

**PROGRAMACIÓN CON  
ADO.NET/C#  
VOLUMEN II**



**Ian Marteens**

**2007**



# ÍNDICE DE MATERIAS

<b>INTRODUCCIÓN</b>	<b>VII</b>
ORGANIZACIÓN DEL CONTENIDO	VII
ACERCA DE LA BASE DE DATOS	VII
<b>00. DISEÑO DE LA BASE DE DATOS</b>	<b>1</b>
INSTALACIÓN DE LA BASE DE DATOS	1
ELEMENTOS COMUNES: CLAVES PRIMARIAS	3
CLAVES VERSUS CÓDIGOS	5
COLUMNAS COMUNES	6
REGLAS SIMPLES BASADAS EN EL PAÍS	7
CLIENTES, DIRECCIONES Y ATRIBUTOS	9
EL MODELO DE PRODUCTOS	10
FACTURAS	11
REGLAS DE NEGOCIO Y PRECIOS DE VENTA	11
<b>01. GESTIÓN DE VENTANAS</b>	<b>12</b>
RUBIAS, MORENAS Y PELIRROJAS	12
GESTORES DE VENTANAS	13
GESTORES DE MENÚS	13
REFLEXIONEMOS...	15
MANIOBRAS ORQUESTALES DENTRO DE UN MENÚ	17
EL GESTOR EN MARCHA	18
<b>02. VENTANAS NO MODALES</b>	<b>20</b>
EL GESTOR DE VENTANAS	20
APLICACIONES MDI	20
INTEGRACIÓN CON EL GESTOR DEL MENÚ	23
<b>03. MÓDULOS DE DATOS</b>	<b>26</b>
LA ORGANIZACIÓN DEL PROYECTO	26
INTERFACES GENÉRICAS DE ACCESO A DATOS	27
COMPONENTES E INSTANCIAS GLOBALES	29
<b>04. VENTANAS DE NAVEGACIÓN</b>	<b>31</b>
¿QUÉ ES UNA “VENTANA”?	31
FRAGMENTOS DE UNA VENTANA DECONSTRUIDA	32
¿EDICIÓN... SOBRE UNA REJILLA?	33
CONTROLES DE USUARIO	33
CAMBIOS EN LOS GESTORES DE MENÚS Y VENTANAS	34
GUARDAR LOS CAMBIOS	36
HERENCIA VISUAL	37
<b>05. ALGORITMOS BÁSICOS DE NAVEGACIÓN Y EDICIÓN</b>	<b>39</b>
PROPIEDADES VISIBLES EN TIEMPO DE DISEÑO	39
EL ALGORITMO BÁSICO DE INICIALIZACIÓN	41
¿HAY CAMBIOS?	41
EL ALGORITMO BÁSICO DE GRABACIÓN	41
CONCILIACIÓN EN PRESENCIA DE IDENTIDADES	42
ACTUALIZACIÓN DEL ESTADO DE ELEMENTOS VISUALES	43

<b>06. CONSULTAS SOBRE TABLAS SENCILLAS</b>	<b>45</b>
UNA CONEXIÓN CENTRAL	45
CONFIGURACIÓN DE LOS ADAPTADORES	47
CREACIÓN DE LOS CONJUNTOS DE DATOS CON TIPOS	49
IMPLEMENTACIÓN DEL ACCESO PARA LECTURA	50
LAS VENTANAS DE PAÍSES E IMPUESTOS	51
<b>07. RECUPERACIÓN DE REGISTROS POR PÁGINAS</b>	<b>53</b>
UNA INTERFAZ GENÉRICA PARA LA RECUPERACIÓN POR GRUPOS	53
GENERACIÓN DINÁMICA DE CONSULTAS	54
MÁS REGISTROS, POR FAVOR...	55
INICIALIZACIÓN	58
CALMA Y SANGRE FRÍA	60
<b>08. EDICIÓN MODAL</b>	<b>61</b>
DIÁLOGOS MODALES Y OPERACIONES SOBRE DATOS	61
EL MOMENTO MÁS ADECUADO	62
EL GESTOR DE VENTANAS ASUME RESPONSABILIDADES	62
UN NOBLE LINAJE DE EDITORES MODALES	65
ENLACES CON LA VISTA DE DATOS	67
<b>09. CONTROLES PARA NAVEGACIÓN Y EDICIÓN</b>	<b>69</b>
AUTOMATIZANDO LA EDICIÓN	69
CREACIÓN DE REGISTROS	70
BORRADOS	71
REAPROVECHANDO BOTONES	71
<b>10. REGLAS DE NEGOCIO DURANTE LA EDICIÓN</b>	<b>73</b>
UN ADAPTADOR PARA PRODUCTOS	73
REGLAS DE NEGOCIO Y CONJUNTOS DE DATOS	74
¿QUIÉN VIGILA AL VIGILANTE?	76
<b>11. ALMACENAMIENTO EN CACHÉ</b>	<b>78</b>
UNA CACHÉ DE CORTA VIDA	78
PERSISTENCIA XML	79
EL MOMENTO DEL ÚLTIMO CAMBIO	81
COMPARANDO CON LA VERSIÓN LOCAL	82
UN SEGURO DE VIDA	84
<b>12. BÚSQUEDAS</b>	<b>85</b>
UN REPASO A LA LISTA DE PALABRAS	85
FUNCIONES DE TABLAS EN SQL SERVER	86
DESMENUZANDO CADENAS	87
<b>13. NAVEGACIÓN MAESTRO/DETALLES</b>	<b>90</b>
CLIENTES Y ATRIBUTOS	90
PRIMERA APROXIMACIÓN	91
PROPAGACIÓN DE FILTROS	94
EXTENSIONES DEL SISTEMA DE PAGINACIÓN	95
EL PRIMER PASO HACIA PROTEUS	98
<b>14. BÚSQUEDAS BASADAS EN ATRIBUTOS</b>	<b>100</b>
SINTAXIS DE LAS CADENAS DE BÚSQUEDA	100
ENUMERADORES	101
GENERACIÓN DE CONSULTAS	103

<b>15. RESOLUCIÓN DE REFERENCIAS</b>	<b>105</b>
COLUMNAS BASADAS EN EXPRESIONES	105
EL ORDEN DE LAS LECTURAS	106
COMBOS PARA LA EDICIÓN DE REFERENCIAS	107
<b>16. EDICIÓN DE ENTIDADES CON ESQUEMAS ABIERTOS</b>	<b>109</b>
ENLACE A DATOS ARTESANAL	109
TRANSACCIONES	111
GRABACIÓN MAESTRO/DETALLES	113
PROPAGANDO LOS CAMBIOS A LA VENTANA DE NAVEGACIÓN	114
<b>17. VALIDACIÓN</b>	<b>116</b>
MÁS RESPONSABILIDADES PARA IWINDOW	116
EL PROVEEDOR DE ERRORES	117
VERIFICACIÓN MEDIANTE EXPRESIONES REGULARES	117
TRAMPAS DE ERRORES	119
<b>18. SELECCIÓN DE REGISTROS EN TABLAS GRANDES</b>	<b>120</b>
CREACIÓN DE FACTURAS Y SELECCIÓN DE CLIENTES	120
EL DÍALOGO DE EDICIÓN DE FACTURAS	121
EL CLIENTE QUIERE SUS GALLETAS	122
UN DÍALOGO PARA LA SELECCIÓN	123
TRÁFICO DE DATOS	125
CUANDO SE TRATA DE UN NUEVO CLIENTE...	126
<b>19. GRABACIÓN DE FACTURAS</b>	<b>127</b>
EL ALGORITMO DE GRABACIÓN	127
UNA ENMARAÑADA CORTINA DE DISPAROS	128
ADAPTACIONES A LA EDICIÓN MAESTRO/DETALLES	130
COMPROBACIONES SOBRE LÍNEAS DE FACTURAS	131
NOTIFICACIONES ENTRE VENTANAS	132
NOTIFICACIONES DURANTE LAS ACTUALIZACIONES	134
CENTINELA ALERTA	136
<b>20. EXTENSIBILIDAD MEDIANTE SCRIPTS</b>	<b>137</b>
EN BUSCA DEL VALLE DE LA EXTENSIBILIDAD PERDIDA	137
EXTENSIBILIDAD MEDIANTE CÓDIGO DINÁMICO	138
MOTORES DE SCRIPT	138
UN, DOS, TRES, PROBANDO...	141
REGLAS PARA DETERMINAR DESCUENTOS	143
EL MOTOR DE DESCUENTOS	145
APLICANDO LAS REGLAS	146
<b>21. NAVEGACIÓN SOBRE FACTURAS</b>	<b>148</b>
ES NATURAL NAVEGAR SOBRE UN ENCUENTRO	148
CAMBIO DE SENTIDO	150
BÚSQUEDA Y NAVEGACIÓN SOBRE FACTURAS	150
LA VERSIÓN MÁS RECIENTE DE LA FACTURA	151
<b>22. MENSAJES DE ERROR</b>	<b>153</b>
RESTRICCIONES CON NOMBRE	153
ADAPTANDO LOS MENSAJES AL CONTEXTO	154
<b>23. RECUPERACIÓN DE ERRORES DE CONCURRENCIA</b>	<b>156</b>
CASOS ESPECIALES	156

CÓMO DISTINGUIR LOS ERRORES DE CONCURRENCIA	157
RELECTURA Y MEZCLA	158
ADVERTENCIAS Y ALTERNATIVAS	161
<b>24. IMPRESIÓN GENÉRICA</b>	<b>162</b>
DIBUJANDO SOBRE LA IMPRESORA	162
LA VISTA PRELIMINAR	163
IMPRIMIENDO UN CONJUNTO DE DATOS	164
LA INTERFAZ IPRINTABLE	167
<b>25. RESUMEN</b>	<b>170</b>
VENTANAS ACOPLADAS	170
EXTENSIONES AL GESTOR DEL MENÚ	171
<b>A. CREACIÓN DE LA BASE DE DATOS</b>	<b>173</b>
<b>B. RELACIONES ENTRE TABLAS</b>	<b>183</b>
<b><u>INDICE ALFABÉTICO</u></b>	<b><u>185</u></b>

---

# INTRODUCCIÓN

---

NOS TOCA AHORA DAR USO PRÁCTICO A LOS conocimientos teóricos adquiridos en las tres series de ejercicios anteriores. Para ello, vamos a desarrollar una pequeña aplicación de facturación. Esta vez no huiremos de los problemas prácticos y *bugs* que podamos encontrar: no nos interesa tanto simplificar para poder explicar como dar una idea real del funcionamiento de esta versión de C# y Visual Studio.

## Organización del contenido

Hay otra diferencia importante respecto al Volumen I: los ejercicios están organizados en una cadena lineal y continua, como exige el contenido de esta parte del curso. Una de las consecuencias de esta organización es que un error que se nos escape en un ejercicio nos perseguirá hasta el final de la serie. Es muy fácil asignar un valor equivocado a una propiedad en tiempo de diseño; lo realmente difícil es descubrirlo. Si el proyecto que está desarrollando se “atasca” al llegar al llegar a determinado nivel, no se deprima ni pierda demasiado tiempo: parta de una copia fresca del ejercicio y continúe a partir de ese punto.

La forma más sencilla de pasar de un ejercicio al siguiente es ésta:

- Copie el ejercicio anterior en un nuevo directorio.
- Cambie el nombre de los ficheros que dan nombre al proyecto y a la solución. Por ejemplo, si va a trabajar con el proyecto *Ej03*, copie el proyecto *Ej02* en un directorio llamado *Ej03* y cambie el nombre a todos los ficheros llamados *Ej02* sin importar la extensión que tengan.
- Importante: active Visual Studio y cierre todas las ventanas. En el menú *Edición|Buscar* encontrará un comando para buscar y reemplazar cadenas en todos los ficheros de un directorio. Utilice este comando para cambiar todas las apariciones del nombre del viejo proyecto. En nuestro ejemplo, encuentre todas las ocurrencias de *Ej02* y reemplácelas por *Ej03*.

## Acerca de la base de datos

En mi opinión, es peligroso enseñar determinadas técnicas de acceso a datos con una base de datos esmirriada. Una técnica ineficiente aplicada a una base de datos pequeña puede ofrecer una falsa sensación de seguridad y provocar desastres irreparables. Por este motivo, he intentado generar una base de datos con un número de registros razonables, y con datos más o menos ajustados a lo que encontraremos en la práctica. Con esto último quiero decir que no nos interesa trabajar con una tabla de clientes en la que se repitan los mismos datos, excepto en la clave primaria. Estas repeticiones distorsionarían los resultados de las búsquedas basadas en índice, por ejemplo.

Pero al ofrecer una base de datos buena, se corre un peligro importante: que alguien piense que hay algo real en los datos. Cualquier parecido con la realidad, es una simple y cochina coincidencia. Y para alejar cualquier duda, describiré cómo he generado los registros con los que vamos a trabajar.

La tabla de clientes ya se ha utilizado en otros cursos a distancia de IntSight, y es la que se puede prestar a más confusión. Para generar sus registros, partimos de una lista de nombres y otra de apellidos; por supuesto, estos nombres y apellidos son “reales” tomados por separado. Luego, los combinamos aleatoriamente, otorgando la misma probabilidad de aparición a cada uno de ellos. Esto no es lo que sucede en la vida real, en la que encontramos nombres “populares” y unos apellidos son más frecuentes que otros. Pero si hubiésemos aplicado una distribución de probabilidad no uniforme, sería más difícil demostrar que no se trata de datos reales; es fácil demostrar el carácter artificial de los nombres analizando sus frecuencias relativas.

En la base de datos original, las direcciones se almacenaban en una tabla independiente, y podíamos asignar más de una dirección a cada cliente. En esta versión, cada cliente sólo tiene una dirección, almacenada dentro del mismo registro.

Las direcciones originales se obtuvieron creando listas de calles imaginarias divididas por ciudad, considerando cuatro ciudades españolas. A cada calle le asignamos un rango posible de números de portal y de códigos postales. Luego mezclamos, sin agitar, estas listas y creamos las direcciones finales. Los números de teléfonos se crearon aleatoriamente, aunque a cada uno se le añadió el prefijo de la ciudad correspondiente. Y algo parecido hicimos para los números de identidad de los clientes: generamos valores aleatorios, y luego calculamos la letra de control de redundancia que correspondería en la realidad. Verá que algunos de los números comienzan con la letra *X*, que se utiliza en España para los permisos de residencia de extranjeros.

Más conflictiva resultó ser la tabla de productos. Partimos de una tabla de datos sobre libros utilizada en la demo de Classique, el motor de comercio electrónico de IntSight. Al migrar los registros, sin embargo, se perdieron muchas de las columnas almacenadas en Classique. De hecho, sólo han sobrevivido el nombre de cada libro y su precio orientativo de venta al público. El coste se ha elaborado aleatoriamente, mediante un valor que oscila alrededor del 70% del precio de venta al público.

Ian Marteens  
[www.marteens.com](http://www.marteens.com)  
Madrid, diciembre de 2007

# EJERCICIO 00

## Objetivos

Diseño de la base de datos

## Técnicas introducidas

Ejecución de *scripts*, tipos de datos en SQL Server

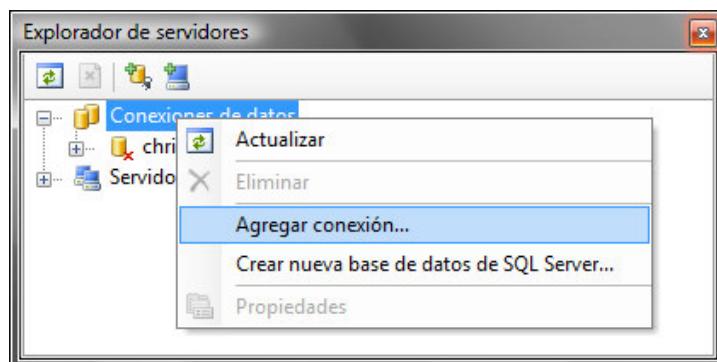
DENTRO DEL DIRECTORIO CORRESPONDIENTE a este ejercicio, encontrará un fichero llamado *curso.mdf*, que es el que utilizaremos como base de datos de ejemplo para todos los ejercicios. En el mismo directorio he ubicado un par de ficheros con código fuente SQL, que contienen las instrucciones necesarias para crear la base de datos, como método alternativo, y de paso poblarla con registros. Estas notas indican cómo instalar la base de datos, a la vez que se explica el diseño relacional de la misma.

## Instalación de la base de datos

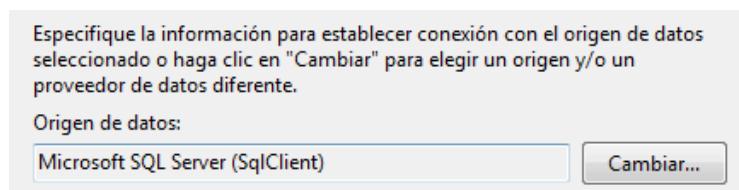
Mi recomendación es utilizar SQL Server 2005, o SQL Server Express 2005 para seguir el curso. Tengo varios motivos:

- El software ya viene incluido con Visual Studio, al menos en la versión Professional. Es fácil, en cualquier caso, obtener una versión de evaluación del mismo.
- SQL Server 2005 soporta bloques de transacciones declarativas, cuando se accede al mismo desde ADO.NET. Las transacciones declarativas simplificarán enormemente la aplicación que desarrollaremos a lo largo de los ejercicios de esta serie.
- Sobre todo, nos será útil la posibilidad de abrir directamente un fichero de datos, como nuestro *curso.mdf*, como si se tratase de un fichero de Access. Esto nos dará mucha libertad: si destruimos la base de datos, sólo tenemos que sustituir el fichero de datos con una copia del fichero original.

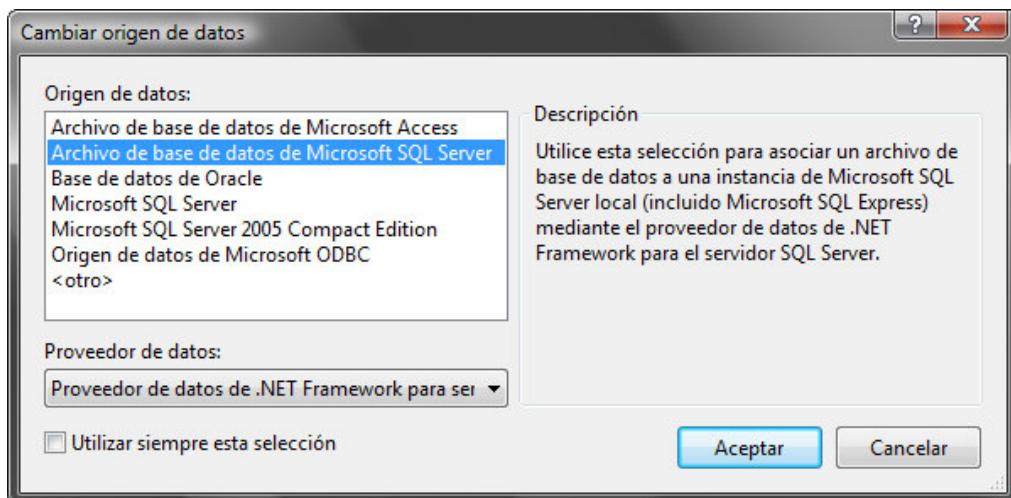
El primer paso para poder utilizar la base de datos es registrarla en el Explorador de Servidores de Visual Studio. Active esta ventana, y ejecute el comando *Agregar conexión* del menú de contexto del nodo principal:



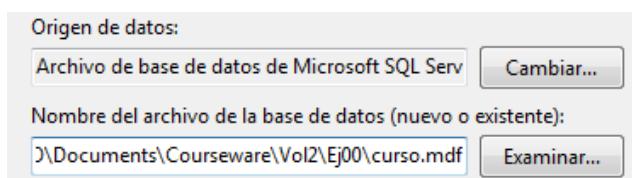
En el diálogo que aparece, debe pulsar el botón *Cambiar* adyacente al control que determina el origen de los datos:



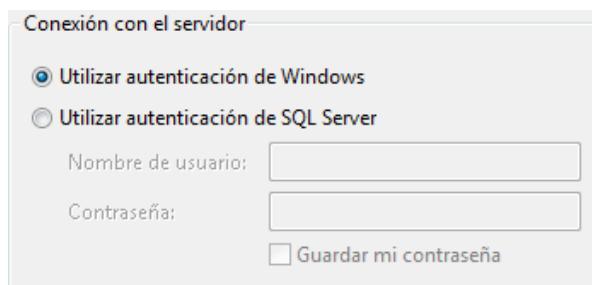
En realidad, queremos usar el proveedor que nos permite trabajar directamente con el fichero de datos:



Al aceptar los cambios, verá que el diálogo original ha modificado su diseño para reflejar las nuevas opciones de configuración. Tenemos ahora un control para introducir el nombre del fichero de datos. Seleccione el fichero incluido en el directorio *Ej00* del curso:



Para evitar el almacenamiento de contraseñas, en todos los ejemplos del curso utilizaremos la seguridad integrada de SQL Server:



Sólo nos queda aceptar el diálogo, para que Visual Studio cree un nodo correspondiente a la conexión, en el Explorador de Servidores.

Si está obligado, o si prefiere, usar un servidor remoto, o un servidor con SQL Server 2000, tendrá que utilizar el sistema tradicional para crear la base de datos en el correspondiente servidor. En tal caso, tendrá que usar la herramienta cliente de SQL Server para ejecutar los dos ficheros de *script* incluidos también en el directorio *Ej00*. El directorio de las bases de datos viene especificado en la instrucción **create database** incluida en el fichero *basedatos.sql*:

```
create database AdoNet
on primary
( name = PrimaryData,
  filename = 'C:\SqlData\AdoNet.mdf',
  size = 2 )
log on
( name = LogFile,
  filename = 'C:\SqlData\AdoNet.ldf' )
go
```

38

Asegúrese de que el directorio mencionado existe, o modifique las rutas donde se ubicarán los ficheros. Ejecute luego el contenido de los dos ficheros: primero, el ya mencionado *basedatos.sql*, y luego *registros.sql*.

Si no dispone de la herramienta interactiva de SQL Server, tendrá que utilizar la aplicación *osql.exe*, desde la línea de comandos del sistema operativo. Para ejecutar el contenido del fichero *basedatos.sql* en un servidor con seguridad integrada, teclee lo siguiente:

```
osql -E -Sservidor -ibasedatos.sql
```

La opción *-E* es la que indica que usaremos la seguridad integrada. Si debe usar un nombre y una contraseña explícita, utilice esta otra variante:

```
osql -Uusuario -Pcontraseña -Sservidor -ibasedatos.sql
```

Asegúrese de usar las mayúsculas y minúsculas correctamente, porque *osql* distingue entre ambos tipos de letra. Y no olvide crear una conexión desde Visual Studio que haga referencia a la nueva base de datos.

## Elementos comunes: claves primarias

A medida que vayamos progresando en el curso, iremos examinando con detalle cada una de las tablas presentes en la base de datos. En este momento sólo quiero explicar los motivos tras algunas de las decisiones de diseño.

- Prácticamente todos las columnas en todas las tablas han sido declaradas con la ayuda de tipos de datos definidos por el usuario. La excepción son las columnas de tipo **rowversion**...

- 48 ... porque SQL Server no permite definir tipos de datos basados en **rowversion** o **timestamp**. A diferencia de lo que ocurre en otros sistemas de bases de datos, incluyendo la versión 2005 del propio SQL Server, la definición de tipos de datos de usuarios en SQL Server 7 y 2000 es un mecanismo bastante primitivo e incompleto. Estos tipos se reducen a ofrecer un sinónimo para los tipos básicos y poco más. Aunque debemos asociar al tipo una cláusula **null** o **not null**, al declarar columnas pertenecientes al nuevo tipo podemos ignorar dicha cláusula.

A pesar de ello, es útil definir estos tipos de datos y utilizarlos consistentemente en toda la base de datos, y la explicación está en los tipos que requieren una especificación de tamaño, escala o precisión. Estas son, por ejemplo, las definiciones de los tipos usados en nombres de ciudades y porcentajes:

```
execute sp_addtype T_CIUDAD, 'varchar(40)', 'not null'  
execute sp_addtype T_PCT, 'numeric(5,2)', 'not null'
```

Gracias a estas declaraciones, un parámetro tan importante y a la vez fácil de olvidar como el número máximo de caracteres o la precisión de nuestros porcentajes, se almacena en un único lugar del fichero de creación de la base de datos. Si tenemos que modificar la capacidad de los nombres de ciudades, sólo tenemos que modificar la definición del tipo... y volver a crear la base de datos.

Es importante que comprenda esto: una vez que ha creado tablas que utilizan un tipo de datos, si realiza una modificación en el tipo, ésta no se propagará a las tablas que lo usan.

- Todas las claves primarias son claves artificiales, formadas siempre por una sola columna de tipo entero y con el atributo **identity** aplicado.

Casi todos los libros y profesores de informática, al explicar el modelo relacional, utilizan en sus ejemplos tablas con claves primarias que corresponden a atributos reales de las entidades modeladas. Si hay que crear una tabla de personas, la clave primaria por antonomasia es el número de la seguridad social, que presuntamente nunca se repite... o aun peor, la combinación del nombre y los apellidos. Si se trata de una tabla de entidades bancarias, se utiliza el código de cuatro dígitos de la entidad.

Esta es una idea horriblemente mala: un atributo presente en el modelo real está sujeto a cambios. El banco puede fusionarse, y aunque se mantenga reservado el código original, las cuentas pueden recibir una nueva numeración. Una persona puede darle un identificador equivocado, y cinco meses más tarde, cuando ha comprado media tienda, darse cuenta del error y pedirle que lo corrija.

¿Y qué hay de malo en una clave primaria modificable? Aquí tiene algunos de los problemas:

- Una de las funciones principales de las claves primarias es relacionar registros por medio de la integridad referencial. Los pedidos de un cliente hacen referencia a la clave primaria del cliente. Si esta clave cambia, hay que propagar el cambio a todos los pedidos. Es cierto que, a partir de la versión 2000, SQL Server permite que esta propagación tenga lugar de forma automática, pero...
- ... resulta que SQL Server, como muchos otros sistemas de bases de datos, utiliza índices agrupados (*clustered indexes*) para implementar las claves primarias. Estos índices son los responsables del almacenamiento físico de los registros de la propia tabla; en particular, esto implica que sólo puede haber un índice agrupado por tabla. Si modificamos la clave primaria de un registro, es muy probable que éste pase a pertenecer a otra página física, y esto provoca operaciones de entrada y salida adicionales. Una modificación en cualquier otro campo se implemente más eficientemente, pues no hay necesidad de cambiar la página donde reside el registro.
- Salvo contadas excepciones, las interfaces de programación para bases de datos utilizan las claves primarias para sincronizar los registros de la base de datos con las copias locales de los mismos que almacenan las aplicaciones clientes. Un registro cuya clave primaria puede ser modificada se convierte en un blanco móvil, que complica extraordinariamente el tratamiento de las actualizaciones concurrentes.
- En SQL Server, en particular, se hace muy difícil programar *triggers* de modificación para tablas cuyas claves primarias son modificables. SQL Server permite acceder a los valores iniciales y los valores propuestos de los registros afectados mediante sendas tablas temporales, llamadas *inserted* y *deleted*. Si se han modificado las claves primarias, es prácticamente imposible establecer la relación entre los registros de estas tablas.

Otra característica de las claves “reales” es que tienden a propagarse a las entidades dependientes. El ejemplo clásico es el de las cuentas bancarias: el programador comienza definiendo el código de banco como la clave primaria de la tabla de entidades bancarias. En la tabla de sucursales, la clave primaria debe ser la combinación del código de banco y el código propio de la oficina. Al llegar a la tabla de cuentas, la clave primaria estará formada por el código de banco, más el de sucursal, más el propio código de cuenta, y comenzará a parecerse a esos dragones que pasean los chinos en sus carnavales... quiero decir, por su longitud, no porque esté llena de chinos.

Estas claves longanizas, además de padecer las mismas enfermedades que todas las claves basadas en columnas “reales”, no se llevan bien con muchos controles visuales. Por desgracia, muchos controles con enlace a datos se programan de forma tal que las propiedades suyas que hacen referencias a campos sólo admiten un campo, no una lista de ellos. Claro está, se trata de un problema artificial, creado por un programador poco previsor, ¡pero es lo que hay! El ejemplo clásico son los combos, cuando se utilizan para “traducir” referencias a otras tablas. En esos casos, la propiedad *DataSource* debe indicar la tabla de referencia, y en *ValueMember* se debe indicar la clave de esa tabla. Pues bien: *ValueMember* sólo nos permite seleccionar el nombre de una sola columna.

Por todas estas consideraciones verá que hemos seguido un patrón a rajatabla al definir todas las tablas de la base de datos. Suponga que tenemos que crear una tabla para almacenar cierto tipo de entidades patafísicas conocidas genéricamente como “chirimbotos”. Esta sería la tabla (he traducido los nombres de los tipos de datos para simplificar):

```
create table Chirimbotos (
    IDChiribolo integer      not null identity,
    /* El resto de las columnas */

    Creacion      datetime     not null default current_timestamp,
    TS            rowversion   not null,
```

```
primary key (IDChirimbolo),
/* Otras restricciones */
)
```

Preste atención a los siguientes detalles:

- El nombre de la tabla está en plural. ¿Cuántas veces se ha tenido que detener a pensar si la tabla de facturas se llama *Facturas* o *Factura*? Hay que decidirse por uno de los dos convenios posibles, y lo natural es que una tabla, por almacenar múltiples registros, reciba un nombre en plural.
- A veces, el nombre resultante es demasiado largo, y el programador cae en la tentación de usar el singular para ahorrarse un par de letras. ¡Evítelo! Esas chorraditas del tipo “toda regla tiene su excepción” son malas justificaciones de, como diría A. Schwarzenegger, *girlie men* (nenazas).
- El nombre de la columna de la clave primaria es el mismo de la tabla, pero en singular y con el prefijo *ID*.
- La clave primaria es una columna de tipo entero. Hay gentes para quienes un **integer** parece muy poca cosa. Si es su caso, entérese: un humilde **integer** sirve para distinguir entre ¡dos mil millones de clientes! ¿Tiene su empresa dos mil millones de clientes? Entonces no sé por qué demonios sólo le he cobrado la porquería de precio de este curso a distancia. Renegociemos.
- La clave primaria está marcada con el atributo **identity**, para que reciba valores consecutivos de forma automática. Es una clave artificial, ¿no? Entonces el usuario ni siquiera tiene que enterarse de que existe. Sobre este tema, abundaremos en breve.
- Hay un par de columnas adicionales: una para la fecha de creación, y otra más esotérica que he bautizado *TS*. Comprenderemos el propósito de ambas dentro de un momento.

**NOTA**

Una alternativa muy razonable al uso de identidades para las claves primarias, es usar el tipo de datos **uniqueidentifier**: un entero de 128 bits con un algoritmo de generación que garantiza la unicidad estadística. Una de las ventajas de **uniqueidentifier** es que garantiza la unicidad incluso si mezclamos el contenido de dos bases de datos; es algo a tener en cuenta si tenemos necesidad de replicar la base de datos y trabajar con varias copias locales de la misma en distintas oficinas. También simplificarían los algoritmos de inserción de registros. La principal desventaja principal es el tamaño necesario para estas columnas; no tanto por el espacio que ocupan en las páginas de datos, sino porque en cada página del índice entrarán menos claves, con lo que las búsquedas serán algo más lentas.

## Claves versus códigos

Siempre que expongo en público mi teoría sobre las claves artificiales, alguien levanta la mano para objetar. Engrosando la voz, para dejar bien claro que no es una nenaza, lo que argumenta se reduce más a menos a que, si por él fuera, los códigos estarían prohibidos por decreto, pero ya sabes (es decir, ya sé), que tiene un cliente que le exige asociar códigos a sus productos. Y claro, el cliente siempre tiene la razón, incluso si se le ocurre pedir una lobotomía...

¿Cuál es el conflicto? No hay ninguno: lo que hay es una vuelta al viejo malentendido que exige pillar el primer atributo no repetido de una entidad para usarlo como clave primaria. ¿Su cliente debe trabajar con códigos? ¡Estupendo! Pero no haga que esos códigos se conviertan en la clave primaria. Los códigos pueden modificarse por mil motivos diferentes. Las claves primarias no deben cambiar.

Hace un par de párrafos escribí que no tiene sentido alguno que el usuario tenga conocimiento de las claves artificiales que estamos usando como claves primarias. La razón es evidente: si el usuario debe utilizar códigos para algunas de las entidades, es preferible añadir una columna con códigos más amigables. Y de ser necesario, puede añadir una restricción **unique** para estos códigos adicionales. En nuestra base de datos, la tabla de países introduce un código de país:

```
create table dbo.Paises (
    IDPais      integer not null identity,
    Codigo      char(2) not null,
    /* ... más columnas ... */
```

```
primary key (IDPais),  
unique          (Codigo),  
/* ... más restricciones ... */  
)
```

**NOTA**

En los ejemplos de este curso, sin embargo, no intentaremos esconder las claves primarias al diseñar las rejillas de datos o los cuadros de diálogo de edición. Como programadores, nos interesa presenciar el comportamiento de las claves durante las operaciones que iremos implementando. Por ejemplo, es interesante ver cómo se propaga el valor asignado a estas columnas por el servidor.

Es cierto que no hay más ejemplos de este tipo en el resto de las tablas, pero he procedido de esta manera para no complicar demasiado el diseño. Permítame, de todos modos, un par de consejos:

- En el caso de la tabla de países, he utilizado un código ya existente: el sufijo de países en nombres de dominio de Internet, y me he arriesgado a declarar la columna con dos caracteres fijos. Lo habitual, en cambio, es declarar columnas de tipo **varchar**, con suficiente espacio para que no se produzca una tragedia si el usuario decide de pronto cambiar el formato de los códigos.
- Si tiene que controlar el formato de los códigos, almacene en algún lugar una máscara o una expresión regular que defina dicho formato. La comprobación puede realizarla el propio servidor de bases de datos, por medio de *triggers*, o la capa intermedia, y es una buena idea que también lo haga la capa de presentación de la aplicación, siempre que no sea demasiado costoso.

Digamos que su cliente trabaja con códigos numéricos de productos, de cinco cifras decimales. Sería un error declarar la columna de los códigos como **numeric(5)**: ¿cuánto apuesta a que, más tarde o más temprano el cliente querrá añadir un prefijo o un sufijo alfabético? Es mejor que defina el código como un **varchar**, con suficiente capacidad, y compruebe que los valores almacenados estén formados por dígitos decimales.

Una última reflexión: la Informática es una técnica con la suficiente edad como para “crear su propia realidad”. Hay muchas exigencias en los pliegos de condiciones de los contratos que no vienen dictadas por necesidades “reales”... sino por viejas y malas costumbres introducidas por aplicaciones anteriores. Preste mucha atención a esta posibilidad, y puede que se ahorre algunos disgustos al diseñar aplicaciones.

## Columnas comunes

Si examina el fichero de creación de la base de datos, verá que hay dos columnas que se repiten en todas las tablas, al final de la lista de columnas de cada una de ellas. Una de las columnas mencionadas se llama *Creacion*, y es de tipo **datetime**: siempre contiene una fecha y una hora. El objetivo se adivina enseguida: gracias a esta columna sabremos en qué momento se ha creado cada registro. Nuestras aplicaciones tratarán este campo como si fuese de sólo lectura, y las columnas *Creacion* recibirán sus valores gracias a la cláusula **default** incluida en sus definiciones:

```
Creacion datetime not null default current_timestamp,
```

En realidad, puede también interesarnos mantener una fecha de última modificación para cada registro. Pero en ese caso tendríamos que programar un *trigger* para las modificaciones en cada tabla. No sería difícil, porque sería una tarea mecánica, pero complicaría un poco el diseño. Los *triggers* mencionados seguirían este esquema:

```
create trigger ccsPaises_u on dbo.Paises for update as  
    update Paises  
        set Modificacion = current_timestamp  
        from Paises p, inserted ins  
        where p.IDPais = ins.IDPais
```

- 333 Otra columna omnipresente es misteriosa *TS*, de tipo *rowversion* o *timestamp*; ambos nombres son aceptados por SQL Server, aunque *timestamp* tiene otro significado dentro del estándar de SQL, por lo que prefiero *rowversion*. Las columnas de este tipo de datos se actualizan automáticamente en dos operaciones:

- Cuando se inserta un registro.
- Cuando se modifica *cualquier columna* del registro.

El valor que recibe una columna de este tipo depende de una variable global e interna de la base de datos, que se puede interpretar como el latido cardiaco de la misma. Cada vez que se inserta o modifica un registro que contiene una columna **timestamp**, se le asigna el valor de esta variable, y el valor de la misma se incrementa.

Suponga que leemos un registro a las 9:00 de la mañana, y que almacenamos los valores de sus columnas. Volvemos a leer el mismo registro a las 10:00. Sabremos con certidumbre que el registro no ha sido modificado durante ese intervalo si su columna de versión de fila sigue conteniendo el mismo valor. Gracias a esta característica, utilizaremos las columnas *TS* para implementar la técnica de control de concurrencia conocida como *bloqueo optimista*.

Existe otra columna candidata a repetirse en cada tabla, pero que no hemos utilizado en nuestra base de datos:

Activo **bit not null default 0**

- 47 El tipo **bit** sirve para declarar columnas de tipo lógico. En este caso, serviría para honrar un principio básico en Informática, Espionaje y en otras disciplinas relacionadas:

- La información vale su precio en oro.

Su principal corolario:

- ¡Nunca borre un registro, al menos mientras pueda evitarlo!

Usted ha estado vendiendo botellas de vino de una cosecha valiosa, y tiene montones de facturas que hacen referencia a este producto. Pero en esta vida todo se acaba. No puede borrar el registro, pues muchos otros registros apuntan a él. Incluso, aunque pudiese asignar nulos en el campo del identificador de producto de las líneas de factura, no le interesaría perder esta información potencialmente útil. ¡Es que todos los nulos son iguales! ¿Por qué no dejamos el registro en su lugar sin más? Muy sencillo: porque el maldito registro ha adquirido la molesta costumbre de aparecer en todas las búsquedas de productos. Alguien tuvo la genial idea de no mostrar en las búsquedas los productos con existencias igual a cero, pero casi provocó un desastre, porque la aplicación debe permitir pedidos para productos sin existencias. ¿Solución? Marque el registro como si estuviese borrado. Ese es precisamente el papel de la columna *Activo*.

Tenga presente que podemos tener problemas con los datos de clientes, en dependencia de la legislación existente. Puede que en su país, en un momento determinado, sea legal marcar como borrado un registro de cliente, para no enviarle publicidad. Pero puede que la ley le exija borrar físicamente todo rastro de información sobre el cliente que se lo pida.

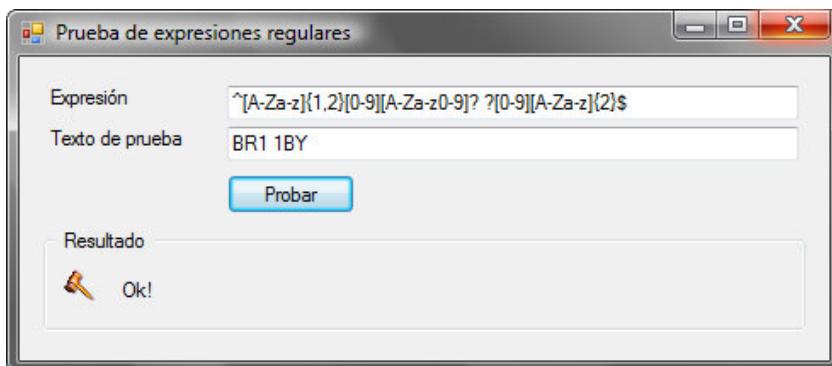
Supongamos, no obstante, que podemos utilizar la técnica de borrados lógicos con todas nuestras entidades. ¿Debemos entonces quitar la operación de eliminación de todas las ventanas de edición? No, no sea extremista. Aunque debe sopesar bien cuándo se puede eliminar o no un registro, recuerde que los seres humanos metemos la pata con frecuencia y, peor aún, no nos gusta que nos lo recuerden. Quizás sea una buena idea permitir que usuarios con privilegios especiales puedan borrar registros que no hayan participado en ninguna actividad “real” de la aplicación.

## Reglas simples basadas en el país

Una vez examinadas las consideraciones de tipo general, ya podemos centrarnos en las peculiaridades del diseño de la base de datos. Vamos a crear una sencilla aplicación de facturación: trabajaremos con clientes, productos y facturas. He simplificado el diseño asumiendo que no hay gastos de envío. El subsistema correspondiente sólo habría añadido complejidad innecesaria, sin introducir técnicas nuevas. En cambio, y hay aquí una contradicción, nuestros registros de direcciones incluirán el país del cliente. He mantenido una tabla de países por dos motivos:

- 1 El corte de este subsistema no habría quedado tan “limpio”. Tengo la intención de reutilizar este ejemplo en el curso de ASP.NET, y en ese caso sí es necesario tener direcciones y gastos de envío más o menos reales.
- 2 La inclusión de una tabla de países nos permitirá exemplificar un tipo muy sencillo de regla de negocio, como veremos a continuación.

Para cada país, almacenamos el formato de los códigos postales y de los números de identificación fiscal. Este formato viene expresado como expresiones regulares, utilizando la sintaxis soportada de manera nativa por la plataforma .NET. En el directorio *Ej00/RegExp* de esta parte del curso, encontrará el código fuente de una sencilla aplicación que puede usar para comprobar la sintaxis de expresiones regulares, y para ver qué tal funcionan con cadenas de prueba. Este es su aspecto:



**NOTA**

No debemos confundir expresiones regulares y máscaras. Una máscara indica qué caracteres se aceptan en cada posición de una cadena, y carece de recursos para indicar repeticiones o elementos opcionales, por lo que es una técnica menos potente de validación. Como compensación, es más sencillo recordar la sintaxis de máscaras, y es mucho más fácil de implementar que, durante la edición, la aplicación introduzca automáticamente los caracteres de uso obligatorio, como los separadores de fecha dentro de una fecha.

Veamos algunos ejemplos sencillos. El formato más frecuente de código postal consiste en cinco dígitos consecutivos. Es el usado en casi toda la Unión Europea, incluyendo a España, y puede representarse con la siguiente expresión regular:

`^ [0-9] { 5 } $`

La expresión encerrada entre corchetes se interpreta como cualquier carácter entre el 0 y el 9. Dicho sea de paso, el operador **like** de SQL Server permite indicar este tipo de conjuntos de caracteres en sus patrones. A continuación de los corchetes viene un indicador de repeticiones: el dígito decimal se debe repetir exactamente cinco veces. Por cierto, SQL Server *no* soporta esta construcción.

Pero quizás lo más novedoso, para quien no ha trabajado nunca con expresiones regulares sean los caracteres que aparecen al principio y al final de la expresión. El acento circunflejo indica que lo que viene a continuación debe encontrarse al principio de la línea, y el signo de dólar, que después de los cinco dígitos debe venir el fin de línea. Estas indicaciones son necesarias porque, por lo general, las expresiones regularse se utilizan para localizar determinada construcción dentro de una cadena más grande. Por este motivo, verá que casi todas las expresiones regulares de nuestra tabla de clientes comienzan y terminan con los caracteres especiales mencionados.

El formato de cinco dígitos también se utiliza en los Estados Unidos, pero en este país se utiliza en ocasiones un formato más largo: después de los cinco dígitos, puede venir un guión y cuatro dígitos más. Esta es la expresión regular correspondiente:

`^ [0-9] { 5 } (- [0-9] { 4 }) ? $`

Observe que la subexpresión asociada al guión con cuatro dígitos está agrupada entre paréntesis, detrás de los cuales se ha escrito una interrogación. La interrogación indica que el grupo anterior es opcional. Otra forma de escribir la misma expresión es ésta:

`^ [0-9] { 5 } (- [0-9] { 4 }) { 0,1 } $`

Las llaves siempre indican repeticiones. En este caso, hay una repetición como máximo y ninguna repetición como mínimo. ¿Quiere un ejemplo más complicado? Esta es la sintaxis de los códigos postales ingleses:

`^ [A-Za-z] {1,2} [0-9] [A-Za-z0-9]? ?[0-9] [A-Za-z] {2} $`

Traduzcamos: una o dos letras, un dígito, y opcionalmente un dígito o letra adicional. A continuación viene un espacio opcional. Para terminar, un dígito y dos letras. El código postal de la dirección de una revista inglesa que recibo regularmente es el BR1 1BY, y pertenece a Kent. Por último, tenemos aquellos casos como Colombia, donde no se utilizan códigos postales. Para impedir que alguien teclee un código postal en una dirección en Colombia, he usado esta expresión regular:

`^ . ^`

Parece el rostro de un gato satisfecho, pero en realidad es la especificación de algo imposible: pedimos el inicio de línea, cualquier carácter y, nuevamente, el inicio de línea. Existen muchas más expresiones de este tipo, pero ésta es la que me ha parecido más “simpática”. Esta otra expresión regular puede utilizarse para aceptar cualquier cadena... siempre que no sea la cadena vacía:

`. +`

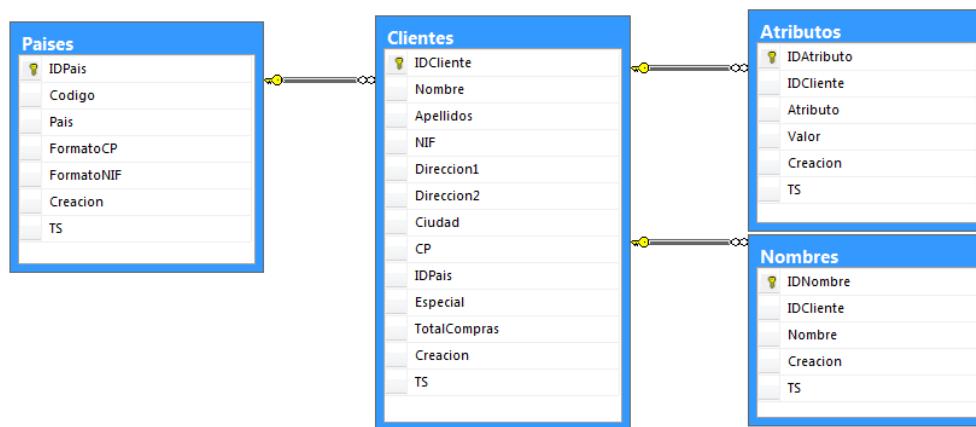
¿Interpretación? Digamos que el gato ha escondido la mitad de la cara tras una pared (incline la cabeza noventa grados hacia la izquierda), y lleva una de sus patas delanteras a la boca implorándonos silencio: feliz caza, Jinkssss...

**ANÉCDOTA**

Incluso para aplicaciones orientadas a un único país, es muy conveniente poder modificar el formato del código postal y del número de identificación fiscal. Viví en primera persona un caso aleccionador: una compañía telefónica que realizó su primera campaña de captación de clientes por teléfono. Un servidor estuvo entre los primeros en darse de alta en la compañía... pero pasaban las semanas y no podía usar los nuevos servicios. Casualmente, tuve que dar un curso de programación a la empresa encargada de la campaña. Al enterarme del contrato que tenían, les comenté mi situación... y para sorpresa de todos, descubrimos entonces la causa del retraso. La aplicación utilizada por las telefonistas utilizaba un formato para los números de identidad españoles que no aceptaba los códigos fiscales de empresas ni los números de identidad para extranjeros, como era mi caso. Las telefonistas, para no bajar su ritmo de trabajo, “inventaban” un valor para esa columna, y anotaban el identificador real en una columna de comentarios.

## Clientes, direcciones y atributos

Por una parte, el modelo de clientes de esta base de datos es una simplificación respecto al modelo utilizado en otros cursos de IntSight. En otros cursos he permitido más de una dirección por cliente. Esto es interesante cuando se trata de una tienda en Internet, pero era innecesariamente complejo para el caso que estábamos modelando.



Lo que sí hemos mantenido es una tabla de atributos, que sirven para implementar un *esquema abierto*. Para explicar en qué consiste esta técnica, pondré como ejemplo el caso de la base de datos de clientes de la propia IntSight. De algunos clientes tenemos la dirección de correo electrónico,

pero no un teléfono de contacto. En otros casos hay teléfono, pero correo electrónico. Unas veces tenemos la dirección física y otras no. Y no se trata sólo de atributos más o menos predecibles: en tal caso podríamos resolverlo añadiendo todas las columnas que se nos ocurran y permitiendo que todas ellas acepten valores nulos. Por ejemplo, en el momento en que empezamos a comercializar productos basados en descargas, como este curso, tuvimos que almacenar, para cada cliente que comprase uno de estos productos, un nombre de usuario y contraseña.

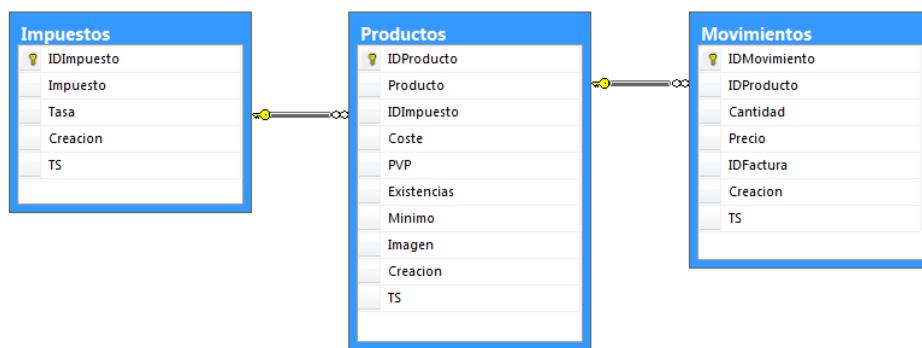
La cura está en una tabla de atributos, configurada como tabla de detalles de la tabla de clientes. En nuestro caso, el nombre de usuario del cliente se almacena mediante un registro adicional dentro de la tabla de atributos. Por supuesto, en la tabla de clientes se sitúan aquellas columnas que positivamente sabemos que serán necesarias.

Otra peculiaridad interesante del módulo de clientes es la tabla de *Nombres*. Supongamos que damos de alta a una clienta llamada María del Carmen del Pozo Seco. Tenemos una tabla preconfigurada con palabras que se pueden ignorar, como la contracción *del* en nuestro ejemplo. Cada uno de los nombres o apellidos restantes se almacena por separado en la tabla de nombres, junto al identificador asociado al cliente. Esta técnica nos permitirá encontrar eficientemente el registro de la nueva clienta independientemente de si buscamos las *Mariás* o las *Cármenes*. El mantenimiento de la tabla de nombres se realiza automáticamente, con la ayuda de *triggers*.

## El modelo de productos

Algo más sencillo es el modelo de productos. Aunque podríamos haber implementado un esquema abierto, como el del modelo de clientes, he preferido no añadir complejidad innecesaria. Para cada producto se almacena su nombre, su precio de venta actual, su coste y el identificador del impuesto de ventas asociado. Si usted tiene la inmensa suerte de vivir en un país sin este estúpido impuesto, le felicito: no se preocupe en entenderlo, porque no tiene lógica alguna.

Hemos implementado un sistema muy sencillo de control del inventario. Para cada producto, mantenemos una columna con el total de existencias y el mínimo permitido: no se pueden vender unidades después que las existencias igualen este valor mínimo. Cada vez que se crea una factura, automáticamente disminuiremos las unidades en existencia. Pero esta columna no puede ser directamente modificada para aumentar las existencias. En realidad, el valor de la columna *Existencias* de la tabla de productos estará controlado por la tabla *Movimientos*. Esta segunda tabla tiene una columna *Cantidad*. Cada vez que se crea un registro en *Movimientos* con *Cantidad* positiva, se incrementan las existencias del producto asociado; un registro de movimientos con *Cantidad* negativa disminuye las existencias. De esta forma, tenemos una justificación detallada de cada cambio en las existencias de cada producto.

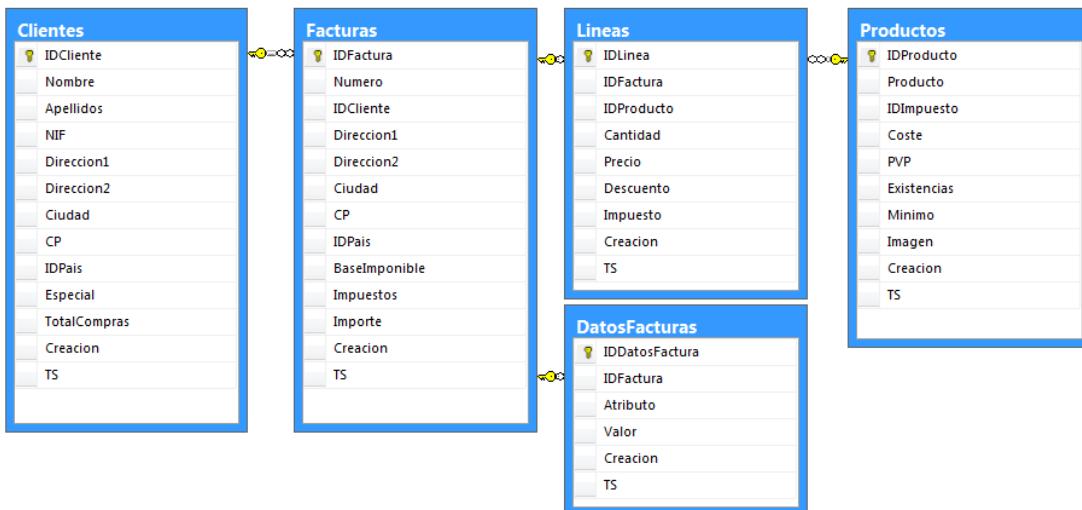


Aunque no se muestra en el diagrama anterior, hemos dispuesto una tabla de *Palabras*, que funciona de forma similar a la tabla de *Nombres* del modelo de clientes. Si la base de datos fuera a utilizarse en una tienda electrónica, deberíamos permitir la adición manual de palabras claves a *Palabras*. Tendríamos que añadir una columna adicional en esta tabla para indicar si una palabra dada fue añadida explícitamente o de forma automática. Además de evitar que los *triggers* de *Productos* borrasen por error las palabras añadidas a mano, nuestro algoritmo de búsqueda mejoraría, al otorgar más importancia a las palabras claves manuales.

## Facturas

El modelo de facturas es también muy simple. Tenemos un *IDFactura* que es una clave primaria artificial, como hemos explicado. Pero hay también un *Número* asociado a la factura, que es el número de factura que maneja el “mundo exterior”: un buen ejemplo de necesidad artificial introducida por viejas aplicaciones. Mientras que los identificadores de facturas pueden tener saltos en su secuencia, los números de factura no sólo deben ser únicos, sino que no deben dejar huecos. Esta característica la lograremos asignando valores desde una tabla de contadores.

La factura almacena, además del inevitable identificador de cliente, una copia de su dirección en el momento de la creación de la factura. Por supuesto, esta dirección puede modificarse durante la creación de la factura.



Las facturas también tienen asociada una tabla de atributos, emulando la técnica aplicada a los registros de clientes. En la tabla *DatosFacturas* podemos anotar las observaciones que nos vengan en ganas, de forma parecida a lo que haríamos con un campo de comentarios. Por ejemplo, atributo: *Hora entrega*, valor: *12:00*. La diferencia respecto a los campos de comentarios es grande, no obstante: podremos localizar fácilmente las facturas que tengan determinado valor asociado a determinado atributo. Eso sí, debemos utilizar estos atributos de forma coherente. La aplicación puede hacer mucho a favor de esta coherencia, obligándole a definir los atributos aceptables antes de poder utilizarlos, por ejemplo.

## Reglas de negocio y precios de venta

Por último, vamos a introducir una novedad en este curso: mantendremos una tabla con reglas especiales para determinar los descuentos que aplicaremos automáticamente a las facturas (con el permiso de la actual ministra de cultura). Intentaremos que estas reglas sean lo más generales que sea posible. Podremos acceder en la regla a toda la información asociada al cliente, la factura y los productos que formen parte de la factura, y en forma algo más limitada, a la historia de las compras realizadas por el cliente. Para ello, hemos añadido un par de columnas especiales a la tabla de clientes. Una es *TotalCompras*, y nos permitirá aplicar descuentos a nuestros mejores clientes. Y la otra es *Especial*, de tipo **bit** o lógico, que nos permitirá favorecer descaradamente a las clientas más apetitosas (¡perdóname, Santa Shirley, patrona de lo políticamente correcto!).

Cada regla consiste en una descripción literal, un par de columnas de tipo fecha, que indican en qué momento entra la regla en vigor, y hasta qué momento debe aplicarse, y un campo de tipo **text**, que contiene código en JScript y que determina la acción de la regla. Este código JScript es evaluado en el lado cliente por el motor de JScript que viene incluido con todas las versiones recientes de Windows. A su debido momento, explicaré los convenios necesarios para la programación de reglas.

# EJERCICIO 01

## Objetivos

Gestión de ventanas

## Técnicas introducidas

Layout managers, reflexión, atributos, metaclasses

**L**A MAYOR PARTE DEL ESFUERZO REQUERIDO para crear una aplicación es de naturaleza mecánica: añade una ventana al proyecto, configúrala para que pueda mostrarse en paralelo con otras ventanas, añade entonces un comando al menú de la ventana principal y un botón a la barra de herramientas, haz que ambos creen y muestren una instancia del nuevo tipo de ventana, deja caer una rejilla en la ventana, haz que apunte a la tabla de robos y homicidios, configura las columnas... Debe quedar claro que algo tendremos que hacer para simplificar todo este tedio.

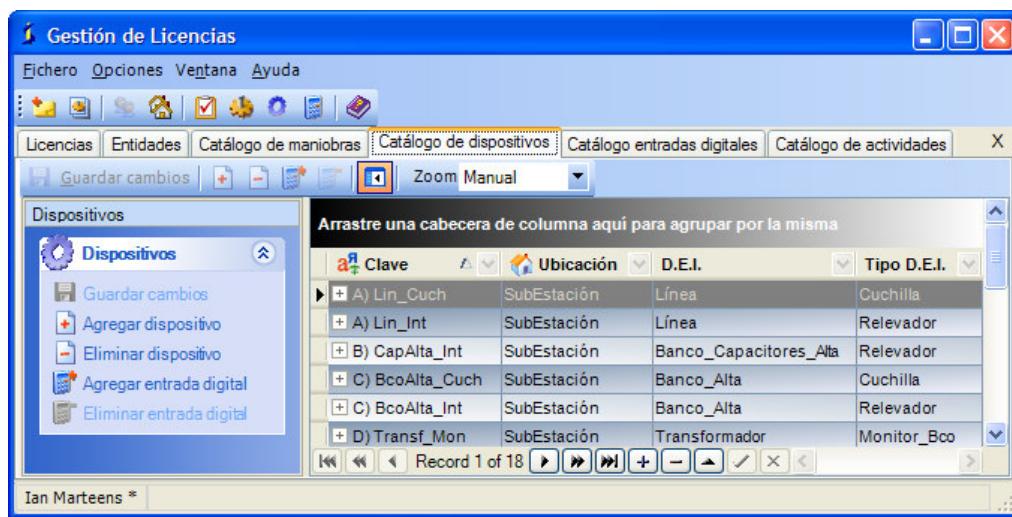
## Rubias, morenas y pelirrojas

Este chiste, además de bastante malo, es más viejo que el mundo. Entrevistan a un individuo y, entre otras cosas, le preguntan si le gustan las rubias, las morenas o las pelirrojas. A lo que el buen señor responde enfáticamente: ¡sí!

Algo parecido sucede al programador típico con los modelos de aplicación que permite Windows: le gustan todos. O al menos, es incapaz de decidirse por uno de ellos. ¿Hay tantos modelos? Así, de repente, se me ocurren tres:

- 1 *Multiple Document Interface* (MDI): o el modelo de Office antes de Office 2000.
- 2 *Single Document Interface* (SDI): está claro que se trata del modelo de Office, a partir de Office 2000. Pura lógica, ¿no?
- 3 El modelo de pestañas de Visual Studio, Internet Explorer 7 y tantas otras aplicaciones, que en vez de ventanas independientes, sitúa los “documentos”, en el sentido genérico de la palabra, en páginas separadas dentro un control de páginas.

El tercer modelo de mi lista se ha puesto de moda, lo cual encuentro inexplicable. La siguiente imagen corresponde a una aplicación que implementa ese modelo:



Mediante botones o comandos de menú, podemos abrir *vistas* en el área central de la ventana. Para acceder a cada una de ellas, tenemos las pestañas que aparecen inmediatamente debajo de la barra de herramientas principal. Cada vista puede cerrarse por separado, pulsando el botón que aparece en el extremo derecho de la hilera de pestañas: ése que dibuja una X.

¿Por qué digo que me parece inexplicable tanta popularidad? Pues porque en este modelo se hace muy difícil comparar el contenido de dos ventanas relacionadas. Un control de páginas sólo permite

ver una de sus páginas por vez. Por este motivo, las aplicaciones que implementan este modelo tienen que añadir trucos como dividir el área de una página, o permitir que las páginas “floten” y puedan ser acopladas según interese.

### DRAMA

Este otro chiste trata sobre una mujer muy celosa. Día a día, esperaba a que su marido llegase de la oficina, y examinaba su ropa. Si encontraba un pelo rubio, le montaba una escena por tener un presunto *affaire* con una rubia. Al día siguiente, si el pelo era de color negro, la bronca ascendía en la escala Richter: había encontrado una prueba de que ni siquiera podía serle fiel a su amante rubia. Un buen día, al atribulado esposo se le enciende la chispa, y cepilla cuidadosamente la ropa antes de traspasar el umbral. Su media naranja lo examina detenidamente, pero no hay pelos a los que agarrarse. Para sorpresa del marido, la arpía se pone a bufar como un buey constipado mientras le recrimina a gritos: ibastardo, hasta con calvas me engañas!

## Gestores de ventanas

¿Por qué limitarnos a las rubias, existiendo las morenas y las pelirrojas? ¿Por qué no dejamos que sea el propio usuario quien decida el modelo de ventanas con el que prefiere trabajar? Esto es lo que hace el propio IDE de Visual Studio, aunque con una limitación con la que también tropezaremos: los cambios en el modelo de ventanas no surten efecto hasta que salimos de la aplicación y volvemos a ejecutarla. Esta es una consecuencia inevitable de las peculiaridades del modelo MDI de Windows.

¿Cómo logra Visual Studio tal “hazaña”? Muy fácil: como siempre sucede en Informática, añadiendo una capa de “indirección”. Es decir, en vez de utilizar instrucciones del tipo *crea una ventana de tal y más cuál tipo*, se ejecutan métodos de un mediador, para que éste sea el que se encargue en realidad de la tarea. Esta técnica tiene una antigüedad respetable, y al componente que nos ayuda a implementarla se le da el nombre de gestor o administrador de ventanas; en inglés suele conocerse con el nombre de *layout manager*. Un gestor de ventanas está formado por varios elementos:

- 1 Un tipo de interfaz que determina qué operaciones necesitamos para la gestión de ventanas.
- 2 Una o más clases que implementen el tipo de interfaz mencionado. Cada una de estas clases puede implementar un modelo diferente, o variaciones sobre un modelo.
- 3 La aplicación trabaja siempre con un puntero perteneciente al tipo de interfaz. Al iniciarse la aplicación, ésta debe crear una instancia de la clase apropiada, según las preferencias del usuario, y asociarla al puntero de interfaz.

En esta parte del curso, crearemos un gestor de ventanas. Para más exactos, implementaremos dos sistemas estrechamente relacionados:

- 1 El gestor de ventanas, en sí.
- 2 Un gestor de menús, que nos permitirá construir, de forma más o menos declarativa y automática, parte del menú y de las barras de herramienta de la aplicación.

Inicialmente, nuestro gestor sólo implementará el modelo MDI. El soporte que Windows y .NET ofrecen para este modelo nos va a ahorrar mucho trabajo: la lista de ventanas abiertas, por ejemplo, o el engranaje de operaciones y eventos para la apertura y cierre de este modelo de ventanas. No obstante, incluiremos más clases junto con el código fuente del curso.

## Gestores de menús

Volvemos al tema introducido en el primer párrafo del ejercicio: es un incordio que, cada vez que añadimos un nuevo tipo de ventana a un proyecto, tengamos que crear un comando de menú, repetir gran parte del trabajo para situar un botón en la barra de herramientas, y escribir un par de líneas de código tonto para mostrar la ventana de mis dolores. Lo ideal sería que, al crear la ventana, ella misma defina el comando de menú que se encargará de mostrarla, y el correspondiente botón de la barra de herramientas. ¿Es posible lograr tal cosa?

Está claro que sí: siempre es posible crear dinámicamente el menú, o parte de él. El problema es *cómo*. La reacción instintiva es ubicar la información necesaria en una estructura de datos central,

pero se trataría de un craso error: cada nueva ventana, un viaje al repositorio central de información.

**NOTA**

Algo muy diferente sería tener un repositorio central que determinase, además del menú, todos o casi todos los detalles de cada ventana de navegación, edición o selección. Lo que debemos evitar es la mezcla de los dos sistemas.

¿Qué opciones tenemos para agrupar la información del menú junto a la definición de la ventana? No podemos registrar dicha información dentro de un método de la clase de ventana, porque necesitamos el menú antes de crear cualquier ventana... a no ser creemos prototipos ocultos, una idea que no me gusta. Podríamos usar trucos: en C++, crear instancias estáticas de clases artificiales y situar el código necesario en los constructores de esas clases. Horrible. En Delphi, podríamos usar el código de inicialización de las unidades. Chapucero. En C#, podríamos crear un constructor estático de clase para cada ventana. Un poco más elegante, pero conozco una idea mejor: para estas necesidades existen los *atributos*.

Un atributo en la plataforma .NET es un objeto que se asocia a una declaración; da lo mismo si se trata de una declaración de clase, de método, de propiedad... incluso podemos *decorar* parámetros de métodos con atributos. Como puede ver, era esto lo que queríamos: asociar información a una clase, de manera que podamos recuperarla sin necesidad de crear una instancia de la misma.

Un atributo es un objeto, y por lo tanto debe pertenecer a alguna clase. Las clases que se utilizan para declarar atributos deben descender de *System.Attribute*. Vamos a crear una clase especial para asociar atributos a las ventanas no modales que queremos incluir automáticamente dentro del menú de la aplicación. La llamaremos *NonModalAttribute*, siguiendo el convenio que exige que las clases de atributos tengan nombres con el sufijo *Attribute*:

```
[AttributeUsage(AttributeTargets.Class, AllowMultiple = false)]
public class NonModalAttribute : System.Attribute
{
    private string description;
    private bool uniqueInstance = false;
    private string location;

    public NonModalAttribute(string description, string location)
    {
        this.description = description;
        this.location = location;
    }

    public string Description
    {
        get { return description; }
    }

    public bool UniqueInstance
    {
        get { return uniqueInstance; }
        set { uniqueInstance = value; }
    }

    public string Location
    {
        get { return location; }
        set { location = value; }
    }
}
```

¿Ve el código resaltado, en la primera línea? Es que nuestra clase de atributos ha sido, a su vez, decorada con un atributo predefinido. *AttributeUsage* indica que el atributo que estamos definiendo, *NonModalAttribute*, sólo puede aplicarse a declaraciones de clases, y que no puede aplicarse dos veces a la misma clase: sólo se permite una instancia de *NonModalAttribute* para cada clase, como máximo.

La clase en sí es muy simple: sólo contiene unos pocos campos, encapsulados dentro de propiedades. Algunas propiedades son de sólo lectura, como *Description*, por lo que sólo pueden recibir un valor dentro del constructor. Las que permiten escrituras van directas al grano... quiero decir, al campo subyacente.

**NOTA**

A primera vista, puede que parezca buena idea encapsular dentro de esta clase la creación de entradas dentro de un menú, por poner un ejemplo. Sin embargo, esto nos ataría a determinadas clases de menú: tendríamos problemas si tuviésemos que trabajar con componentes de menú de otros fabricantes.

Antes de explicar qué significa cada propiedad de la clase de atributos, veamos cómo vamos a utilizarla. Suponga que *Ventana1* y *Ventana2* corresponden a sendas clases de ventanas:

```
[NonModal("Mostrar ventana #&1", "&Ver")]
public partial class Ventana1 : Form
{
    ;
}

[NonModal("Mostrar ventana #&2", "&Ver|&Otros")]
public partial class Ventana2 : Form
{
    ;
}
```

En primer lugar, podemos omitir el sufijo *Attribute* al mencionar la clase: de *NonModalAttribute* hemos pasado al menos soleme *NonModal*. Los parámetros dentro de paréntesis corresponden a los parámetros del constructor de la clase. Es como si el compilador crease una instancia de la clase de atributos durante la compilación, y la enganchase a la tabla de metadatos que va generando como resultado. Podemos modificar también el valor de las propiedades que lo permitan, utilizando parámetros con nombre al declarar el atributo:

```
[NonModal("Mostrar ventana #&1", "&Ver", UniqueInstance=true)]
```

Las propiedades de la clase *NonModalAttribute* no tienen un significado predefinido: hará falta otra clase para recopilar las instancias de *NonModalAttribute* y para interpretar qué demonios significa eso de *UniqueInstance=true*. Le adelanto, por lo tanto, mis intenciones:

- *Description*: cuando el gestor del menú cree un comando, esta cadena será el texto del comando.
- *Location*: indicará la ubicación del comando de menú. En el primer ejemplo, el comando se crea inmediatamente dentro del menú *Ver*, de la barra de menú principal. En el segundo, el comando se crea dentro del submenú *Otros* del menú *Ver*.
- *UniqueInstance*: cuando su valor es **false**, podemos crear varias ventanas del mismo tipo en paralelo. Si su valor es **true**, sólo se permitirá una instancia para ese tipo de ventana.

En ejercicios posteriores, añadiremos más propiedades al atributo *NonModal*. Observe que, para simplificar, no hay información alguna sobre cómo configurar botones en la barra de herramientas.

### Reflexionemos...

Próxima parada: creación del gestor de menús. En vez de apilar todo el código dentro de una sola clase, voy a definir primero una clase *abstracta*: una clase marcada de tal manera que no se puedan crear instancias de la misma. En nuestro ejemplo, la clase contendrá métodos *abstractos*. Un método abstracto es simplemente un espacio en blanco, que hay que llenar en algún momento en alguna clase derivada. La separación en dos clases no es capricho mío: ahora vamos a gestionar menús basados en *MenuStrip*, la clase que viene de serie con .NET. Pero puede interesarnos utilizar alguna colección de componentes alternativa, como los de Developer Express. Si creásemos una clase monolítica, sería más complicado adaptarla a nuevas situaciones.

