

intsight

Boletines Técnicos

Intuitive Sight

Los productos y marcas registradas mencionados son propiedad de sus respectivos propietarios. Igualmente, todo el mundo es hijo de su madre y de su padre, mientras no se demuestre lo contrario.

El autor de estos artículos es buena persona y es completamente incapaz de romper un plato o matar una mosca, y es inocente de cualquier cosa de la que pueda ser acusado, y ruega a Dios que lo mantenga alejado de las zarpas de los abogados.

Las opiniones expresadas aquí son simplemente opiniones, y no hace falta recurrir a la tortura para que su autor niegue tajantemente estar de acuerdo con ellas.

Queda terminante y fehacientemente prohibida la reproducción de este documento para fines no autorizados explícitamente por el presunto autor del documento.

Copyright © 2003, Ian Marteens & Intuitive Sight

IMPRESIONES SOBRE DELPHI 7

En las últimas semanas he tenido oportunidad de evaluar el funcionamiento de la versión 7 de Delphi en lo que atañe al acceso a bases de datos como SQL Server y Oracle. Resultado: he encontrado problemas graves en algunas de las nuevas interfaces de acceso a datos, la mayoría de ellos reconocidos por la propia Borland. Estos son los hechos:

- 1 El controlador de DB Express para SQL Server (me refiero al parche que Borland ya ha lanzado) tiene problemas con las relaciones maestro/detalles basadas en claves enteras y no nulas. Según Ramesh Theivendran, de Borland, se trata de un *bug* de OLE DB que están intentando rodear. Según la opinión de un humilde servidor, no debe ser un *bug* tan horrible si ADO y otras tantas interfaces de acceso a SQL Server han logrado evitarlo hasta el momento. Como se trata de un error en una funcionalidad tan importante, el controlador de DB Express para SQL Server (una de las pocas novedades de Delphi 7) es completamente inútil de momento.
- 2 DB Express para Delphi 6 tenía problemas para acceder a Oracle 9i, y muchos esperaban la aparición de Delphi 7 para resolver el problema. Por desgracia, el nuevo controlador tiene problemas con las actualizaciones generadas por DataSnap cuando el servidor de Oracle 9i... se ejecuta en un sistema operativo en castellano. Al parecer, se trata de un *bug* de novatos: las constantes reales utilizan la coma por el punto. Causa probable: que a nadie se le haya ocurrido hacer la prueba con un servidor que no estuviese en inglés. ¿Y qué pasa entonces con Oracle 8i? ¿Sigue funcionando con DB Express? Resulta que Borland sólo certifica el "correcto" funcionamiento de su controlador con Oracle 9i. Como si actualizar la versión de Oracle en una empresa real fuese tan simple como ir de compras al supermercado. Tal vez en Enron las cosas funcionasen precisamente así...
- 3 Delphi 6 introdujo un nuevo sistema para diseñar interfaces visuales, basado en el componente *Action Manager*. Lamen-

tablemente, el sistema no funcionaba correctamente para aplicaciones MDI. Cuando se maximiza una ventana hija en este tipo de aplicaciones, por ejemplo, los iconos de la barra de título de la ventana hija deben moverse al extremo derecho de la barra de menú de la ventana principal. El *Action Manager* de Delphi 6 no soportaba esta característica, y hacía imposible desarrollar aplicaciones MDI: una vez que se maximiza la ventana hija, perdemos la posibilidad de restaurar su tamaño. En teoría, Delphi 7 debía resolver ese problema. Inexplicablemente, los cambios necesarios en el componente de menú no fueron incluidos en el CDROM, al parecer, ipor olvido o descuido de alguien! Dicho sea de paso, esto es un claro síntoma de que en Borland no se programan aplicaciones MDI de ningún tipo.

- 4 Hablando de chapuzas: Delphi 7 incluye nuevas acciones predefinidas que actúan sobre conjuntos de datos clientes. Es una idea que estaba en el aire desde hace tiempo; yo mismo hice la prueba en Delphi 6, con éxito. Pero encontré enseguida un problema: si registraba las nuevas acciones dentro de una categoría de acciones ya existente (*DataSet*, en este caso), se perdían las imágenes registradas para las acciones anteriores dentro de la categoría. Al parecer, nadie se dio cuenta en Borland de que esto sucedía... y como resultado, las acciones predefinidas de la categoría *DataSet* no se configuran con sus imágenes correspondientes.
- 5 Y hay una larga lista de quejas "menores": se introdujeron varios errores en la identificación de usuarios en WebSnap, siguen habiendo problemas con los módulos de datos remotos basados en SOAP, incluso he encontrado nuevos problemas al ejecutar determinados métodos remotos vía SOAP...

Mi consejo: manténgase alejado de Delphi 7, al menos mientras Borland no haga ejercicio de humildad y saque de una maldita vez el parche que todos estamos esperando desde hace ya dos meses.

UN ENLACE DE DATOS UNIVERSAL

¿Cómo se puede mostrar el valor de una columna de un conjunto de datos en una barra de estado? Sí, es verdad que podemos interceptar el evento `OnDataChange` de un `TDataSource`, pero ¿para qué escribir código cuando un componente puede hacerlo por nosotros? Además, con un componente podemos tener en tiempo de diseño una idea del aspecto final en tiempo de ejecución...

