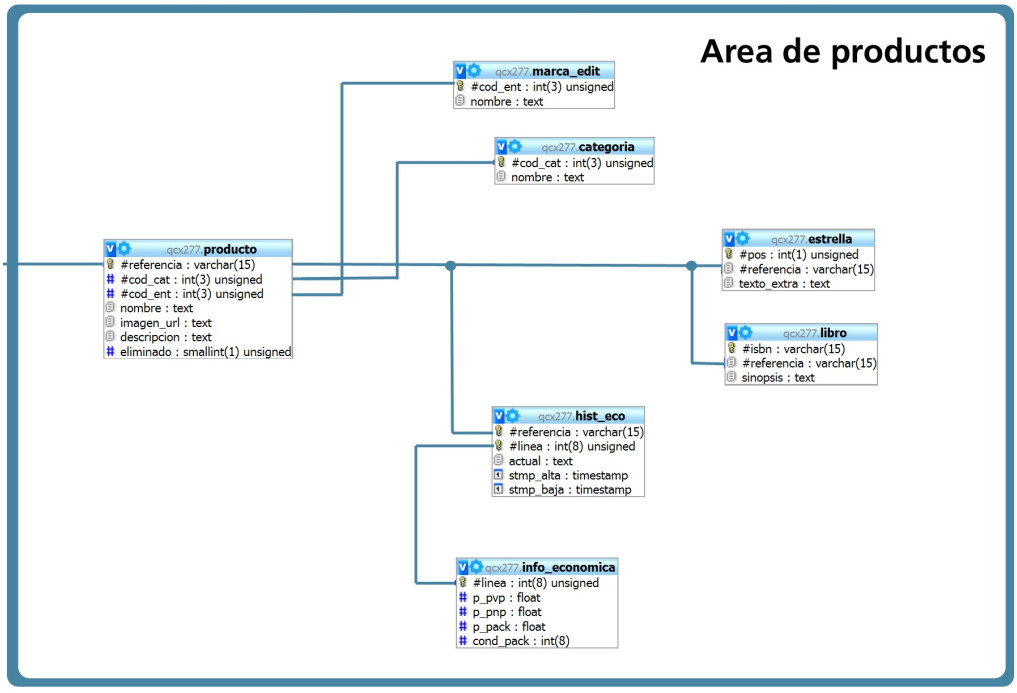


Figura 6.6: Modelo final del área de producto.

6.2.2 Modelado del área de clientes

Vamos, en primer lugar, a hacer el estudio de dependencias entre atributos del modelo entidad relación. Debemos señalar que se harán una serie de suposiciones que nos ayudarán a plasmar la realidad en nuestro modelo.

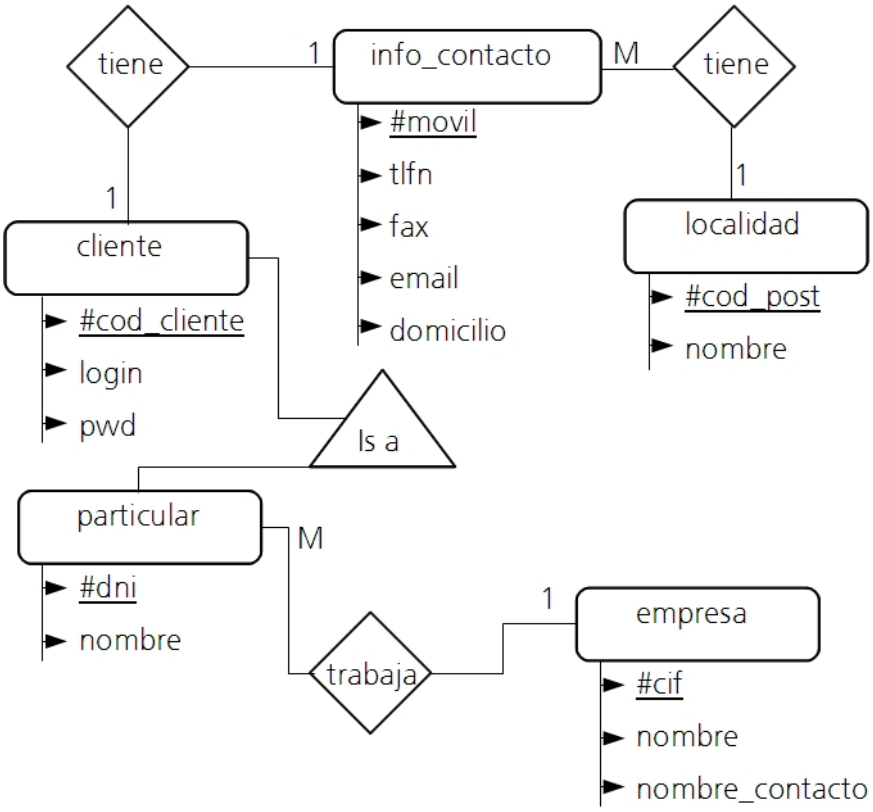


Figura 6.7: Diagrama entidad relación de cliente.

En la figura 6.7 se puede observar el conjunto de entidades y atributos. Explicamos a continuación los que nos se obtienen de forma obvia.

- *#cod_cliente*: es un "*auto_increment*" y su valor se incrementa desde la aplicación web con cada alta de un nuevo usuario. Este valor entero le será de utilidad al propio cliente web, ya que se le requerirá a la hora de confirmar cualquier compra, además de su password de cliente.
- *login* y *pwd* son las variables cadena que contienen los datos de acceso web del cliente.
- En la entidad "*empresa*", el atributo *"nombre"* se corresponde con el nombre de la entidad, mientras que "*nombre_contacto*" es la persona de contacto que responde a la empresa a dar de alta.

Para simplificar la notación en el estudio de dependencias funcionales nos serviremos de la nomenclatura propuesta en la tabla 6.8

#cod_cliente	A
login	B
pwd	C
tlfn	D
fax	E
email	F
movil	G
domicilio	H
#cod_postal	I
localidad	J

Figura 6.8: Conversión nomenclatura.

- A, clave candidata: podríamos pensar que es estrictamente necesario que el "login" sea único, pero no es así ya que, sirviéndonos del código de cliente, identificamos de forma unívoca al cliente. Independientemente de este hecho, la aplicación web restringirá el espacio, de forma que dicho "login" sí sea único. A->ABCDEFGHIJ.
- Tanto el teléfono fijo "*tlfn*", como el "*fax*" o "*email*" podrían pertenecer a una empresa, pudiendo en este caso, existir varios usuarios que usen los mismos valores de atributos. Suponemos que el móvil es personal e intransferible, de aquí que G->DEFGHIJ.
- Un código postal define una localidad. No ocurre que "*domicilio*" determine un código postal, ya que pueden existir calles igualmente nombradas en distintas ciudades. I->J.

Según estas condiciones, obtenemos el grafo de dependencias mostrado en la figura 6.9. En dicho grafo podemos observar que el sistema se encuentra en la

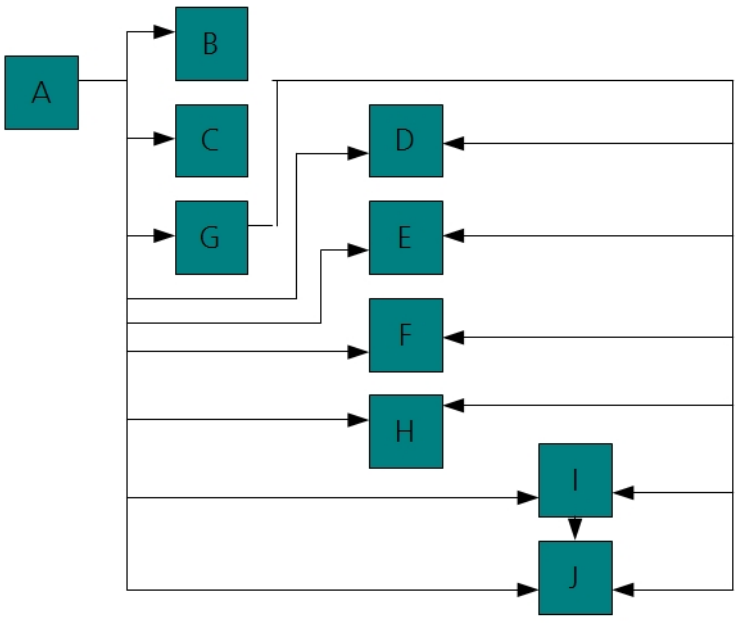


Figura 6.9: Grafo de dependencias inicial (cliente).

segunda forma normal, siendo aún necesario depurar el sistema de tablas para

eliminar redundancias y transitividades.

Realizamos los siguientes cambios sobre el esquema anterior, quedando el modelo del área de cliente en forma normal de Boyce-Codd, tal y como se muestra en la figura siguiente.

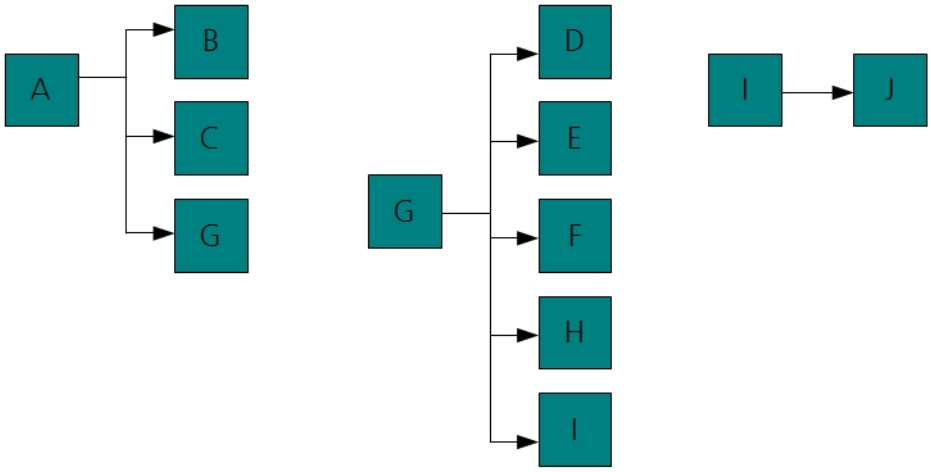


Figura 6.10: Grafo de dependencias final (cliente).

Se puede observar que hemos dejado fuera del estudio la especialización CLIENTE-EMPRESA-PARTICULAR, ya que no añadirá redundancia o transitividad alguna al modelo final. Para llevar a cabo esta especialización usaremos la forma estándar descrita en varios manuales. Según el método de diseño 1, implicaría crear una tabla por entidad y la más general de todas (cliente) mantendría sus atributos y clave primaria, y la especialización heredaría la clave superior. Explicamos como quedaría el segmento del modelo:

- Crearemos una tabla por entidad.
- La entidad *cliente* permanece inalterada, sin agregar atributo o clave alguna.
- La entidad *particular* añade el código de cliente (especialización) y el CIF de empresa.
- La entidad *empresa* permanece inalterada.

Esta realización implica varios conceptos:

- Todo cliente tendrá una empresa asociada. En caso de que el cliente no tenga/pertenezca a ninguna empresa, se le asignará una *"empresaPorDefecto",* que mantendrá la integridad de los datos. Esta asociación se hace mediante la clave externa *cif* en la entidad *particular***.**
- En la entidad *particular*, a pesar de que la clave primaria "*#dni*" es suficiente,

disponemos de otro identificador fuerte "*#cod_cliente*", que sirve para referenciar la tabla desde la entidad superior *cliente.*

Por otro lado, y con posterioridad al estudio, surgió la necesidad de incluir un nuevo atributo de cliente, "*eliminado*". Este atributo no añade ninguna redundancia al esquema resultante de este estudio y tampoco referencia a otra variable de forma transitiva. El significado de este nuevo atributo es bien claro: surge la necesidad desde la aplicación de poder eliminar un usuario dado de alta. Esto supone un serio riesgo para la integridad de la información dentro del esquema total, ya que el usuario a eliminar podría estar implicado en algún pedido. Por definición y requerimientos de diseño, nuestro sistema tiene que ser capaz de almacenar un histórico de pedidos. Por tanto, la eliminación de un usuario se puede definir de forma que no implique un borrado de información en tablas. Desde ahora, la eliminación de usuario consistirá en un borrado real de tabla si y sólo si no se encuentra implicado en un pedido. De otra forma, el sistema actualizará la variable "*eliminado*" a 1, indicando a todos los procesos del sistema que el usuario, aunque presente, ya no está activo.

En la figura 6.11 podemos ver el modelo de datos, resultado final de este estudio, en el área de clientes.

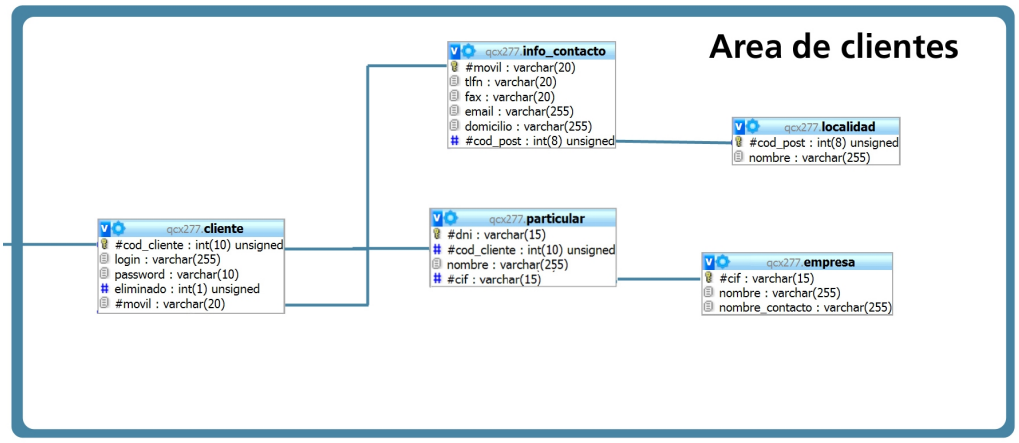


Figura 6.11: Modelo final del área de cliente.

6.2.3 Modelado del área de pedidos

En la siguiente figura se presenta el modelo entidad relación del área de pedidos. Este modelo se compone exclusivamente de una única entidad "*pedido*", explicada en párrafos posteriores, y dos relaciones.