Nuestra clase abstracta, a la que llamaremos *BaseMenuManager*, tendrá como misión descubrir las clases de ventanas que existen dentro de nuestra aplicación, y separar de ellas las que hayan sido

decoradas con el atributo *NonModal*. Finalmente, con la información que encuentre en cada uno de estos atributos, debe “crear” comandos de menú. El verbo *crear* va entre comillas porque esta clase, en particular, no sabe todavía con qué tipo de menú tendrá que lidiar.

```
public abstract class BaseMenuManager
{
    protected Dictionary<object, Type> menuTable =
        new Dictionary<object, Type>();

    public virtual void CreateMenu()
    {
        foreach (Type type in
            Assembly.GetExecutingAssembly().GetTypes())
        if (type.IsSubclassOf(typeof(Form)) &&
            type.IsDefined(typeof(NonModalAttribute), false))
        {
            NonModalAttribute[] attrs = type.GetCustomAttributes(
                typeof(NonModalAttribute), false) as NonModalAttribute[];
            if (attrs != null && attrs.Length > 0)
            {
                object item = CreateMenuItem(attrs[0]);
                menuTable[item] = type;
            }
        }
    }

    protected abstract object CreateMenuItem(NonModalAttribute attr);

    public Type this[object item]
    {
        get { return menuTable[item]; }
    }
}
```

Amigo mío, acaba de presenciar la magia de la *reflexión* en casi todo su esplendor (habrá más). Nuestra aplicación puede averiguar *todas* las clases que existen dentro de nuestro ensamblado, sea éste una DLL o un fichero ejecutable:

- La clase *Assembly* representa ensamblados.
- Su método estático *GetExecutingAssembly* obtiene una instancia que representa al ensamblado al que pertenece el método que se está ejecutando. Respire hondo, que esto puede ser profundo.
- La clase *Assembly* tiene también un método *GetTypes*, que nos devuelve un vector con todos los tipos de datos presentes en el ensamblado. *GetTypes* sabe lo que sabe gracias a la información de metadatos que el compilador genera durante la compilación. Preste atención de todos modos: se trata de los tipos dentro del ensamblado en ejecución. La aplicación puede estar haciendo referencia a otros ensamblados, de forma similar a como una aplicación *nativa* de Win32 cargaba DLLs en aquellos buenos tiempos. Nuestra llamada a *GetTypes* no devolverá los tipos residentes en *esos otros* ensamblados. No los necesitamos ahora. ¿Queda claro?
- La instrucción **foreach** nos ayuda a recorrer el vector devuelto por *GetTypes*. Nos ahorraremos tener que declarar una variable para el vector, una variable de control entera para el recorrido, y tener que acertar con la inicialización y condición de parada del inevitable bucle. Viva **foreach** y la madre que la parrío.
- No, no nos sirven todos los tipos que devuelve *GetTypes*. Sólo queremos clases de ventanas, que son las clases que descienden de *Form*, y para averiguarlo usamos el método *IsSubclassOf* de la clase *Type*. De las clases que sobrevivan a la criba, ignoramos aquellas que no hayan sido decoradas con el atributo *NonModal*. Esa es la tarea de otro método, *IsDefined*. El segundo parámetro de *IsDefined* indica si aceptamos clases con la decoración introducida en un ancestro. No, no lo aceptamos.

Enfoquemos el código dentro del bucle para orientarnos mejor:

```
NonModalAttribute[] attrs = type.GetCustomAttributes(
    typeof(NonModalAttribute), false) as NonModalAttribute[];
```

```
if (attrs != null && attrs.Length > 0)
{
    object item = CreateMenuItem(attrs[0]);
    menuTable[item] = type;
}
:
```

¿Recuerda que teníamos un *type* en la mano que correspondía a una clase de ventana no modal? Ahora pedimos que nos den la lista de objetos *NonModalAttribute* asociada a la definición de la clase. No se sulfure: recuerdo perfectamente que no podemos aplicar más de una instancia de *NonModal* a una misma clase. Usted lo sabe, yo lo sé, pero .NET no lo sabe de antemano, y para evitar sorpresas nos devuelve todo un vector. Comprobamos que haya al menos una instancia, para ser prudentes, y utilizamos el objeto de atributo para “crear”, nuevamente entre comillas, un comando de menú. Observe que *CreateMenuItem* ha sido declarado como método con la directiva **abstract**.

¿A qué tipo pertenecerá el comando de menú? Le confieso que no tengo idea. Por lo tanto, lo guardo en una variable de tipo **object**. Y a continuación ocurre algo interesante: utilizo un diccionario para asociar el puntero al objeto recién creado con el tipo de datos de la ventana que le corresponde. El diccionario se inicializa al declarar la variable, y el indizador de la clase, que es esa misteriosa propiedad que aparentemente se llama **this**, tiene como objetivo que quienes usen instancias de clases derivadas de *BaseMenuManager* puedan hurgar dentro del diccionario y recuperar el tipo de ventana asociado a un comando de menú determinado.

## Maniobras orquestales dentro de un menú

Si queremos que *BaseMenuManager* sea útil, tendremos que derivar otra clase a partir de ella, y darle una implementación al abstracto y esotérico *CreateMenuItem*:

```
public class MenuManager: BaseMenuManager
{
    protected MenuStrip menu;
    protected EventHandler clickHandler;

    public MenuManager(MenuStrip menu, EventHandler clickHandler)
    {
        this.menu = menu;
        this.clickHandler = clickHandler;
    }

    :
}
```

La clase *MenuManager* creará los comandos de menú dentro de un prosaico *MenuStrip*: el componente que sirve para crear barras de menú a partir de .NET 2. Precisamente, el constructor de *MenuManager* recibe como parámetro una referencia a un objeto de esa clase, y un puntero a método... perdón, un *delegado* (*delegate*) en la terminología puntonética. Con esta información, *MenuManager* puede crear un comando de menú, que es la parte más fácil, y además, insertarlo donde corresponda, que es un poco más complicado:

```
protected override object CreateMenuItem(NonModalAttribute attr)
{
    ToolStripMenuItem newItem = new ToolStripMenuItem(attr.Description);
    newItem.Click += clickHandler;
    ToolStripItemCollection items = menu.Items;
    foreach (string s in attr.Location.Split('|'))
        items = MenuByName(items, s).DropDownItems;
    items.Add(newItem);
    return newItem;
}
```

Debemos recordar que el atributo *Location* indica una ruta dentro del menú. Por ejemplo:

Ver | Otros | Ventanas inútiles

Para navegar por esta ruta, utilizamos una operación con cadenas de caracteres muy sencilla, pero que utilizaremos mucho en esta parte del curso. Se trata del método *Split*, de la clase *System.String*:

```
public string[] Split(params char[] separadores);
```

*Split* se aplica a una cadena, y recibe como parámetro una lista de caracteres separadores; en nuestro caso, usaremos la barra vertical como separador. El método fractura la cadena en trozos y nos los devuelve dentro de un vector. Con un estupendo bucle **foreach** vamos recorriendo ese vector y sumergiéndonos dentro de la estructura del menú. Para localizar el submenú que toque visitar en cada paso, y para crearlo si no existe, usamos un método auxiliar:

```
protected static ToolStripMenuItem MenuByName(
    ToolStripItemCollection collection, string itemText)
{
    string findText = itemText.Replace("&", "").ToLower();
    foreach (ToolStripItem item in collection)
        if (item.Text.Replace("&", "").ToLower() == findText)
            return item;
    return collection.Add(itemText);
}
```

En este caso, el método *Replace* quita los *ampersands* (como quiera que se traduzcan) que pueda contener el nombre del submenú, y *ToLower* convierte la cadena de búsqueda a minúsculas.

## El gestor en marcha

¿Qué tal si probamos el funcionamiento de nuestro invento? En un proyecto vacío, que sólo tenga la ventana principal con un *MenuStrip* en su interior, añada dos ventanas y “decórelas” como hemos mostrado al principio del ejercicio. No hace falta que configure las ventanas, porque de momento, no vamos a mostrarlas. Regrese luego a la ventana principal y modifique el constructor de la misma:

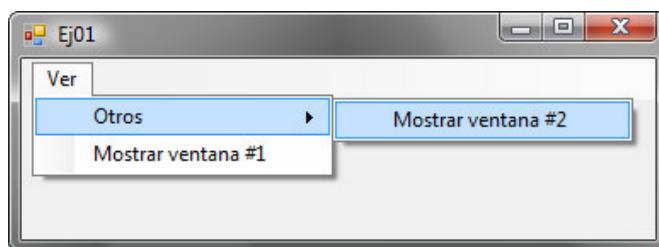
```
private BaseMenuManager menuManager;

public MainForm()
{
    InitializeComponent();
    menuManager = new MenuManager(
        menuStrip, new EventHandler(MenuClicked));
    menuManager.CreateMenu();
}
```

Hemos declarado una variable *menuManager* dentro de la ventana principal, y la hemos inicializado con una instancia de la clase *MenuManager*, a pesar de que la declaración dice *BaseMenuManager*. Eso es lo que llaman *polimorfismo*, y si tuviésemos más adelante que usar otra clase de gestor, nos simplificaría la migración.

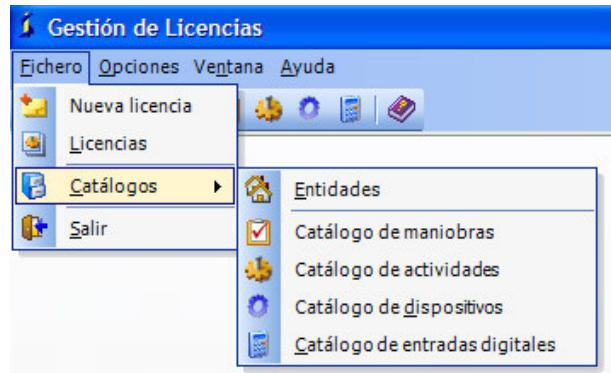
Al crear la instancia de *MenuManager*, tenemos que pasarle la referencia al menú y un puntero o delegado que haga referencia a un tal método *MenuClicked*. Tenemos que definir ese método, y por el momento podemos conformarnos con lo siguiente:

```
private void MenuClicked(object sender, EventArgs e)
{
    MessageBox.Show(menuManager[sender].FullName);
}
```



Es decir, cuando ejecutemos cada uno de los dos comandos de menú, se mostrará el nombre del tipo de ventana que tendríamos que crear y mostrar.

Por supuesto, hay una gran diferencia si comparamos el resultado con la siguiente imagen:



Para completar el gestor de menú, tendríamos que añadir imágenes, soporte para grupos de comandos, que se separarían mediante las típicas líneas horizontales, y permitir indicaciones sobre el orden relativo de los comandos.

# EJERCICIO 02

## Objetivos

Ventanas no modales

## Técnicas introducidas

Aplicaciones MDI, el método *InvokeMember*

**S**I EN EL EJERCICIO ANTERIOR PRESENTAMOS UN gestor de menús sencillo, pero adecuado, nos toca ahora diseñar el primer borrador de nuestro gestor de ventanas. El gestor no sólo consistirá en una clase encargada de crear ventanas y mostrarlas, sino que deberá definir también las *reglas del juego*: qué condiciones deben cumplir nuestras “ventanas”, qué capacidades deben ofrecer, para poder funcionar con esta pieza importante de la aplicación.

## El gestor de ventanas

Ejecute el comando de menú *Proyecto | Agregar clase*, y elija *LayoutManager* como nombre de la clase, y por lo tanto del fichero, que creará Visual Studio. A continuación, elimine el contenido dentro del fichero *LayoutManager.cs*, porque en realidad no crearemos una clase sino varios tipos de interfaz. Comenzaremos añadiendo estas dos declaraciones:

```
public interface IWindow
{
    // En próximos ejercicios, será ampliada.
}

public interface ILayoutManager
{
    IWindow Create(Type type);
    IWindow Activate(Type type);
    event EventHandler CreateWindow;
}
```

Una cosa son las ventanas y otra completamente distinta es el gestor de ventanas. Por este motivo, hemos comenzado por definir un tipo de interfaz *IWindow*, que nos dirá las capacidades que el gestor espera de cada “ventana”; de momento, *IWindow* está vacía, pero iremos añadiéndole métodos según nos surja la necesidad. Todas nuestras ventanas no modales deberán implementar dicha interfaz para poder ser manejadas por el componente de gestión. Este otro, por su parte, debe implementar dos métodos y disparar un evento. Los métodos son:

- *Activate* recibe un tipo de datos que debe corresponder a una clase que implemente la interfaz *IWindow*, y que esté decorada con el atributo *NonModal*. *Activate* nunca crea una ventana, sino que busca una ventana del tipo dado entre las ventanas abiertas de la aplicación, para hacerla visible y llevarla al primer plano.
- *Create*, por el contrario, comienza ejecutando *Activate*, pero si no se encuentra una ventana del tipo indicado, la crea.

El evento *CreateWindow* definido por *ILayoutManager* deberá dispararse cada vez que *Create* se vea obligado a crear una nueva instancia.

## Aplicaciones MDI

Como prometí en el ejercicio anterior, vamos a implementar un gestor de ventanas para aplicaciones MDI. Agregue otro fichero de código al proyecto, y llámelo *MdiLayoutManager...* o como le venga en ganas, siempre que recuerde el nombre. Primero, escribiremos el boceto de la nueva clase *MdiLayoutManager*, indicando qué interfaces implementa y definiendo un constructor:

```
public class MdiLayoutManager: ILayoutManager
{
    private Form mdiParent;
```

```

public MdiLayoutManager (Form mdiParent)
{
    this.mdiParent = mdiParent;
}

public event System.EventHandler CreateWindow;

:
}

```

¿Habíamos dicho algo sobre los prototipos de constructores de los gestores de ventana? Por supuesto que no. La interfaz *ILayoutManager* dicta los métodos que debe ofrecer la clase que pretenda implementarla, pero no dice nada sobre cómo llegan a existir las instancias de la clase en cuestión. Esto es una suerte para nosotros, porque tenemos libertad para añadir al constructor los parámetros que necesitemos para la inicialización del gestor. Estos parámetros dependen mucho del modelo de aplicación que deseemos. Para una aplicación MDI, como la nuestra, hay que pasar la ventana principal al gestor de ventanas. En una aplicación basada en pestañas, en cambio, el gestor debe saber con cuál control de pestañas tendrá que trabajar.

Veamos como aperitivo, la implementación del método *Activate*:

```

public IWindow Activate(Type type)
{
    foreach (Form f in mdiParent.MdiChildren)
        if (f.GetType() == type)
        {
            if (f.WindowState == FormWindowState.Minimized)
                f.WindowState = FormWindowState.Normal;
            f.BringToFront();
            return f as IWindow;
        }
    return null;
}

```

Nuestra implementación recorre la colección *MdiChildren* de la ventana principal para localizar una clase perteneciente a un tipo dado. Era a esto a lo que me refería cuando elogia el modelo MDI por todo lo que ya nos ofrece masticado; en un modelo basado en *TabControl*, por ejemplo, tendríamos que mantener nosotros mismos una estructura parecida a *MdiChildren*.

Si *Activate* encuentra una ventana apropiada, comprobará si está minimizada, para pasarlal al estado normal. Si el nuestro fuese un mundo perfecto, deberíamos restaurar la ventana al estado en que se encontraba antes de ser minimizada. Pero la única forma que conozco de hacerlo es mediante una llamada directa al API de Windows. Y aunque no costaría mucho, temo lo que podría suceder si la aplicación se ejecutase en una versión de la plataforma que no soportase la llamada mencionada, como podría suceder en Mono, la versión de .NET para el sistema del pingüino.

Este es el código del método *Create*:

```

public IWindow Create(Type type)
{
    IWindow wnd = IsUniqueInstance(type) ? Activate(type) : null;
    if (wnd == null && type != null)
    {
        Form f = type.InvokeMember(null,
            BindingFlags.DeclaredOnly |
            BindingFlags.Public | BindingFlags.NonPublic |
            BindingFlags.Instance | BindingFlags.CreateInstance,
            null, null, null) as Form;
        f.MdiParent = mdiParent;
        f.Show();
        if (CreateWindow != null)
            CreateWindow(f, EventArgs.Empty);
        wnd = f as IWindow;
    }
    return wnd;
}

```

Los primeros pasos que da *Create* dependen del campo *UniqueInstance* del atributo *NonModal* asociado a la ventana. Si este atributo vale **true**, quiere decir que queremos una instancia de la clase solicitada como máximo, y para evitar duplicados, buscamos si existe una ventana perteneciente al tipo... con la ayuda de *Activate*, por supuesto. La implementación del método auxiliar *IsUniqueInstance* es la siguiente:

```
protected static bool IsUniqueInstance(Type type)
{
    NonModalAttribute[] attrs = type.GetCustomAttributes(
        typeof(NonModalAttribute), false) as NonModalAttribute[];
    return attrs[0].UniqueInstance;
}
```

Como puede ver, seguimos usando reflexión a diestra y siniestra. Si permitimos más de una instancia de la clase, se pasa al núcleo del método, y volvemos a tropezar con métodos “reflexivos”:

```
Form f = type.InvokeMember(null,
    BindingFlags.DeclaredOnly |
    BindingFlags.Public | BindingFlags.NonPublic |
    BindingFlags.Instance | BindingFlags.CreateInstance,
    null, null, null) as Form;
:
```

Esta instrucción merece una explicación cuidadosa: recibimos un tipo de clase arbitrario, pero asumimos que la clase tiene un constructor sin parámetros. La llamada a *InvokeMember* ejecuta indirectamente ese constructor y nos devuelve el objeto creado. El método tiene varias versiones, pero el prototipo de la que hemos empleado es el siguiente:

```
public object InvokeMember(
    string name, BindingFlags invokeAttr, Binder binder,
    object target, object[] args);
```

*InvokeMember* no sólo sirve para ejecutar constructores, sino para acceder a métodos y propiedades en general. Su primer parámetro es el nombre del método que queremos ejecutar; como se trata de un constructor, en nuestro caso, pasamos un puntero nulo en vez de una cadena. El segundo parámetro indica las características del método, propiedad o constructor que nos interesa: observe en este caso que tenemos incluso la posibilidad de llamar a un constructor protegido. Además, la constante *CreateInstance* es necesaria para activar un constructor. Le confieso que no entiendo por qué es necesaria también la constante *Instance*, pero puede comprobar que si la omite, la maquinaria se detiene.

El tercer parámetro de *InvokeMember* es un *binder*: lo necesitaríamos si tuviésemos que seleccionar una variante concreta del constructor distinguiendo entre tipos de parámetros, o si necesitásemos forzar la conversión de algunos de los parámetros. Pero nuestro constructor no tiene parámetros, y eso también nos permite pasar un puntero nulo en el último argumento. Finalmente, el penúltimo parámetro, *target*, sería el objeto que se asociaría a **this** si estuviésemos ejecutando un método. Como estamos luchando a brazo partido con un constructor, le endilgamos otro nulo al sufrido *InvokeMember*.

Para terminar, *Create* maquilla un poco la ventana recién creada por *InvokeMember*, antes de devolverla como valor de retorno:

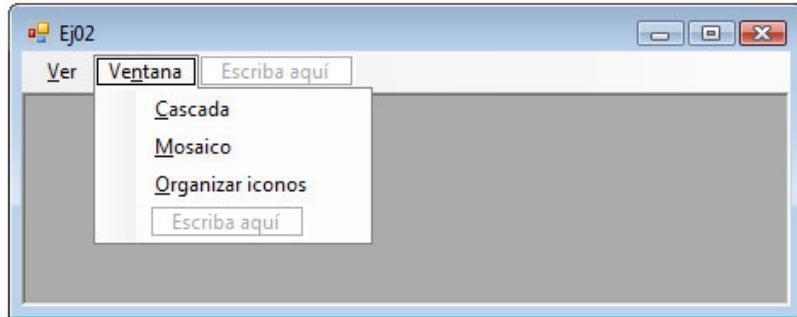
```
:
f.MdiParent = mdiParent;
f.Show();
if (CreateWindow != null)
    CreateWindow(f, EventArgs.Empty);
wnd = f as IWindow;
```

Primero, le asigna una referencia a la ventana principal en su propiedad *MdiParent*, para convertirla en una ventana hija MDI. Luego la muestra con una llamada a *Show*. Si alguien ha interceptado nuestro evento *CreateWindow*, lo disparamos. Y nos vamos, no sin antes convertir el objeto de tipo *Form* en un puntero a su interfaz *IWindow*.

## Integración con el gestor del menú

¿Comprobamos hasta dónde hemos llegado? Tenemos, por una parte, un controlador de menú, que puede generar automáticamente los comandos de menú para mostrar ventanas no modales a partir de atributos con los que decoraremos estas ventanas. Contamos, además, con una clase que nos permite crear instancias de las clases de ventanas no modales, comprobando la existencia de otras instancias si el atributo asociado lo exige. No es mucho, de momento, pero el gestor de ventanas seguirá aumentando sus capacidades a lo largo del curso.

Vamos a poner a prueba las clases que hemos desarrollado, y para ello prepararemos la ventana principal del ejercicio como un contenedor MDI, con todo lo que hace falta en este modelo.



Es muy sencillo configurar la ventana principal: nos basta con asignar `true` en `IsMdiContainer`. Añada un submenú *Ventana* sobre la barra principal, y asígnelo en la propiedad `MdiWindowListItem` del menú; de este modo, la aplicación creará y mantendrá la lista de ventanas hijas abiertas dentro de este submenú.

A continuación, configure los típicos comandos de manejos de ventanas para este modelo de aplicación, como en la imagen anterior. Estos son los manejadores de eventos que necesitamos:

```
private void miCascada_Click(object sender, EventArgs e)
{
    LayoutMdi(MdiLayout.Cascade);
}

private void miMosaico_Click(object sender, EventArgs e)
{
    LayoutMdi(MdiLayout.TileHorizontal);
}

private void miOrganizar_Click(object sender, EventArgs e)
{
    LayoutMdi(MdiLayout.ArrangeIcons);
}
```

He dejado fuera el comando *Mosaico vertical*, pero se implementa con la misma facilidad. He preferido el mosaico horizontal porque, cuando se trata de ventanas de navegación, es preferible limitar la altura antes que el ancho de las ventanas.

Ahora añadiremos un par de campos y un método auxiliar a la clase de la ventana principal. Para que el código quede bien organizado, le sugiero que encierre estas declaraciones dentro de una *región*, como muestro en el siguiente listado:

```
#region Componentes inicializados en código

BaseMenuManager menuManager;
ILayoutManager layoutManager;

private void InitData()
{
    menuManager = new MenuManager(
        mainMenu, new EventHandler(MenuClicked));
    menuManager.CreateMenu();
```

```

        layoutManager = new MdiLayoutManager(this);
    }

#endregion

```

*InitData* inicializa el controlador de menús, e inmediatamente ejecuta su método *CreateMenu*. De momento, observe el segundo parámetro del constructor de *MenuManager*; en breve explicaré su significado y propósito. Otra pregunta que se le puede ocurrir: ¿por qué nos quedamos con la referencia al *BaseMenuManager*, si ya hemos creado los comandos de menú? Recuerde que el controlador contiene un diccionario que asocia los comandos de menú que ha creado con los tipos de las ventanas asociadas a estos. La otra tarea del método *InitData* es la inicialización del gestor de ventanas.

Pero *InitData* es completamente inútil si no lo llamamos desde el constructor de la clase de la ventana principal. Este es el código necesario:

```

public MainForm()
{
    InitializeComponent();
    InitData();
}

```

Al llegar a este punto, tenemos dos gestores que se ocupan de sendos aspectos de la aplicación: comandos de menú y creación de ventanas... pero no existe comunicación entre ellos. Para comunicar ambos subsistemas, utilizaremos el manejador de eventos que hemos pasado en el segundo parámetro del constructor de *MenuManager*. Tenemos que definir un método como éste:

```

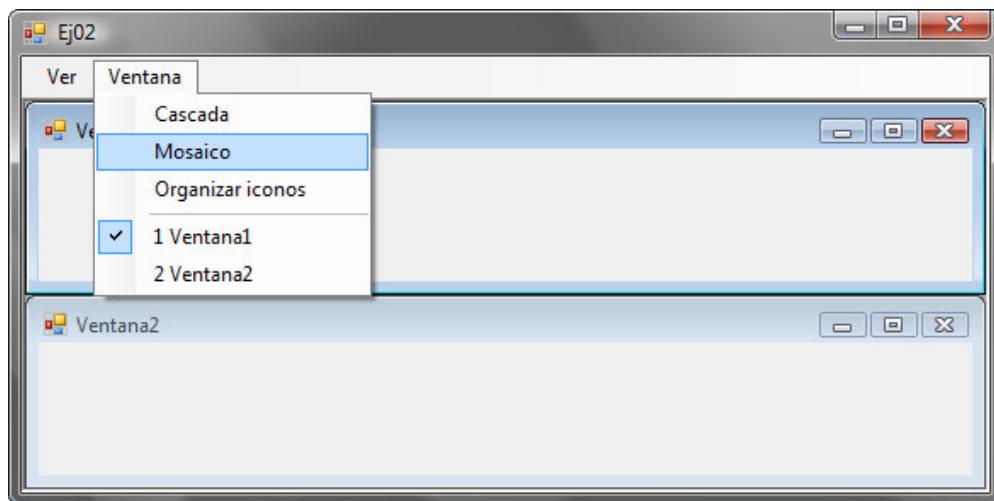
private void MenuClicked(object sender, EventArgs e)
{
    layoutManager.Create(menuManager[sender]);
}

```

Para pasar la “dirección” de este método al gestor del menú, necesitamos crear un tipo “delegado”. Ese es el propósito de la siguiente expresión, utilizada al construir el controlador del menú:

```
new EventHandler(MenuClicked)
```

Tenga presente que *MenuClicked* es el nombre de un método. Normalmente, cuando usamos un método en C# le añadimos la lista de parámetros entre paréntesis, incluso si el método no requiere parámetros. Cuando necesitamos la dirección del método, en cambio, tenemos que construir una instancia de un tipo delegado, como acabamos de ver.



Para poner la aplicación a prueba, añada dos formularios al proyecto, como en el ejercicio anterior. Esta vez, además del atributo *NonModal*, las clases correspondientes deben implementar la interfaz *IWindow*, como muestro a continuación:

```

[NonModal("Mostrar ventana #&1", "&Ver")]
public partial class Ventana1 : Form, IWindow { ... }

```

```
[NonModal("Mostrar ventana #&2", "&Ver")]
public class Ventana2 : Form, IWindow { ... }
```

Por fortuna, la interfaz *IWindow* está vacía de momento, y no necesitamos más retoques en las dos ventanas de prueba. Ejecute la aplicación y compruebe que dentro del menú *Ver* se crean los dos comandos necesarios para mostrar *Ventana1* y *Ventana2*. Verifique que se puedan crear estas ventanas, y de paso, ponga a prueba los comandos del submenú *Ventana*.

---

### EJERCICIO PROPUESTO

Cree un gestor de ventanas para un sistema de ventanas SDI. Puede hacer que las ventanas gestionadas tengan a la ventana principal como propietaria, utilizando la propiedad *Owner* de las ventanas gestionadas. Para obtener la lista de ventanas creadas, puede entonces usar la propiedad *OwnedForms* de la ventana principal.

---

# EJERCICIO 03

## Objetivos

Módulos de datos

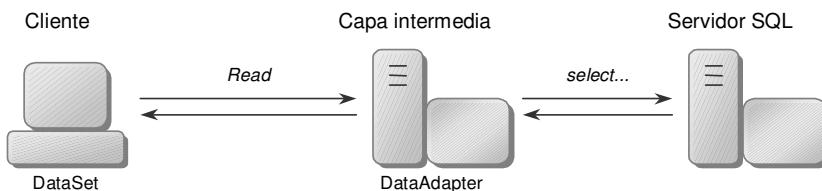
## Técnicas introducidas

Campos estáticos, campos de sólo lectura

**F**ALTA TODAVÍA MUCHA FUNCIONALIDAD POR añadir a *MenuManager* y *LayoutManager*. No perdamos de vista, no obstante, que este curso trata sobre programación de bases de datos. Dejaremos por un momento el diseño visual de la aplicación para esbozar el sistema de acceso a datos que utilizaremos durante el resto del curso.

## La organización del proyecto

No quiero tener que repetir aquí las razones que han llevado a la sustitución de la arquitectura básica cliente/servidor por sistemas divididos en múltiples capas, ni por qué *stateless* es casi siempre mejor que *statefull*.



Pocas veces se dan las circunstancias necesarias para que una división física como la representada en la imagen anterior sea obligatoria, o rinda beneficios tangibles. En una instalación para un entorno de 10 a 20 usuarios, es más que probable que una aplicación programada “a la antigua” hubiese conseguido un rendimiento aceptable. Pero no debemos olvidar un detalle:

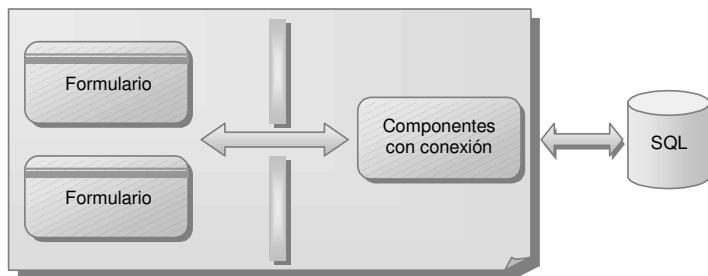
*Los entornos modernos de programación han sido y están siendo diseñados asumiendo que serán usados para crear aplicaciones divididas en capas.*

Dicho de otra manera: incluso en los casos en que nos bastaría con una arquitectura tradicional cliente/servidor, sería demasiado costoso ir contra la corriente. Además, si una aplicación tiene éxito, la empresa que la explota crece. Y las condiciones iniciales de baja carga terminan por desaparecer, más temprano que tarde.

Este curso dedica uno de sus módulos al estudio de los métodos de conexión remota, y las adaptaciones que debe sufrir una aplicación típica para sobrevivir en ese nuevo mundo. Pero de momento, nuestros ejercicios tendrán que mezclar dos capas dentro de una misma aplicación: la capa de presentación y el servidor de capa intermedia. Para no perder el tiempo, no obstante, seguiremos unas pocas pautas para que la segregación del código del servidor de capa intermedia sea lo más sencilla posible.

- 1 Bajo ningún pretexto permitiremos que los componentes de un formulario accedan a componentes “conectados”, como los que residen en el espacio de nombres *System.Data.SqlClient*.
- 2 Los componentes dependientes de una conexión deben residir en el menor número de módulos posibles. En nuestro ejemplo, intentaremos que todos ellos vengan encapsulados dentro de un solo fichero. Se trata de una actitud extrema, por supuesto, pero esto simplificará mucho nuestras existencias.
- 3 El objetivo de agrupar los componentes conectados en un único módulo es lograr que el resto de la aplicación se comunique con ellos a través de una sola interfaz de acceso.
- 4 Esa interfaz de acceso debe estar definida mediante un tipo de interfaz, no mediante una clase.
- 5 Los métodos y propiedades del tipo de interfaz deberán diseñarse teniendo en mente la posibilidad de migración del módulo conectado a un servidor remoto.

El siguiente diagrama representa el reparto de responsabilidades dentro de la aplicación:



Soy consciente de que no tengo futuro como pintor: he intentado ilustrar el principio que establece que, mientras más estrecho sea el canal de comunicación entre dos partes de una aplicación, más sencillo será aislar una de esas partes y sustituirla por otra de funcionalidad equivalente. Otra forma de explicar la estructura de nuestros ejercicios consiste en determinar qué tipos de componentes pueden ir en cada una de las partes del proyecto:

- Formularios: cualquier tipo de control, conjuntos de datos (*datasets*), vistas de datos y, en general, cualquier otro componente de datos sin conexión.
- El módulo especial de acceso a datos: componentes de conexión, comandos y adaptadores de datos. Excepcionalmente, podríamos usar algún que otro conjunto de datos, pero no será lo más habitual.

Reconozco que suena un poco abstracto, pero confío que a medida que vayamos progresando en los ejercicios, se despejen los detalles dudosos.

## Interfaces genéricas de acceso a datos

Para ir materializando las ideas, concentrémonos en el tipo de interfaz que definirá la comunicación entre la capa visual y la capa intermedia (recuerde que ambas residirán dentro de la misma aplicación en la presente serie de ejercicios). Como el tipo de interfaz debe poder ser utilizado para acceso remoto, debemos hacernos algunas preguntas durante su diseño:

- 1 ¿Pueden transmitirse remotamente todos los tipos de datos relacionados con la interfaz?
- 2 ¿Cuáles de estos tipos deben transmitirse por valor y cuáles por referencia?

Añada una clase al proyecto, y teclee *DataServer* cuando el asistente de Visual Studio le pida el nombre de la clase. Elimine la clase *DataServer* que se creará de forma automática, y sustitúyala con la siguiente declaración:

```
public interface IDataServer
{
    DataSet Read(string dsname);
    DataSet Write(string dsname, DataSet delta);
}
```

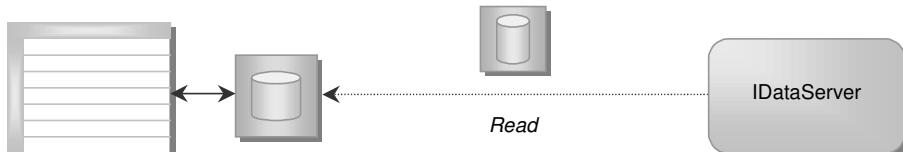
Le advierto que se trata de un primer intento, y que en ejercicios posteriores retocaremos *mucho* esta declaración. Por ahora tenemos dos métodos con significados evidentes:

- *Read* recibe una cadena que representa el nombre de un conjunto de datos y nos devuelve el correspondiente conjunto de datos. Por ejemplo:

```
IDataServer server = ... ;
DataSet ds = server.Read("DsProductos");
```

El “nombre” del conjunto de datos es un simple convenio que estableceremos para saber qué datos hay que recuperar. Como vamos a usar conjuntos de datos con tipos, la elección más natural consistirá en usar el nombre de la clase del conjunto de datos con tipo. La alternativa a usar un método como *Read* sería definir un método de lectura para cada tipo de conjunto de datos que necesitemos. Entonces tendríamos todo un puñado de métodos: *LeerClientes*, *LeerProductos*, *LeerPedidos*... y nunca lograríamos cerrar el diseño de la interfaz *IDataServer*.

Un detalle muy importante: *Read* devuelve un conjunto de datos. El truco consiste en que los conjuntos de datos, aunque son tipos de referencia dentro de la CLR, se transmiten por valor cuando hay llamadas remotas, ya sea a través de .NET Remoting o de XML Web Services. La primera consecuencia de este hecho es que cada llamada a *Read* devolverá un conjunto de datos nuevo. Esto puede convertirse en un problema, porque la forma más natural para la presentación de datos consiste en configurar el enlace a datos en tiempo de diseño, y eso implica usar un conjunto de datos fijo. El siguiente diagrama ilustra el problema y su solución:



Suponga que tenemos que mostrar los datos en una rejilla. Configuraremos este control con la ayuda de un conjunto de datos con tipos que residirá dentro del mismo formulario que la rejilla. Cuando llamemos al método *Read* de *IDataServer*, recibiremos un segundo conjunto de datos, esta vez de tipo genérico, que he representado flotando sobre la llamada. De este valor de retorno sólo nos interesa su *contenido*, por lo que *verteremos* sus registros sobre el conjunto de datos fijo y con tipos, con la ayuda del método *Merge* de la clase *DataSet*. Por esta razón, verá repetirse el siguiente patrón en los ejercicios:

```

IDataServer server = ... ;
DataSet ds = server.Read(dsProductos.DataSetName);
dsProductos.Merge(ds, false);
// ... y "ds" se liberará durante la próxima recogida de basura
  
```

¿Y por qué no pasamos el conjunto de datos fijo directamente como parámetro de *Read*? Piense en lo que ocurriría si la llamada a *Read* fuese una llamada remota. El mensaje de activación tendría que transmitir el conjunto de datos fijo (¡por valor!), y eso aumentaría mucho el tráfico en red. Pero lo peor es que el contenido del conjunto de datos se ignoraría al generar la respuesta. Es por detalles como éste por los que insisto en prestar mucha atención al diseño de la capa intermedia.

Pasemos al método de grabación:

- *Write* recibe nuevamente un nombre de conjunto de datos, y un conjunto de datos con los cambios provocados por la edición.

Evidentemente, el primer parámetro sigue el mismo convenio de nombres que elegimos para el método *Read*. También debe quedar claro que no podemos pasar el conjunto de datos original, con todo su contenido, a la capa intermedia. Por este motivo, al segundo parámetro de *Write* lo hemos llamado *delta*: contendrá solamente la “diferencia”, es decir, los registros modificados desde la última llamada a *Read* o *Write*.

¿Por qué *Write* devuelve también un conjunto de datos? Cada vez que guardemos el más pequeño e insignificante de los cambios, la columna *rowversion* de la fila cambiará su valor. Si no recuperamos el nuevo valor, nos veremos impedidos de grabar cualquier cambio posterior, al no coincidir nuestro valor de *rowversion* con el valor almacenado en la base de datos. Localmente, el módulo de acceso a datos actualizará todas las columnas de *delta* que lo necesiten, pero si no tomamos medidas, el cambio sobre *delta* se perderá. Por ello devolvemos el contenido de *delta* (¡por valor, otra vez!), para mezclarlo con el conjunto de datos fijo y de esta forma mantenerlo actualizado. Esquemáticamente:

```

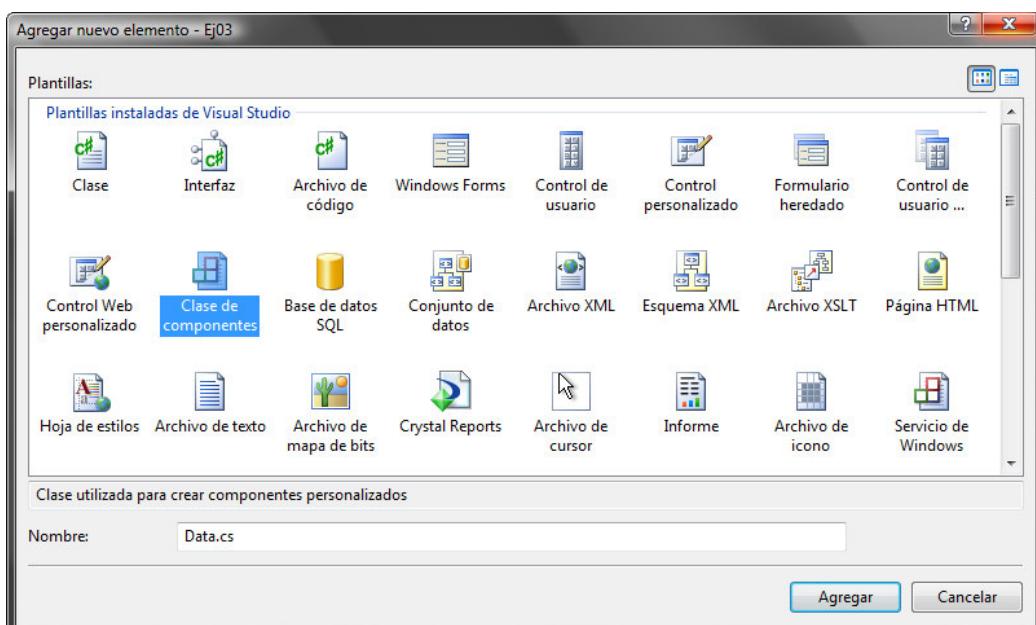
IDataServer server = ...;
DataSet delta = dsProductos.GetChanges();
delta = server.Write(dsProductos.DataSetName, delta);
dsProductos.Merge(delta, false);
  
```

Hay detalles que faltan en el listado anterior. Por ejemplo, ignoraremos por ahora lo que sucede cuando se produce una violación del bloqueo optimista. O cómo debemos actualizar el conjunto de datos fijo si realizamos inserciones de registros con columnas de tipo identidad. Todas estas técnicas las estudiaremos a su debido tiempo.

## Componentes e instancias globales

Es estupendo que dispongamos ya de una interfaz para el acceso a datos. Pero, ¿sobre cuáles hombres recaerá la responsabilidad de implementar dicha interfaz? Los kamikazes adictos a Java no dudarían ni un momento, y arrancarían a teclear una clase directamente derivada de *Object*. Cuando necesitasen un componente de conexión, teclearían las instrucciones necesarias para construirlo en tiempo de ejecución, y en el caso en que tuviesen uno de esos buenos días, incluso encapsularían la creación dentro de un método o incluso dentro de otra clase. En el fondo, son entrañables...

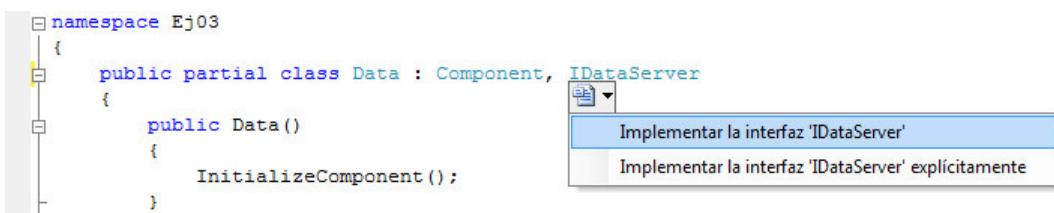
Los programadores inteligentes, como usted y como yo, utilizaríamos una superficie de edición visual para poder configurar las conexiones, comandos y adaptadores en tiempo de diseño. Y de las superficies de diseño soportadas por Visual Studio, la más apropiada es la que menos funcionalidad ofrece: la superficie de un componente. Ejecute el comando *Proyecto | Agregar componente*, y teclee *Data* como nombre de la nueva clase y de su fichero de código:



Abra el fichero de código y añada la interfaz *IDataServer* a la lista de interfaces implementadas por la clase, en su cabecera:

```
public class Data : Component, IDataServer
{
    public static readonly IDataServer Instance = new Data();
}
```

En cuanto haya terminado de teclear *IDATA SERVER*, el editor mostrará un *smart tag*, o etiqueta inteligente, para ayudarle a implementar los métodos de esta interfaz. Active la “etiqueta”, y ejecute el primero de los comandos, en el menú emergente:



Visual Studio creará automáticamente un método *Read* y un método *Write* dentro de la clase *Data*. Tenemos que modificarlos para que tengan implementaciones triviales, en vez de las implementaciones originales que disparan excepciones:

```
#region Miembros de IDataServer

public DataSet Read(string dsname)
{
    return null;
}

public DataSet Write(string dsname, DataSet delta)
{
    return null;
}

#endregion
```

Recuerde, no obstante, que habíamos quedado en utilizar una sola instancia global para todo el acceso a datos de la aplicación. Como C# no ofrece variables globales, como C++ o Pascal, tendremos que usar una propiedad o campo *estático* (**static**) declarado dentro de cualquier clase. Ahora bien, ¿dentro de qué clase mejor que la propia *Data* que estamos definiendo? Esta es la declaración que necesitamos:

```
public class Data: Component, IDataServer
{
    public static readonly IDataServer Instance = new Data();

    ...
}
```

Le confieso que me he vuelto adicto a la combinación de modificadores **static readonly**. Ya sabemos para qué sirve **static**; **readonly**, por su parte, quiere decir que el campo asociado sólo puede modificarse, para ser inicializado, durante la construcción de... bueno, si es un campo estático, durante la construcción de la clase, y si es un campo “normal”, de instancia, durante la construcción de la instancia, es decir, dentro de un constructor. La inicialización de nuestro campo *Instance* no está, hablando con rigor, dentro de un constructor. Pero esto se debe a que C# reúne todas estas inicializaciones asociadas a declaraciones y las copia dentro del constructor que corresponda.

Para entender mi afición a **static readonly** tiene que ver cuál sería la alternativa:

```
private static IDataServer instance = new Data();

public static IDataServer Instance
{
    get { return instance; }
}
```

Mucho más código para escribir y mantener. Y el tiempo es oro, ¿no?

Para terminar, si le preocupa saber cómo accederán los formularios a la instancia de *Data*, aquí tiene los dos ejemplos esbozados en la sección anterior de llamadas a la interfaz *IDataServer*:

```
DataSet ds = Data.Instance.Read("DsProductos");

// ...

DataSet delta = dsProductos.GetChanges();
dsProductos.Merge(Data.Instance.Write("DsProductos", delta), false);
```

He abreviado un poco combinando algunas de las instrucciones. Tampoco hay que obsesionarse con este tipo de “optimizaciones”: el JIT de .NET, es decir, el módulo de la plataforma que traduce el código en lenguaje intermedio en instrucciones ejecutables, suele funcionar muy bien. Mi propósito al mezclar instrucciones es el mismo de antes: abreviar para trabajar lo menos posible.

# EJERCICIO 04

Objetivos

Ventanas de navegación

Técnicas introducidas

Herencia visual

**N**UESTRO MODELO DE INTERACCIÓN ES muy sencillo. El usuario parte de una ventana de búsqueda, que le permite examinar el resultado de sus consultas sobre una rejilla. Luego, puede elegir registros para modificarlos, o añadir nuevos registros a los ya existentes. En este ejercicio nos ocuparemos de los detalles de diseño comunes a todas nuestras ventanas de búsqueda y navegación.

## ¿Qué es una “ventana”?

Ya tenemos una interfaz genérica para definir gestores de ventanas, y hemos implementado al menos un tipo de gestor, para ser usado en aplicaciones MDI. Dicho gestor, sin embargo, presenta un problema importante: nos ayuda a administrar clases de ventanas descendientes de *Form*, la clase base ofrecida por Windows Forms. ¿Qué hay de malo en ello? Piense cómo tendría que arreglárselas un gestor que trabajase sobre un *TabControl* para encajar toda una ventana, con marco, barra de título, botones de maximizar y minimizar y menú de sistema, dentro de un *TabPage*. Quizás podríamos resolverlo manipulando *FormBorderStyle* para eliminar el borde, y usar algún truco para que la ventana resida como un control más dentro de una página del *TabControl* principal. Sé cómo hacerlo con el API de Windows, pero nunca lo he intentado ciñéndome a las clases de Windows Forms, y no sé si será fácil, difícil o imposible...

... y para serle sincero, tampoco me preocupa demasiado, porque conozco una técnica mejor. ¿Nos obliga alguien a utilizar la clase *Form* para definir nuestras ventanas? Como alternativa, le propongo que definamos nuestras ventanas como *controles de usuarios*, o *user controls*; esto es, como clases derivadas de *UserControl*. Un control de usuario ofrece una superficie de diseño similar a la de un formulario, pues puede albergar otros controles. Una vez compilado, el control de usuario puede añadirse, incluso en tiempo de diseño, sobre cualquier otra superficie de diseño soportada por Visual Studio. Por supuesto, nuestro gestor de ventanas MDI alojará los controles de usuarios que definamos dentro de formularios MDI verdaderos, en tiempo de ejecución. Y un gestor basado en *TabControl* simplemente tendría que hacer la misma inclusión, pero dentro de controles *TabPage* creados dinámicamente. Asunto concluido.

Tenemos que establecer unas reglas mínimas para el juego, no obstante. En la página 20 presenté por primera vez la declaración del tipo de interfaz *IWindow*: las clases que implementasen esta interfaz serían reconocidas por el *layout manager* como “ventanas”. En aquel momento, declaramos el tipo sin definir sus métodos. Ahora completaremos la declaración:

```
public interface IWindow
{
    // Inicialización
    void InitData();
    // Métodos de control de cambios
    bool IsModified { get; }
    bool SaveChanges();
    void DiscardChanges();
    // Métodos de restauración del diseño
    void SaveLayout();
    void RestoreLayout();
}
```

Como puede ver, hay tres subsistemas representados en la interfaz. La P.O.O. más purista exigiría fragmentar *IWindow* en otras tantas interfaces... pero una cosa es la teoría, y otra es la práctica. No pongo en duda que un sistema cerrado, del tipo “Juan Palomo” (yo me lo guiso y yo me lo como), funcione mejor con una división modular de grano fino. Pero cuando se trata de componentes, o de ejemplos para un curso de programación, la explosión de tipos sólo trae problemas. El autor de este

curso trabajó bastante tiempo con WebSnap, un *application framework* para desarrollo en Internet de Borland, que incluía el código fuente completo. ¿El código? Parecía generado con alguna de esas astutas herramientas mecánicas, que pretenden transformar cuatro garabatos en algo ejecutable. Y con trillones de tipos de interfaz, cada uno con dos o tres métodos como máximo. El sistema en sí era bastante primitivo, porque sólo ofrecía funciones básicas, y se suponía que el programador debía aportar sus propios componentes para suplir ciertas carencias. En la práctica, poca gente fue capaz de hacerlo. La complejidad del sistema, provocada entre otras cosas por una división modular excesivamente fina, resultó ser inmanejable.

## Fragmentos de una ventana deconstruida

Toca explicar el papel de cada uno de los métodos de *IWindow*. El primer método forma un subsistema relativamente independiente:

- 1 *InitData*: se ocupa de la inicialización de la ventana. Usted se preguntará por qué hace falta simular un segundo constructor: ¿por qué no añadimos el código necesario dentro de los constructores de las clases derivadas? Tenemos una razón poderosa: las instancias de *IWindow* se crean en estado “flotante”, sin estar aún contenidas dentro de un formulario. Por lo tanto, la inicialización dentro del constructor no tiene acceso al contenedor del control. *InitData*, en cambio, será llamado por el *layout manager* una vez que el control de usuario se haya situado dentro de su contenedor. Es posible que esto mismo se pudiese lograr redefiniendo métodos de *UserControl*, o interceptando algún evento, pero la solución basada en *IWindow* e *InitData* es más elegante, por su carácter explícito, y nos aísla de posibles problemas provocados por futuros cambios en Windows Forms.

El siguiente bloque de métodos se ocupa de la faceta de la ventana como depósito de contenido editable. Si hubiésemos fragmentado *IWindow* en interfaces más simples, estos métodos irían a parar a un tipo con un nombre en la línea de *IEditable*.

- 2 *IsModified*: es una propiedad virtual que nos indicará si el contenido de la ventana ha sido modificado por el usuario. Fácil, ¿verdad? Pero hay un aspecto de la semántica de *IsModified* que, por el momento, sólo se lo puedo explicar en “lenguaje natural”: supondremos que *IsModified* se utilizará con poca frecuencia. El *layout manager*, de hecho, consulta *IsModified* sólo cuando se intenta cerrar una ventana o un diálogo modal. Esta premisa permitirá que *IsModified* incluya operaciones relativamente costosas, como terminar el modo de edición de una fila. ¿Por qué es necesaria esta aclaración? Pues porque no debemos usar *IsModified* para controlar el estado de controles, como podríamos hacer interceptando el evento *Idle* de la aplicación.
- 3 *SaveChanges*: dos céntimos si adivina qué hace. Observe, de todos modos, que *SaveChanges* devuelve un valor de tipo lógico. Suponga que estamos editando contenido XML y que debemos guardarlo en un fichero nuevo. Al cerrar la ventana, la aplicación nos pide el nombre del fichero como parte de la implementación de *SaveChanges*. Pero nos arrepentimos y cancelamos la ejecución del cuadro de diálogo. *SaveChanges* ha terminado, pero no hemos podido guardar los cambios. En ese extraño y rebuscado caso, este método debe devolver **false**. En todos los ejemplos del curso nos bastará con devolver **true**.
- 4 *DiscardChanges*: ¿hay que hacer algo si cancelamos una modificación en marcha? Si la respuesta es afirmativa, tendremos que colocar el código necesario dentro de *DiscardChanges*.

El tercer y último bloque de funcionalidad está formado por dos métodos:

- 5 *SaveLayout*: si dentro de una ventana hay controles como *DataGrid*, que el usuario puede personalizar modificando el ancho o la posición de sus columnas, nos interesaría guardar estos cambios en el diseño, ya sea en el registro de Windows, en un fichero XML o donde quiera que se le antoje. *SaveLayout* debe ser el responsable de esta operación, al cerrarse una ventana.
- 6 *RestoreLayout*: como puede imaginar, es la operación que restaura el estado de una ventana cuando volvemos a crear una instancia suya. Si cree que *RestoreLayout* sobra, porque podríamos hacerlo lo mismo con el constructor o incluso con *InitData*, se equivoca. A decir verdad, *no* se equivoca, pero si mezclamos la inicialización del contenido con la restauración del diseño visual, el código se complicará innecesariamente.

## ¿Edición... sobre una rejilla?

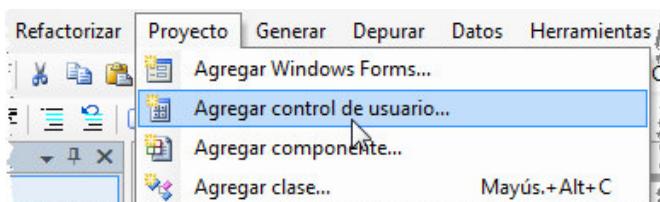
Estamos hablando de métodos de edición para una ventana de búsqueda y navegación, que en el noventa y nueve por ciento de los casos contendrá una rejilla... de sólo lectura. Le confieso que detesto que el usuario pueda modificar directamente el contenido de una rejilla con los resultados de una búsqueda. Conozco rejillas muy sofisticadas y muy bien hechas, pero incluso en las mejores la edición *in situ* es una operación preñada de peligros:

- Para empezar, el usuario es el primer peligro. Pide los datos de los cincuenta mejores clientes. Intenta robarle el diario deportivo a su vecino de puesto, tropieza y pulsa la tecla SUPR. *Bye, bye*, cincuenta mejores clientes. Y la culpa será del programador, como siempre.
- El segundo peligro lo aporta el propio programador, por supuesto. He observado a un usuario añadiendo un registro con devoción dentro de una rejilla, y poniendo su alma y corazón en la faena. He visto como el usuario no ha tenido la precaución de cambiar la fila activa. He visto al usuario cerrando la ventana, y he imaginado al código fuente permitiendo el cierre sin detectar el cambio en la rejilla (cambio que todavía no ha pasado al conjunto de datos). He contemplado al usuario regresando a la ventana en cuestión. Y le he oído maldecir al programador con palabras blasfemas que no deberían pronunciarse ni siquiera en los dominios de Lucifer.

¿Por qué me he tomado tanto trabajo añadiendo los métodos de control de cambio a *IWindow*? Pues porque creo en la Libertad (con mayúscula inicial). Y si usted, de todos modos, se empeña en permitir ediciones sobre la rejilla de una ventana de búsqueda y navegación, no seré yo quien se lo impida. En definitiva, será sobre usted que lloverá fuego y azufre, y entonces se oirá el llanto y crujir de dientes. Pues muchos son los llamados, pero pocos los elegidos, y etcétera, etcétera, etcétera...

## Controles de usuario

Para crear el control de usuario que utilizaremos como clase base, abrimos el menú *Proyecto* y ejecutamos el comando *Agregar control de usuario*:



Al nuevo control lo llamaremos *BaseWindow*, y tenemos que ir al fichero de código para añadirle la interfaz *IWindow* en la cabecera de la clase:

```
public class BaseWindow : UserControl, IWindow
{
    :
}
```

En cuanto haya terminado de teclear la última *w* de *IWindow*, Visual Studio mostrará una ayuda flotante sugiriéndole que pulse la tecla TAB para implementar automáticamente los métodos y propiedades del tipo de interfaz. Si no hace caso al aviso y pierde el tren, no se tire de los pelos: puede ir a la *Vista de clases*, localizar el nodo correspondiente a *BaseWindow*, buscar dentro de éste el nodo asociado a *IWindow*, y ejecutar el comando *Implementar interfaz*, del menú emergente.

La implementación que Visual Studio añade consiste en propiedades y métodos públicos, como estos dos que he seleccionado entre todos los exigidos por *IWindow*:

```
public virtual void InitData()
{
}

[Browsable(false)]
public virtual bool IsModified
```

```
{
    get { return false; }
}
```

Me he permitido la libertad, como puede ver, de añadir un modificador virtual a las propiedades y métodos añadidos. De esta forma, *BaseWindow* proporcionará una implementación lo más general y completa posible de *IWindow*, y las clases derivadas podrán afinar los algoritmos.

Como curiosidad técnica, veamos qué alternativas tenemos. C# no permite implementar tipos de interfaz mediante recursos protegidos. La alternativa que tenemos a los métodos públicos es la implementación explícita. Podríamos modificar las dos declaraciones anteriores de la siguiente forma:

```
void IWindow.InitData()
{
    // ¡No es una llamada recursiva!
    InitData();
}

bool IWindow.IsModified
{
    get { return IsModified; }
}
```

Ha desaparecido el modificador de accesibilidad, y se asume que los nuevos métodos son privados. Observe también que el nombre del tipo de interfaz se tiene que usar ahora como prefijo del nombre del recurso implementado. Para ser más exactos, no se trata de que hayamos definido un método privado *InitData*: en realidad, no podemos llamar al nuevo *InitData* de forma explícita, ni siquiera dentro de la clase. Lo que sí podemos hacer es convertir una referencia de tipo *BaseWindow* en una referencia de tipo *IWindow*, y llamar entonces a *InitData*, de forma indirecta. La técnica de implementación explícita sirve precisamente para este objetivo: obligar al cliente de la clase a obtener un puntero de interfaz explícitamente si desea aprovechar la funcionalidad del tipo de interfaz.

No obstante, el listado anterior contiene llamadas a unos misteriosos *InitData* e *IsModified*. No se trata de llamadas recursivas sino que, en el caso de implementaciones explícitas, suelen añadirse métodos protegidos virtuales para proporcionar el código necesario:

```
protected virtual void InitData()
{
}

protected virtual bool IsModified
{
    get { return false; }
}
```

Si es la primera vez que tropieza con estos detalles de C# y la digresión le suena a física teórica explicada en chino mandarín por un hotentote con dolor de muelas, no se preocupe por el gasto en aspirinas: nos vamos a quedar con la técnica inicial de implementación implícita mediante recursos públicos. Recursos virtuales, eso sí...

**NOTA**

Los hotentotes hablan un extraño lenguaje que en su repertorio de fonemas incluye crujidos y chasquidos de diversos tipos. Imagine que está desayunando y que en la cocina entra su cónyuge recién levantada y sin peinar, y le pide: "... pásame la (k/k)cafetera, (krk)cariño". El idioma chino, o como quiera que los chinos lo llamen, distingue significados diferentes para una misma palabra según la entonación que se le dé. Explico esto porque no quiero que piense que creo imposible que un hotentote domine los misterios de la física teórica y la musicalidad del idioma de los Han. Lo que he querido decir es que, incluso en tal caso, ni usted ni yo entenderíamos una jota.

## Cambios en los gestores de menús y ventanas

Necesitamos hacer algunos ajustes en las clases de gestión globales para aprovechar la funcionalidad añadida a *IWindow*. Comenzaremos, no obstante, haciendo una pequeña mejora en la clase asociada al atributo *NonModal*. Añadiremos una propiedad *Caption*, de lectura y escritura:

```
// Clase: NonModalAttribute
public string Caption
{
    get { return caption != "" ? caption : description; }
    set { caption = value; }
}
```

El cambio es necesario porque vamos a trabajar con descendientes de *UserControl* para las ventanas de navegación. Cuando usábamos ventanas, podíamos configurar el título de la ventana en tiempo de diseño, y usar la propiedad *Description* del atributo *NonModal* sólo para el texto del comando de menú. Ya no tenemos esa posibilidad, y necesitaremos *Caption* para que el gestor de ventanas asigne la propiedad *Text* del formulario donde incrustará el control de usuarios.

**NOTA**

Como alternativa, podríamos añadir una propiedad modificable en tiempo de diseño a la clase *BaseWindow*. Pero esto añadiría una dependencia indeseable: el gestor de ventanas comenzaría a exigir controles derivados de *BaseWindow*, mientras que ahora sólo necesita un control de usuarios que implemente el tipo de interfaz *IWindow*: un requisito menos exigente.

A continuación, modificaremos el método *CreateMenu* dentro de la clase *BaseMenuManager*:

```
public virtual void CreateMenu()
{
    foreach (Type type in
        Assembly.GetExecutingAssembly().GetTypes())
        if (type.IsSubclassOf(typeof(UserControl)) &&
            type.IsDefined(typeof(NonModalAttribute), false))
    {
        NonModalAttribute[] attrs = type.GetCustomAttributes(
            typeof(NonModalAttribute), false) as NonModalAttribute[];
        if (attrs != null && attrs.Length > 0)
            menuTable[CreateMenuItem(attrs[0])] = type;
    }
}
```

El cambio aparece resaltado, y consiste en filtrar la lista de tipos dejando sólo los descendientes de *UserControl*, en vez de las clases derivadas de *Form*, como en el código original.

La próxima parada será en la clase *MdiLayoutManager*. Como vamos a necesitar el acceso al atributo *NonModal* para más operaciones, modificaremos la implementación de *IsUniqueInstance*, delegando parte de su código en el nuevo método *GetAttr*:

```
protected static NonModalAttribute GetAttr(Type type)
{
    NonModalAttribute[] attrs = type.GetCustomAttributes(
        typeof(NonModalAttribute), false) as NonModalAttribute[];
    return attrs[0];
}

protected static bool IsUniqueInstance(Type type)
{
    return GetAttr(type).UniqueInstance;
}
```

Y ya estamos listos para los cambios realmente importantes en el gestor de ventanas. Esta es la nueva implementación del método *Activate* en la clase *MdiLayoutManager*:

```
public IWindow Activate(Type type)
{
    foreach (Form f in mdiParent.MdiChildren)
        if (f.Controls.Count == 1 && f.Controls[0].GetType() == type)
    {
        if (f.WindowState == FormWindowState.Minimized)
            f.WindowState = FormWindowState.Normal;
        f.BringToFront();
        return f.Controls[0] as IWindow;
    }
}
```

```
        return null;
    }
```

La condición resaltada en el listado anterior sustituye a la siguiente condición, que era la que usábamos en el ejercicio anterior:

```
if (f.GetType() == type)
```

Antes, asumíamos que la ventana era el propio formulario hijo MDI. Ahora, la “ventana” que buscamos debe ser el primer control añadido dentro del formulario hijo MDI. ¿Por qué tanta certeza? Muy fácil: nos ocuparemos nosotros mismos de que se cumpla la profecía.

## Guardar los cambios

Esta es la nueva implementación del método *Create* del gestor de ventanas:

```
public IWindow Create(Type type)
{
    IWindow wnd = IsUniqueInstance(type) ? Activate(type) : null;
    if (wnd == null && type != null)
    {
        UserControl ctrl = type.InvokeMember(null,
            BindingFlags.DeclaredOnly | 
            BindingFlags.Public | BindingFlags.NonPublic | 
            BindingFlags.Instance | BindingFlags.CreateInstance,
            null, null, null) as UserControl;
        wnd = ctrl as IWindow;
        ctrl.Dock = DockStyle.Fill;
        Form f = new Form();
        f.Text = GetAttr(type).Caption;
        f.Controls.Add(ctrl);
        wnd.InitData();
        f.MdiParent = mdiParent;
        f.Closing +=
            new System.ComponentModel.CancelEventHandler(FormClosing);
        f.Show();
        if (CreateWindow != null)
            CreateWindow(ctrl, EventArgs.Empty);
    }
    return wnd;
}
```

El cambio más importante es que la referencia a objeto devuelta por *InvokeMember* se convierte ahora al tipo *UserControl*. Una vez que ha creado el control de usuario, el método crea también un formulario, le asigna un título y añade el control de usuarios a la lista de controles del formulario; como hemos forzado el valor *DockStyle.Fill* en la propiedad *Dock* del control, éste se estira para ocupar toda su área interior. Entonces es que ejecutamos el método *InitData* de la interfaz *IWindow* implementada por el control de usuario. A continuación, se asigna la ventana principal en la propiedad *MdiParent* del nuevo formulario, para convertirlo en una ventana hija MDI.

Y entonces encontramos un detalle interesante: el gestor de ventanas añade un método a la lista de manejadores del evento *Closing* del formulario. El método se llama *FormClosing* y está declarado e implementado dentro de la propia *MdiLayoutManager*:

```
private void FormClosing(object sender, CancelEventArgs e)
{
    IWindow wnd = (sender as Form).Controls[0] as IWindow;
    if (wnd.IsModified)
        switch (MessageBox.Show(
            "Esta ventana ha sido modificada.\n" +
            "¿Desea guardar los cambios?", "Confirmación",
            MessageBoxButtons.YesNoCancel, MessageBoxIcon.Information))
    {
        case DialogResult.Cancel:
            e.Cancel = true;
            break;
```

```

        case DialogResult.Yes:
            if (! wnd.SaveChanges())
                e.Cancel = true;
            break;
    }
}

```

El evento *Closing* se dispara cuando se va a cerrar un formulario, para comprobar si podemos echar el cerrojo. Si al terminar el método, la propiedad *Cancel* del segundo parámetro del evento vale **true**, Windows Forms interrumpe la operación. Nuestro manejador de eventos implementa el típico comportamiento de un procesador de textos cuando intentamos abandonar un documento modificado sin antes guardarlo. Para poder programar este algoritmo, dependemos de la implementación del tipo de interfaz *IWindow* por el control incrustado dentro del formulario. Observe la forma en la que obtenemos el puntero *IWindow* a partir del formulario emisor del evento:

```
IWindow wnd = (sender as Form).Controls[0] as IWindow;
```

Para terminar con el método *Create*, estas son las últimas instrucciones que ejecuta antes de hacer mutis por el foro:

```

f.Show();
if (CreateWindow != null)
    CreateWindow(ctrl, EventArgs.Empty);

```

Es decir, mostramos el formulario, y avisamos a cualquiera que esté interesado que hemos creado una ventana, disparando el evento *CreateWindow* del gestor de ventanas.

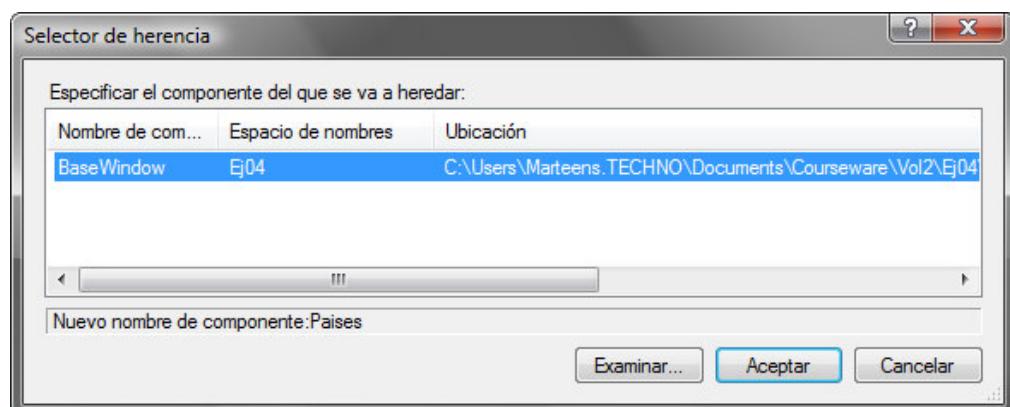
**NOTA**

Al asignar un manejador para el evento *Closing* del formulario, no impedimos que el programador pueda crear *otro* manejador para el mismo evento en una clase derivada hipotética. Esto es así gracias a que los eventos en .NET soportan *multidifusión (multicasting)*, y podemos asignar más de un manejador por evento.

## Herencia visual

En el segundo ejercicio, en la página 24, creamos dos formularios, llamados *Ventana1* y *Ventana2*, para probar el funcionamiento del gestor de ventanas. Ahora los eliminaremos, y los sustituiremos por los controles de usuario equivalentes, para mostrar la técnica que usaremos en los restantes ejercicios cada vez que necesitemos una ventana de búsqueda y navegación. Para eliminar los formularios, basta con activar el Explorador de Soluciones, localizar los nodos correspondientes, y ejecutar el comando Eliminar del menú de contexto asociado a estos nodos. Como todo el código que tenía que ver con estos formularios estaba aislado dentro de sus ficheros de código, no tenemos que retocar ni una sola línea en el resto del proyecto.

Para añadir los sustitutos, debe ejecutar el comando *Proyecto | Agregar control heredado*. Primero nos pedirán el nombre del control: llámelo *Paises* (¡puede usar el acento si así lo desea!). A continuación aparecerá un cuadro de diálogo, para que seleccionemos el control que utilizaremos como base de la herencia. Cómo sólo existe *BaseWindow*, por el momento, la elección es clara:



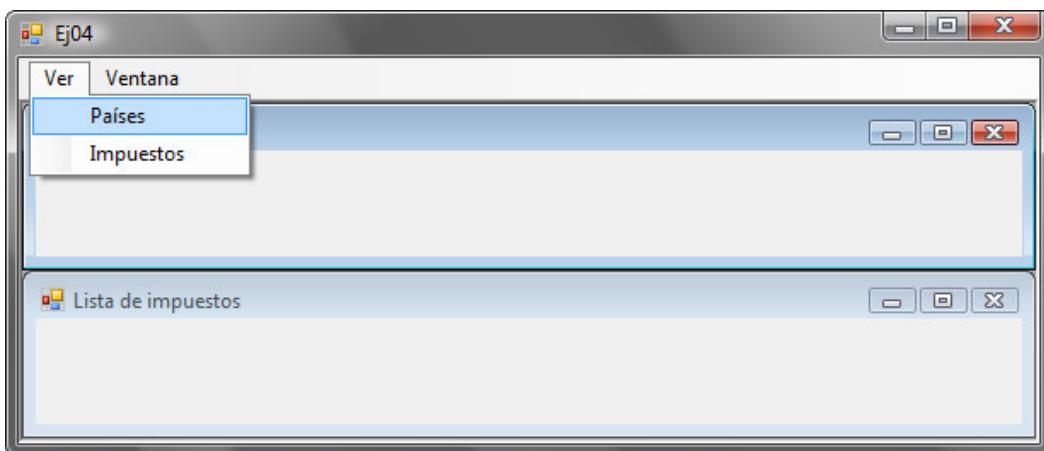
Una vez creado el control heredado, debemos ir al código fuente, a la cabecera de la clase, y decorarla con el atributo *NonModal*:

```
[NonModal ("Países", "Ver",
    Caption="Lista de países", UniqueInstance=true) ]
public class Países: Ej04.BaseWindow
{
    :
}
```

Traduzco, por si ha olvidado el significado de los parámetros de *NonModal*:

- 1** El comando de menú mostrará *Países* como texto.
- 2** El comando de menú se creará dentro del submenú *Ver*.
- 3** La ventana que se creará con el comando de menú se titulará *Lista de países*.
- 4** Sólo habrá una ventana de este tipo como máximo, en cualquier momento.

Repita los pasos para crear una “ventana” *Impuestos*, y ejecute la aplicación:



Compruebe lo que ocurre si modifica el valor de la propiedad *UniqueInstance* del atributo *NonModal*.

---

### EJERCICIO PROPUESTO

Intente programar un *layout manager* que utilice un control de páginas para alojar las ventanas, de forma parecida al conocido modelo de Visual Studio. El principal problema será el mantenimiento de un menú con la lista de “ventanas” disponibles, como la lista mantenida automáticamente en aplicaciones MDI.

---

# EJERCICIO 05

## Objetivos

Algoritmos básicos de navegación y edición

## Técnicas introducidas

Grabación, conciliación, manejo de identidades

**T**ENEMOS YA PREPARADAS LAS VENTANAS de navegación, pero éstas aparecen en el monitor como cascarones vacíos. En este ejercicio les insuflaremos vida, añadiendo los algoritmos más sencillos de recuperación de registros y de grabación de cambios como parte de la implementación del tipo de interfaz *IWindow*.