El componente `TimDBLink`, incluido en este boletín, tiene ese propósito. Por una parte, `TimDBLink` tiene el aspecto de un control de datos orientado a campos: hay propiedades `DataSource` y `DataField`. Pero `TimDBLink` no es un control, sino que desciende directamente de `TComponent`, aunque tiene una propiedad `Control`, de tipo `TControl`. Para indicar cuál propiedad del control asociado debe ser modificada, tiene también una propiedad de tipo **string**, llamada precisamente `Property`.

En esta propiedad podemos teclear un *object path*, o ruta de objetos. El caso más simple es un nombre de propiedad perteneciente al control. Por ejemplo, si el control es un `TDBEdit`, podemos usar `Hint` como valor para la propiedad `Property`. También podemos utilizar un punto para escoger una propiedad dentro de una referencia a un objeto, como en este ejemplo:

```
Font.Name
```

También podemos entrar dentro de los elementos de una colección. Supongamos que el control es una barra de estado dividida

en tres paneles. Entonces podemos cambiar el texto del tercer panel con esta ruta:

```
Panels@2.Text
```

Por supuesto, el símbolo `@` no se interpreta como en Delphi. La expresión anterior se debe interpretar así:

```
Panels[2].Text
```

Para el componente es más sencillo utilizar la sintaxis basada en `@` que no tener que analizar los corchetes. Además, esta forma de indexar un vector no es un invento mío: al menos, existe en Eiffel.

Por último, se permite utilizar la pseudo propiedad `Text` con una propiedad de tipo `TStrings`. Es cierto que esta clase tiene "realmente" una propiedad llamada `Text...` pero no es una propiedad **published**, por lo que no existe información en tiempo de ejecución (RTTI) sobre la misma. Sin embargo, el intérprete de rutas de objetos la admite por definición. De esta manera podemos utilizar la siguiente expresión para controlar el contenido de un campo memo:

```
Lines.Text
```

Internamente, `TimDBLink` utiliza un componente `TFieldDataLink` para actualizar el control asociado. El enlace de datos se inicializa en el constructor:

```

constructor TimDBLink.Create(
  AOwner: TComponent);
begin
  inherited Create(AOwner);
  FDataLink := TFieldDataLink.Create;
  FDataLink.OnDataChange := DataChange;
end;

```

En la última instrucción del constructor, se asigna la dirección de un método interno de TimDBLink a un evento disparado por FDataLink. La implementación del método es la siguiente:

```

procedure TimDBLink.DataChange(
  Sender: TObject);
var
  F: TField;
  S: string;
begin
  if not Assigned(FDataLink) then Exit;
  F := FDataLink.Field;
  if Assigned(F) then
    if F is TAggregateField then
      S := AggrDisplayText(TAggregateField(F))
    else
      S := F.DisplayText

```

```

else if (csDesigning in ComponentState)
  and FDesignTime and (DataField <> '') then
  S := '[' + DataField + ']'
else
  Exit;
if (FDataLink.Control <> nil) and
  (FPropertyName <> '') then
  SetStringValue(FDataLink.Control,
    FPropertyName, S);
if Assigned(FOnDataChange) then
  FOnDataChange(Self, S);
end;

```

Hay dos funciones resaltadas en el código anterior. La primera de ellas, AggrDisplayText, es una solución artesanal a una chapuza délfica: los campos agregados ignoran por lo general el valor de su propiedad DisplayFormat, que determina el resultado de la propiedad DisplayText del propio campo. Esto ha sido resuelto parcialmente en Delphi 7: los campos de tipo fecha, hora y los de tipo real funcionan ahora... pero han olvidado corregir el problema cuando el campo es de tipo entero. Por último, SetStringValue, cuyo código fuente completo encontrará en el fichero que acompaña al boletín, es quien implementa la interpretación de las rutas de objetos, haciendo uso de la RTTI ofrecida por el control.

LIMITAR EL NÚMERO DE FILAS EN ORACLE

¿Cómo se puede limitar el número máximo de filas devueltas por una consulta en Oracle? En La Cara Oculta de Delphi 6, y también en un truco de mi página, menciono la existencia de un pseudo atributo llamado **rownum**, que puede utilizarse con ese propósito. Por ejemplo:

```

select *
from CLIENTES
where rownum <= 10

```

Este atributo devuelve siempre el número ordinal de la fila que se está procesando en cada momento. Esto quiere decir que la siguiente consulta no devolverá nunca fila alguna, independientemente del número de filas que tenga la tabla base:

```

select *
from CLIENTES
where rownum > 10

```

Por desgracia, cuando se trata de consultas que ordenan sus filas, **rownum** no puede utilizarse con sentido, al menos de forma directa. Considere la siguiente instrucción:

```

select *
from CLIENTES
where rownum <= 10
order by Apellidos

```

El problema consiste en que la ordenación es la última operación que se ejecuta al evaluar la consulta. Consecuentemente, la

instrucción escoge en primer lugar diez registros de clientes, según un algoritmo no determinista, y luego ordena esos diez registros. Como comprenderá, el resultado no tiene utilidad en la práctica...