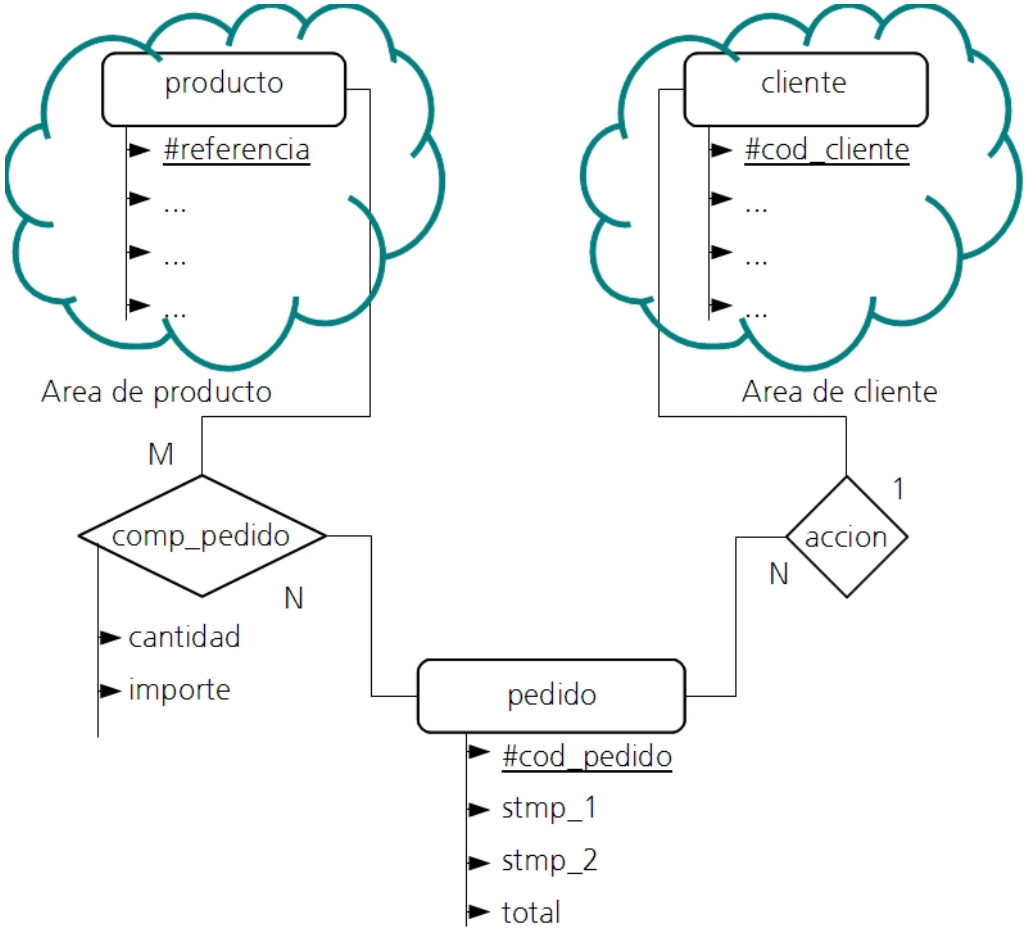


Figura 6.12: Diagrama entidad relación de pedido.

La entidad *pedido* se compone de un identificador de pedido *#cod_pedido*. Esta variable será un *auto_increment,* gestionado por la aplicación y el sistema gestor de base de datos, y por tanto, transparente al cliente y usuario de KnoE. Los atributos *ts_1* y *ts_2* son *timestamp* (marcas de tiempo). La forma de usarlos es la siguiente:

- Al dar de alta un nuevo pedido (proceso realizado por el cliente desde la web), se crea un asiento en la base de datos con un nuevo *#cod_pedido*, la marca de tiempo de realización en *ts_1* y valor nulo para *ts_2*.
- La aplicación KnoE, para buscar pedidos pendientes de ser ejecutados, buscará registros de pedidos con un valor no nulo para *ts_1* y nulo para *ts_2*"
- Cuando un pedido es preparado y enviado (ejecutado), se actualiza el valor de *ts_2*. Existirá en Knoe un botón de actualización de pedidos ejecutados. Desde este botón se iniciará la rutina de actualización del timestamp *ts_2*.
- KnoE, para crear una vista de histórico de pedidos, buscará asientos con valores no nulos en *ts_2*.

Por otra parte, la relación *comp_pedido* (muchos a muchos) indica el detalle de

pedido. Un pedido, con su valor particular de *#cod_pedido*, referencia a varios productos. Cada producto referenciado de esta forma debe tener una *cantidad*.

En la figura 6.13, se muestra el modelo final del área de pedidos.

Figura 6.13: Modelo final del área de pedidos.

6.3 Implementación de la aplicación

En la figura 6.14 se ve, de forma muy generalizada, una estructura de capas correspondiente a las interacciones entre los distintos paquetes de clases que conforman la aplicación completa. Vamos, a continuación, a pormenorizar los detalles de la pila expuesta, explicando desde la capa superior la forma de lanzar las rutinas más usuales dentro del programa diseñado. Obviaremos algunas de estas rutinas, pues la forma de proceder será muy similar entre las distintas áreas definidas (producto, cliente y pedido).

Mencionamos en este punto que la opción secuencial por la que nos hemos decantado a la hora de lanzar los distintos procesos no es la única. De hecho, en versiones anteriores la forma de proceder era bien distinta. En sus inicios, el modelo de base de datos diseñado únicamente ejecutaba la inserción/eliminación de una fila en la tabla principal, y la generación de vistas se reducía a la selección de columnas completas. No existía solución de atomicidad en la versión previa a la presente, concepto en el que ahondaremos en otro apartado.

Un aspecto importante de toda la aplicación es la forma en que ejecuta sentencias SQL *insert*/*update*. En versiones anteriores de KnoE, se ejecutaban inserciones/actualizaciones individuales, con select's intermedios, y de forma aislada. Tras ahondar en la documentación de MySQL, se obtuvo el dato importante de que un conjunto de sentencias *insert*/*update* enviadas en un mismo bloque componen una unidad atómica. Así, si se produce una violación de la integridad de datos tras haber realizado alguna inserción/actualización, el SGBD de MySQL ejecuta un *rollback* automático.

Es por esto que se decidió en esta versión de KnoE cambiar radicalmente la filosofía de escritura, haciendo una recopilación previa de datos y evaluación de los mismos, con el objeto de obtener una o varias sentencias de escritura finales. Estas serán enviadas al SGBD de forma conjunta. Además de esta condición de seguridad añadida de forma externa, las actualizaciones son un bloque transaccional. Se inicia toda transacción desactivando el sistema *autocommit*, y se gestionan constantemente las directivas *START TRANSACTION, COMMIT* y *ROLLBACK*.

6.3.1 Estructura de la aplicación

Una aproximación a la estructura de la herramienta podría ser la representada en la figura 6.14. En ella se observan tanto las distintas entidades que componen la aplicación como sus dependencias.

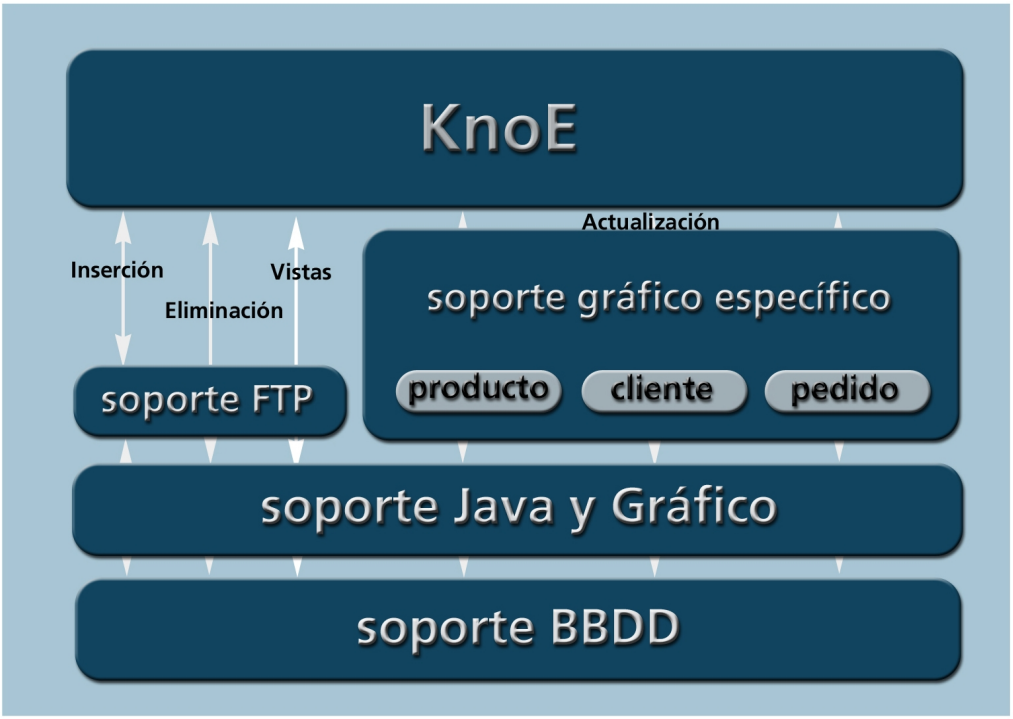


Figura 6.14: Estructura de la aplicación KnoE.

En el siguiente apartado se explicará el árbol de clases existente y la forma de instanciación de dichas clases para la creación de las rutinas básicas.

Anteriormente, se han comentado brevemente los mecanismos de seguridad frente a errores que implementa el sistema creado.

6.3.2 Área de producto

En este apartado, vamos a dar forma a los métodos y rutinas necesarios para dotar al sistema de las funcionalidades requeridas sobre el área de productos del modelo relacional de datos.

Recordamos que los requerimientos analizados en su correspondiente apartado fueron:

- Extracción de datos de productos y visualización de los mismos.
- Modificación ordenada de los componentes del área de producto.
- Eliminación de productos, sin violar reglas que lleven al modelo de datos a inconsistencias.

Basándonos en la arquitectura de la aplicación propuesta, la generación y presentación de las vistas se obtendrán de la interacción directa entre la capa final y los paquetes de soporte Java, FTP, gráfico y de base de datos. Por contra, para la actualización de estados en tablas se interpondrá entre estas capas el paquete de apoyo knoe.soporteGrafico.producto.

6.3.2.1 Vistas de producto

Una vista es usada generalmente para navegar a través de información, abrir un editor o mostrar las propiedades de un elemento de un editor. En contraste con un editor, el cual tiene un ciclo de vida del tipo abrir-modificar-guardar-cerrar, los cambios realizados en una vista son guardados inmediatamente.

El proceso de selección y muestra de información se explica a continuación. En primer lugar, en el área de productos, seleccionamos el "tabbedPane"

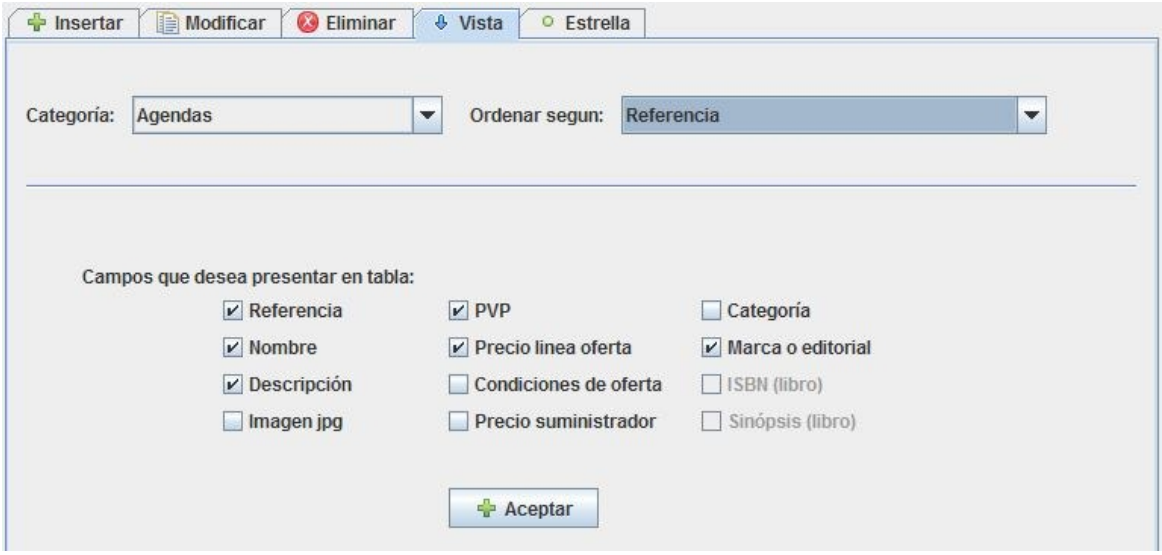


Figura 6.15: Selección de campos a generar en la tabla de productos.

correspondiente.

En esta vista, podemos seleccionar los campos que se desean generar en la tabla, las distintas categorías de producto y el criterio de ordenación de datos.

Se agrega a la vista el limitador de categoría de producto para evitar presentar todos los productos en una sola vista.

Los productos presentados en la tabla serán exclusivamente ordenados por referencia, nombre o marca.

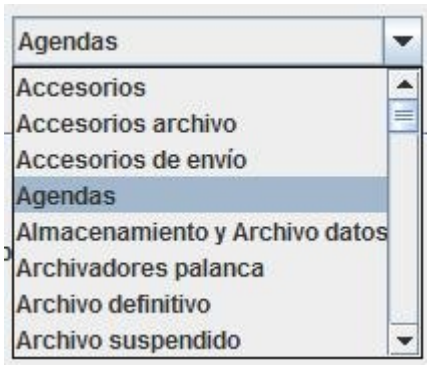


Figura 6.16: Limitación categorías de producto.

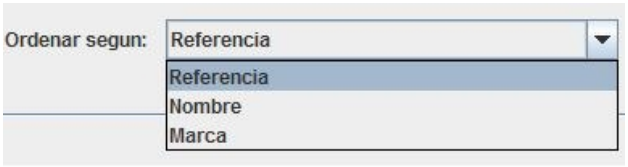
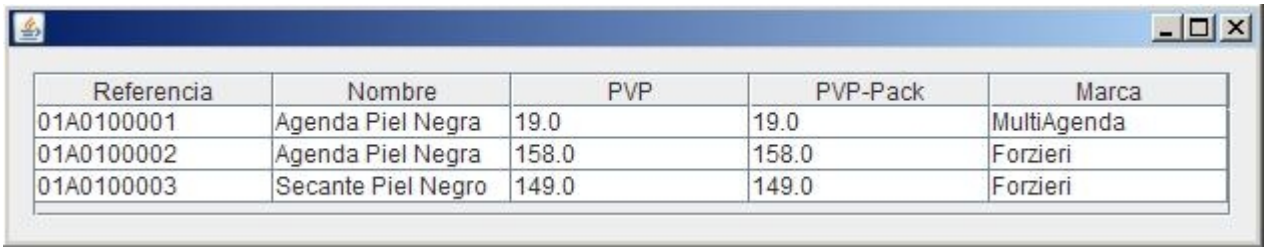


Figura 6.17: Criterios de ordenación.