## Propiedades visibles en tiempo de diseño

Los algoritmos de inicialización y control de cambios dependen principalmente del conjunto de datos *principal* de la ventana. Enfatizo lo de “principal” porque más adelante veremos casos en los que nos convendrá tener varios conjuntos de datos dentro de una misma ventana.

¿Cómo podemos indicar cuál es el conjunto de datos principal? Como hemos estado trabajando con atributos y reflexión, estoy casi seguro de que su primer impulso será detectarlo por reflexión o especificarlo por medio de un atributo. Si es así, conténgase:

- 1 La técnica de atributos es estupenda para configurar información a nivel de la clase, no de la instancia. Enseguida veremos por qué.
- 2 Por una parte, la reflexión añade carga en tiempo de ejecución. El tiempo de ejecución añadido no es importante en la mayoría de los casos, para ser honestos, pero ¿por qué pagar el sobreprecio cuando existen alternativas? Además, hay un problema con la reflexión, que probablemente ya haya descubierto trabajando con el gestor de menús: es imposible predecir el orden en que se detectan las instancias. Podríamos poner un parche indicando el orden de tratamiento mediante atributos, pero ya habríamos complicado una técnica que nos gustaba por su sencillez.

La alternativa es sorprendentemente sencilla, y la hemos tenido siempre ante nuestros ojos: añadimos una propiedad de tipo *DataSet* a la ventana base, de modo que podamos configurarla en tiempo de diseño. En un entorno de programación RAD, ¡no hay nada más cotidiano que esta operación! En el primer punto de la lista anterior dije que los atributos eran útiles cuando había que añadir información a nivel de la clase, no de las instancias. Ahora puedo explicárselo:

- Cuando hay que configurar información a nivel de instancias, lo mejor es usar de propiedades en tiempo de diseño. En cambio, ninguno de los entornos RAD con los que he trabajado permiten editar propiedades estáticas en tiempo de diseño (¡se me acaba de ocurrir una buena idea!). Por lo tanto, tenemos que realizar la configuración manualmente, modificando el código fuente. Y en estas circunstancias, los atributos son casi siempre la mejor opción, al estar claramente aislados del resto del código ejecutable.

### NOTA

Pensándolo un poco, es extraño que Visual Studio no incluya ningún editor gráfico para los atributos de un formulario o de una instancia... al menos, hasta donde sé. Y hay que reconocer que el modelo de edición de propiedades en VS es muy potente: tenemos las propiedades dinámicas, el sistema de enlace de datos, que en Whidbey mejora extraordinariamente, los extensores de propiedades...

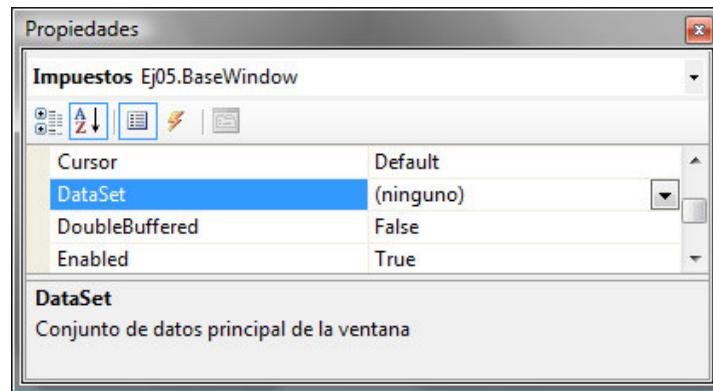
Es juego de niños declarar una propiedad visible en tiempo de diseño para un formulario. Active el editor de código y teclee la siguiente declaración dentro de la clase *BaseWindow*:

```
[DefaultValue(null)]
[Description("Conjunto de datos principal de la ventana")]
[Browsable(true)]
public DataSet DataSet
{
    get { return dataset; }
    set { dataset = value; }
}
```

Necesitaremos también la siguiente variable:

```
private DataSet dataset = null;
```

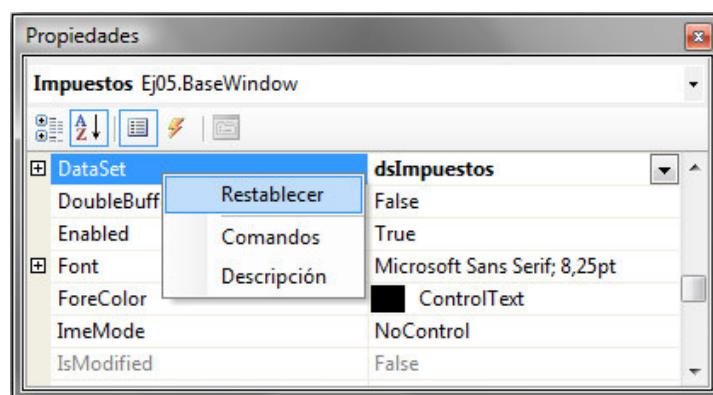
Compile el proyecto y abra entonces algún control derivado de *BaseWindow*, como el control utilizado para la ventana de *Impuestos*:



Observe que en el Inspector de Propiedades ha aparecido automáticamente una propiedad *DataSet*, incluso con una descripción que en la imagen anterior se muestra en el panel inferior, pero que también aparecería en la ventana de IntelliSense, en el editor de código. En realidad, lo complicado sería *evitar* que la propiedad pública *DataSet* se comportase de esta forma: toda propiedad pública de un formulario o control cuyo tipo de retorno pueda ser “serializado en código”, aparece espontáneamente en el Inspector de Propiedades si no tomamos medidas para prevenirlo.

Veamos para qué sirven los tres atributos que hemos asociado a la propiedad *DataSet*:

- *Browsable*: decide si la propiedad estará disponible en tiempo de diseño o no, según el valor del parámetro de tipo lógico de su constructor. Si no especificamos *Browsable*, se asume que sí, como acabo de explicar. Mi propósito al utilizar explícitamente este atributo es facilitar la comprensión del código fuente.
- *Description*: descripción de la propiedad. Aparecerá en el Inspector de Propiedades y en la ventana de ayuda de IntelliSense.
- *DefaultValue*: indica cuál es el valor por omisión de la propiedad. El módulo de Visual Studio que genera el código de inicialización del formulario utiliza *DefaultValue* para determinar si tiene que inicializar explícitamente la propiedad o si puede confiar en que el constructor de la clase se encargue de ello. Además, cuando *DefaultValue* se ha configurado para una propiedad, el Inspector de Objetos puede mostrar el comando *Restablecer*, para asignar el valor por defecto a una propiedad cuyo valor ha sido modificado.



Otro atributo interesante para configurar el comportamiento de una propiedad en tiempo de diseño es *Category*, que sirve para indicar la categoría a la que pertenece la propiedad. No lo he utilizado porque confieso que uso muy poco la vista agrupada por categorías del Inspector de Propiedades.

## El algoritmo básico de inicialización

El primer método en aprovechar la presencia de la nueva propiedad es *InitData*, el método que se llama cuando el control de usuario ya se ha añadido al formulario en el que residirá definitivamente. La implementación inicial de este método recuperará los registros asociados al conjunto de datos indicado utilizando la instancia global del módulo de datos:

```
public virtual void InitData()
{
    if (dataset != null)
        dataset.Merge(Data.Instance.Read(dataset.DataSetName), false);
}
```

Es decir, si hay un conjunto de datos asignado a la propiedad, pedimos a la interfaz *IDataServer* global que lea el conjunto de registros asociados al nombre del conjunto de datos. El resultado se mezcla con el contenido original del conjunto de datos. Como se trata de un método llamado en un momento dado de la inicialización, podemos confiar en que el conjunto de datos esté vacío.

Un detalle técnico: en vez de usar la propiedad en sí, *DataSet*, la inercia me ha llevado a utilizar directamente el campo asociado. En términos de eficiencia, debería dar lo mismo, porque se supone que el JIT debería expandir las llamadas a *DataSet* en línea, sustituyéndolas por el acceso directo al campo. Sin embargo, el JIT actual nunca expande métodos o propiedades pertenecientes a clases derivadas de la clase *MarshalByRefObject*, no sea que alguien acceda remotamente a una de sus instancias.

## ¿Hay cambios?

Si conocemos cuál es el conjunto de datos que vamos a examinar y probablemente modificar, podemos escribir algoritmos genéricos para las operaciones de soporte de la edición. Por ejemplo, podríamos comprobar de esta forma si hay cambios en el conjunto de datos:

```
public virtual bool IsModified
{
    get
    {
        return dataset != null && dataset.HasChanges();
    }
}
```

Recuerde, no obstante, que es muy poco probable que realicemos cambios directos sobre la ventana de navegación: estamos añadiendo este código porque nunca viene mal tenerlo a mano, y porque con pequeñas precisiones y adiciones lo volveremos a ver en las ventanas modales de edición.

Hay también que tener mucho cuidado con el comportamiento de los controles de edición. Muchos controles retrasan todo lo posible la propagación de los cambios sufridos por el control hacia el conjunto de datos al que están vinculados. Si fuese éste el caso, deberíamos forzar la propagación de cambios desde el control antes de llamar a nuestra versión de *IsModified*. Más adelante, resolvaremos este problema indirectamente, a través de los componentes *BindingSource* que colocaremos como intermediarios del enlace a datos en cada ventana. He preferido no añadir el código en este punto del sistema porque serán pocas las rejillas que permitiremos modificar directamente.

## El algoritmo básico de grabación

Lo más sencillo en este Universo es enviar los cambios realizados sobre el conjunto de datos a la base de datos SQL... sobre todo si ignoramos todo lo que puede ir mal. De momento, nos conformaremos con el siguiente algoritmo:

```
public virtual bool SaveChanges()
{
    DataSet delta =
        Data.Instance.Write(dataset.DataSetName, dataset.GetChanges());
```

```
        dataset.Merge(delta, false);
        dataset.AcceptChanges();
        return true;
    }
```

Estoy asumiendo que existen cambios en el conjunto de datos. Para ello hemos comprobado antes el valor de *IsModified*, ¿no? De no ser así, tendríamos que comprobar que el resultado devuelto por *GetChanges* no sea un puntero vacío.

También conviene tener una implementación para *DiscardChanges*:

```
public virtual void DiscardChanges()
{
    dataset.RejectChanges();
}
```

Normalmente, esta llamada es innecesaria, porque el conjunto de datos que editamos residirá casi siempre dentro del formulario de edición. Si cerrásemos el formulario sin guardar los cambios no pasaría nada: de todos modos, el conjunto de datos deja también de ser accesible y queda a merced del implacable recolector de basura.

## Conciliación en presencia de identidades

La implementación que hemos proporcionado a *SaveChanges* es mínima, y no es difícil encontrar 58 casos para los que tendremos que complicarla. El primer ejemplo: la inserción de registros en tablas que contienen columnas con el atributo **identity**. De hecho, todas nuestras tablas tienen como clave primaria una columna con dicho atributo. Para entender en qué consiste el problema, sigamos el rastro de una inserción de registros. Supongamos que el usuario ha tecleado el siguiente registro para la tabla de países:

```
< -1, 'PN', 'Polo Norte'... >
```

He puesto **-1** como clave primaria del nuevo registro, para evitar confusiones. Supondremos que este valor ha sido asignado automáticamente en el lado cliente. No obstante, al grabar el registro, el algoritmo de grabación recupera el valor asignado a la clave por el servidor. Este sería el estado del registro una vez insertado y actualizado:

```
<1234, 'PN', 'Polo Norte'... >
```

Recuerde, sin embargo, que la grabación se ejecuta sobre un conjunto de datos auxiliar que sólo contiene las filas con algún tipo de cambio. Para que el conjunto de datos original se actualice, tenemos que ejecutar la operación *Merge*. Este método utiliza las claves primarias para identificar los registros originales con sus copias más recientes. Y esto es un problema, precisamente porque la clave primaria ha sido modificada durante la inserción. Despues de ejecutar *Merge* tendríamos dos copias del mismo registro, con valores diferentes en la clave primaria:

```
< -1, 'PN', 'Polo Norte'... >
<1234, 'PN', 'Polo Norte'... >
```

¿Se da cuenta de que nos sobra el primero de los registros? Para evitar esta duplicación de las inserciones, tendríamos que eliminar los registros añadidos al conjunto de datos original antes de mezclar el contenido de *delta*. Esta es una versión más completa de *SaveChanges*:

```
public virtual bool SaveChanges()
{
    DataSet delta =
        Data.Instance.Write(dataset.DataSetName, dataset.GetChanges());
    foreach (DataTable tab in dataset.Tables)
        foreach (DataRow row in
            tab.Select(null, null, DataViewRowState.Added))
            row.Delete();
    dataset.Merge(delta, false);
    dataset.AcceptChanges();
    return true;
}
```

}

No importa si alguna tabla no utiliza el atributo de identidad. Si borramos una fila añadida dentro del conjunto de datos original, de todos modos volveremos a incorporarla al mezclar el *delta*. Observe que, para cada tabla, detectamos los registros añadidos gracias a su estado, aplicando el método *Select* a la tabla.

## Actualización del estado de elementos visuales

Aprovecharemos para configurar un sistema que nos permitirá activar y desactivar controles de acuerdo al estado del formulario. Por ejemplo, cuando añadimos un botón para eliminar registros, querremos que esté inactivo mientras no haya registros en el conjunto de datos principal, o en la tabla correspondiente al control de edición enfocado. La mejor técnica en estos casos consiste en interceptar el evento *Idle* de la aplicación al construir el control:

```
public BaseWindow()
{
    InitializeComponent();
    Application.Idle += new EventHandler(ApplicationIdle);
}
```

Observe que *Idle* es un evento estático de la clase *Application*; no obstante, su manejador es un método de instancia. El evento se dispara cada vez que la aplicación termina de responder a un mensaje de Windows y encuentra vacía la cola de mensajes. El manejador del evento es el siguiente:

```
private void ApplicationIdle(object sender, EventArgs e)
{
    UpdateButtons();
}

protected virtual void UpdateButtons()
```

El manejador es un método privado, que llama a un método protegido virtual que podemos redefinir en las clases derivadas de *BaseWindow*. Esto tiene sentido porque no hay ninguna información de importancia en los parámetros del evento. ¿Por qué obligar entonces al heredero de la clase a cargar con estos parámetros?

Y ahora preste mucha atención: cuando un formulario intercepta un evento disparado por alguna instancia global o estática, es sumamente importante eliminar el manejador de la cadena de delegados del evento al cerrar el formulario. Si no lo hacemos, el formulario, aunque esté cerrado, seguirá estando disponible a través de la siguiente ruta de objetos:



Y si el formulario sigue estando accesible, no será eliminado por el recolector de basura y nuestra aplicación perderá memoria cada vez que mostremos una instancia del formulario. Peor aún: el formulario zombie seguirá recibiendo notificaciones del evento *Idle*, y puede provocar un desastre al intentar actuar sobre controles para los que ya no existe un recurso de Windows correspondiente.

- 278 Por desgracia, no existe una forma sencilla de eliminar la asociación: aparentemente, no existe ningún evento o método que podamos interceptar para nuestros fines. Por fortuna, sólo en apariencia... El control de usuario es un componente, y todos los componentes tienen un evento llamado *Disposed*, que se dispara al ser destruido el componente. El evento no está disponible en tiempo de diseño, por razones históricas, pero sigue estando ahí, a nuestra merced, en tiempo de ejecución. Lo que haremos es registrar un manejador para este evento al inicializar el control, y aprovecharlo para romper el enlace con el evento:

```
public BaseWindow()
```

```
{  
    InitializeComponent();  
    Application.Idle += new EventHandler(ApplicationIdle);  
    this.Disposed += new EventHandler(BaseWindow_Disposed);  
}  
  
private void BaseWindow_Disposed(object sender, EventArgs e)  
{  
    Application.Idle -= new EventHandler(ApplicationIdle);  
}
```

Puede comprobar estableciendo un punto de ruptura, que este método es realmente ejecutado cuando cerramos el formulario.

# EJERCICIO 06

## Objetivos

Consultas sobre tablas sencillas

## Técnicas introducidas

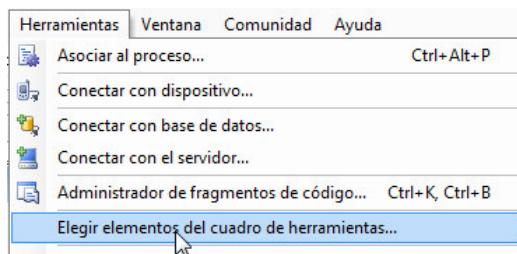
Adaptadores, conjuntos de datos con tipos

**V**AMOS A PONER A PRUEBA LA INFRAESTRUCTURA que con tanto trabajo hemos montado en los ejercicios anteriores. Nuestra primera tarea será mostrar el contenido de la tabla de países: una tabla que sabemos que no va a crecer desmesuradamente... al menos mientras nuestro negocio se limite al planeta Tierra. De paso, prepararemos las condiciones para poder modificar registros de esta tabla, aunque necesitaremos más ejercicios para añadir los diálogos modales de edición que usaremos en la aplicación final.

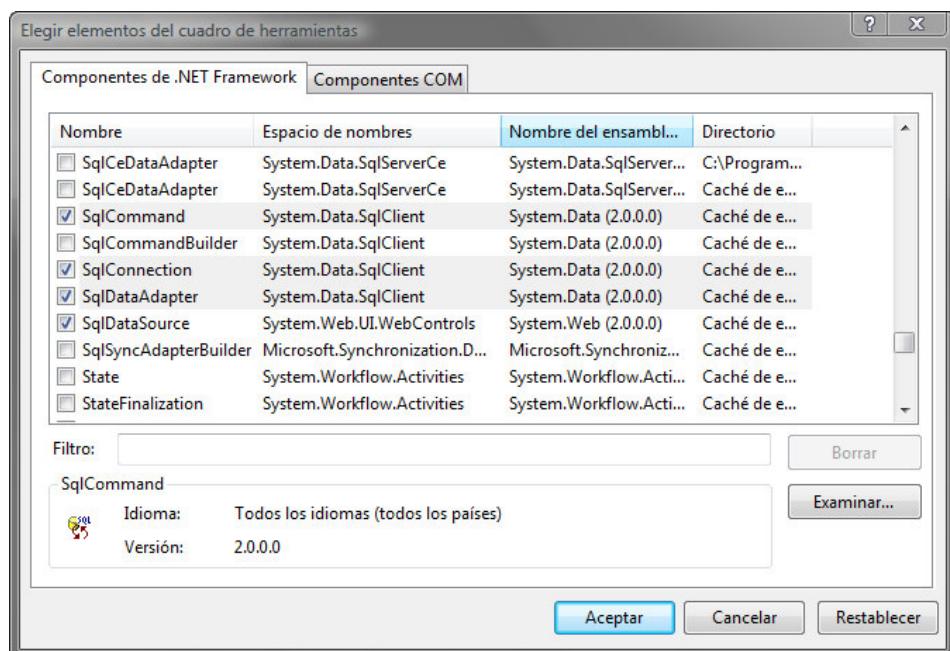
## Una conexión central

El primer problema con el que tropezaremos es que los componentes de acceso a SQL han sido escondidos por Microsoft, a partir de Visual Studio 2005. Se supone que los nuevos *adaptadores de tablas*, que se estudian en la serie C del Volumen I de este curso, son la alternativa ideal y completa de los viejos *adaptadores de datos*. Al nuevo sistema, sin embargo, le cuesta más adaptarse al estilo de programación en tres capas, por lo que en este curso vamos a usar los componentes más potentes que, de todos modos, son utilizados en el fondo por el nuevo sistema.

Para desenterrar los componentes ocultos, debemos ejecutar el comando *Elegir elementos del cuadro de herramientas*, del menú *Herramientas*:



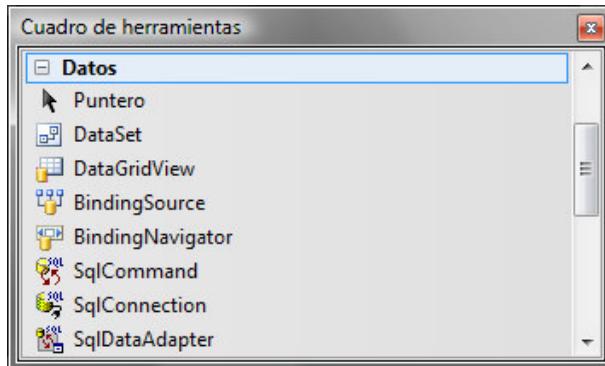
El diálogo que aparecerá como respuesta es el siguiente:



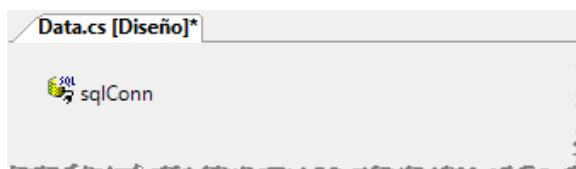
Tenemos que activar las casillas de estos tres componentes:

- *SqlConnection*
- *SqlCommand*
- *SqlDataAdapter*

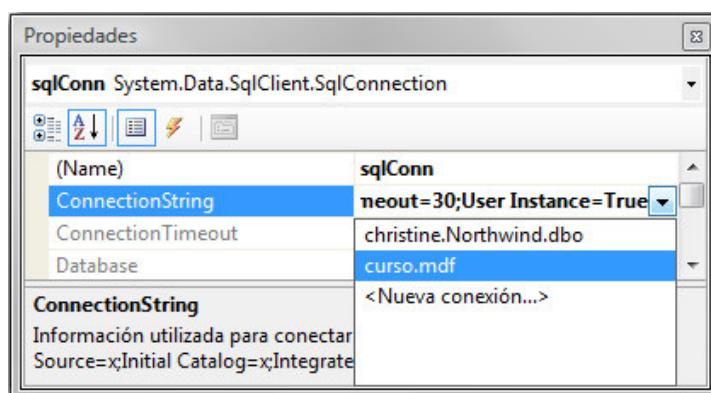
Para comprobar si el cambio de configuración ha funcionado correctamente, abra el *Cuadro de herramientas* de Visual Studio y compruebe si los tres componentes seleccionados aparecen en la sección *Datos*:



Active el editor visual del componente *Data*, que estamos usando como intermediario para la [256](#) comunicación con la base de datos. Añada un *SqlConnection* sobre la superficie de diseño y configure su propiedad *ConnectionString* para conectarlo con la base de datos del curso:



Para configurar la cadena de conexión, despliegue el combo asociado a la propiedad *ConnectionString* del componente:



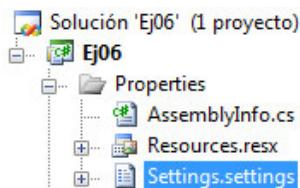
Si me hizo caso al leer el primer ejercicio (página 1), debe tener una entrada en la lista que corresponda a la base de datos del curso, tal y como fue registrada en el Explorador de servidores. Al elegirla, Visual Studio copiará la cadena de conexión correspondiente en nuestro componente. En mi caso, se trata de la siguiente cadena, en la que he abreviado la ruta al fichero:

```
DataSource=.\SQLEXPRESS;AttachDbFilename=c:\...\curso.mdf; Integrated Security=True;Connect Timeout=30;User Instance=True
```

Esta cadena, por desgracia, queda almacenada en el ejecutable de la aplicación. No se trata de un problema de seguridad, porque al utilizar la seguridad integrada, hemos evitado almacenar claves y contraseñas que pueden quedar a la vista de todos. Se trata de la posibilidad de modificar parámetros de la conexión sin necesidad de recomilar el programa. Y la solución más sencilla es pasar la

cadena al fichero de configuración de la aplicación. No es una solución trivial: en Windows Vista, por ejemplo, el fichero de configuración suele instalarse en sitios insospechados, y tengo serias dudas sobre lo que ocurre con los permisos de escritura. Pero al menos, reduce la complejidad del mantenimiento.

Busque en el Explorador de Servidores, el nodo *Properties* del proyecto. Si lo expande, encontrará un nodo titulado *Settings.settings* en su interior:



Haciendo doble clic sobre este nodo, debe aparecer el editor de la configuración del proyecto:

Nombre	Tipo	Ámbito	Valor
Conexion	(Cadena de ...)	Aplicación	Data Source=.\SQLEXPRESS;AttachDbFilename=C:\Users\Marlene.TECHNO\Documents\Courseware\Vol2\Ej00\curso.mdf;Integrated Security=True;Connect Timeout=30;User Instance=True
*			

Debemos añadir un nuevo elemento al editor, del tipo especial *Cadena de configuración...* y volver a especificar los parámetros de la conexión. Llamaremos *Conexion* al nuevo parámetro.

Después de guardar los cambios y compilar, vaya al código fuente del componente *Data* y modifique los dos constructores predefinidos del componente de la siguiente manera:

```
public Data()
{
    InitializeComponent();
    sqlConn.ConnectionString = Properties.Settings.Default.Conexion;
}

public Data(IContainer container)
{
    container.Add(this);
    InitializeComponent();
    sqlConn.ConnectionString = Properties.Settings.Default.Conexion;
}
```

Observe que no existe vínculo automático alguno entre la cadena almacenada en la conexión, en tiempo de diseño, y el valor que se lee y asigna en tiempo de ejecución. No hay nada malo en ello, siempre que usted tenga conciencia de lo que está pasando. En cualquier caso, esta conexión será la única de todo el proyecto.

## Configuración de los adaptadores

- 321 El siguiente paso es configurar un adaptador de datos para poder leer y modificar los registros de la tabla de países. Añada un componente *SqlDataAdapter* sobre *Data* para que aparezca el asistente de configuración de adaptadores.

Seleccione el componente de conexión en su primera página, y en la segunda indique que va a utilizar instrucciones SQL para configurar el componente. Ya en la tercera página, teclee la siguiente consulta, o constrúyala con la ayuda del generador de consultas:

```
select IDPais, Codigo, Pais, FormatoCP, FormatoNIF, Creacion, TS
from Paises
```

Aunque usted utilice la abreviatura “**select \***”, el asistente se empeñará en generar la lista completa de campos: no tiene mayor importancia. A continuación, pulse el botón de opciones avanzadas que se encuentra en la esquina inferior izquierda de la página, y asegúrese de que las tres opciones siguientes están activas:

Se pueden generar instrucciones adicionales Insert, Update y Delete para actualizar el origen de datos.

**Generar instrucciones Insert, Update y Delete**

Genera instrucciones Insert, Update y Delete basadas en la instrucción Select.

**Usar concurrencia optimista**

Modifica instrucciones Update y Delete para detectar si la base de datos ha cambiado desde que se cargó el registro en el conjunto de datos. Esto ayuda a prevenir conflictos de concurrencia.

**Actualizar el conjunto de datos**

Agrega una instrucción Select después de instrucciones Insert y Update para recuperar valores de columnas de identidad, valores predeterminados y otros valores calculados por la base de datos.

La segunda opción, *Usar concurrencia optimista*, sirve para controlar la posibilidad de grabaciones sobre un mismo registro desde diferentes puestos. Si nuestra tabla no tuviese un campo de tipo **rowversion**, esto significaría incluir todas las columnas de la tabla en las cláusulas **where** de las instrucciones **update** y **delete** que genera el asistente. En nuestro caso, sin embargo, el código generado será más sencillo y eficiente.

Una vez que cierre el asistente de configuración del adaptador, active el Inspector de Propiedades, para examinar los tres comandos SQL incrustados dentro del adaptador de datos. Veamos primero la propiedad *CommandText* del componente asociado a la propiedad *DeleteCommand* del adaptador:

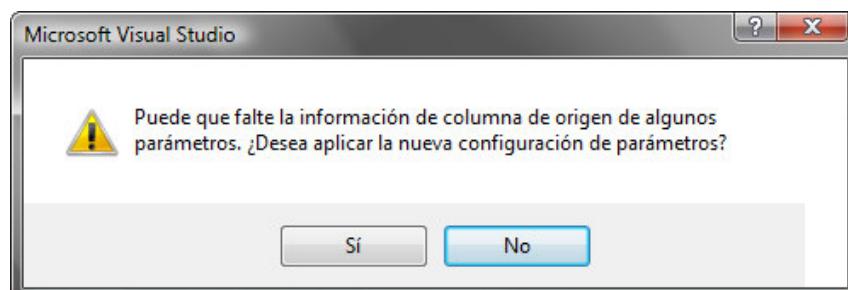
```
delete from Paises  
where IDPais = @Original_IDPais and TS = @Original_TS
```

La particularidad está en la comparación que he resaltado. Esta comparación verifica que no se hayan producido cambios en el registro desde el momento en que lo leímos en nuestra memoria local. Si tiene muchas ganas de trabajar, modifique la propiedad *UpdateRowSource* del comando de borrado asignándole *None*, porque después de un borrado no hay fila que releer. 347

Nuestro siguiente objetivo es el comando almacenado en *UpdateCommand*. Aquí debemos poner algo más de atención, porque necesitamos un par de cambios, y no podemos usar directamente el editor modal de la propiedad:

```
update Paises  
set Codigo = @Codigo, Pais = @Pais,  
FormatoCP = @FormatoCP, FormatoNIF = @FormatoNIF  
where IDPais = @IDPais and TS = @Original_TS;  
  
select TS  
from Paises  
where IDPais = @IDPais
```

Por una parte, he quitado de la cláusula **set** la asignación sobre la columna *Creacion*: no queremos que la fecha de creación de registro pueda ser alterada. Además, en la consulta posterior sobran la mayoría de las columnas, al menos mientras no modifiquemos esas columnas desde el *trigger* correspondiente. La única columna que necesitamos releer tras una modificación es *TS*, para averiguar el nuevo número de versión de la fila. Al realizar el cambio, Visual Studio reconstruirá la lista de parámetros, e intentará meterle el miedo en el cuerpo:



No le haga mucho caso: los nuevos parámetros suelen seguir estando en condiciones, aunque no es mala idea confirmarlo mediante un rápido examen de la propiedad *Parameters*. Para terminar con el

comando de actualización, cambie la propiedad *UpdateRowSource*, esta vez con el valor *FirstReturnedRecord*.

Finalmente, nos ocuparemos de las inserciones. Esta es la instrucción necesaria:

```
insert into Paises(Codigo, Pais, FormatoCP, FormatoNIF)
values (@Codigo, @Pais, @FormatoCP, @FormatoNIF);

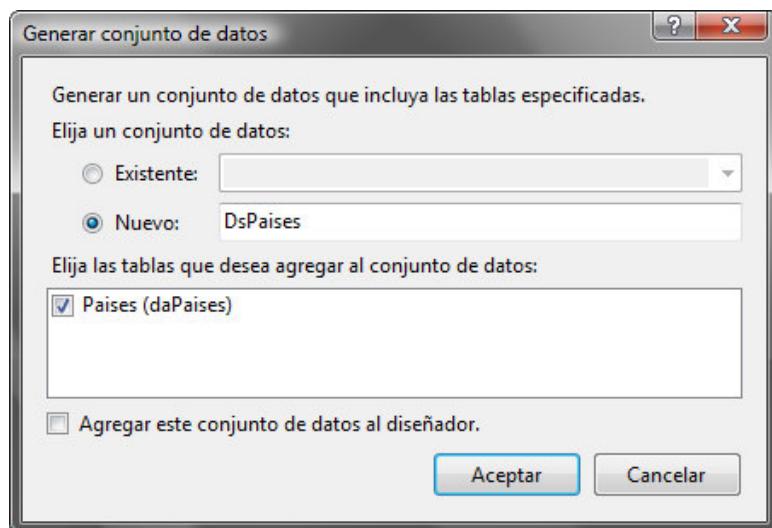
select IDPais, Creacion, TS
from Paises
where IDPais = scope_identity()
```

Hay varios cambios. Para empezar, no asignamos explícitamente el valor de la columna *Creacion*, para dejar que sea el servidor quien asigne la fecha... del servidor, no del cliente. En la consulta de relectura, dejamos sólo tres columnas: la clave primaria, porque tiene el atributo identidad, la fecha y hora de creación, porque se asigna en el servidor, y la versión de la fila, igual que antes. Cambie también el valor de *UpdateRowSource* a *FirstReturnedRecord*. Igual que antes, Visual Studio se quejará de falta de información sobre los parámetros, pero se trata de los típicos achaques de un hipocondríaco.

Observe el uso de la función *scope\_identity*. Esta función fue añadida en SQL Server 2000, y permite ignorar los valores de identidad asignados a otros registros insertados por *triggers*.

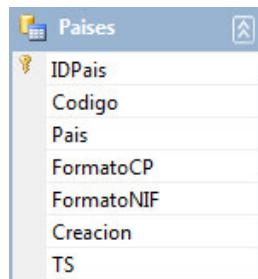
## Creación de los conjuntos de datos con tipos

- 321 El siguiente paso es crear un conjunto de datos con tipos para que almacene registros de la tabla de países. Cambie el nombre del adaptador para que sea *daPaises*. Selecciónelo y active su menú de contexto para ejecutar el comando *Generar conjunto de datos*:



Adoptaremos el convenio de nombrar las clases de conjuntos de datos con el prefijo *Ds*, con la primera letra en mayúsculas. De esta manera, cuando creamos instancias de la clase en tiempo de diseño, Visual Studio utilizará automáticamente el prefijo *ds*, en minúsculas, para las instancias. Además, al menos mientras el conjunto de datos con tipo contenga una sola tabla, utilizaremos su nombre para la clase. Si generamos un conjunto de datos para la tabla *Paises*, llamaremos *DsPaises* a la clase generada.

Note también que no vamos a añadir una instancia de *DsPaises* dentro de *Data*. Una vez cerrado el asistente, active el editor del conjunto de datos para modificar gráficamente la definición del conjunto de datos; para ello, busque el nodo titulado *DsPaises.xsd* en el Explorador de soluciones y haga doble clic sobre él. No hay mucho que modificar, pues se trata de una sola tabla. Cuando tengamos más de una tabla en un conjunto de datos, por el contrario, tendremos que configurar manualmente las posibles relaciones entre ellas:



Con la tabla de países sólo debemos modificar la configuración de la columna *Creacion*, porque queremos que reciba su valor en el servidor, y para ello debemos permitir que la columna abandone el lado cliente con un valor nulo. Seleccione la fila correspondiente a esta columna en el editor XSD, y asigne *True* en el Inspector de Propiedades, en la propiedad *Allownull* de la columna.

## Implementación del acceso para lectura

Ahora tenemos que modificar la implementación del método *Read*, proveniente de la interfaz *IData-Server* e implementado por la clase *Data*, para que reconozca la cadena *DsPaises* y reaccione leyendo la tabla de países:

```
public DataSet Read(string dsname)
{
    if (dsname == "DsPaises")
    {
        DataSet ds = new DataSet(dsname);
        daPaises.Fill(ds, "Paises");
        return ds;
    }
    else
        return null;
}
```

294

Observe que, a pesar de existir ya un conjunto de datos con tipos que permite almacenar países, estamos creando y devolviendo un conjunto de datos genérico. ¿Por qué? Bueno, más bien yo le preguntaría para qué queremos aquí un conjunto de datos con tipos. Esa categoría de clases las necesitaremos para el enlace de datos en el formulario de presentación. En la capa intermedia, por el contrario, los conjuntos de datos con tipos molestan y retrasan el ritmo de desarrollo.

¿No me cree? Piense un momento en qué sucedería en un proyecto grande, en el que tendríamos una biblioteca de clases con el servidor de capa intermedia y un ejecutable separado para la capa de presentación. Cada vez que el equipo que trabaja con el ejecutable tuviese necesidad de un conjunto de datos con tipos, nos obligaría a modificar el servidor de capa intermedia para que reconociese y pudiese crear instancias de esa clase. Recuerde, además, que el valor devuelto por *Read* se descarta después de que vaciemos su contenido sobre un verdadero conjunto de datos con tipos situado en el formulario de presentación.

Por otra parte, el código de grabación de países es engañosamente simple:

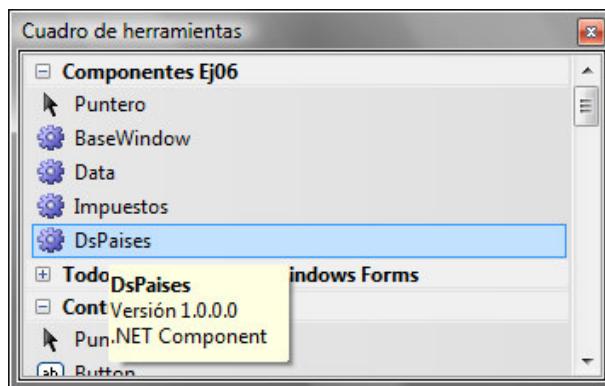
325

```
public DataSet Write(string dsname, DataSet delta)
{
    if (dsname == "DsPaises")
    {
        daPaises.Update(delta);
        return delta;
    }
    else
        return null;
}
```

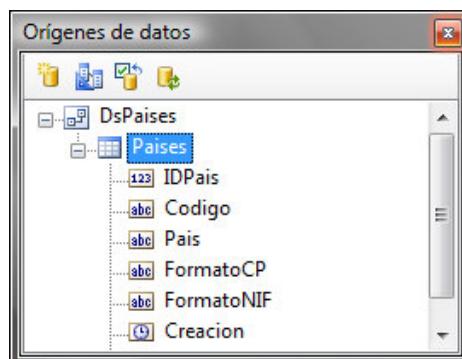
De momento, estamos ignorando la posibilidad de que intentemos grabar más de un registro y se produzcan errores después de una grabación parcial, además de no tomar medidas para manejar los errores de concurrencia. Hay un lugar y momento para cada cosa.

## Las ventanas de países e impuestos

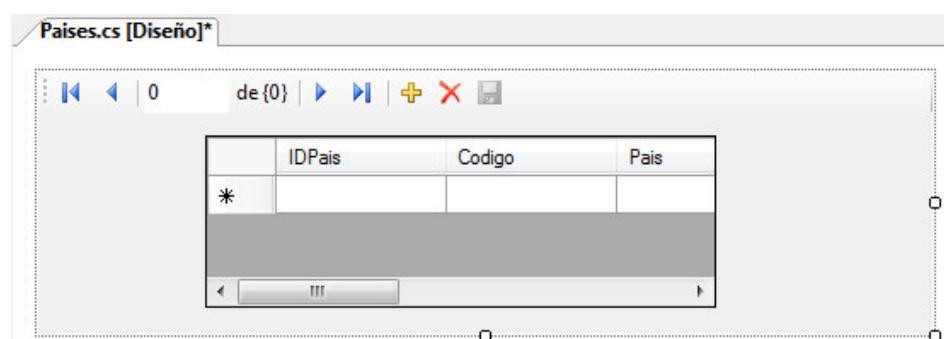
Ya estamos preparados para mostrar los registros de la tabla *Países* dentro de la correspondiente ventana. Primero, asegúrese de compilar correctamente el proyecto, y luego active el control heredado *Paises.cs*. Si ahora echa un vistazo al Cuadro de Herramientas, verá el nuevo conjunto de datos en una de las secciones de esta ventana:



Sin embargo, no vamos a crear directamente un conjunto de datos dentro de nuestro control de países. Active la ventana *Orígenes de datos*, que se encuentra disponible a través del menú *Datos* de la barra principal de menú de Visual Studio. Verá un árbol parecido al siguiente:



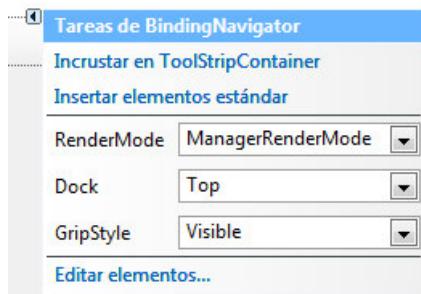
Tenemos que arrastrar el nodo que representa a la tabla *Países* sobre la superficie del control, y Visual Studio creará automáticamente un conjunto de datos, un componente *BindingSource*, una rejilla y una barra de navegación, y para rematar la faena, los conectará como mandan los cánones:



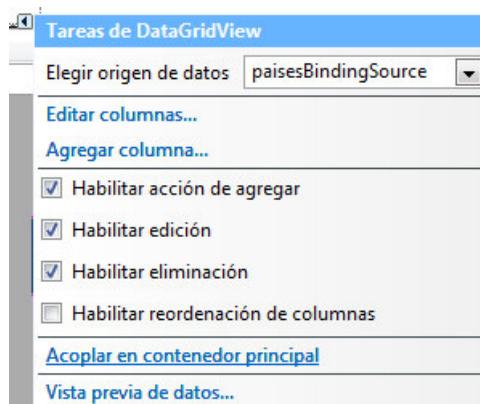
Quizás sea mucho pedir que Visual Studio coloque los dos controles visuales dentro de un contenedor que coordinase el tamaño y posición de los mismos. Si asignásemos *Fill* en la propiedad *Dock* de la rejilla, el área ocupada por ésta se solaparía con el área de la barra de navegación. Tenemos que ocuparnos manualmente de estos cambios:

- Si activa la lista de tareas rápidas de la barra de navegación, observará que ya viene con un comando titulado *Incrustar en ToolStripContainer*. Ejecútelo. Como resultado, aparecerá un control

del tipo mencionado en la superficie de diseño, y la barra de navegación se alojará automáticamente en el panel superior del mismo.



- 2** Tenemos que arrastrar la rejilla dentro del área central del *ToolStripContainer*. Una vez en su interior, active la lista de tareas de la rejilla, y ejecute el comando *Acoplar en contenedor principal*.



- 3** Finalmente, seleccione el propio *ToolStripContainer*, y ejecute su propio comando *Acoplar en control de usuario*.

Sólo queda asignar el conjunto de datos en la propiedad *DataSet*, que nuestro control de usuarios hereda de *BaseWindow*, y configurar la lista de columnas de la rejilla. Lo único que le recomiendo es que elimine la columna que muestra el contenido de *TS*, la columna que contiene el número de versión de la fila. Se trata de un valor que no dirá nada interesante al usuario. Y por favor: no se obsesione demasiado con obtener un resultado perfecto. Deténgase en cuánto tenga algo que se parezca a esto:

ID País	Código	País	C.P.	N.I.F.
1	DE	Alemania	^[0-9]{5}\$	^[0-9]{9}\$
2	AD	Andorra		
3	AR	Argentina	^[A-Za-z][0-9]{4}[A-Za-z]{3}^[0-9]{4}\$	^[0-9]{2}-?[0-9]{8}/?[0-9]\${
4	AU	Australia		
5	AT	Austria		^U[0-9]{3}\$
6	BB	Barbados		
7	BE	Bélgica		^[0-9]{9}\$
8	BO	Bolivia		
9	BR	Brasil		

#### EJERCICIO PROUESTO

Pruebe fuerzas implementando la ventana de navegación sobre impuestos; la implementación de esta ventana perteneciente al curso sólo se añadirá en el próximo ejercicio, para que no estorbe a la suya.

# EJERCICIO 07

## Objetivos

Recuperación de registros por páginas

## Técnicas introducidas

Generación dinámica de consultas

**L**A TÉCNICA QUE ESTAMOS USANDO PARA la lectura de conjuntos de datos es demasiado tosca y limitada para ser usada en la vida real. Debemos tener la posibilidad de indicar filtros, para limitar el conjunto de filas recibidas. En la mayoría de los casos, los filtros deberán ser de tipo arbitrario. Y en contadas ocasiones nos valdrá la solución de los libros de textos que consiste en preparar consultas con parámetros para uno o dos tipos posibles de filtros. Por último, necesitamos poder dividir el resultado de una consulta en páginas, para traer registros al lado cliente en paquetes de tamaño manejable.

## Una interfaz genérica para la recuperación por grupos

En este momento, no existe o más bien no conozco técnica alguna en las interfaces comerciales de acceso a datos que permite recuperar el resultado de una consulta por grupos de registros de igual tamaño. Si un usuario siente curiosidad por explorar una tabla con 500.000 registros, la única ayuda consiste en crear consultas especiales para dividir los registros en grupos. Si se tratase de la tabla de clientes, y quisieramos páginas de 50 registros, comenzaríamos con esta consulta:

```
select top 50 *
from Clientes
order by IDCliente
```

Si el mayor identificador retornado por la primera consulta fuese el 75, por ejemplo, la siguiente consulta sería como ésta:

```
select top 50 *
from Clientes
where IDCliente > 75
order by IDCliente
```

Y así sucesivamente. Para automatizar esta técnica y añadirla a nuestra aplicación tendríamos dos caminos:

- *Análisis y síntesis* (o *resíntesis*, siguiendo la terminología de la música electrónica): partiríamos de una consulta “normal” que no tendría en cuenta la división en grupos. Descompondríamos esa consulta en sus partes integrantes: cláusula **select**, cláusula **from**, filtro, subconsultas, cláusulas de ordenación... y luego recompondríamos la consulta, no sin antes inyectar el código necesario para la ordenación y limitación del número de registros. Esta técnica exige que realicemos un análisis sintáctico más o menos detallado, en tiempo de ejecución y puede ser potencialmente costosa.
  - *Síntesis* (desde el primer momento): más sensato es olvidarnos de una consulta inicial, y proveer al cliente con todo lo necesario para componer sus propias consultas con la partición en grupos incorporada desde el primer momento. Esta será la técnica que utilizaremos en el curso.
- 91 Añadiremos un método a *IDataServer* que nos permita especificar el tipo de consulta más general posible: le suministramos un nombre de tabla o vista, un filtro y un criterio de ordenación. Asumiremos siempre que se desean todos los campos de la tabla en la cláusula **select**: en caso contrario, siempre podríamos crear una vista con menos columnas y usarla en vez de la tabla. Lo mismo sucede con los encuentros naturales: si necesitásemos uno, podríamos sustituirlo mediante otra vista. E incluso tenemos un as debajo de la manga para casos extremos: en vez de una tabla o una vista, podemos usar una *tabla derivada* para usar dentro de la cláusula **from**.

Ahora bien, en vez de suministrar estos parámetros uno a uno, vamos a usar una técnica diferente: vamos a encapsularlos dentro una clase auxiliar, a la que llamaremos *PageCookie*. ¿Por qué *cookie*?

Supongo que conocerá la técnica de *cookies* que utilizan las aplicaciones en Internet para mantener la continuidad entre las peticiones de páginas. Nuestras *cookies* tendrán una función similar: serán el hilo conductor entre varias peticiones sobre una misma tabla. La primera vez que llamemos a la nueva versión de *Read*, pasaremos una *cookie* creada a partir de los datos antes mencionados. Al terminar la ejecución de *Read*, sin embargo, la *cookie* que recibiremos habrá cambiado de forma imperceptible: en realidad, *Read* nos devolverá una *cookie* perteneciente a otra clase, como veremos al analizar la implementación. Esta otra *cookie* contendrá la información necesaria para retomar la consulta a partir del último registro enviado en la petición anterior.

Pasemos a los detalles concretos. Esta es la interfaz *IDataServer* ampliada con el nuevo método:

```
public interface IDataServer
{
    DataSet Read(string dsname);
    DataSet Read(ref PageCookie cookie);
    DataSet Write(string dsname, DataSet delta);
}
```

Le advierto que podemos tener problemas para exportar esta interfaz a través de un servicio Web: una de las limitaciones del modelo de estos es que no se admite la sobrecarga de nombres. No obstante, veremos que la nueva versión de *Read* puede llegar a sustituir por completo la versión anterior. La primera versión de *Read* nos exige configurar de antemano un adaptador para permitir las lecturas, pero la técnica basada en *cookies* nos ahorrará esa parte del trabajo.

La clase *PageCookie* que utiliza *Read* se define de esta manera dentro del propio fichero *DataServer.cs*:

```
public class PageCookie
{
    public readonly string Table;
    public readonly string Filter;
    public readonly string Sort;
    public readonly int Count;

    public PageCookie(string table, string filter, string sort,
                      int count)
    {
        this.Table = table;
        this.Filter = filter;
        this.Sort = sort;
        this.Count = count;
    }

    :
}
```

De momento, sólo tenemos unos cuantos campos **readonly**, aunque en este caso son campos de instancias, y un constructor para inicializarlos. Reconozco que podíamos haber usado un diseño inicial más elegante, definiendo una clase abstracta o una interfaz común todas las clases de *cookies*. Pero he pensado que ese tipo de diseño complicaría aún más la explicación. Queda advertido...

## Generación dinámica de consultas

La *raison d'être* de nuestras *cookies* (no, no sé francés, pero no cuesta nada echarle un vistazo al diccionario) es que pueden “sintetizar” consultas SQL a partir de sus partes básicas. Para ello, montaremos un mecanismo articulado sobre dos elementos: una propiedad pública SQL, de tipo **string**, que es la que realmente nos interesa, y un método protegido virtual *GetFilter* que será usado por el algoritmo de generación de la propiedad SQL. Más adelante, declararemos clases más especializadas que reemplazarán la implementación original de *GetFilter*. Hay que teclear este código dentro de la clase *PageCookie*:

```
protected virtual string GetFilter()
{
    return Filter.Trim();
}
```

```
public string SQL
{
    get
    {
        StringBuilder sb = new StringBuilder();
        sb.Append("select top ");
        sb.Append(Count);
        sb.Append(" * from ");
        sb.Append(Table);
        string expression = GetFilter();
        if (expression != "")
        {
            sb.Append(" where ");
            sb.Append(expression);
        }
        sb.Append(" order by ");
        sb.Append(Sort);
        return sb.ToString();
    }
}
```

La clase *StringBuilder* nos permite evitar tener que sintetizar la cadena con la instrucción SQL por medio de concatenaciones sucesivas. Estas concatenaciones darían demasiado trabajo al sistema de asignación de memoria y al colector de basura. *StringBuilder*, sin embargo, funciona como una especie de *stream* en memoria. Al construir la cadena dentro de un *StringBuilder*, el número de asignaciones y reasignaciones de memoria disminuye drásticamente.

### Más registros, por favor...

Antes mencioné que *PageCookie* sólo se utiliza al llamar a *Read* por primera vez. Nuestra *cookie* llega al servidor, éste lee el primer grupo de registros... y en ese momento crea una instancia de otra clase que heredará de *PageCookie*. Gracias al polimorfismo, el código que ejecuta la llamada a *Read* no tiene por qué darse cuenta del cambazo.

La clase alternativa se llamará *ReturnedCookie*, y debe contener un fragmento adicional de información muy importante: cuál fue el último registro devuelto por la llamada anterior. Eso sí, en vez de almacenar los valores de las columnas necesarias para retomar la consulta, guardaremos la condición generada a partir de la comparación de esos mismos valores. Enseguida veremos los detalles.

De momento, aquí tiene la nueva clase:

```
// Clase: Ej07.Data.ReturnedCookie

protected sealed class ReturnedCookie : PageCookie
{
    private string startFrom;

    public ReturnedCookie(PageCookie source, string startFrom) :
        base(source.Table, source.Filter, source.Sort, source.Count)
    {
        this.startFrom = startFrom;
    }

    protected override string GetFilter()
    {
        string expression = base.GetFilter();
        if (startFrom == "")
            return expression;
        else if (expression != "")
            return "(" + expression + " and (" + startFrom + "))";
        else
            return startFrom;
    }
}
```

Hemos “sellado” la clase declarándola con el modificador **sealed**, que indica que no se puede heredar de esta nueva clase. Este modificador permite optimizar algunas llamadas a métodos virtuales. No veremos un cambio dramático en la velocidad, pero estoy seguro de que no voy nunca a necesitar un heredero de *ReturnedCookie*, y qué diantras, así practicamos un poco de C#.

Ahora le explicaré el papel que pinta la misteriosa condición *startFrom* en todo este asunto. Suponga que queremos todos los registros de la tabla de productos. Bueno, para ser exactos, vamos a recuperar productos en grupos de veinte. Creamos una *PageCookie* con Productos como tabla, una cadena vacía como filtro, un veinte en el parámetro *count* del constructor... y un criterio de ordenación en el que estamos obligados a incluir una clave única. Por ejemplo:

```
Producto, IDProducto
```

La consulta generada por esta *PageCookie* tendría la forma:

```
select top 20 *
from Productos
order by Producto, IDProducto
```

Para precisar, digamos que el último registro devuelto por esta consulta comienza así:

```
<20, 'Essential COM'...>
```

La consulta que necesitamos que genere la siguiente *cookie* debe ser como ésta:

```
select top 20 *
from Productos
where Producto > 'Essential COM' or
      Producto = 'Essential COM' and IDProducto > 21
order by Producto, IDProducto
```

La condición resaltada es la que necesitamos almacenar en *startFrom*, en la *cookie* que se enviará con la segunda llamada a *Read*. Hemos tenido un poco de suerte, porque la consulta original no tenía un filtro; en caso contrario, tendríamos que mezclar el filtro original con *startFrom*, que es precisamente lo que hace la versión redefinida de *GetFilter* de la clase *ReturnedCookie*. Pero tome nota de cómo se complica la comparación por la existencia de dos columnas en el criterio de ordenación. Y recuerde que nos vimos obligados a añadir *IDProducto* al criterio para tener una combinación de columnas con valores combinados únicos, algo fundamental para que nuestra técnica funcione.

¿Cómo se genera la condición *startFrom*? La implementación de *Read* se ocupará de ello, siempre que echemos una mano como es el caso de este método auxiliar, que convierte un valor arbitrario, de tipo **object**, en una expresión constante aceptable por SQL Server:

```
protected string SqlValue(object v, DataColumn col)
{
    if (col.DataType == typeof(System.String))
        return "'" + v.ToString().Replace("'", "''") + "'";
    else if (col.DataType == typeof(System.Guid))
        return "'" + v.ToString() + "'";
    else if (col.DataType == typeof(System.Decimal))
        return ((Decimal)v).ToString(
            System.Globalization.CultureInfo.InvariantCulture);
    else if (col.DataType == typeof(System.DateTime))
        // Standard ODBC format for date & times
        return "CONVERT(DATETIME, '" +
            ((DateTime)v).ToString("yyyy-MM-dd hh:mm:ss:fff") + "', 121)";
    else
        return v.ToString();
}
```

Creo que la mayor parte del código se explica por sí solo. Normalmente, basta con aplicar el método *ToString* sobre el objeto recibido para obtener una constante SQL decente. Por ejemplo, si el objeto encapsula un valor entero, *ToString* devolverá la cadena correspondiente al entero... eso sí, sin las comillas. Si el tipo de la columna es una cadena de caracteres o un campo **uniqueidentifier**, por el contrario, hay que añadir las comillas, y en el caso de la cadena, hay que reemplazar las comi-

llas simples aisladas por pares de comillas simples, como manda SQL. Los casos más extraños son estos dos:

- Si se trata de una columna de tipo real o numérico, con valores fraccionarios, *ToString* utilizará la “cultura” asociada al hilo, y si el hilo está en español, la cadena generada tendrá una coma decimal. Ahora bien, SQL Server exige siempre un punto decimal. Para resolverlo, realizamos la llamada a *ToString* indicando que queremos una traducción al inglés americano del valor real.
- Más extraño aún puede parecer el tratamiento de los valores de fecha y hora. ¿Se atreve usted a apostar por el formato correcto de una constante de cadenas en SQL Server? ¿Qué debe ir primero, el mes o el día? Por suerte para nosotros, SQL Server ofrece una vía de escape: podemos transformar el valor en una cadena de caracteres, para aplicarle a continuación la función **convert**, de SQL Server. Esta función nos permite indicar en qué formato hemos escrito la cadena. El formato que se utiliza en *SqlValue* es el formato definido por ODBC para escribir constantes de cadenas.

¿Qué tal un ejemplo de esto último? Si pasamos una fecha a *SqlValue* correspondiente al 21 de septiembre de 2004, a las 3:00 de la madrugada, el método devolverá la siguiente expresión:

```
convert(datetime, '2004-09-21 03:00:00.000', 121)
```

Le prometo que SQL Server sabrá de qué estamos hablando. Con la ayuda de *SqlValue*, podemos añadir ahora otro método auxiliar a la clase *ReturnedCookie*:

```
protected string SynthesizeExpression(DataTable t, string columns)
{
    DataRow row = t.Rows[t.Rows.Count - 1];
    StringBuilder sb = new StringBuilder();
    StringBuilder acc = new StringBuilder();
    foreach (string cname in columns.Split(','))
    {
        if (acc.Length > 0)
        {
            sb.Append(" or ");
            sb.Append(acc.ToString());
            sb.Append(" and ");
        }
        DataColumn column = t.Columns[cname.Trim()];
        string val = SqlValue(row[column], column);
        sb.Append(column.ColumnName);
        sb.Append(" > ");
        sb.Append(val);
        if (acc.Length > 0)
            acc.Append(" and ");
        acc.Append(column.ColumnName);
        acc.Append(" = ");
        acc.Append(val);
    }
    return sb.ToString();
}
```

Este método recibe una tabla con las filas devueltas por una consulta, y aísla el último registro de la tabla. El algoritmo que sigue a continuación es algo embrollado, pero debe ejecutarlo paso a paso si quiere comprenderlo. La complejidad es culpa de la expresión de comparación necesaria cuando hay más de una columna en el criterio de ordenación. En términos generales, si hay  $n$  columnas en el criterio de ordenación, la condición a generar debe contener  $n(n-1)$  comparaciones. Preste también atención a la técnica usada para descomponer el criterio de búsqueda en las columnas que lo integran: hemos usado el método *Split* de la clase *System.String*.

Finalmente, ya estamos en condiciones de implementar la nueva versión del método *Read*. Nuestro método comenzará creando un adaptador de datos al vuelo, utilizando la conexión ya existente en nuestro módulo de datos y la consulta SQL generada por la *cookie* recibida. Observe esta llamada

puede ejecutarse tanto con una *PageCookie* como con una *ReturnedCookie*. En ambos casos, el polimorfismo nos garantiza que la propiedad SQL nos devolverá la consulta apropiada:

```
public DataSet Read(ref PageCookie cookie)
{
    using (SqlDataAdapter da = new SqlDataAdapter(cookie.SQL, sqlConn)) {
        da.TableMappings.Add("Table", cookie.Table);
        DataSet ds = new DataSet();
        da.Fill(ds);
        if (ds.Tables[0].Rows.Count > 0)
            cookie = new ReturnedCookie(cookie,
                SynthesizeExpression(ds.Tables[0], cookie.Sort));
        return ds;
    }
}
```

Añadimos entonces un elemento a la colección *TableMappings* del adaptador. Como puede ver, el método *Fill* se llama sobre un conjunto de datos genérico, y eso quiere decir que este método definirá una nueva tabla con su correspondiente esquema. La modificación de *TableMappings* nos sirve para que la nueva tabla reciba el verdadero nombre que le corresponde, en vez del nombre genérico *Table* que genera por omisión el adaptador.

**NOTA**

En realidad, el nombre de tabla que utiliza por omisión el método *Fill* se extrae de una constante pública de la clase *DbDataAdapter* que se llama *DefaultTableName*. Si le pone nervioso usar directamente la constante *Table* dentro del método anterior, puede sustituirla por esta otra constante simbólica.

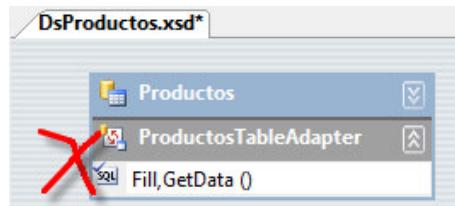
Como era previsible, a continuación *Read* ejecuta el método *Fill*, y si hay filas en la tabla generada a partir de la consulta, se devuelve una nueva *cookie* del tipo *ReturnedCookie*, creada con la ayuda del método *SynthesizeExpression* que hemos visto antes. Más adelante añadiremos un mecanismo para que la *cookie* informe al cliente cuando lleguemos al final de la lista de registros de la tabla consultada.

## Inicialización

Ahora nos toca diseñar la ventana de productos. La primera dificultad que encontramos es que, al no existir un adaptador para la tabla de productos, no podremos crear el conjunto de datos con tipos en la forma habitual, partiendo del adaptador. Por supuesto, existen alternativas. Ejecute primero el comando de menú *Proyecto | Agregar nuevo elemento*, seleccione el ícono *DataSet* y proponga *DsProductos* como nombre para el nuevo fichero:



En respuesta, Visual Studio mostrará un editor vacío de conjuntos de datos. Active el *Explorador de servidores*, localice la tabla de productos y arrástrela sobre la superficie de diseño. Basta con esta operación para que dispongamos de un flamante conjunto de datos con tipos para productos:



Observe, sin embargo, que Visual Studio ha creado también un adaptador de datos, acoplado a la parte inferior del conjunto de datos: elimínelo. No muestre piedad alguna, que no la merece. Compile el proyecto, para que se actualice la ventana Orígenes de datos, cree un control de usuario *Productos*, a partir de *BaseWindow*, y arrastre el “origen de datos” correspondiente sobre la superficie de *Productos*. No olvide asignar el valor de la propiedad *DataSet* que *Productos* hereda de *BaseWindow*,

y retoque un poco el aspecto del control de usuarios. Esconda, por ejemplo, las columnas de la rejilla que corresponden a las columnas *Imagen* y *TS* de la tabla de productos.

El siguiente problema se nos presentará al intentar inicializar el conjunto de datos. Recuerde que nuestro gestor de ventanas intenta la carga automática de datos cuando hay algo asignado en dicha propiedad. La solución es reescribir el método *InitData*, también heredado de *BaseWindow*:

```
private PageCookie cookie = null;

public override void InitData()
{
    // ¡No estamos llamando al método heredado!
    cookie = new PageCookie("Productos", "", "Producto", IDProducto", 20);
    dsProductos.Merge(Data.Instance.Read(ref cookie), false);
}
```

Observe que no llamamos al método heredado de la clase base, porque es ese método el que realizaría la inicialización automática. Nuestra implementación recupera la primera página de veinte registros de la tabla de productos. Como criterio de ordenación, usamos primero el nombre del producto, y añadimos la clave del producto para lograr una clave única; si el nombre del producto fuese único, no necesitaríamos esa segunda columna.

Sólo nos queda traer las sucesivas páginas de registros, para lo cual deberíamos añadir botones o comandos de menú. Para no complicar un ejercicio ya bastante complicado, vamos a conformarnos de momento con traer la página siguiente mediante un botón que añadiremos a la barra de navegación de productos. Este botón responderá a las pulsaciones de la siguiente manera:

```
private void bnMasRegistros_Click(object sender, EventArgs e)
{
    dsProductos.Merge(Data.Instance.Read(ref cookie), false);
}
```



El código fuente del ejercicio es un poco más sofisticado, porque comprueba que realmente existen más registros antes de intentar la operación. He preferido ahorrarle estos detalles nimios, pero puede consultarlos analizando el código fuente.

**NOTA**

Hay otra simplificación importante: la tabla de productos tiene una columna *IDImpuesto*, y la estamos mostrando directamente, aunque deberíamos mostrar más bien la descripción del impuesto. Lo aconsejable sería incluir una copia de la tabla de impuestos en el conjunto de datos de productos, para añadir una columna calculada en la tabla de productos que mostrase el nombre del impuesto (o la tasa asociada) en vez del poco informativo indicador. Tenga presente que todos los registros de esta segunda tabla deberían leerse junto con la primera página de productos. En la práctica, es probable que tablas pequeñas, como la de tipos de impuestos, se almacenen en una caché local al cliente. Déjaremos los detalles para más adelante.

Esta vez tenemos que configurar unas cuantas columnas en la rejilla para obtener un resultado aceptable.

- Para las columnas que representan dinero, hay que editar la propiedad *Format*, que se encuentra dentro de la propiedad *DefaultCellStyle* de cada columna. El valor apropiado es la cadena *C2*, esto es, *currency* con dos decimales.
- Para las columnas que representan valores numéricos, suele modificarse *Alignment*, que también es una propiedad de *DefaultCellStyle*, para que valga *MiddleRight* y muestre el contenido de las correspondientes celdas alineado a la derecha.

## Calma y sangre fría

Hay muchas líneas en este ejercicio. Como estamos utilizando un entorno de desarrollo del tipo llamado RAD (*rapid application development*), es natural que nos preguntemos si hemos elegido el buen camino. ¿Qué alternativas teníamos? Una de ellas sería preparar un adaptador con una consulta paramétrica. Por ejemplo, para leer productos ordenados por su identificador en grupos de veinte registros, podríamos usar esta consulta:

```
select top 20 *
from Productos
where IDProducto > @IDProducto
order by IDProducto
```

Para obtener el primer grupo, asignaríamos un cero al parámetro, y en las sucesivas llamadas utilizaríamos el identificador mayor recuperado hasta ese momento. Tendríamos, en cambio, los siguientes problemas:

- 1 El número de registros por grupo está escrito a sangre y fuego en la consulta. Esto no será tan grave en SQL Server 2005, que permitirá parametrizar la cláusula **top**.
- 2 Lo más importante: ¿cómo aplicamos un filtro a esta consulta? La experiencia nos dice que casi nunca consultaremos todos los registros de una tabla, sin filtros.

Otra forma de saber si vamos por buen o mal camino es comparar con el trabajo necesario para implementar esta funcionalidad en otros entornos de programación. El autor ha pasado muchos años programando en Delphi. Aunque DataSnap, el sistema de desarrollo en múltiples capas de Delphi, ofrece una técnica muy sencilla para la recuperación por grupos, en la práctica ésta es poco útil, porque exige una relación *statefull* entre el cliente y el servidor de datos. La técnica *stateless* equivalente tiene una complejidad similar.

En cualquier caso, el éxito o el fracaso de esta técnica deben medirse por su facilidad de uso. Y creo que en ese apartado no tendrá quejas.

# EJERCICIO 08

## Objetivos

Edición modal

## Técnicas introducidas

Ejecución modal, la clase *DataRowView*

**H**ACE UN PAR DE EJERCICIOS, PONTIFICABA contra la edición directa dentro de la rejilla usada para la búsqueda y navegación. Hemos tenido el cuidado, no obstante, de comprobar la presencia de cambios al cerrar una ventana de navegación no modal, por si el programador olvidaba marcar la rejilla como no modificable. Ha llegado el momento de añadir el mecanismo apropiado para realizar cambios en los registros recuperados durante la navegación.

## Diálogos modales y operaciones sobre datos

- 325 La principal dificultad que plantean nuestras interfaces desconectadas de acceso a datos es saber cuándo hemos terminado una edición. Todo cambio en un registro es necesariamente un cambio en una copia local, que en algún momento habrá que reflejar en el origen de los datos. La pregunta es, precisamente: ¿cuándo?

Veamos primero una solución extrema: cuando lo diga el propio usuario. Debemos hacer saber a éste que está trabajando con una mera copia, y que los cambios que realice sobre ella no se aplicarán sobre la base de datos mientras no pulse el botón de guardar los cambios. Eso es lo que sucede en un procesador de texto, como el que estoy usando en estos precisos momentos. Las desventajas que presenta esta metáfora del procesador de texto:

- 1 Demasiada responsabilidad sobre los débiles hombros del usuario.
- 2 Mientras más se distancien el momento de la modificación local y el momento de la sincronización con la base de datos, surgirán más conflictos debido a modificaciones concurrentes. Es decir, más problemas que tendrá que enfrentar el usuario.
- 3 El intervalo que media entre modificación local y final puede también entrar en conflicto con el intervalo definido de forma natural por ciertas operaciones de carácter transaccional.

Pasemos al extremo: que el sistema determine automáticamente cuando el usuario ha terminado sus modificaciones. Este extremo es aún peor:

- 1 Muchas aplicaciones mal diseñadas consideran que ese momento llega cuando el usuario cambia la fila activa. El problema es que el usuario puede quedarse sobre una fila por los siglos de los siglos.
- 2 Una técnica ligeramente mejor es esperar a que el usuario cambie de celda activa, pulse la tecla INTRO, o transcurra un tiempo determinado sin realizar cambios. El problema está en que no toda actualización individual sobre una columna genera un registro con un estado aceptable. Además, esto puede provocar tráfico innecesario.
- 3 No descarto que, para algunos tipos muy sencillos de entidades y sistemas, se pueda llegar a un compromiso más o menos sensato. De todos modos, a mí como usuario, este tipo de decisiones tomadas por el sistema me harían desconfiar. Soy un desconfiado, es cierto.

Mi solución favorita está más cerca del primer extremo que del segundo:

- 1 El usuario debe decidir: "ya, ya puedes grabar mis modificaciones".
- 2 Como detesto mezclar navegación con edición, prefiero que los cambios se realicen sobre una segunda ventana. La mejor forma de presentar los datos para navegación no es la mejor forma de presentarlos para su edición.
- 3 Para reducir el intervalo entre modificación local y grabación en la base de datos, es aconsejable que la ventana de edición sea un diálogo modal.

Con esto no quiero decir que haya siempre que recurrir a diálogos modales. En ciertos tipos de sistemas, y estoy pensando en aplicaciones para centros de recepción de llamadas telefónicas, es más

interesante que la edición de un registro no bloquee la navegación y búsqueda de otros registros, y es mejor editar con diálogos no modales. Imagine la siguiente escena:

- Llama un señor para solicitar un préstamo.
- La telefonista busca el registro de cliente y crea un registro de solicitud, o simplemente abre el diálogo que le permite ver todos los detalles del cliente y modificarlos.
- Para aprobar la cantidad solicitada, hace falta un avalista. El buen señor propone a su esposa como tal.
- La telefonista realiza una segunda búsqueda sobre su listado de clientes, algo que sería imposible si el diálogo anterior hubiese bloqueado la ejecución.

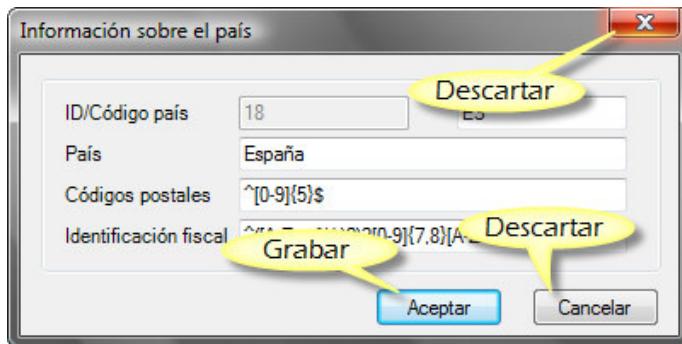
No debemos olvidar que las ventanas no modales permiten mayor libertad a los usuarios. En las primeras aplicaciones para Windows, creadas por programadores procedentes de UNIX y MS DOS, se abusaba de la navegación modal, y esto generaba aplicaciones más fáciles de programar pero inmanejables desde el punto de vista del usuario.

**NOTA**

Hay un pecado de diseño relacionado: considerar que se resuelve algún problema sustituyendo la navegación mediante rejillas por la navegación registro a registro. No soy partidario de este tipo de restricciones: las rejillas son muy útiles para mostrar con una simple ojeada cuántos registros cumplen con determinada condición, o la relación que existe entre filas ordenadas de acuerdo a algún criterio. La ordenación, por ejemplo, es una forma muy sencilla de detección de valores duplicados.