Hojeando los manuales de Oracle 9i he encontrado una solución a esta dificultad. La instrucción anterior debería formularse del siguiente modo:

```

select *
from (
  select *
  from CLIENTES
  order by Apellidos)
where rownum <= 10

```

Observe que hay dos consultas en el ejemplo: la primera de ellas, la que ordena los registros por apellidos, se ha incluido dentro de la cláusula **from** de la otra, como si fuese una tabla física. Por lo tanto, primero se realiza la ordenación, y luego se devuelven los diez primeros registros de la tabla ordenada.

Debo advertirle que sólo he verificado este truco con Oracle 9i, y no sé todavía si funcionará en las versiones anteriores. Las consultas anidadas en la cláusula **from** también están disponibles en SQL Server, aunque no son necesarias para limitar el número de filas evaluadas en este sistema.

INTERBASE 7

Casi al cierre de este boletín, Borland ha anunciado la disponibilidad de su nueva versión de InterBase. Evidentemente, todavía no tenemos una copia en mano, por lo que me limitaré a comentar las novedades enumeradas en el comunicado.

Al parecer, hay pocas características nuevas, pero esto no es necesariamente algo malo. Por el contrario, si realmente se ha logrado depurar el código fuente y hacer más estable el motor relacional, la ausencia de sorpresas sería verdaderamente aceptable. Y parece ser que los principales cambios se han centrado en el área del manejo de concurrencia y transacciones. InterBase 7 es ahora capaz de aprovechar a fondo los servidores con multiprocesamiento simétrico (SMP). En las versiones anteriores, cuando el servidor de InterBase detectaba la presencia de más de un procesador, forzaba su ejecución en uno sólo de los procesadores (*CPU affinity*). Con el hardware correspondiente abaratarán

dose día a día, tener un servidor SMP ya no es un lujo al alcance de pocos.

Otra mejora importante es la adición de tablas de sistema especiales. Se trata de tablas temporales, que sólo están disponibles inicialmente para el administrador del servidor y para los propietarios de bases de datos. En estas tablas se puede encontrar información relacionada con el funcionamiento del servidor: las conexiones activas, las transacciones en ejecución, las instrucciones y procedimientos SQL ejecutados, el uso de la memoria... Incluso es posible, en algunos casos, influir sobre el comportamiento de InterBase. Por ejemplo, el administrador puede detener una operación en curso, o desconectar a determinado cliente.

También es destacable la adición de un tipo **boolean** al repertorio existente. La documentación afirma que el nuevo tipo ocupa 32

bits... pero no sé si creerlo o pensar que se trata de un error en la documentación. Ya veremos...

Finalmente, hay algunos cambios de menor trascendencia. Se han introducido nuevas funciones en el API para detectar la versión del cliente de InterBase, hay nuevas reglas para el manejo de las tablas externas, y los nombres de metadatos pueden llegar hasta 67 caracteres. Esto último implica la presencia de nuevas funciones del API para el trabajo con blobs y matrices, aunque de momento DB Express e InterBase Express no las soportan. Hay un

nuevo controlador JDBC, para los que gusten de ese tipo de cosas. Y al parecer, en determinadas herramientas ya no será necesario utilizar el engorroso comando **set term** para compilar un procedimiento almacenado o un *trigger*.

Por supuesto, en cuanto esté disponible el producto, anunciaremos nuestras ofertas para el Server Media Kit (que contiene el software y la documentación impresa a muy bajo precio) y para los paquetes de licencias de clientes y servidores.

WEBSNAP Y ADO

Esta es una alerta de error reconocida por Borland, pero que puede tomar por sorpresa a alguien. Y de todos modos, aquí explico un poco la solución al problema:

Se trata de un error que ocurre cuando se ejecuta una aplicación WebSnap con Delphi 7, que utilice ADO o ADO Express como interfaz de acceso a datos, especialmente cuando se utiliza el Web App Debugger. Cualquier intento de ejecución provoca un error que más o menos dice: "No se ha llamado a CoInitialize".

La función CoInitialize, junto a su extensión CoInitializeEx, pertenecen al API de COM, y sirven para inicializar este subsistema. Si no se ha llamado previamente a una de ellas, no se puede utilizar COM. El principal escollo es que, en ocasiones, hay que llamarlas para cada hilo del proceso: no basta con incluir una llamada a CoInitialize en el código de inicio del proyecto para eliminar el error descrito. Y la razón es que estas funciones tienen la importantísima responsabilidad de indicar el modelo de concurrencia COM de los hilos de un proceso, indicando el *apartamento* en el que debe ingresar un hilo que se inicializa para realizar llamadas a componentes.

Un *apartamento* en COM es una entidad lógica que agrupa a uno o más hilos; todo hilo que realiza llamadas COM, por su parte, tiene que pertenecer a un único apartamento. Simplificando un poco, hay dos tipos de apartamentos: STA (*single threaded apartment*) y MTA (*multiple threaded apartment*). Como sugieren sus nombres, en un STA sólo puede existir un hilo, al contrario de lo que sucede en un MTA. Además, en un proceso sólo puede existir un único MTA.