Todos los campos seleccionados y elementos modificadores de la vista (criterio de ordenación y categoría de producto) son gestionados por la clase superior de la aplicación para ser pasados como parámetros a las clases de soporte de base de datos, soporte de Java y soporte gráfico.

Como resultado de la creación de la vista, se generará una tabla que contiene la información solicitada, ordenada como se propuso.



Referencia	Nombre	PVP	PVP-Pack	Marca
01A0100001	Agenda Piel Negra	19.0	19.0	MultiAgenda
01A0100002	Agenda Piel Negra	158.0	158.0	Forzieri
01A0100003	Secante Piel Negro	149.0	149.0	Forzieri

Figura 6.18: Tabla resultado.

Secuencia

- KnoE recolecta en la ventana activa los atributos de tabla que se desean mostrar, junto con los criterios modificadores "*categoría*" y "ordenar según" en arrays independientes. En el Texto 6.1 se muestra la secuencia de código que implementa la generación de vistas de producto.

```
ConectarQcx277 con = new ConectarQcx277();
SoporteArray soporte = new SoporteArray();
DialogoTabla dt = new DialogoTabla();
dt.setDialogoModel(new javax.swing.table.DefaultTableModel
  (con.generalSelect(con.construccionStatementProducto(campos, modificadores), campos), //Data[][]---Los
datos de la tabla
  soporte.arrayStringParaTableModelProducto(campos)));                    //String[]---Cabecera de la tabla
dt.setVisible(true);
```

Texto 6.1: Fragmento de KnoE.java: generación de vistas de productos.

- Se instancia la clase *ConectarQcx277* (paquete de apoyo *soporteBD)* para disponer de toda la funcionalidad y conectividad SQL.
- Se instancia la clase *SoporteArray* (paquete de apoyo *soporteJava*) para disponer de métodos necesarios para la creación de tablas. Concretamente, desde ésta clase se generan los arrays de cadenas que necesita el modelo de tabla para indicar la fila de cabecera.
- Se crea una ventana de diálogo desde la clase *DialogoTabla,* incluida en el paquete *soporteGrafico*, que servirá de contenedor a la tabla a generar.
- La linea siguiente de código se explica a continuación:
  - La función *construccionStatementProducto(campos,modificadores)* crea una cadena con la sentencia SQL a ejecutar en el SGBD para obtener la lista ordenada de resultados.
  - La función *generalSelect(<statement_construida>,campos)* devolverá la tabla de datos ordenada, ejecutando la cadena que se le pasa como parámetro.
  - La función *arrayStringParaTableModelProducto(campos)* devuelve el array que contiene la fila *header* de la tabla.
  - La llamada a la función *setDialogoModel(new javax.swing.table.DefaultTableModel(Object[][], Header[]))* rellena la tabla instanciada en momentos anteriores.
- Se hace visible la tabla generada.

Volvemos a presentar un esquema del funcionamiento y uso de funciones/clases a la hora de generar las vistas de productos en la figura 6.19, con objeto de mostrar la forma en que la capa principal se apoya en los paquetes de clases existentes.

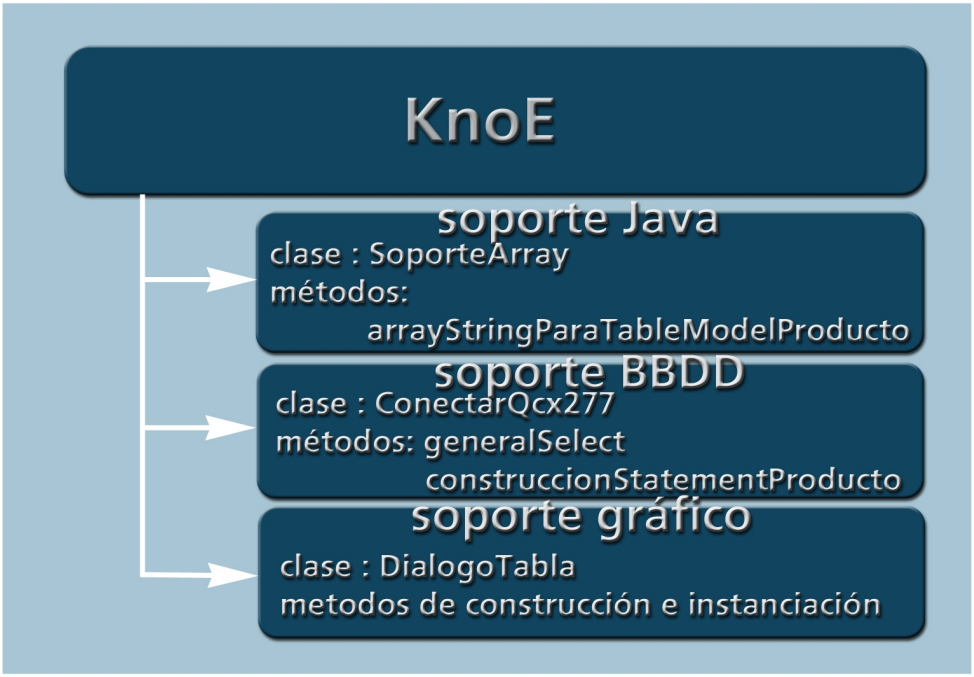


Figura 6.19: Clases/funciones usados en la generación de vistas de producto.

6.3.2.2 Inserción de producto

En este apartado se explicará el proceso que conlleva la inserción de un producto.

Desde la aplicación, se elegirá el "tabbedPane" del área de productos y, dentro de él, el de inserción. Por defecto, esta es la vista inicial una vez que se escoge el área de productos.

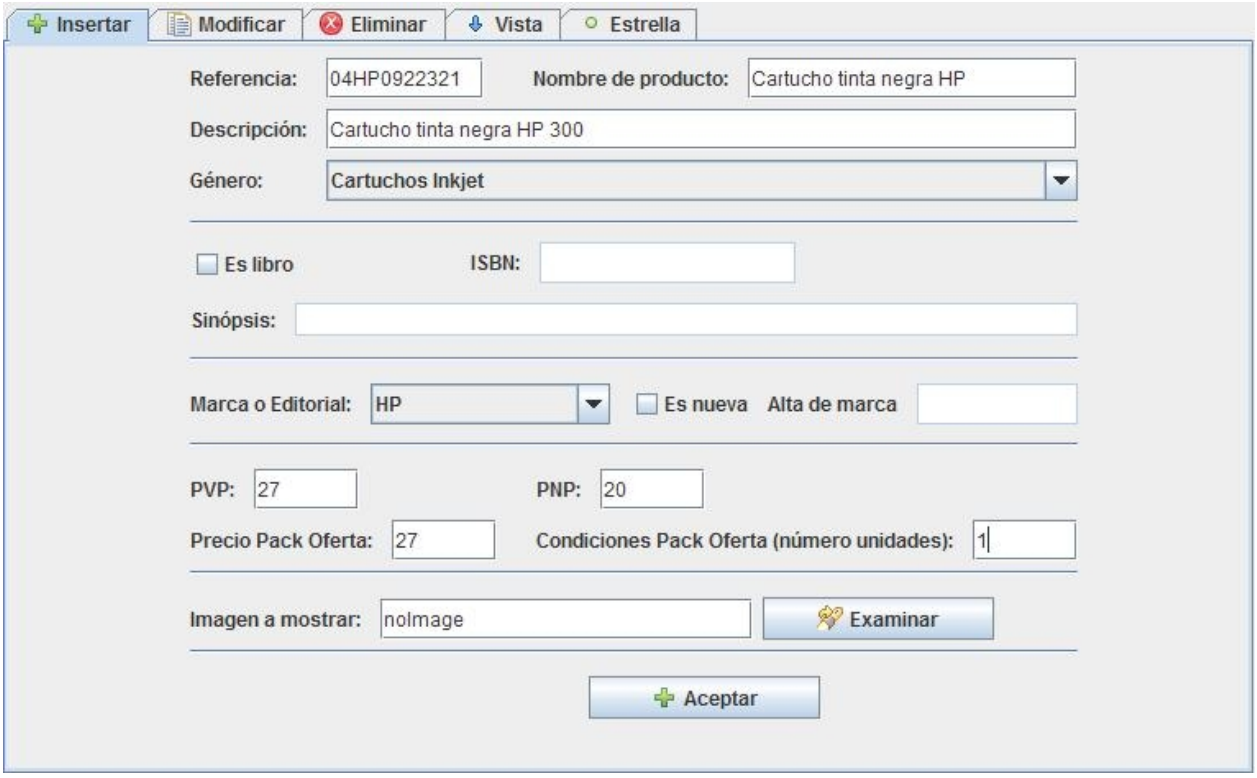


Figura 6.20: TabbedPane de inserción de producto.

La inserción un producto conlleva varias connotaciones, ya que, para realizar dicha acción, se debe actuar sobre varias tablas, tanto para obtener datos y chequear redundancias y consistencias, como para la propia inserción en ellas. A tal efecto, se produce una instanciación de la clase *ConectarQcx277*, surtiendo a la vista presente de los métodos necesarios para ejecutar los accesos a base de datos antes mencionados. Será en esta instancia donde se gestione toda la lógica de datos requerida por el modelo.

Por otra parte, se crea una instancia de la clase que aporta la funcionalidad de cliente del protocolo FTP (ver clase *ClienteFTP*), con objeto de subir la imagen referenciada en el campo de texto al servidor. Si no se explicita dicha imagen, se subirá una imagen por defecto interna al sitio web. Se ha dotado al sistema de la apertura de un *JfileChooser* que permite la navegación por la estructura de directorios y la selección de la imagen.

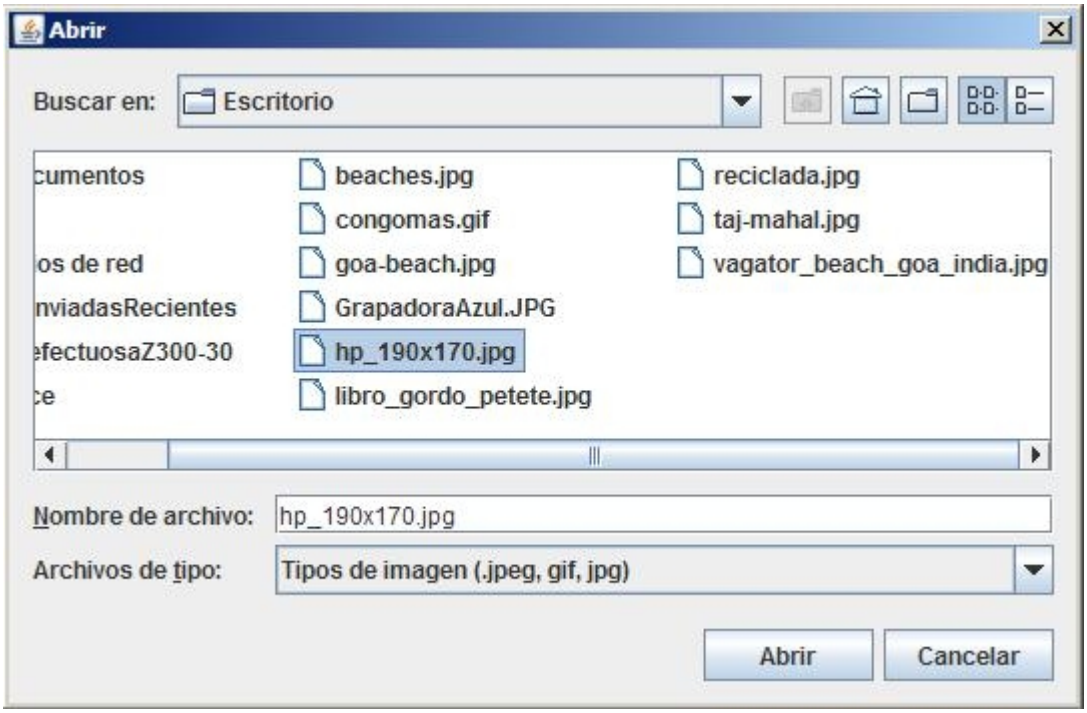


Figura 6.21: Selección de campos a generar en la tabla de productos.

Una de las clases implementadas en el paquete de soporte Java (*knoe.soporteJava*) tiene como fin aportar a la instancia de *JfileChooser* mencionada anteriormente los filtros de tipo de archivo permitidos (ver clase *FiltroFile*).

Siguiendo con el proceso, una vez insertados los datos de producto en sus correspondientes campos, y aceptado desde la vista principal, el sistema nos informará del resultado de la operación. La aplicación KnoE, cuenta en su paquete de apoyo gráfico (ver *knoe.soporteGrafico*) con una ventana de información de diagnóstico y resultado de operaciones (ver clase *DialogoACK*).

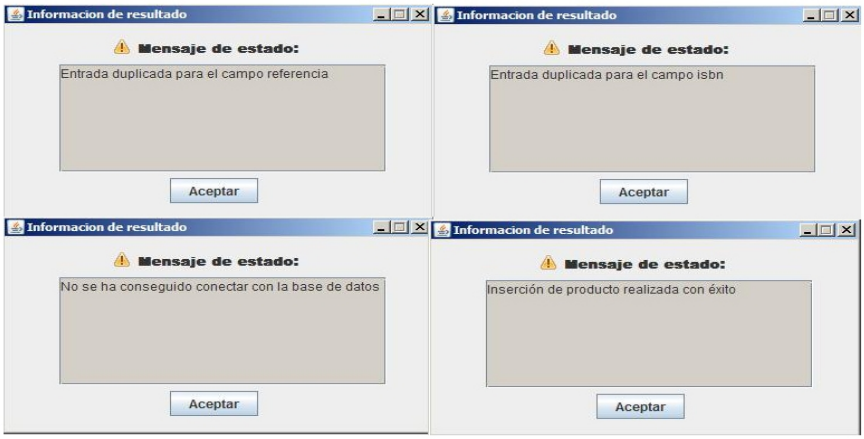


Figura 6.22: Ventana de resultado.

Secuencia

Una vez recopilados los datos desde la vista inicial, se lanzan la función *generalInsertProducto* de la clase *ConectarQcx277,* y la subida de la imagen de producto.

```
try {
    ConectarQcx277 con = new ConectarQcx277();
    ClienteFTP cftp = new ClienteFTP();
    //insertamos en la base de datos
    ack = con.generalInsertProducto(cads, isLibro);
    //subimos la imagen con el miniclienteFTP
    cftp.subirImagen(imagen);
} catch (Exception e) {
}
    DialogoACK dial = new DialogoACK(ack);
    dial.setVisible(true);
```

Texto 6.2: Fragmento de KnoE.java: inserción de productos.