## El momento más adecuado

Una vez que elegimos diálogos modales para realizar cambios, tenemos que enfrentar un problema menor, de carácter técnico. Sabemos que la grabación debe tener lugar al cerrarse el cuadro de diálogo, pero ¿exactamente en qué momento? Un error frecuente consiste en interceptar el evento *Click* de los dos típicos botones de *Aceptar* y *Cancelar*:



¿Y qué sucede cuando el usuario pulsa las teclas CTRL+F4, o cierra el diálogo mediante el botón de la barra de títulos del diálogo? Hay que contar, además, con la posibilidad de que se produzcan errores durante la grabación. Si asociamos la grabación al evento *Click* de un botón, se hace más difícil impedir el cierre de la ventana en caso de error. Mi consejo es cumplir la siguiente regla al pie de la letra:

*"La grabación de cambios asociada a un cuadro de diálogo modal debe tener lugar dentro de un manejador del evento Closing"*

251

Este evento se dispara justo antes de que la ventana inicie su protocolo de cierre, y nos permite abortar la operación. El evento *Closing* se dispara siempre, sin importar la forma en que el usuario haya iniciado el cierre. En cambio, cuando se dispara el evento relacionado *Closed*, es ya demasiado tarde para impedir el cierre si se produjese una excepción.

## El gestor de ventanas asume responsabilidades

Hablando con propiedad, no sería obligatorio que la ejecución de los diálogos modales de edición fuese responsabilidad del gestor de ventanas que ya se ocupa de las ventanas no modales. Y a pri-

mera vista es difícil encontrar una justificación. Pero un poco de análisis le convencerá de lo contrario: dígame, por ejemplo, ¿qué es para usted un diálogo modal? Se me ocurren modelos de aplicación en las que la forma más natural de edición modal no es utilizar un diálogo de los de toda la vida. Por ejemplo, últimamente he estado experimentando con interfaces de usuarios que imitan el funcionamiento de HTML. En un sistema de este tipo, un diálogo modal sería simplemente otra página más, con un tratamiento del historial de navegación ligeramente distinto.

Por este motivo, vamos a añadir un nuevo método, al que llamaremos *Execute*, a la interfaz que describe la funcionalidad del gestor de ventanas:

```
public interface ILayoutManager
{
    IWindow Create(Type type);
    IWindow Activate(Type type);
    DialogResult Execute(Type type, Form owner);

    event EventHandler CreateWindow;
}
```

Al igual que el método *Create*, *Execute* recibe el tipo del nuevo cuadro de diálogo, pero también recibe un parámetro adicional: un puntero al formulario que lanza el cuadro de edición. Nuestras ventanas de búsqueda y navegación son lanzadas directamente desde el menú, pero salvo excepciones, los diálogos de edición serán lanzados desde las ventanas de navegación. Este puntero o referencia es muy importante, porque permitirá que el contenido del diálogo se ajuste a la fila activa en la ventana que lo lanza.

¿Y cómo accederemos a esa fila activa? Podríamos añadir los métodos necesarios a la ya existente interfaz *IWindow*, pero esto sería un poco chapucero. Lo que haremos es definir un nuevo tipo de interfaz, al que llamaremos *IDataWindow*, que deberá ser implementada por cualquier clase que quiera lanzar un cuadro de edición modal. Esta es la declaración de *IDataWindow*:

```
public interface IDataWindow
{
    // Acceso a datos
    DataSet DataSet { get; }
    BindingSource BindingSource { get; }
}
```

- 218 Observe que el implementador de *IDataWindow* debe indicarnos cuál es el conjunto de datos que se supone que debemos editar, y cuál de sus posibles tablas, en concreto, es la tabla maestra. Pero la clase *DataSet* no incluye información sobre un registro activo. Por eso, en vez de solicitar directamente la tabla, pediremos a *IDataWindow* que nos diga cuál es el componente *BindingSource* principal de la ventana. A partir de esa información deduciremos cuál es la tabla editada y la fila seleccionada en el momento de ejecutar el cuadro de diálogo.

Es muy sencillo hacer que *BaseWindow* implemente el nuevo tipo de interfaz... porque ya hemos adelantado gran parte del trabajo:

```
#region Miembros de IDataWindow

private DataSet dataset = null;
private BindingSource bindingSource = null;

[DefaultValue(null)]
[Description("Conjunto de datos principal de la ventana")]
[Browsable(true)]
public System.Data.DataSet DataSet
{
    get { return dataset; }
    set { dataset = value; }
}

[DefaultValue("")]
[Description("Origen de enlace principal de la ventana")]
[Browsable(true)]
```

```

public BindingSource BindingSource
{
    get { return datamember; }
    set { datamember = value; }
}

#endregion

```

Ya estamos en condiciones de explicar la implementación de *Execute* por parte de *MdiLayoutManager*:

```

public DialogResult Execute(Type type, Form owner)
{
    using (Form f = type.InvokeMember(null,
        BindingFlags.DeclaredOnly |
        BindingFlags.Public | BindingFlags.NonPublic |
        BindingFlags.Instance | BindingFlags.CreateInstance,
        null, null, null) as Form)
    {
        IWindow wnd = f as IWindow;
        if (wnd != null)
        {
            wnd.InitData(owner.Controls[0] as IDataWindow);
            f.Closing += new CancelEventHandler(ModalFormClosing);
        }
        return f.ShowDialog(owner);
    }
}

```

La primera parte del método debe resultar familiar: a partir del método, usamos reflexión para ejecutar indirectamente un constructor sin parámetros. A continuación, buscamos el primer control de la ventana pasada como propietaria: recuerde que nuestras “ventanas” de navegación se definen en realidad como controles de usuarios, y se ubican dentro de formularios tradicionales. Una vez localizado el control que actúa como ventana de navegación, extraemos de él su implementación de la nueva interfaz *IDataWindow* y la pasamos como parámetro al método *InitData* del diálogo recién creado. Esto significa que tenemos que modificar también el prototipo del método *InitData* dentro de la interfaz *IWindow*:

```
void InitData(IDataWindow parent);
```

¿Hay algo más natural que, al inicializar una ventana, informarle cuál ventana la ha invocado? Por supuesto, cuando inicialicemos una ventana de navegación, este parámetro será necesariamente nulo, y como tal lo ignoraremos.

Antes de ejecutar el diálogo con *ShowModal*, el gestor de ventanas le añade un manejador para el evento *Closing*, de forma parecida a como se hace con las ventanas de navegación. Esta vez la implementación es algo diferente:

```

private void ModalFormClosing(
    object sender, System.ComponentModel.CancelEventArgs e)
{
    IWindow wnd = sender as IWindow;
    if (wnd.IsModified)
        if ((sender as Form).DialogResult == DialogResult.OK)
        {
            if (!wnd.SaveChanges())
                e.Cancel = true;
        }
    else if (MessageBox.Show(
        "Ha realizado cambios en el registro.\n" +
        "¿Desea realmente abandonarlos?", "Advertencia",
        MessageBoxButtons.YesNo,
        MessageBoxIcon.Question) == DialogResult.Yes)
        wnd.DiscardChanges();
    else
        e.Cancel = true;
}

```

¡Volvemos a aprovechar la funcionalidad de la interfaz *IWindow*, que suponemos que el diálogo deberá implementar! Recuerde:

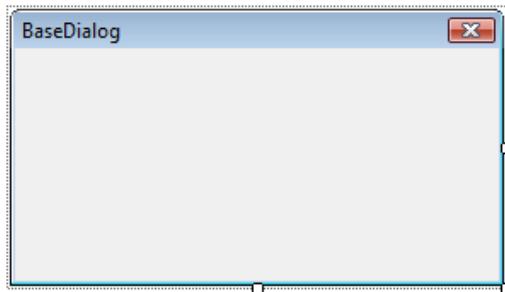
Tipo de ventana	Interfaces a implementar
Ventana de navegación	<i>IWindow</i> , <i>IDataWindow</i>
Diálogo de edición modal	<i>IWindow</i>

No obstante, más adelante veremos que algunos diálogos deberán también implementar *IDataWindow* para mostrar a su vez otros diálogos de edición anidados.

## Un noble linaje de editores modales

Cuando se trata de ventanas no modales, nos vale cualquier control de usuario, siempre que implemente *IWindow*. Una precisión: a partir de ahora, las que quieran lanzar diálogos modales de edición tendrán también que implementar *IDataWindow*. No obstante, para ahorrar trabajo al desarrollador, tenemos una clase *BaseWindow* que sirve de base para la mayoría de las ventanas de navegación. Ponga dobles comillas en todas las apariciones de la palabra “ventana” en este párrafo.

Algo parecido haremos con los cuadros de diálogos: en vez de obligar al programador a partir de cero, vamos a definir un *BaseDialog* que nos ahorre parte del trabajo. Observe que los diálogos sí serán formularios: no vamos a repetir el truco del control de usuario. Añada un nuevo formulario al proyecto y cambie su nombre a *BaseDialog*:



Aunque *BaseDialog* no implementará *IDataWindow*, nos interesa tener una propiedad *BindingSource* en esta familia de ventanas, que nos indicará cuál es el componente *BindingSource* principal del diálogo. En muchos sentidos, la función de *BindingSource* en *BaseDialog* será similar a la de la propiedad *DataSet* en *BaseWindow*:

```
// Código en BaseDialog.cs:
using System;
using System.ComponentModel;
using System.Windows.Forms;

public class BaseDialog : IWindow, IDataWindow
{
    [DefaultValue(null)]
    [Description("Origen de enlace principal de la ventana")]
    [Browsable(true)]
    public BindingSource BindingSource
    {
        get { return bindingSource; }
        set { bindingSource = value; }
    }
}
```

En realidad, podemos deducir cuál es el conjunto de datos y la tabla principal a partir de esta propiedad. Utilizaremos propiedades para tener acceso a estos valores:

```
// Código en BaseDialog.cs:
using System;
using System.ComponentModel;
using System.Windows.Forms;

public class BaseDialog : IWindow, IDataWindow
{
    [Browsable(false)]
    private DataSet DataSet
    {
        get { return (DataSet)bindingSource.DataSource; }
    }
}
```

```
[Browsable(false)]
private DataTable DataTable
{
    get { return ((DataGridView)bindingSource.List).Table; }
}
```

Es fácil deducir que el conjunto de datos es el mismo objeto enganchado a la propiedad *DataSource* del *BindingSource*; sólo tenemos que realizar la conversión de tipos. Es más complicado el razonamiento utilizado para encontrar la tabla. Para ser sinceros, no se trata de *razonamiento*, sino de *conocimiento*: cuando el enlace a datos se produce sobre un conjunto de datos, internamente se utiliza un componente *DataView* para acceder a los registros. La propiedad *List* del *BindingSource* es la que nos permite el acceso a la vista de datos, y a partir de ella podemos hallar la tabla.

Llegados a este punto, debemos diferenciar entre dos posibles técnicas para coordinar el contenido del diálogo con el de la ventana base. Las dos alternativas son:

- 1 El diálogo utiliza una vista sobre el mismo conjunto de datos utilizado en la ventana de navegación. Esa vista, a su vez, puede ser la misma vista de navegación o una segunda vista.
- 2 El diálogo utiliza un segundo conjunto de datos, que puede tener exactamente el mismo esquema o no que el conjunto de datos de la ventana de navegación. Los datos que se editan en el diálogo son, en consecuencia, una copia de la columna activa de la ventana de navegación.

Las dos técnicas son correctas, pero vamos a usar la segunda. En primer lugar, existirán ocasiones en los que necesitaremos más cantidad de detalles en la ventana de edición. Por ejemplo, podemos estar buscando y navegando cabeceras de pedidos, pero al hacer doble clic sobre una de ellas, querremos traer desde la base de datos todas las líneas de detalles y los datos adicionales de la factura. En segundo lugar, si usamos un mismo conjunto de datos para navegar y editar, tendremos que realizar el enlace a datos dentro del diálogo de forma manual. No se complicaría excesivamente el código, pero es preferible que los ejemplos eviten la complejidad innecesaria.

Esta será la implementación de *InitData* dentro de la clase *BaseDialog*:

```
protected IDataWindow parentWindow = null;

public virtual void InitData(IDataWindow parent)
{
    // Comprobamos si bindingSource está asignado,
    // y si su DataSource apunta a un DataSet.
    System.Diagnostics.Debug.Assert(
        bindingSource != null &&
        bindingSource.DataSource != null &&
        bindingSource.DataSource is DataSet);
    parentWindow = parent;
    DataTable.Rows.Add(
        (DataRowView)parent.BindingSource.Current).Row.ItemArray;
    DataSet.AcceptChanges();
}
```

Guardamos el puntero a la ventana nodriza, localizamos en ésta la fila activa, y creamos una copia en el conjunto de datos local que supondremos enlazado a *BaseView*. La propiedad *ItemArray* de una fila de datos devuelve un vector de objetos con los valores de todas las columnas de la fila dada. Como queremos que el diálogo crea que la fila añadida “ya estaba ahí”, ejecutamos *AcceptChanges* para quitar las marcas de modificaciones del conjunto de datos.

Para saber si hemos realizado algún cambio sobre la fila activa dentro del diálogo, podemos usar la propiedad *IsModified*:

```
[Browsable(false)]
public virtual bool IsModified
{
    get
    {
        bindingSource.EndEdit();
```

```

        return DataSet.HasChanges();
    }
}

```

Si no llamamos a *EndEdit*, los cambios propuestos sobre la fila no pasarían al conjunto de datos, y el método *HasChanges* de éste siempre devolvería **false**.

Espero que el código de *SaveChanges* y *DiscardChanges* le resulte familiar:

```

public bool SaveChanges()
{
    System.Data.DataSet delta = Data.Instance.Write(
        DataSet.DataSetName, DataSet.GetChanges());
    foreach (DataTable tab in DataSet.Tables)
        foreach (DataRow row in tab.Select(
            null, null, DataViewRowState.Added))
            row.Delete();
    DataSet.Merge(delta, false);
    DataSet.AcceptChanges();
    parentWindow.DataSet.Merge(DataTable);
    return true;
}

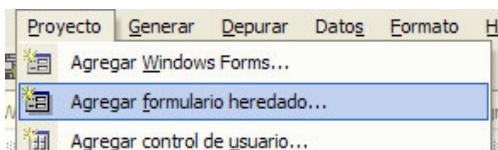
public void DiscardChanges()
{
    DataSet.RejectChanges();
}

```

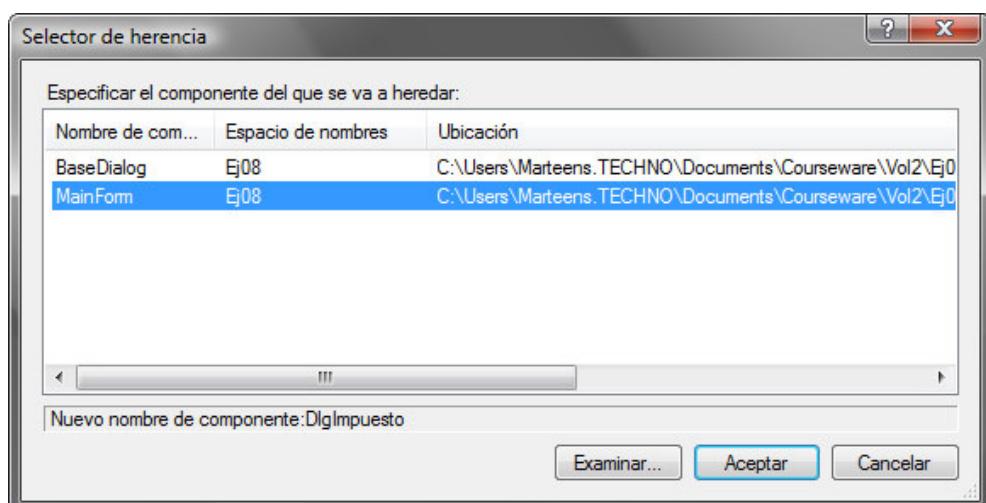
La única diferencia importante está en la línea resaltada. Antes de dar por terminada la ejecución del diálogo, debemos copiar los cambios en el conjunto de datos original ubicado en la ventana de navegación. Para lograrlo, volvemos a echar mano del más que útil método *Merge*.

## Enlaces con la vista de datos

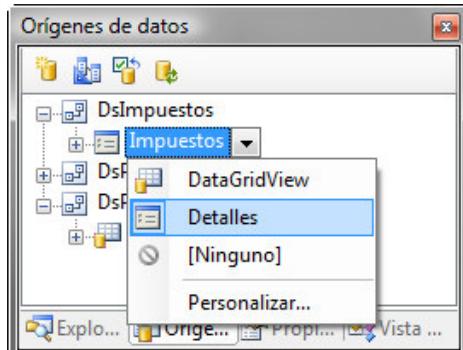
Toca ver cómo aprovechar la infraestructura creada, y vamos a poner como ejemplo el diálogo de edición de impuestos. Comenzamos añadiendo un “formulario heredado” al proyecto:



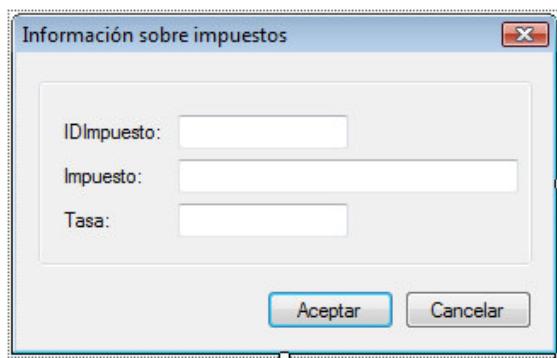
Llamaremos *DlgImpuesto* al nuevo formulario, para identificarlo claramente como diálogo modal. Una vez indicado el nombre, Visual Studio nos preguntará cuál será la clase base del nuevo diálogo. Evidentemente, seleccionaremos *BaseDialog*:



Para configurar el interior del diálogo de impuestos, arrastraremos un “origen de datos” desde la ventana correspondiente. Pero antes, tenemos que configurar el origen de datos para que, en vez de crear una rejilla, cree controles de edición independientes al ser arrastrado y soltado:



El resultado final, tras eliminar el editor de la columna *Creación*, que no nos interesa modificar, debe parecerse al siguiente:



Para configurar los botones, asigne el valor *OK* en la propiedad  *DialogResult* del botón *Aceptar*, y *Cancel* en el segundo botón. No cometa el error de muchos programadores, que crean un manejador de eventos para algo tan sencillo como terminar la ejecución modal del diálogo. Y muy importante: al soltar el origen de datos, Visual Studio ha creado un *BindingSource* en el diálogo. No olvide asignar este valor en la propiedad *BindingSource* del diálogo, pues esta es la base de toda la técnica de enlace y sincronización.

Por último, para probar el funcionamiento del sistema de lanzamiento de diálogos, vaya a la ventana de navegación sobre impuestos, e intercepte el evento *CellDoubleClick* de la rejilla:

```
private void impuestosDataGridView_CellDoubleClick(
    object sender, EventArgs e)
{
    if (e.RowIndex >= 0)
        MainForm.LayoutManager.Execute(typeof(DlgImpuestos), ParentForm);
}
```

Comprobamos el valor de la propiedad *RowIndex* para asegurarnos que el doble clic no ha tenido lugar sobre la fila de cabeceras de columnas. *LayoutManager*, por su parte, es una propiedad estática que hemos declarado en *MainForm*, para dar acceso global al gestor de ventanas de la aplicación.

# EJERCICIO 09

## Objetivos

Controles para navegación y edición

## Técnicas introducidas

Barras de herramientas, contexto de enlace, eventos del contexto de enlace

**N**O SE PREOCUPE: NO HAY POR QUÉ OBLIGAR al usuario a hacer doble clic sobre las cabeceras de filas para que permitirle que edite un registro. En este ejercicio vamos a redondear algunas aristas de nuestras ventanas de navegación, añadiendo botones a la barra de navegación con los comandos más necesarios para las tareas habituales de mantenimiento.

## Automatizando la edición

Vamos a automatizar en lo posible el código que se encarga de lanzar el diálogo de edición asociado a un tipo de ventana dado. Lo más sencillo sería poder crear una propiedad dentro de *BaseWindow* que nos indicase el tipo de editor apropiado para cada ventana de navegación. ¿El problema? Pues que no es sencillo crear una propiedad disponible en tiempo de diseño cuyos valores pertenezcan al tipo *System.Type*, que es el tipo que necesitamos. En consecuencia, intentaremos otra técnica: realizaremos esa asociación con un atributo aplicado sobre la clase del control de usuario.

En vez de crear un nuevo atributo, vamos a añadir una nueva propiedad a *NonModal*, el atributo que hemos venido utilizando para configurar las ventanas no modales. La propiedad se llamará *Editor* y su tipo será, precisamente, *System.Type*:

```
private System.Type editor = null;

public System.Type Editor
{
    get { return editor; }
    set { editor = value; }
}
```

Este es un ejemplo de cómo debemos usar la nueva propiedad:

```
[NonModal("Productos", "Ver", Caption="Catálogo de productos",
           UniqueInstance=true, Editor=typeof(DlgProducto))]
public partial class Productos : BaseWindow
{
    :
}
```

Para que una instancia de una clase derivada de *BaseWindow* pueda recuperar el valor de *Editor* en el atributo que le hemos asociado, declararemos una propiedad protegida, de tipo *System.Type*:

```
// Código en BaseWindow.cs
protected System.Type EditorType
{
    get
    {
        object[] attrs = this.GetType().GetCustomAttributes(
            typeof(NonModalAttribute), false);
        if (attrs.Length == 0)
            return null;
        else
            return (attrs[0] as NonModalAttribute).Editor;
    }
}
```

Nuevamente, usamos *GetCustomAttributes* para recuperar todos los atributos asociados al tipo que pertenezcan a la clase *NonModalAttribute*. Sabemos, gracias a la declaración del atributo, que el compilador sólo permitirá una instancia del atributo por clase, por lo que buscamos solamente el primer elemento del vector devuelto por *GetCustomAttributes*.

De esta manera, para mostrar el diálogo de edición de un registro, ya sea en respuesta a un doble clic sobre la rejilla, o al clic sobre un botón de la barra de navegación, tendremos que llamar a un método similar a éste, que situaremos también en la clase *BaseWindow*:

```
// Código en BaseWindow.cs
// En este ejercicio, modificaremos su prototipo.
protected void EditRecord()
{
    Type editor = this.EditorType;
    if (editor != null)
        MainForm.LayoutManager.Execute(editor, ParentForm);
}
```

Si se nos olvida indicar el editor asociado en el atributo *NonModal*, no habrá catástrofe, porque la propiedad *EditorType* devolverá un puntero vacío y no tendrá lugar la llamada a *Execute*.

## Creación de registros

Hasta el momento hemos visto cómo modificar un registro existente. Al crearse el diálogo de edición, sacamos una copia de la fila de datos original, y si el diálogo termina aceptando los cambios, modificamos la fila original *mezclando* la fila modificada con el conjunto de datos existente en la ventana de navegación. Tenga en cuenta que podemos hacerlo con tanta tranquilidad gracias a que no permitimos cambios en las claves primarias de registros existentes.

Es muy sencillo extender el código ya programado para tener en cuenta la inserción de nuevos registros. Necesitaremos, no obstante, un último cambio en el prototipo del método *InitData* del tipo de interfaz *IWindow*:

```
// IWindow: Inicialización
void InitData(IDataWindow parent, bool newRow);
```

También faremos un pequeño cambio en el prototipo de *Execute*, en la interfaz *ILayoutManager*:

```
// ILayoutManager: Ejecución modal
DialogResult Execute(System.Type type, Form owner, bool newRow);
```

Primero debemos propagar el nuevo parámetro a la implementación de *InitWindow* dentro de la clase *BaseWindow*. Como esta clase no aprovecha ninguno de los dos parámetros del método, el cambio consiste solamente en la puesta al día del prototipo. Donde evidentemente faremos cambios importantes es en la implementación de *InitData* dentro de *BaseDialog*:

```
public virtual void InitData(IDataWindow parent, bool newRow)
{
    parentWindow = parent;
    if (newRow)
        bindingSource.AddNew();
    else
    {
        DataTable.Rows.Add(
            ((DataRowView)parent.BindingSource.Current).Row.ItemArray);
        DataSet.AcceptChanges();
    }
}
```

¿Ha visto qué fácil? Si se trata de una inserción sólo tenemos que llamar al método *AddNew* sobre el componente de origen de enlace... y está claro que en ese caso no hay necesidad de copiar el registro subyacente. No necesitamos más cambios en el resto de los métodos ya implementados del diálogo.

Regresemos a *BaseWindow*, para encapsular la activación del editor dentro de un método protegido:

```
// Código en BaseWindow.cs
protected void EditRecord(bool newRow)
{
    Type editor = EditorType;
```

```

    if (editor != null)
        MainForm.LayoutManager.Execute(editor, ParentForm, newRow);
}

```

## Borrados

La implementación de los borrados es sencilla, pero hay que tener cuidado con ciertos detalles. Al igual que hicimos con las modificaciones e inserciones, vamos a crear un método auxiliar, al que llamaremos *DeleteRecord*. Este método recibirá un parámetro de tipo lógico, para saber si debemos mostrar el mensaje de confirmación del borrado, o si debemos ir al grano y cargarnos el registro a balazos:

```

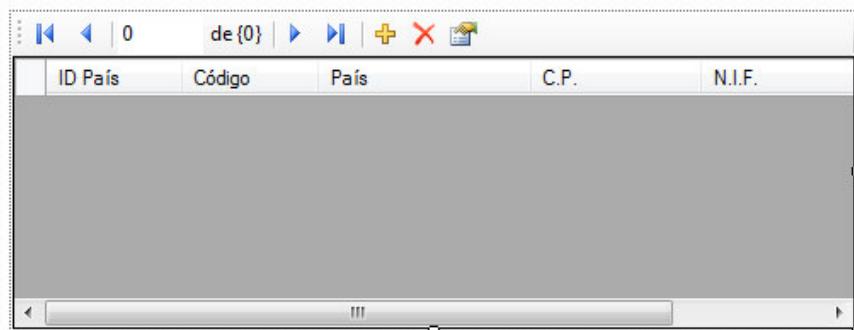
protected void DeleteRecord(bool ask)
{
    if (!ask ||
        MessageBox.Show("Va a eliminar el registro activo.\n" +
        "¿Está seguro?", "Confirmación", MessageBoxButtons.YesNo,
        MessageBoxIcon.Question) == DialogResult.Yes)
    {
        int bookmark = bindingSource.Position;
        bindingSource.RemoveCurrent();
        try
        {
            SaveChanges();
        }
        catch
        {
            DiscardChanges();
            bindingSource.Position = bookmark;
            throw;
        }
    }
}

```

El método *RemoveCurrent* del origen de datos elimina el registro del conjunto de datos, pero no en la base de datos. Para hacer definitivo el borrado, intentamos guardar los cambios, y si no lo logramos, restauramos el registro a su estado original. O todo, o nada. Observe que también restauramos la posición del cursor sobre la tabla afectada, si se produce algún problema.

## Reaprovechando botones

Me gustaría poder ahorrar más trabajo encapsulando una barra de herramientas en *BaseWindow*, pero no creo que sea buena idea, por varias razones:



- La técnica de arrastrar y soltar, que utilizamos para configurar automáticamente las rejillas, ya crea una barra de navegación. Es cierto que podríamos eliminar la rejilla superflua.
- Visual Studio no se lleva bien con los controles heredados de una ventana base, sobre todo cuando la edición de las propiedades de esos controles es complicada. Ese es el caso de las barras de navegación, y dicho sea de paso, de las rejillas de datos. Al situar la barra de navegación en *BaseWindow*, es probable que perdamos la posibilidad de añadir o modificar botones en la

misma. Tendríamos que prever al detalle cuáles botones podrían aparecer en una barra: ya sabemos, por ejemplo, que la ventana de productos necesita un botón adicional para la paginación.

Lo que sí haremos es reaprovechar los botones ya existentes en la barra de cada ventana de navegación particular para nuestras necesidades. La barra de navegación de Visual Studio viene de serie con los controles de navegación, hablando con propiedad, y tres botones adicionales, para añadir registros, borrarlos y para guardar los cambios. Para empezar, eliminaremos el botón de guardar, pues no lo necesitamos.

A continuación, tenemos que desconectar los botones de añadir y eliminar de su funcionalidad predeterminada. Para ello, seleccionamos la barra de navegación y asignamos una referencia nula en sus propiedades *AddNewItem* y *DeleteItem*. De esa manera, conservamos los botones junto con sus imágenes, pero eligiendo nosotros qué es lo que harán. Para esto último, por supuesto, interceptaremos sus eventos *Click*:

```
private void bindingNavigatorAddNewItem_Click(
    object sender, EventArgs e)
{
    EditRecord(true);
}

private void bindingNavigatorDeleteItem_Click(
    object sender, EventArgs e)
{
    DeleteRecord(true);
}
```

Añadiremos un botón para invocar el diálogo de edición de registros, y daremos una respuesta similar a su evento *Click*:

```
private void bnModificar_Click(object sender, EventArgs e)
{
    EditRecord(false);
}
```

---

### EJERCICIO PROUESTO

Puede investigar una alternativa a nuestra técnica de borrado. En vez de borrar directamente en la ventana de navegación, podríamos delegar la operación al viejo y conocido diálogo de edición e inserción. En tal caso, modificaremos los botones, incluyendo botones especiales para eliminar el registro o cancelar la operación. De esta manera el usuario tendría una información más completa sobre el contenido del registro que va a pasar a mejor vida, en vez de un simple aviso genérico.

---

# EJERCICIO 10

## Objetivos

Reglas de negocio durante la edición

## Técnicas introducidas

Actualizaciones, validaciones, *ErrorProvider*

**T**RAS UN EJERCICIO EN ALLEGRO VIVACE, inevitablemente toca un sereno *andante ma non troppo*, que aprovecharemos para retocar algunos detalles que las prisas nos han obligado a dejar pendientes. Por ejemplo, estamos realizando la búsqueda de productos mediante adaptadores creados al vuelo, en tiempo de ejecución, y ahora tenemos que arreglárnoslas para modificar estos registros de productos. De paso, veremos cómo implementar algunas validaciones sencillas y cómo manejar los errores de validación mediante el componente *ErrorProvider*.

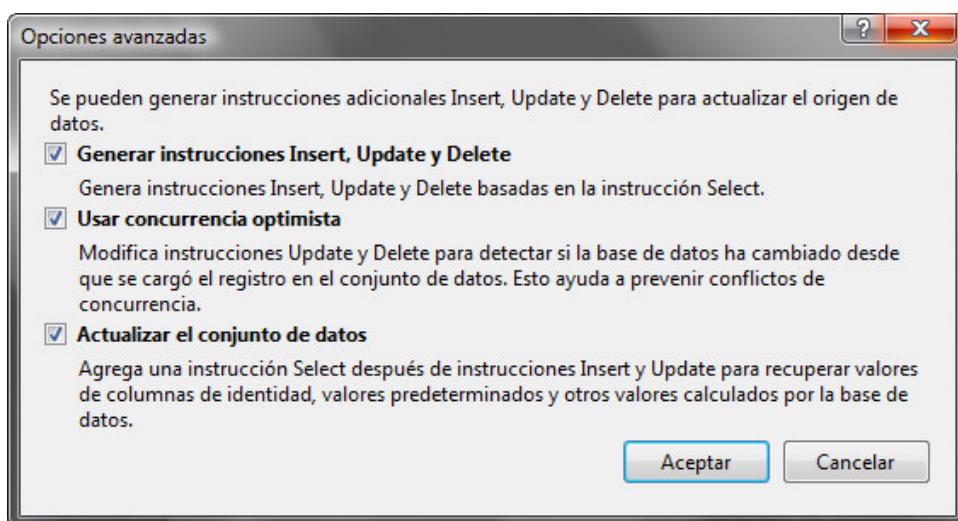
## Un adaptador para productos

Cuando añadimos la ventana de navegación sobre países, configuramos un adaptador de datos vinculado a esta tabla, para que controlase las lecturas y escrituras sobre la misma. La lectura, en particular, se implementó a través de la versión del método *Read* de *IDataServer* más sencilla: la que recibe solamente el nombre simbólico del conjunto de datos deseado. En cambio, la navegación sobre productos ha sido implementada mediante la segunda variante de *Read*: la que utiliza *cookies* para generar dinámicamente las consultas necesarias para la lectura incremental. Y para esta técnica no necesitamos un adaptador de datos. Pero si queremos añadir la posibilidad de modificar los registros de productos, necesitaremos un adaptador, a pesar de todo.

No pasa nada si configuramos un adaptador para que recupere *todos* los registros de productos, de golpe... siempre que no utilicemos esa faceta de su personalidad, sino los comandos de actualización generados a partir de dicha consulta. No obstante, le propongo algo mejor: en vez de una consulta para leer todos los registros, partiremos de una consulta que sólo devuelva un registro de producto, dada su clave primaria. Añada un *SqlDataAdapter* en el módulo de datos (*Data.cs*), y cuando el asistente de configuración le solicite la consulta SQL, teclee la siguiente instrucción:

```
select IDProducto, Producto, IDImpuesto, Coste, PVP,  
Existencias, Minimo, Imagen, Creacion, TS  
from Productos  
where IDProducto = @idProducto
```

@*idProducto* es un parámetro, para el que Visual Studio podrá deducir su tipo de datos sin mayor dificultad. Antes de cerrar el asistente, pulse el botón *Opciones avanzadas*, y asegúrese de que las tres opciones de generación estén activas:



¿Qué sentido tiene la consulta paramétrica que hemos preparado? De momento no la vamos a usar, pero nos será muy útil cuando tengamos que manejar la respuesta a violaciones de concurrencia. En ese momento, la consulta nos permitirá leer solamente el registro con conflictos de actualización, para que el usuario decida entonces si quiere repetir la operación, o si desea cancelarla.

Como ya es costumbre, modificaremos las instrucciones generadas automáticamente por el asistente de Visual Studio. Primero nos ocuparemos de las inserciones:

```
insert into Productos (Producto, IDImpuesto, Coste, PVP,
    Existencias, Minimo, Imagen)
values (@Producto, @IDImpuesto, @Coste, @PVP,
    @Existencias, @Minimo, @Imagen);
select IDProducto, Creacion, TS
from Productos
where IDProducto = scope_identity()
```

El primer cambio ha sido la eliminación de la columna *Creacion* de la lista de columnas en la instrucción **insert**. Recuerde que queremos que el registro contenga en esa columna la hora de creación en el servidor, no en el cliente, que siempre podría trucar las fechas. El segundo cambio también ha consistido en eliminar columnas, aunque esta vez ha sido en la consulta posterior a la inserción: sólo nos interesa recuperar el valor asignado a la clave primaria, a la columna *Creacion*, y el valor de la versión de fila que nos ha correspondido.

También hay algún cambio en la instrucción del comando asociado a la propiedad *UpdateCommand* del adaptador de datos:

```
update Productos
set Producto = @Producto, IDImpuesto = @IDImpuesto,
    Coste = @Coste, PVP = @PVP,
    Existencias = @Existencias, Minimo = @Minimo, Imagen = @Imagen
where IDProducto = @Original_IDProducto and TS = @Original_TS;
select TS
from Productos
where IDProducto = @Original_IDProducto
```

Aquí hemos quitado la asignación sobre *Creacion* de la cláusula **set** de **update** y hemos simplificado la consulta de recuperación de los datos más recientes.

La instrucción de borrados es la única que no sufrirá cambios:

```
delete from Productos
where IDProducto = @Original_IDProducto and TS = @Original_TS
```

Para terminar, sería aconsejable modificar también el valor de las propiedades *UpdateRowSource* en los tres comandos de actualización:

Comando	Valor de UpdateRowSource
<i>UpdateCommand</i>	<i>FirstReturnedRecord</i>
<i>InsertCommand</i>	<i>FirstReturnedRecord</i>
<i>DeleteCommand</i>	<i>None</i>

## Reglas de negocio y conjuntos de datos

Para conectar estos componentes con nuestro sistema de actualizaciones, tenemos todavía que retopear la implementación del método *Write*, dentro de la clase *Data*:

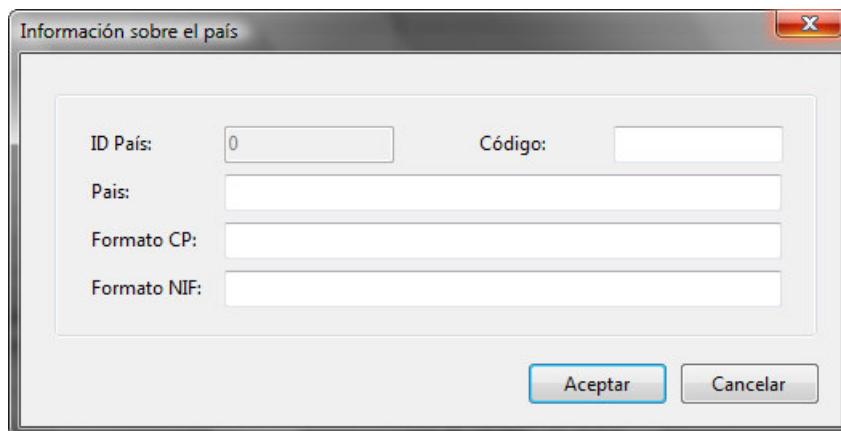
```
public System.Data.DataSet Write(
    string dsname, System.Data.DataSet delta)
{
    if (dsname == "DsPaises")
        daPaises.Update(delta);
    else if (dsname == "DsImpuestos")
        daImpuestos.Update(delta);
    else if (dsname == "DsProductos")
        daProductos.Update(delta);
```

```

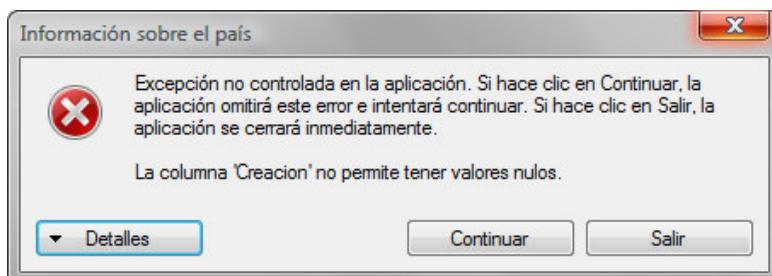
    else
        throw new ApplicationException(
            "La operación no ha sido implementada.");
    return delta;
}

```

Las modificaciones de registros existentes funcionará de maravillas, pero le apuesto que tendrá problemas para añadir registros. Cree, por ejemplo, un diálogo de edición para países:



Todo intento de añadir un país a la tabla terminará con una advertencia de esta guisa:



La raíz de nuestros males está en la combinación de restricciones de no nulidad junto a la carencia de reglas de inicialización automática. Visual Studio es capaz de deducir las primeras, al crear el conjunto de datos con tipos, pero no se esfuerza en encontrar las segundas. Tendremos que hacerlo nosotros mismos.

Primero activaremos el editor de conjuntos de datos, cargando la definición de *DsPaises.xsd*. Necesitamos tres cambios:

- Para la columna *Creacion*, asignaremos *True* en *AllowDBNull*.
- Para *FormatoCP*, asignaremos una cadena vacía en su propiedad *DefaultValue*. El valor por omisión de esta propiedad es *<DBNull>*. Evidentemente, no es lo mismo una cadena vacía que un valor nulo.
- Para *FormatoNIF*, también asignaremos una cadena vacía en *DefaultValue*.

También necesitamos cambiar algunas columnas en la definición de *DsProductos*:

- Para la columna *Creacion*, volveremos a asignar *True* en *AllowDBNull*.
- Asignaremos un cero en la propiedad *DefaultValue* de *Existencias* y *Mínimo*.

Le advierto que esta elegante técnica de especificación de valores por omisión no sirve para todos los tipos de inicializaciones que se pudieran presentar. En concreto, fallaría en casos tan simples como estos:

- Campos de fecha, o de fecha y hora, que deben inicializarse con la fecha actual.
- Campos de tipo *System.Guid* (*uniqueidentifier* en SQL Server) que deben recibir un valor aleatorio en el lado cliente.

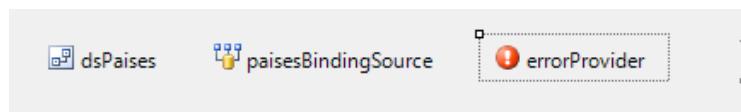
En esos casos no nos quedaría otra alternativa que realizar la inicialización manualmente, por medio de instrucciones, en vez de utilizar un mecanismo declarativo como el que acabamos de presentar.

## ¿Quién vigila al vigilante?

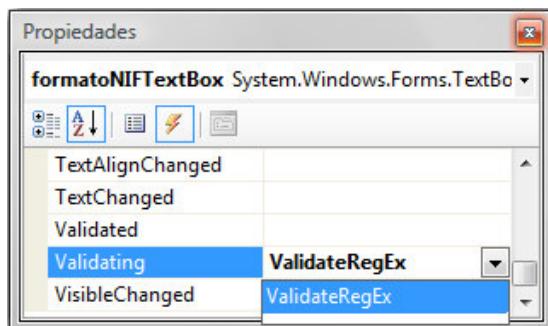
Para cada país, estamos almacenando el formato de sus códigos postales y sus números de identificación fiscal, y para ello utilizamos expresiones regulares. Sin embargo, las expresiones regulares, a su vez, tienen una sintaxis algo complicada. ¿Quién nos garantiza que la expresión regular que acabamos de teclear sea correcta y tenga sentido?

Es difícil saber si una expresión regular “tiene sentido” o no. Lo más que podemos hacer es ofrecer un asistente para que el usuario compruebe si determinadas cadenas de caracteres son aceptadas o rechazadas por una expresión dada. Pero sí estamos obligados a verificar la sintaxis de la propia expresión regular antes de modificar un registro de país.

Active el diálogo de edición de países, y añada un componente *ErrorProvider* en la bandeja de componentes:



Necesitamos crear un manejador compartido para los eventos *Validating* de los controles asociados a las columnas *FormatoCP* y *FormatoNIF*. Para ello, en vez de hacer doble clic sobre el editor del evento en el Inspector de Propiedades, teclee directamente el nombre que quiere darle al método, en el caso del primer cuadro de texto, y despliegue la lista de posibilidades para seleccionar el mismo método, en el caso del segundo *TextBox*:



Este es el método de validación que necesitamos:

```
private void ValidateRegEx(object sender,
    System.ComponentModel.CancelEventArgs e)
{
    TextBox ctrl = sender as TextBox;
    try
    {
        new System.Text.RegularExpressions.Regex(ctrl.Text);
    }
    catch(Exception exc)
    {
        e.Cancel = true;
        ctrl.SelectAll();
        errorProvider.SetError(ctrl, exc.Message);
    }
}
```

A falta de una mejor técnica para validar una expresión regular, me he limitado a crear un objeto de la clase *RegEx* basado en la expresión, e interceptar cualquier excepción que el constructor de esta clase generase. Si capturamos una excepción, abortamos la validación, seleccionamos el control afectado por el error, y le asociamos un mensaje de error, con la ayuda del proveedor de errores. Si

no encontramos problemas, el objeto creado se descarta automáticamente, y queda a merced del terrible devorador de objetos descarriados que es el Colector de Basura.

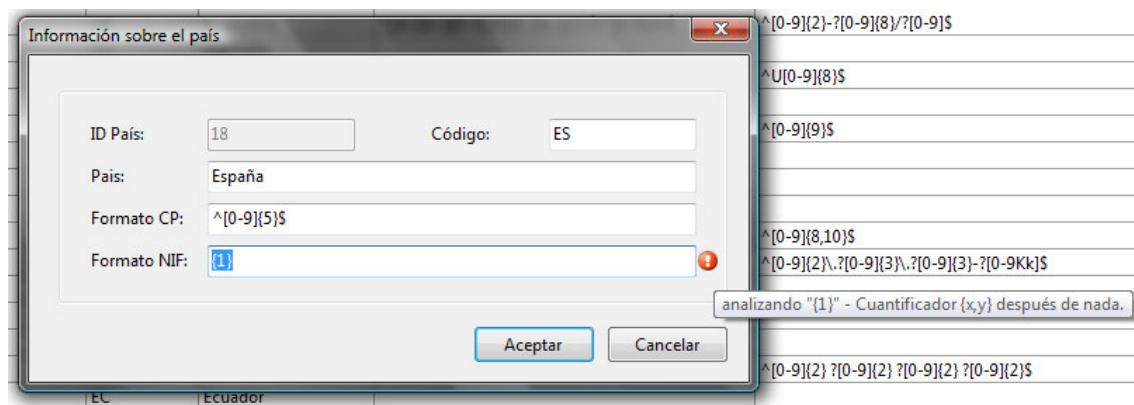
**NOTA**

Es sensato esperar que las expresiones regulares necesarias para validar códigos postales e identificadores fiscales no utilicen todas las características que éstas permiten. Una alternativa interesante para validar estas expresiones sería definir una sintaxis restringida para las expresiones aceptadas, y validar entonces las expresiones con la ayuda de otra expresión predefinida. Sé que puede sonar un poco al cuento del huevo y la gallina, pero no hay tal cosa.

En compensación, cuando alguno de los controles de edición logra validar su contenido, debemos eliminar el mensaje de error y el ícono asociado. Para ello crearemos un manejador compartido para los eventos *Validated* de los dos cuadros de edición:

```
private void CleanRegEx(object sender, EventArgs e)
{
    errorProvider.SetError(sender as TextBox, "");
}
```

Para probar el funcionamiento de la técnica, tendrá que teclear alguna expresión regular incorrecta. Los errores más frecuentes tienen que ver con paréntesis, corchetes y llaves mal pareados. Si quiere provocar un error más sofisticado, utilice un indicador de repeticiones aislado, que no esté asociado a nada “repetible”, como en la siguiente imagen:



Por supuesto, en ningún lugar está escrito que tengamos que propagar el mensaje original de la excepción. Para la mayoría de los usuarios, un mensaje del tipo *Hace falta al menos dos neuronas para editar este valor* será más que suficiente.

# EJERCICIO 11

## Objetivos

Almacenamiento en caché

## Técnicas introducidas

Versiones de filas, *ReadXml*, *WriteXml*

**A**LEJÉMONOS UN POCO DE LA aplicación. No, no quiero decir físicamente: en vez de pensar en la dinámica que enlaza cada acción con la siguiente, desentendámonos de estos detalles y pensemos a la manera Zen. Situémonos, por ejemplo, en el lado del servidor. ¿Qué veríamos si nos encarnásemos en un servidor SQL? A un puñado de aplicaciones estúpidas que insisten en que le envíemos, una y otra vez, la misma aburrida, tediosa e insopportable tabla de países.

## Una caché de corta vida

Hay muchas tablas en una base de datos típicas que satisfacen estas dos condiciones:

- 1 Tienen un número relativamente pequeño de registros.
- 2 Es muy poco probable que, una vez inicializadas, cambie su contenido.

Sin embargo, estas tablas son utilizadas intensamente por las aplicaciones. Cada vez que localizamos un registro de cliente, tenemos que traducir el identificador de país almacenado en el registro de cliente en el nombre del país. Cada vez que navegamos sobre un registro de producto, hay que traducir el identificador del impuesto en el nombre dado al robo legal y su tasa de latrocínio. ¿No sería buena idea almacenar estas tablas en una caché local, de manera que disminuyamos el tráfico de red innecesario provocado por estas tablas?

Comencemos por lo más simple: mantener el conjunto de datos de países en memoria durante la vida de la aplicación. Tal como está organizado nuestro código de acceso a datos, cada vez que abrimos una ventana de navegación se ejecuta la consulta pertinente. No se trata de una mala implementación: al repetir la consulta, garantizamos una copia actualizada de los datos reales. Recordemos, no obstante, que los datos sobre países son casi de sólo lectura...

El código necesario es muy sencillo. Añadiremos una variable de tipo *DsPaises*, el conjunto de datos con tipos que almacena países, como campo de la clase *Data*, la llamaremos *ldsPaises* y la inicializaremos con un puntero vacío. El método auxiliar *LeerPaises* debe devolver el conjunto de datos al que apunta *ldsPaises*. Si esta variable contiene **null**, quiere decir que tenemos que leer por primera vez el conjunto de datos:

```
#region Cache local de países

private DsPaises ldsPaises = null;

protected System.Data.DataSet LeerPaises(bool forzarLectura)
{
    if (ldsPaises == null || forzarLectura)
    {
        ldsPaises = new DsPaises();
        daPaises.Fill(ldsPaises.Paises);
    }
    return ldsPaises;
}

#endregion
```

La nueva función *LeerPaises* se utiliza entonces en la implementación del método *Read*:

```
public System.Data.DataSet Read(string dsname)
{
    if (dsname == "DsPaises")
        return LeerPaises(false);
```

```
:  
}
```

Es más difícil ver lo que sucede durante las escrituras. ¿Recuerda que los conjuntos de datos devueltos por *Read* se “vacían” sobre un segundo conjunto de datos que se enlaza a la rejilla? Esto quiere decir que cuando cambiamos algún registro de país, estamos trabajando con una copia de los datos diferente de la almacenada por *ldsPaises*. Por lo tanto, debemos aprovechar la llamada a *Write* para sincronizar también el contenido de *ldsPaises*. Primero transformamos el código de *Write*:

```
public System.Data.DataSet Write(  
    string dsname, System.Data.DataSet delta)  
{  
    if (dsname == "DsPaises")  
        return EscribirPaises(delta);  
    :  
}
```

Y ésta es la implementación del nuevo método *EscribirPaises*:

```
protected System.Data.DataSet GuardarPaises(DataSet delta)  
{  
    foreach (DataRow row in delta.Tables[0].Select(  
        "", "", DataViewRowState.Deleted))  
    {  
        DataRow r = ldsPaises.Paises.FindByIDPais(  
            Convert.ToInt32(row["IDPais"], DataRowVersion.Original));  
        if (r != null)  
            r.Delete();  
    }  
    daPaises.Update(delta);  
    ldsPaises.Merge(delta, false);  
    ldsPaises.AcceptChanges();  
    return delta;  
}
```

Primero borramos en *ldsPaises* los registros que vienen marcados como borrados en el parámetro *delta*. Preste atención al detalle siguiente:

```
row["IDPais", DataRowVersion.Original]
```

Esta expresión nos devuelve el valor de la clave primaria del registro *row*. Pero esta variable apunta a una fila marcada como borrada. Si no especificamos la versión del valor de *IDPais*, se produciría una excepción. Por cierto, observe cómo aprovechamos los métodos del conjunto de datos con tipos para localizar filas a partir de su clave primaria.

Por último, estas tres líneas se encargan de sincronizar las inserciones y modificaciones, y de aceptar todos los cambios realizados:

```
daPaises.Update(delta);  
ldsPaises.Merge(delta, false);  
ldsPaises.AcceptChanges();
```

La llamada a *Update*, además de grabar los cambios en la base de datos, recupera los valores más recientes de los registros afectados.

## Persistencia XML

Si sólo implementásemos este tipo de caché, obtendríamos una ganancia considerable: mientras la aplicación estuviese cargada en memoria, no tendría que volver a leer la lista de países. Pero podemos ir más allá, por supuesto. En su estado actual, la aplicación debe recuperar la tabla de países cada vez que se ejecuta. Si la aplicación se utiliza durante una jornada de ocho horas, puede que no parezca importante que haya que realizar esta lectura una vez, al inicio de la jornada. Piense, sin embargo, en que a primera hora tendremos una avalancha de peticiones precipitándose sobre la indefensa tabla, a medida que los soñolientos empleados encienden con resignación sus ordenadores.

182

res. Si queremos seguir ahorrando, tendremos que crear una copia de la tabla en el disco duro local, para que esa lectura mañanera no implique un viaje de ida y vuelta al servidor.

Vamos a modificar la implementación de *LeerPaises* en la forma que muestro a continuación:

```
protected System.Data.DataSet LeerPaises(bool forzarLectura)
{
    if (ldsPaises == null || forzarLectura)
    {
        string filename = System.IO.Path.Combine(
            System.Windows.Forms.Application.CommonAppDataPath,
            "paises.xml");
        ldsPaises = new DsPaises();
        bool loaded = false;
        if (System.IO.File.Exists(filename))
            try
            {
                ldsPaises.ReadXml(filename, XmlReadMode.ReadSchema);
                ldsPaises.AcceptChanges();
                loaded = true;
            }
            catch {}
        if (!loaded)
        {
            daPaises.Fill(ldsPaises.Paises);
            ldsPaises.WriteXml(filename, XmlWriteMode.WriteSchema);
        }
    }
    return ldsPaises;
}
```

La piel que rodea al núcleo del método sigue siendo la misma: nuestro objetivo a corto plazo es evitar que sudemos la camiseta si ya hay un conjunto de datos asociado a la variable *ldsPaises*. Esta vez, cuando *ldsPaises* contiene un puntero vacío, en vez de lanzarnos a leer desde el servidor SQL, comprobamos antes si existe un fichero *paises.xml* en el mismo directorio donde reside la aplicación. Si la respuesta es afirmativa, leemos el contenido del fichero dentro de un conjunto de datos vacío, creado para la ocasión. No se preocupe de dónde ha salido el fichero, porque lo veremos a continuación. Efectivamente si no hay tal fichero, ejecutamos el código que recupera los registros desde el servidor SQL... y nos tomamos la molestia adicional de guardar datos y esquema en el fichero XML antes mencionado.

También necesitamos un cambio pequeño en *GuardarPaises*:

```
protected System.Data.DataSet GuardarPaises(DataSet delta)
{
    foreach (DataRow row in delta.Tables[0].Select(
        "", "", DataViewRowState.Deleted))
    {
        DataRow r = ldsPaises.Paises.FindByIDPais(
            Convert.ToInt32(row["IDPais"], DataRowVersion.Original));
        if (r != null)
            r.Delete();
    }
    daPaises.Update(delta);
    ldsPaises.Merge(delta, false);
    ldsPaises.AcceptChanges();
    ldsPaises.WriteXml(System.IO.Path.Combine(
        System.Windows.Forms.Application.CommonAppDataPath,
        "paises.xml"), XmlWriteMode.WriteSchema);
    return delta;
}
```

Si modificamos la copia en memoria de la tabla de países, es necesario que actualicemos también la copia en disco. Alternativamente, podríamos borrar el fichero, para que la aplicación lo recrease la próxima vez que fuese ejecutada. En cualquier caso, no se preocupe excesivamente por buscar la solución menos “costosa”. ¿No habíamos quedado en que las actualizaciones serían infrecuentes?

## El momento del último cambio

La técnica que acabo de mostrarle funciona cuando las modificaciones se hacen desde el mismo ordenador, pero no nos ayuda a mantener la caché actualizada cuando las modificaciones se realizan desde otro sitio. La solución más sencilla es obtener, de alguna manera, la fecha y hora en que fue modificada la tabla de países por última vez, y compararla con la fecha en que creamos el fichero para la caché.

Ahora bien, ¿cómo averiguar si una tabla ha sido modificada después de un momento determinado? Una solución de fuerza bruta consistiría en tener una tabla de una sola fila, en una de cuyas columnas almacenaríamos la fecha y hora de la última modificación sobre la tabla de países. Para ello, necesitaríamos también *triggers* que se dispararían durante las tres operaciones clásicas de actualización sobre *Paises*, para modificar la tabla con la hora del último cambio. Esta técnica, como puede ver, penalizaría las actualizaciones, aunque reconozco que no demasiado.

Pero existe una técnica mejor: aprovechemos esas columnas de tipo **rowversion** que hemos reparado por toda la base de datos. El tipo mencionado se almacena como un entero de ocho bytes, y los valores asignados por el servidor siguen una secuencia monótonamente creciente. Por lo tanto, podemos hacernos una idea de cuándo fue la última vez que se modificó la tabla de países ejecutando la siguiente consulta:

```
select cast(max(TS) as bigint) as MaxTS  
from dbo.Paises
```

A partir de aquí, podemos implementar el siguiente algoritmo para mantener actualizada la caché de la tabla de países:

- 1 La aplicación, al cargarse, ejecuta la consulta antes mencionada, para averiguar el momento de la última modificación. Para no ralentizar el proceso de carga, podemos mezclar esta comprobación dentro del proceso de identificación y validación del usuario. También es recomendable, si implementamos la caché para otras tablas, agrupar todas estas consultas en un mismo procedimiento almacenado.
- 2 Una vez recuperado el mayor número de versión dentro de la tabla de países, comparamos este valor con el mayor número de versión en la copia de la tabla almacenada en la caché.
- 3 Si el valor local es menor que el valor recuperado desde la base de datos, hay que releer la caché. En caso contrario, la caché sigue siendo válida.

**NOTA**

En realidad, este algoritmo falla en un caso muy especial: si el único cambio en la tabla consiste en la desaparición de uno o varios registros por borrados. No obstante, en muchos sistemas nunca se eliminan registros físicamente (recuerde, la información es oro!). Y en el peor de los casos, nos bastaría con recuperar también el número de registros en la tabla y compararlo con el número de registros en la caché.

Vamos a añadir un nuevo método a la interfaz *IDataServer*, que nos permitirá leer los números de versión máximos desde la base de datos:

```
public interface IDataServer  
{  
    long[] GetVersions();  
    System.Data.DataSet Read(string dsname);  
    System.Data.DataSet Read(ref PageCookie cookie);  
    System.Data.DataSet Write(string dsname, System.Data.DataSet delta);  
}
```

Observe que el nuevo *GetVersions* devuelve todo un vector de números de versión: así evitamos tener que modificar la interfaz cada vez que implementemos una caché para otra tabla. Puede que le extrañe que usemos **long** para representar los números de versión dentro de la aplicación. En tal caso, debe saber que **long** se utiliza en C# para los enteros de 64 bits del *Common Language Runtime*. Para implementar el nuevo método, podemos añadir un *SqlCommand* dentro del componente *Data*, y configurarlo con la consulta antes mostrada:

```
select cast(max(TS) as bigint) as MaxTS
from    dbo.Paises
```

El código de *GetVersions* sería entonces el siguiente:

```
public long[] GetVersions ()
{
    long[] versions = new long[1];
    sqlConn.Open();
    try
    {
        versions[0] = (long) cmdVersion.ExecuteScalar();
    }
    finally
    {
        sqlConn.Close();
    }
    return versions;
}
```

Hemos aplicado el método *ExecuteScalar* sobre el comando porque el conjunto de resultados de la consulta consiste en una sola columna dentro de una sola fila. El método se llamaría dentro de los constructores de la clase *Data*:

```
public Data(System.ComponentModel.IContainer container)
{
    container.Add(this);
    InitializeComponent();
    sqlConn.ConnectionString = Properties.Settings.Default.Conexion;
    versions = Login();
}

public Data()
{
    InitializeComponent();
    sqlConn.ConnectionString = Properties.Settings.Default.Conexion;
    versions = Login();
}
```

En realidad, nuestra aplicación sólo está usando el segundo constructor, pero no nos cuesta nada respetar las reglas del juego.

## Comparando con la versión local

Nos queda comprobar si el número de versión asociado a la tabla de países que acabamos de leer es igual al número de versión de alguna de las filas de la tabla almacenada en la caché local. Me gustaría poder comprobarlo con un filtro, o con la ayuda del método *Compute*, pero las expresiones de filtros y columnas calculadas en .NET no ofrecen demasiados recursos para trabajar con columnas de tipo **rowversion**. Tendremos que convertir manualmente los valores de la columna TS al tipo **long**, y realizar las comparaciones recorriendo la colección de filas. Nos ocuparemos primero de la conversión al tipo **long**:

```
protected long ConvertRowVersion(byte[] rv)
{
    long result = 0;
    for (int i = 0; i < rv.Length; i++)
        result = result * 256 + rv[i];
    return result;
}
```

En .NET, las columnas de tipo **rowversion** contienen valores que son vectores de ocho bytes. El algoritmo anterior considera que el primer elemento del vector es el más significativo. Con este método, podemos averiguar el mayor número de versión almacenado en la caché:

```
protected long ComputeLocalVersion()
{
    long result = 0;
```

```

foreach (DsPaises.PaisesRow r in ldsPaises.Paises)
{
    long v = ConvertRowVersion(r.TS);
    if (v > result)
        result = v;
}
return result;
}

```

Seamos objetivos: no debe costar demasiado recorrer las pocas filas del conjunto de datos de países. Pero, en caso de que le preocupe el tiempo requerido, vamos a almacenar explícitamente el valor calculado por *ComputeLocalVersion* en las propiedades extendidas del conjunto de datos. Estas “propiedades” se refieren simplemente a un diccionario almacenado en la propiedad *ExtendedProperties* de la clase *DataSet*, cuyo contenido se guarda en disco como parte del contenido del conjunto de datos. Esta será la versión final de *LeerPaises*:

```

protected System.Data.DataSet LeerPaises (bool forzarLectura)
{
    if (ldsPaises == null || forzarLectura)
    {
        string filename = System.IO.Path.Combine(
            System.Windows.Forms.Application.CommonAppDataPath,
            "paises.xml");
        ldsPaises = new DsPaises ();
        bool loaded = false;
        if (System.IO.File.Exists(filename))
        try
        {
            ldsPaises.ReadXml(filename, XmlReadMode.ReadSchema);
            ldsPaises.AcceptChanges();
            if (System.Int64.Parse(ldsPaises.ExtendedProperties[
                "VersionPaises"].ToString()) >= versions[0])
                loaded = true;
            else
                ldsPaises.Clear();
        }
        catch {}
        if (!loaded)
        {
            daPaises.Fill(ldsPaises.Paises);
            ldsPaises.ExtendedProperties["VersionPaises"] =
                ComputeLocalVersion();
            ldsPaises.WriteXml(filename, XmlWriteMode.WriteSchema);
        }
    }
    return ldsPaises;
}

```

Del mismo modo, tenemos que actualizar el valor de la propiedad extendida cada vez que se produzca una modificación local, dentro del método *EscribirPaises*:

```

protected System.Data.DataSet GuardarPaises (DataSet delta)
{
    foreach (DataRow row in delta.Tables[0].Select(
        "", "", DataViewRowState.Deleted))
    {
        DataRow r = ldsPaises.Paises.FindByIDPais(
            Convert.ToInt32(row["IDPais", DataRowVersion.Original]));
        if (r != null) r.Delete();
    }
    daPaises.Update(delta);
    ldsPaises.Merge(delta, false);
    ldsPaises.AcceptChanges();
    ldsPaises.ExtendedProperties["VersionPaises"] =
        ComputeLocalVersion();
    ldsPaises.WriteXml(System.IO.Path.Combine(
        System.Windows.Forms.Application.CommonAppDataPath,

```

```
    "paises.xml"), XmlWriteMode.WriteSchema);
    return delta;
}
```

## Un seguro de vida

Siempre habrá alguien que se preocupe: ¿y si alguien modifica un registro de país desde otro puesto mientras nuestra aplicación está en marcha? Bueno, si existe tal posibilidad, quizás sea preferible que no utilice esta técnica de programación. Pero seamos honestos: ¿qué posibilidad hay de que tal cosa ocurra? O más bien, sopesa el coste de aceptar este pequeño riesgo con la ganancia en eficiencia que obtendrá a cambio.

No obstante, es conveniente que ofrezcamos a nuestros pilotos un botón de eyección para casos críticos. En el menú de la ventana principal he añadido un comando titulado *Actualizar caché*, dentro del menú *Ficheros*:

```
private void miActualizar_Click(object sender, System.EventArgs e)
{
    (Data.Instance as Data).LeerPaises(true);
}
```

En general, si tenemos más de una tabla en caché, tendremos que decidir si actualizamos todas las tablas de golpe, o si ofrecemos distintos comandos para cada una de ellas.

Antes de terminar el ejercicio, debo decirle que existe otra técnica para la detección de cambios en una tabla. Eche un vistazo a la siguiente consulta:

```
select checksum_agg(checksum(*))
from   dbo.Paises
```

La función interna, **checksum**, calcula una especie de valor condensado a partir de los valores de las columnas de una fila. Estos valores vuelven a digerirse por medio de la función estadística exterior, **checksum\_agg**. El resultado es un valor que resume de algún modo todos los valores de todas las columnas de la tabla. Nadie nos garantiza que todo cambio se traduzca en un valor diferente para la consulta mostrada, pero es muy probable que esto ocurra: tenga presente que las dos funciones utilizadas devuelven enteros de 32 bits, y piense en el número total de posibles valores que puede obtener como resultado.

La función **checksum** no funciona con columnas de tipo texto o imagen.

# EJERCICIO 12

Objetivos

Búsquedas

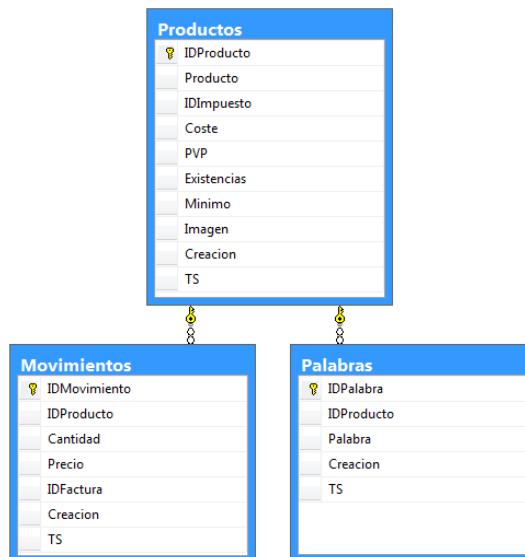
Técnicas introducidas

Funciones de tablas

**Y**A HEMOS IMPLEMENTADO DOS formas de leer registros: cargando toda la tabla, o recuperando paquetes de registros de un tamaño predeterminado. Vamos ahora a implementar otra técnica, mucho más potente, que nos permitirá recuperar registros que cumplan con determinada condición. En nuestro caso, utilizaremos la tabla de productos para recuperar productos asociados a determinadas palabras claves.

## Un repaso a la lista de palabras

A modo de recordatorio, aquí tiene la definición de la tabla de productos, junto con algunas de las tablas relacionadas:



Falta la tabla de impuestos en el diagrama... pero voy a seguir ignorándola de momento. Como éste muestra, para cada producto hay una lista de movimientos, que detalla las entradas y salidas de cada producto. Esta tabla de movimientos se administrará de forma explícita por parte del usuario. En realidad, deberíamos haber desactivado el control de edición asociado a la columna *Existencias*, en el cuadro de diálogo de productos.

Por el contrario, la otra tabla, *Palabras*, se mantiene de forma automática, con la ayuda de *triggers*. Cuando insertamos un producto en la tabla correspondiente, el *trigger* de inserción descompone el nombre del producto en palabras individuales. Primero se descartan las palabras que están enumeradas en la tabla auxiliar *Exclusiones* (no mostrada en el diagrama). De esta forma, eliminamos palabras y partículas que aportan poca información, como los artículos, conjunciones y ciertas preposiciones. Finalmente, las palabras que sobreviven a la criba se insertan en la tabla *Palabras*, asociándolas al producto recién insertado.

Se trata de una técnica bastante común, que soporta las siguientes variantes:

- En vez de almacenar dos veces una palabra encontrada en dos productos distintos, se puede usar un diccionario de palabras, para ahorrar algo de espacio.
- Si usamos un diccionario, podemos mantener un contador automático para cada palabra, que nos indique el número de productos en los que se ha usado. Esta información nos permitiría crear condiciones de búsqueda más eficientes.

- También trabajar con un diccionario predefinido con índices de relevancia para cada palabra. La palabra *Aranzado*, por ejemplo, es menos importante que *Geometría*. Además de mejorar las consultas de búsquedas, esta técnica nos permitiría localizar productos relacionados con más tino.
- Si en la propia tabla de *Palabras* almacenásemos la posición de cada palabra dentro del título, podríamos ejecutar consultas por proximidad: productos que contienen las palabras *X* e *Y* en orden consecutivo.

Hay infinitas variantes de diseño, pero hemos elegido la más sencilla de ellas para nuestra base de datos de ejemplo.

¿Para qué tanto trabajo? Piense, si no, en qué habría que hacer para buscar libros sobre Windows. Casi siempre, esta palabra estará situada en medio del título o al final. Es cierto que podríamos usar una condición basada en el operador **like**, como ésta:

```
Producto like '%Windows%'
```

Pero este tipo de condiciones, en las que el patrón de búsqueda contiene un comodín al principio, impiden el uso de índices por parte del optimizador SQL. En cambio, veremos que al descomponer el título en palabras, los posibles comodines se situarían siempre al final del patrón, y en ese otro caso, SQL sí puede aprovechar los índices existentes para optimizar la consulta.

## Funciones de tablas en SQL Server

- 129 En vez de obligar al programador a generar una complicada consulta SQL cada vez que al usuario se le ocurra una búsqueda con una, dos o cincuenta palabras claves, es mejor definir por adelantado *funciones de tablas* que se ocupen al menos de los casos más frecuentes.

Por ejemplo, nuestra base de datos ya contiene una función llamada *Productos01*:

```
create function dbo.Productos01 (
    @key01 varchar(30))
returns table as return (
    select *
    from dbo.Productos
    where IDProducto in (
        select IDProducto
        from dbo.Palabras
        where Palabra like @key01
            collate modern_spanish_ci_si))
go
```

Tome nota de la cláusula **collate** que aparece a continuación de la expresión basada en el operador **like**. Gracias a ella, la comparación ignora las mayúsculas y minúsculas, y los acentos. Aparte de esto, **collate** determina el orden relativo entre los caracteres, aunque en una comparación con **like** esta información no tenga mayor utilidad.

Con esta función ya definida, si alguien quiere listar los productos asociados a una sola palabra clave puede teclear una consulta como la siguiente:

```
select *
from Productos01('ADO')
order by Producto, IDProducto
```

Como el parámetro se utiliza en una comparación de patrones, podemos también incluir comodines en la palabra clave. Para buscar los libros que contienen Win32 o Windows en su título podríamos ejecutar esta consulta:

```
select *
from Productos01('Win%')
order by Producto, IDProducto
```

También he definido una función *Productos02* que recibe dos palabras claves como parámetros de entrada:

```

create function dbo.Productos02 (
    @key01  varchar(30),
    @key02  varchar(30))
returns table as return (
    select *
    from   dbo.Productos
    where  IDProducto in (
        select IDProducto
        from   dbo.Palabras
        where  Palabra like @key01
               collate modern_spanish_ci_si) and
               IDProducto in (
        select IDProducto
        from   dbo.Palabras
        where  Palabra like @key02)
               collate modern_spanish_ci_si)
go

```

Cada palabra se vincula a una subconsulta, y ambas subconsultas se unen mediante el operador lógico **and**. Esto significa que, si especificamos dos palabras claves, cada uno de los registros que obtenemos contiene *ambas* palabras. Esta es una decisión importante: la alternativa sería mostrar los registros que contienen al menos una de las dos palabras. Prefiero la interpretación basada en **and** porque permite afinar la búsqueda añadiendo más palabras.

La función anterior se utilizaría como en el siguiente ejemplo:

```

select *
from   Productos02('ADO', 'NET')
order by Producto, IDProducto

```

La base de datos contiene también una función *Productos03*, pero ahí me he detenido. No tiene mucho sentido realizar una búsqueda por diez palabras claves: cada palabra adicional restringe el tamaño del conjunto resultado, y es muy probable que la consulta en cuestión no devolviese producto alguno... a no ser que hayamos usado palabras muy generales o hayamos tecleado el título completo del libro. Es cierto que tres palabras son pocas: le propongo que escriba un par de funciones más.