¿Cómo se determina el tipo de apartamento de un hilo? Esto sucede al inicializar el hilo para COM. Si el hilo llama a CoInitialize, que es una función obsoleta, se crea un STA nuevo, y el hilo entra en su interior. La nueva función CoInitializeEx permite pasar un valor entero como parámetro, que si vale COINIT_APARTMENTTHREADED también provoca la creación de un STA. Ahora bien, esta segunda función también permite indicar COINIT_MULTITHREADED, para pedir que el hilo entre en un MTA si ya existe uno. En caso contrario, el MTA se crea en ese momento. Si el hilo principal del proceso crea un MTA, y se crea un hilo nuevo que no llama a las funciones de inicialización de COM en absoluto, se entiende que el nuevo hilo desea formar parte automáticamente del MTA global existente. Recuerde esta última regla...

MÓDULOS DE DATOS TRANSACCIONALES

Si alguna vez ha creado un módulo de datos remoto para DataSnap, habrá visto el icono adyacente del Almacén de Objetos, para crear *módulos de datos transaccionales*, para MTS/COM+. Lamentablemente, muy pocos libros sobre Delphi tocan este tema, y creo que ninguno se lo toma muy a pecho. Incluso he leído alguna vez información increíblemente incorrecta sobre el uso de los módulos transaccionales.

¿Qué nos ofrece un módulo transaccional? En parte, depende de la interfaz de acceso a datos que estemos utilizando. Si se trata de ADO Express, tenemos suerte: podemos hacer que COM+ sea quien inicie y termine las transacciones del módulo remoto. En el módulo no tendremos que llamar jamás a los métodos BeginTrans, CommitTrans y RollbackTrans de la clase TADOConnection. Si usted conoce a fondo el comportamiento de los módulos remotos, especialmente el modo en que se manejan las transacciones dentro del método AS_ApplyUpdate, esta afirmación le sorprenderá: resulta que TDataSetProvider inicia

Lo que acabo de explicar son unas reglas para asociar hilos y apartamentos, pero ¿para qué se utilizan los apartamentos? Cuando llamamos a un componente COM, el código que realiza la llamada pertenece a un hilo, y por lo tanto, a un apartamento bien definido. Y con el objeto que debe recibir la llamada sucede exactamente lo mismo. Pues bien, de acuerdo al tipo de los dos apartamentos en juego, COM determina si permite una llamada directa, sin intermediarios, o si debe colocarse un *proxy* entre el cliente del objeto y el propio objeto. En determinados casos, esta interceptación corre a cargo del propio sistema; en otros casos, principalmente en servidores ejecutables, el programador del servidor tiene que echar una mano. No quiero entrar en detalles aquí, pero puede consultar la unidad VCLCom para que se haga una idea de las implicaciones.

Pero lo que ahora nos interesa es cómo resolver el problema de WebSnap con ADO. Está muy claro que hay que llamar a una de las funciones de inicialización de COM en "algún" lugar o momento. Ese momento puede ser la creación de un módulo de datos, antes de que se establezca la conexión a la base de datos. Podríamos llamar a CoInitialize, o pedir un STA explícitamente mediante CoInitializeEx... Pero mejor aún, podemos crear un MTA global, y evitar la creación de un STA separado para cada hilo. ¿Qué hacemos entonces, incluir una llamada a la segunda función en el fichero *dpr* del proyecto? Mejor que eso: cuando un proyecto de Delphi realiza llamadas a COM, el método Application.Initialize es el encargado de crear el apartamento del hilo principal del proceso. Podemos pedir que ese apartamento sea un MTA global, modificando una variable global de la unidad ComObj en la primera línea del código de inicio del proyecto:

```
begin
  CoInitFlags := COINIT_MULTITHREADED;
  Application.Initialize;
  // ... etcétera ...
end.
```

De esta forma, todo hilo creado dentro del proceso que no llame explícitamente a CoInitialize, como sucede en el nuevo WebSnap, pertenecerá automáticamente al MTA global, y podrá inmediatamente hacer uso de ADO, o de cualquier otro conjunto de componentes de COM.

también una transacción automáticamente al recibir datos para grabar procedentes de un conjunto de datos cliente. ¿Qué sucede si el proveedor se encuentra dentro de un módulo transaccional? La respuesta se encuentra en la implementación del método PSInTransaction, de la clase TCustomADODataset, en la unidad ADODB:

```
begin
  if Assigned(Connection) then
    Result := Connection.InTransaction
  else
    Result := False;
  if not Result then
    Result := InMTSTransAction;
end;
```

Como ve, a efectos del comportamiento de un TDataSetProvider, se considera que existe una transacción activa tanto si ha sido iniciada explícitamente por el programador, a través del

objeto de conexión, o si ha sido iniciada por el entorno de COM+, de forma automática.