La función *generalInsertProducto* realiza numerosas operaciones y devuelve un entero con código de error. Su lógica sigue el algoritmo expuesto a continuación:

- En primer lugar comprobamos la no existencia de la referencia dada.
- Si es libro, comprobamos que no existe el isbn dado.
- Si no hay entrada duplicadas de isbn y/o referencia:
    - Comprobamos si la marca a introducir con el producto existe y asignamos valores de inserción:
        - Si existe, la elegimos para insertarla.
        - Si no existe, incrementamos en 1 el código más alto.
    - Comprobamos la existencia de la linea económica:
        - Si no existe debemos incrementar el numero de linea máximo.
    - Seleccionamos el código de categoría de producto en función del nombre.
    - Creamos la query a ejecutar.
    - Ejecutamos la query.
    - Devolvemos el código de error.

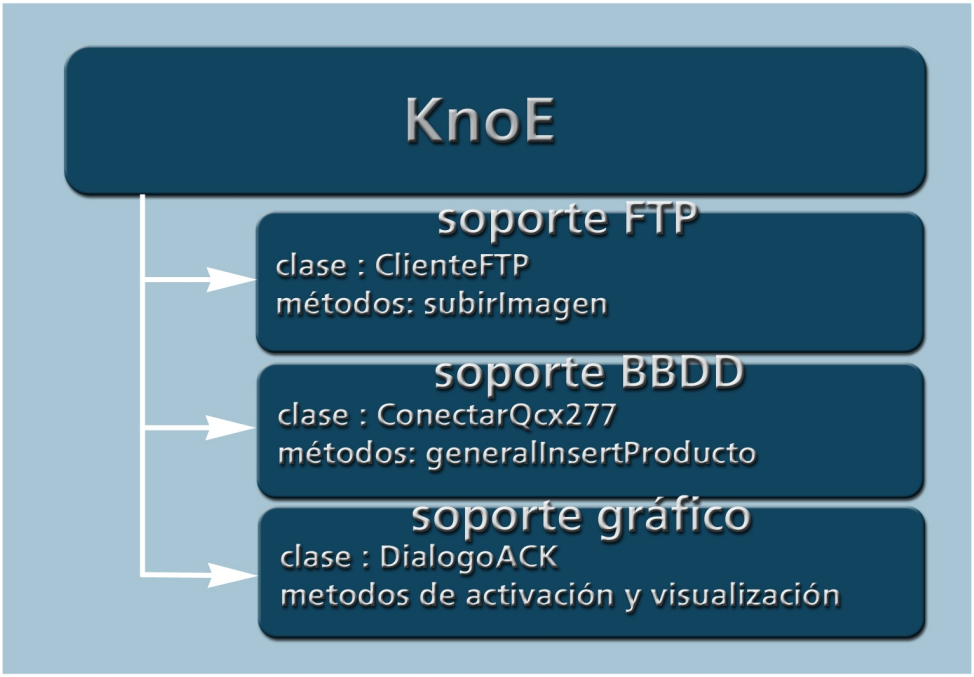


Figura 6.23: Clases/funciones usados en la inserción de producto.

6.3.2.3 Modificación de producto

Al igual que para la inserción, la modificación de un producto requiere atomicidad en las operaciones SQL de inserción, desde el momento en que se actualizan varias tablas del modelo relacional. *ConectarQcx277* implementa un mecanismo de ROLLBACK en todas las operaciones UPDATE o INSERT que permite devolver la base de datos a estados de consistencia.

El proceso de modificación es más complejo que la inserción, ya que se necesita chequear el estado previo y comparar con los datos nuevos en varias tablas. Para ello, las clases del paquete de soporte gráfico del área de productos (ver paquete *knoe.soporteGrafico.producto*) gestionan un procedimiento de creación de nuevos editores gráficos. Este procedimiento, va totalmente en paralelo a la lógica de modificación, por lo que la gestión de la lógica de modificación recae enteramente en este paquete de apoyo.

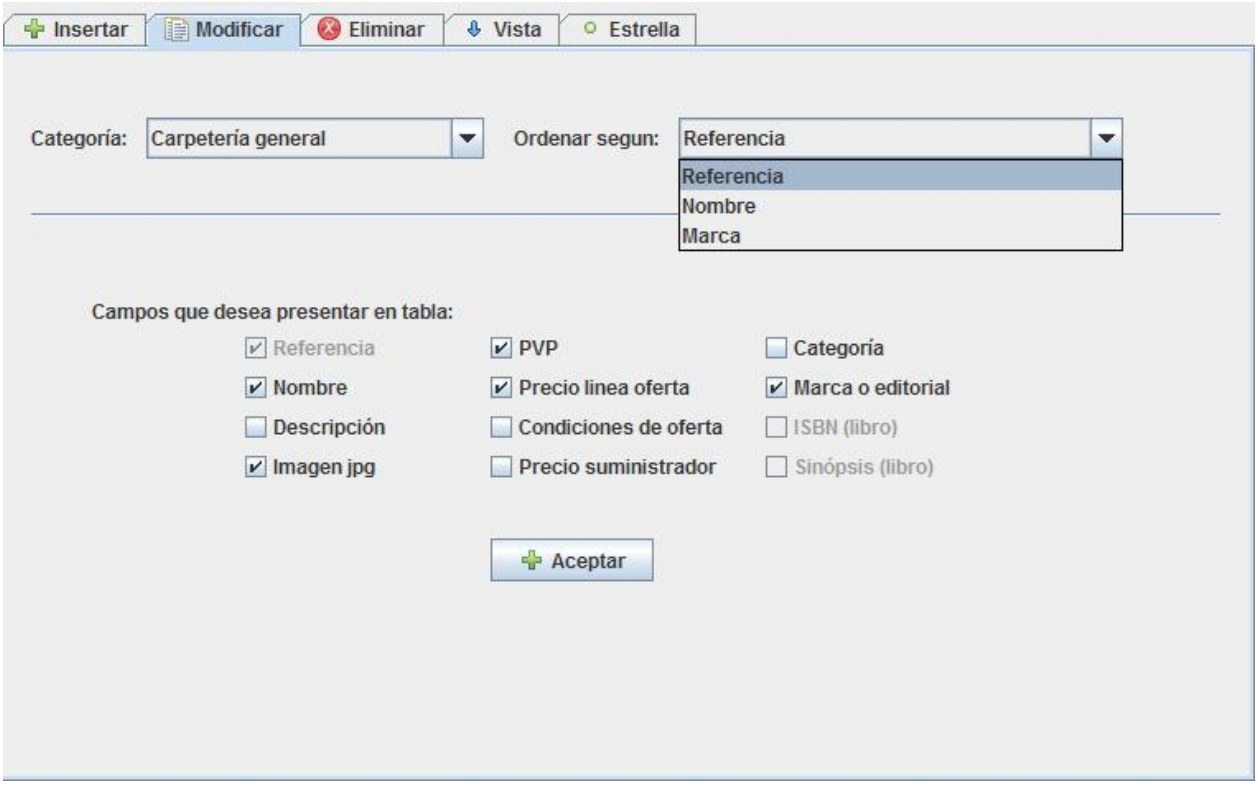


Figura 6.24: Selección de campos a generar en la tabla de modificación de productos.

Parte este proceso de una vista inicial semejante a la creada en la inserción. Para modificar un producto, se debe seleccionar desde una vista antes. Esta situación se refleja en la figura 6.24. Desde aquí, seleccionamos los campos que queremos que se muestren en la vista de partida del proceso. Una vez aceptado, se elegirá el producto a modificar, iniciando una nueva rutina de la aplicación. Debemos hacer notar al lector que la modificación de producto es una tarea ejecutada uno a uno, y no masiva, como puede ser la presentación de datos o su eliminación.

Parámetros de intercambio entre la clase superior y la clase *ModeloTabla* hacen que dicha tabla sea usada con fines de actualización o de eliminación, bien de productos o bien de clientes.

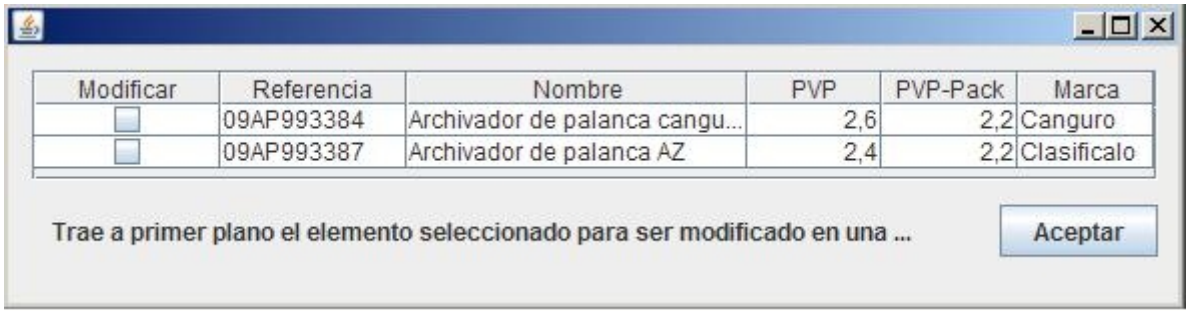


Figura 6.25: Selección del producto a modificar.

Pulsando sobre el producto a cambiar, se nos abrirá un editor que nos permite realizar las actualizaciones de valores. Se muestra el editor en la figura 6.26.

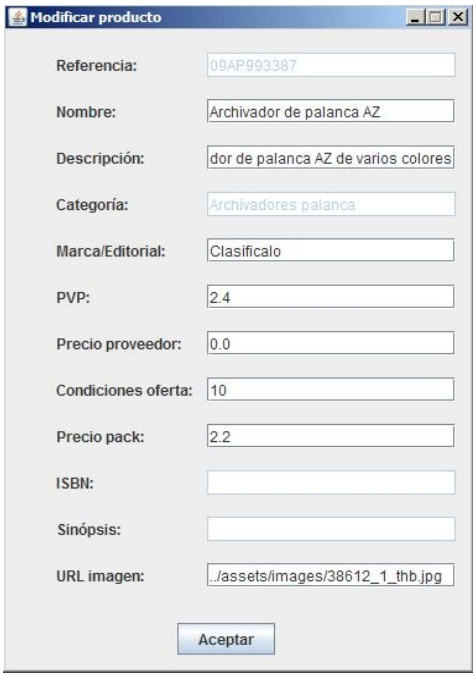


Figura 6.26: Editor de modificaciones de producto.

Como punto extra a analizar, mencionamos el caso de modificación de la marca de producto. En este caso, se ha generado una nueva ventana de proceso que permite elegir entre las marcas dadas de alta en la base de datos. Cuando la subrutina es llamada al pulsar sobre el campo de texto "*Marca/Editorial*", se crea una instancia de la nueva subrutina (figura 6.27). Tras la edición de la marca de producto, se vuelve al editor de modificaciones de producto.

Se han separado ambos procesos porque el cambio de la marca puede tener un resultado exitoso o no independientemente de la actualización del resto de campos en diferentes tablas. Esta escisión en la rutina no afecta en ningún momento a la integridad de los datos de la base de datos.

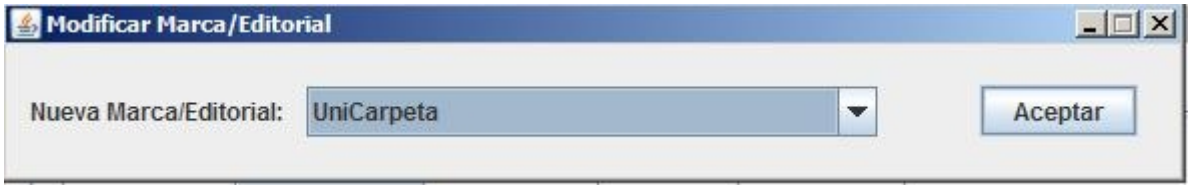


Figura 6.27: Diálogo de modificación de marca de producto.

Nuevamente, la aplicación informa del resultado del proceso de actualización de producto a la finalización del mismo.

Figura 6.28: Resultado de la modificación de producto.

Secuencia

Vamos a estudiar el caso de la modificación de producto. La eliminación es un proceso totalmente similar a la modificación. La eliminación solo supone el borrado de un registro en base de datos o la actualización de un atributo ("*eliminado*") en una de las tablas. Por oto lado, modificar un producto tiene unas implicaciones mucho más complejas: se puede cambiar marca, precio, categoría y un largo etcétera que pueden afectar a la integridad de los datos, desde el punto y hora en que varias tablas conexas son susceptibles de cambios simultáneos. Veremos que la actualización puede implicar la escritura de varias tablas.

Una vez generado el panel, mediante la clase *DialogoTabla*, que contiene la tabla de datos (productos modificables/eliminables), se tienen filas activas y seleccionables que apuntan a los elementos referenciados. La instanciación de la clase se realiza mediante constructores que distinguen entre varias posibilidades (modificación/eliminación de productos/clientes o edición de producto estrella). Si se instancia la ventana de diálogo en modo eliminación, la tabla de datos mostrada no tiene elementos activos, sino que, al aceptar, se leerán las filas seleccionadas y se procederá al borrado de registros de la base de datos. Por contra, para la modificación, las filas implementan un "listener" que las hace activas. En este sentido, la actualización de producto es una operación uno a uno. Es por eso que, cuando se pulsa sobre una fila, se activa la rutina de modificación propiamente, siendo pasado el producto – su referencia – como parámetro.

```
ConectarQcx277 con = new ConectarQcx277();
SoporteArray soporte = new SoporteArray();
Object[][] data = con.generalUpdate(con.construccionStatementProducto(campos, modificadores), campos);
if (data != null) {
 DialogoTabla dt = new DialogoTabla(soporte.arrayStringParaTableModel_UpdateProducto(campos, true),
                    data, con.selectCodCatFromCategoriaProducto(modificadores[0]));
 dt.setVisible(true);
} else {
 DialogoACK dial = new DialogoACK(16);
 dial.setVisible(true);
}
```

Texto 6.3: Fragmento de KnoE.java: modificación de productos.