#### EJERCICIO PROPUESTO

Si le sobra el tiempo y está aburrido, explore tipos alternativos de búsquedas, estén o no basadas en la tabla auxiliar de palabras. Por ejemplo, puede que le interese permitir búsquedas de productos que incluyan una u otra palabra: en vez de usar una conjunción entre las condiciones, se utilizaría una disyunción. O, dado un producto, podemos buscar otros productos que tengan alguna palabra en común con él. Y si quiere esmerarse, podría incluso ordenar el resultado de acuerdo al número de palabras coincidentes.

## Desmenuzando cadenas

Veamos ahora cómo conectar este mecanismo de búsquedas dentro de la aplicación. Lo más sencillo es añadir un cuadro de texto en la barra de navegación de la ventana de productos, junto a un botón para iniciar la búsqueda:



He bautizado el cuadro de texto con el nombre de *txBuscar*, y el botón como *bnBuscar*. Esta es la respuesta al evento *Click* del botón de marras:

```

private void bnBuscar_Click(object sender, System.EventArgs e)
{
    System.Text.RegularExpressions.Regex regex =
        new System.Text.RegularExpressions.Regex("[^A-Za-z0-9%_]+");
    string s = "";
    int count = 0;

```

```

foreach (string word in regex.Split(txBuscar.Text.Trim()))
{
    s += " " + word.Replace("'", "''") + "%", ";
    if (++count) == 3)
        break;
}
if (s != "")
{
    cookie = new PageCookie("Productos",
        "Productos" + count.ToString() + "(" +
        s.Remove(s.Length - 2, 2) + ")",
        "", "Producto", IDProducto", 20);
    dsProductos.Clear();
    dsProductos.Merge(Data.Instance.Read(ref cookie), false);
    dsProductos.AcceptChanges();
    bnMasRegistros.Enabled = !cookie.Disabled;
}
}

```

Hemos vuelto a utilizar expresiones regulares; esta vez, para separar la cadena en palabras aisladas. La expresión regular empleada es la siguiente:

[^A-Za-z0-9%\_]+

He resaltado el acento circunflejo ubicado dentro de los corchetes. Sin él, la expresión entre corchetes concordaría con cualquier carácter alfanumérico, el porcentaje y el subrayado. El acento circunflejo indica que aceptaremos cualquier carácter excepto los mencionados. El signo de adición final indica que aceptaremos uno o más caracteres del tipo mencionado.

La expresión regular es utilizada por el método *Split* de la clase *RegEx*: está función utiliza la expresión para identificar separadores entre palabras, y devuelve un vector con las palabras encontradas. Recorremos el vector con la instrucción **foreach**, y construimos la lista de parámetros que pasaremos a la función de tabla que corresponda. Para elegir la función apropiada, contamos las palabras encontradas, y nos detenemos en la tercera, porque ese es el número de funciones que he definido en la base de datos. Para terminar, el método utiliza la función como si se tratase de un nombre de tabla, y la pasa a nuestro más que familiar método *Read*, de la interfaz *IDataServer*.

Hay un detalle desagradable: me he visto obligado a añadir un nuevo campo a la clase *PageCookie*:

```

public class PageCookie
{
    public readonly string TableName;
    :

    public PageCookie(string tablename, string table,
        string filter, string sort, int count)
    {
        :
    }
}

```

Necesitamos el nombre de la tabla para que la implementación de *Read* pueda añadirlo en la propiedad *TableMappings* del adaptador de datos, y darle así un nombre a la tabla generada por el método *Fill* del mismo adaptador. Encontrará los detalles en el método *Read* de la clase *Data*:

```

public System.Data.DataSet Read(ref PageCookie cookie)
{
    using (SqlDataAdapter da = new SqlDataAdapter(cookie.SQL, sqlConn))
    {
        da.TableMappings.Add("Table", cookie.TableName);
        DataSet ds = new DataSet();
        da.Fill(ds);
        if (ds.Tables[0].Rows.Count < cookie.Count)
            cookie = new ReturnedCookie(cookie);
    }
}

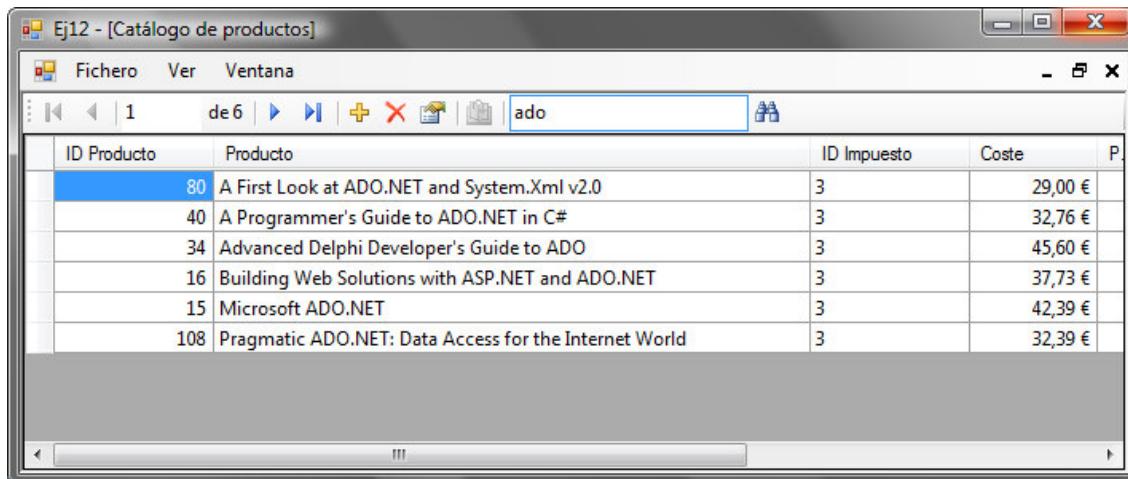
```

```
        else
            cookie = new ReturnedCookie(cookie,
                SynthesizeExpression(ds.Tables[0], cookie.Sort));
        return ds;
    }
}
```

Y esto es todo, amigos... excepto que podemos portarnos bien y disparar también la búsqueda cuando el usuario teclee INTRO sobre el cuadro de edición:

```
private void txBuscar_KeyPress(
    object sender, System.Windows.Forms.KeyPressEventArgs e)
{
    if (e.KeyChar == '\r')
    {
        bnBuscar.PerformClick();
        e.Handled = true;
    }
}
```

El resultado puede verse en la siguiente imagen:



### EJERCICIO PROUESTO

Compruebe que el mecanismo de paginación sigue funcionando correctamente. Tendrá que disminuir el número de registros leídos en cada petición, porque ninguna de las palabras claves existentes devuelve los veinte registros que hacen falta para llenar una página... o puede usar una letra aislada como criterio de búsqueda.

# EJERCICIO 13

## Objetivos

Navegación maestro/detalles

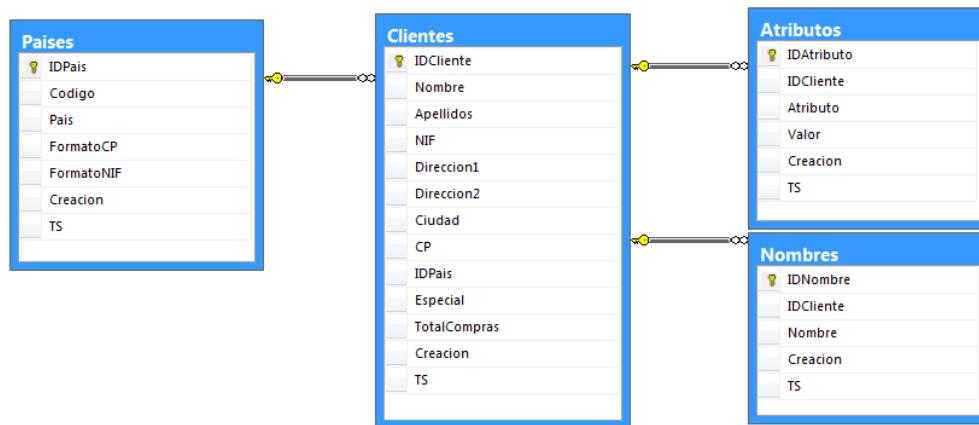
## Técnicas introducidas

Esquemas abiertos, lotes de consultas, propagación de filtros, cláusula **top** en subconsultas

**H**ASTA ESTE MOMENTO, TODAS nuestras ventanas de navegación han mostrado el contenido de tablas aisladas. Es frecuente, sin embargo, que tengamos que navegar sobre una relación maestro/detalles. Dentro de nuestro esquema relacional, ese sería el caso de la ventana de pedidos. Pero comenzaremos por algo más simple: al mostrar los clientes, queremos incluir también la información sobre sus atributos.

## Clients y atributos

Veamos antes qué son estos atributos. Este diagrama muestra las relaciones inmediatas de la tabla de clientes:



Al igual que mantenemos una tabla llamada *Palabras* para facilitar las búsquedas sobre productos, la tabla de clientes se asocia a *Nombres*, que almacena nombres y apellidos por separado. El mantenimiento de *Nombres* es automático, por medio de *triggers*, por lo que podemos olvidarnos de ella sin remordimientos... excepto cuando tengamos que programar búsquedas, claro.

La otra tabla de detalles es *Atributos*, e implementa una técnica que algunos llaman *esquemas abiertos*. Supongo que en algún momento habrá experimentado la angustia existencial que provoca el diseño de una “simple” tabla de clientes. Todos los jefes de departamentos y sus mindundis adjuntos tienen alguna columna que sugerir: desde la secretaria solterona que exige conocer el *EstadoCivil*, hasta el ingeniero en sistemas de eliminación de residuos, también conocido como “el chico de la fregona”, que propone una columna de tipo lógico, *LeGustaElHeavyMetal*. Al final, cuando la aplicación ya está funcionando, descubrimos que nos sobra más de la mitad de las columnas, aunque se nos han olvidado otras que son indispensables para que la aplicación tenga sentido.

La técnica de esquemas abiertos evita estos patinazos de última hora. Al diseñar la tabla de clientes, por ejemplo, sólo incluimos las columnas indispensables, aquellas con las que obligatoriamente tendremos que trabajar. No se trata de quedarnos con dos columnas, pero si tiene dudas sobre el uso de alguna columna, no la incluya en esta tabla. En la base de datos de ejemplos, cada registro de cliente tiene una dirección con el formato habitual. Observe, sin embargo, que no hay direcciones de correo electrónico, ni números de teléfonos, ni faxes, ni fechas de nacimiento...

¿Y qué pasaría si de repente al usuario de la aplicación se le ocurriese que tiene que almacenar las fechas de nacimiento para enviar felicitaciones por cumpleaños (me ocurrió con un proyecto)? En estos casos, muchos usuarios buscan un campo *Comentarios*, de tipo texto, y apuntan estos detalles ahí. Y entonces comienza la pesadilla: ¿cómo buscar los clientes que cumplen años mañana?

Estos son los problemas que resuelve nuestra tabla de atributos. Para cada partida de información que deseemos asociar a un cliente, creamos un registro de atributos. Además de las típicas columnas de identidad y de referencia al cliente, un atributo tiene dos columnas principales: el nombre del atributo y el valor asociado. En nuestra base de datos, todos los clientes tienen un atributo llamado *Teléfono*, y una gran parte de ellos tienen un segundo atributo llamado *Móvil*.

	IDAtributo	IDCliente	Atributo	Valor	Creacion	TS
▶	1	1	Teléfono	91 489 9752	14/12/2007 15:32:42	<Datos binarios>
	128	1	Móvil	663 282 852	14/12/2007 15:32:59	<Datos binarios>
*	NULL	NULL	NULL	NULL	NULL	NULL

◀ ▶ | 1 de 2 | ▶ ▶ ⌂ | Celda de sólo lectura.

Hay infinidad de variantes de la técnica. Por ejemplo:

- Podríamos exigir un conjunto mínimo de atributos para cada cliente. Aparte de estos atributos obligatorios, tendríamos que decidir si permitimos o no atributos adicionales.
- Podríamos definir *clases* de productos, y especificar un conjunto de atributos obligatorios para cada una de las clases. Estos atributos podrían tener reglas asociadas. La clase *Libros* requeriría un atributo ISBN, y los valores de este atributo tendrían que adaptarse a la sintaxis fijada por medio de una expresión regular. Classique, el motor de comercio electrónico desarrollado por IntSight, recibe su nombre precisamente porque utiliza esquemas abiertos basados en clases para la mayoría de las entidades con las que trabaja.

## Primera aproximación

Al diseñar la base de datos no pensamos que hiciera falta quedarnos con el teléfono de cada cliente, pero resulta que estábamos equivocados y hemos tenido que recurrir a los atributos. En tal caso, podríamos modificar la ventana de clientes para que mostrase siempre la siguiente consulta:

```
select Clientes.*, Atributos.Valor as 'Telefono'
from Clientes left outer join Atributos
on Clientes.IDCliente = Atributos.IDCliente
where Atributos.Atributo = 'Teléfono'
```

El *left outer join*, o encuentro exterior por la izquierda, garantiza que si un cliente no tiene un teléfono asociado, de todos modos se incluya el registro de cliente en el resultado; eso sí, con un valor nulo en la columna del teléfono. Debemos tener cuidado, en cualquier caso, con los atributos repetidos: si no tomamos medidas, el registro de cliente aparecería tantas veces como teléfonos se hubiesen asociado al mismo.

Una manera más formal de trabajar con estos encuentros externos consiste en definir una vista:

```
create view XClientes as
select Clientes.*, Atributos.Valor as 'Telefono'
from Clientes left outer join Atributos
on Clientes.IDCliente = Atributos.IDCliente
where Atributos.Atributo = 'Teléfono'
```

El objetivo principal no es encapsular la lectura, sino simplificar la actualización de los registros de la vista. Podemos definir *triggers* del tipo **instead of** para indicar a SQL Server cómo debe interpretar las operaciones de inserción, modificación y borrado. Las modificaciones podrían implementarse de la siguiente manera:

```
create trigger XClientes_u on dbo.XClientes
instead of update as
```

```

begin
    set nocount on

    update dbo.Clientes
    set Nombre = i.Nombre, Apellidos = i.Apellidos,
        NIF = i.NIF,
        Direccion1 = i.Direccion1,
        Direccion2 = i.Direccion2,
        Ciudad = i.Ciudad, CP = i.CP, IDPais = i.IDPais,
        Especial = i.Especial
    from dbo.Clientes c, inserted i
    where c.IDCliente = i.IDCliente

    if update(Teléfono)
    begin
        delete dbo.Atributos
        from dbo.Atributos a, deleted d
        where a.IDCliente = d.IDCliente and
            a.Atributo = 'Teléfono'

        insert into dbo.Atributos(IDCliente, Atributo, Valor)
        select IDCliente, 'Teléfono', Teléfono
        from inserted
    end
end
go

```

**NOTA**

El trigger que acabo de mostrar no es una maravilla de elegancia. Observe que cuando actualizamos la columna Teléfono de XClientes, borramos la fila del atributo correspondiente y luego insertamos una con el nuevo valor. Este extremo puede evitarse con una mejor programación, pero he preferido mantener el trigger lo más sencillo posible.

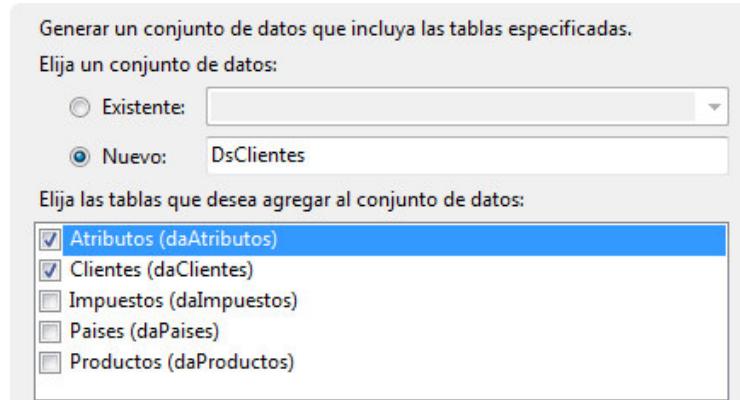
En este ejercicio, por el contrario, utilizaremos una técnica más general... aunque menos vistosa. En vez de trabajar sobre un encuentro externo, vamos a leer clientes y atributos por separado, para ensamblar la relación en la capa de presentación. En una primera aproximación, supondremos que la tabla de clientes no es excesivamente grande, y que podemos traer todos sus registros en una sola lectura sin preocuparnos por la velocidad. De modo que traemos dos adaptadores al módulo de datos, y los configuramos con las siguientes consultas:

```

select * from dbo.Clientes
select * from dbo.Atributos

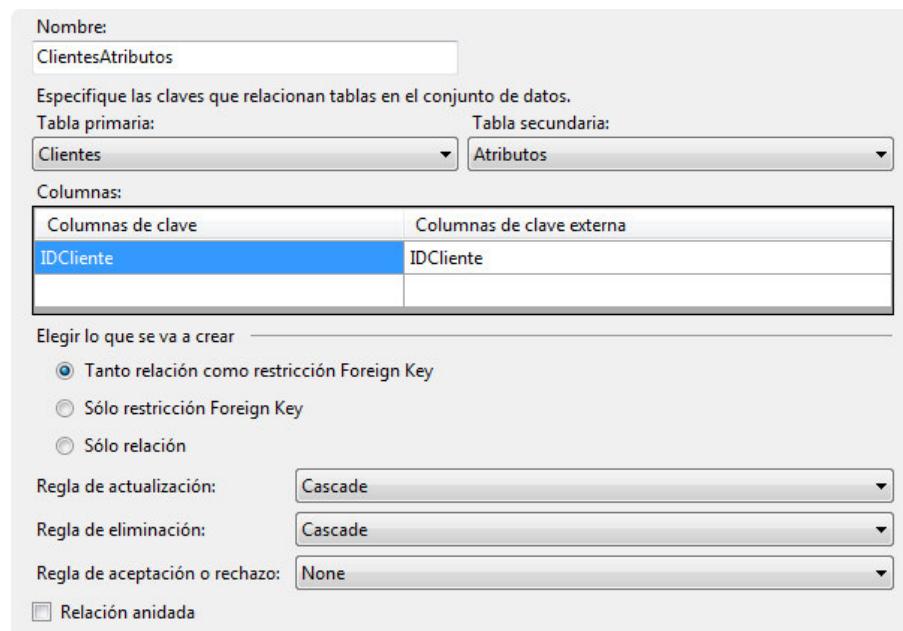
```

Luego creamos un conjunto de datos con tipos a partir de estos dos adaptadores:



A continuación, busque la ventana del *Explorador de soluciones*, y haga doble clic sobre el nodo titulado *DsClientes.xsd* para editar la definición del esquema. El asistente de Visual Studio define correctamente las tablas y sus columnas, pero tenemos que añadir manualmente la relación existente entre las tablas. Seleccione, dentro de la tabla de clientes, la columna *IDCliente*, que es su clave primaria. Arrastre esa columna sobre la columna *IDCliente*, esta vez de la tabla de atributos. Aparecerá el siguiente diálogo, donde terminaremos de configurar la relación:

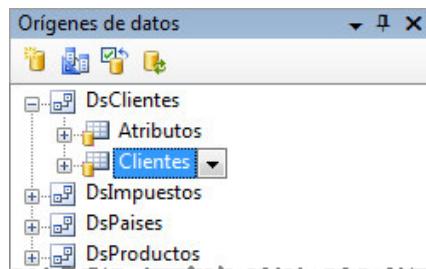
## Ejercicio 13: Navegación maestro/detalles



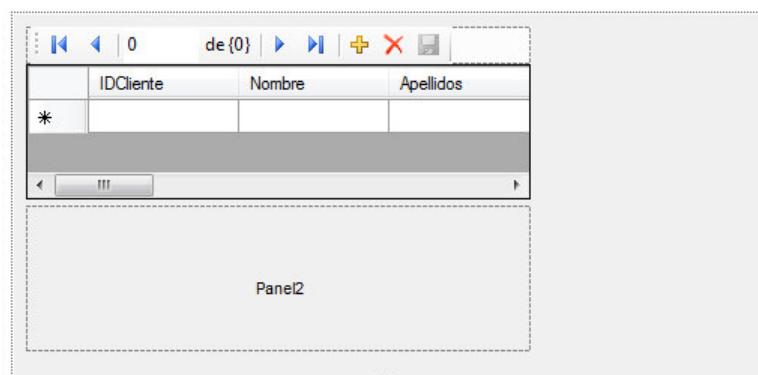
Regrese al código de la clase *Datos*, y añada las siguientes instrucciones dentro del método *Read*:

```
else if (dsname == "DsClientes")
{
    DataSet ds = new DataSet(dsname);
    daClientes.Fill(ds, "Clientes");
    daAtributos.Fill(ds, "Atributos");
    return ds;
}
:
```

Ahora necesitamos una “ventana” de navegación sobre clientes. Siga el procedimiento habitual, derivando una nueva clase a partir del control de usuario *BaseWindow*, y decorando la clase con el atributo *NonModal*, para crear automáticamente un comando en el menú de la ventana principal. Para traer las rejillas necesarias, utilizaremos la ventana Orígenes de datos:



Esta vez, situaremos un *SplitContainer* dentro del área central del *ToolStripContainer*, para ubicar luego la rejilla de clientes en su panel superior:



Para crear la rejilla de atributos, arrastraremos sobre el panel inferior el nodo Atributos que se encuentra dentro del nodo *Clientes* en la ventana de orígenes de datos. Preste atención al detalle: hay que arrastrar el nodo anidando dentro de *Clientes*, no el nodo que se encuentra en el primer nivel, si queremos que funcione la relación maestro/detalles. Ejecute entonces la aplicación y compruebe si está todo en orden.

ID Cliente	Nombre	Apellidos	N.I.F.	Dirección	Dirección	Ciudad	C.P.
1	Irma	Guxot Collado	37043071F	Carretera de Scooby Doo, 362		Madrid	28031
2	Alberto	Alonso Solís	26889213M	Paseo de Pixie y Dixie, 488		Madrid	28671
3	Carlos	Lavandeira Pérez	46584836S	Carrer de Sant Donald, 1582		Barcelona	08170
4	Yolanda	Muñoz Ferrer	31612702F	Av. Pato Lucas, 294		Madrid	28840
5	Sara	Matos Guixot	88503102E	Paseo de la Princesa Leia, 440		Sevilla	41367

ID Atributo	ID Cliente	Atributo	Valor	Creación
1	1	Teléfono	91 489 9752	14/12/2007 15:32
128	1	Móvil	663 282 852	14/12/2007 15:32

## Propagación de filtros

Es demasiado arriesgado asumir que la tabla de clientes se mantendrá en el futuro con un número relativamente pequeño de registros. La suposición va, además, contra las expectativas de cualquier empresa con un poco de amor propio, ¿no? Por lo tanto, consideraremos el intento anterior como una prueba para calentar motores. Debemos leer clientes y atributos en grupos de registros. Ya tenemos una implementación de esta operación y la estamos usando en la ventana de productos. Ahora debemos extender la técnica para poder manejar relaciones maestro/detalles.

Antes de arremangarnos, veamos un par de consideraciones teóricas. Suponga que tenemos una relación como la que existe entre clientes y atributos. A no ser que estos atributos sean opcionales, el tamaño de la tabla de atributos será igual o mayor que el de la tabla maestra. Si teníamos que paginar la tabla de clientes debido a su volumen, por la misma razón tendríamos que dividir la tabla de atributos en fragmentos manejables. Para no perder generalidad, digamos que deseamos aplicar a la tabla maestra una condición de búsqueda que representaremos con la letra griega  $\Theta$ .

```
select *
from TablaMaestra
where Θ
```

Si traemos estos registros al lado cliente, ¿qué registros de detalles son los que tenemos que leer? Resulta que la condición necesaria puede derivarse mecánicamente:

```
select *
from TablaDetalles as td
where td.FKey in (
    select tm.PKey
    from TablaMaestra as tm
    where Θ)
```

La condición, al propagarse a la tabla de detalles, se ha convertido en una subconsulta. Hay, no obstante, un par de elementos nuevos que he destacado en color verde: las columnas *PKey*, de la tabla maestra, y *FKey*, en los detalles. Como sus nombres sugieren, *PKey* debe corresponder a la clave primaria de los registros maestros, y *FKey* es la correspondiente clave externa dentro de la tabla de detalles.

- 89) Hay trampa, por supuesto. Es cierto que en nuestra base de datos todas las claves son simples. Pero, ¿qué pasaría si tuviésemos que tratar con claves compuestas? En ese caso, en vez de recurrir a una subconsulta **in**, tendríamos que usar **exists**:

```
select *
from TablaDetalles as td
where exists (select *
    from TablaMaestra as tm
    where td.FKey1 = tm.PKey1 and
        td.FKey2 = tm.FKey2 and @)
```

Hemos combinado las dos comparaciones que surgen de la integridad referencial con el filtro original que queríamos aplicar a la tabla maestra.

Si la consulta maestra exige un criterio de ordenación o filtra los *n* primeros registros, la consulta de detalles debe incluir también estas condiciones adicionales. Digamos, para volver a nuestro caso, que queremos recuperar una página de clientes que no es la primera:

```
select top 50 *
from TablaMaestra
where IDCliente > @ultimoID
order by IDCliente
```

La correspondiente consulta sobre la tabla de atributos sería:

```
select *
from Atributos
where IDCliente in (
    select top 50 IDCliente
    from Clientes
    where IDCliente > @ultimoID
    order by IDCliente)
```

Estamos pisando territorio minado. En teoría, no deberíamos usar un criterio de ordenación en una subconsulta: se supone que una subconsulta representa una expresión relacional “pura”. Pero la cláusula **top** no tiene sentido sin el criterio de ordenación... y lo más importante: SQL Server nos permite estas libertades. La expresión sería algo más complicada si tuviésemos claves compuestas (otro argumento en contra de las claves primarias compuestas!). En tal caso, tendríamos que transformar la subconsulta basada en **in** en una subconsulta **exists**. Y ahí comienzan las dificultades:

- Dentro de una subconsulta **exists** no tiene sentido alguno emplear **order by** y **top**.

Para resolver ese caso, tendríamos que recurrir a otro tipo de consulta anidada: las *tablas derivadas*.

```
select *
from Atributos
where exists (
    select *
    from (select top 50 IDCliente
          from Clientes
          where IDCliente > @ultimoID
          order by IDCliente) as Clientes
    where Atributos.IDCliente = Clientes.IDCliente)
```

Esta vez hemos anidado una consulta dentro de la cláusula **from**, como si el resultado de la subconsulta se hubiese almacenado en una tabla “física” temporal.

## Extensiones del sistema de paginación

Nuestro esquema relacional ha sido diseñado para evitar algunas de las situaciones patológicas que acabamos de ver. Gracias a ello, nos resultará sencillo extender el mecanismo existente para recuperar registros en grupos mediante la generación de consultas en tiempo de ejecución.

Lo primero que debemos hacer es añadir tres campos nuevos a la clase *PageCookie*, que hemos definido dentro del fichero *DataServer.cs*:

```
// Nuevos campos en PageCookie
public readonly string DetailsName;
public readonly string MasterField;
```

```
public readonly string DetailsField;
```

El primer campo, *DetailsName*, almacenará el nombre de la tabla de detalles. Si este campo contiene una cadena vacía, asumiremos que se trata de una consulta sobre una sola tabla. *MasterField*, por su parte, es el nombre de la columna de la tabla maestra que actúa como clave primaria, y *DetailsField* es la columna, dentro de la tabla de detalles, que apunta al registro maestro.

**NOTA**

En realidad, *PageCookie* define dos campos para señalar la tabla maestra: *TableName*, que es siempre el nombre de la tabla, y *Table*, que puede ser también el nombre de la tabla o una llamada a una función de tablas. En cambio, no necesitamos dos columnas para la tabla de detalles porque, al menos por ahora, no utilizaremos funciones de tablas para este papel.

Al existir nuevos campos que deben ser inicializados, crearemos dos constructores: uno que configure una búsqueda sobre una sola tabla, y otro que contemple la posibilidad de consultar entidades relacionadas entre sí.

```
public PageCookie(string tablename, string table,
                  string filter, string sort, int count) { ... }
public PageCookie(string tablename, string master,
                  string filter, string sort, int count,
                  string details, string masterfield, string detailsfield) { ... }
```

Dentro de la misma clase *PageCookie*, tenemos que extender ahora el mecanismo de generación de consultas SQL. Comenzaremos definiendo un método protegido auxiliar:

```
protected void Generate(StringBuilder sb,
                       int count, string fields, string table, string filter, string sort)
{
    sb.Append("select top ").Append(count).Append(" ").Append(fields);
    sb.Append(" from ").Append(table);
    if (filter != "")
    {
        sb.Append(" where ").Append(filter);
    }
    sb.Append(" order by ").Append(sort);
}
```

Este método se utiliza dos veces dentro de la implementación de la propiedad *SQL* de *PageCookie*:

```
public string SQL
{
    get
    {
        System.Text.StringBuilder sb = new System.Text.StringBuilder();
        Generate(sb, Count, "*", Table, GetFilter(), Sort);
        if (DetailsName != "")
        {
            sb.Append("; \nselect * from ");
            sb.Append(" where ").Append(DetailsField);
            sb.Append(" in (");
            Generate(sb, Count, MasterField, Table, GetFilter(), Sort);
            sb.Append(")");
        }
        return sb.ToString();
    }
}
```

Como puede ver, la propiedad *PageCookie.SQL* devuelve ahora *dos* consultas (por el precio de una). Eso significa que vamos a configurar un adaptador con un lote de dos consultas, para leer ambos conjuntos de resultado en una sola operación.

Una vez que hemos terminado con *PageCookie*, y ahora tenemos que abrir el fichero *Data.cs*, para propagar los cambios a la clase *ReturnedCookie*, que desciende de *PageCookie*. Los cambios son muy sencillos porque la clase, al heredar de *PageCookie*, ya contiene los campos añadidos, y sólo nos resta retocar sus constructores:

```

public ReturnedCookie(PageCookie source, string startFrom):
    base(source.TableName, source.Table,
          source.Filter, source.Sort, source.Count,
          source.DetailsName, source.MasterField, source.DetailsField)
{
    this.startFrom = startFrom;
}

public ReturnedCookie(PageCookie source):
    base(source.TableName, source.Table,
          source.Filter, source.Sort, source.Count,
          source.DetailsName, source.MasterField, source.DetailsField)
{
    this.startFrom = "1 = 0";
    this.disabled = true;
}

```

Si examina el código fuente resultante, comprobará que la propiedad SQL, aplicada indistintamente sobre *PageCookie* o *ReturnedCookie*, ya devuelve los resultados que deseábamos. Sólo necesitamos un pequeño retoque dentro del método *Read* que trabaja con *PageCookie*:

```

public System.Data.DataSet Read(ref PageCookie cookie)
{
    using (SqlDataAdapter da = new SqlDataAdapter(cookie.SQL, sqlConn))
    {
        da.TableMappings.Add("Table", cookie.TableName);
        if (cookie.DetailsName != "")
            da.TableMappings.Add("Table1", cookie.DetailsName);
        DataSet ds = new DataSet();
        da.Fill(ds);
        DataTable master = ds.Tables[cookie.TableName];
        if (master.Rows.Count < cookie.Count)
            cookie = new ReturnedCookie(cookie);
        else
            cookie = new ReturnedCookie(cookie,
                                         SynthesizeExpression(master, cookie.Sort));
        return ds;
    }
}

```

El código resaltado añade una nueva entrada dentro de la propiedad *TableMappings* del adaptador de datos. Cuando un adaptador lee un lote de consultas, las tablas que crea dentro del conjunto de datos donde volcamos los resultados se nombran *Table*, *Table1*, *Table2*... Al añadir entradas a *TableMappings*, estamos indicando al adaptador el nombre que queremos darle a las tablas generadas durante la ejecución del método *Fill*.

Volvamos a la “ventana” de clientes. Primero añadimos un campo llamado *cookie* dentro de la clase:

```
private PageCookie cookie = null;
```

A continuación, cambiamos la implementación al método heredado *InitData*:

```

public override void InitData(IDataWindow parent, bool newRow)
{
    cookie = new PageCookie("Clientes", "Clientes",
                           "", "Nombre,Apellidos,ICliente", 20,
                           "Atributos", "IDCliente", "IDCliente");
    dsClientes.Merge(Data.Instance.Read(ref cookie), false);
    bnMasRegistros.Enabled = !cookie.Disabled;
}

```

Para terminar, al igual que hicimos en la “ventana” de productos, añadimos un botón a la barra de herramientas, *bnMasRegistros*, para recuperar el siguiente grupo de registros a petición del usuario:

```

private void bnMasRegistros_Click(object sender, EventArgs e)
{
    if (!cookie.Disabled)
    {

```

```

        dsClientes.Merge(Data.Instance.Read(ref cookie), false);
        bnMasRegistros.Enabled = !cookie.Disabled;
    }
}

```

El resultado final debe parecerse a la siguiente imagen:

ID Cliente	Nombre	Apellidos	N.I.F.	Dirección	Dirección	Ciudad	C.P.
124	Adrián	Hernández Vázquez	46169272S	123 Fake Street		Springfield	26505
120	Aitor	Laguna Echaniz	30064128T	Plaza Darth Vader, 75		Sevilla	41988
2	Alberto	Alonso Solís	26889213M	Paseo de Pixie y Dixie, 488		Madrid	28671
90	Angel	Casal Fuentes	60291671T	C/ Lucas Skywalker, 564		Sevilla	41341
115	Angel	Matos Flores	51143771M	Carretera de Scooby Doo, 816		Madrid	28774

ID Atributo	ID Cliente	Atributo	Valor	Creación			
124	124	Teléfono	636 251 0404	14/12/2007 15:32			
216	124	Móvil	679 118 264	14/12/2007 15:32			

## El primer paso hacia Proteus

Es hora de ir extrayendo algunas conclusiones iniciales sobre la metodología de desarrollo que estamos aplicando, al menos en lo que respecta a las técnicas de acceso a datos. Hemos diseñado una interfaz genérica para la lectura y escritura de conjuntos de datos. Es una interfaz pequeña, con sólo tres métodos... e incluso puede que nos sobre uno de los dos métodos de lectura.

Los dos métodos de lectura son los siguientes:

```

public DataSet Read(string dsname);
public DataSet Read(PageCookie cookie);

```

La implementación del primero de ellos exige la configuración por adelantado de los adaptadores de datos necesarios para satisfacer la petición. En cambio, el segundo método no requiere configuración previa, y además permite la recuperación por páginas de registros, incluso cuando se trata de una relación maestro/detalles. La segunda versión es más potente, más fácil de usar y, para ser consecuentes, debería sustituir definitivamente a la primera.

Sin embargo, al menos hasta el momento, la configuración de las modificaciones ha sido completamente manual. Hemos tenido que configurar cuidadosamente los adaptadores necesarios, porque las instrucciones generadas por Visual Studio no se adaptan a nuestras necesidades. Y sobre todo, hemos especificado explícitamente el orden de grabación entre los adaptadores y las versiones de registros modificados.

¿Adónde debería llevarnos la extrapolación de las técnicas que hemos visto? Estos son algunos requisitos que consideramos deseables para una interfaz general de acceso a datos:

- 1 Debe ser posible recuperar conjuntos de datos definidos dinámicamente, incluyendo la posibilidad de trabajar con relaciones maestro/detalles en más de un nivel.
- 2 Deberíamos ser capaces de deducir el algoritmo de grabación a partir de la misma especificación utilizada para la lectura. El algoritmo debería tener en cuenta las principales variantes de implementación del bloqueo optimista, las peculiaridades de la actualización de determinados tipos de columnas y, sobre todo, ofrecer una implementación flexible y robusta para la recuperación y tratamiento de las violaciones del bloqueo optimista.
- 3 Sería bueno poder indicar que determinadas tablas se lean a partir de una copia local, o que se mantuviesen al menos en una caché en memoria. Si la capa de acceso a datos se mueve a un servidor de capa intermedia independiente, es menos importante disponer de una copia local, y puede que nos baste con una copia en memoria, en el espacio de procesos del servidor de capa intermedia.

- 4** Por último, deberíamos poder especificar, en algunos casos, consultas basadas en encuentros naturales y externos, sin necesidad de tener que usar una vista para ello.

Todos estos aspectos, y unos cuantos más, son resueltos por Proteus, el motor de capa intermedia genérico desarrollado en Intuitive Sight como parte de un proyecto de investigación.

---

**EJERCICIO PROPUESTO**

Al igual que hicimos con la tabla de *Productos*, implemente un mecanismo de búsqueda de clientes a través de los nombres almacenados en la tabla *Nombres*.

---

# EJERCICIO 14

## Objetivos

Búsquedas basadas en atributos

## Técnicas introducidas

Enumeradores

**S**I EN EL EJERCICIO ANTERIOR VIMOS cómo mostrar los atributos asociados a cada cliente, en este ejercicio veremos cómo implementar una búsqueda de clientes que también permita interrogar la tabla de atributos.

## Sintaxis de las cadenas de búsqueda

Lo correcto sería diseñar algún tipo de ventana de búsqueda amigable, con muchos mensajes e ilustraciones para que el usuario sepa lo que tiene que hacer para encontrar lo que desea. Incluso estoy dispuesto a añadir algún carácter animado, siempre que no salga respondón como cierto chuchito que conozco:



Nosotros, por el contrario, lo haremos siguiendo el estilo de los 70s: definiremos un lenguaje en miniatura para la búsqueda de clientes. Así nos ahorraremos el trajín con los componentes visuales. De todas maneras, una búsqueda basada en asistentes visuales puede implementarse como capa de presentación de un mecanismo basado en texto como el que mostraré a continuación. De modo que comenzaremos por definir la sintaxis de las cadenas de búsqueda que admitiremos.

En primer lugar, al igual que hicimos con la tabla de productos, mantendremos la posibilidad de indicar una o más palabras claves; en este caso, no se trata de palabras, sino de “nombres”, en sentido general. Además, seguiremos considerando que hay una conjunción implícita para unir las condiciones generadas. Por ejemplo, la siguiente cadena serviría para buscar las *Maria García*, no las *Marias* o las *Garcias*.

maria garcia

Recuerde que preferimos las conjunciones porque así cada nueva palabra sirve para estrechar la búsqueda. Por cierto, la cadena anterior también localizaría a *José María Martínez García*, porque se busca cada nombre sin importar su posición.

Como novedad, permitiremos búsquedas por los valores de cualquier columna de la tabla *Clientes*. El usuario podría teclear algo parecido a esto:

Ciudad=Sevilla

y la aplicación respondería como puede imaginar. En teoría, incluso podríamos deducir el nombre de la columna con sólo examinar ciertos patrones de valores. Si recibimos la siguiente cadena:

66047341N

podríamos inferir que se trata de un número de identificación fiscal español. Pero se supone que nuestra base de datos contiene datos internacionales, y es casi imposible decidir de antemano si determinada cadena representa un número de identificación fiscal o un código postal plausible. Por ese motivo no implementaremos esta característica.

Finalmente, podremos incluir una consulta por el valor de los atributos guardados en la tabla del mismo nombre:

```
maria telefono=636
```

Esta consulta busca las *Marias* cuyo número de teléfono comienza por el *636*. Observe que no he acentuado la palabra *teléfono*. Eso indica que la búsqueda sobre la tabla de atributos ignorará acentos y, por supuesto, las diferencias entre mayúsculas y minúsculas.

## Enumeradores

Para dividir la cadena de búsqueda en fragmentos, y de paso practicar un poco con C#, vamos a implementar un *iterador*, que podremos usar con una instrucción **foreach** para recibir secuencialmente cada una de las partes constituyentes de la cadena. Primero debemos indicar qué significa la palabra “fragmento”. Pues bien, esta clase es la respuesta:

```
public class Token
{
    public readonly string Param;
    public readonly string Value;

    public Token(string param, string value)
    {
        Param = param;
        Value = value;
    }

    public Token(string value)
    {
        Param = KEYWORD;
        Value = value;
    }
}
```

Es decir, que consideraremos como “fragmentos” subcadenas como las siguientes:

```
maria
telefono=636
nif=X
```

Cada fragmento está formado por un nombre de parámetro y un valor asociado. En el caso de los nombres sueltos, imaginaremos que se trata de una abreviatura para un parámetro llamado *keyword*.

Necesitamos ahora una clase que almacene la cadena de búsqueda y que implemente el llamado *patrón de enumeración* de C#. Esta clase, que en el ejercicio hemos situado dentro de un fichero llamado *traductor.cs*, es la siguiente:

```
public class Traductor
{
    public const string KEYWORD = "keyword";
    protected string input;

    public Traductor(string input)
    {
        this.input = input;
    }

    public string Input
    {
        get { return input; }
    }

    public IEnumarator<Token> GetEnumerator()
    {
        // ... paciencia ...
    }
}
```

```
:  
}
```

**NOTA**

En realidad, la clase `Token` que hemos presentado es una clase anidada dentro de la clase `Traductor`, y deberíamos referirnos a ella, si estamos fuera de la clase `Traductor`, con su nombre completamente cualificado: `Traductor.Token`. Al recurrir a una clase anidada, evitamos tener demasiados tipos de datos en el primer nivel de declaraciones.

El famoso patrón de enumeración, a partir de C# 2.0, se reduce a que la clase debe implementar un método que se llame `GetEnumerator` y que devuelva un valor del tipo genérico `IEnumerator`: en este caso, necesitamos que sea exactamente `IEnumerator<Token>`. El método aislará todos los fragmentos dentro de la cadena, y cada vez que encuentre alguno, ejecutará la instrucción `yield return`, que “cederá” el control al bucle `foreach` desde donde utilizaremos el enumerador:

```
public IEnumerator<Token> GetEnumerator()  
{  
    for (int i = 0; ; )  
    {  
        while (i < input.Length && Char.IsWhiteSpace(input, i))  
            i++;  
        if (i >= input.Length)  
            yield break;  
  
        int start = i;  
        while (i < input.Length && !Char.IsWhiteSpace(input, i) &&  
               input[i] != ':' && input[i] != '=')  
            i++;  
        string s1 = input.Substring(start, i - start);  
  
        while (i < input.Length && Char.IsWhiteSpace(input, i))  
            i++;  
        if (i < input.Length && (input[i] == ':' || input[i] == '='))  
        {  
            i++;  
            while (i < input.Length && Char.IsWhiteSpace(input, i))  
                i++;  
            int start1 = i;  
            while (i < input.Length && !Char.IsWhiteSpace(input, i) &&  
                   input[i] != ':' && input[i] != '=')  
                i++;  
            yield return new Token(  
                s1, input.Substring(start1, i - start1));  
        }  
        else  
            yield return new Token(s1);  
    }  
}
```

La instrucción alternativa `yield break`, que utilizamos dentro del bucle más externo, simplemente indica que ya no hay más fragmentos y que la iteración ha terminado. El método es largo y laborioso, pero creo que se comprenderá sin problemas.

Si le intriga saber cómo vamos a usar la clase `Traductor`, aquí está la respuesta:

```
foreach (Traductor.Token token in new Traductor(txBuscar.Text))  
    MessageBox.Show(token.Param + " = " + token.Value);
```

Al definir un iterador, podemos acceder secuencialmente a cada uno de los fragmentos con sentido de la cadena, como si estuviésemos trabajando de manera directa con una colección o un vector de fragmentos.

---

**EJERCICIO PROPUESTO**

Sería bueno permitir cadenas de caracteres encerradas entre comillas o doble comillas, al menos detrás de un signo de igualdad. En cambio, no tendría mucho sentido permitir espacios dentro de valores sin un nombre explícito de parámetro, porque estos valores se utilizan para buscar nombres, y los nombres almacenados en la tabla correspondiente no tienen espacios intermedios.

## Generación de consultas

Sólo nos queda saber qué debemos hacer cada vez que recibimos un fragmento. Estas son las reglas que necesitamos, extraídas de la definición informal del inicio del ejercicio:

- Para cada *token* de tipo *keyword*, se generará una subconsulta **in** sobre la tabla de palabras.
- Para los *tokens* que no son de tipo *keyword*, se buscará un nombre de columna que coincida con el nombre del parámetro, o un sinónimo de nombre de columna. Si se encuentra, se generará una comparación simple basada en el operador **like**.
- En caso contrario, se creará una subconsulta sobre la tabla de atributos.
- Todas estas condiciones parciales se combinarán con la ayuda del operador **and**.

Nos vamos a la ventana de navegación sobre clientes, pues ahí es donde usaremos el iterador recién creado y generaremos un filtro para la tabla de clientes. Comenzamos por definir algunas constantes de cadenas, que nos servirán de plantillas durante este proceso:

```
private const string TPL_COLUMNAS =
    "{0} like '{1}%'";
private const string TPL_ATRIBUTOS =
    "IDCliente in (select IDCliente from Atributos " +
    "where Atributo = {0} collate modern_spanish_ci_ai and " +
    "Valor like '{1}%' collate modern_spanish_ci_ai)";
private const string TPL_NOMBRES =
    "IDCliente in (select IDCliente from Nombres " +
    "where Nombre like '{0}%' collate modern_spanish_ci_ai);
```

Preste atención al uso de la cláusula **collate**, para indicar en cuáles comparaciones se deben ignorar mayúsculas, minúsculas y acentos. También definiremos una estructura para indicar los nombres que consideraremos que son nombres de columnas, y algunos sinónimos aceptable. Para ello, definiremos un campo de sólo lectura como una matriz, o vector de dos dimensiones, de entradas de tipo **string**:

```
private readonly string[,] ATTRNAMES = new string[,] {
    {"id", "IDCliente"}, {"nombre", "Nombre"}, {"apellidos", "Apellidos"}, {"apellido", "Apellido"}, {"nif", "NIF"}, {"direccion", "Direccion1"}, {"direccion1", "Direccion1"}, {"dir", "Direccion1"}, {"direccion2", "Direccion2"}, {"cp", "CP"}, {"ciudad", "Ciudad"}, {"pais", "Pais"}, {"pais", "Pais"}};
```

Le confieso que es la primera vez que utilizo un vector multidimensional en C#. De hecho, esta es una característica más en la que C# da vueltas en círculos alrededor de Java. La estructura anterior es utilizada por el método auxiliar *ConsultaAtributos*:

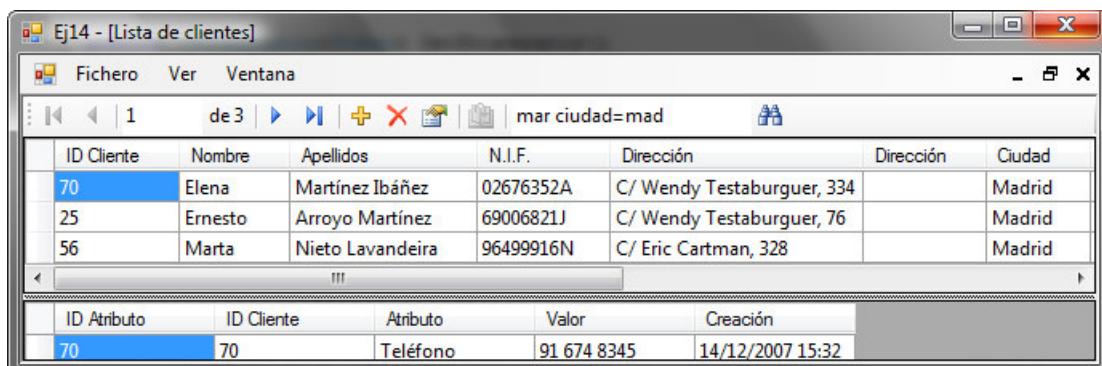
```
private void ConsultaAtributos(StringBuilder sb,
    string param, string value)
{
    for (int i = 0; i < ATTRNAMES.GetLength(0); i++)
        if (String.Compare(param, ATTRNAMES[i, 0], true) == 0)
        {
            sb.AppendFormat(TPL_COLUMNAS, ATTRNAMES[i, 1], value);
            return;
        }
    sb.AppendFormat(TPL_ATRIBUTOS, param.Replace("''", "'''"), value);
}
```

*ConsultaAtributos* busca un nombre de columna o un sinónimo que coincide con el nombre del parámetro que nos pasan en el segundo parámetro. Si triunfa, utiliza la plantilla más simple de comparación directa. En caso contrario, asumimos que hay que realizar la búsqueda sobre la tabla de atributos. Observe que estamos creando la consulta con la ayuda de un *StringBuilder*, para evitar la creación innecesaria de cadenas con resultados intermedios.

Finalmente, podemos ya implementar la respuesta al clic sobre el botón de búsqueda:

## Ejercicio 14: Búsquedas basadas en atributos

```
private void bnBuscar_Click(object sender, EventArgs e)
{
    StringBuilder sb = new StringBuilder();
    foreach (Traductor.Token token in new Traductor(txBuscar.Text))
    {
        string s = token.Value.Replace("!!", "!!");
        if (sb.Length > 0)
            sb.Append(" and ");
        if (token.Param == Traductor.KEYWORD)
            sb.AppendFormat(TPL_NOMBRES, s);
        else
            ConsultaAtributos(sb, token.Param, s);
    }
    string filter = sb.ToString();
    if (filter != "")
    {
        cookie = new PageCookie("Clientes", "Clientes",
            filter, "Nombre,Apellidos,IDCliente", 20,
            "Atributos", "IDCliente", "IDCliente");
        dsClientes.Clear();
        dsClientes.Merge(Data.Instance.Read(ref cookie), false);
        dsClientes.AcceptChanges();
        bnMasRegistros.Enabled = !cookie.Disabled;
        txBuscar.SelectAll();
        txBuscar.Focus();
    }
}
```



### EJERCICIO PROUESTO

Más que una implementación, le propondré un ejercicio de diseño: ¿cómo puede extenderse nuestro pequeño lenguaje de consulta para permitir disyunciones entre condiciones? Dicho de otra manera, queremos poder buscar los clientes que se llaman *María* o *Margarita*, pero no *Marcela*, en una misma consulta.

# EJERCICIO 15

## Objetivos

Resolución de referencias

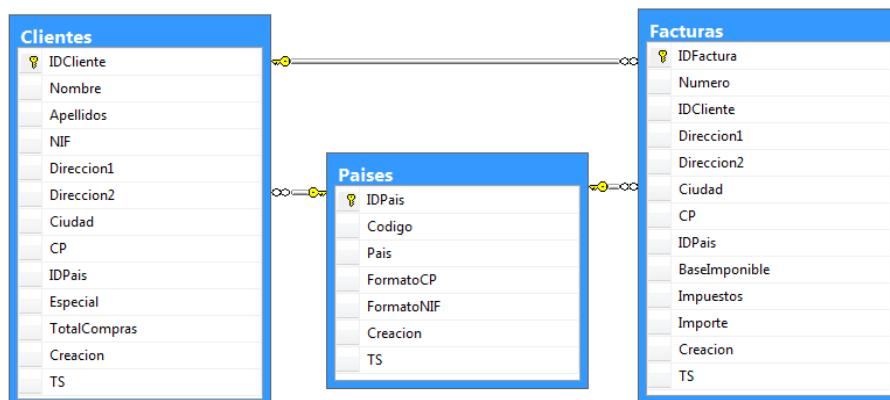
## Técnicas introducidas

Columnas calculadas, relaciones, personalización de columnas de rejillas

Incluso si echa un vistazo superficial al estado actual de la aplicación, se dará cuenta de un problema que hemos estado esquivando hasta ahora. En la rejilla de navegación de clientes, por ejemplo, mostramos un código numérico de país que no significa nada para nadie. En la de productos, hay un identificar de impuesto similar. Peor aún: para modificar o crear un producto o un cliente, hay que utilizar el código de impuesto o de país. En este ejercicio resolveremos ambas limitaciones.

## Columnas basadas en expresiones

¿Por qué surgen estas situaciones? Si nos atenemos a la información contenida en las restricciones de integridad referencial, no hay diferencia entre la relación que vincula clientes y pedidos, y la que existe entre países y clientes:



... bueno, sí existe una diferencia: la dirección de la relación. La tabla *Clientes* es una tabla de detalles respecto a la tabla de países, pero por necesidades de la aplicación, la utilizamos como tabla *raíz* en una de las ventanas de navegación.

En estos casos, hay un truco que nos permitiría mostrar el nombre del país asociado a cada cliente:

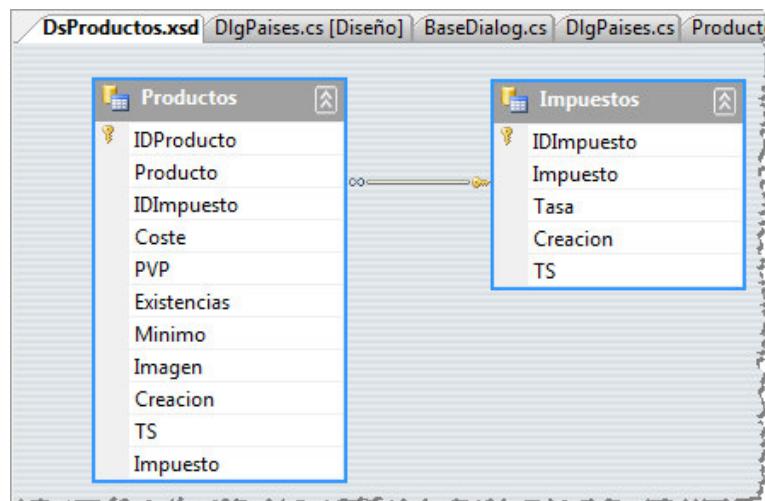
- Traer, en el mismo conjunto de datos donde se encuentra la tabla de clientes, la tabla de países.
- Crear un objeto de relación que vincule países y clientes. Supondremos que la relación se llama *Paises.Clientes*.
- Crear una columna calculada en la tabla de clientes, basada en la siguiente expresión:

`Parent(Paises.Clientes).País`

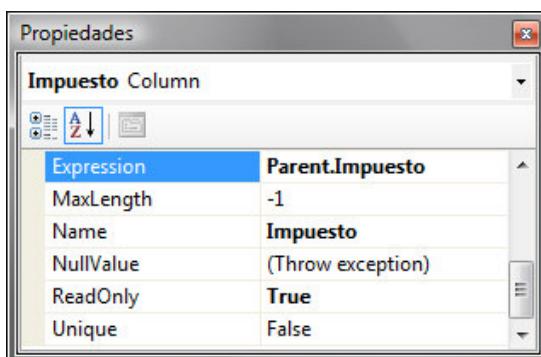
- 209 En esta expresión, *Parent* se utiliza para buscar la fila maestra asociada a la fila para la que se evalúa la columna. Si existe más de una relación maestra, tenemos que indicar su nombre entre paréntesis. Luego, pedimos el valor de la columna país de esa fila maestra... que ha resultado ser una fila de la tabla de países. Si sólo quisieramos ver el nombre del país asociado a cada cliente, o el valor de cualquier otra columna de la tabla de países, con esto nos bastaría. Por supuesto, la columna calculada no sería directamente modificable, pero podríamos utilizar otra técnica, que presentaré más adelante, para editar el país dentro de un registro de cliente.

Voy a mostrarle cómo resolver el problema, pero comenzaré por la tabla de productos, porque aún nos quedan un par de detalles antes de poder editar registros de clientes, y quiero mostrar tanto la navegación como la edición de una tabla con referencias externas. Los cambios comenzarán en el conjunto de datos con tipos: haga doble clic sobre el nodo *DsImpuestos*, en el Explorador de

Servidores. No, no me he equivocado: vamos a seleccionar la tabla de impuestos y a copiarla en el portapapeles de Windows. Ahora sí haremos doble clic sobre *DsProductos*, y una vez en el editor, pegaremos la tabla de impuestos dentro del conjunto de datos de productos:



Como muestra la imagen, tenemos que crear una relación entre las columnas *IDImpuesto* de ambas tablas. A continuación, añadiremos una columna a *Productos*, la llamaremos *Impuesto*, y la configuraremos de la siguiente manera:



La expresión utilizada ha sido *Parent.Impuesto*: no es necesario indicar cuál es la relación maestra, pues hay una solamente. Observe que también he asignado *True* en *ReadOnly*, por si a Visual Studio le queda alguna duda sobre la posibilidad de modificar directamente la nueva columna.

## El orden de las lecturas

Cuando hay relaciones por medio, hay que ser muy cuidadosos con el orden de las lecturas. Normalmente, al crear la relación entre tablas, se crea también la restricción de integridad referencial. Esto implica que, si leemos un producto antes de leer la tabla de impuestos, se producirá una violación de la restricción, al no existir todavía la fila de referencia. Es verdad que podemos desactivar la restricción, pero creo que merece la pena seguir las reglas del juego.

De momento, hay dos puntos en los que tenemos que realizar algún ajuste. En la ventana *Productos*, debemos modificar la implementación de *InitData*:

```
public override void InitData(IDataWindow parent, bool newRow)
{
    dsProductos.Merge(Data.Instance.Read("DsImpuestos"));
    cookie = new PageCookie(
        "Productos", "Productos", "", "Producto", IDProducto", 20);
    dsProductos.Merge(Data.Instance.Read(ref cookie), false);
    bnMasRegistros.Enabled = !cookie.Disabled;
}
```

Hemos añadido una instrucción para leer la tabla de impuestos y mezclar sus registros dentro del conjunto de datos de productos... antes de leer el primer producto, claro. También tenemos que retocar la respuesta al botón de búsqueda, en la misma clase:

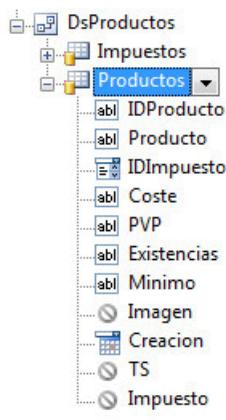
```
private void bnBuscar_Click(object sender, System.EventArgs e)
{
    System.Text.RegularExpressions.Regex regex =
        new System.Text.RegularExpressions.Regex("[^A-Za-z0-9%_]+");
    string s = "";
    int count = 0;
    foreach (string word in regex.Split(txBuscar.Text.Trim()))
    {
        s += "!" + word.Replace("!", "!!") + "%!";
    }
    if ((++count) == 3)
        break;
}
if (s != "")
{
    cookie = new PageCookie("Productos",
        "Productos0" + count.ToString() + "(" +
        s.Remove(s.Length - 2, 2) + ")",
        "", "Producto", IDProducto", 20);
    dsProductos.Productos.Clear();
    dsProductos.Merge(Data.Instance.Read(ref cookie), false);
    dsProductos.AcceptChanges();
    bnMasRegistros.Enabled = !cookie.Disabled;
}
```

La instrucción resaltada limpia la tabla de productos, mientras que antes limpiábamos todo el conjunto de datos. Tenga presente que, en circunstancias reales, es más que probable que mantuviésemos una caché local para la tabla de impuestos. La técnica sería casi idéntica, en cualquier caso.

Para terminar, edite la columna de la rejilla que muestra *IDImpuesto*, para que muestre *Impuesto*, la columna calculada que acabamos de añadir a la tabla de productos.

## Combos para la edición de referencias

Hemos resuelto el problema de la visualización de columnas de referencia, pero nos queda ver cómo hacer para facilitar su edición. La solución que daremos en este ejercicio exige que la tabla de referencias sea lo suficientemente pequeña para que no nos importe leer todos sus registros en memoria. Cuando se trata de traducir identificadores de países o impuestos, esta exigencia no es un obstáculo.

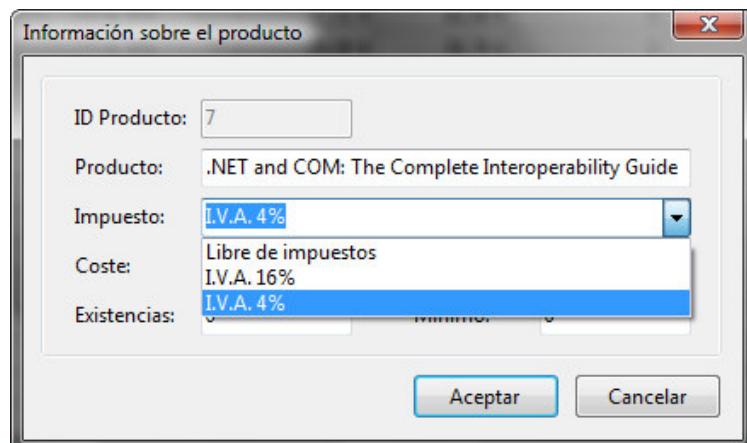


Para crear un diálogo de edición de productos, necesitamos introducir un par de cambios en el origen de datos que Visual Studio ha creado automáticamente para el conjunto de datos *DsProductos*. En primer lugar, tenemos que desplegar la lista asociada a la tabla de productos, e indicar que queremos crear una vista de detalles. De esta manera, cuando arrastremos y soltemos el nodo *Productos* sobre la superficie del diálogo, se crearán controles individuales, en vez de la clásica rejilla.

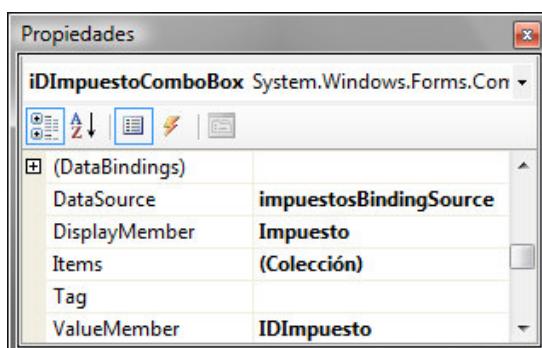
Los cambios restantes afectan a las columnas de esta misma tabla. Primero tenemos que seleccionar *Impuesto*, la columna calculada que hemos creado en este ejercicio, desplegar su lista de selección de controles, e indicar que no queremos control alguno para esta columna. Luego seleccionaremos *IDImpuesto*, la columna original de referencia, y en su lista de selección de controles elegiremos un combo.

Ahora ya podemos arrastrar y soltar *Productos* sobre un diálogo derivado de *BaseDialog*. Como podrá comprobar, Visual Studio creará un combo para editar el impuesto asociado al producto. Este combo, sin embargo, estará incompleto: le faltan los datos para el enlace con la tabla de impuestos.

Por fortuna, Visual Studio nos ofrece un atajo para esta tediosa operación: arrastre la tabla de Impuestos, la que se encuentra dentro del mismo conjunto de datos *DsProductos*, desde la ventana Orígenes de datos, y déjela caer sobre el combo. ¿A que da gusto trabajar así?



Con este simple gesto, hemos traído un *BindingSource* adicional para la tabla de impuestos, y hemos configurado tres propiedades del combo:



Necesitamos ahora garantizar la lectura de la tabla de impuestos, en el orden correcto. Sólo tenemos que redefinir el método *InitData*:

```
public override void InitData(IDataWindow parent, bool newRow)
{
    dsProductos.Impuestos.Merge(parent.DataSet.Tables["Impuestos"]);
    base.InitData(parent, newRow);
}
```

Le aviso que, en vez de utilizar la propiedad *Tables* del conjunto de datos genérico para localizar la tabla de impuestos en la ventana de navegación, podíamos haber efectuado una conversión de tipos utilizando la clase *DsProductos*. Pero al final, el trabajo necesario era más o menos el mismo.

### EJERCICIO PROPUESTO

Repita el truco con el conjunto de datos *DsClientes* y la ventana de clientes, para traer la descripción del país correspondiente a la dirección de cada cliente.

# EJERCICIO 16

## Objetivos

Edición de entidades con esquemas abiertos

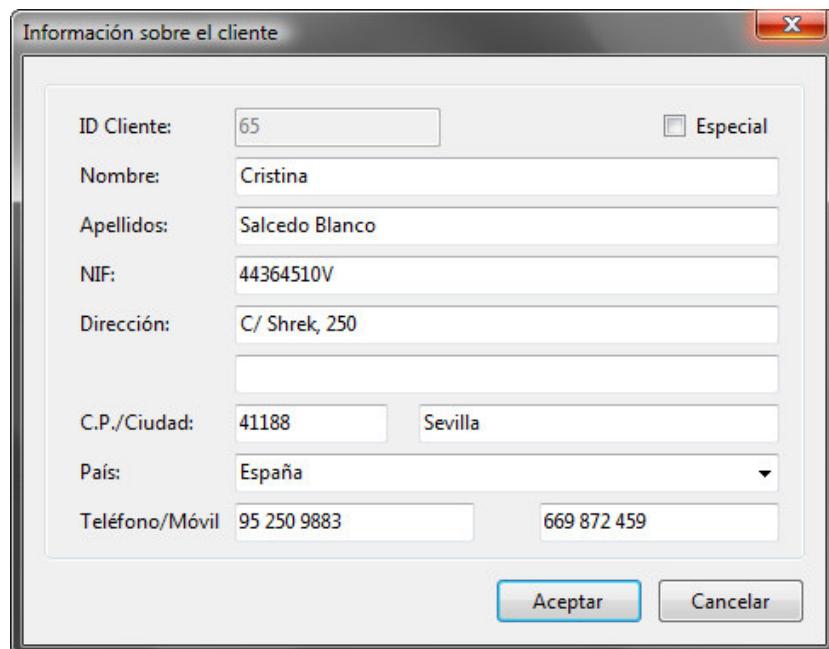
## Técnicas introducidas

Enlace manual a datos, transacciones, la interfaz *IDisposable*

**L**A EDICIÓN DE REGISTROS DE CLIENTES, además de presentar el mismo problema de resolución de referencias que la edición de productos, plantea una dificultad adicional: recuerde que nuestros clientes son entidades de esquema abierto, y que junto con el registro de cliente, debemos editar los registros de atributos relacionados. Además de mostrar una técnica adecuada, aprovecharemos para presentar la primera grabación que involucra explícitamente registros provenientes de más de una tabla.

## Enlace a datos artesanal

¿Qué hay de complicado en la edición de clientes y atributos? Si decidísemos editar las columnas de la tabla *Clientes* por medio de controles individuales, y poner todos los registros de atributos en una rejilla de datos, no habría más que hablar... aunque sería un poco más incómodo para el usuario, al menos bajo ciertas circunstancias. Por ejemplo, si fuese obligatorio que todos los clientes tuvieran atributos para los números de teléfono y de móvil (o celular), ¿para qué tratar visualmente estos atributos como si fueran opcionales? Mejor sería reconocer esta necesidad y ofrecer también controles individuales para dichos atributos:



Los cuadros de texto asociados al teléfono y al móvil no pueden usar la técnica ya habitual del enlace a datos en tiempo de diseño, sino que tendremos que ocuparnos manualmente de asignarles valores al comienzo de la edición, y de trasladar los cambios en ellos al conjunto de datos, si se quiere grabar el resultado de la edición.

Para inicializar los nuevos controles, redefiniremos la implementación del método *InitData*:

```
private System.Data.DataView attrView = null;  
  
public override void InitData(IDataWindow parent, bool newRow)  
{  
    dsClientes.Merge(Data.Instance.Read("DsPaises"), false);  
    base.InitData(parent, newRow);
```

```
if (!newRow)
{
    foreach (DataRow r in
        ((DataRowView)parent.BindingSource.Current)
        .Row.GetChildRows("ClientesAtributos"))
        dsClientes.Atributos.ImportRow(r);
    dsClientes.AcceptChanges();

    attrView = ((DataRowView)clientesBindingSource.Current)
        .CreateChildView("ClientesAtributos");
    attrView.Sort = "Atributo";
    int idx = attrView.Find("Teléfono");
    if (idx >= 0)
        txPhone.Text = attrView[idx]["Valor"].ToString();
    idx = attrView.Find("Móvil");
    if (idx >= 0)
        txMobile.Text = attrView[idx]["Valor"].ToString();
    txPhone.Modified = false;
    txMobile.Modified = false;
}
}
```

Primero leemos la tabla de referencia, leyendo el conjunto de datos de países. Recuerde que esta tabla está almacenada en una caché local, y que esta lectura es realmente eficiente. A continuación llamamos a la implementación heredada, que se ocupará de copiar la fila del cliente, en el caso de modificación, o de crear una nueva fila, en el caso de una inserción.

Supongamos ahora que estamos modificando un registro ya existente. Como el método heredado sólo ha copiado la fila maestra, algo tendremos que hacer para copiar los registros de detalles, es decir, los atributos del cliente:

```
foreach (DataRow r in
    ((DataRowView)parent.BindingSource.Current)
    .Row.GetChildRows("ClientesAtributos"))
    dsClientes.Atributos.ImportRow(r);
dsClientes.AcceptChanges();
```

El método *ImportRow*, de la clase *DataTable*, copia todas las columnas de otra fila con la misma estructura, y deja la nueva fila en el mismo estado que la fila original. La llamada a *AcceptChanges* sirve para marcar el punto de partida de la edición: cualquier cambio a partir de este momento se considerará como un cambio a grabar en la base de datos.

Sabemos que cuando editamos una fila de una tabla, los controles están trabajando en realidad con un objeto de la clase *DataRowView*, insertado dentro de un *DataView*. Esta distinción se complica cuando se trata de editar los detalles asociados a esa fila particular. En ese caso necesitamos acceder a una vista de datos dinámica, que se crea mediante una llamada al método *CreateChildView*:

```
attrView = ((DataRowView)clientesBindingSource.Current)
    .CreateChildView("ClientesAtributos");
```

Necesitamos esta vista dependiente por otro motivo adicional: tenemos que buscar filas de atributos partiendo del nombre del atributo, y la columna con el nombre del atributo *no es* la clave primaria de la tabla de atributos. Por ese motivo tenemos que modificar la propiedad *Sort* de la vista dependiente antes de poder realizar dicha búsqueda:

```
attrView.Sort = "Atributo";
int idx = attrView.Find("Teléfono");
if (idx >= 0)
    txPhone.Text = attrView[idx]["Valor"].ToString();
```

Con esto resolvemos la inicialización de los controles correspondientes a los atributos que se nos ocurra que deben estar siempre presentes. Pero nos queda ver cómo transferimos los valores tecleados en los controles anteriores al conjunto de datos, cuando decidamos guardar los cambios. Como el algoritmo de grabación va precedido de una consulta al valor de la propiedad *IsModified*

proveniente de la interfaz *IWindow*, podemos injectar el código necesario redefiniendo la implementación de dicha propiedad:

```
public override bool IsModified
{
    get
    {
        FlushControls();
        return base.IsModified;
    }
}
```

La implementación de *FlushControls* es la siguiente:

```
protected void FlushControls()
{
    if (txPhone.Modified)
    {
        ModifyAttribute("Teléfono", txPhone.Text.Trim());
        txPhone.Modified = false;
    }
    if (txMobile.Modified)
    {
        ModifyAttribute("Móvil", txMobile.Text.Trim());
        txMobile.Modified = false;
    }
}
```

Como puede ver, el código importante está encapsulado en el siguiente método auxiliar:

```
protected void ModifyAttribute(string attribute, string value)
{
    DataRowView rowview = (DataRowView)clientesBindingSource.Current;
    if (rowview.IsNew)
        rowview.EndEdit();
    int idx = attrView.Find(attribute);
    if (!String.IsNullOrEmpty(value))
        if (idx >= 0)
            attrView[idx]["Valor"] = value;
        else {
            DataRowView r = attrView.AddNew();
            try {
                r["Atributo"] = attribute;
                r["Valor"] = value;
                r.EndEdit();
            }
            catch {
                r.CancelEdit();
                throw;
            }
        }
    else if (idx >= 0)
        attrView[idx].Delete();
}
```

Entendemos que, cuando hay una cadena vacía para un atributo, podemos optimizar su almacenamiento eliminando la fila correspondiente al atributo, si es que existe. Podríamos simplificarlo todo dejando viva la fila, pero con una cadena vacía en la columna del valor.