¿Qué ventajas tiene el inicio automático de transacciones? A primera vista, cuesta un poco verlas: las características transaccionales de COM+ han sido diseñadas teniendo en mente un mundo de objetos de "grano fino", que interactúan entre sí constantemente. En contraste, en DataSnap se utiliza un modelo "grosso", en el que un único objeto (el módulo) es quien encapsula todos los servicios de capa intermedia. No obstante, hay un punto a favor evidente: gracias a COM+, *todas* las llamadas a métodos remotos del módulo quedan automáticamente incluidas en una transacción.

Como sabemos, si utilizamos módulos tradicionales, esto sólo ocurre con la grabación de datos, a través de `AS_ApplyUpdates`; otros métodos, como `AS_GetRecords`, quedan fuera del paraguas protector. Y debe saber que esto crea un problema serio cuando se utiliza InterBase a través de DB Express: si las transacciones implícitas de lecturas utilizan el nivel superior de aislamiento, puede que no sean inmediatamente visibles los cambios efectuados por el propio módulo. Además, se pueden crear muchas versiones de objetos innecesarias. Si InterBase tuviese un controlador OLE DB, podríamos aprovechar los módulos transaccionales para también englobar las lecturas en una transacción breve, con el nivel de aislamiento correcto. Por desgracia, *no* existe ese controlador, por el momento, y el programador que utiliza InterBase debe modificar el componente `TDataSetProvider` si quiere evitar ese problema (más detalles, en nuestro curso a distancia de DB Express/DataSnap).

Ahora bien, si las transacciones se inician automáticamente, ¿cuándo es que acaban? Un objeto transaccional, en general, puede influir "votando" con la ayuda de los métodos `SetComplete` y `SetAbort`, pertenecientes a su *contexto de objeto*. Gracias a que estas funciones representan votos a favor o en contra, podemos ejecutar sin mayores problemas métodos de otros objetos transaccionales; si conoce Transact SQL, sepa que se trata de un mecanismo parecido al del contador de anidamiento de transacciones de SQL Server. Por otra parte, si el programador olvida llamar a una estas funciones al terminar un método, COM+ espera cierto tiempo establecido para el sistema, y anula la transacción transcurrido dicho intervalo. Como buena noticia, los módulos transaccionales de Delphi incluyen una propiedad llamada `AutoComplete`, que si es activada, provoca que cada método de la interfaz `IAppServer` termine llamando a `SetComplete` al terminar sin errores. Si extendemos la interfaz del módulo, no obstante, las implementaciones de los nuevos

ACCIONES DE FILAS EN WEBSNAP

Esta es la historia de un evento perdido en el bosque. O más bien, de cómo un componente perdió sus quince minutos de fama por un olvido estúpido. El componente se llama `TDataSetAdapterRowAction`, y su antecesor casi idéntico e inmediato, llamado `TCustomDataSetAdapterRowAction`, es el ancestro de todas las acciones orientadas a filas disponibles en un adaptador de WebSnap para conjuntos de datos.

Antes de explicar la utilidad de este ignorado componente, debo confesar que, hasta el momento, he utilizado muy poco WebSnap en proyectos "serios" (y veremos que, al parecer, lo mismo sucede con la mayoría de los programadores de Delphi). Pero hace muy poco, tuve que desarrollar a toda prisa una aplicación para uso interno que debía combinar un acceso muy directo a datos, seguridad y privilegios de usuarios, Y elegí WebSnap, en parte para ver qué tal me iba.

En general, me fue bastante bien. Cuando dije "acceso muy directo" quise decir que el modelo de datos era muy simple, y que la aplicación debía permitir consultar determinadas tablas y modificar directamente algunas filas del resultado. Para este tipo de tareas, los adaptadores de conjuntos de datos (`TDataSet-`

métodos deben ocuparse explícitamente de votar sobre el resultado de las transacciones.

La ventaja principal de utilizar COM+, sin embargo, consiste en la posibilidad de reutilizar módulos de forma transparente para el usuario de los mismos. Como precondition, es necesario antes que el objeto pueda ser desactivado y reactivado por demanda, pero esta característica se activa automáticamente si al registrar la clase indicamos que requiere transacciones automáticas. El efecto de reciclar módulos transaccionales es similar al que ocurre si utilizamos `TwebConnection` y activamos la caché de objetos ejecutando `RegisterPooled` durante el registro de la clase del servidor. Pero con `TsocketConnection` no existe directamente esa posibilidad... a menos que utilicemos módulos transaccionales. Para que un objeto pueda ser reutilizado, su clase debe implementar la interfaz `IObjectControl`, tal como hacen los módulos transaccionales. Estos publican una propiedad llamada `Pooled`, que debemos activar para permitir el reciclado.

Basta con que activemos la activación por demanda (ni siquiera es necesario que los módulos se reciclen) para que un módulo transaccional se transforme en una útil herramienta de depuración. Y aunque parezca sorprendente, la explicación es inmediata: por lo general, las pruebas iniciales de una aplicación multicapas las realiza el propio desarrollador... en un entorno en que él mismo es el único usuario. Por lo tanto, se le pueden escapar problemas o suposiciones incorrectas que sólo se detectarían con suficiente carga de concurrencia. Uno de tales problemas suele ser la paginación de conjuntos de datos clientes por demanda. Es muy sencillo activar la propiedad `PacketRecords` en el conjunto de datos clientes y dejar que la continuidad entre los grupos de registros recuperados la garantice la permanencia del estado del conjunto de datos remotos en el servidor. Incluso si el desarrollador está haciendo sus pruebas con una conexión Web, a menos que lance dos copias simultáneas de la aplicación, no se percatará del comportamiento anómalo.