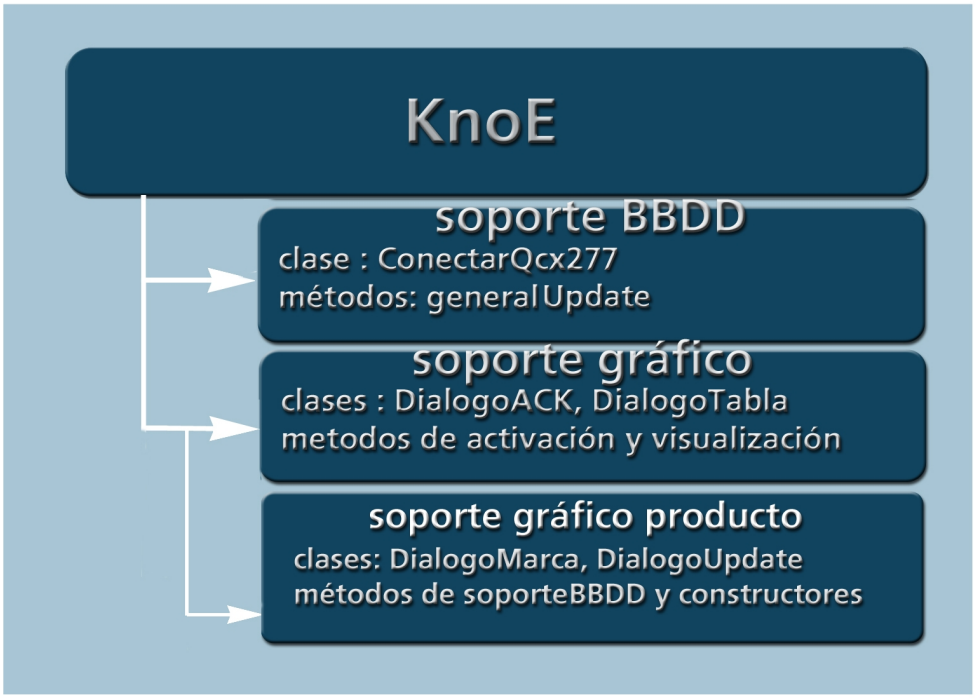


Figura 6.29: Clases/funciones usados en la modificación/eliminación de producto.

En la figura 6.30 se muestra la ventana (clase *DialogoUpdate*) que se abre tras pulsar sobre una de las filas de la tabla de productos modificables. Se observan los distintos campos modificables, y sombreadas las agrupaciones de tablas que se ven afectadas por la actualización. En la figura mencionada, se listan 5 grupos de tablas afectadas:

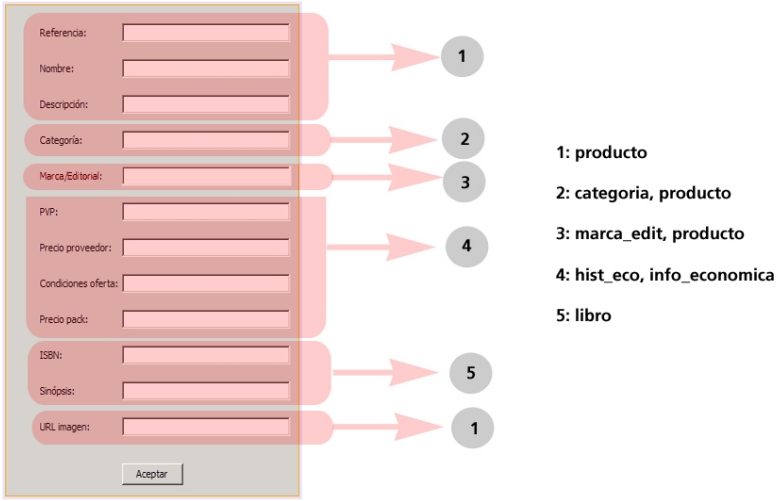


Figura 6.30: Conjuntos de tablas afectadas en producto.

Desde la clase *DialogoUpdate* se invoca el método *generalUpdateProducto* de la clase *ConectarQcx277* para gestionar definitivamente la actualización de los campos que han cambiado. Al inicio del apartado 6.3 se explicó con mayor profundidad la gestión frente a errores de las escrituras en base de datos, atomicidad de operaciones e integridad del polígono de datos.

Una vez que se pulsa Aceptar en el editor mostrado anteriormente, se realiza una llamada a la función *generalUpdateProducto* de *ConectarQcx277* con los valores actuales de los campos del editor. El algoritmo que se sigue es el siguiente:

- Si hay cambio de precios.
  - Se comprueba si la línea de precios existe o no.
- Si además es libro, comprobamos que el nuevo ISBN no existe.
- Generamos la query con el siguiente criterio:
  - Hay cambio de precios y la línea no existe: generar nueva linea. Actualización de producto.
  - Hay cambio de precios y la linea existe: asignar código existente de línea. Actualización de producto.
  - No hay cambio de precios: Actualización de producto.
  - Si es libro y no existe: Actualización de libro.
- Ejecutamos la query.
- Devolvemos código de error.

Ya mencionamos anteriormente que si se pulsa sobre el campo de texto de marca/editorial, se lanza la subrutina *DialogoMarca.* La secuencia que se sigue desde este editor es la contenida en la función *updateMarcaProducto* de *ConectarQcx277,* que hace una actualización con el campo elegido en el scroll de pantalla.

6.3.2.4 Eliminación de producto

Se gestiona esta rutina de forma muy parecida a la modificación de producto. De hecho, si el producto se ve implicado en algún pedido, la eliminación real no se llevará a cabo, sino que consistirá en la actualización del atributo de estado *eliminado* de la tabla *producto* a “1”. En otro caso, si se eliminará el registro de la tabla.

El motivo de la casuística propuesta es claro. No podríamos llevar a cabo esta eliminación sin dejar el sistema relacional en un evidente estado de inconsistencia. Por otra parte, el sistema de gestión de base de datos no nos permitiría la realización de esta acción, ya que contraviene las restricciones de integridad definidas en su

momento en la creación del modelo relacional.

Mencionamos ahora que en este caso si se puede tratar la eliminación como una acción masiva. En la figura 6.32, se contrasta este hecho, mostrándose como se eliminan varios registros de una sola vez.

Se inicia el proceso desde la vista de selección de campos, al igual que en apartados anteriores, con el objetivo de generar una tabla en la que mostrar los productos de la categoría seleccionada.

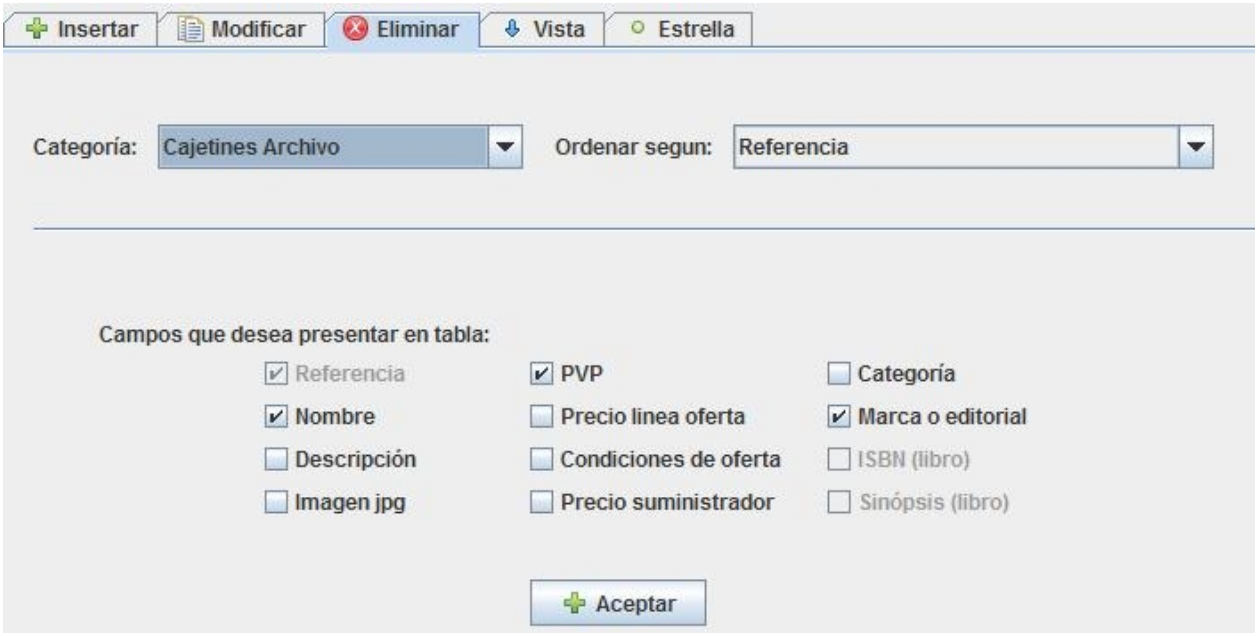


Figura 6.31: Selección de campos a generar en la tabla de eliminación de productos.

Tras la acción anterior, nos aparecerá una tabla con los productos, que pertenece a la misma clase que la usada en la rutina de modificación. Parámetros de intercambio entre la clase superior y la clase *ModeloTabla* hacen que dicha tabla sea usada con fines de actualización o de eliminación, bien de productos o bien de clientes.

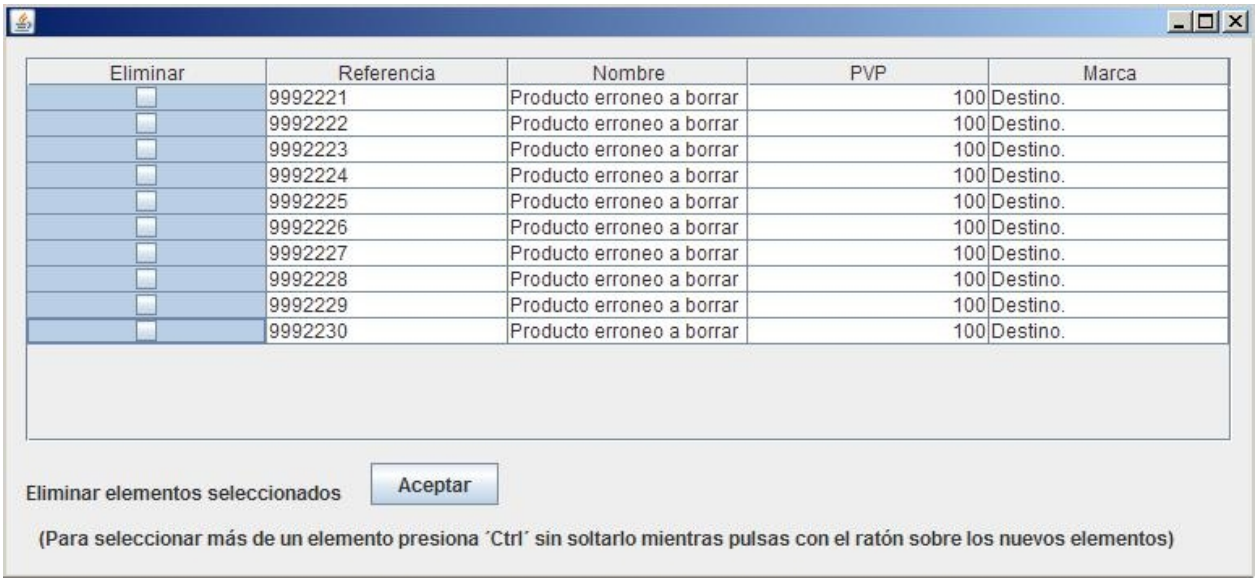


Figura 6.32: Selección de productos a eliminar.

Una vez que aceptamos, el sistema devolverá una ventana de información del proceso, que nos comunicará el éxito o fracaso de la acción y, en éste último caso, el motivo.

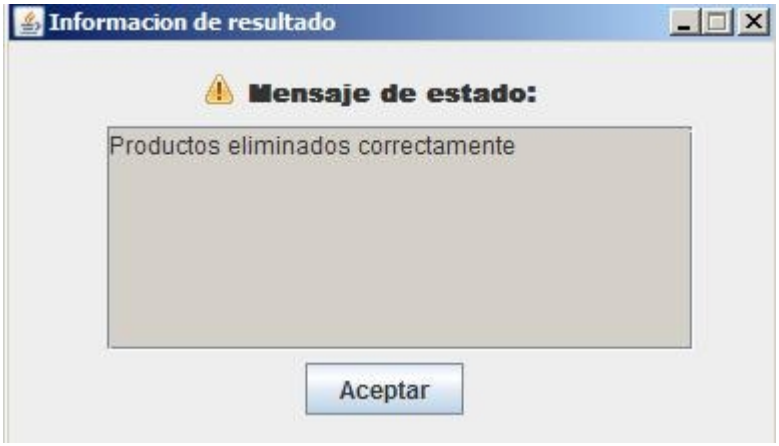


Figura 6.33: Resultado de la eliminación de productos.

Secuencia

La eliminación del producto es una rutina completamente similar a la modificación. La única diferencia de código fuente entre lanzamientos de rutinas de modificación o eliminación radica en un parámetro de la función *arrayStringParaTableModel*. En este caso, desde KnoE, se instancia la clase *DialogoTabla* indicando al constructor que se trata de una eliminación. Desde la propia rutina se inicia la secuencia de eliminación, en la que se utilizan nuevamente métodos definidos en la clase *ConectarQcx277*.

En cualquier caso, ya sea modificación o bien eliminación del producto, la

gestión de errores y presentación por ventana de alerta se realiza mediante la clase *DialogoACK*.

6.3.2.5 Producto Estrella

Según las necesidades de la explotación del sitio web en cuestión, el producto estrella es un producto que se desea reflejar de manera especial en dicha web. La ubicación de este producto se enmarca en un lugar privilegiado de acceso al cliente final, durante la navegación por la página. La gestión de presentación de este producto desde la aplicación KnoE se realiza mediante un nuevo "*tabbedPane*", mediante el que se origina la rutina de actualización de los productos estrella.

Partiendo de la vista inicial de selección de campos a mostrar, nuevamente, se crea una tabla con los productos seleccionables para ser etiquetados como "estrella".