## Transacciones

Regresaremos enseguida al diálogo de edición de clientes, pero tenemos que ocuparnos ahora de la grabación de conjuntos de datos con más de una tabla. ¿La diferencia? Que antes teníamos que grabar un solo registro, y ahora nos enfrentamos a la grabación de más de un registro en una misma operación. Es cierto que, incluso si el conjunto de datos tiene una sola tabla, podríamos tener varios registros modificados para grabar. Pero la técnica de edición que hemos venido usando nos ha

permitido aislar las modificaciones de manera que sólo actualizábamos un registro a la vez. Con la tabla de clientes, se acabaron estas facilidades.

Cuando hay más de un registro en cola para ser guardado, es imprescindible proteger toda la operación dentro de una transacción, de manera que, o se puedan grabar todos los registros pendientes, o no se produzca la grabación parcial de algunos. Todo o nada. Si intentamos usar directamente las clases de transacciones que nos ofrece ADO.NET, nos encontraremos con mucho código estúpido y repetido que teclear. Estas son algunas molestias que quisiéramos paliar:

- Un objeto de transacción es siempre un objeto temporal. Una vez que aprobamos o cancelamos la transacción, podemos tirar el objeto: no existe un método para “reiniciar” la transacción aprovechando un mismo objeto.
- Sin embargo, los comandos que se deben ejecutar dentro de la transacción deben configurarse para que hagan referencia a la misma, por medio de la propiedad *Transaction*. Hay que crear el objeto de transacción, asignarlo en la propiedad *Transaction* de todos los comandos que tengamos que ejecutar. Finalmente, deberíamos asignar un puntero nulo en esos mismos objetos al terminar la transacción, para que la instancia de ésta pueda ser reciclada por el colector de basura.
- El código que inicia la transacción y la confirma o anula, casi siempre debe ir protegido por una instrucción **try/catch**. Eso no tendría mayor importancia... si no fuese porque también debemos garantizar que la conexión correspondiente esté abierta. Eso nos obliga a proteger la apertura de la conexión dentro de un bloque **try/finally**. Dos bloques anidados de manejo de excepciones, una y otra vez.

Para facilitarnos las cosas, vamos a crear una clase auxiliar que nos permitirá simplificar el código necesario para iniciar una transacción e inicializar con ella los comandos que entrarán en juego. La clase implementará el tipo de interfaz *IDisposable* para poder usar sus instancias en una instrucción **using** de C#. Como debe saber, esta instrucción nos permite simplificar ciertos tipos de bloques de excepciones, del tipo **try/finally**, garantizándonos la ejecución del método *Dispose* de la instancia creada en la cabecera de la instrucción.

Vamos a declarar e implementar la siguiente clase en un fichero nuevo, *Transacciones.cs*:

```
public sealed class Transaccion : IDisposable
{
    private SqlConnection connection;
    private SqlTransaction transaction;
    private SqlDataAdapter[] adapters;
    bool committed;

    public Transaccion(SqlConnection connection,
        params SqlDataAdapter[] adapters)
    {
        this.committed = false;
        this.connection = connection;
        this.adapters = adapters;
        connection.Open();
        transaction = connection.BeginTransaction(
            IsolationLevel.Serializable);
        Assign(transaction);
    }

    void IDisposable.Dispose()
    {
        if (!committed)
            transaction.Rollback();
        committed = true;
    }
    Assign(null);
    if ((connection.State & ConnectionState.Open) != 0)
        connection.Close();
}
```

```

public void Commit()
{
    transaction.Commit();
    committed = true;
}

public void Rollback()
{
    transaction.Rollback();
    committed = true;
}

private void Assign(SqlTransaction trans)
{
    foreach (SqlDataAdapter ad in adapters)
    {
        if (ad.UpdateCommand != null)
            ad.UpdateCommand.Transaction = trans;
        if (ad.InsertCommand != null)
            ad.InsertCommand.Transaction = trans;
        if (ad.DeleteCommand != null)
            ad.DeleteCommand.Transaction = trans;
    }
}
}

```

El constructor de *Transaccion* recibe una conexión y una lista de adaptadores de datos. Como efecto secundario, debe abrir la conexión, crear una transacción y almacenarla internamente. Además, debe recorrer la lista de adaptadores para asignar la transacción en todos sus comandos, con la ayuda del método auxiliar *Assign*. El método *Dispose* de la clase, que suponemos que se ejecutará automáticamente gracias a la instrucción **using**, comprobará si la transacción sigue activa, consultando el valor de un campo interno llamado *committed*. Este campo recibe el valor **true** si cerramos explícitamente la transacción llamando a uno de los métodos *Commit* o *Rollback* de la clase *Transaccion*. Si olvidamos hacerlo, *Dispose* se encargará de ello anulando la transacción. Finalmente borrará la referencia a la difunta transacción de los adaptadores indicados en el constructor y cerrará la conexión. Enseguida veremos cómo usar esta clase tan servicial.

**NOTA**

Me he resistido a la tentación de simplificar el código de control de transacciones echando mano al nuevo sistema de transacciones de ADO.NET para SQL Server 2005 y posterior. El nuevo sistema, descrito en la Serie C del Volumen I del curso, nos evitaría, de entrada, tener que asignar explícitamente la transacción a cada comando involucrado en la operación. De hecho, desaparecería toda la clase *Transaccion*. He vuelto a usar el antiguo sistema para que le sirva como referencia si tiene que trabajar con otros servidores que no admitan ámbitos de transacciones. No obstante, si está siguiendo el curso con SQL Server 2005, puede modificar el ejercicio y usar transacciones declarativas, como alternativa.

## Grabación maestro/detalles

Al igual que hicimos con las tablas de productos, impuestos y países, tendremos que relajar algunas de las restricciones a nivel de columnas de la tabla de clientes para poder grabar estos registros. Active el editor del conjunto de datos *DsClientes*, y realice estos cambios en sus columnas:

- El valor por omisión de *Especial* debe ser *False*.
- El valor por omisión de *TotalCompras* debe ser *0*.
- Asigne *False* en la propiedad *AllowDBNull* de *Creacion*.
- Realice el mismo cambio en la columna *Creacion* de la tabla de atributos.

Para la grabación en sí, añadimos una llamada dentro del método *Write* de la clase *Data*:

```

else if (dsname == "DsClientes")
    return GuardarClientes(delta);

```

Y el método *GuardarClientes* debe parecerse a éste:

```
protected DataSet GuardarClientes(DataSet delta)
{
    using (Transaccion trans =
        new Transaccion(sqlConn, daClientes, daAtributos))
    {
        DataTable attrs = delta.Tables["Atributos"];
        daAtributos.Update(attrs.Select("", "", 
            DataViewRowState.Deleted));
        daClientes.Update(delta.Tables["Clientes"]);
        daAtributos.Update(attrs.Select("", "", 
            DataViewRowState.Added | DataViewRowState.ModifiedCurrent));
        trans.Commit();
        return delta;
    }
}
```

---

### EJERCICIO PROPUESTO

Observe cómo creamos un objeto *Transaccion* y nos aseguramos de su “destrucción” mediante la instrucción **using**. Como ejercicio, puede intentar programar este método sin la clase auxiliar, para que compruebe cómo su longitud crecería, junto con su complejidad.

---

- 348 Lo más importante dentro de *GuardarClientes*, no obstante, es el orden en que se realizan las actualizaciones. Primero nos ocupamos de los atributos borrados, a continuación se actualizan los registros de clientes sin importar el tipo de actualización, y sólo al final grabamos las inserciones y modificaciones de atributos.

## Propagando los cambios a la ventana de navegación

No podemos olvidar que, lo mismo si se trata de un nuevo cliente o de un cliente ya existente que acabamos de modificar, el diálogo de edición trabaja siempre con una copia, no con el conjunto de datos originalmente utilizado en la ventana de navegación. De modo que al terminar con el diálogo, si todo ha salido bien, hay que copiar los cambios en el conjunto de datos original.

Tendremos que realizar algunos cambios en la versión del método *SaveChanges* que ofrece *BaseDialog* como punto de partida para sus descendientes:

```
public virtual bool SaveChanges()
{
    DataSet delta = Data.Instance.Write(
        this.DataSet.DataSetName, this.DataSet.GetChanges());
    foreach (DataTable tab in this.DataSet.Tables)
        foreach (DataRow row in tab.Select(
            null, null, DataViewRowState.Added))
            row.Delete();
    this.DataSet.Merge(delta, false);
    this.DataSet.AcceptChanges();
    parentWindow.DataSet.Merge(delta, false);
    parentWindow.DataSet.AcceptChanges();
    return true;
}
```

El cambio ha sido resaltado y es muy simple: en vez de mezclar sobre el conjunto de datos original el contenido de la tabla local del diálogo, como hemos hecho hasta ahora, mezclaremos *todo* el contenido del conjunto de datos diferencial, es decir, el conjunto al que hace referencia la variable local *delta*.

Por supuesto, necesitamos también algunas modificaciones específicas para la grabación de clientes. Redefiniremos la implementación de *SaveChanges* en *DlgCliente* para añadir, al principio del método, instrucciones para sincronizar las dos tablas de atributos: la original y la copia que modificamos dentro del diálogo.

```
public override bool SaveChanges()
{
    DataTable attrs = parentWindow.DataSet.Tables["Atributos"];
    parentWindow.DataSet.AcceptChanges();
    parentWindow.DataSet.Merge(delta, false);
    parentWindow.DataSet.AcceptChanges();
    parentWindow.DataSet.Merge(delta, false);
    parentWindow.DataSet.AcceptChanges();
    return true;
}
```

```
foreach (DataRow row in dsClientes.Atributos.Select(
    "", "", DataViewRowState.Deleted))
{
    DataRow target = attrs.Rows.Find(
        row["IDAtributo", DataRowVersion.Original]);
    if (target != null)
        target.Delete();
}
return base.SaveChanges();
}
```

En concreto, si eliminamos un atributo en la copia, tendremos que eliminarlo manualmente del conjunto de datos original. Solo entonces transferimos el control a la versión heredada del método *SaveChanges*.

---

### EJERCICIO PROPUESTO

Hay un riesgo importante en la implementación de *SaveChanges*: si ocurriese un error durante la grabación, y hubiésemos eliminado un atributo, tendríamos que restaurar los registros de atributos que antes habríamos eliminado de la copia usada para la navegación. ¿Qué tal si practica un poco su C#?

---

# EJERCICIO 17

## Objetivos

Validación

## Técnicas introducidas

Expresiones regulares, relación entre *ErrorProvider* y conjuntos de datos, trampas de errores

CUANDO PRESENTÉ LA TABLA DE PAÍSES, EXPLIQUÉ que mantendríamos dos columnas con expresiones regulares: una para el formato de los códigos postales y la otra para la identificación fiscal de cada país. Sería una tontería habernos tomado tantas molestias para que ahora no aprovechásemos estas dos columnas para validar los registros de clientes.

## Más responsabilidades para *IWindow*

Hay muchas validaciones que pueden implementarse en el propio servidor SQL. Pero esta es sólo la última parada del proceso de actualización de datos. Muchas veces nos conviene duplicar, o incluso implementar por primera y única vez, algunas de las validaciones necesarias en la capa intermedia o en la de presentación. Siempre que estas validaciones se mantengan en sincronía con lo que exige la aplicación o el servidor SQL, es muy recomendable comprobar nuestros registros antes de enviarlos a través de la red, para evitar tráfico innecesario sobre la misma.

Las validaciones que veremos en este ejercicio se refieren, precisamente, a validaciones realizadas en la capa de presentación, asociadas a la edición mediante diálogos modales. Es cierto que podemos incluir comprobaciones como parte del proceso de grabación, en el módulo de datos representado en nuestros ejercicios por la clase *Data*. Pero muchas de las verificaciones deben tener acceso a los controles de edición para que el diagnóstico del error que se le ofrece al usuario sea realmente útil.

Para tener en cuenta este tipo de validación, vamos a añadir otro método a la interfaz *IWindow*:

```
public interface IWindow
{
    // Inicialización
    void InitData(IDataWindow parent, bool newRow);
    // Métodos de control de cambios
    bool IsModified { get; }
    bool ValidateData();
    bool SaveChanges();
    void DiscardChanges();
    // Métodos de restauración del diseño
    void SaveLayout();
    void RestoreLayout();
}
```

Recuerde que uno de los principios más importantes de la buena programación es que las “conversaciones” entre módulos se realicen en público y en voz alta. Si dos módulos, en el sentido genérico del concepto, interactúan de algo manera, ¡haga que el intercambio sea explícito!, ¡anúncielo en algún lugar! La nueva declaración de *IWindow* grita a los cuatro vientos que una “ventana” puede validar su contenido en algún lugar o momento.

Ahora veremos quién aprovecha esta nueva capacidad de *IWindow*: el método *ModalFormClosing* del gestor de ventanas, que se usa para dar respuesta al evento *Closing* de los diálogos de edición:

```
private void ModalFormClosing(
    object sender, System.ComponentModel.CancelEventArgs e)
{
    IWindow wnd = sender as IWindow;
    if (wnd.IsModified)
        if ((sender as Form).DialogResult == DialogResult.OK)
        {
            if (!wnd.ValidateData() || !wnd.SaveChanges())
                e.Cancel = true;
        }
}
```

```

        else if (MessageBox.Show(
            "Ha realizado cambios en el registro.\n" +
            "¿Desea realmente abandonarlos?", "Advertencia",
            MessageBoxButtons.YesNo,
            MessageBoxIcon.Question) == DialogResult.Yes)
            wnd.DiscardChanges();
        else
            e.Cancel = true;
    }
}

```

La implementación por omisión del nuevo método en las clases *BaseDialog* y *BaseWindow* es neutral:

```

public virtual bool ValidateData()
{
    return true;
}

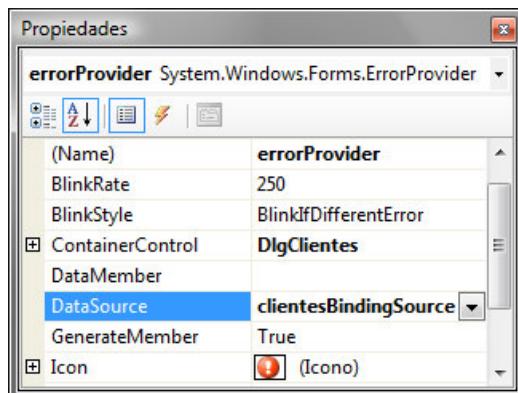
```

Todos los datos sometidos al examen de este *ValidateData* por omisión superarán el examen.

## El proveedor de errores

En el caso de la tabla de clientes, debemos verificar el contenido de dos columnas: el código fiscal del cliente y el código postal de la dirección asociada al mismo. Esta validación no debería realizarse en el momento de la edición de la columna, como hicimos con las propias expresiones regulares en la tabla de países. Como la verificación depende del país seleccionado, no es buena idea adelantar la comprobación mientras no sepamos definitivamente cuál país se asociará al cliente. Por lo tanto, es buena idea dejar la comprobación en manos del nuevo método *ValidateData*.

Para hacer saber al usuario dónde se ha equivocado, añadiremos un componente *ErrorProvider* al diálogo de clientes; ya hicimos algo parecido en el diálogo de edición de países. Pero esta vez habrá una pequeña diferencia:



Vamos a asociar el proveedor de errores a la vista de datos que se modifica dentro del diálogo; para concretar, asignaremos una referencia a *clientesBindingSource* en la propiedad *DataSource* del proveedor de errores. Con esta técnica, en vez de tener que asociar errores directamente a los controles de edición afectados, algo que nos exigiría dar nombres descriptivos a estos controles, podremos asociar el error a las columnas del conjunto de datos, para que sea el proveedor de errores quien busque el control enlazado a dicha columna.

Esto quiere decir que también debemos indicar al *ErrorProvider* cuál es el control donde debe buscar los controles enlazados. Como muestra la imagen anterior, debemos cambiar la propiedad *ContainerControl* del proveedor para que apunte al propio formulario.

## Verificación mediante expresiones regulares

Tenemos que redefinir ahora la implementación de *ValidateData* dentro de la clase *DlgCliente*. Como vamos a usar métodos estáticos de la clase *RegEx*, es buena idea añadir el espacio de nombres donde reside esta clase en una cláusula **using**, al inicio del fichero:

```
using System.Text.RegularExpressions;
```

Esta es la implementación que necesitamos:

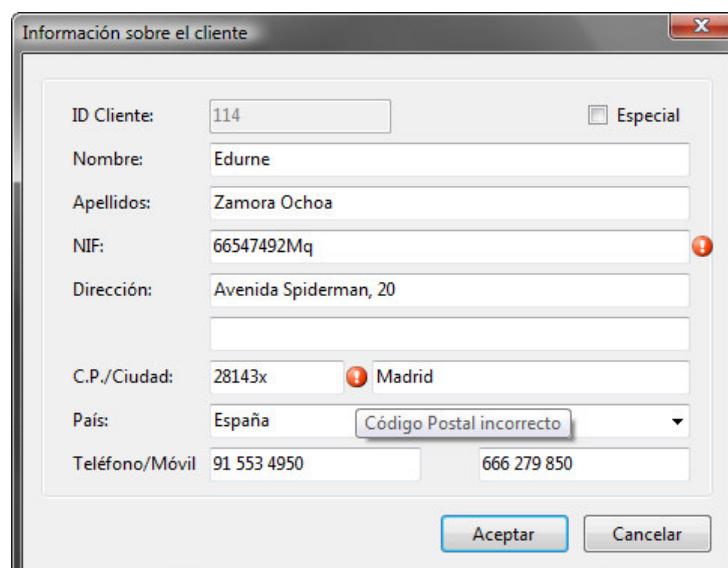
```
public override bool ValidateData()
{
    DsClientes.ClientesRow cliente = dsClientes.Clientes[0];
    cliente.ClearErrors();
    DsClientes.PaisesRow pais =
        dsClientes.Paises.FindByIDPais(cliente.IDPais);
    if (pais != null) {
        if (pais.FormatoCP != "" &&
            !Regex.IsMatch(cliente.CP, pais.FormatoCP))
            cliente.SetColumnError("CP", "Código Postal incorrecto");
        if (pais.FormatoNIF != "" &&
            !Regex.IsMatch(cliente.NIF, pais.FormatoNIF))
            cliente.SetColumnError("NIF", "N.I.F. incorrecto");
    }
    return !cliente.HasErrors;
}
```

A primera vista, parece que hay un error potencialmente peligroso en el método. Estoy usando el conjunto de datos *dsClientes* que hemos añadido al diálogo al crearlo por primera vez, para localizar la primera fila de la tabla *Clientes* y realizar las comprobaciones sobre esta fila. Ahora bien, ¿no habíamos quedado en que el diálogo realmente actuaba sobre un *DataRowView*, antes de enviar los cambios propuestos a la fila definitiva? Es cierto... pero tenga en cuenta que, antes de llamar a *ValidateData*, el manejador del evento *Closing* del formulario consulta el valor de *IsModified*, y que nuestra implementación de este método sincroniza los valores del *DataRowView* con los de la fila correspondiente. Tenemos, además, un motivo convincente para acceder a la fila: como estamos trabajando con un conjunto de datos con tipos, la fila correspondiente pertenece a la clase anidada *ClientesRow*, y en vez de vernos obligados a acceder a sus columnas mediante indizadores, podemos utilizar las propiedades generadas por el asistente de creación de conjuntos de datos con tipos.

El código de verificación intenta localizar, en primer lugar, la fila del país al que pertenece el cliente. Con esta fila en la mano, las comprobaciones se parecen a esto:

```
if (pais.FormatoCP != "" &&
    !Regex.IsMatch(cliente.CP, pais.FormatoCP))
    cliente.SetColumnError("CP", "Código Postal incorrecto");
```

Si detectamos un error, utilizamos el método *SetColumnError*, heredado por *ClientesRow* de la clase base *DataRow*, para establecer el mensaje de error. Como hemos asociado un proveedor de errores a la vista de datos, éste se encarga de buscar los controles de edición enlazados a dicha columna para colocar el icono de advertencia a su lado.



## Trampas de errores

¿Código postal incorrecto? ¿Sólo... “eso”? Si su usuario se parece a los míos, le garantizó que habrá una persona maldiciendo por la estúpida aplicación programada por un estúpido programador que ni siquiera saber reconocer cuando un código postal es correcto o incorrecto... porque vamos a ver... ¿qué demonios no le gusta en el código postal para el que tantas molestias me he tomado al teclear? Si no damos un diagnóstico más exacto de este tipo de errores, tendremos que oír lamentos como éste, y aún peores.

El hecho es que no conozco algoritmo alguno para generar un mensaje más exacto cuando una cadena no concuerda con una expresión regular. Puede que exista, pero repito: no lo conozco. Una técnica que puede mejorar el diagnóstico consiste en usar señuelos para la captura de errores: en vez de establecer qué es lo correcto, ponga ejemplos sobre qué es incorrecto. Para esto necesitaríamos, por cada expresión regular, una lista de pares <patrón de error; mensaje de error>. Por ejemplo:

//	La expresión correcta	
	<code>^[0-9]{5}\$</code>	
//	Trampa de error	Descripción
	<code>^[0-9]{0,4}\$</code>	Faltan dígitos
	<code>[0-9]{6,}</code>	Demasiados dígitos
	<code>[^0-9]</code>	Sólo se aceptan dígitos

Si falla la verificación, se recorre la lista de trampas, y se muestra la descripción asociada a la primera trampa que concuerde con el valor tecleado. Si se recorren todas las trampas y ninguna tiene éxito, se muestra un mensaje de error genérico. Observe que algunas de las trampas comprueban el inicio y fin de líneas, y otras no. También podríamos haber añadido una trampa trivial al final de la lista, para que el mensaje “genérico” sea el más adecuado para la columna que estamos verificando.

El problema principal de esta técnica es la dificultad para almacenar toda la lista de trampas de errores. Podríamos mantener la lista dentro de una columna de tipo **text**, y designar un separador para poder distinguir dónde acaba cada expresión regular y comienza la descripción del error asociado.

# EJERCICIO 18

## Objetivos

Selección de registros en tablas grandes

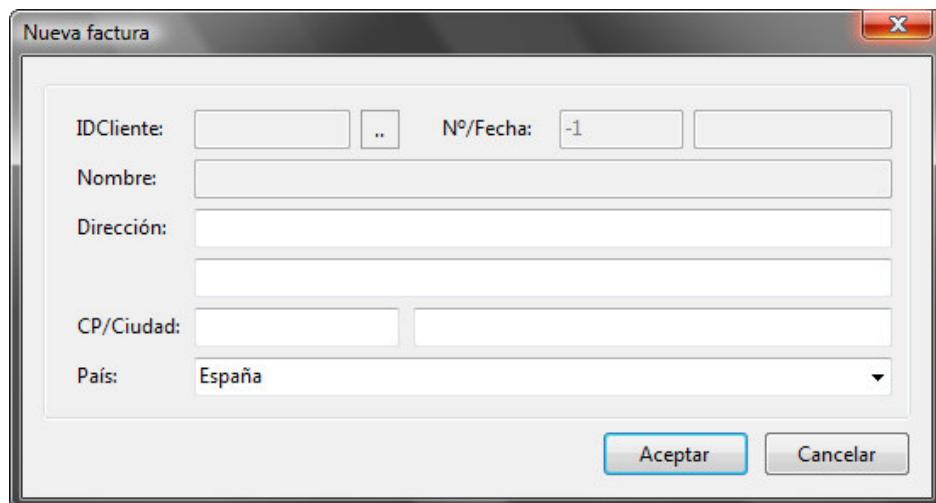
## Técnicas introducidas

Atributos, manejo de registros mediante *DataRowView*

**P**ODEMOS LLENAR LA LISTA DESPLEGABLE DE UN combo con los nombres de todos los países del planeta Tierra, y aún nos quedaría espacio para algún que otro país jupiterino con el que mantengamos relaciones comerciales. No sería sensato, en contraste, pretender que el usuario seleccione un cliente para una factura por medio de un combo. Para empezar, porque si las cosas nos van bien, tendremos una tabla de clientes suficientemente grande. Pero también porque sería muy incómodo buscar un cliente determinado usando una lista desplegable que sólo puede mostrar un número reducido de filas cada vez.

## Creación de facturas y selección de clientes

... y la misma situación se repetiría al intentar seleccionar productos para las líneas de la factura. En estos casos, la técnica de edición de referencias debe cambiar. Este es el aspecto del diálogo de altas de facturas que vamos a crear en este ejercicio:



Nos falta el mecanismo que permitirá la edición de las líneas de facturas, pero para simplificar, en este ejercicio sólo nos ocuparemos de la cabecera. Como muestra la imagen, hemos situado un botón al lado del control que corresponde al identificador del cliente, y hemos añadido un control adicional redundante, que muestra información sobre el cliente seleccionado para la factura. Ese control, como puede imaginar, no podrá ser modificado directamente por el usuario.

Para seleccionar un cliente, el usuario deberá pulsar el botón mencionado. Entonces aparecerá un segundo diálogo modal con una rejilla en su interior, con la ayuda de la cual podrá buscar y seleccionar un cliente, usando las mismas técnicas que en la ventana de navegación. Si el usuario finalmente pulsa el botón de aceptar, la selección se propaga al formulario original.

Para crear y configurar este diálogo, primero necesitaremos tener adaptadores de datos para facturas y líneas (obviaremos de momento la tabla de datos o atributos de facturas), y el conjunto de datos con tipos correspondiente. Esta será la consulta del nuevo adaptador de facturas:

```
select Facturas.*, Clientes.Nombre, Clientes.Apellidos, Clientes.NIF
from Facturas inner join Clientes
on Facturas.IDCliente = Clientes.IDCliente
where Facturas.IDFactura = @IDFactura
```

Otra vez tenemos una consulta para recuperar un único registro. Al igual que en los casos anteriores, la navegación sobre la tabla de facturas se realizará con la ayuda de la versión de *Read* que utiliza *cookies* de páginas. Pero en este ejercicio activaremos directamente el diálogo de altas desde el menú principal, por lo que tampoco debemos preocuparnos en exceso. Observe, eso sí, que al registro de factura leído se le añaden, mediante un encuentro natural, los datos más importantes sobre el cliente asociado a la factura.

De momento, nos interesan más las instrucciones de actualización. Esta es la instrucción para insertar cabeceras de facturas:

```
insert into Facturas (Numero, IDCliente, Direccion1, Direccion2,
                      Ciudad, CP, IDPais, BaseImponible, Impuestos, Importe)
values (-1, @IDCliente, @Direccion1, @Direccion2,
        @Ciudad, @CP, @IDPais, @BaseImponible, @Impuestos, @Importe);
select IDFactura, Numero, Creacion, TS
from Facturas
where IDFactura = scope_identity()
```

La actualización de una cabecera tendrá lugar de la siguiente manera:

```
update Facturas
set IDCliente = @IDCliente, Direccion1 = @Direccion1,
    Direccion2 = @Direccion2, Ciudad = @Ciudad, CP = @CP,
    IDPais = @IDPais, BaseImponible = @BaseImponible,
    Impuestos = @Impuestos, Importe = @Importe
where IDFactura = @Original_IDFactura and TS = @Original_TS;
select TS
from Facturas
where IDFactura = @Original_IDFactura
```

Y esta la instrucción para borrar cabeceras de facturas:

```
delete from Facturas
where IDFactura = @Original_IDFactura and TS = @Original_TS
```

Como en este ejercicio sólo insertaremos cabeceras, podemos limitarnos a dar la consulta que leerá líneas de detalles, y dejar que sea Visual Studio quien configure el resto de las instrucciones:

```
select *
from Lineas
where IDFactura = @IDFactura
```

Una vez configurados los adaptadores, cree un conjunto de datos con tipos basado en los adaptadores de facturas, líneas y países, y llámelo *DsFacturas*. Tendremos que retocar algunas columnas en la tabla de facturas:

- El valor por omisión de *Numero* debe ser *-1*. El número de factura será asignado en el servidor, con la ayuda de un *trigger*.
- El valor por omisión de *BaseImponible*, *Impuestos* e *Importe* debe ser *0*.
- Asigne *False* en la propiedad *AllowDBNull* de *Creacion*.

Hemos asignado valores por omisión apropiados para algunas columnas que deberán ser modificadas en el servidor. Y como siempre, permitimos que la columna *Creacion* pueda abandonar la capa de presentación con un valor nulo en su interior.

## El diálogo de edición de facturas

Compile el proyecto, para actualizar los orígenes de datos, y configure los controles antes mostrados para el diálogo de facturas arrastrando y soltando el origen de datos maestro. No olvide que la columna del país debe editarse mediante un combo, y que puede ahorrarse mucho trabajo arrastrando el origen de datos de *Paises*, que está dentro de *DsFacturas*, sobre el combo.

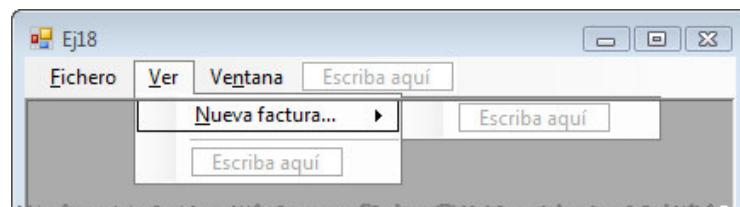
Tenemos que redefinir la implementación de *InitData*, para leer la tabla de países:

```
public override void InitData(IDataWindow parent, bool newRow)
{
    dsFacturas.Merge(Data.Instance.Read("DsPaises"), false);
    base.InitData(parent, newRow);
}
```

Interceptaremos también el evento *CurrentItemChanged* del origen de enlace principal, asociado a la tabla de cabeceras de facturas. Cada vez que modifiquemos alguno de los campos de esta cabecera, se ejecutará el siguiente método:

```
private void facturasBindingSource_CurrentItemChanged(
    object sender, EventArgs e)
{
    DataRowView drv = (DataRowView)facturasBindingSource.Current;
    if (drv["IDCliente"] == System.DBNull.Value)
        txNombre.Text = "";
    else
        txNombre.Text = String.Format("{0} {1}/{2}",
            drv["Nombre"], drv["Apellidos"], drv["NIF"]);
}
```

Como ve, el propósito de *CurrentItemChanged* es mostrar información sobre el cliente seleccionado en la cabecera en el cuadro de texto correspondiente. Por último, hay que agregar un comando de menú en la ventana principal, para mostrar directamente este cuadro de diálogo:



La respuesta al comando de menú debe ser la siguiente:

```
private void miNuevaFactura_Click(object sender, System.EventArgs e)
{
    LayoutManager.Execute(typeof(DlgFactura), this, true);
}
```

Estamos lanzando un cuadro de edición modal desde una ventana que no implementa *IDataWindow*, pero no se producirá ninguna catástrofe, entre otros motivos, porque se trata de una inserción y no hay necesidad de copiar registros desde una ventana nodriza.

## El cliente quiere sus galletas

Antes de ocuparnos del diálogo de selección de clientes, vamos a encapsular algo de código. Ahora mismo, para realizar una búsqueda de clientes necesitamos el mecanismo de *cookies*, para todo lo relacionado con la paginación, más algo de código *ad hoc* para la condición de búsqueda, hablando con propiedad. Para el diálogo de selección necesitaremos nuevamente gran parte de esta funcionalidad. ¿Qué tal si encapsulamos ambos sistemas en uno?

Vamos a crear una nueva clase, *CustomerCookie*, que derivaremos por herencia de *PageCookie*, y que situaré en el fichero *Traductor.cs*. Como la funcionalidad que necesitamos añadir está relacionada con la condición de búsqueda, la mayor parte de la acción de la nueva clase tendrá lugar en los constructores:

```
public CustomerCookie(string searchString, string sort, int count):
    base("Clientes", "Clientes", Filtro(searchString), sort, count)
{ }

public CustomerCookie(string searchString, string sort, int count,
    string details, string masterfield, string detailsfield):
    base("Clientes", "Clientes", Filtro(searchString), sort, count,
        details, masterfield, detailsfield) { }
```

Observe que el truco está en generar el filtro por medio de un método llamado... vale, el método se llama *Filtro*, que al usarse en la forma indicada está gritando a los cuatro vientos que debe ser un método estático. Aquí tiene la implementación del método:

```
private static void ConsultaAtributos(
    StringBuilder sb, string param, string value)
{
    // ATTRNAMES también debe declararse static
    for (int i = 0; i < ATTRNAMES.GetLength(0); i++)
        if (String.Compare(param, ATTRNAMES[i,0], true) == 0)
        {
            sb.AppendFormat(TPL_COLUMNAS, ATTRNAMES[i,1], value);
            return;
        }
    sb.AppendFormat(TPL_ATRIBUTOS, param.Replace("''", "'''"), value);
}

private static string Filtro(string searchString)
{
    StringBuilder sb = new StringBuilder();
    foreach (Traductor.Token token in new Traductor(searchString)) {
        string s = token.Value.Replace("''", "'''");
        if (sb.Length > 0)
            sb.Append(" and ");
        if (token.Param == Traductor.KEYWORD)
            sb.AppendFormat(TPL_NOMBRES, s);
        else
            ConsultaAtributos(sb, token.Param, s);
    }
    return sb.ToString();
}
```

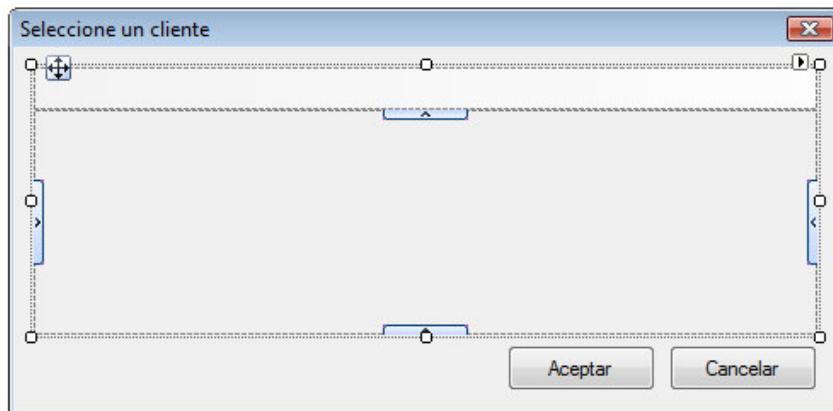
En efecto, este código estaba originalmente escrito en la clase *Clientes*, y ahora lo hemos movido, con algunos pequeños retoques, a *CustomerCookie*. En la ventana de clientes, además de borrar el código ahora innecesario, podemos también simplificar la respuesta al evento *Click* del botón de búsquedas por palabras claves:

```
private void bnBuscar_Click(object sender, EventArgs e)
{
    cookie = new CustomerCookie(txBuscar.Text,
        "Nombre,Apellidos,ICliente", 20,
        "Atributos", "ICliente", "ICliente");
    dsClientes.Atributos.Clear();
    dsClientes.Clientes.Clear();
    dsClientes.Merge(Data.Instance.Read(ref cookie), false);
    dsClientes.AcceptChanges();
    bnMasRegistros.Enabled = !cookie.Disabled;
    txBuscar.SelectAll();
    txBuscar.Focus();
}
```

Este es casi un ejemplo de manual de uso del polimorfismo: aunque no hay nuevos métodos virtuales implicados, teníamos un algoritmo que funcionaba con una clase base. Le pasamos ahora una clase derivada por herencia y todo sigue funcionando como debe ser.

## Un diálogo para la selección

Nos ocuparemos ahora del diálogo de selección en sí. A diferencia de lo que hicimos con los diálogos de edición, no involucraremos al gestor de ventanas en el funcionamiento de estos diálogos especiales. El motivo principal es no complicar demasiado las cosas a estas alturas del desarrollo del curso; aprenderíamos muy pocas novedades a cambio, créame. Ni siquiera crearemos una clase base: es muy poco el código común a encapsular, y no tendremos ocasiones suficiente de reutilizarlo como para que merezca la pena el esfuerzo. De manera que añadiremos directamente un nuevo formulario al proyecto y lo llamaremos *SelClientes*. Primero arrastraremos un control *ToolStripContainer* sobre la superficie del nuevo diálogo:



Lo habitual es que acoplemos el contenedor para que ocupe toda el área interna del formulario, pero esta vez tiene que compartir espacio con los botones de finalización del diálogo. A continuación, active la ventana Orígenes de datos, y arrastre la tabla de clientes para soltarla sobre el área central del *ToolStripContainer*.



Esta operación crea también una barra de navegación, como ya sabe, que hay que reubicar sobre el panel superior del *ToolStripContainer*. Mi recomendación: active la ventana Esquema del Documento, localice los nodos correspondientes a la barra y el contenedor, y arrastre estos nodos a discreción dentro de dicha ventana. Luego, elimine las columnas superfluas de la rejilla, y añada a la barra un cuadro de texto y un botón para la búsqueda.

Veamos ahora cómo quiero ejecutar el diálogo de selección de clientes. Vaya al diálogo de facturas e intercepte el evento *Click* del botón de selección de clientes de esta manera:

```
private const string COPY_COLUMNS =
    "IDCliente,Direccion1,Direccion2," +
    "CP,Ciudad,>IDPais,Nombre,Apellidos,NIF";

private void bnCliente_Click(object sender, EventArgs e)
{
    using (SelClientes dlg = new SelClientes(
        (DataRowView)facturasBindingSource.Current, COPY_COLUMNS))
    if (dlg.ShowDialog() == DialogResult.OK)
        facturasBindingSource.ResetCurrentItem();
}
```

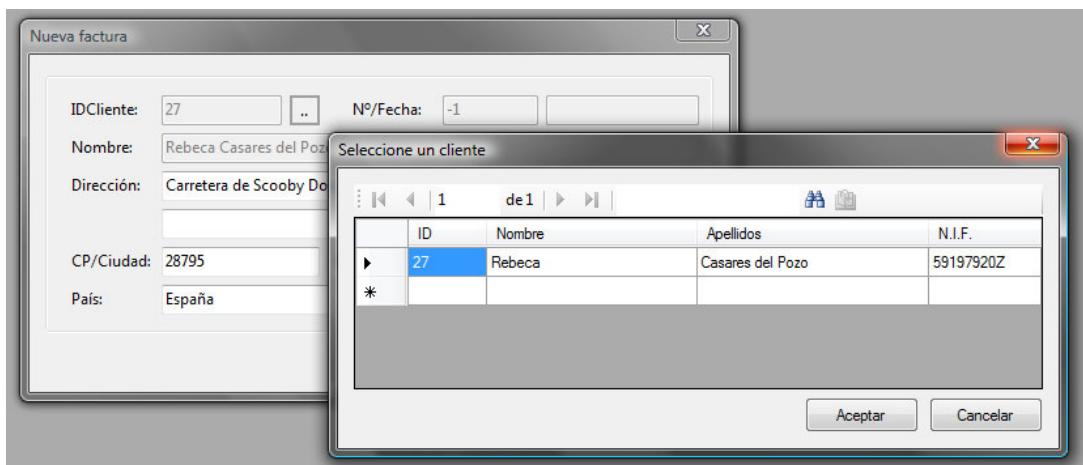
Al diálogo de selección le pasaremos dos parámetros. Uno de ellos será la fila de facturas que estamos editando. Para más exactitud, se trata de la *vista* de fila de factura: recuerde que ésta puede almacenar cambios que aún no hayan pasado a la fila de factura subyacente. El segundo campo es una cadena con la lista de columnas que queremos inicializar en la factura a partir del cliente seleccionado. Como hemos sido cuidadosos en el diseño, estas columnas se llaman igual en la tabla de facturas y en la tabla de clientes. De hecho, el nombre del cliente, sus apellidos y su número de identificación fiscal no existen realmente en la tabla de facturas, sino que se han sintetizado a partir de un encuentro natural, al definir el conjunto de datos. Pero nos da igual: aunque estas tres columnas no se graben finalmente en la base de datos, nos interesa tenerlas en memoria, para mostrarlas siempre al usuario.

Si el diálogo de selección se cierra aceptando, asumiremos que nos adelantará también el trabajo de copiar esos valores desde la fila del cliente seleccionado hacia la fila en edición. La llamada al método *ResetCurrentItem* del origen de enlace es necesaria para que los controles visuales del diálogo de

facturas se den por enterados del cambio en la fila que dicen representar. Si quiere hacer el experimento, cuando esté terminado el ejercicio, quite esta llamada para que vea lo que ocurriría de no existir.

## Tráfico de datos

El diálogo de selección de clientes debe devolver, sobre la fila de factura que recibe, la información sobre el cliente seleccionado. Esta fila de factura, no obstante, tiene otro importante papel: indica al diálogo de selección cuál es el cliente ya seleccionado, si es que existe alguno, para la factura:



El diálogo de edición de facturas, en la imagen anterior, contenía un registro que se refería al cliente #27. Al mostrar el diálogo de selección, la rejilla de éste debe mostrar, como mínimo, el registro ya seleccionado. En este caso, el único registro mostrado inicialmente es ése, precisamente, pero no tiene que ser siempre así: podríamos tener registros adicionales.

Este será el constructor del diálogo de selección:

```
private DataRowView filaFactura;
private string copyColumns;
private PageCookie cookie;

public SelClientes(DataRowView filaFactura, string copyColumns)
{
    InitializeComponent();
    this.filaFactura = filaFactura;
    this.copyColumns = copyColumns;
    if (filaFactura["IDCliente"] == System.DBNull.Value)
        cookie = new CustomerCookie("", "Apellidos,Nombre,IDCliente", 20);
    else
        cookie = new PageCookie(
            "Clientes", "Clientes",
            "IDCliente=" + filaFactura["IDCliente"],
            "Apellidos,Nombre,IDCliente", 20);
    dsClientes.Merge(Data.Instance.Read("DsPaises"));
    dsClientes.Merge(Data.Instance.Read(ref cookie), false);
    dsClientes.AcceptChanges();
    bnMasRegistros.Enabled = !cookie.Disabled;
}
```

Además de almacenar en campos privados los parámetros recibidos, el constructor lee la tabla de países y realiza una primera consulta sobre la tabla de clientes. Si hay un cliente asignado en la fila de la factura, la consulta se realiza mediante *PageCookie*; en caso contrario, se genera la condición con la nueva clase *CustomerCookie*, que también usaremos para la respuesta del botón de búsqueda:

```
private void bnBuscar_Click(object sender, EventArgs e)
{
    cookie = new CustomerCookie(txBuscar.Text,
        "Apellidos,Nombre,IDCliente", 20);
```

```
        dsClientes.Clientes.Clear();
        dsClientes.Merge(Data.Instance.Read(ref cookie), false);
        dsClientes.AcceptChanges();
        bnMasRegistros.Enabled = !cookie.Disabled;
        txBuscar.SelectAll();
        txBuscar.Focus();
    }
```

Finalmente, debemos interceptar el evento *Closing* y comprobar cuál botón ha sido pulsado. Si hemos aceptado, copiaremos información de vuelta en la fila de la factura:

```
private void SelectionBase_Closing(
    object sender, System.ComponentModel.CancelEventArgs e)
{
    if (this.DialogResult == DialogResult.OK)
    {
        DataRowView currentRowView =
            (DataRowView)clientesBindingSource.Current;
        if (!filaFactura.IsEdit)
            filaFactura.BeginEdit();
        foreach (string cName in copyColumns.Split(','))
            if (currentRowView != null)
                filaFactura[cName] = currentRowView[cName];
            else
                filaFactura[cName] = System.DBNull.Value;
    }
}
```

Este evento sólo hace algo cuando el usuario acepta la ejecución modal. Ponemos el registro maestro en modo de edición y copiamos campo a campo, desde la fila seleccionada a la fila maestra que nos han pasado en el constructor del formulario. Debemos tener cuidado, y asumir que, cuando no hay ninguna fila seleccionada en el diálogo, lo que el usuario desea es eliminar la selección de la fila maestra que estaba editando.

## Cuando se trata de un nuevo cliente...

Un último detalle: cuando estamos seleccionando un registro de una tabla grande, es muy probable que necesitemos crear un registro nuevo. Por este motivo, es conveniente tener a mano algún método para crear un nuevo cliente y seleccionarlo de forma automática en la factura que estamos rellenando. Puede hacer la prueba, adaptando el actual diálogo de edición de clientes a este propósito. Este es otro motivo por el que los diálogos modales tienen tan mala fama: si no hemos previsto esta necesidad, el usuario tendría que cancelar la creación de la factura, crear el registro de cliente, y sólo entonces regresar a la factura, comenzando desde el inicio otra vez.

# EJERCICIO 19

## Objetivos

Grabación de facturas

## Técnicas introducidas

Grabaciones y triggers

MÁS QUE INTRODUCIR TÉCNICAS NUEVAS, ESTE ejercicio nos servirá para completar la más importante ventana de la aplicación: el diálogo de creación de facturas. Debido a la pobreza de los controles visuales de la actual versión de Windows Forms, tendremos que ingeniarlas para editar las líneas de detalles sin tener que crear tres o cuatro componentes para la ocasión. Y veremos un curioso problema que se presenta al grabar facturas; un problema relacionado con la relectura de registros modificados.

## El algoritmo de grabación

En el ejercicio anterior dejamos pendiente retocar las instrucciones para la actualización de las líneas de facturas. Antes de hacerlo, deberíamos modificar también la consulta de selección, para incluir en la misma el nombre del producto, como columna redundante:

```
select Lineas.*, Productos.Producto
from Lineas inner join Productos
on Lineas.IDProducto = Productos.IDProducto
where Lineas.IDFactura = @IDFactura
```

Estas son las instrucciones necesarias para las actualizaciones:

```
/* Inserciones */
insert into Lineas (
    IDFactura, IDProducto, Cantidad, Precio, Descuento, Impuesto)
values (
    @IDFactura, @IDProducto, @Cantidad, @Precio, @Descuento, 0);

select IDLinea, Impuesto, Creacion, TS
from Lineas
where IDLinea = scope_identity()

/* Modificaciones */
update Lineas
set IDFactura = @IDFactura, IDProducto = @IDProducto,
    Cantidad = @Cantidad, Precio = @Precio, Descuento = @Descuento
where IDLinea = @Original_IDLinea and TS = @Original_TS;

select Lineas.Impuesto, Lineas.TS, Productos.Producto
from Lineas inner join Productos
on Lineas.IDProducto = Productos.IDProducto
where IDLinea = @Original_IDLinea

/* Borrados */
delete from Lineas
where IDLinea = @Original_IDLinea and TS = @Original_TS
```

Nada del otro mundo, excepto que después de una modificación releemos el nombre del producto, para el caso en que cambiamos el producto al que hace referencia la línea. En cuanto al código necesario para sincronizar la acción de los dos adaptadores durante las actualizaciones, es casi idéntico al código utilizado para grabar clientes y atributos:

```
protected DataSet GuardarFacturas(DataSet delta)
{
    using (Transaccion trans =
        new Transaccion(sqlConn, daFacturas, daLineas))
    {
        DataTable facturas = delta.Tables["Facturas"];
        DataTable lineas = delta.Tables["Lineas"];
        daLineas.Update(lineas.Select("", "", DataViewRowState.Deleted));
```

```
        daFacturas.Update(facturas);
        daLineas.Update(lineas.Select("", "", 
            DataViewRowState.Added | DataViewRowState.ModifiedCurrent));
        trans.Commit();
        return delta;
    }
}
```

No obstante, el peligro acecha tras esta apariencia de cotidiana normalidad...

## Una enmarañada cortina de disparos

Si usted sospechaba de tan poco código para guardar una factura, tengo que reconocer que no iba desencaminado. Las tres cuartas partes del algoritmo no estaban a la vista, sino escondida dentro de *triggers* de la base de datos. Vamos a seguir las huellas que deja tras de sí una factura que se añade a la base de datos.

El primer registro que se inserta es la cabecera de la factura, en la tabla *Facturas*. Este es el *trigger* que se dispara una vez que ya se ha insertado el registro:

```
create trigger ccsFacturas_i on dbo.Facturas
    with encryption for insert as
begin
    set nocount on
    declare @id T_IDENT, @num T_IDENT
    declare _fact cursor local forward_only static read_only for
        select IDFactura from inserted
    open _fact
    fetch _fact into @id
    while (@@fetch_status = 0)
        begin
            update dbo.Contadores
            set @num = IDUltimo,
                IDUltimo = IDUltimo + 1
            update dbo.Facturas
            set Numero = @num
            where IDFactura = @id
            fetch _fact into @id
        end
    close _fact
    deallocate _fact
end
go
```

Como los *triggers* de SQL Server se disparan después de una instrucción, es posible que hayamos insertado más de un registro en esa inserción; he dicho “posible”, no “probable”. Pero de todos modos es buena idea comportarse como si la tabla temporal *inserted* pudiese contener más de una fila. Por este motivo, abrimos un cursor dentro del *trigger* para que recorra cada fila insertada y realice la acción que determinemos sobre ella.

**NOTA**

Si Microsoft me preguntase mi opinión (que no lo va a hacer) pondría en la lista de tareas pendientes para el equipo de desarrollo de SQL Server la adición de una operación **for**, o **foreach**, o como demonios la quieran llamar, que simplifique la escritura de bucles cerrados sobre las filas de un cursor de Transact SQL. Oracle, por ejemplo, ofrece cursos y ofrece esa instrucción simplificadora. InterBase no ofrece cursos, pero soporta una instrucción de iteración equivalente. ¡Por favor, Bill, apiádate de este pecador...!

En particular, para cada cabecera insertada modificamos el valor de su columna *Numero*. Los números de factura se toman de una tabla llamada *Contadores*, que tiene una sola fila. Consultamos el valor de la columna *IDUltimo* en esa única fila, a la vez que incrementamos su valor. Luego copiamos el valor obtenido en la columna *Numero* de la factura examinada. Fuera de la estrambótica sintaxis asociada al recorrido del cursor, no hay nada de qué asombrarse.

Después de grabada la cabecera, se supone que la aplicación seguirá grabando las líneas de la factura. Y nuevamente hay un *trigger* que se dispara después de grabar cada línea:

```

create trigger ccsLineas_i on dbo.Lineas
    with encryption for insert as
begin
    set nocount on
    update Lineas
    set     Impuesto = round(imp.Tasa *
                           (ins.Precio - ins.Descuento) * ins.Cantidad / 100.0, 2)
    from   dbo.Productos pro, dbo.Impuestos imp, inserted ins
    where  ins.IDLinea = Lineas.IDLinea and
           ins.IDProducto = pro.IDProducto and
           pro.IDImpuesto = imp.IDImpuesto
    update Facturas
    set     BaseImponible =
           (select sum((Precio - Descuento) * Cantidad)
            from Lineas lin
            where lin.IDFactura = Facturas.IDFactura),
           Impuestos =
           (select sum(Impuesto)
            from Lineas lin
            where lin.IDFactura = Facturas.IDFactura),
           Importe =
           (select sum((Precio - Descuento) * Cantidad + Impuesto)
            from Lineas lin
            where lin.IDFactura = Facturas.IDFactura)
    from   inserted ins
    where  ins.IDFactura = Facturas.IDFactura
    insert into Movimientos(IDProducto, IDFactura, Cantidad, Precio)
    select IDProducto, IDFactura, -Cantidad, Precio - Impuesto
    from   inserted
end
go

```

- 96 Esta vez no necesitamos un cursor sobre *inserted* porque existen operaciones de conjuntos que pueden realizar las acciones que necesitamos. En concreto, actualizamos la columna *Impuesto* de cada fila insertada, luego vamos a las cabeceras de facturas asociadas, que potencialmente pueden ser varias, pero que en la práctica es una solamente. Y para cada cabecera, actualizamos las tres columnas de importes totales almacenadas en ella: el total antes de impuestos, el total de impuestos, y la suma de estos dos importes parciales. Los tres totales se vuelven a calcular, en vez de intentar una corrección incremental. Aunque es posible que la corrección incremental fuese un poco más eficiente, la diferencia en coste no es tan grande como para arriesgarnos con la mayor complejidad de esa alternativa. Finalmente, para cada línea insertada, se crea un registro dentro de la tabla *Movimientos*, para reflejar la salida del inventario de las unidades vendidas del producto.

Lo importante es que este *trigger* se ejecuta después que hemos actualizado explícitamente el registro de factura, e incluso después de que hayamos recuperado su “última” versión. Al actualizar las tres columnas con totales, se modifica el valor de la columna *TS*, y cualquier operación sobre la copia de la fila que tenemos en memoria fallará, al no coincidir el valor real de *TS* con el valor que conocemos. Las consecuencias son éstas:

- Tendremos que releer las facturas en una segunda operación, al terminar todo el proceso de grabación. Tenemos que contar con la posibilidad de que hayamos actualizado más de una factura en una misma operación.
- Aunque puede parecer lo contrario, no podemos quitar la instrucción **select** que va después de la instrucción **insert** sobre la tabla de *Facturas*. A pesar de que no nos devolverá la fila final, es sumamente necesaria para conocer la clave primaria de las facturas insertadas.

Para terminar con esta historia, la grabación de un registro en *Movimientos* provoca cambios en la tabla de productos, específicamente en la columna *Existencias*:

```

create trigger ccsMovimientos_i on dbo.Movimientos
    with encryption for insert as
begin
    set nocount on
    update Productos
    set Existencias = Existencias + Cantidad
    from inserted
    where Productos.IDProducto = inserted.IDProducto
end
go

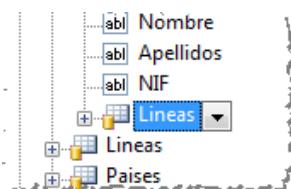
```

Si está activada la restricción que impide que las existencias sean menores que un valor mínimo admisible, este *trigger* puede provocar un error que detendría el proceso de grabación. En ese caso, la transacción que sirve de paraguas a la grabación se anularía, deshaciendo todos los cambios parciales ejecutados hasta ese momento.

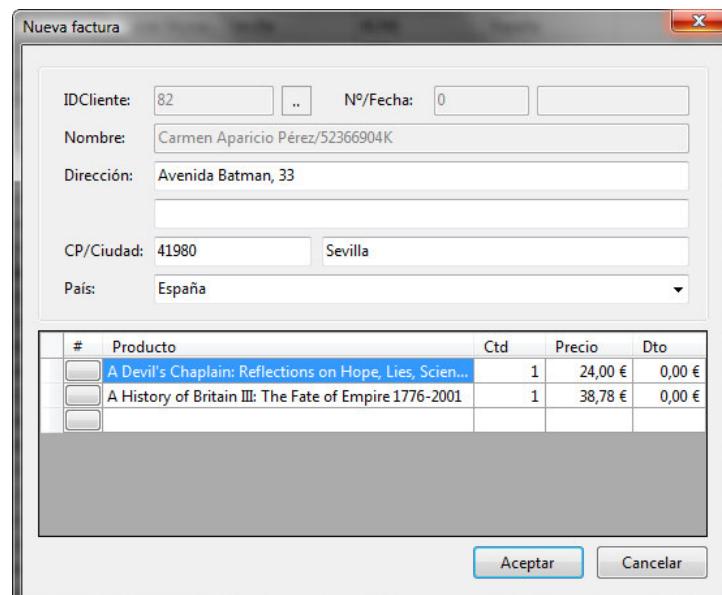
## Adaptaciones a la edición maestro/detalles

Resumiendo: después de terminar una actualización de facturas, tendremos que releer las facturas presentes en el conjunto de datos *delta*, porque sus versiones de filas estarán fuera de sincronía. No obstante, pospondremos esta operación por el momento. En definitiva, después de insertar en este ejercicio, no volvemos a una ventana de navegación, por lo que no es urgente la relectura hasta el ejercicio en que introduzcamos la ventana de búsqueda de facturas.

El primer paso es crear una rejilla para editar las líneas de detalle. Para ello, arrastre el nodo *Líneas* anidado dentro del origen de datos correspondiente a la tabla de facturas:



Nuestro objetivo es configurar la rejilla resultante para que tenga un aspecto similar al siguiente:



Note que hemos escondido la mayoría de las columnas: todas las columnas escondidas recibirán sus valores de manera automática como parte del proceso de edición. Tan sólo han sobrevivido cinco columnas. La primera de ellas es una columna de tipo *DataGridViewButtonColumn*, y no está enlazada a la tabla de rejillas, pues su única función es proveer un botón para editar cada fila de detalles. La siguiente columna debe declararse de sólo lectura, y corresponde a la columna con el nombre del

producto asignado a la línea. Recuerde que esta columna proviene de un encuentro natural en la consulta utilizada para las líneas:

```
select Lineas.* , Productos.Producto
from Lineas inner join Productos
    on Lineas.IDProducto = Productos.IDProducto
where Lineas.IDFactura = @IDFactura
```

Las tres últimas columnas son más normales. El único detalle digno de mención es que debemos tener cuidado con los formatos de visualización de cada una.

Cuando el usuario pulse el botón para editar o añadir una nueva fila de detalle, se disparará el evento *CellContentClick* de la rejilla. Debemos averiguar primero cuál columna es la que ha provocado el evento:

```
private void lineasDataGridView_CellContentClick(
    object sender, DataGridViewCellEventArgs e)
{
    if (e.ColumnIndex == buttonCol.Index)
    {
        facturasBindingSource.EndEdit();
        if (lineasBindingSource.Current == null)
            lineasBindingSource.AddNew();
        using (SelProductos f = new SelProductos(
            (DataRowView)lineasBindingSource.Current,
            "IDProducto, Producto, PVP",
            "IDProducto, Producto, Precio"))
        if (f.ShowDialog() == DialogResult.OK)
        {
            lineasBindingSource.EndEdit();
            lineasBindingSource.ResetCurrentItem();
        }
    }
}
```

Hay bastante código defensivo en el listado anterior. Para empezar, intentamos confirmar las posibles ediciones en la cabecera sobre el conjunto de datos local al diálogo. De esta manera evitamos problemas con las restricciones de integridad referencial que hemos propagado desde la base de datos hacia el conjunto de datos. Esto tiene una consecuencia desagradable: no podremos grabar líneas hasta que no hayamos elegido el cliente. Si esta restricción fuese inaceptable, tendríamos que relajar las verificaciones y restricciones del conjunto de datos de facturas... o inicializar la cabecera con valores “razonables”. Como no creo que sea un asunto demasiado grave, he decidido pasarlo por alto: cualquier solución sería sólo un apañío que complicaría el ya complejo código fuente del ejercicio.

Observe que hay un pequeño cambio en la forma en que se llama al diálogo de selección de productos, si lo comparamos con el diálogo de selección de clientes:

```
using (SelProductos f = new SelProductos(
    (DataRowView)lineasBindingSource.Current,
    "IDProducto, Producto, PVP",
    "IDProducto, Producto, Precio")) { ... }
```

Esta vez, pasamos dos listas de columnas al constructor del diálogo. El motivo es que la columna con el precio unitario se llama *PVP* en la tabla de productos, y *Precio* en la de facturas. Tome nota también de las dos llamadas al aceptarse el cuadro de diálogo. La primera da por concluida la edición de la línea activa, y la segunda provoca que la rejilla refleje los cambios realizados. Puede comprobar, si lo desea, que no podemos prescindir de ninguna de las dos.

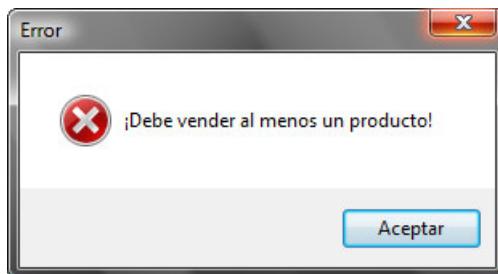
## Comprobaciones sobre líneas de facturas

Veamos ahora cómo realizar algunas validaciones sencillas dentro del diálogo de edición de facturas. Existe una comprobación, en particular, que muchas veces olvidamos: ¿tiene líneas la factura? Lo interesante de esta validación es que no se ajusta a los patrones habituales: no es una validación a

nivel de columna ni a nivel de fila, para ser exactos. Si usamos un *ErrorProvider* para mostrar los errores, ¿a qué control o columna habría que asociar el correspondiente mensaje de error? En mi opinión, no habría nada deshonroso en tratar este fallo lanzando una excepción. Eso sí, para evitar ese feo mensaje que deja al desnudo nada menos que la traza de la pila de la aplicación, necesitaremos algunos pequeños cambios en el código de *Program.cs*:

```
static void Main()
{
    Application.EnableVisualStyles();
    Application.SetCompatibleTextRenderingDefault(false);
    Application.ThreadException += Application_ThreadException;
    Application.Run(new MainForm());
}

static void Application_ThreadException(
    object sender, ThreadExceptionEventArgs e)
{
    MessageBox.Show(e.Exception.Message, "Error",
        MessageBoxButtons.OK, MessageBoxIcon.Error);
}
```



El siguiente método muestra como detectar este error y otros tipos de error adicionales:

```
public override bool ValidateData()
{
    /* Comprobar si hay líneas en la factura */
    if (dsFacturas.Lineas.Count == 0)
        throw new ApplicationException(
            "¡Debe vender al menos un producto!");
    /* Eliminar posibles errores anteriores */
    if (dsFacturas.Lineas.HasErrors)
        foreach (DataRow row in dsFacturas.Lineas.GetErrors())
            row.ClearErrors();
    /* Comprobar si quedan errores */
    foreach (DsFacturas.LineasRow row in dsFacturas.Lineas)
        if (row.Cantidad == 0)
            row.SetColumnError("Cantidad", "La cantidad no puede ser 0");
        else if (row.Desuento > row.Precio)
            row.RowError = "El descuento es superior al precio!";
    return !dsFacturas.Lineas.HasErrors;
}
```

Aunque el componente *ErrorProvider* esté asociado a la vista de la cabecera, la rejilla de datos permite mostrar errores sin necesidad de configurarla especialmente:

## Notificaciones entre ventanas

Hace unos cuantos ejercicios que no tocamos el gestor de ventanas, y ya estoy sintiendo el síndrome de abstinencia (... no, me acaban de sonar las tripas; debe ser sólo hambre). Tenemos un pretexto para ampliar ahora su funcionalidad. Se lo explicaré con una simulación:

- El usuario abre la ventana de productos, y busca los libros relacionados con ASP.NET. En la rejilla se muestra un pequeño grupo de ellos. Inmediatamente, activa la ventana de creación de facturas, y crea una en la que incluye uno de los libros que ha encontrado, y un segundo libro sobre un tema diferente. Cuando guarda la factura, la base de datos disminuye las existencias

asociadas a los dos productos facturados. Pero en la pantalla queda un registro correspondiente a uno de los productos, con el número de unidades en inventario equivocado. Queremos que ese registro se actualiza con los valores más recientes, sin intervención del usuario, y con el mínimo coste para el sistema.

Es muy importante que no confunda este requisito con una exigencia muy diferente, que para muchos usuarios e incluso programadores se llega a convertir en una obsesión:

- El usuario deja abierta una ventana de búsquedas con unos cuantos registros en su interior. Alguien modifica uno de esos registros desde otro puesto, al otro lado del mundo. Los programadores con tendencias místicas pretenden que la copia abandonada a su suerte por el usuario del primer sistema se actualice automágicamente.

Esto último es un planteamiento muy peligroso: cualquier intento de implementación añade una carga excesiva al servidor y a la red. En estos casos, es preferible correr el riesgo de encontrar una copia fuera de sincronía al intentar una modificación y corregir el problema en ese momento. Como dice el refrán: es preferible equivocarse y pedir perdón antes que pasarse la vida pidiendo permiso.

Volvamos a lo nuestro. ¿No es también un riesgo aceptable dejar un registro de producto obsoleto en pantalla? Claro que es aceptable. La diferencia está en que existe una técnica muy sencilla y poco costosa de corregirlo. En nuestro caso, las actualizaciones realizadas desde el propio ordenador generarán notificaciones a las ventanas de navegación que puedan estar abiertas en ese momento. Es cierto que la solicitud de copias actualizadas de los registros afectados aumentará algo el tráfico en red, pero al menos habremos evitado el tráfico de las propias notificaciones. Y sobre todo, el sistema podrá reaccionar de forma más inteligente a las mismas, como veremos enseguida.

Cada vez que se produzcan determinadas acciones dentro de la aplicación, queremos que se envíen notificaciones locales a todas las ventanas no modales que estén abiertas en ese momento. ¿Quién sabe cuáles ventanas están abiertas? Está claro que el gestor de ventanas. Por lo tanto, los agentes que lleven a cabo las operaciones interesantes, deberán informar sobre ello al gestor de ventanas, para que este se encargue entonces de anunciarlo a todas las partes interesadas.

Necesitamos añadir un par de métodos a la interfaz *ILayoutManager*, y definir un nuevo tipo de interfaz, al que llamaremos *IDataNotifySink* (más o menos, *receptor de notificaciones de datos*):

```
public interface ILayoutManager
{
    IWindow Create(System.Type type);
    IWindow Activate(System.Type type);
    DialogResult Execute(System.Type type, Form owner, bool newRow);
    void DataModified(string eventname, object data);
    void ApplyNotifications();

    event System.EventHandler CreateWindow;
}

public interface IDataNotifySink
{
    void DataModified(string eventname, object data);
    void ApplyNotifications();
}
```

Es fácil detectar que los dos métodos añadidos a *ILayoutManager* son los mismos de *IDataNotifySink*. No se trata de una coincidencia: una forma alternativa de entender el sistema que implementaremos es considerar que el gestor de ventanas contiene una implementación muy particular del nuevo tipo de interfaz. Pero ésta es una distinción formal que aporta muy poco al funcionamiento de la aplicación, por lo que he preferido evitar complicaciones formales.

¿Qué significado, o qué uso se les dará a los dos métodos mencionados? ¿Por qué dos métodos en vez de uno solamente? El propio proceso de grabación de facturas nos lo aclarará:

- El usuario graba una factura en la que incluye dos libros. Ambos libros están presentes en la última búsqueda de la ventana de navegación de productos.

- La detección de la operación sobre los libros se realiza en serie. El sistema detecta primero el movimiento de uno de los libros, y después, la operación sobre el otro. Por lo tanto, envía dos notificaciones independientes.
- Si la ventana de productos no toma medidas, reaccionará por separado a las dos notificaciones. Como consecuencia, enviaría dos peticiones separadas al servidor de capa intermedia para recuperar información puesta al día. Sería mejor, sin embargo, si pudiésemos agrupar ambas peticiones en una sola.

El propósito de *ApplyNotifications* será ese: permitir la agregación de las acciones desencadenadas por las notificaciones. Los subsistemas que reaccionen a las notificaciones acumularán trabajo cada vez que recibe una notificación mediante una llamada a *DataModified*. Pero sólo ejecutarán lo que tengan que ejecutar cuando reciban el visto bueno mediante una llamada a *ApplyNotifications*. La ejecución de este método se interpretará como “bueno señores, esto ha sido todo, haced ahora lo que creáis correcto”.

Veamos la implementación de los métodos nuevos en el gestor de ventanas, en *MdiLayoutManager*:

```
/* Clase: MdiLayoutManager */
void ILayoutManager.DataModified(string eventname, object data)
{
    foreach (Form f in mdiParent.MdiChildren)
        if (f.Controls.Count > 0) {
            IDataNotifySink sink = f.Controls[0] as IDataNotifySink;
            if (sink != null)
                sink.DataModified(eventname, data);
        }
}

void ILayoutManager.ApplyNotifications()
{
    foreach (Form f in mdiParent.MdiChildren)
        if (f.Controls.Count > 0)
        {
            IDataNotifySink sink = f.Controls[0] as IDataNotifySink;
            if (sink != null)
                sink.ApplyNotifications();
        }
}
```

¿Queda claro por qué hemos involucrado al gestor de ventanas en este sistema? Sólo él sabe cómo recorrer la lista de ventanas no modales, y localizar el elemento de ellas que implementa la nueva interfaz *IDataNotifySink*. Observe que hemos preferido separar *IDataNotifySink* de *IWindow*: de esta manera, podremos tener ventanas que implementen *IWindow* y no *IDataNotifySink*. Sólo las ventanas que realmente tengan que hacer cambios al recibir estas notificaciones deben implementar el nuevo tipo de interfaz.

## Notificaciones durante las actualizaciones

Ya tenemos una implementación de dos métodos relacionados con las notificaciones en el gestor de ventanas. Ninguna clase de ventana implementa todavía *IDataNotifySink*, pero esta es una situación aceptable, como acabamos de explicar. Para seguir avanzando, necesitamos ahora un agente que genere notificaciones. Para ello, añadiremos un grupo de métodos al módulo de datos central, implementado por la clase *Data*, que nos permitirán enviar notificaciones agrupadas. Para facilitar la lectura del código, encerraremos los métodos, junto con una clase auxiliar y un campo relacionado, dentro de una región de código:

```
#region Notificaciones de datos

internal class Notificación {
    public string eventname;
    public object data;
}
```

```

private List<Notificación> notificaciones = new List<Notificación>();

private void IniciarNotificaciones()
{
    notificaciones.Clear();
}

private void AñadirNotificación(string eventname, object data)
{
    Notificación n = new Notificación();
    n.eventname = eventname;
    n.data = data;
    notificaciones.Add(n);
}

private void EnviarNotificaciones()
{
    if (notificaciones.Count > 0) {
        foreach (Notificación n in notificaciones)
            MainForm.LayoutManager.DataModified(n.eventname, n.data);
        MainForm.LayoutManager.ApplyNotifications();
    }
    notificaciones.Clear();
}

#endregion

```

Dos de los métodos anteriores se utilizan dentro de la nueva implementación que le daremos al método *Write*:

```

public System.Data.DataSet Write(string dsname, DataSet delta)
{
    IniciarNotificaciones();
    if (dsname == "DsPaises")
        delta = GuardarPaises(delta);
    else if (dsname == "DsImpuestos")
        daImpuestos.Update(delta);
    else if (dsname == "DsProductos")
        daProductos.Update(delta);
    else if (dsname == "DsClientes")
        delta = GuardarClientes(delta);
    else if (dsname == "DsFacturas")
        delta = GuardarFacturas(delta);
    else
        throw new ApplicationException(
            "La operación no ha sido implementada.");
    EnviarNotificaciones();
    return delta;
}

```

Con estos cambios, si cualquier operación de grabación emite una notificación ejecutando el método *AñadirNotificación*, el código de *EnviarNotificaciones* lo detectará y enviará los avisos al gestor de ventanas, el cuál actuará como repetidor, informando a todas las ventanas no modales interesadas.