Sin embargo, si está trabajando con conexiones ADO en un módulo transaccional, COM+ se encargará de desactivar el módulo entre llamada y llamada, y entre otras cosas cerrará los cursores que el programador pueda dejar inadvertidamente abiertos. Es decir, que un módulo transaccional puede obligarnos a trabajar con objetos remotos "sin estado" (*stateless*), lo que nos puede ayudar a simular un entorno de trabajo más verosímil.

Adapter) y las acciones predefinidas por WebSnap se prestan estupendamente.

El problema surgió porque necesitaba realizar una operación muy sencilla: tenía que insertar de manera automática y con una pulsación de un botón, cierto grupo de registros como detalles de una relación maestro/detalles. La consulta maestra estaba disponible en una tabla HTML, y la acción debía dispararse pulsando un botón asociado a cada fila de la tabla mencionada. Lo que hice fue crear una nueva acción dentro del adaptador asociado a la tabla maestra, para indicar su funcionamiento mediante eventos. ¿Qué tipo de acción debía utilizar como base? Recurrí a una simple `TAdapterAction`, la más elemental de todas. Mediante su evento `OnGetParams` la obligué a almacenar la clave primaria de la tabla maestra, para la fila seleccionada por el usuario. Y en la respuesta a su evento `OnExecute`, recuperé ese parámetro para localizar nuevamente la fila y modificarla según era necesario. Funcionó.

No obstante, enseguida me surgió la duda: ¿no existirá algo más sencillo, que no me obligue a recordar la clave primaria en un evento, y a localizar explícitamente la fila necesaria durante la

ejecución de la acción? Resulta que sí: podía haber utilizado como base el componente `TDataSetAdapterRowAction`. Ahora bien, cuando intenté sustituir la acción simple por el nuevo objeto tropecé con la desagradable constatación de que este componente no publica el evento `OnExecute`! En la ayuda de Delphi, sin embargo, dice muy claramente que podemos utilizar dicho componente para acciones que actúen sobre una fila determinada del conjunto de datos, y que para ello tenemos que interceptar su evento `OnExecute`. Lo que ha sucedido, simplemente, es que "alguien" ha olvidado promocionar el evento mencionado a la sección **published** del componente. Otro descuido tonto... y una indicación de que muy poca gente está utilizando WebSnap en serio.

LIBROS RECOMENDADOS

Ultimamente, leo bastante sobre la plataforma .NET de Microsoft. Por supuesto, no es que tenga interés alguno en programar en Visual Basic.NET (la mona vestida de seda... ya se sabe), pero sí estoy preparándome para Delphi.NET, y puede que en algún momento necesite hacer algo con ASP.NET.

En esta época, hay montones de libros sobre .NET gritando como las galletas de crecimiento del País de las Maravillas: "¡cómprame, cómprame!". He comprado unos cuantos, y muchos de ellos me han decepcionado, incluso algunos escritos por autores de renombre. Por este motivo, me he alegrado al topar finalmente con un buen libro. Esta es su edición en inglés:

Applied Microsoft .NET Framework Programming
Autor: Jeffrey Richter
Editorial: Microsoft Press
ISBN: 0735614229

La sangre no llegó al río, afortunadamente. Es posible utilizar, alternativamente, el evento `OnAfterExecute` de la acción. Revisando el código fuente del componente, se puede comprobar que al activarse la acción en el lado servidor se ejecuta el método `ImplExecuteActionRequest`, y que cuando éste detecta que no hay asignado un manejador para el evento `OnExecute` dispara este código alternativo:

```
if not Adapter.LocateAndApply(  
    AActionRequest) then  
    Adapter.EchoActionFieldValues := True;
```

Por consiguiente, se localiza la fila necesaria y podemos actuar sobre ella, en el evento `OnAfterExecute` de la acción, o en `OnAfterExecuteAction`, del adaptador que la contiene.

Applied Microsoft .NET Framework Programming es un libro escrito por Jeffrey Richter, también autor de algunos libros muy interesantes (Programación Avanzada en Windows, por ejemplo, donde aprendí lo que sé sobre procesos, hilos y semáforos). Este nuevo libro no se dedica a describir un lenguaje específico, aunque como es natural, encontrará unos cuantos listados en C#. Por el contrario, el centro de interés es la propia plataforma .NET, conocida con el nombre de CLR (*Common Language Runtime*). Por ejemplo, se dedica todo un capítulo a describir la recolección de basura, con un sorprendente nivel de detalles. Mi opinión es que, al ser el CLR aproximadamente similar al API de Windows para las aplicaciones nativas, es sumamente importante comprenderlo bien para desarrollar aplicaciones... cuando el Delphi.NET esté finalmente disponible.