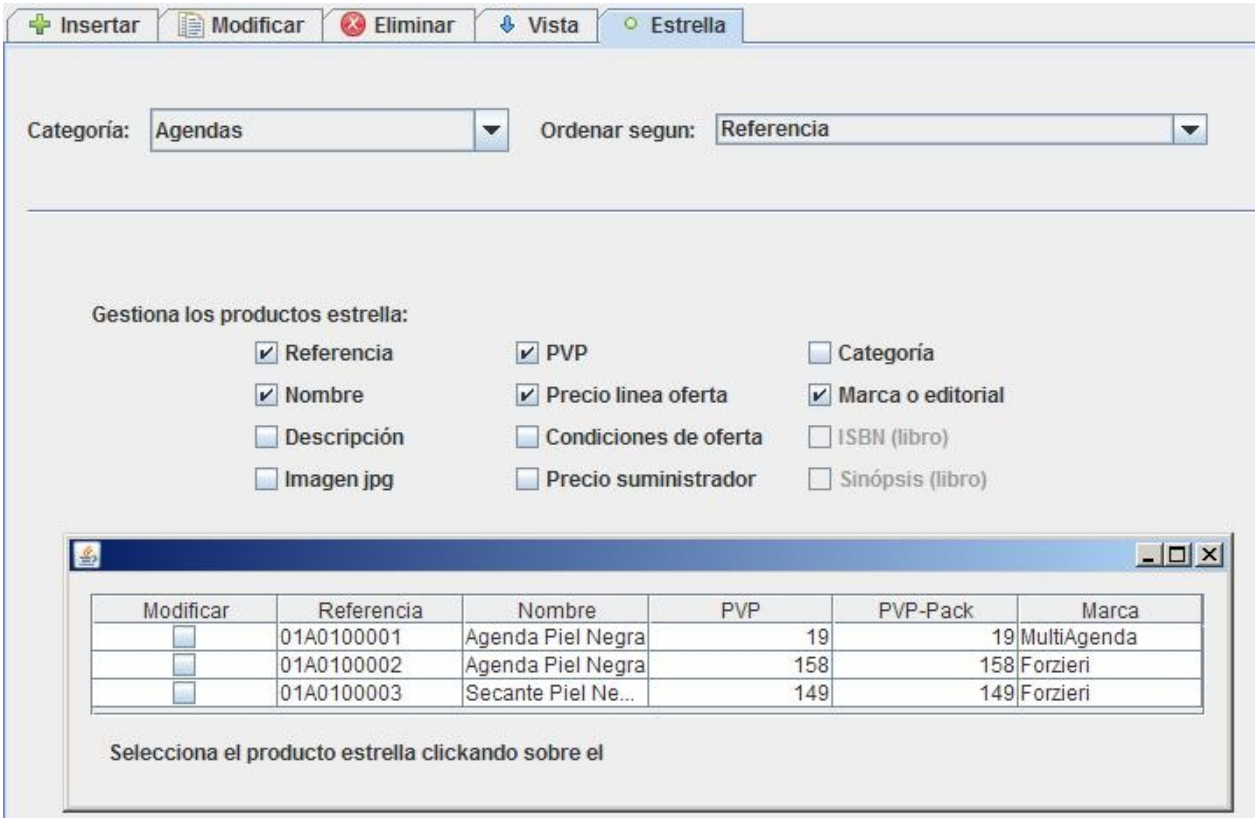


Figura 6.34: Vista de tabla de productos seleccionables.

Figura 6.35: Instancia de la clase DiálogoEstrella.

Al pulsar con el ratón sobre alguno de los elementos de la tabla mostrada en la figura 6.34, se genera una instancia de la clase *DialogoEstrella*. Esta clase consiste en un nuevo editor que se muestra en la figura 6.35.

Sobre la instancia creada seleccionaremos la posición del producto estrella (1, 2 o 3) e introduciremos el texto que pretendemos que se presente de forma especial en el sitio web.

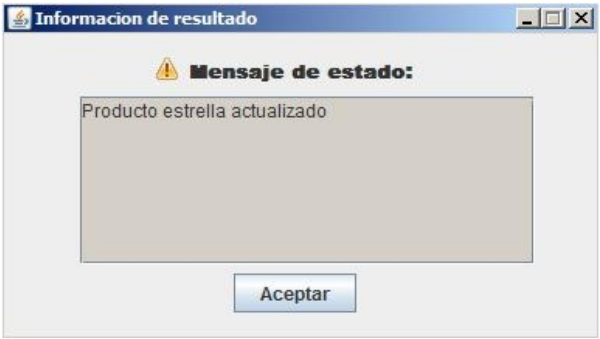


Figura 6.36: Ventana de información de resultado.

La ejecución del proceso lanza a su finalización el correspondiente mensaje de resultado, indicando el éxito o el fracaso de la ejecución.

Secuencia

Se puede observar en el texto 6.4 que no existe diferencia alguna en la la forma de instanciar clases con respecto a los apartados anteriores. La diferencia radica en el constructor de la clase *DialogoTabla* usado. El último parámetro de la llamada al constructor es un int a 0, que distingue el tipo de tabla que se generará. Esta tabla tendrá listeners activos en los campos que harán correr una instancia nueva de la

subrutina – ventana de diálogo *DialogoEstrella*.

Una vez finalizado el diálogo, tal y como se mostró en el apartado anterior, se actualizará el producto seleccionado como estrella, usando la función *updateEstrellaProducto* de la clase *ConectarQcx277*.

```
ConectarQcx277 con = new ConectarQcx277();
SoporteArray soporte = new SoporteArray();
Object[][] data = con.generalUpdate(con.construccionStatementProducto(campos, modificadores), campos);
if (data != null) {
 DialogoTabla dt = new DialogoTabla(soporte.arrayStringParaTableModel_UpdateProducto(campos, true), data,
                                    con.selectCodCatFromCategoriaProducto(modificadores[0]), 0);
                          dt.setVisible(true);
} else {
 DialogoACK dial = new DialogoACK(16);
 dial.setVisible(true);
}
```

Texto 6.4: Fragmento de KnoE.java: estrella.

Figura 6.37: Clases/funciones producto-estrella.

6.3.3 Área de cliente

La forma de proceder en este área de datos del modelo relacional será muy similar a la propuesta en el apartado anterior. En este sentido, la reutilización del

código se hace patente.

La capa superior de la aplicación se apoyará en una serie de elementos genéricos aportados por las clases de los paquetes *knoe.soporteJava* y knoe.soporteGráfico en la generación de las vistas de clientes. De manera específica, el paquete *knoe.soporteGrafico.cliente* aporta la interfaz para interactuar con la clase de soporte de base de datos (ver clase *ConectarQcx277*) a la hora de ejecutar modificaciones sobre el área involucrada.

Exponemos las funcionalidades requeridas a esta área:

- Generación de vistas de clientes.
- Modificación de clientes.
- Eliminación.

6.3.3.1 Vistas de cliente

Podemos elegir en el "*tabbedPane*" inicial (figura 6.39) los campos que deseamos mostrar de los clientes. En esta vista, al contrario que en el caso de la generación de tablas de productos, no hemos incluido ningún modificador de la sentencia SQL final. No se nos propuso en ningún momento la necesidad de ordenar

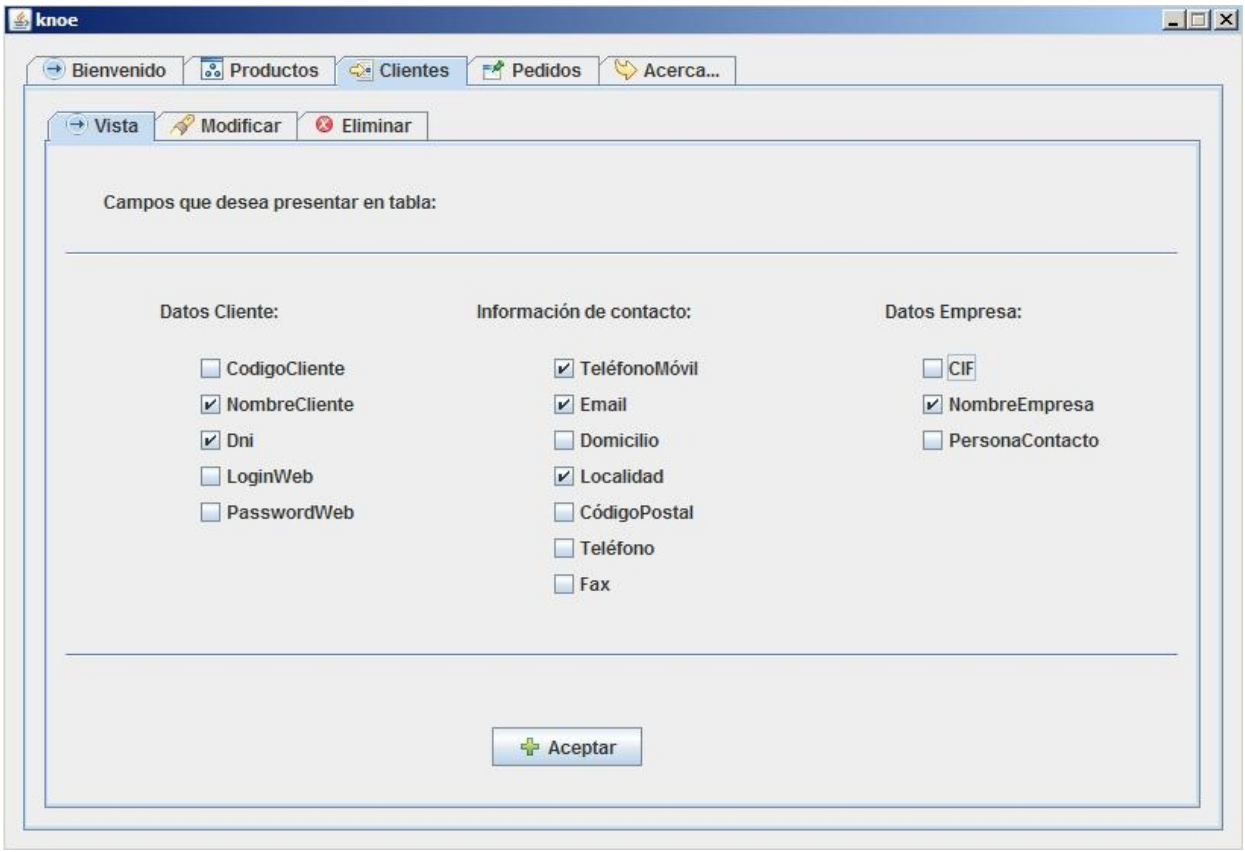
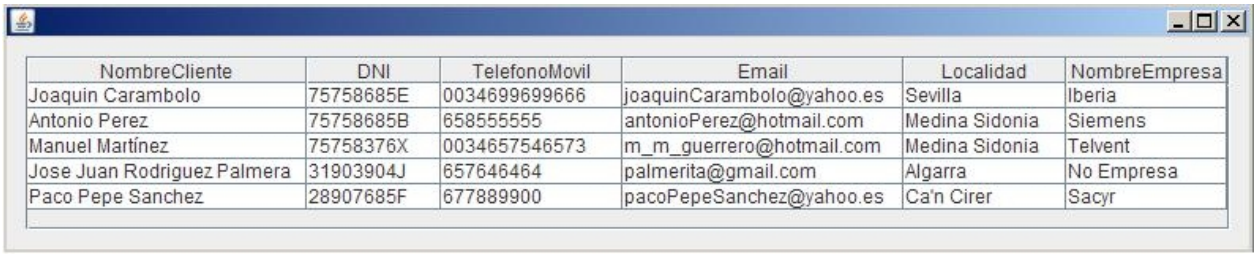


Figura 6.38: Generación de tabla de clientes.

los clientes por un determinado criterio, o la extracción de un subgrupo de ellos.

Al pulsar sobre el botón Aceptar, generamos la consulta SQL, la ejecutamos y se presenta la totalidad de los clientes en una tabla.



NombreCliente	DNI	TelefonoMovil	Email	Localidad	NombreEmpresa
Joaquin Carambolo	75758685E	0034699699666	joaquinCarambolo@yahoo.es	Sevilla	Iberia
Antonio Perez	75758685B	658555555	antonioPerez@hotmail.com	Medina Sidonia	Siemens
Manuel Martínez	75758376X	0034657546573	m_m_guerrero@hotmail.com	Medina Sidonia	Telvent
Jose Juan Rodriguez Palmera	31903904J	657646464	palmerita@gmail.com	Algarra	No Empresa
Paco Pepe Sanchez	28907685F	677889900	pacoPepeSanchez@yahoo.es	Ca'n Cirer	Sacyr

Figura 6.39: Vista de tabla de clientes.

Secuencia

La generación de la tabla se hace de forma completamente similar a la expuesta para la tabla de productos (apartado 6.3.2.1).

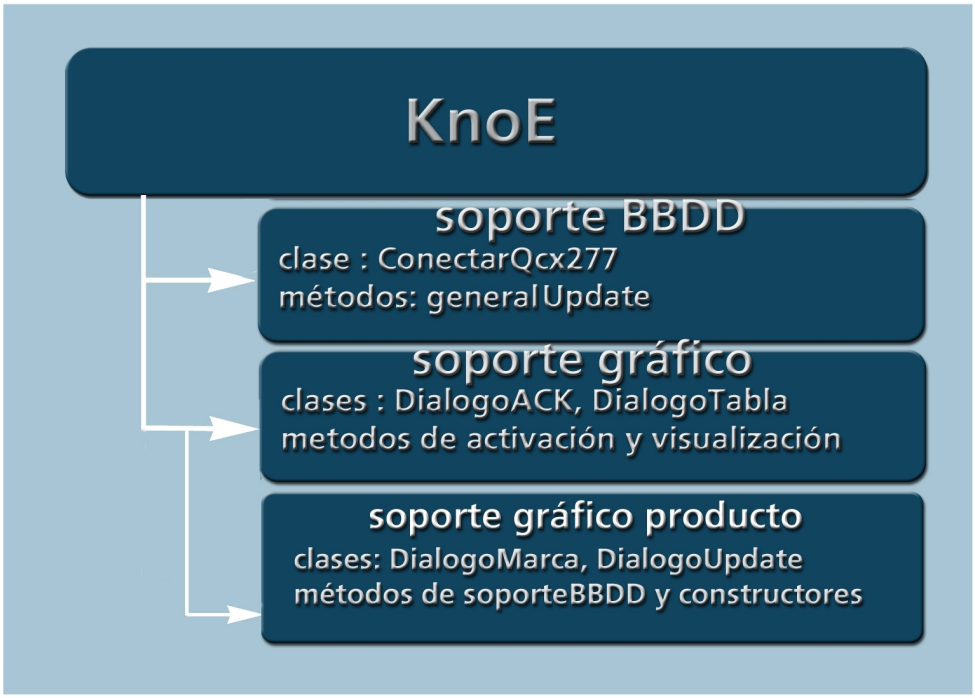


Figura 6.40: Clases/funciones usados en la vista de clientes.

6.3.3.2 Modificación de cliente

Nuevamente, un proceso de modificación implica la aparición de una rutina de editores destinados a modificar diferentes campos. La interacción entre la capa superior y el acceso a la base de datos, como ya mencionamos anteriormente, se produce con la mediación de las clases del paquete *knoe.soporteGrafico.cliente* a la hora de la actualización de datos.