- 344 ¿Quiere ver un ejemplo? Interceptor el evento *RowUpdated* del adaptador de datos asociado a la grabación de líneas de facturas:

```

private void daLineas_RowUpdated(
    object sender, System.Data.SqlClient.SqlRowEventArgs e)
{
    if (e.Status == System.Data.UpdateStatus.Continue)
        AñadirNotificación("productos", e.Row["IDProducto"]);
}

```

El contenido de las notificaciones es ya cosa nuestra. En este caso he decidido que el parámetro *eventname* contenga el nombre de la entidad actualizada, en minúsculas y en plural, y que *data* sea la clave primaria del registro afectado. Usted puede establecer el convenio que le venga en gana. Dis-

frute del nuevo mecanismo de notificaciones. Y ya sabe: quien avisa, no es traidor... y el árbol que nace torcido, se lo lleva la corriente.

## Centinela alerta

Antes de dar por cerrado el ejercicio, veamos qué puede hacer una atribulada ventana de búsqueda y navegación sobre productos cuando alguien le viene con chismorros acerca de que los productos que anuncia están caducados:

```
/* Clase: Productos */

#region Miembros de IDataNotifySink

private string keysModified = "";
private int totalKeys = 0;

public void DataModified(string eventname, object data)
{
    if (eventname == "productos" &&
        dsProductos.Productos.FindByIDProducto((int)data) != null)
    {
        if (totalKeys++ > 0)
            keysModified += ",";
        keysModified += data.ToString();
    }
}

public void ApplyNotifications()
{
    if (totalKeys > 0)
    {
        PageCookie cookie = new PageCookie(
            "Productos", "Productos",
            "IDProducto in (" + keysModified + ")",
            "IDProducto", totalKeys);
        dsProductos.Merge(Data.Instance.Read(ref cookie), false);
        keysModified = "";
        totalKeys = 0;
    }
}

#endregion
```

Nuestra clase, ante tamaña insidia, se prepara para pedir una copia más reciente de los registros difamados. Cada vez que le lanzan un *DataModified*, almacena la clave primaria en una variable interna... pero sólo si el chismorreo se refiere realmente a uno de los productos en pantalla. Cuando recibe la llamada a *ApplyNotifications*, compone una *cookie* de paginación basada en la tabla de productos, y un filtro que exige que la clave de los registros a devolver pertenezca a la lista de claves generadas. Esta *cookie* generaría consultas como la siguiente:

```
select top 2 *
from Productos
where IDProducto in (34, 68)
order by IDProducto
```

Sí, es cierto que hay cláusulas que sobran en esta instrucción, pero sería fácil modificar *PageCookie* para optimizar el texto de la consulta generada en casos como éste. En todo caso, la *cookie* se pasa al método *Read* de la interfaz *IDataServer* usada por la ventana para pedir datos, y el conjunto de datos que se obtiene como resultado se mezcla con los registros que está mostrando la ventana.

# EJERCICIO 20

## Objetivos

Extensibilidad mediante *scripts*

## Técnicas introducidas

*Microsoft.Vsa, JScript, IVsaEngine*

**¿**UÁNDO SE PUEDE DAR UNA APLICACIÓN POR terminada? La mayoría de los programadores padecemos un tipo de neurosis que nos obliga a repasar una y otra vez el código de esa aplicación que ya hace todo lo que el contratista había pedido y pagado y que funciona perfectamente. Esa misma neurosis nos puede mantener añadiendo características a la aplicación que nadie nos ha pedido y nadie nos paga. En verdad, nos cuesta dar por liquidado cualquier proyecto de desarrollo... y siempre hay algún listo que intenta sacar provecho de ello.

## En busca del valle de la extensibilidad perdida

La obsesión también se manifiesta cuando estamos diseñando una aplicación. Me pasó cuando decidí que los descuentos en el ejemplo de este curso deberían calcularse automáticamente. El usuario tendría la posibilidad de establecer reglas, del tipo más general posible, y el programa podría aplicarlas en tiempo de ejecución, sin necesidad de retocar la aplicación. Y como vi que era buena idea, me enrollé las mangas de la camiseta (metafóricamente hablando, claro) y puse todo mi empeño en diseñar una tabla para almacenar dichas reglas.

Cualquier ejemplar de *Annuslistus vulgaris*, una especie parásita que medra en oficinas y grandes empresas, diría sonriente que estamos ante un típico caso de *parametrización*... sea eso lo que sea. Si a un *Annuslistus* se le pide que divida un plano en dos mediante una curva, reptará hasta encontrar un programador y le obligará a programar la ecuación de la recta, almacenando los dos *parámetros* necesarios en una tabla. Si se trata de un espécimen engordado en una facultad de Matemática, porque casi siempre se trata de individuos que han fracasado en alguna facultad, incluida la de Informática, puede que se produzca un destello en su invertebrado sistema nervioso, y le pida que utilice un par más de *parámetros*, por si la curva resulta ser cuadrática o cúbica. Luego del atracón con sangre de la presa, se echará a rumiar satisfecho, e hibernará hasta su próximo encargo.

Hasta hace relativamente poco tiempo, la teoría paramétrica del *Annuslistus* era a lo más que podíamos aspirar, lo que explica el éxito reproductivo de la especie. Pero un buen día, la Programación Orientada a Objetos cayó sobre las empresas de desarrollo como el meteorito de Chicxulub sobre los dinosaurios del Cretácico. La P.O.O. predicaba la unión mística entre datos y código, mientras que la religión paramétrica dibujaba un universo oscuro en el que un Código Supremo, sin principio ni fin (ni posibilidad de mejora) interpretaba a su antojo los datos sacrificados ante Su altar. Si el todopoderoso Código sólo sabía tratar con líneas rectas, mala suerte. Para la próxima, piense usted en una categoría más amplia de curvas en el plano, señor programador.

Pero el *Annuslistus* no se extinguío de un día para otro, sino que hubo un largo período de decadencia que alimentó vanas esperanzas en las almas perdidas. Algunas subespecies sufrieron extravagantes mutaciones, como en el caso del *A. Yourdonis* o el *A. Irremediabilis Petruscoadis*, que encontraron su nicho ecológico inundando el mercado editorial con metodologías absurdas que cambiaban cada año. Uno de ellos, no contento con arruinar selvas enteras para imprimir sus indigeribles panfletos, se atrevió a predecir la extinción del “programador americano” en 1992... sólo para retratarse cuatro años más tarde y zurrarnos un mamotretro sobre el “renacimiento del programador americano”. Tal como suena.

### NOTA

Algunos naturalistas han informado recientemente del avistamiento de algunos ejemplares de *A. I. Petruscoadis* en Silicon Valley, en las cercanías de las oficinas de Boreland. Se les ha visto fabricando sus nidos con cajas desechadas de Quilix (la mayoría de los productos con nombres terminados en equis suelen ser tampones), y gorjeando intrascendencias sobre algo que suena aproximadamente como “*Agile What-the-heck*” y que no hay forma de explicar en menos de cien páginas (ilustraciones no incluidas).

## Extensibilidad mediante código dinámico

¿Por qué ha tardado tanto la definitiva extinción de esta especie junto a su teoría de los parámetros? Para comprenderlo, suponga que estamos creando una aplicación financiera, para manejar préstamos e hipotecas. Hay mil variantes de préstamos, y no queremos caer en el error de pensar en un utópico “modelo de préstamo más general” para extraer una lista de parámetros y hacer precisamente lo que acabamos de criticar. Por el contrario, nuestro diseño parte de una clase o tipo de interfaz que define las características *mínimas* de un préstamo, y que deja que gran parte de las variantes se generen por el diferente comportamiento de las operaciones definidas. Después de definir la clase o tipo de interfaz base, definimos un conjunto de clases derivadas para los tipos de préstamos más comunes. Debe quedar claro que no seremos capaces de predecir la existencia de todos los tipos de préstamos posibles, al igual que antes no podíamos crear el modelo paramétrico más general. Pero ahora hay algo que juega a nuestro favor: en principio, lo único que debemos hacer para que el sistema pueda trabajar con un nuevo tipo de préstamos es derivar otra clase, con nuevos algoritmos para las viejas operaciones, y registrarla dentro del sistema. El efecto neto es que, con la nueva clase, aportamos nuevo código al sistema. Ahí es donde está la potencia de la POO: un sistema POO puede aceptar nuevo código no previsto en el momento de su programación.

Pero imagino que se habrá dado cuenta ya de que estoy esbozando una fantasía: ¿de verdad podemos crear una clase de objetos en la base de datos, y extender la aplicación creando clases derivadas? Con los sistemas relacionales comerciales de los que disponemos, ni de broma. No quiero tener que buscar una explicación al hecho innegable de que no existe *ni un solo sistema* de bases de datos orientado a objetos que haya triunfado comercialmente. Pero la respuesta a la pregunta que nos hacíamos es ésta: las bases de datos relacionales actuales nos impiden aprovechar toda la potencia de la POO.

Si queremos saber *toda* la verdad, hay que reconocer que otra parte de la culpa cae sobre la arquitectura de hardware y, por transitividad, sobre los sistemas operativos actuales. Nuestras unidades de procesamiento establecen una distinción muy clara entre código ejecutable y datos... y cuando relajan un poco la distinción, los malditos fabricantes de virus atacan como fieras lo que parece un punto débil. Pero es en esta dirección donde están apareciendo técnicas esperanzadoras, que algún día nos permitirán romper definitivamente la barrera entre datos y código.

¿Cuál es el problema, que no hay verdadera extensibilidad mientras no podamos añadir *nuevo código* a la aplicación? ¿Qué tal entonces si enlazamos dentro de la aplicación un motor de *scripts*, que nos permita definir el código que nos dé la gana en el momento que nos dé la gana? Usted me preguntará entonces qué hay de nuevo, si desde hace mucho tiempo es posible usar este tipo de motores en virtualmente cualquier escenario. Esta es mi respuesta:

- El motor de *scripts* que ofrece .NET permite generar código que conceptualmente es idéntico al código precompilado de la aplicación. Hay total homogeneidad, de modo que el código precompilado puede usar clases definidas en el código *script* y viceversa.

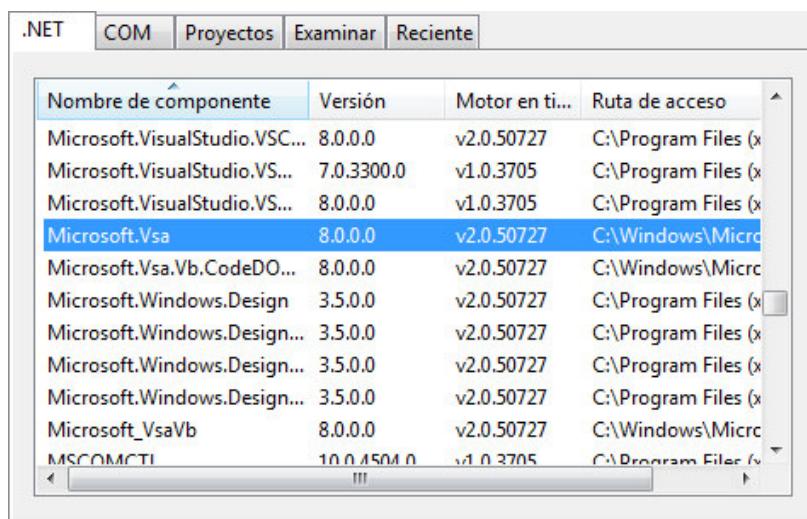
Al principio le costará algo de trabajo comprender por qué es una gran diferencia. Este ejercicio pretende ayudarle a verlo: las reglas que extenderán nuestra aplicación consistirán en definiciones de clases en JScript. Desde nuestro código en C#, crearemos instancias de esas clases en JScript y las utilizaremos como máquinas de calcular descuentos.

## Motores de script

Las aplicaciones que se ejecutan sobre la plataforma .NET pueden incorporar un motor de *script* para su lenguaje preferido aprovechando la funcionalidad del ensamblado *Microsoft.Vsa*, más conocido como *Visual Studio for Applications*. Este ensamblado proporciona una infraestructura común que debe ser complementada mediante motores concretos. En .NET hay dos de tales motores: el de JScript.NET, y el de VBScript. En este curso me voy a referir solamente al motor de JScript. Aunque VSA intenta disimular en lo posible las diferencias entre motores, las posibilidades de JScript y VBScript son muy diferentes. Por mencionar sólo dos:

- VBScript no permite ejecutar “código global”, mientras que JScript sí.
- Los eventos se enlazan en VBScript por su nombre, mientras que en JScript hay que utilizar el mecanismo habitual de .NET, pero con una sintaxis diferente.
- El motor de JScript puede hacer referencia a ensamblados almacenados en ficheros ejecutables y DLLs, mientras que VBScript sólo lo permite con DLLs. Esta capacidad de JScript es importante si se quiere que desde el código *script* se tenga acceso a los tipos de datos del fichero ejecutable que lo alberga.

Para incorporar la infraestructura genérica y el motor de JScript, tendremos que ir al Explorador de Soluciones, seleccionar el nodo de *Referencias* y ejecutar el comando de menú *Agregar referencia*, que mostrará entonces el siguiente cuadro de diálogo:



Tenemos que localizar los siguientes “componentes” y añadirlos al proyecto:

```
Microsoft.Vsa
Microsoft.JScript
```

Vamos a añadir una nueva clase al proyecto, a la que llamaremos *ScriptEngine*. La clase deberá encapsular un puntero a interfaz de tipo *IVsaEngine*. Este es el constructor que debemos declarar para la clase, junto con algunos campos necesarios:

```
private IVsaEngine engine;
private System.Collections.Hashtable globals;

public ScriptEngine(string moniker, string rootNamespace)
{
    engine = new Microsoft.JScript.Vsa.VsaEngine();
    engine.RootMoniker = moniker;
    engine.Site = this;
    engine.InitNew();
    engine.RootNamespace = rootNamespace;

    AddReference("system.dll");
    AddReference("mscorlib.dll");

    globals = new System.Collections.Hashtable();
}
```

*Moniker* es una palabra británica un poco rara que significa simplemente *apodo*. Forma parte del infoslang gracias a un ingeniero inglés que trabajaba en Microsoft, en el equipo de COM. En este contexto, el *moniker* es un nombre único, con el que tenemos que identificar la instancia del motor que vamos a crear. La sugerencia que aparece en la documentación es utilizar el nombre del dominio de la empresa a la inversa, partiendo de la raíz, y completar la cadena con un identificador para la aplicación. Por ejemplo:

```
com.intsight.adonet://ejd20
```

Pero sospecho que con identificadores menos extravagantes tendríamos el mismo éxito, y me temo que tanta insistencia en el “apodo” tenga algo que ver con el famoso humor británico. Lo que sí es muy importante es el valor que pasemos al constructor en el parámetro `rootNamespace`, porque es el nombre que se utilizará para el espacio de nombres por omisión donde se situarán las clases compiladas:

```
ScriptEngine eng = new ScriptEngine(
    "com.intsight.adonet://ejd20", "IntSight.AdoNet01.Ejd20");
```

Al inicializar el motor, hay que proporcionarle un puntero a un objeto que implemente la interfaz `IVsaSite`:

```
engine.Site = this;
```

Es la propia clase `ScriptEngine` quien se encarga de implementar los métodos de `IVsaSite`. Para la mayoría de los métodos, tenemos más que suficiente con una implementación vacía. Sólo he añadido algo de código en la implementación del método `OnCompilerError`, en donde disparamos un evento para informar a quienes estén interesados que se ha producido un error de compilación.

Dentro del constructor de `ScriptEngine` hay también dos llamadas al siguiente método:

```
public void AddReference(string assemblyname)
{
    IVsaItem item = engine.Items.CreateItem(
        assemblyname, VsaItemType.Reference, VsaItemFlag.None);
    ((IVsaReferenceItem) item).AssemblyName = assemblyname;
}
```

Como puede imaginar, este método añade una referencia a un ensamblado ya existente, para que al compilar, podamos acceder a las declaraciones contenidas en éste. En el constructor añadimos, por omisión, referencias a `system.dll` y `mscorlib.dll`, las dos DLLs que implementan el núcleo de la CLR.

¿Y dónde le pasamos el código fuente al motor? Aquí he simplificado un poco el modelo original, que permite añadir varios `code items` al motor, como si se tratase de ficheros distintos compilados en paralelo. He pensado que no necesitamos tantas complicaciones y he implementado una propiedad llamada `Code`, de tipo `string`, que utiliza un `code item` para almacenar y recuperar el código fuente que enviamos al motor para ser compilado y ejecutado.

```
private IVsaCodeItem mainCodeItem = null;

public string Code {
    get {
        if (mainCodeItem == null)
            return "";
        else
            return mainCodeItem.SourceText;
    }
    set {
        if (engine.IsRunning)
            engine.Reset();
        if (mainCodeItem == null)
            mainCodeItem = (IVsaCodeItem) engine.Items.CreateItem(
                "MainCode", VsaItemType.Code, VsaItemFlag.None);
        mainCodeItem.SourceText = value;
    }
}
```

Una vez que el motor ha sido inicializado, tiene las referencias necesarias, y le hemos asignado código fuente, podemos llamar al siguiente método público, para compilar el código y ejecutarlo:

```
public void Run()
{
    if (engine.IsRunning) engine.Reset();
    engine.Compile();
    engine.Run();
}
```

Tenga presente que si el código fuente consiste sólo, como será lo habitual, en una serie de clases, el concepto de “ejecución” significará cargar el código dentro del espacio de procesos de la aplicación que se está ejecutando. Por supuesto, si nos aseguramos de añadir una referencia al ensamblado que contiene Windows Forms, usando una llamada como ésta:

```
eng.AddReference("System.Windows.Forms.dll");
```

podríamos ejecutar código en JScript como el siguiente, que incluye instrucciones que se ejecutarán durante la “carga”, es decir, dentro de la llamada a *Run*:

```
// Atención: ¡esto es JScript.NET, no C#!

import System.Windows.Forms;

MessageBox.Show('¡Hola, Mundo!');
```

Ahora nos ocuparemos de otra categoría de métodos, que serán de utilidad una vez que hayamos compilado *algo* con el motor. Por ejemplo, nos será conveniente disponer de un método como el siguiente, para localizar una clase dentro del ensamblado generado por la compilación:

```
public System.Type GetType(string fullname)
{
    return engine.Assembly.GetType(fullname, true);
}
```

Lo único “especial” que hemos necesitado es que el motor nos ofreciese la referencia al ensamblado generado, mediante la propiedad *IVsaEngine.Assembly*. El método *GetType* que aplicamos sobre esta propiedad es ya un método .NET de uso general, que podemos aplicar a cualquier ensamblado. Si sólo necesitamos el tipo devuelto por *GetType* para crear una instancia con un constructor sin parámetros, podemos tomar un atajo:

```
public object CreateObject(string classname)
{
    return engine.Assembly.GetType(classname, true).InvokeMember(
        null,
        BindingFlags.DeclaredOnly |
        BindingFlags.Public | BindingFlags.NonPublic |
        BindingFlags.Instance | BindingFlags.CreateInstance,
        null, null, null);
}
```

¿Se ha dado cuenta de que es la reflexión lo que hace tan especial los motores de *script* en .NET?

## Un, dos, tres, probando...

Ya podemos poner a prueba el motor, para ver si funciona como esperamos. En la sección anterior mostré un pequeño fragmento de código en JScript.NET, que mostraba un mensaje mediante la archifamosa función estática *MessageBox.Show*, de Windows Forms. Pero más común será tener declaraciones de clases como la siguiente:

```
// Atención: ¡esto es JScript.NET, no C#!

class Hello
{
    function ToString(): System.String
    {
        return 'Hi, from the engines!';
    }
}
```

Aquí tenemos una clase *Hello* que, al no indicar de quién desciende, asumimos que lo hace de *System.Object*. La nueva clase redefine la implementación del método público virtual *ToString*, que sirve como sabemos para obtener una representación textual de cualquier instancia de la clase.

## NOTA

Reconozco que tampoco se trata del viejo JScript de andar por casa. La función `ToString`, por poner un ejemplo, se define con un tipo de retorno: algo impensable en JavaScript tradicional. De hecho, vamos a algo aún más básico: aunque JavaScript es “orientado a objetos”, en un sentido más general, las clases en JavaScript clásico se crean dinámicamente con la ayuda de prototipos, y no existe una construcción `class` como la del ejemplo.

Veamos un poco de código, que si quiere, puede copiar en el constructor de la ventana principal a modo de experimento:

```
ScriptEngine eng = new ScriptEngine()
    "com.intsight.adonet://ejd20", "IntSight.AdoNet01.Ejd20");
eng.Code =
    "class Hello {" +
    "    function ToString(): System.String {" +
    "        return 'Hi, from the engines!';" +
    "    }" +
    "}";
eng.Run();
object obj = eng.CreateObject("IntSight.AdoNet01.Ejd20.Hello");
MessageBox.Show(obj.ToString());
```

He elegido un ejemplo de este tipo porque se parece a la forma en la que evaluaremos las reglas definidas por el usuario:

- Con la ayuda del motor de *script*, definimos una clase en .NET que herede de una clase conocida por el código en C#. En el ejemplo anterior, la clase base es la vulgar `object`; para evaluar reglas, usaremos una nueva clase a la que llamaremos *Regla*... y que definiremos en C#, dentro del ensamblado principal.
- Por medio de reflexión, creamos un objeto perteneciente a la clase definida en JScript. El objeto será utilizado por código escrito en C#, que no conoce el tipo exacto de la instancia.
- Pero, gracias a la herencia y el polimorfismo, el código en C# puede usar tranquilamente toda la funcionalidad definida en la clase base que sí conocemos por adelantado.

Esta técnica es elegante y lo suficientemente potente como para resolver nuestros problemas. De todos modos, no está de más que conozca otras posibilidades de Visual Studio for Applications. ¿Recuerda que hemos instanciado una tabla de *hash* dentro del constructor de `ScriptEngine`? Esta tabla nos permitirá añadir referencias a entidades globales para que puedan ser usadas en JScript como variables globales. Las entidades se añadirán con este método:

```
public void AddGlobal(string name, object value)
{
    IVsaGlobalItem item = (IVsaGlobalItem)engine.Items.CreateItem(
        name, VsaItemType.AppGlobal, VsaItemFlag.None);
    item.TypeString = value.GetType().FullName;
    globals[name] = value;
}
```

El método anterior sólo comunica al motor que debe reconocer el nombre que pasamos en el primer parámetro como una variable global del tipo asociado al valor pasado en el segundo parámetro. Para que el motor obtenga, además, la referencia a esa variable global, debemos implementar también uno de los métodos de la interfaz `IVsaSite`:

```
object IVsaSite.GetGlobalInstance(string name)
{
    return globals[name];
}
```

Creo que el uso de clases derivadas con la ayuda de reflexión es más elegante, pero reconozco que esta otra técnica puede ser más sencilla para otros posibles usos, como la automatización de objetos visuales de la aplicación desde macros.

## Reglas para determinar descuentos

El usuario experto de nuestra aplicación que quiera automatizar los descuentos en facturas tendrá que programar una clase en JScrip que debe descender de cierta clase base que programaremos en C#. Esta clase base se llamará *Regla*, y tendrá un método abstracto, *Aplicar*, que deberá examinar una factura y aplicar descuentos a las líneas que el experto considere necesario. Aunque *Regla* debe residir dentro de la propia aplicación, la ubicaremos dentro de otro espacio de nombres. ¿El motivo? Pues que el espacio de nombres al que pertenezca *Regla* deberá ser mencionado en los registros de la tabla *Reglas*, en la base de datos. ¿Se da cuenta la que podríamos liar si el nombre en cuestión cambiase para cada ejercicio?

Esta es la declaración de la clase, dentro del espacio de nombres especialmente diseñado para ella:

```
using System;
using System.Collections;
using Factura = Ej20.DsFacturas.FacturasRow;
using Linea = Ej20.DsFacturas.LineasRow;

namespace CursoAdoNet01
{
    public abstract class Regla
    {
        private Factura factura = null;

        public Regla()
        {
        }

        public decimal AplicarDescuentos(Factura factura) { ... }

        protected abstract void Aplicar();

        ...
    }
}
```

**NOTA**

Las dos cláusulas **using** resaltadas sirven para abreviar el nombre de las clases de filas generadas para el conjunto de datos con tipos asociado a *Facturas*. Como se trata de clases anidadas, el truco evitara que nos dejemos los dedos tecleando.

*Regla* tiene un constructor sin parámetros bastante tonto. Está el método abstracto *Aplicar*, que ya hemos mencionado, y un método llamado *AplicarDescuentos*, que al ser público, será el que llamaremos para aplicar la regla. Este método recibirá una factura y supondremos que actúa directamente sobre el estado de la misma. *AplicarDescuentos* no tiene plena libertad para actuar, sino que tiene que cumplir dos criterios dictados por el sentido común:

- Los descuentos no son acumulables.
- De todas las reglas aplicables, debe elegirse aquella que ofrezca mayor descuento al comprador.

Es cierto que podríamos diseñar reglas para descuentos que no cumpliesen con estos requisitos. Pero mi propósito es enseñarle la técnica, no sus aplicaciones, por lo que aceptaremos estas “reglas sobre las reglas” sin chistar. Esta es la implementación de *AplicarDescuentos*:

```
public decimal AplicarDescuentos(Factura factura)
{
    if (factura == null)
        return 0;

    this.factura = factura;
    Linea[] lineas = factura.GetLineasRows();
    decimal[] dtos = new decimal[lineas.Length];
    decimal dtoInicial = 0;
    for (int i = 0; i < lineas.Length; i++)
    {
        dtoInicial += lineas[i].Descuento;
        dtos[i] = lineas[i].Descuento;
    }
}
```

```

        lineas[i].Descuento = 0;
    }
    Aplicar();
    decimal dtoFinal = 0;
    for (int i = 0; i < lineas.Length; i++)
        dtoFinal += lineas[i].Descuento;
    if (dtoFinal < dtoInicial)
    {
        for (int i = 0; i < lineas.Length; i++)
            lineas[i].Descuento = dtos[i];
        return dtoInicial;
    }
    else
        return dtoFinal;
}

```

El método comienza guardando los descuentos actuales aplicados a la factura, para ponerlos en cero a continuación. Entonces se ejecuta el método *Aplicar*: como es un método abstracto, su implementación la tendrá que proporcionar el usuario en JScript. Luego veremos qué armas le daremos al usuario experto para que no tenga que conocer la estructura física de la factura. Una vez que ha terminado la ejecución de *Aplicar*, sumamos los descuentos para calcular el descuento total. Si éste es menor que el descuento inicial, deshacemos la acción de *Aplicar* restaurando los descuentos antes guardados.

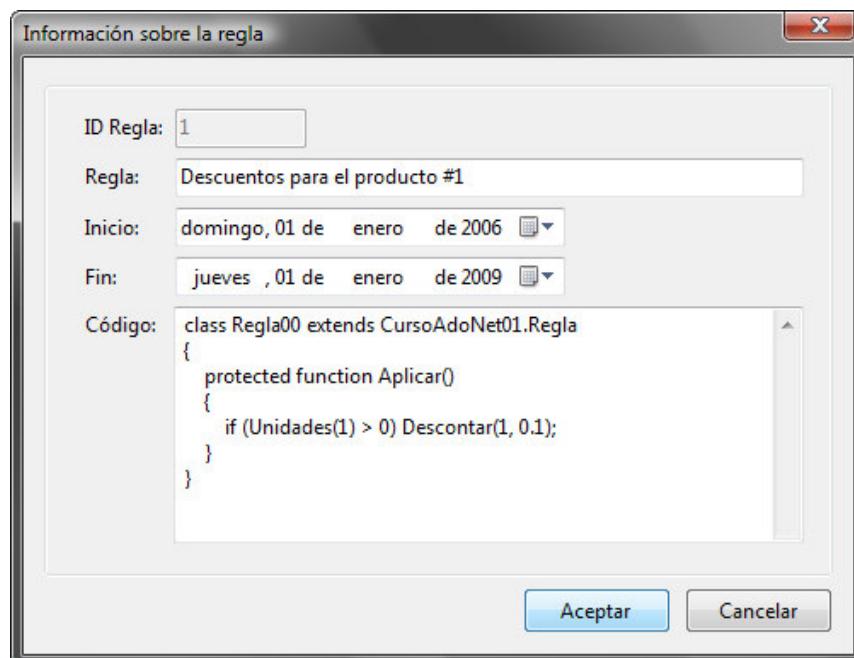
¿Qué métodos podremos usar desde JScript? Aquí le presento dos de ellos:

```

protected int Unidades(int idProducto)
{
    int total = 0;
    foreach (Linea linea in factura.GetLineasRows())
        if (linea.IDProducto == idProducto)
            total += linea.Cantidad;
    return total;
}

protected void Descontar(int idProducto, decimal descuento)
{
    foreach (Linea linea in factura.GetLineasRows())
        if (linea.IDProducto == idProducto)
            linea.Descuento += descuento * linea.Cantidad;
}

```



*Unidades* suma las cantidades solicitadas para determinado producto, teniendo en cuenta la posible existencia de líneas con el mismo producto, mientras que *Descontar* no pregunta, sino que actúa, añadiendo un importe al descuento ya existente para un producto. He creído más natural que el descuento que especificaremos en *Descontar* sea un descuento *unitario*. Esa es la razón por la que multiplicamos el parámetro por la cantidad antes de añadir el descuento a la línea de factura. La imagen anterior muestra cómo programar una regla muy sencilla, que podríamos parafrasear así:

- El producto cuyo identificador es uno, tiene un descuento de 10 céntimos.

Si quisiéramos aumentar la potencia de estas reglas de negocio, sólo tendríamos que añadir métodos como *Unidades* o *Descontar*. Por ejemplo, podríamos consultar el importe total de las facturas del cliente, o comprobar si se trata de un cliente marcado como “especial”. Lo importante es que todas estas extensiones se programarían en C#, no en JScript.

## El motor de descuentos

Ahora nos ocuparemos de otra clase, *MotorDescuentos*, que tendrá las siguientes responsabilidades:

- Construirá la instancia de *ScriptEngine* que utilizaremos para aplicar las reglas.
- Extraerá el código fuente de las reglas de la tabla correspondiente en la base de datos.
- Añadirá el código al motor de JScript y lo compilará.
- Construirá entonces un vector de objetos de tipo *Regla*, barriendo el ensamblado generado por el motor de *scripts* en busca de clases derivadas de *Regla*, y construyendo una instancia para cada tipo encontrado.
- Por último, se ocupará de evaluar todas las reglas encontradas sobre la factura sobre la que debamos aplicar descuentos automáticos.

Encontrará el código fuente de esta clase en el fichero *Data.cs*:

```
public sealed class MotorDescuentos
{
    private static ScriptEngine motor = null;
    private static object[] reglas = null;

    private static string LeerReglas()
    {
        DsReglas dsReglas = new DsReglas();
        dsReglas.Merge(Data.Instance.Read("DsReglas"), false);
        StringBuilder sb = new StringBuilder();
        foreach (DsReglas.ReglasRow row in dsReglas.Reglas)
        {
            sb.AppendLine(row.Codigo);
        }
        return sb.ToString();
    }

    private static void Inicializar()
    {
        motor = new ScriptEngine("com.intsight.adonet://ejd20",
            "IntSight.AdoNet01.Ejd20");
        motor.CompileError += ErrorCompilacion;
        motor.AddReference(
            System.Reflection.Assembly.GetExecutingAssembly().Location);
        motor.Code = LeerReglas();
        motor.Run();
        reglas = motor.CreateObject(typeof(CursoAdoNet01.Regla));
    }

    private static void ErrorCompilacion(
        object sender, CompileEventArgs e)
    {
        MessageBox.Show(e.Error.Description, "Error en script",
            MessageBoxButtons.OK, MessageBoxIcon.Error);
    }
}
```

```

public static void Aplicar(Ej20.DsFacturas.FacturasRow factura)
{
    if (motor == null) Inicializar();
    foreach (CursoAdoNet01.Regle regla in reglas)
        regla.AplicarDescuentos(factura);
}
}

```

He resaltado las instrucciones que crean el vector de reglas y lo aplican contra la factura que debemos modificar. El método *CreateObject* resaltado es una versión sobrecargada diferente de la que explicamos al presentar la clase *ScriptEngine* (p. 141). Esta versión, en vez de recibir un nombre de clase completamente cualificado para construir una instancia, recibe un tipo de clase para ser usado como ancestro, y devuelve un vector con una instancia por cada tipo encontrado.

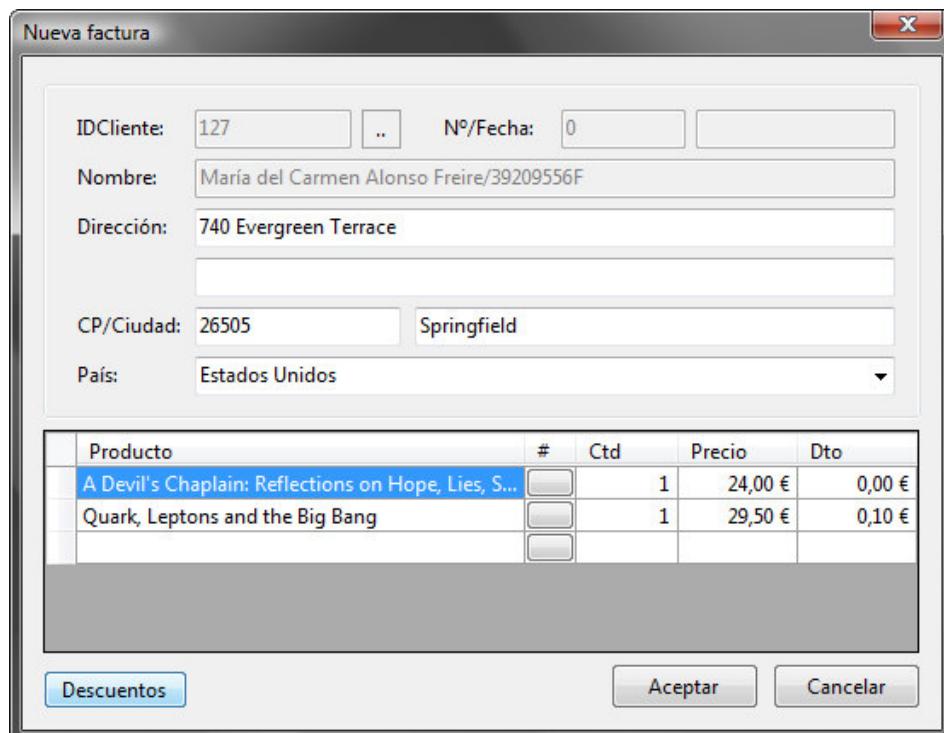
#### EJERCICIO PROPUESTO

Puede vigilar la actualización de reglas para recompilar las clases del motor de descuentos. Si la tarea se le complica demasiado, puede poner un parche advirtiendo de la necesidad de reiniciar la aplicación. Otra extensión interesante sería la validación de las reglas antes de su almacenamiento. Podría intentar la validación creando y utilizando una instancia auxiliar el motor de *scripting*.

## Aplicando las reglas

Sólo nos queda aprovechar toda la parafernalia que hemos construido para aplicar descuentos automáticos durante la edición de facturas. He creído que sería mejor disparar esta operación de manera explícita, a petición del usuario, en vez de dejar que tuviese lugar automáticamente. Así, el usuario tendría la libertad de elegir entre permitir los descuentos automáticos o hacer la vista gorda y rechazarlos.

Añada un botón en algún sitio libre del diálogo de edición de facturas y titúlelo *Descuentos*:



Intercepte su evento *Click* de la siguiente manera:

```

private void bnDescuentos_Click(object sender, System.EventArgs e)
{
    lineasBindingSource.EndEdit();
    facturasBindingSource.EndEdit();
    if (dsFacturas.Count > 0)

```

```
        MotorDescuentos.Aplicar(dsFacturas.Facturas[0]);  
    }
```

Las dos llamadas iniciales tienen el objetivo de dar por concluida la edición y transferir las actualizaciones propuestas en las *DataRowView* al conjunto de datos subyacente.

### EJERCICIO PROUESTO

Si desconfía de la naturaleza humana igual que un servidor, puede idear algún formalismo que evite la necesidad de "programar" las reglas de descuentos: un lenguaje más abstracto que pueda ser fácilmente traducido a JScript. Recuerde también que se nos habían colado algunas metarreglas en el diseño. ¿Se le ocurre alguna forma de evitarlas?

**NOTA**

A partir de Visual Studio 2005, Microsoft ha marcado el API del motor de scripts como obsoleto, sin ofrecer una alternativa. De todos modos, mis objetivos con este ejercicio eran, por una parte, mostrar cómo incorporar un API complejo como éste en una aplicación. Y en segundo lugar, inculcar la idea de que, si se desea verdaderamente que una aplicación sea extensible, no podemos basarnos únicamente en un conjunto cerrado de código y parámetros que modulen su funcionamiento.

# EJERCICIO 21

## Objetivos

Navegación sobre facturas

## Técnicas introducidas

Paginación sobre encuentros, ordenación descendente, sincronización tras las grabaciones

DESPUÉS DE UN EJERCICIO LARGO Y complicado como el anterior, creo que nos merecemos un descanso, o al menos un cambio en el ritmo de presentación. Tenemos unos cuantos temas menores pendientes, y aprovecharemos ahora para resolverlos de una vez. Estamos trabajando a ciegas con las facturas: podemos crearlas, pero no podemos ver el resultado, porque no tenemos una ventana de navegación para las facturas. Aprovecharemos esta oportunidad para ampliar la actual técnica de paginación al caso en que el criterio de ordenación sea descendente. Y resolveremos un problema que ya hemos comentado (p. 129) sobre la relectura de una factura tras su grabación exitosa.

## Es natural navegar sobre un encuentro

La tabla *Facturas*, si nos van bien las cosas, debería ser una de las tablas más grandes de la base de datos, sólo superada en cantidad de registros por la tabla de líneas de facturas. Tendremos que usar *cookies* de paginación para la navegación y búsqueda, porque nos resultará imposible traer todos los registros de golpe al lado cliente. Pero esta situación nos plantea dos retos:

- 1 Si respetamos la actual definición del conjunto de datos *DsFacturas*, tenemos que leer tres columnas de la tabla de clientes junto a los registros de facturas. Dicho de otra forma: tenemos que leer registros de un encuentro natural, y al no haber previsto la situación, no hemos definido una vista a la medida de esta necesidad.
- 2 El criterio de ordenación más natural para las facturas es usar la columna *Numero*, en sentido descendente. Nuestro algoritmo de paginación actual sólo acepta criterios ascendentes.

Una forma de resolver la primera dificultad sería, en efecto, crear una vista o una función que devolviese una tabla, como hicimos inicialmente con las búsquedas de productos. Pero prefiero mostrarle ahora otra posibilidad: no siempre podremos contar con un administrador de sistemas complaciente, que esté dispuesto a permitirnos crear cuantas vistas se nos antojen.

La técnica que usaremos está basada en las *tablas derivadas* de SQL Server, un recurso que nos permite indicar en la cláusula **from** de una consulta, en vez de una tabla física, una expresión relacional cualquiera:

```
select *
from (...expresión relacional...) as nombre
where condición
```

Recordemos el prototipo del constructor de *PageCookie* que debemos usar:

```
public PageCookie(string tablename, string table,
                  string filter, string sort, int count)
```

En el parámetro *tablename* debemos pasar un nombre de tabla de verdad, porque este valor se usa para añadir un elemento a la propiedad *TableMappings* del adaptador de datos. Ahora bien, en el segundo parámetro podemos pasar cualquier construcción sintáctica que funcione como una tabla. Ya hemos pasado tablas, vistas y funciones que devuelven tablas. Ahora pasaremos también consultas construidas al vuelo. En cualquier caso, nos conviene tener métodos como el siguiente para evitar detalles mecánicos en los que podemos equivocarnos:

```
/* Clase: PageCookie */

public static string Join(string select,
                          string table1, string table2, string condition)
{
    return new StringBuilder()
```

```

        .Append(" (select ") .Append(select)
        .Append(" from ") .Append(table1)
        .Append(" inner join ") .Append(table2)
        .Append(" on ") .Append(condition)
        .Append(") as ") .Append(table1)
        .ToString();
    }
}

```

Los cambios no se detienen en la tabla maestra: necesitamos traer el nombre del producto con las líneas de facturas. Hay que uniformizar los parámetros que pasamos para la tabla de detalles en el segundo constructor de *PageCookie*. Hasta ahora pasábamos sólo un nombre para esta tabla, pero de aquí en adelante separaremos el nombre de la tabla de la expresión relacional que se incluirá en la instrucción SQL que generemos. El cambio es más tedioso que otra cosa, porque hay que propagar los cambios en el constructor a las clases *ReturnedCookie*, *CustomerCookie* y *ProductCookie*.

Antes de dar por zanjado el incidente, quiero echar un vistazo a las causas por la que estamos usando un encuentro natural para traer columnas asociadas al cliente de la factura. La sabiduría popular en ADO.NET establece que, cuando trabajamos con columnas de referencia, es mejor traer aquellos registros de la tabla de referencia que están involucrados antes que leer mediante un encuentro natural. Suponga que necesitamos esta información:

```

select f.* , c.Nombre, c.Apellidos, c.NIF
from Facturas f inner join Clientes c
      on f.IDCliente = c.IDCliente
where Importe > 1000

```

La recomendación de los “expertos” en ADO.NET es dividir la consulta en dos:

```

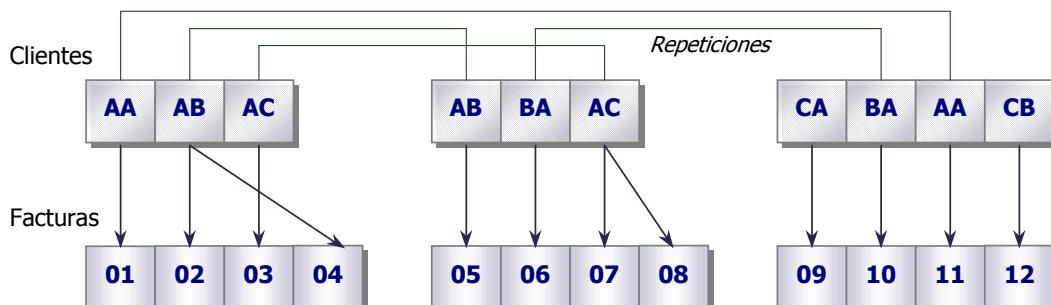
select *
from Facturas
where Importe > 1000

select c.IDCliente, c.Nombre, c.Apellidos, c.NIF
from Clientes c
where IDCliente in (
    select IDCliente
    from Facturas
    where Importe > 1000)

```

¿La explicación? Si como norma hay más de una factura por cliente, en la consulta basada en el encuentro tendremos mucha información redundante, mientras que la segunda técnica sólo trae la información estrictamente necesaria. Es posible que evaluar las dos consultas de la segunda técnica cueste ligeramente más que el encuentro original, pero la pérdida se compensa sobradamente con el menor tráfico en red. En otras palabras, hemos elegido la que parece ser la peor técnica posible...

... si no fuese por la paginación. Recuerde que estamos recuperando registros de facturas en grupos. Si hiciéramos caso a la sabiduría popular, se produciría la siguiente situación:



Para cada grupo, habría que traer los clientes asociados a facturas de ese grupo, y no podríamos descartar los clientes de facturas anteriores, por la evidente complejidad que implicaría. Como resultado, donde normalmente nos bastaría con traer seis clientes, hemos terminado trayendo diez.

## Cambio de sentido

Es mucho más sencillo admitir cláusulas de dirección en el criterio de ordenación, esto es, en el parámetro *sort* de los constructores de *PageCookie* y sus descendientes. El valor de *sort* se utiliza para dos tareas: para generar la cláusula **order by** de la consulta, y para sintetizar la condición de continuación, en un método llamado *SynthesizeExpression*, dentro de la clase *Data*. Dentro de este método, el contenido de *sort* se descompone en las columnas que lo forman, para crear la condición a partir de esas columnas y sus valores en la última fila leída. Por lo tanto, lo único que tenemos que hacer es asegurarnos de que las cláusulas **asc** y **desc** desaparezcan al descomponer en criterio de ordenación en columnas sueltas:

```
protected string SynthesizeExpression(DataTable t, string columns)
{
    DataRow row = t.Rows[t.Rows.Count - 1];
    StringBuilder sb = new StringBuilder();
    StringBuilder acc = new StringBuilder();
    foreach (string cname in columns.Split(','))
    {
        string cn = cname.ToLower().Trim();
        bool descending = cn.EndsWith("desc");
        if (descending)
            cn = cn.Substring(0, cn.Length - 4);
        else if (cn.EndsWith("asc"))
            cn = cn.Substring(0, cn.Length - 3);
        if (acc.Length > 0)
        {
            sb.Append(" or ").Append(acc).Append(" and ");
        }
        DataColumn column = t.Columns[cn.Trim()];
        string val = SqlValue(row[column], column);
        sb.Append(column.ColumnName)
            .Append(descending ? " < " : " > ")
            .Append(val);
        if (acc.Length > 0) acc.Append(" and ");
        acc.Append(column.ColumnName).Append(" = ").Append(val);
    }
    return sb.ToString();
}
```

Observe que también debemos invertir el sentido de la comparación cuando encontramos una columna ordenada con criterio descendiente. No olvide, por cierto, que SQL permite criterios de ordenación con direcciones mezcladas.

## Búsqueda y navegación sobre facturas

Ya estamos preparados para crear la ventana de navegación sobre facturas. Cree un control que herede de *BaseWindow*, y llámelo *Facturas*. Decórelo con el atributo *NonModal* para que aparezca automáticamente en el menú principal, y no olvide asociarle como editor la clase *DlgFactura*. Traiga un conjunto de datos *DsFacturas* al control heredado y modifique la propiedad *DataSet* del control para que apunte al conjunto de datos. Asigne la cadena *Facturas* en la propiedad *DataMember*, para que el diálogo de edición sepa que es ésta, y no *Lineas*, la tabla principal que debe editar. Y añada un botón *bnMore* a la barra de herramientas para recuperar el siguiente grupo de registros según corresponda.

Para cargar el primer grupo de facturas al abrir la ventana, redefina la implementación de *InitData*, el método heredado de *BaseWindow* que pertenece a la interfaz *IWindow*:

```
cookie = new PageCookie("Facturas",
    PageCookie.Join(
        "Facturas.*", "Clientes.Nombre, Clientes.Apellidos, Clientes.NIF",
        "Facturas", "Clientes", "Facturas.IDCliente=Clientes.IDCliente"),
    "", "Número desc", 20,
    "Lineas",
```

```

PageCookie.Join(
    "Lineas.*", Productos.Producto", "Lineas", "Productos",
    "Lineas.IDProducto = Productos.IDProducto"),
    "IDFactura", "IdFactura");
dsFacturas.Merge(Data.Instance.Read("DsPaises"), false);
dsFacturas.Merge(Data.Instance.Read(ref cookie), false);
bnMasRegistros.Enabled = !cookie.Disabled;

```

He resaltado los encuentros naturales que se utilizan en vez de las tablas ramplonas de facturas y líneas. Cuando se pulse el botón *bnMore* para traer más facturas a la ventana, tendremos que ejecutar el siguiente método:

```

private void bnMasRegistros_Click(object sender, EventArgs e)
{
    if (!cookie.Disabled)
    {
        dsFacturas.Merge(Data.Instance.Read(ref cookie), false);
        bnMasRegistros.Enabled = !cookie.Disabled;
    }
}

```

Número	Nombre	Apellidos	N.I.F.	Ciudad	Base imponible	Impuestos	Importe	Creación
8	Maria del Carmen	Alonso Freire	39209556F	Springfield	53,40 €	2,14 €	55,54 €	21/12/2007 21:23
7	Begoña	Albizu del Pozo	08707363T	Sevilla	599,90 €	24,00 €	623,90 €	21/12/2007 4:52
6	Maria del Carmen	Alonso Freire	39209556F	Springfield	38,78 €	1,55 €	40,33 €	21/12/2007 2:54
5	Francisco	Arias Collado	17327312D	Barcelona	44,99 €	1,80 €	46,79 €	21/12/2007 2:50
4	Carmen	Aparicio Pérez	52366904K	Sevilla	39,99 €	1,60 €	41,59 €	21/12/2007 2:50
3	Ramón	Aparicio Pintado	48445170V	Madrid	49,99 €	2,00 €	51,99 €	21/12/2007 2:48
2	Ramón	Aparicio Pintado	48445170V	Madrid	39,99 €	1,60 €	41,59 €	21/12/2007 2:48
1	Carmen	Aparicio Pérez	52366904K	Sevilla	173,71 €	6,95 €	180,66 €	21/12/2007 2:46

Por último, necesitamos hacer un pequeño cambio en el formulario *DlgFactura*, para que al ejecutarse en modo de actualización, copie a su conjunto de datos interno las filas correspondientes a las líneas de factura:

```

public override void InitData(IDataWindow parent, bool newRow)
{
    dsFacturas.Merge(Data.Instance.Read("DsPaises"), false);
    base.InitData(parent, newRow);
    if (!newRow)
    {
        DataRowView drv = (DataRowView)parent.BindingSource.Current;
        foreach (DataRow r in drv.Row.GetChildRows("FacturasLineas"))
            dsFacturas.Lineas.ImportRow(r);
        dsFacturas.AcceptChanges();
    }
}

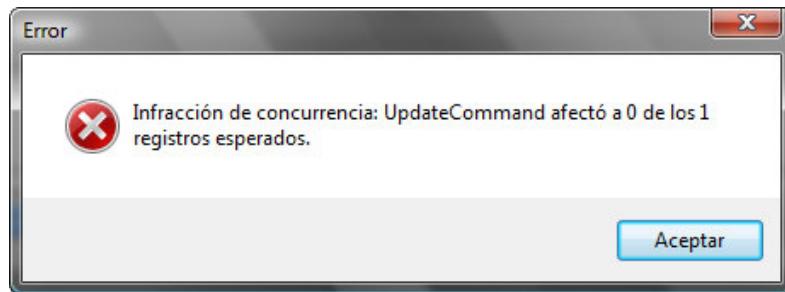
```

El código añadido es casi idéntico al usado en el diálogo de edición de clientes. Sólo hemos cambiado el nombre de las tablas y las relaciones.

## La versión más reciente de la factura

Estando abierta la ventana de navegación sobre facturas, invoque el diálogo de altas de facturas desde el botón presente en la ventana de navegación. Cree una factura simple, con un par de líneas y pulse el botón de aceptar para cerrar el diálogo. Verá que el nuevo registro aparece automáticamente en la ventana de navegación: así lo hemos programado en la clase base. Si presta atención verá que los campos *Importe*, *Base* e *Impuestos* aparecen con ceros. ¿El motivo? Pues que estos campos se actualizan mediante *triggers*, y no reciben sus valores finales hasta que se graba la última línea de la factura. Sin embargo, la versión más reciente de la factura la hemos leído inmediatamente después de crear la cabecera, antes de grabar la primera línea. Otro detalle, que no podrá comprobar directamente: el valor de la columna *TS* de la cabecera es incorrecto. Para comprobarlo, edite el

nuevo registro, modifique cualquier campo de la cabecera e intente la grabación. Le apuesto un euro que recibirá un mensaje de error como el siguiente:



En cambio, el error no se produce si cierra la ventana de navegación y vuelve a abrirla. En ese caso, habría recuperado la última versión de la factura.

Vamos a escaparnos por el camino del mínimo esfuerzo. Busque el método auxiliar *GuardarFacturas*, dentro de la clase *Data*. ¿Por qué no pasamos el *delta* por un filtro antes de devolverlo al diálogo de grabación?

```
protected DataSet GuardarFacturas(DataSet delta)
{
    using (Transaccion trans = new Transaccion(
        sqlConn, daFacturas, daLineas)) {
        :
        return ReleerFactura(delta);
    }
}
```

El “filtro”, en realidad, releerá todas las filas en el conjunto de datos que recibe como parámetro. Para no tener que reescribir todos los encuentros naturales, he creado una clase *InvoiceCookie* dentro del fichero *Traductor.cs*, que encapsule la llamada al constructor de *PageCookie* necesaria para leer cabeceras y líneas de facturas. Esta es la implementación de *ReleerFactura*:

```
protected DataSet ReleerFactura(DataSet delta)
{
    StringBuilder sb = new StringBuilder();
    foreach (DataRow row in delta.Tables["Facturas"].Rows)
    {
        if (sb.Length > 0) sb.Append(", ");
        sb.Append(row["IDFactura"].ToString());
    }
    string filtro = sb.ToString();
    if (filtro != "")
    {
        PageCookie c = new InvoiceCookie(
            "IDFactura in (" + filtro + ")", 10000);
        delta.Merge(Read(ref c), false);
    }
    return delta;
}
```

Para ser sincero, éste no es el mejor sitio para averiguar cuáles registros debemos volver a leer, porque ya hemos perdido la distinción entre facturas nuevas y facturas modificadas... y sólo necesitamos releer las primeras.

 Un detalle *muy importante*: Cuando ponía a punto el ejercicio, encontré de repente que las modificaciones de facturas existentes provocaban un fallo en las restricciones del conjunto de datos con tipos. Me costó algo de trabajo dar con el motivo: al generar el conjunto de datos para *DsFacturas*, Visual Studio fue capaz de detectar correctamente *IDFactura* como clave única... pero no como clave primaria. Por esta causa fallaban las llamadas a *Merge*, que se basan en el conocimiento de la clave primaria. Para corregirlo, sólo tuve que editar la clave existente en el editor gráfico para XSD, y marcarla también como clave primaria.

# EJERCICIO 22

## Objetivos

Mensajes de error

## Técnicas introducidas

Restricciones con nombres

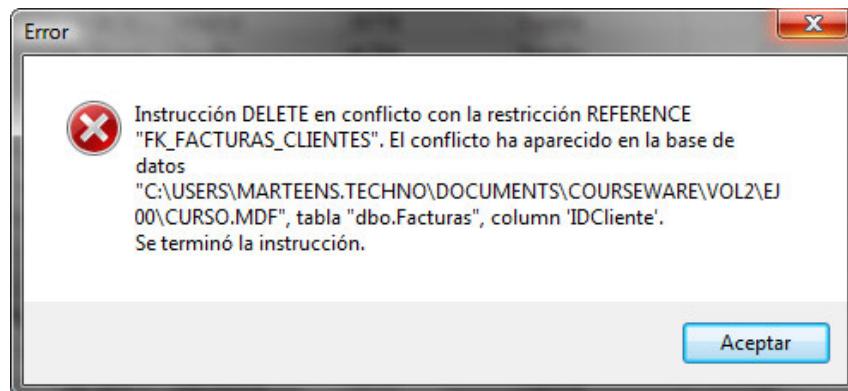
**E**L MANEJO DE ERRORES MEDIANTE EXCEPCIONES ES una técnica que debería simplificar la programación. Lamentablemente, varias causas contribuyen a que pocos programadores comprendan cómo deben programar el control de errores. Por ejemplo, quienes vienen del Visual Basic clásico arrastran consigo un sistema estrambótico de control de errores, herencia de los tiempos en que VB era un lenguaje interpretado. Incluso quienes tienen experiencia en C++ han aprendido el uso de excepciones con aplicaciones que no tienen un bucle de mensajes. El resultado son aplicaciones con demasiado código dedicado al control de errores.

## Restricciones con nombre

La mejor estrategia para el control de excepciones es no “controlarlas”: en casi todos los casos, es mejor dejar flotar la excepción hasta el bucle de mensajes, y dejar que éste se encargue de notificar la excepción al usuario. Es cierto que Windows Forms utiliza un diálogo de notificación de errores demasiado aparatoso, que permite incluso terminar prematuramente la aplicación. Pero ya hemos visto cómo usar nuestro propio diálogo interceptando el evento estático *ThreadException*, definido dentro de la clase *Application*. También debo reconocer que hay partes de .NET que consumen excepciones que deberían flotar libremente, por un mal diseño de los programadores de esas clases.

No obstante, hay algunos casos en que no nos interesa tanto modificar la propagación de una excepción como cambiar su mensaje de texto. No se trata de traducir el mensaje: la traducción, en general, debe realizarse mediante el mecanismo más general de “localización”. El problema al que me refiero bien podría bautizarse como *contextualización*: adaptar el mensaje de error a las circunstancias concretas que lo provocaron.

Para ganar tiempo, ejecute la aplicación y active la ventana de navegación sobre clientes. Localice un cliente que esté asociado a una factura e intente eliminarlo. Debe aparecer entonces un mensaje de error como el siguiente:



El mensaje está en español, pero parece redactado en chino! De paso, observe la referencia a la restricción *FK\_FACTURAS\_CLIENTES*, que hemos definido para la tabla de facturas:

```
primary key      (IDFactura),
constraint UQ_FACTURAS_NUMERO
    unique        (Numero),
constraint FK_FACTURAS_CLIENTES
    foreign key   (IDCliente) references dbo.Clientes(IDCliente),
constraint FK_FACTURAS_PAISES
    foreign key   (IDPais) references dbo.Paises(IDPais)
```

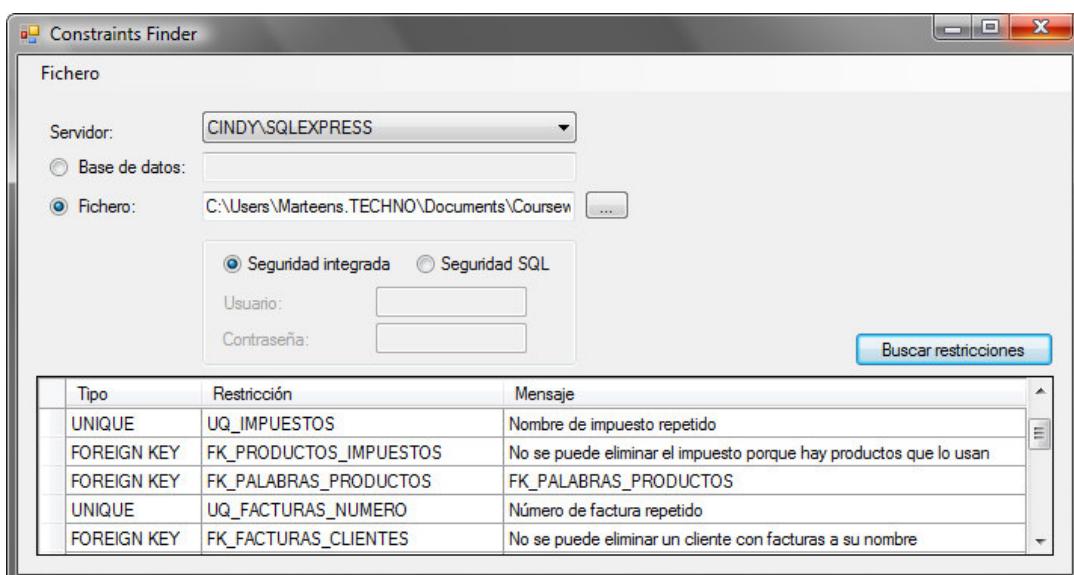
## Adaptando los mensajes al contexto

Estamos de acuerdo en que el mensaje anterior no es muy apropiado, pero ¿qué deberíamos mostrar en su lugar? Si el error se ha producido al intentar borrar un cliente con facturas, ¿qué tal algo como ‘No puede eliminar este cliente, porque tiene facturas a su nombre’?

Ahora bien, el problema no consiste en buscar un mensaje más apropiado, sino en cómo sustituir esos mensajes sin llenar el código fuente de instrucciones **try/catch**. El primer impulso del programador es precisamente ése: intentar “proteger” la llamada a la instrucción que elimina el cliente dentro de un **try/catch**... y repetir la jugada en todos los puntos similares. Además, ¿cómo saber qué tipo exacto de excepción es el que tenemos entre manos? Llegados a este punto, la gente comienza a sospechar de la existencia de un manuscrito apócrifo con códigos de error, atesorado dentro de una botella emparedada dentro de un monasterio bantú de monjes budistas.

El problema es que no existe tal manuscrito. Incluso en el caso de que existiese, no le recomendaría usar dicha información. Mi propuesta es más sencilla, y consiste en capturar estos mensajes en un punto único del proyecto, para evitar la proliferación innecesaria de código. Además, vamos a localizar las excepciones que queremos cambiar analizando el texto de sus mensajes. La clave está en usar siempre restricciones con nombres al definir la base de datos. Como el nombre de la excepción se incluye dentro del mensaje de error, siempre que utilicemos nombres significativos para las restricciones, podremos aplicar nuestra técnica.

La siguiente imagen corresponde a una aplicación auxiliar, cuyo código fuente encontrará dentro del directorio *Ej22/ConstraintsFinder*:



Esta aplicación crea un fichero en formato XML con todas las restricciones con nombres que encuentre en una base de datos. Las restricciones son leídas mediante la siguiente instrucción, que hace referencia a una vista del sistema en SQL Server:

```
select Constraint_name, Constraint_type
from INFORMATION_SCHEMA.TABLE_CONSTRAINTS
```

Inicialmente, la columna titulada *Mensaje* contiene el mismo nombre de la restricción, pero la aplicación auxiliar le permite modificar ese mensaje a su antojo, como puede ver en la misma imagen. La aplicación también permite guardar el fichero con el nombre que prefiera, y abrirlo para retomar una sesión de edición ya iniciada.

Regresando a la aplicación del ejercicio, ¿cuál es el mejor lugar para modificar los mensajes de excepciones? Está claro que debemos hacerlo dentro de la respuesta al evento *ThreadException* de la clase *Application*. Abra el fichero *Program.cs* y modifique la implementación del método *OnException*

como muestro a continuación, para delegar la traducción de los mensajes de excepciones a otro método auxiliar, que bautizaremos como *TraducirError*:

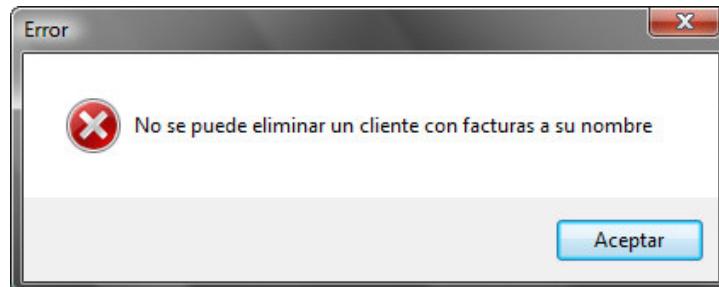
```
private static void OnException(object sender,
    ThreadExceptionEventArgs e)
{
    string msg = e.Exception.Message;
    try {
        msg = TraducirError(msg);
    }
    catch {}
    MessageBox.Show(msg, "Error",
        MessageBoxButtons.OK, MessageBoxIcon.Error);
}
```

Observe que hay un bloque **try/catch** algo inusual para nuestro estilo de programación. En este caso, estamos escribiendo un manejador de excepciones. Si se produce un fallo, corremos el riesgo de entrar en un bucle recursivo infinito. De ahí nuestra precaución para impedir el disparo de una excepción dentro de este método.

La implementación de *TraducirError* busca un fichero *mensajes.xml* en el mismo directorio de la aplicación. Si lo encuentra, intenta abrirlo como conjunto de datos. El fichero debe contener pocas filas, por lo que no hay mayor inconveniente para recorrer sus registros de forma lineal. Si localizamos un nombre conocido de restricción en la primera columna de la tabla en memoria, devolvemos como resultado el valor de la segunda columna. En caso de no encontrar un registro apropiado, devolvemos el mensaje original:

```
private static string TraducirError(string msg)
{
    string filename = Path.Combine(
        Application.StartupPath, "mensajes.xml");
    if (File.Exists(filename))
        using (DataSet ds = new DataSet())
    {
        ds.ReadXml(filename, XmlReadMode.ReadSchema);
        foreach (DataRow row in ds.Tables[0].Rows)
            if (msg.IndexOf(row[0].ToString()) != -1 &&
                row[0].ToString() != row[1].ToString())
                return row[1].ToString();
    }
    return msg;
}
```

Y el resultado es el que muestro a continuación:



Una variante de esta técnica consiste en mantener el conjunto de datos con los mensajes de error cargado en memoria, a partir de la primera vez que lo abrimos. Si lo desea, puede experimentar con esta variante.

# EJERCICIO 23

## Objetivos

Recuperación de errores de concurrencia

## Técnicas introducidas

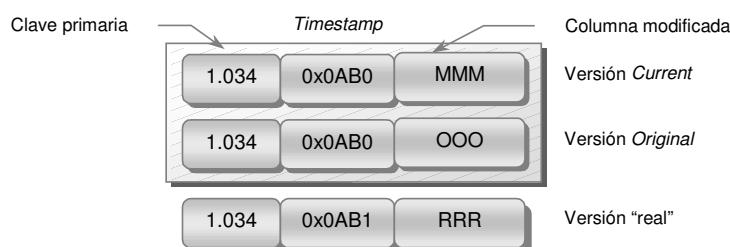
Relectura, mezcla de versiones de registros

**L**A DIFERENCIA ENTRE UNA APLICACIÓN profesional y el fruto del esfuerzo de un aficionado está en los detalles; incluso diría en los pequeños detalles. Si programamos ateniéndonos a la ayuda en línea y a los libros de la primera oleada, podremos detectar conflictos provocados por actualizaciones concurrentes. Sin embargo, no es tan sencillo corregir un fallo de este tipo en una grabación, y repetir la operación, con los cambios indispensables. Esto puede ser aceptable en algunos casos, pero una aplicación bien programada debe ir más allá.

## Casos especiales

Además, nuestro mecanismo de control de concurrencia prohíbe también operaciones aceptables en otras circunstancias. Un claro ejemplo: actualizaciones concurrentes que afectan a diferentes columnas. Alicia modifica la fecha de contratación de un empleado, mientras Benito actualiza el sueldo del mismo individuo. ¿Hay algún peligro en consentir ambas operaciones a la vez? Ninguno... excepto en el caso en que existan reglas que vinculen la antigüedad del empleado con su salario. Pero incluso en tal situación, la actualización concurrente seguiría siendo segura bajo la condición de que el servidor SQL verifique la regla mediante reglas, *triggers* o cualquier otro mecanismo equivalente.

Incluso cuando se produce un conflicto verdadero en la actualización de un registro, puede que nos interese tomar medidas para que el usuario cuya actualización se ha rechazado pueda decidir si desea repetir la operación o no. Alicia y Benito intentan actualizar el salario de Carlos al mismo tiempo. Alicia gana la partida y Benito recibe un mensaje de error. Sin embargo, Benito considera que es él quien tiene realmente potestad para modificar dicha columna. Con nuestra técnica actual de grabación, Benito tendría que cancelar la ejecución del diálogo de modificación y repetir la operación desde el principio... porque la causa del fallo es el valor de la versión de fila almacenado en la copia local del registro, que ya no está en sincronía con el valor real en la base de datos:



Como mínimo, para poder repetir la operación, necesitaríamos leer el valor real de la columna que contiene la versión de fila (*timestamp*) y modificar así la copia local que se intenta grabar. A partir de este punto, se disparan las variantes del algoritmo:

- 1 ¿Debemos realizar esta operación de forma automática? Mi opinión es que no: deberíamos advertir al usuario sobre el problema, y dejar la decisión en sus manos.
- 2 ¿Y si todas las columnas modificadas por el usuario A son diferentes de las modificadas por el usuario B? En ese caso, podríamos realizar la “mezcla” de las actualizaciones de manera automática: debe decidir el programador. El autor es partidario de la mezcla automática.
- 3 ¿Qué ocurre con las columnas modificadas por otro usuario, que no han sido modificadas por el usuario víctima del conflicto? Está claro que deberíamos respetar esos valores modificados desde otro puesto.

## Cómo distinguir los errores de concurrencia

- 357 Lo más importante, para poder empezar, es saber cómo distinguir los errores provocados por actualizaciones concurrentes de todos los demás. En este caso, tenemos suerte: existe una clase específica de excepciones, *DbConcurrencyException*, para este tipo de conflictos. Como hemos sido buenos chicos, nuestro código está organizado de tal manera que tendremos que vigilar las grabaciones en un único lugar del proyecto: el método *SaveChanges* de la clase *BaseDialog*. Y en particular, la instrucción que puede fallar es la llamada al método *Write* del módulo de acceso a datos.

```
try
{
    delta = Data.Instance.Write(dataset.DataSetName, delta);
}
catch (System.Data.DbConcurrencyException)
{
    :
}
```

... y a usted le extrañará que haya señalado tan rápidamente este punto caliente. ¿No podríamos hacer algo dentro del propio módulo de acceso a datos? La respuesta es *no*... al menos si pretendemos que el usuario dé su opinión sobre lo que debe suceder a continuación del fallo. Recuerde que este módulo de acceso a datos debe poder moverse a un servidor de capa intermedia, que estaría fuera del alcance del usuario.

**NOTA** No pierda de vista, sin embargo, que se trata de una encrucijada: podemos decidir que *algunos* tipos de conflictos de concurrencia se resuelvan automáticamente, sin intervención del usuario. No es la vía que seguiremos, pero es una posibilidad.

Mi propuesta es la siguiente:

- 1 Cuando se produzca un conflicto de actualización, avisaremos al usuario, y le pediremos que confirme si quiere aplicar sus cambios, de todos modos, o si se arrepiente y abandona.
- 2 Si el usuario decide reintentar la grabación, reenviaremos sus cambios al módulo de acceso y actualización. Dentro de este módulo, de manera automática, intentaremos “modificar las modificaciones” de la forma más sensata posible.

Observe que la pregunta que haremos al usuario será muy sencilla. Dentro de la clase *BaseDialog* definiremos el siguiente método para que se ocupe de dicha tarea:

```
protected virtual bool ConfirmarRepeticion()
{
    return MessageBox.Show(
        "Este registro ha sido actualizado por otro usuario.\n" +
        "¿Desea repetir la operación?", "Error de acceso simultáneo",
        MessageBoxButtons.YesNo,
        MessageBoxIcon.Question) == DialogResult.Yes;
}
```

¿Pregunta sencilla? Bueno, habría que comparar con lo que proponen algunos autores: presentar al usuario el estado *original* del registro, el estado *actual* del mismo en la base de datos, y el estado que el usuario quiere guardar. Tres versiones del mismo registro. ¿Cree usted que algún usuario podría reprimir las ganas de aporrear el monitor? Mi pregunta, por el contrario puede resumirse así: *¿está usted convencido de que sus datos son correctos?* Es cierto que una pregunta de este tipo pide a gritos una respuesta afirmativa, pero recuerde que he prometido realizar una mezcla razonable de las modificaciones.

Un rasgo importante que debe poseer nuestro algoritmo es una cantidad mínima de movimiento de información entre capas. Si la grabación triunfa en el primer intento, habrá un único viaje de ida y vuelta. Si la grabación falla y el usuario no quiere reintentárla, no hay más intercambio. Y si se reintenta, hay solamente un viaje de ida y vuelta adicional. De hecho, vamos incluso a mantener la interfaz pública actual del módulo de acceso a datos. Simplemente asumiremos que algunas grabaciones requerirán una relectura de los datos más recientes antes de su ejecución, para mezclarlos con los

cambios explícitamente indicados por el usuario. Para distinguir ese caso especial, en vez de añadir parámetros al método *Write*, incluiremos información adicional dentro del propio parámetro *delta*; es decir, dentro del conjunto de datos diferencial que el usuario envía al módulo de acceso a datos.