PROPIEDADES DE TIPO INTERFAZ

Hace ya algún tiempo, escribí un artículo para la sección de trucos de mi página, y lo llamé *Creación automática de componentes* (www.marteens.com/trick07.htm) Trataba principalmente del tratamiento estándar de las propiedades de tipo clase en los componentes de Delphi. Por ejemplo, explicaba que, en su mayor parte, las propiedades que apuntan a un derivado de TComponent son referencias. El componente que contiene la propiedad no suele ser el propietario del objeto, y por el contrario, debe estar muy atento a la posibilidad de que el objeto referido sea destruido en algún momento.

También mencionaba que en casi todas las restantes propiedades de tipo clase (TStrings, TFont, TPicture), se consideraba que el objeto apuntado por la referencia era propiedad del objeto principal, y que éste debía no sólo ocuparse de la construcción y destrucción del mismo, sino también de implementar las asignaciones a dicha propiedad mediante copias, y vigilar las asignaciones a subpropiedades del objeto "incrustado". Como consecuencia de ese artículo, escribí un sencillo asistente para el entorno de Delphi que generaba componentes, permitiendo especificar propiedades y qué semántica debía satisfacer cada una de ellas.

En estos días he estado actualizando el código original del asistente... que era bastante sucio, por cierto. Como ha pasado tanto tiempo desde entonces, Delphi ha añadido dos tipos adicionales de propiedades: las *referencias a interfaces*, y los *punteros a subcomponentes*. En este boletín vamos a ver en qué consisten las primeras, dejando los subcomponentes para más adelante.

¿Por qué deben ser diferentes las propiedades que apuntan a una interfaz y las que apuntan a un objeto directamente? En realidad, sólo son especiales las propiedades de tipo interfaz que están en una sección **published**, o que en alguna clase descendiente pueden moverse a dicha sección. El primer tropiezo lo provoca la forma en que Delphi "serializa" componentes para almacenarlos en ficheros *dfm*. Cuando un componente que se está guardando en un *dfm* tiene una propiedad publicada que apunta a un descendiente de TComponent, Delphi escribe en el *dfm* el nombre de dicho objeto; si el componente en cuestión reside en un módulo o formulario diferente, se incluye como prefijo cualificador el nombre de dicho módulo o formulario. Con esta técnica, cuando hay que restaurar el componente original y se lee el valor de la propiedad mencionada, el nombre del componente referido se almacena en una lista de *fix-ups*. Al terminar la lectura del fichero *dfm*, Delphi realiza una pasada sobre los *fix-ups* para buscar los componentes apuntados por propiedades y corregir entonces los valores de las propiedades con los punteros correctos. Todo este jaleo es responsabilidad, en su mayor parte, de la clase TReader.

Está claro que una propiedad de tipo interfaz no entra con facilidad en este esquema de serialización; para empezar, apunta sólo a un "trocito" de un componente, en el mejor de los casos, claro.

La solución más lógica es la aplicada por Delphi: disponer de algún mecanismo para, dado un puntero de interfaz, poder encontrar el componente original, en toda su santa gloria. La operación inversa no es problema: dado un componente, extraer de él interfaz de determinado tipo. Para esto último existen las mil y una variantes de la función Supports:

```
if Supports(MiComponente, ITipoInterfaz,
PunteroInterfaz) then
// PunteroInterfaz es la respuesta ...
```

Para encontrar el componente, dado un puntero de interfaz, Delphi hace que TComponent implemente el tipo de interfaz IInterfaceComponentReference, introducido en Delphi 6:

```
type
  IInterfaceComponentReference = interface
    ['{E28B1858-EC86-4559-8FCD-6B4F824151ED}']
    function GetComponent: TComponent;
  end;
```

La implementación que TComponent hace del método anterior es muy sencilla: se limita a devolver un puntero a sí mismo. Por lo tanto, si quiero saber qué componente es el que implementa un puntero de interfaz que tengo en mi mano, sólo tengo que seguir un par de pasos:

- 1 Preguntar al puntero de interfaz, si su objeto subyacente implementa también IInterfaceComponentReference. Para eso puede utilizar el método QueryInterface, que todas las interfaces incluyen.
- 2 Con un puntero de tipo IInterfaceComponentReference en la mano, sólo nos queda ejecutar su método GetComponent para llegar al componente.

Como puede imaginar, de ahí en adelante el proceso de guardar el valor de una propiedad de tipo interfaz vuelve a coincidir con la técnica para guardar un puntero normal a componente: se almacena el nombre del componente que implementa la interfaz.