Al igual que en el área de productos, la modificación de clientes no es una acción masiva, sino uno a uno.

Figura 6.41: Vista de tabla activa de clientes.

Partiendo de una vista similar a la expuesta en el apartado anterior, se lanza una rutina de modificación. La tabla de clientes que se genera (figura 6.41), contiene elementos que la hacen activa, y es capaz de lanzar subrutinas propias. De esta forma, al pulsar con el ratón sobre un cliente de la tabla, se lanzará el editor de clientes, entrando ya en el paquete de apoyo específico al área de clientes.

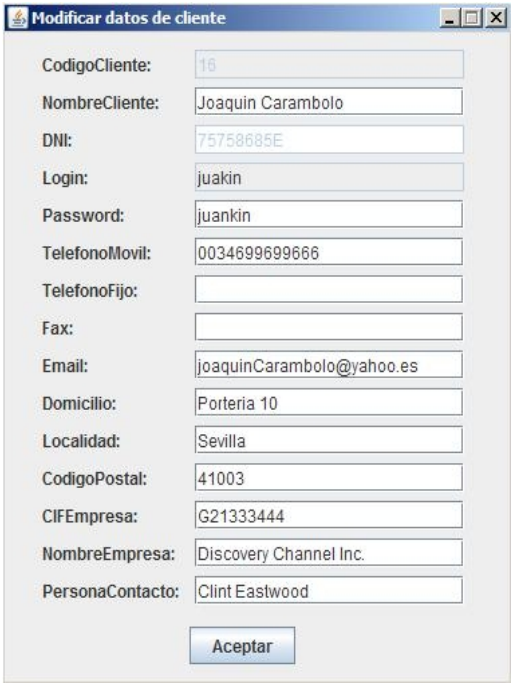


Figura 6.42: Editor de modificaciones de cliente.

El editor propio de modificación de cliente es el expuesto en la figura 6.42. Dicho editor consta de campos activos que vuelven a lanzar nuevas subrutinas, a saber:

- El campo de *localidad*, lanza una rutina de alta/selección de localidad.
- Los campos relacionados con la entidad del modelo de datos *empresa,* lanzan una rutina de alta/selección de empresa.

Los editores de estas subrutinas se muestran en las figuras 6.43 y 6.44.

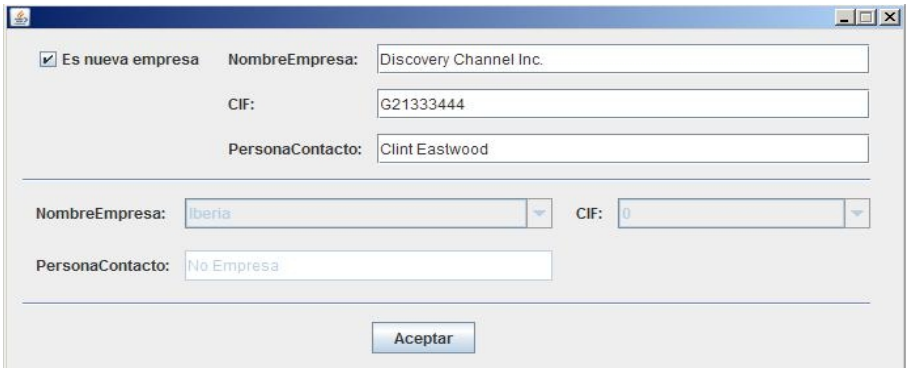


Figura 6.43: Editor de modificaciones de empresa.

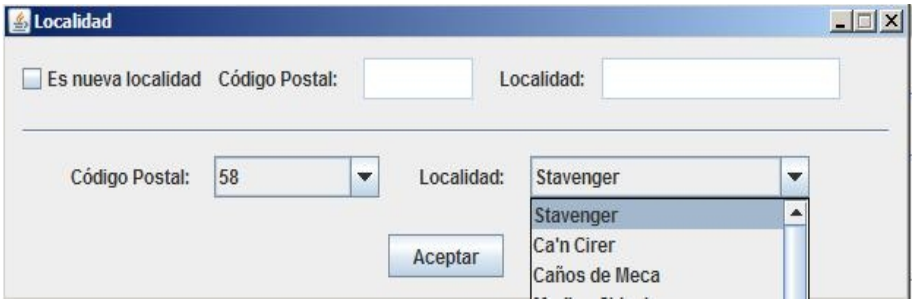


Figura 6.44: Editor de modificaciones de localidad.

Estas dos subrutinas presentadas anteriormente, devuelven el control al editor principal de modificaciones en cliente. Una vez terminado todo el proceso, el editor cede el control, con gestión de errores, a la capa principal que muestra un mensaje de éxito o fracaso del proceso.

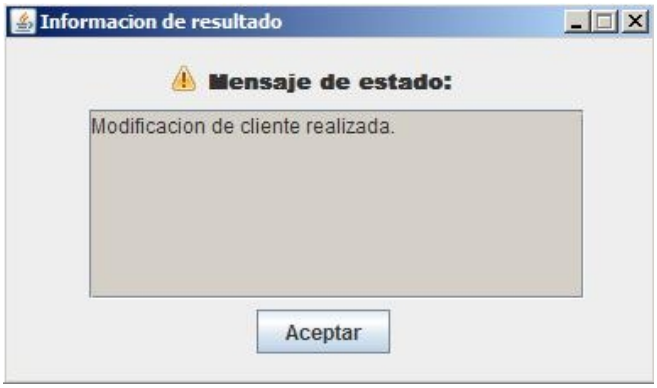


Figura 6.45: Ventana de información de resultado.

Secuencia

La modificación de cliente, al igual que en el caso de producto, implica la lectura/escritura en varias tablas. En la figura 6.46, mostramos los grupos de tablas afectadas según se modifican campos en el editor de *DialogoUpdateCliente.*

DialogoUpdateCliente es el editor de *soporteGrafico.cliente* que lanza la

lógica secuencial.

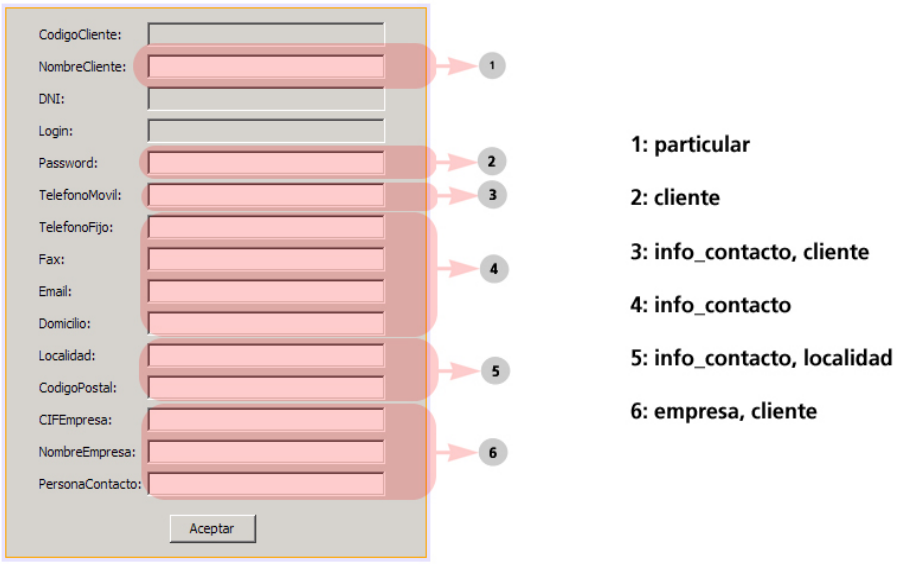


Figura 6.46: Conjuntos de tablas afectadas en cliente.

Pulsar sobre los campos *localidad* o *código postal* lanzan la subrutina de *DialogoLocalidad*, que gestiona una actualización. En función de si la localidad existe o se trata de una nueva, se usa la función *updateLocalidadCliente* de *ConectarQcx277*. Se expone el algoritmo que sigue la función:

- Si la localidad no existe, se prepara la query para un nuevo registro en la tabla *localidad*.
- Adición a la query de condiciones de actualización de tablas *localidad* e *info_contacto.*
- Ejecución de la query.
- Devolución de boolean de control.

Pulsar sobre *CIFEmpresa*, *NombreEmpresa* o *PersonaContacto* lanzan de igual forma el editor de la subrutina *DialogoEmpresa.* En función de si la empresa existe o se trata de un alta de empresa, se usa la función *updateEmpresaCliente* de *ConectarQcx277*. Se expone el algoritmo que sigue la función:

- Si la empresa no existe, se prepara la query para un nuevo registro en la tabla *empresa*.
- Adición a la query de condiciones de actualización de tablas *cliente* y *empresa.*
- Ejecución de la query.
- Devolución de boolean de control.

Una vez se pulsa Aceptar, se recopilan del editor los datos escritos en los campos que implican una modificación de la entidad cliente. Los valores iniciales,

son mantenidos internamente a efectos de comparar con los nuevos valores. De esta forma, ahorramos una lectura de base de datos. Si hay cambios en dichos campos, se procede a realizar la llamada al método *generalUpdateCliente* de *ConectarQcx277*. Se expone la secuencia a continuación:

- Si no hay cambios en la información de contacto:
  - Se prepara la query para actualizar las tablas *particular* y *cliente*.
- Si hay cambios en la información de contacto:
  - Además se actualizará la tabla *info_contacto*.
- Se ejecuta la query.
- Se devuelve el código de error.

La figura 6.47 muestra un árbol de clases y métodos implicados en el proceso descrito.

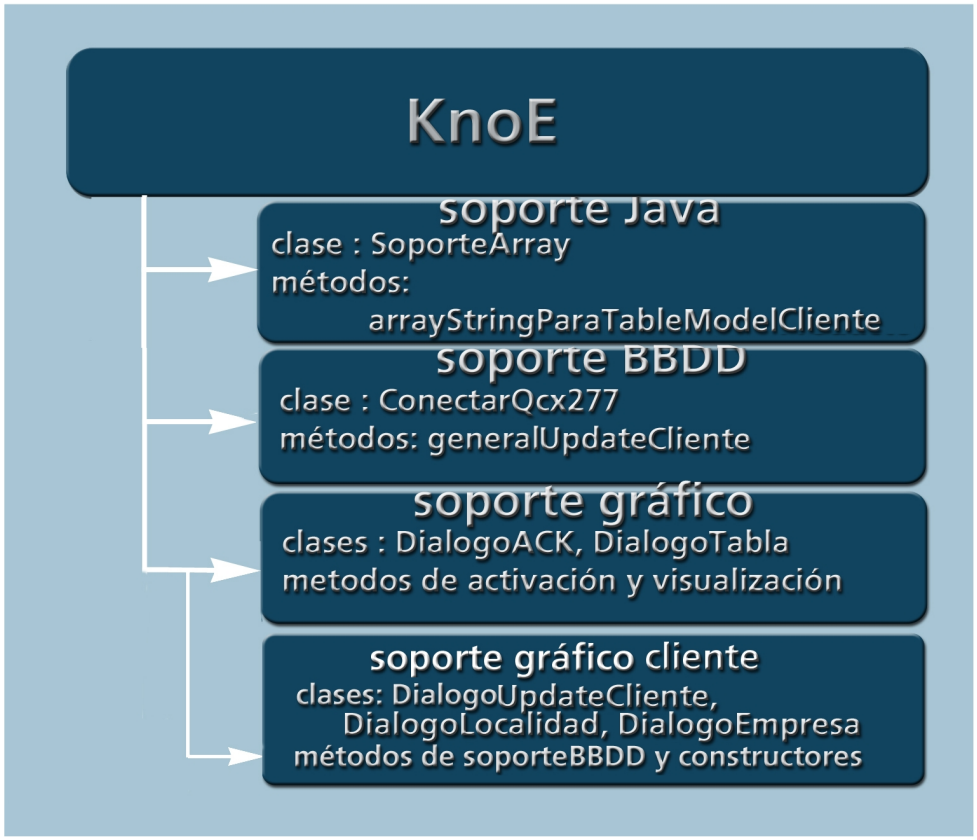


Figura 6.47: Clases/funciones usados en la modificación/eliminación de cliente.

6.3.3.3 Eliminación de cliente

La forma de proceder será totalmente similar en cuanto a forma, como a fondo respecto al área de producto.

Un cliente será eliminado realmente si no se ve involucrado en ningún pedido. Si no se cumple esta condición, se actualizará un variable de estado que indique que, aunque el registro existe, el cliente ha sido eliminado.

La aplicación, tanto para generar la vista inicial como para realizar la eliminación, interactúa directamente con el paquete de soporte de base de datos.

Desde la vista inicial en el "tabbedPane" de eliminación de clientes, se genera la tabla con la totalidad de clientes. Hacemos notar en este punto que es posible la eliminación masiva, eligiendo en la tabla de varios clientes (figura 6.48).

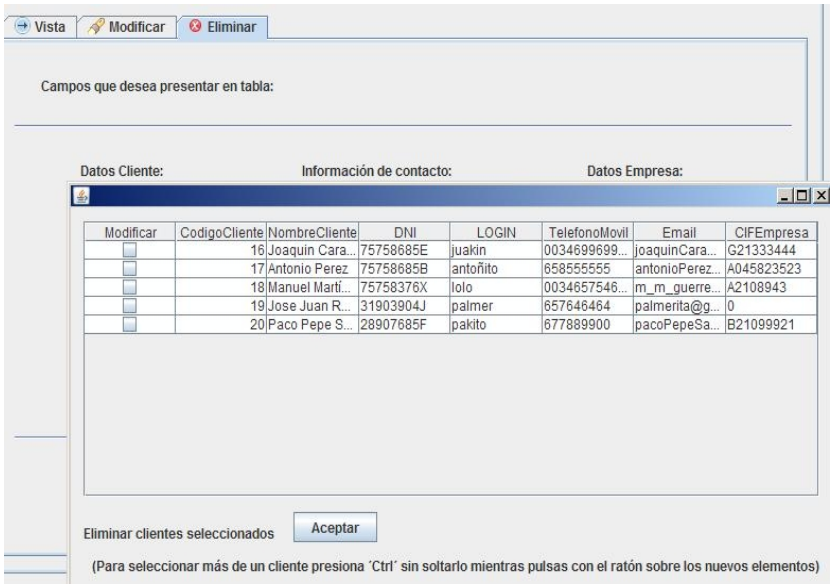


Figura 6.48: Tabla de clientes seleccionables.

Al aceptar, se ejecuta la sentencia SQL apropiada y se devuelve un mensaje de éxito o fracaso en el proceso (figura 6.49).