La clave del truco está en la propiedad *ExtendedProperties*, de la clase *DataSet*:

```
public PropertyCollection ExtendedProperties { get; }
```

Esta propiedad es un diccionario implementado sobre una tabla de *hash*, que aguanta lo que le echemos. Cuando necesitemos releer el registro antes de la grabación, enviaremos un *delta* que contenga una clave *force* con el valor **true**. El nombre de la clave, por supuesto, es arbitrario. Esta es la nueva implementación de *SaveChanges*:

```
public virtual bool SaveChanges()
{
    DataSet delta = DataSet.GetChanges();
    try
    {
        delta = Data.Instance.Write(DataSet.DataSetName, delta);
    }
    catch (DBConcurrencyException)
    {
        if (ConfirmarRepeticion())
        {
            delta.ExtendedProperties["force"] = true;
            delta = Data.Instance.Write(DataSet.DataSetName, delta);
        }
    }
    foreach (DataTable tab in identityTables)
        foreach (DataRow row in tab.Select(
            null, null, DataViewRowState.Added))
            row.Delete();
    DataSet.Merge(delta, false);
    if (parentWindow != null)
    {
        parentWindow.DataSet.Merge(delta, false);
        parentWindow.DataSet.AcceptChanges();
    }
    return true;
}
```

## Relectura y mezcla

Vamos ahora a la clase *Data*, nuestro módulo de acceso a datos. Primero filtraremos el parámetro *delta*, pasándolo a través de un método *Releer*. Si detectamos la petición de relectura, este método se encargará simultáneamente de la relectura y de la mezcla de valores en los registros implicados:

```
public DataSet Write(string dsname, DataSet delta)
{
    IniciarNotificaciones();
    if (dsname == "DsPaises")
        delta = GuardarPaises(Releer(delta, "paises"));
    else if (dsname == "DsImpuestos")
        daImpuestos.Update(Releer(delta, "impuestos"));
    else if (dsname == "DsProductos")
        daProductos.Update(Releer(delta, "productos"));
    else if (dsname == "DsReglas")
        daReglas.Update(Releer(delta, "reglas"));
    else if (dsname == "DsClientes")
        delta = GuardarClientes(Releer(delta, "clientes"));
    else if (dsname == "DsFacturas")
        delta = GuardarFacturas(Releer(delta, "facturas"));
    else
        throw new ApplicationException(
            "La operación no ha sido implementada.");
    EnviarNotificaciones();
```

```

        return delta;
    }
}

```

El esquema general de *Releer* es muy sencillo, y casi toda la acción real tiene lugar en dos métodos auxiliares, que veremos enseguida. Este es el código de *Releer*:

```

private DataSet Releer(DataSet delta, string tableName)
{
    if (delta.ExtendedProperties.ContainsKey(S_FORCE) &&
        (bool) delta.ExtendedProperties[S_FORCE] == true)
        using (SqlDataAdapter da = new SqlDataAdapter(
            GenerarSelect(delta.Tables[tableName]), sqlConn))
    {
        DataSet ds = new DataSet();
        da.MissingSchemaAction = MissingSchemaAction.AddWithKey;
        da.Fill(ds, delta.Tables[tableName].TableName);
        Remezclar(delta.Tables[tableName], ds.Tables[0]);
        delta.Merge(ds, true);
    }
    return delta;
}

```

Este método recuerda en parte el mecanismo de relectura de facturas que ya está implementado. Ahora necesitamos un método de generación automática de consultas más general, y ése es el papel de *GenerarSelect*:

```

private string GenerarSelect(DataTable table)
{
    StringBuilder sb = new StringBuilder();
    sb.Append("select * from ");
    sb.Append(table.TableName);
    sb.Append(" where ");
    int i = 0;
    DataColumn col = table.PrimaryKey[0];
    foreach (DataRow row in table.Rows)
    {
        if (i > 0) sb.Append(" or ");
        sb.Append(col.ColumnName).Append(" = ");
        sb.Append(SqlValue(row[col], col));
    }
    return sb.ToString();
}

```

Le advierto que hay una simplificación importante: para no enmarañar el código, he supuesto que todas las claves primarias constan de una sola columna. De todos modos no es complicado eliminar esta restricción del método anterior. Observe también el uso del método *SqlValue*, que introdujimos en ejercicios anteriores (p. 56).

Concentrémonos en la operación de mezcla:

```

Remezclar(delta.Tables[tableName], ds.Tables[0]);
delta.Merge(ds, true);

```

Nos falta ver qué es lo que hace *Remezclar*, pero hay que notar que la llamada final a *Merge* nos evita complicar excesivamente su algoritmo:

```

private void Remezclar(DataTable delta, DataTable original)
{
    object[] keys = new object[delta.PrimaryKey.Length];
    foreach (DataRow r1 in original.Rows) {
        for (int i = 0; i < original.PrimaryKey.Length; i++)
            keys[i] = r1[original.PrimaryKey[i]];
        DataRow r2 = delta.Rows.Find(keys);
        if (r2 != null)
            foreach (DataColumn c1 in original.Columns) {
                DataColumn c2 = delta.Columns[c1.ColumnName];
                if (!c2.ReadOnly &&
                    r2[c2, DataRowVersion.Original].Equals(
                        r2[c2, DataRowVersion.Current]))

```

```

        r2[c2] = r1[c1];
    }
}
}

```

Para cada fila releída, buscamos su equivalente en el conjunto de datos diferencial. Luego, a cada columna del conjunto *delta* que no haya sido modificada explícitamente por el usuario, le asignamos su valor más reciente recién leído desde la base de datos. Esta operación sólo retoca las versiones propuestas (en realidad, la versión *Current*) de las columnas, pero nos falta modificar las *Original*. Por ese motivo es que ejecutamos *Merge* a continuación:

```
delta.Merge(ds, true);
```

El segundo parámetro se titula *preserveChanges*, y cuando vale **true**, la mezcla tiene lugar precisamente sobre la versión *Original* de los registros afectados, respetando cualquier cambio explícito. 359

¿Qué tal funciona nuestro algoritmo? Es complicado predecirlo de modo abstracto, por lo que es mejor simular su funcionamiento. Supongamos que Alicia y Benito (¡no, no conozco a la tal Alicia y al tal Benito, es una forma de evitar hablar de los usuarios *A* y *B*!) leen el siguiente registro:

```
<1, "Ian", "Marteens", 0x0ABC>
```

La cuarta columna corresponde a la versión de fila: la columna **timestamp** que estamos usando para implementar el bloqueo optimista. Alicia, que es más rápida que Billy the Kid (¡otro candidato a usuario *B*!), se apresura a modificar el nombre, con lo que la base de datos contendrá la siguiente versión del registro:

```
<1, "Juan", "Marteens", 0x0ABD>
```

Benito sustituye *Marteens* por *Martínez*, pero su intento de grabación falla... ¡porque la versión de fila en su copia local no coincide con la real! Recuerde que cada vez que modificamos un registro, el valor de su versión de fila avanza. Supongamos ahora que Benito insiste en traducir el apellido. El algoritmo de mezcla se enfrentará a estas dos versiones:

```
<1, "Juan", "Marteens", 0x0ABD>: versión real
<1, "Ian", "Martínez": "Marteens", 0x0ABC>: intento de grabación
```

He resaltado las columnas modificadas en cada fila, y en el caso del registro de Benito, he indicado también el valor original de la columna modificada. Note también las diferencias en la versión de fila. Cuando *Remezclar* se ejecuta, el registro resultante se parece al siguiente:

```
<1, "Juan": "Ian", "Martínez": "Marteens", 0x0ABC>
```

Esto nos permitirá respetar la modificación efectuada por Alicia en el nombre, y además, mantener el cambio en el apellido. La versión de fila no se ha modificado: ¡es una columna de sólo lectura! Y seguimos teniendo un problema: la instrucción **update** que generará el adaptador de datos seguirá usando el valor *original* de la versión de fila, que sigue sin ser actualizada. No obstante, este valor se modificará como corresponde con la posterior llamada a *Merge*:

```
<1, "Juan": "Ian", "Martínez": "Marteens", 0x0ABC: 0x0ABD>
```

Ahora he subrayado el valor de la versión *Original*, porque es la que se utilizará para generar la instrucción **update**. Está claro que este registro podrá grabarse sin problemas... siempre que no se haya metido otro usuario por medio. El resultado final de todas estas manipulaciones será el tendrá el siguiente aspecto:

```
<1, "Juan", "Martínez", 0x0ABE>
```

Es decir, habremos modificado tanto el nombre como los apellidos, y el número de versión, como es lógico, ha avanzado un paso más. Incluso tenemos la fortuna de que el ordenador de Benito contendrá ahora la versión actualizada del registro, que contiene los dos cambios. Es cierto que Alicia no ve la modificación de Benito, pero si intenta modificar otra columna, se repetirá el mismo proceso de mezcla... siempre que Aliciapersevere en su propósito.

## Advertencias y alternativas

He dejado, con toda intención, un caso importante sin tratar: una modificación puede fallar también si el registro ha sido borrado. En ese caso, la relectura no devuelve el registro solicitado. Si en cada actualización, es decir, si en cada llamada a *Write* se envía un solo registro, este caso es fácil de detectar y de señalar con un error más acorde a su contexto. Si existe la posibilidad de enviar más de un registro, el algoritmo se complica un poco, y por esta razón no he querido contemplar esta posibilidad por separado. De todos modos, el algoritmo que he presentado produciría un segundo error, y no ocurriría nada más grave que la cara de desconcierto del usuario.

A parte de este problema, ¿hay algo que no le guste en mi propuesta? ¡Estupendo, cuénteme entonces la suya! En estos temas, no hay solución perfecta conocida. He intentado diseñar un algoritmo suficientemente razonable, pero nunca lloverá a gusto de todos. No pierda de vista que una de mis premisas más importantes ha sido lograr un algoritmo eficiente, teniendo en cuenta la posibilidad de división en capas físicas.

¿Quiere una alternativa sencilla? Algunos sistemas de acceso a datos, ya obsoletos por ciertos, forzaban una relectura de cada registro que se ponía en modo de edición. Esta técnica, aplicada a nuestra aplicación, nos forzaría a releer cada registro inmediatamente antes de mostrar el diálogo de edición. Si no se modifica el registro, esta relectura es superflua. En un sistema con un número mediano de usuarios, dicha política podría ser un derroche. En un sistema pequeño, funcionando sobre una red local, puede que el coste sea aceptable. Esta forma de trabajo no sirve para resolver conflictos, por supuesto, sino para minimizar su probabilidad de aparición.



# EJERCICIO 24

Objetivos

Impresión genérica

Técnicas introducidas

Clases para la impresión en .NET

**N**O QUIERO QUE SE ENGAÑE o confunda: cuando tenga que generar listados para impresión, ¡utilice alguna herramienta profesional y ahórrese sufrimientos! Visual Studio, a partir de la versión Professional, incluye Crystal Reports para estos menesteres. Incluso hay una nueva herramienta de informes desarrollada por Microsoft y que debe acompañar a las futuras versiones de SQL Server.

Para los propósitos de este curso, no obstante, he creído preferible mostrarle algunos de los entresijos de la impresión directa en .NET: no sólo puede ahorrarle dinero en circunstancias extremas, sino que también conviene que sepa cómo funciona el subsistema de impresión en la plataforma .NET.

## Dibujando sobre la impresora

Como sospechará, para imprimir sólo tenemos que dibujar sobre una página del mismo modo que dibujamos sobre la superficie de un control; eso sí, hay que tener en cuenta que el número de píxeles por página es siempre muy superior al número de píxeles de cualquier control, y que tenemos que contar con que el trabajo de impresión se dividirá entre varias páginas. En cualquier caso, el elemento común entre la impresión y el dibujo en pantalla es la siguiente clase:

```
System.Drawing.Graphics
```

Esta es la clase que representa una superficie de dibujo abstracta, sin importar su procedencia. Si asumimos que es sencillo dibujar con la ayuda de la clase *Graphics*, lo que más nos interesa, para empezar, es saber cómo podemos obtener una instancia de *Graphics* que represente a una página en la impresora. Y la respuesta está en la siguiente clase de componentes:

```
System.Drawing.Printing.PrintDocument
```

Esta clase ofrece un método público *Print*, que al ser ejecutado dispara una serie de eventos: *BeginPrint*, antes de comenzar la impresión, *PrintPage* para imprimir cada página, y *EndPrint* cuando ha terminado la operación. Tenemos que realizar todo el dibujo dentro del evento *PrintPage*, y para ello, el segundo parámetro del evento contiene, entre otras propiedades, una referencia a la superficie de dibujo, de clase *Graphics*, donde tenemos que dibujar la página. No hace falta conocer el total de páginas por adelantado: sólo tenemos que indicar, al terminar con una página, si quedan más páginas por imprimir.

Para imprimir un “documento” sencillo, tendríamos que desencadenar la operación con las siguientes instrucciones:

```
void Imprimir(string titulo)
{
    using (PrintDocument printDoc = new PrintDocument())
    {
        printDoc.PrintPage += new PrintPageEventHandler(ImprimirPagina);
        printDoc.DocumentName = titulo;
        printDoc.Print();
    }
}
```

La propiedad *DocumentName* es el título asociado al documento que se mostrará en el Administrador de Impresoras de Windows. Observe que sólo es obligatorio interceptar el evento *PrintPage*. Si sólo tenemos que imprimir una página con una elipse, nos bastaría lo siguiente:

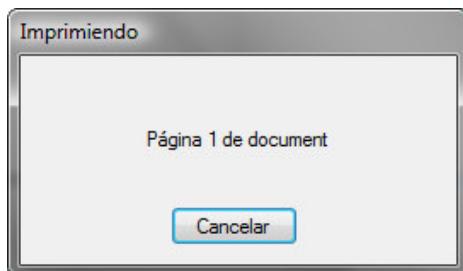
```
void ImprimirPagina(object sender, PrintPageEventArgs e)
{
    e.Graphics.DrawEllipse(
        System.Drawing.Pens.Black, e.MarginBounds);
}
```

Este método no modifica el valor inicial de la propiedad *HasMorePages* del parámetro *e* del evento. Como ese valor es **false**, el documento imprimirá una sola página. La página en sí consiste en una gigantesca elipse. Observe que utilizamos la propiedad *MarginBounds* del segundo parámetro del evento: esta propiedad, por omisión, hace referencia a un rectángulo contenido dentro del rectángulo mayor *PageBounds*, con una distancia de 100 unidades para cada margen. Estas “unidades” están calculadas de modo que el margen de 100 unidades sea equivalente a una pulgada, o a 25.4 milímetros, en el sistema métrico.

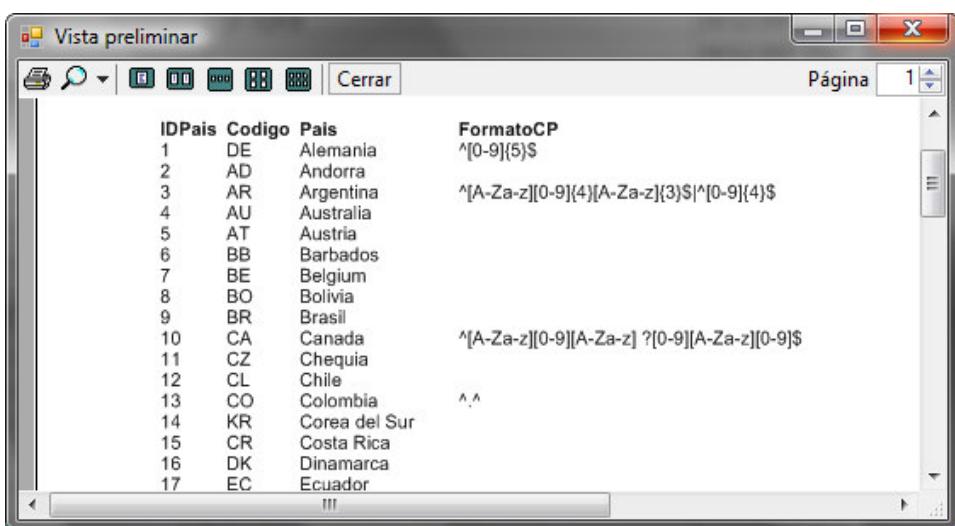
Además de los márgenes, las restantes características de impresión que ya esperamos como habituales, se encuentran en la propiedad *PrinterSettings* del documento de impresión; el valor de esta propiedad se recibe también en una propiedad del segundo parámetro del evento *PrintPage*. Podemos asignar valores para *PrinterSettings* en la respuesta al evento *QueryPageSettings* de la clase *PrintDocument*, pero es más frecuente realizar esta configuración con la ayuda de un componente llamado *PageSetupDialog*, que corresponde al diálogo común de Windows para la configuración de páginas.

### La vista preliminar

Si llamamos al método *Print* de un *PrintDocument* sin más, veremos aparecer un fugaz cuadro de diálogo como el siguiente:



En la mayoría de los casos bastará con este mecanismo para saber que la impresora está procesando nuestro documento... y en algunos otros, incluso sobrará. Pero es conveniente poder mostrar al usuario una vista preliminar del documento a imprimir. La plataforma .NET ya incluye un componente de serie, que se ocupa de esta tarea: *PrintPreviewDialog*. Este es el aspecto del diálogo que muestra en tiempo de ejecución, antes de imprimir realmente el documento:



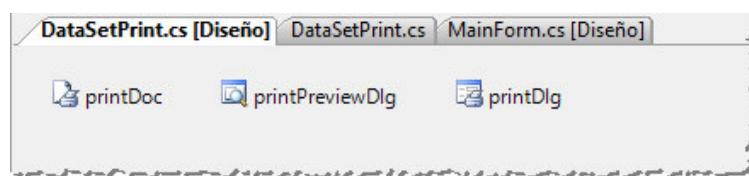
ID País	Código País	País	Formato CP
1	DE	Alemania	^[0-9]{5}\$
2	AD	Andorra	
3	AR	Argentina	^[A-Za-z][0-9]{4}[A-Za-z]{3}\$ ^0-9{4}\$
4	AU	Australia	
5	AT	Austria	
6	BB	Barbados	
7	BE	Belgium	
8	BO	Bolivia	
9	BR	Brasil	
10	CA	Canada	^[A-Za-z][0-9][A-Za-z] ?[0-9][A-Za-z][0-9]\$
11	CZ	Chequia	
12	CL	Chile	
13	CO	Colombia	^A
14	KR	Corea del Sur	
15	CR	Costa Rica	
16	DK	Dinamarca	
17	EC	Ecuador	

En verdad, no es para dar saltos de alegría. No hay un botón que nos permita hacer un zoom del 100% en único paso, sino que tenemos que desplegar la lista de acercamientos del botón correspondiente. Los controles utilizan el modelo *plano* de los controles nativos .NET, que es bastante aburrido. No hay una indicación del número total de páginas. Y por último, aunque no puede deducirse de la imagen, no tenemos la posibilidad de seleccionar otra impresora una vez que llegamos a este diálogo, porque el botón identificado por el icono de impresora envía directamente el trabajo a la impresora que hayamos seleccionado antes. Qué le vamos a hacer.

Lo importante, no obstante, es que podemos crear nuestro diálogo mejorado si fuese necesario. Eso es posible gracias a otro componente relacionado: *PrintPreviewControl*, que es el control principal del diálogo gestionado por *PrintPreviewDialog*. El control se enlaza a un componente *PrintDocument*, y su respuesta al mensaje de dibujo consiste en llamar a los métodos de impresión del documento.

### Imprimiendo un conjunto de datos

Nosotros, sin embargo, nos conformaremos de momento con utilizar la vista preliminar predefinida, para mostrar cómo suelen manejarse las tareas de impresión en .NET. Para crear un componente de impresión para el ejemplo, necesitaremos al menos tres componentes nativos de .NET: un *PrintDocument*, que implementará el algoritmo de impresión como respuesta a uno de sus eventos, un *PrintDialog*, para seleccionar y configurar la impresora, y un *PrintPreviewDialog* para que se encargue de la vista preliminar. Para simplificar la configuración de los tres componentes mencionados, nos conviene disponer de una superficie de diseño. Por lo tanto, vamos a añadir una clase de componente al proyecto, mediante el comando *Proyecto | Agregar componente* del menú de Visual Studio:



Observe que he abreviado un poco los nombres. La propiedad *Document* de *printPreviewDlg* y de *printDlg* deben apuntar al documento, *printDoc*. Necesitaremos también algunas variables para controlar el algoritmo de impresión, que declararemos dentro de una región, para una mejor organización del código fuente:

```
#region Variables necesarias para la impresión

private System.Data.DataTable dataTable = null;
private System.Drawing.Font printFont = null;
private System.Drawing.Font headerFont = null;
private int printRow = 0;
private float[] tabs = null;

#endregion
```

Como puede ver, vamos a exigir una tabla para imprimir su contenido. Con esto dejaremos fuera los listados maestro/detalles. Una alternativa a la tabla, sería disparar el algoritmo de impresión mediante una vista de datos o un origen de enlace, para aprovechar cualquier filtro temporal o cambio en el criterio de ordenación de las filas. Pero eso se lo dejaré como ejercicio.

Las variables *printFont* y *headerFont*, por su parte, harán referencia a tipos de letras temporales, que crearemos y destruiremos durante la impresión. Como imaginará, *printRow* será el índice del registro de la tabla que debemos imprimir, y en *tabs* calcularemos por adelantado el ancho de las columnas de datos en el informe.

Desencadenaremos la impresión llamando al siguiente método público, que definiremos dentro de nuestro componente:

```
public void Print(System.Data.DataTable dataTable)
{
    this.dataTable = dataTable;
```

```
    if (printDlg.ShowDialog() == System.Windows.Forms.DialogResult.OK)
        printPreviewDlg.ShowDialog();
}
```

Observe que, en vez de provocar la impresión llamando directamente al método *Print* de *PrintDocument*, delegamos el control en el cuadro de diálogo de previsualización. Ejecutamos su método *ShowDialog*, después de seleccionar una impresora mediante el componente *PrintDialog*, y dejamos que *PrintPreviewDialog* se ocupe del resto.

Los tipos de letras que utilizaremos los crearemos en la respuesta al evento *BeginPrint* del documento, y los “destruiremos” durante la respuesta a *EndPrint*. Es verdad que no existe destrucción determinista en .NET, pero recuerde que el método *Dispose* asume esta función en muchos componentes.

```
private void printDoc_BeginPrint(object sender,
    System.Drawing.Printing.PrintEventArgs e)
{
    printFont = new System.Drawing.Font("Arial", 8);
    headerFont = new System.Drawing.Font(printFont,
        System.Drawing.FontStyle.Bold);
    printRow = 0;
}

private void printDoc_EndPrint(object sender,
    System.Drawing.Printing.PrintEventArgs e)
{
    tabs = null;
    printFont.Dispose();
    printFont = null;
    headerFont.Dispose();
    headerFont = null;
}
```

Sin embargo, no podemos calcular los anchos de columnas en *BeginPrint*, porque para ello necesitaríamos la superficie de dibujo, y ésta no se suministra al evento mencionado. El método auxiliar que dedicaremos a estos menesteres se llamará *CalcularColumnas*, y se llamará antes de imprimir la primera página. Esta es su implementación:

```
private void CalcularColumnas(System.Drawing.Graphics g)
{
    tabs = new float[dataTable.Columns.Count];
    for (int i = 0; i < tabs.Length; i++)
    {
        System.Data.DataColumn col = dataTable.Columns[i];
        tabs[i] = g.MeasureString(col.Caption, headerFont).Width;
    }
    foreach (System.Data.DataRow row in dataTable.Rows)
        for (int i = 0; i < tabs.Length; i++)
        {
            float w = g.MeasureString(row[i].ToString(), printFont).Width;
            if (w > tabs[i])
                tabs[i] = w;
        }
}
```

Sopesamos dos variantes para el cálculo de los anchos de columnas:

- 1** Que los anchos fuesen proporcionales a los anchos de columnas en una rejilla tomada como patrón visual.
- 2** Por el contrario, podemos calcular el ancho máximo requerido para cada columna de acuerdo a su contenido real, e imprimir solamente aquellas columnas que quepan completamente dentro del ancho de la página soportada por la impresora. Como todos los registros se encuentran en memoria, en un conjunto de datos, hemos estimado que la operación nunca será excesivamente lenta.

Me he decidido por la segunda variante para evitar complicaciones con los detalles sucios de la implementación de *DataGridView*. Pero si decide implementar la primera variante por su cuenta, tenga presente que la proporcionalidad de las columnas sólo necesita ser aproximada. No hace falta que se enzarce con el número de píxeles por pulgada en el monitor ni nada semejante.

Observe que la operación más importante de *CalcularColumnas* es el cálculo del espacio necesario para imprimir una cadena de caracteres con determinado tipo de letra. Para ello usamos el método *MeasureString* de la clase *Graphics*, pasándole la cadena y el tipo de letra:

```
float w = g.MeasureString(row[i].ToString(), printFont).Width;
```

Primero obtenemos el ancho necesario para la cabecera de columna, que repetiremos en cada página, y luego recorremos todos los registros de la tabla. Para el cálculo de las cabeceras, por supuesto, utilizamos el tipo de letra en negritas que utilizaremos para su impresión.

El núcleo de la impresión se ejecuta dentro de la respuesta al evento *PrintPage* del documento. Es un método bastante largo, a pesar de la simplicidad de nuestros informes, pero no hace falta estudiar física nuclear para comprenderlo:

```
private void printDoc_PrintPage(object sender,
    System.Drawing.Printing.PrintPageEventArgs e)
{
    System.Drawing.Graphics g = e.Graphics;
    if (tabs == null)
        CalcularColumnas(g);
    float altura = printFont.GetHeight(g);
    float yPos = e.MarginBounds.Top;
    int lineas = (int)(e.MarginBounds.Height / altura);
    float xPos = e.MarginBounds.Left;
    for (int colNo = 0; colNo < dataTable.Columns.Count; colNo++)
    {
        if (xPos >= e.MarginBounds.Right) break;
        float right = xPos + tabs[colNo];
        if (right > e.MarginBounds.Right) break;
        g.DrawString(dataTable.Columns[colNo].Caption,
            headerFont, System.Drawing.Brushes.Black,
            System.Drawing.RectangleF.FromLTRB(
                xPos, yPos, right, yPos + altura));
        xPos = right + 2;
    }
    yPos += altura;
    lineas--;
    while (lineas > 0 &&
        printRow < dataTable.Rows.Count)
    {
        System.DataDataRow row = dataTable.Rows[printRow];
        int colNo = 0;
        xPos = e.MarginBounds.Left;
        while (colNo < dataTable.Columns.Count &&
            xPos < e.MarginBounds.Right)
        {
            float right = xPos + tabs[colNo];
            if (right > e.MarginBounds.Right) break;
            g.DrawString(
                row[colNo].ToString(),
                printFont, System.Drawing.Brushes.Black,
                RectangleF.FromLTRB(xPos, yPos, right, yPos + altura));
            xPos = right + 2;
            colNo++;
        }
        yPos += altura;
        printRow++;
        lineas--;
    }
    e.HasMorePages = printRow < dataTable.Rows.Count;
}
```

Hay un par de puntos a destacar:

- Al final del método, la asignación de la propiedad *HasMorePages* indica si quedan más páginas por imprimir o no.
- Para calcular la altura de las líneas, usamos el método *GetHeight* del objeto *Font*, en vez del método *MeasureString*, que es más exacto pero costoso, porque tiene en cuenta los caracteres reales para los que se debe calcular la altura. Para simplificar, hemos asumido que todas las líneas tendrán la misma altura. Observe que *GetHeight* incluye la separación entre líneas.
- La impresión de cada celda del informe se realiza mediante *DrawString*, un método con varias sobrecargas definido en la clase *Graphics*. Nuestra variante es la más sencilla, porque ni siquiera tiene en cuenta la posibilidad de alinear el texto de acuerdo a la columna, pero puede ensayar esta y otras posibilidades de *DrawString* por su cuenta.

### EJERCICIO PROUESTO

Este algoritmo de impresión exige ciertas mejoras a gritos. Para empezar, podríamos introducir variantes en la forma de resaltar las cabeceras de columnas. Podríamos evitar también la impresión de columnas vacías... o mejor aún, decidir las columnas a imprimir y sus anchos relativos de acuerdo a la configuración de la rejilla de datos correspondiente. También habría que utilizar la alineación de cada columna, y el formato de visualización. Por último, podríamos diseñar cabeceras y pies de página, sobre todo con el objetivo de imprimir números de páginas.

---

## La interfaz IPrintable

Al llegar a este punto, tenemos a nuestra disposición un componente *DataSetPrint* que puede imprimir una tabla de datos, aunque sin muchos floripondios: tonterías, sólo las precisas. Pero ahora tenemos que montar un mecanismo más o menos automático de impresión, que esté disponible al menos para todas las ventanas de navegación. Podríamos programar este algoritmo directamente dentro de la clase base de estas “ventanas” (controles de usuarios, en realidad). Pero la experiencia nos aconseja interponer al menos un tipo de interfaz, para abstraer un poco el diseño y evitar dependencias que más tarde podrían resultar onerosas.

Abra el fichero *LayoutManager.cs*, en el que hemos declarado las interfaces necesarias para el gestor de ventanas, para definir un nuevo tipo de interfaz:

```
public interface IPrintable
{
    bool CanPrint { get; }
    void Print();
}
```

Consideraremos que un objeto de ventana es *imprimible* si, en primer lugar, puede “imprimirse”; no está de más, por añadidura, permitir que el objeto advierta si está listo para ser impreso en un momento dado. En nuestra aplicación, los objetos imprimibles *siempre* estarán listos, pero esto ya es harina de otro costal.

Seguimos en el mismo fichero, porque necesitamos tapar ahora una carencia del diseño del gestor de ventanas. No tenemos una propiedad o función que nos indique cuál es la ventana activa en un momento dado. El algoritmo para determinar la ventana activa depende claramente del modelo de ventanas implementado, por lo que es buena idea que el gestor se ocupe de esta tarea. Añadiremos una propiedad *ActiveWindow*, de sólo lectura, a la interfaz *ILayoutManager*.

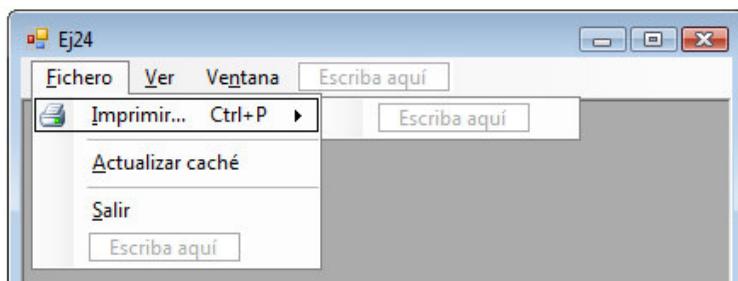
```
public interface ILayoutManager
{
    IWindow Create(System.Type type);
    IWindow Activate(System.Type type);
    DialogResult Execute(System.Type type, Form owner, bool newRow);
    IWindow ActiveWindow { get; }
    void DataModified(string eventname, object data);
    void ApplyNotifications();

    event System.EventHandler CreateWindow;
}
```

La implementación de la misma en la clase *MdiLayoutManager* es sencilla, sin embargo. Sólo tenemos que recordar que, en el caso en que existe una ventana MDI hija activa, el control de usuario que implementa la interfaz *IWindow* es el primer control que se añade a la ventana:

```
IWindow ILayoutManager.ActiveWindow
{
    get
    {
        Form f = mdiParent.ActiveMdiChild;
        if (f != null && f.Controls.Count > 0)
            return f.Controls[0] as IWindow;
        else
            return null;
    }
}
```

Añadiremos un comando al menú principal de la aplicación, con el propósito de imprimir el contenido de la ventana activa... siempre que ésta sea “imprimible”, claro:



Para activar o desactivar el comando de menú, interceptaremos el evento *DropDownOpening* del comando de menú que lo contiene en la barra de herramientas. Este comando es el que tiene como título *Fichero*, y ésta es la implementación de la respuesta al evento:

```
private void miFichero_DropDownOpening(object sender, EventArgs e)
{
    IPrintable printWindow = LayoutManager.ActiveWindow as IPrintable;
    miImprimir.Enabled = printWindow != null && printWindow.CanPrint;
}
```

En pocas palabras, obtenemos una referencia a la ventana activa e intentamos convertirla en una referencia a un objeto *IPrintable*. Si tenemos suerte, y el objeto está de buen humor para dejarse imprimir, activamos el comando *Imprimir*. Cuando seleccionamos el comando para ejecutar la operación correspondiente, volvemos a repetir la prueba:

```
private void miImprimir_Click(object sender, System.EventArgs e)
{
    IPrintable printWindow = LayoutManager.ActiveWindow as IPrintable;
    if (printWindow != null && printWindow.CanPrint)
        printWindow.Print();
}
```

Hemos repetido la verificación por si se nos ocurre asociar alguna tecla de atajo al comando de menú. Ciertas precauciones nunca están de más.

Para terminar, tenemos que hacer que las ventanas de navegación sean imprimibles. Tenemos la suerte de contar con una clase base común a todas ellas: *BaseWindow*. Añadiremos el tipo *IPrintable* a la lista de interfaces que implementa dicha clase:

```
public class BaseWindow : UserControl, IWindow, IDataWindow, IPrintable
{
    ;
}
```

E implementamos los dos métodos definidos en este tipo de interfaz:

```
#region Miembros de IPrintable

public bool CanPrint
{
    get { return true; }
}

public void Print()
{
    using (DataSetPrint pObj = new DataSetPrint())
        pObj.Print(((DataGridView)bindingSource.List).Table);
}

#endregion
```

Como anunciamos, la implementación de *CanPrint* es trivial. La implementación de *Print*, por su parte, crea un objeto *DataSetPrint* con alcance local, e imprime la tabla principal de la ventana de navegación. Si quisieramos un mecanismo más potente, que nos permitiese cambiar la implementación del informe por omisión de algunas ventanas, podríamos añadir la directiva **virtual** al método *Print*, para permitir su redefinición en algunas clases derivadas cuando fuese necesario.

# EJERCICIO 25

## Objetivos

Resumen

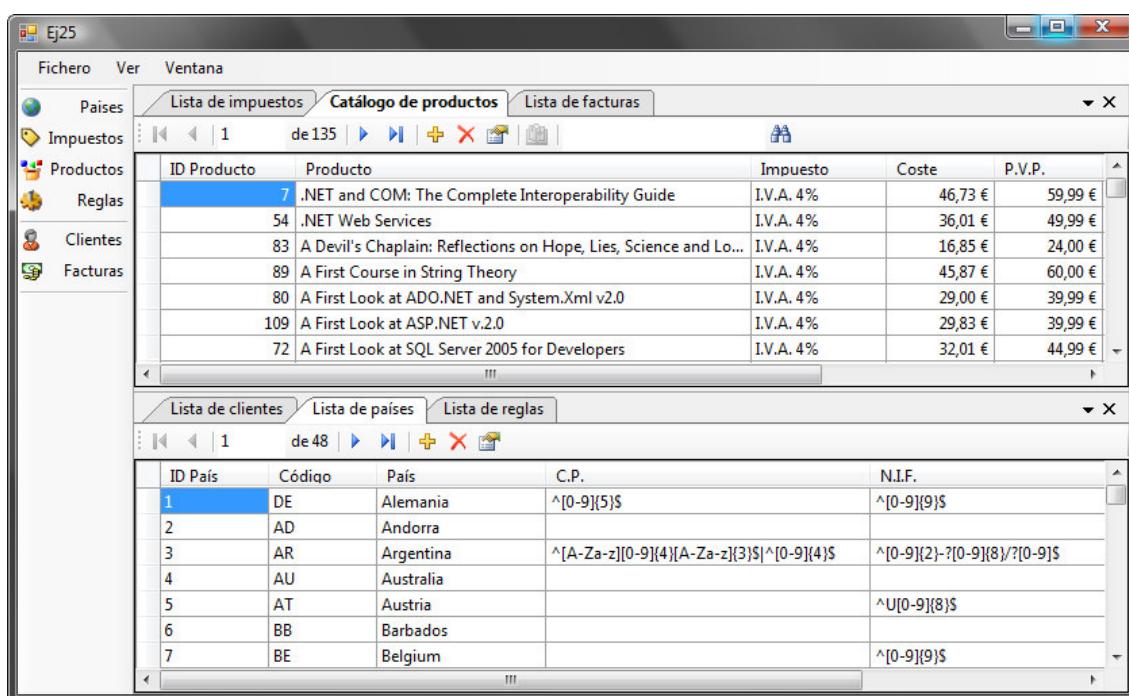
## Técnicas introducidas

Ventanas acopladas, imágenes en el menú

**H**A LLEGADO EL MOMENTO DE HACER cuentas. Hasta el momento, le he pedido confianza: confianza en que todo el esfuerzo que estábamos invirtiendo, definiendo gestores de ventanas, menús y cosas parecidas, no era un vano derroche. En teoría, la existencia del gestor de ventanas nos permitiría utilizar otros modelos de interacción sin necesidad de destrozar el código fuente. ¿Es cierto, o era sólo una falsa promesa?

## Ventanas acopladas

La respuesta está en la siguiente imagen:



He adaptado el código del gestor de ventanas para que utilice la colección de componentes de ventanas de Weifen Luo. Esta es una biblioteca de clases cuyo código fuente puede descargarse desde la siguiente página:

<http://sourceforge.net/projects/dockpanelsuite>

En el ejemplo del curso sólo he incluido la biblioteca compilada, como un ensamblado DLL, en el subdirectorio *Library* de la carpeta *Ej25*. El proyecto hace referencia al ensamblado binario, pero si lo desea, puede descargar el código fuente y añadirlo como proyecto secundario dentro de la solución correspondiente al ejercicio.

Detalles a tener en cuenta:

- He definido una nueva clase, *DockLayoutManager*, que sirve de alternativa a nuestra vieja conocida, *MdiLayoutManager*. No obstante, esta otra clase sigue existiendo en el código fuente.
- En el fichero *MainForm.cs*, en su primera línea, se define el símbolo *USE.Dock*. Cuando este símbolo está definido, el proyecto utiliza el nuevo gestor. Si comenta la primera línea, se utilizará el antiguo gestor. Esto se logra mediante directivas de compilación condicional.

- Si lo desea, puede experimentar con el código, para permitir ambos modos de ejecución, a petición del usuario de la aplicación.
- El principal problema tiene que ver con la inicialización de uno u otro modelo: es complicado pasar dinámicamente del uno al otro una vez que ya se han creado ventanas.
- De hecho, para la inicialización del modelo de ventanas acopladas he tenido que usar algún truco. Este modelo necesita que un componente *DockPanel* esté presente en el área interior de la ventana principal. Por desgracia, si se asigna la propiedad *Dock* del componente en el constructor de la ventana, se produce una excepción. He tenido que realizar esta asignación durante la respuesta al evento *Load* del formulario.

Tenga presente que los componentes de Weifen ofrecen muchas más posibilidades que las que he incluido en la demostración. Por ejemplo, es posible crear ventanas de herramientas, como el Inspector de propiedades o el Explorador de servidores de Visual Studio, que se acoplen en los bordes de la ventana principal, y que se escondan al no estar activos, según decida el usuario. También es posible, y sencillo, que la aplicación “recuerde” la disposición de ventanas de la última sesión, para restaurarla al iniciarse.

## Extensiones al gestor del menú

He introducido una pequeña mejora en el gestor de menú. Ahora podemos asociar una imagen a cada ventana de navegación, de manera que la imagen se muestra en el comando de menú:



La ventana de facturas, por ejemplo, está ahora decorada con el siguiente atributo:

```
[NonModal("Facturas", "Ver",
    Image = "Payment",
    Caption = "Lista de facturas", UniqueInstance = true,
    Editor = typeof(DlgFacturas))]
public partial class Facturas : Ej25.BaseWindow ...
```

En este caso, *Payment* debe ser el nombre de un recurso de imagen presente en el proyecto. El gestor utiliza reflexión para comprobar si existe una propiedad con el nombre dado en el gestor de recursos del proyecto.

La barra de herramientas de la izquierda, sin embargo, ha sido creada manualmente. Si lo desea, puede crear un segundo gestor de menú que se encargue solamente de poblar la barra. Luego, puede inicializar los dos gestores: no hay nada malo en ello. ¿Otras extensiones con las que experimentar? Puede añadir una propiedad al atributo *NonModal* que indique a qué grupo pertenece la ventana: los grupos de ventanas se mostrarían entonces por separado. Puede también introducir una prioridad relativa entre las ventanas, para determinar el orden en que serán creados los comandos o botones.



## CREACION DE LA BASE DE DATOS

---

```
/* ELIMINAR CUALQUIER VERSION ANTERIOR DE ESTA BASE DE DATOS */
use MASTER
go

if exists (select * from sysdatabases where name = 'AdoNet')
    drop database AdoNet
go

/* CREACION DE LA BASE DE DATOS */
/* ATENCION: Compruebe que existe el directorio
   donde se crearán los ficheros de la base de datos */

create database AdoNet
    on primary
        ( name = PrimaryData,
          filename = 'C:\SqlData\AdoNet.mdf',
          size = 2 )
    log on
        ( name = LogFile,
          filename = 'C:\SqlData\AdoNet.ldf' )
go

use AdoNet
go

/* TIPOS DE DATOS */

execute sp_addtype T_APELLIDOS,      'varchar(128)', 'not null'
execute sp_addtype T_BOOL,            'bit',           'not null'
execute sp_addtype T_CANT,            'integer',       'not null'
execute sp_addtype T_CIUDAD,          'varchar(40)',  'not null'
execute sp_addtype T_CODPAIS,         'char(2)',       'not null'
execute sp_addtype T_CODPOST,         'varchar(15)',  'not null'
execute sp_addtype T_DESCR,           'varchar(128)', 'not null'
execute sp_addtype T_DIRECCION,       'varchar(35)',  'not null'
execute sp_addtype T_FECHA,           'datetime',     'not null'
execute sp_addtype T_IDENT,            'integer',      'not null'
execute sp_addtype T_IMAGEN,           'image',         'null'
execute sp_addtype T_NIF,              'varchar(20)',  'not null'
execute sp_addtype T_NOMBRE,           'varchar(40)',  'not null'
execute sp_addtype T_PAIS,             'varchar(40)',  'not null'
execute sp_addtype T_PCT,              'numeric(5,2)', 'not null'
execute sp_addtype T_PRECIO,            'money',         'not null'
execute sp_addtype T_REGEX,            'varchar(255)', 'not null'
execute sp_addtype T_VALOR,            'varchar(128)', 'not null'
go

/* TABLAS: MODULO DE CLIENTES */

create table dbo.Paises (
    IDPais          T_IDENT      not null identity,
    Código          T_CODPAIS    not null,
    País            T_PAIS       not null,
    FormatoCP       T_REGEX      not null,
    FormatoNIF      T_REGEX      not null,

    Creacion        T_FECHA     not null default current_timestamp,
    TS              rowversion   not null,

    primary key      (IDPais),
    constraint UQ_PAISES_CODIGO
        unique        (Código),
    constraint UQ_PAISES_PAIS
        unique        (País)
)
go
```

```

create table dbo.Clientes (
    IDCiente      T_IDENT      not null identity,
    Nombre        T_NOMBRE     not null,
    Apellidos     T_APELLIDOS not null,
    NIF           T_NIF        not null,

    Direccion1   T_DIRECCION not null,
    Direccion2   T_DIRECCION null,
    Ciudad        T_CIUDAD     not null,
    CP            T_CODPOST   not null,
    IDPais        T_IDENT     not null,

    Especial      T_BOOL       not null default 0,
    TotalCompras T_PRECIO    not null default 0,

    Creacion      T_FECHA     not null default current_timestamp,
    TS            rowversion   not null,

    primary key  (IDCiente),
    constraint FK_CLIENTES_PAISES
        foreign key (IDPais) references dbo.Paises(IDPais)
)
go

create table dbo.Nombres (
    IDNombre      T_IDENT      not null identity,
    IDCiente      T_IDENT     not null,
    Nombre        T_NOMBRE     not null,

    Creacion      T_FECHA     not null default current_timestamp,
    TS            rowversion   not null,

    primary key  (IDNombre),
    constraint FK_NOMBRES_CLIENTES
        foreign key (IDCiente) references dbo.Clientes(IDCiente)
        on delete cascade
)
go

create table dbo.Atributos (
    IDAtributo    T_IDENT      not null identity,
    IDCiente      T_IDENT     not null,
    Atributo      T_NOMBRE     not null,
    Valor          T_VALOR     not null,

    Creacion      T_FECHA     not null default current_timestamp,
    TS            rowversion   not null,

    primary key  (IDAtributo),
    constraint FK_ATRIBUTOS_CLIENTES
        foreign key (IDCiente) references dbo.Clientes(IDCiente)
        on delete cascade
)
go

/* TABLAS: MODULO DE PRODUCTOS */

create table dbo.Impuestos (
    IDImpuesto    T_IDENT      not null identity,
    Impuesto      T_DESCR     not null,
    Tasa          T_PCT        not null,

    Creacion      T_FECHA     not null default current_timestamp,
    TS            rowversion   not null,

    primary key  (IDImpuesto),
    constraint UQ_IMPUESTOS
        unique          (Impuesto)
)
go

create table dbo.Productos (

```

```

IDProducto    T_IDENT      not null identity,
Producto       T_DESCR      not null,
IDImpuesto    T_IDENT      not null,
Coste          T_PRECIO     not null,
PVP            T_PRECIO     not null,
Existencias   T_CANT       not null default 0,
Minimo         T_CANT       not null default 0,
Imagen         T_IMAGEN    null,
Creacion       T_FECHA     not null default current_timestamp,
TS             rowversion   not null,
primary key  (IDProducto),
constraint FK_PRODUCTOS_IMPUESTOS
  foreign key (IDImpuesto) references dbo.Impuestos(IDImpuesto)
)
go

create table dbo.Palabras (
IDPalabra      T_IDENT      not null identity,
IDProducto     T_IDENT      not null,
Palabra         T_NOMBRE     not null,
Creacion       T_FECHA     not null default current_timestamp,
TS             rowversion   not null,
primary key  (IDPalabra),
constraint FK_PALABRAS_PRODUCTOS
  foreign key (IDProducto) references dbo.Productos(IDProducto)
on delete cascade
)
go

/* TABLAS: MODULO DE FACTURAS */

create table dbo.Facturas (
IDFactura      T_IDENT      not null identity,
Numero         T_IDENT      not null,
IDCliente      T_IDENT      null,
Direccion1    T_DIRECCION  not null,
Direccion2    T_DIRECCION  null,
Ciudad          T_CIUDAD    not null,
CP              T_CODPOST   not null,
IDPais          T_IDENT      not null,
BaseImponible  T_PRECIO     not null,
Impuestos       T_PRECIO     not null,
Importe         T_PRECIO     not null,
Creacion       T_FECHA     not null default current_timestamp,
TS             rowversion   not null,
primary key  (IDFactura),
constraint UQ_FACTURAS_NUMERO
  unique        (Numero),
constraint FK_FACTURAS_CLIENTES
  foreign key  (IDCliente) references dbo.Clientes(IDCliente),
constraint FK_FACTURAS_PAISES
  foreign key  (IDPais) references dbo.Paises(IDPais)
)
go

create table dbo.Lineas (
IDLinea         T_IDENT      not null identity,
IDFactura       T_IDENT      not null,
IDProducto      T_IDENT      not null,
Cantidad        T_CANT       not null,
Precio          T_PRECIO     not null,
Descuento       T_PRECIO     not null,
Impuesto        T_PRECIO     not null default 0,
Creacion       T_FECHA     not null default current_timestamp,

```

```

    TS          rowversion not null,
    primary key (IDLinea),
constraint FK_LINEAS_FACTURAS
    foreign key (IDFactura) references dbo.Facturas(IDFactura)
        on delete cascade,
constraint FK_LINEAS_PRODUCTOS
    foreign key (IDProducto) references dbo.Productos(IDProducto)
)
go

create table dbo.DatosFacturas (
    IDDatosFactura T_IDENT      not null identity,
    IDFactura       T_IDENT      not null,
    Atributo         T_NOMBRE     not null,
    Valor            T_VALOR     not null,
    Creacion        T_FECHA     not null default current_timestamp,
    TS              rowversion   not null,
    primary key (IDDatosFactura),
constraint FK_DATOS_FACTURAS
    foreign key (IDFactura) references dbo.Facturas(IDFactura)
        on delete cascade
)
go

create table dbo.Movimientos (
    IDMovimiento   T_IDENT      not null identity,
    IDProducto     T_IDENT      not null,
    Cantidad        T_CANT       not null,
    Precio          T_PRECIO     not null,
    IDFactura       T_IDENT      null,
    Creacion        T_FECHA     not null default current_timestamp,
    TS              rowversion   not null,
    primary key (IDMovimiento),
constraint FK_MOVIMIENTOS_PRODUCTOS
    foreign key (IDProducto) references dbo.Productos(IDProducto)
        on delete cascade,
constraint FK_MOVIMIENTOS_FACTURAS
    foreign key (IDFactura) references dbo.Facturas(IDFactura)
        on delete cascade
)
go

/* OTRAS TABLAS */

create table dbo.Exclusiones (
    IDExclusion    T_IDENT      not null identity,
    Palabra         T_NOMBRE     not null,
    Creacion        T_FECHA     not null default current_timestamp,
    TS              rowversion   not null,
    primary key (IDExclusion),
constraint UQ_EXCLUSIONES
    unique          (Palabra)
)
go

create table dbo.Reglas (
    IDRegla         T_IDENT      not null identity,
    Regla           T_DESCR      not null,
    Inicio          T_FECHA     not null,
    Fin             T_FECHA     null,
    Código          text         not null,
    Creacion        T_FECHA     not null default current_timestamp,
    TS              rowversion   not null,
)

```

```

        primary key      (IDRegla)
)
go

create table dbo.Contadores (
    IDContador      T_IDENT      not null,
    IDUltimo        T_IDENT      not null,

    primary key      (IDContador),
    constraint CK_CONTADORES
        check        (IDContador = 1)
)
go

insert into dbo.Contadores values(1, 1)
go

/* VISTAS Y FUNCIONES DE TABLAS */

create view dbo.XClientes as
    select c.*, a.Valor as Telefono
    from dbo.Clientes c left outer join dbo.Atributos a
        on c.IDCliente = a.IDCliente
    where a.Atributo = 'Teléfono'
go

create function dbo.Productos01 (
    @key01 varchar(30))
returns table as return (
select *
from dbo.Productos
where IDProducto in (
    select IDProducto
    from dbo.Palabras
    where Palabra like @key01 collate Modern_Spanish_CI_AI))
go

create function dbo.Productos02 (
    @key01 varchar(30),
    @key02 varchar(30))
returns table as return (
select *
from dbo.Productos
where IDProducto in (
    select IDProducto
    from dbo.Palabras
    where Palabra like @key01 collate Modern_Spanish_CI_AI) and
    IDProducto in (
        select IDProducto
        from dbo.Palabras
        where Palabra like @key02 collate Modern_Spanish_CI_AI))
go

create function dbo.Productos03 (
    @key01 varchar(30),
    @key02 varchar(30),
    @key03 varchar(30))
returns table as return (
select *
from dbo.Productos
where IDProducto in (
    select IDProducto
    from dbo.Palabras
    where Palabra like @key01 collate Modern_Spanish_CI_AI) and
    IDProducto in (
        select IDProducto
        from dbo.Palabras
        where Palabra like @key02 collate Modern_Spanish_CI_AI) and
    IDProducto in (
        select IDProducto
        from dbo.Palabras
        where Palabra like @key03 collate Modern_Spanish_CI_AI))
go

```

```

create function dbo.Coincidencias (
    @idP1 int, @idP2 int) returns int as
begin
    declare @rslt int
    select @rslt = count(distinct Palabra)
    from Palabras
    where Palabra in (
        select Palabra
        from Palabras
        where IDProducto = @idP1) and
        Palabra in (
            select Palabra
            from Palabras
            where IDProducto = @idP2)
    return @rslt
end
go

create function dbo.Recomendaciones
    (@idProducto int)
    returns table return (
        select top 100 IDProducto, Producto, IDImpuesto,
        Coste, PVP, Existencias, Minimo,
        Imagen, Creacion, TS
        from
        (select *, dbo.Coincidencias(IDProducto, @IDProducto) Coin
        from Productos
        where IDProducto <> @idProducto and
        IDProducto in (
            select IDProducto
            from Palabras
            where Palabra in (
                select Palabra
                from Palabras
                where IDProducto = @IDProducto))) as Tmp
        order by Coin desc)
go

/* PROCEDIMIENTOS */

create procedure ccsNormalizar
    @nstr varchar(255) output,
    @lstr varchar(255) output
    with encryption as
begin
    declare @p int
    while (@l=1)
    begin
        set @p = patindex('%[A-Za-z0-9]%', @nstr)
        if (@p = 0)
            return 0
        if (@p > 1)
            set @nstr = substring(@nstr, @p, 255)
        set @p = patindex('^[A-Za-z0-9]', @nstr)
        if (@p = 0)
            begin
                set @lstr = @nstr
                set @nstr = ''
            end
        else
            begin
                set @lstr = left(@nstr, @p - 1)
                set @nstr = right(@nstr, datalength(@nstr) - @p)
            end
        if not exists (select * from dbo.Exclusiones where Palabra = @lstr)
            return 1
    end
end
go

create procedure ccsExtraerNombres
    @idCliente T_IDENTIFIER
    with encryption as

```

```

begin
    declare @nstr varchar(255), @lstr varchar(255), @rslt integer
    select @nstr = Nombre + ' ' + Apellidos
    from dbo.Clientes
    where IDCliente = @idCliente
    execute @rslt = ccsNormalizar @nstr output, @lstr output
    while (@rslt = 1)
    begin
        insert dbo.Nombres(IDCliente, Nombre) values(@idCliente, @lstr)
        execute @rslt = ccsNormalizar @nstr output, @lstr output
    end
end
go

create procedure ccsExtraerPalabras
    @idProducto T_IDENT
    with encryption as
begin
    declare @nstr varchar(255), @lstr varchar(255), @rslt integer
    select @nstr = Producto
    from dbo.Productos
    where IDProducto = @idProducto
    execute @rslt = ccsNormalizar @nstr output, @lstr output
    while (@rslt = 1)
    begin
        insert dbo.Palabras(IDProducto, Palabra) values(@idProducto, @lstr)
        execute @rslt = ccsNormalizar @nstr output, @lstr output
    end
end
go

/* TRIGGERS */
create trigger ccsClientes_u on dbo.Clientes
    with encryption for update as
    if update(Nombre) or update(Apellidos)
begin
    set nocount on
    delete from dbo.Nombres
    where IDCliente in (select IDCliente from inserted)
    declare @id T_IDENT
    declare _clnt cursor local forward_only static read_only for
        select IDCliente from inserted
    open _clnt
    fetch _clnt into @id
    while (@@fetch_status = 0)
    begin
        execute ccsExtraerNombres @id
        fetch _clnt into @id
    end
    close _clnt
    deallocate _clnt
end
go

create trigger ccsClientes_i on dbo.Clientes
    with encryption for insert as
begin
    set nocount on
    declare @id T_IDENT
    declare _clnt cursor local forward_only static read_only for
        select IDCliente from inserted
    open _clnt
    fetch _clnt into @id
    while (@@fetch_status = 0)
    begin
        execute ccsExtraerNombres @id
        fetch _clnt into @id
    end

```

```

    end
    close _clnt
    deallocate _clnt
end
go

create trigger ccsProductos_u on dbo.Productos
    with encryption for update as
        if update(Producto)
begin
    set nocount on
    delete from dbo.Palabras
    where IDProducto in (select IDProducto from inserted)
    declare @id T_IDENT
    declare _clnt cursor local forward_only static read_only for
        select IDProducto from inserted
    open _clnt
    fetch _clnt into @id
    while (@@fetch_status = 0)
begin
    execute ccsExtraerPalabras @id
    fetch _clnt into @id
end
    close _clnt
    deallocate _clnt
end
go

create trigger ccsProductos_i on dbo.Productos
    with encryption for insert as
begin
    set nocount on
    declare @id T_IDENT
    declare _clnt cursor local forward_only static read_only for
        select IDProducto from inserted
    open _clnt
    fetch _clnt into @id
    while (@@fetch_status = 0)
begin
    execute ccsExtraerPalabras @id
    fetch _clnt into @id
end
    close _clnt
    deallocate _clnt
end
go

create trigger xClientes_u on dbo.XClientes
    instead of update as
begin
    set nocount on
    update dbo.Clientes
    set Nombre = i.Nombre,
        Apellidos = i.Apellidos,
        NIF = i.NIF,
        Direccion1 = i.Direccion1,
        Direccion2 = i.Direccion2,
        Ciudad = i.Ciudad,
        CP = i.CP,
        IDPais = i.IDPais,
        Especial = i.Especial
    from dbo.Clientes c, inserted i
    where c.IDCliente = i.IDCliente
    if update(Teléfono)
begin
    delete dbo.Atributos
    from dbo.Atributos a, deleted d
    where a.IDCliente = d.IDCliente and
        a.Atributo = 'Teléfono'

```

```

    insert into dbo.Atributos(IDCliente, Atributo, valor)
    select IDCliente, 'Teléfono', Telefono
    from inserted
end
go

create trigger ccsMovimientos_i on dbo.Movimientos
    with encryption for insert as
begin
    set nocount on
    update Productos
    set Existencias = Existencias + Cantidad
    from inserted
    where Productos.IDProducto = inserted.IDProducto
end
go

create trigger ccsLineas_i on dbo.Lineas
    with encryption for insert as
begin
    set nocount on
    update Lineas
    set Impuesto = round(
        imp.Tasa * (ins.Precio-ins.Descuento) * ins.Cantidad / 100.0, 2)
    from dbo.Productos pro, dbo.Impuestos imp, inserted ins
    where ins.IDLinea = Lineas.IDLinea and
        ins.IDProducto = pro.IDProducto and
        pro.IDImpuesto = imp.IDImpuesto

    update Facturas
    set BaseImponible =
        (select sum((Precio - Descuento) * Cantidad)
        from Lineas lin
        where lin.IDFactura = Facturas.IDFactura),
        Impuestos =
        (select sum(Impuesto)
        from Lineas lin
        where lin.IDFactura = Facturas.IDFactura),
        Importe =
        (select sum((Precio - Descuento) * Cantidad + Impuesto)
        from Lineas lin
        where lin.IDFactura = Facturas.IDFactura)
    from inserted ins
    where ins.IDFactura = Facturas.IDFactura
    insert into Movimientos(IDProducto, IDFactura, Cantidad, Precio)
    select IDProducto, IDFactura, - Cantidad, Precio - Impuesto
    from inserted
end
go

create trigger ccsLineas_u on dbo.Lineas
    with encryption for update as
begin
    set nocount on
    update Lineas
    set Impuesto = round(
        imp.Tasa * (ins.Precio-ins.Descuento) * ins.Cantidad / 100.0, 2)
    from dbo.Productos pro, dbo.Impuestos imp, inserted ins
    where ins.IDLinea = Lineas.IDLinea and
        ins.IDProducto = pro.IDProducto and
        pro.IDImpuesto = imp.IDImpuesto

    update Facturas
    set BaseImponible =
        (select sum((Precio - Descuento) * Cantidad)
        from Lineas lin
        where lin.IDFactura = Facturas.IDFactura),
        Impuestos =
        (select sum(Impuesto)
        from Lineas lin
        where lin.IDFactura = Facturas.IDFactura),
        Importe =
        (select sum((Precio - Descuento) * Cantidad + Impuesto)
        from Lineas lin
        where lin.IDFactura = Facturas.IDFactura),

```

```

        select sum((Precio - Descuento) * Cantidad + Impuesto)
        from Lineas lin
        where lin.IDFactura = Facturas.IDFactura)
from inserted ins
where ins.IDFactura = Facturas.IDFactura
insert into Movimientos(IDProducto, IDFactura, Cantidad, Precio)
select IDProducto, IDFactura, Cantidad, Precio - Impuesto
from deleted
insert into Movimientos(IDProducto, IDFactura, Cantidad, Precio)
select IDProducto, IDFactura, - Cantidad, Precio - Impuesto
from inserted
end
go

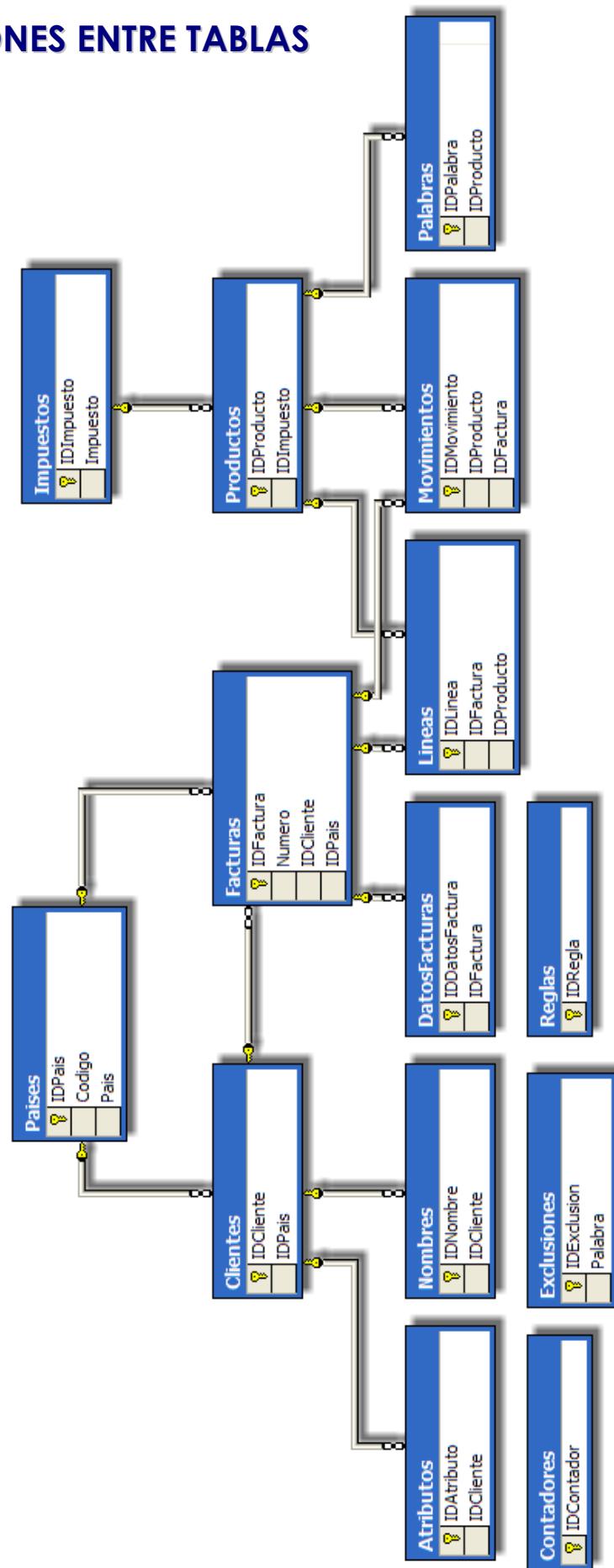
create trigger ccsLineas_d on dbo.Lineas
with encryption for delete as
begin
    set nocount on
    update Facturas
    set BaseImponible =
        select sum((Precio - Descuento) * Cantidad)
        from Lineas lin
        where lin.IDFactura = Facturas.IDFactura),
    Impuestos =
        select sum(Impuesto)
        from Lineas lin
        where lin.IDFactura = Facturas.IDFactura),
    Importe =
        select sum((Precio - Descuento) * Cantidad + Impuesto)
        from Lineas lin
        where lin.IDFactura = Facturas.IDFactura)
    from deleted del
    where del.IDFactura = Facturas.IDFactura
    insert into Movimientos(IDProducto, IDFactura, Cantidad, Precio)
    select IDProducto, IDFactura, Cantidad, Precio - Impuesto
    from deleted
end
go

create trigger ccsFacturas_i on dbo.Facturas
with encryption for insert as
begin
    set nocount on
    declare @id T_IDENT, @num T_IDENT
    declare _fact cursor local forward_only static read_only for
        select IDFactura from inserted
    open _fact
    fetch _fact into @id
    while (@@fetch_status = 0)
    begin
        update dbo.Contadores
        set @num = IDUltimo,
            IDUltimo = IDUltimo + 1
        update dbo.Facturas
        set Numero = @num
        where IDFactura = @id
        fetch _fact into @id
    end
    close _fact
    deallocate _fact
end
go

/* FIN DE FICHERO */

```

## RELACIONES ENTRE TABLAS





# INDICE ALFABETICO

## A

abstract · 15  
AcceptChanges · 42, 66, 110  
ActiveMdiChild · 168  
AddNew · 70  
AllowDBNull · 50  
Application  
    Idle · 43  
    StartupPath · 80  
    ThreadException · 153  
Assembly · 16, 35  
atributos · 14  
AttributeUsage · 14

## B

BindingSource · 51, 63, 68  
    AddNew · 70  
    Current · 151  
    CurrentItemChanged · 122  
    DataSource · 66  
    EndEdit · 67  
    List · 66  
    RemoveCurrent · 71  
    ResetCurrentItem · 124, 131  
bit · 7, 11

## C

CellContentClick · 131  
CellDoubleClick · 68

## Ch

checksum · 84  
checksum\_agg · 84

## C

clases  
    abstractas · 15  
    anidadas · 102  
    de atributos · 14  
    indizadores · 17  
Closing · 36, 116, 118  
collate · 86, 103  
ComboBox  
    DataSource · 4  
    ValueMember · 4  
Control  
    Dock · 36  
    Validated · 77  
    Validating · 76  
controles de usuarios · 31  
CreateChildView · 110  
CurrentItemChanged · 122

## D

DataGridView · 51, 166  
    CellContentClick · 131  
    CellDoubleClick · 68  
DataRow  
    ItemArray · 66  
DataRowVersion · 159  
DataRowView · 61, 147  
    CreateChildView · 110  
DataSet  
    AcceptChanges · 42, 66, 110  
    ExtendedProperties · 83, 158  
    HasChanges · 41, 67  
    Merge · 28, 67, 160  
    RejectChanges · 42  
    Tables · 108  
DataSource  
    ComboBox · 4  
DataTable  
    ImportRow · 110  
    Select · 43  
DataView  
    Sort · 110  
DbConcurrencyException · 157  
Dock · 36  
Document  
    PrintDialog · 164  
    PrintPreviewDialog · 164  
DrawString · 167  
DropDownOpening · 168

## E

encuentro  
    exterior · 91  
EndEdit · 67  
ErrorProvider · 73, 76, 117, 132  
    enlace a datos · 117  
esquemas abiertos · 9, 90  
eventos  
    multidifusión · 37  
ExecuteScalar · 82  
expresiones regulares · 76, 88, 117  
ExtendedProperties · 83, 158

## F

Find · 79  
    DataRowCollection · 79  
foreach · 88, 101  
Form  
    ActiveMdiChild · 168  
    Closing · 36, 116, 118  
    FormBorderStyle · 31  
    IsMdiContainer · 23  
    MdiChildren · 21, 35  
    MdiParent · 22, 36  
    OwnedForms · 25  
    WindowState · 21

FormBorderStyle · 31

---

**G**

GetCustomAttributes · 22, 35, 69  
GetExecutingAssembly · 16  
Graphics · 162  
    DrawString · 167  
    MeasureString · 166  
Guid · 75

---

**H**

HasChanges · 41, 67

---

**I**

identity · 3, 42, 49  
IDisposable · 112  
Idle · 43  
ImportRow · 110, 151  
instead of · 91  
InvokeMember · 21, 22, 36, 64  
IsDefined · 16  
IsMdiContainer · 23  
IsSubclassOf · 16, 35  
IsWhiteSpace · 102  
ItemArray · 66  
iteradores · 101  
IVsaEngine · 139

---

**J**

JScript · 11

---

**L**

like · 8, 86, 103

---

**M**

MdiChildren · 21, 35  
MdiParent · 22, 36  
MdiWindowListItem · 23  
MeasureString · 166  
MenuStrip · 15, 18  
    MdiWindowListItem · 23  
Merge · 28, 67, 160  
multidifusión · 37

---

**O**

object · 17  
osql · 3  
OwnedForms · 25

---

**P**

PageSetupDialog · 163  
Path · 80  
Print · 165  
PrintDialog · 164  
PrintDocument · 162  
    Print · 165  
    PrintPage · 162  
    QueryPageSettings · 163  
PrintPage · 162  
PrintPreviewDialog · 163  
    Document · 164

---

**Q**

QueryPageSettings · 163

---

**R**

reflexión · 13, 16  
RegEx · 76, 88  
    Split · 88  
regiones · 23  
RejectChanges · 42  
RemoveCurrent · 71  
Replace · 18  
ResetCurrentItem · 124, 131  
RowUpdated · 135

---

**S**

scope\_identity · 49  
sealed · 55  
Select · 43  
ShowDialog · 165  
ShowModal · 64  
Sort · 110  
Split · 18, 57, 88  
SplitContainer · 93  
SqlCommand  
    ExecuteScalar · 82  
    Transaction · 112  
    UpdateRowSource · 48  
SqlConnection · 46  
SqlDataAdapter · 47, 73  
    RowUpdated · 135  
    TableMappings · 58, 88, 97  
    Update · 79  
StartupPath · 80  
static · 30  
StringBuilder · 55, 103

---

**T**

tablas derivadas · 95  
TableMappings · 58, 88, 97  
text · 11, 119  
TextBox  
    Validated · 77  
ThreadException · 153  
ToLower · 18  
ToolStripContainer · 52, 93, 124

top · 60  
Transact SQL  
    atributo identity · 3  
    bit · 7, 11  
    checksum · 84  
    cláusula top · 60  
    funciones agregadas · 84  
    left outer join · 91  
    like · 8, 86, 103  
    scripts · 1  
    subconsultas · 94  
    tablas derivadas · 95  
    text · 11, 119  
    uniqueidentifier · 5, 75  
Transaction · 112  
triggers · 4, 10  
    for insert · 128, 129  
    for update · 6  
    instead of · 91  
    scope\_identity · 49  
Type  
    GetCustomAttributes · 22, 35, 69  
    InvokeMember · 21, 22, 36, 64  
    IsDefined · 16  
    IsSubclassOf · 16, 35

---

## U

uniqueidentifier · 5, 75  
Update · 79  
UpdateRowSource · 48  
UserControl · 31

---

## V

Validated · 77  
Validating · 76  
ValueMember · 4  
versiones de filas · 79

---

## X

XSD  
    edición · 58, 92

*Programación con ADO.NET en C#*

*Copyright © 2004-2008, by Intuitive Sight*

*Prohibida la reproducción total o parcial de esta obra,  
por cualquier medio, sin autorización escrita del autor.*

*Madrid, España, 2008*