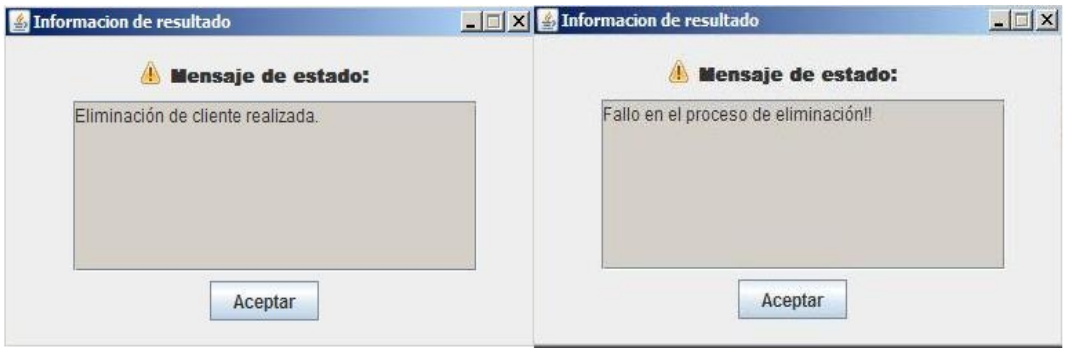


Figura 6.49: Mensajes de resultado de proceso.

Secuencia

La operación de eliminación es completamente similar desde la capa KnoE, a excepción del constructor que se usa en la llamada a *DialogoTabla*. En este caso, el constructor polimorfo genera una tabla con un selector de filas. Una vez que se pulsa el botón se lanza directamente la función *deleteFromCliente* de *ConectarQcx277*. La secuencia que sigue dicho método es la siguiente:

- Para todos los clientes seleccionados y uno a uno:
  - Comprobar si hay algún registro en la tabla *pedido* con el código de cliente actual.
  - Si existe el registro, la sentencia en la query a crear consiste en una actualización de la tabla *cliente*, atributo *eliminado* a 1.
  - Si no es así:
    - Preparar la sentencia de la query para eliminar registro de las tablas *cliente*, *particular* e *info_contacto*.
- Se ejecuta la query.
- Se devuelve el código de error.

6.3.4 Área de pedido

En este apartado, no se pretende introducir datos nuevos ni actualizar información, desde el punto de vista del operador del sistema. La inserción del pedido en las tablas del modelo se hará desde el sitio web por parte del cliente. El operador solo necesitará ver los pedidos pendientes de realización y, una vez realizados pasarlos al registro histórico de pedidos.

Hacemos hincapié en este punto en que el sistema no cuenta con dos tablas para los pedidos pendientes y el histórico, sino que una variable de estado servirá de elemento de distinción entre ambos. Una vez que se actualiza la marca de tiempo *ts_2* a un valor distinto de nulo, se da por entendido que el pedido ha sido realizado.

6.3.4.1 Vistas de pedidos pendientes

Esta rutina, a pesar de ser catalogada como “vista”, es realmente el inicio de una rutina de edición, ya que, una vez visualizado, se tiene la opción de salir de la vista sin realizar ninguna acción, o bien aceptar y dar el pedido por ejecutado.

Por tratarse de una actualización de registro, la clase principal se apoya en el editor creado en el paquete de apoyo específico de pedidos (ver *knoe.soporteGrafico.pedido*).

La forma de proceder en esta rutina se muestra a continuación.

En primer lugar, desde el "tabbedPane" principal del apartado de pedidos, se puede acceder a la visualización de los pedidos pendientes o del histórico. Analizamos aquí la rutina anteriormente citada de edición, dejando para el posterior la generación de la vista del registro de históricos.

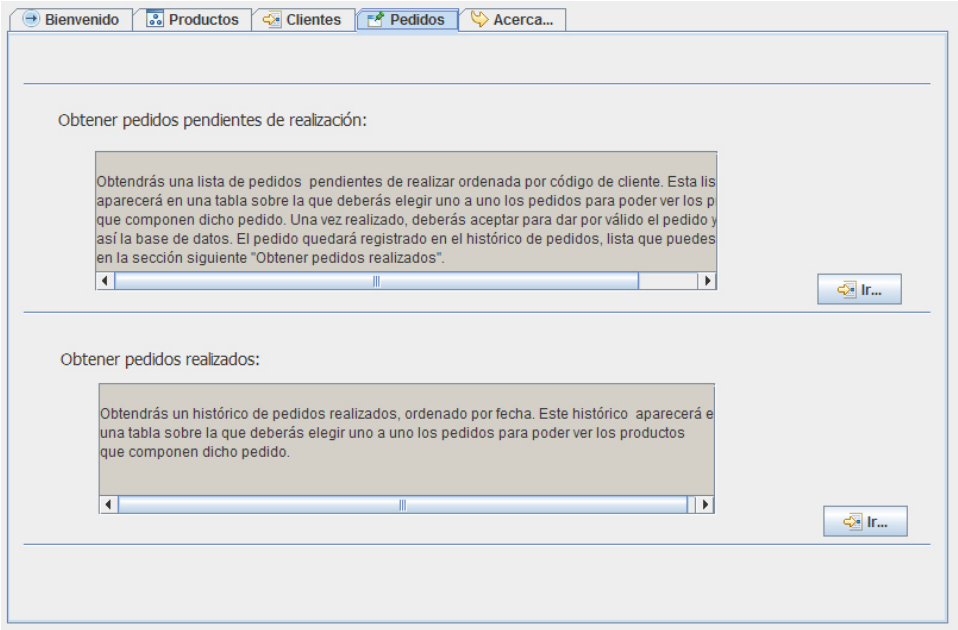


Figura 6.50: Elección de rutina.

La elección de la rutina de realización de pedido lanza una ventana activa que lee de la base de datos, mediante el uso de la clase de apoyo de base de datos *ConectarQcx277*. El criterio de obtención de datos es una cláusula modificadora SQL *WHERE* que extrae registros donde el campo *stmp2* está a *NULL*. Este campo se actualiza con la función SQL *NOW()* en el momento en el que se da el pedido por realizado. La vista generada en la tabla es estática a petición del cliente, pues la herramienta permite conjugar las diferentes vistas generables desde KnoE. En este sentido, es posible abrir varias tablas, como clientes, pedidos y productos a la vez.

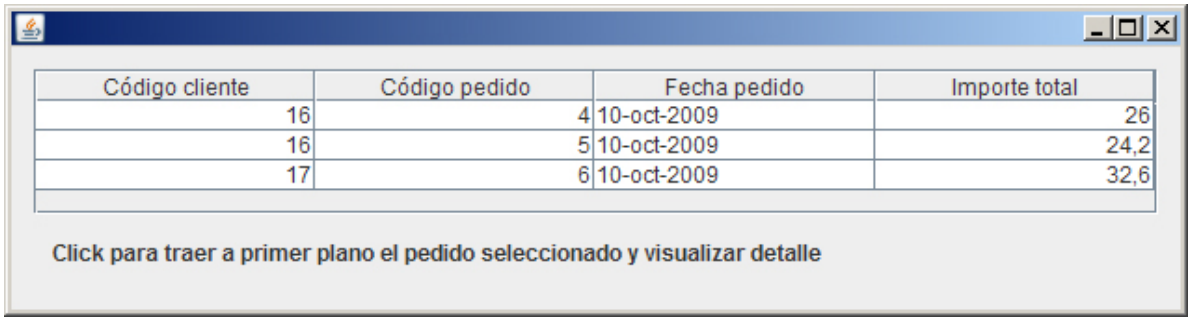


Figura 6.51: Tabla de pedidos pendientes.

Una vez que se pulsa sobre alguno de los registros mostrados en la tabla, la rutina conduce a otra tabla, en la que se muestra el detalle de los productos que componen el pedido, la cantidad y el importe.

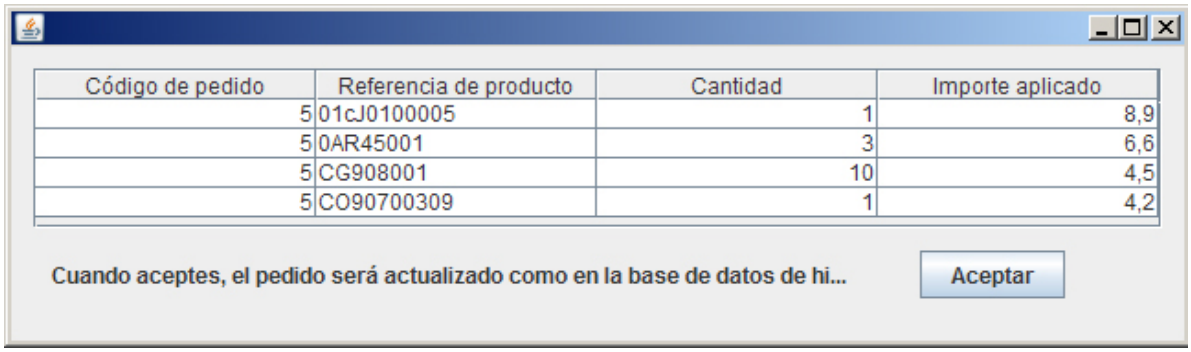


Figura 6.52: Detalle de pedido pendiente.

Al pulsar sobre aceptar, se ejecuta la rutina de actualización comentada en párrafos anteriores. El proceso termina con la aparición de una ventana de información de resultado.

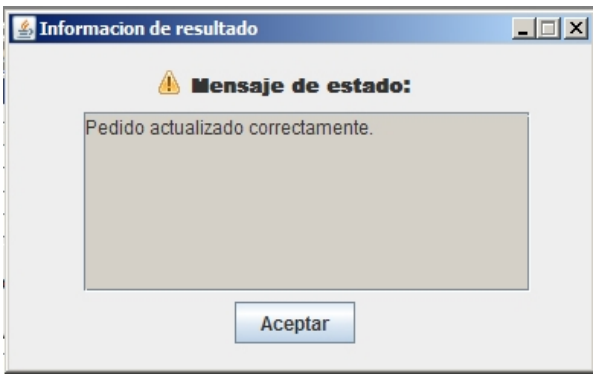


Figura 6.53: Ventana de resultado.

Secuencia

Desde la capa superior, se lanza el fragmento de código mostrado en el texto 6.5.

```
ConectarQcx277 con = new ConectarQcx277();
Object[][] data = con.selectPedido(con.construccionStatementSelectPedido(true), true);
if (data != null) {
        DialogoPedido dp = new DialogoPedido(true, data);
        dp.setVisible(true);
} else {
    DialogoACK dial = new DialogoACK(29);
    dial.setVisible(true);
 }
```

Texto 6.5: Fragmento de KnoE.java: generación de vistas de pedidos pendientes.

La función *selectPedido* de *ConectarQcx277* devuelve la tabla con los datos necesarios.

La clase *DialogoPedido* del paquete *soporteGrafico.pedido,* tiene la posibilidad de, mediante el parámetro boolean, de distinguir entre pendiente o histórico. En este caso, la tabla pasada como parámetro contiene pedidos pendientes, y se informa con el citado boolean del tratamiento que se debe dar a dicha tabla.

La tabla visible de *DialogoPedido* detecta eventos *mouseClicked,* de forma que genera una nueva tabla *DialogoPedido* con el detalle de cada pedido cuando se pulsa sobre la fila.

La secuencia que se lanza desde la ventana padre *DialogoPedido* al pulsar Aceptar consiste en el lanzamiento del método *updatePedido* de *ConectarQcx277*. La función se limita a actualizar la marca de tiempo *ts_2* según el código de pedido que se le pasa por la línea de parámetros.

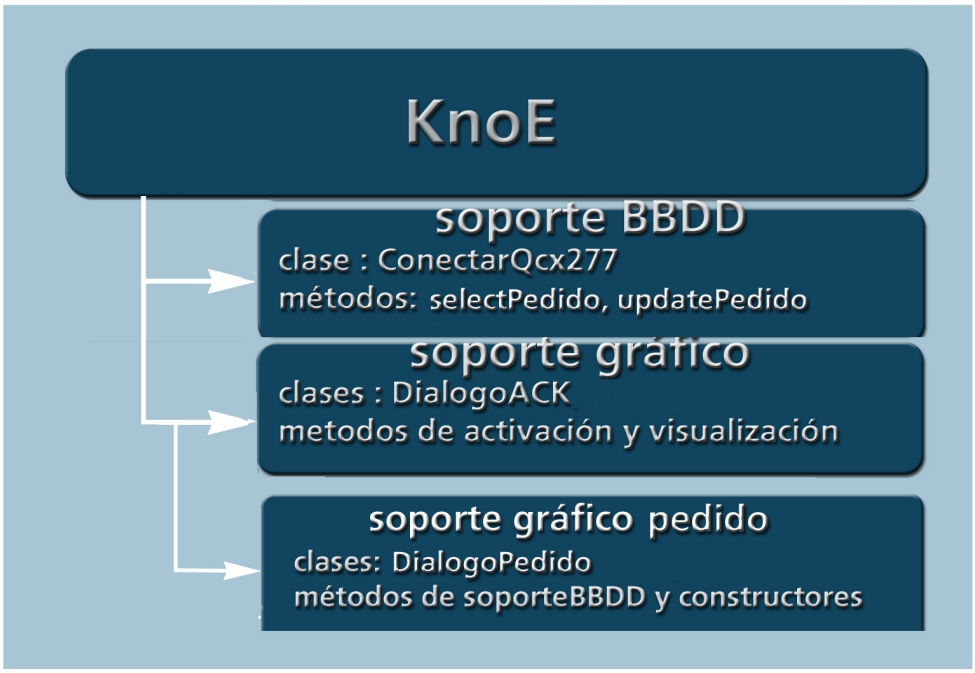


Figura 6.54: Clases/funciones usados en pedidos.

6.3.4.2 Vistas de pedido realizado

Para obtener el histórico de pedidos, partimos del componente mostrado en la figura 5.30. El modelo de proceso es similar al descrito en el apartado anterior, pero sin el acceso final a base de datos para actualizar estados. En esta ocasión, la cláusula SQL *WHERE* buscará asientos en la tabla correspondiente en los que exista una fecha concreta para el atributo *stmp2*.

Figura 6.55: Histórico de pedidos.

De forma similar, la tabla generada es activa y, pulsando sobre alguna de las filas se mostrará el detalle de pedido. Se muestra en esta vista, como atributo añadido a la vista del apartado anterior, la fecha de ejecución del pedido.

Secuencia

El algoritmo seguido en esta rutina es idéntico al explicado en la secuencia del apartado anterior, con la salvedad de que no se genera ninguna escritura en base de datos cuando se pulsa el botón aceptar de *DialogoPedido*.