Pongámonos ahora en el papel de un diseñador de componentes que tiene que crear una propiedad publicada de tipo interfaz. Estas son las reglas que debe cumplir:

- 1 Para empezar, no hay restricciones en el acceso de lectura de la propiedad: podemos emplear un campo, o la habitual función GetXXX.
- 2 Por supuesto, el acceso para escritura debe realizarse a través de un método. Supongamos que la propiedad que vamos a implementar se llama Propiedad. Podríamos utilizar una variable FProp y un método SetProp en su declaración:

```
private
  FProp: IMiInterfaz;
  procedure SetProp(Value: IMiInterfaz);
  // ...
published
  property Prop: IMiInterfaz
    read FProp write SetProp;
  // ...
```

- 3 En el método de escritura debemos obligatoriamente incluir un par de llamadas al método auxiliar ReferenceInterface, de la clase TComponent. Este método, que recibe el puntero de interfaz a asignar y un parámetro enumerativo para indicar si se añade o elimina la referencia al puntero, se encarga de llamar a los conocidos FreeNotification y RemoveFreeNotification, para que el componente asociado al puntero de interfaz avise al componente que diseñamos cuando sea destruido. El patrón del método debe parecerse a esto:

```
procedure TMiComp.SetProp(Value: IMiInterfaz);
begin
  ReferenceInterface(FProp, opRemove);
  FProp := Value;
  ReferenceInterface(FProp, opInsert);
end;
```

- 4 Como consecuencia de lo anterior, debemos redefinir el método virtual Notification, de manera muy parecida a como lo haríamos para un propiedad que haga referencia a un componente independiente:

```

procedure TMIComp.Notification(
  C: TComponent; Op: TOperation);
begin
  inherited Notification(C, Op);
  if Op = opRemove then
    if C.IsImplementorOf(FProp) then
      SetProp(nil);
end;

```

Observe el uso del método `IsImplementorOf` de la clase `TComponent`. Este método verifica si el componente que se va a eliminar es, precisamente, el componente que implementaba

CONTROLES DE LISTAS VIRTUALES

En mis propias aplicaciones, practico el uso y abuso del control `TListView`. Por ejemplo, muchas veces lo utilizo como sustituto en modo sólo lectura del omnipresente `TDBGrid`, poblándolo de elementos mediante rutinas auxiliares muy generales. Otras veces, recorro a él como sustituto sofisticado de la vieja `TListBox`.

Pero `TListView` no es precisamente una gema de buen comportamiento. La culpa en parte es de Microsoft, que ofrece una funcionalidad de base algo retorcida y difícil de usar, y en parte de Borland, que ha encapsulado bastante chapuceramente el control de Microsoft. Sea quien sea el responsable, lo cierto es que `TListView` consume mucha memoria para sus elementos. Cuando lo utilizo como rejilla de datos, esto no es demasiado grave, porque al trabajar con `DataSnap`, lo normal es que traiga al lado cliente, a la capa de presentación, muy pocos registros de datos en cada consulta. De todos modos, la preocupación por el consumo de memoria existe...

Por fortuna, hay un mecanismo imbricado dentro del control que permite ahorrar memoria: utilizar el control como *control de lista virtual*. Traduzco: no hace falta crear un elemento de tipo `TListItem` para cada línea del control, sino que podemos instruirlo para que utilice un único `TListItem` "físico", suministrando nosotros los valores asociados de cada línea "lógica" mediante eventos. Si nuestra intención es sustituir un `TDBGrid`, la técnica implica que vamos a estar moviendo el cursor activo del conjunto de datos asociado todo el tiempo. Por lo tanto, sólo tiene sentido trabajar con este tipo de lista virtual cuando el conjunto de datos es un `TClientDataSet`, como supondremos de ahora en adelante. Incluso con tal suposición, mover desenfrenadamente la fila activa de un conjunto de datos cliente es peligroso, porque ese movimiento puede provocar el disparo automático de muchos de los eventos del conjunto de datos. En consecuencia, para evitar efectos secundarios sobre las posibles modificaciones que podamos hacer en paralelo sobre el `TClientDataSet` cuyos registros vamos a mostrar, "clonaremos" el conjunto de datos original, de manera que el clon comparta la memoria de datos del original... pero con una posición del cursor independiente.

Para activar el modo "virtual" del control de listas, debemos asignar `True` en la propiedad `OwnerData` del `TListView`. Tenga cuidado y no confunda esta propiedad con `OwnerDraw`, que se utiliza para modificar la forma en que se dibuja el control. Cuando se activa `OwnerData`, el control espera:

- 1 Que le digamos el número de líneas o elementos que debe mostrar, asignando ese valor directamente en su propiedad `Items.Count`. Aunque no lo parezca, dicha asignación *no* provoca la creación del correspondiente número de objetos físicos.
- 2 Que escribamos, como mínimo, una respuesta para el evento `OnData` del control, para que indiquemos el contenido a mostrar para cada línea o elemento.
- 3 Si estamos de buen humor, podemos interceptar también el evento `OnDataFind`, para implementar la búsqueda incremental sobre el control. Si nuestro humor es aún mejor, quizás nos interese interceptar `OnDataHint` también, que nos ayuda a mejorar la eficiencia de la lista al agrupar la actuali-

nuestro puntero a interfaz. `IsImplementorOf`, como puede suponer, aprovecha la existencia del tipo `IInterfaceComponentReference` en su propia implementación.

Para terminar, debo advertirle que, aunque he mencionado el uso de `IInterfaceComponentReference` en esta técnica, la propia Borland recomienda que no utilicemos ese tipo directamente en nuestro código, porque la técnica de implementación de los propiedades de tipo interfaz puede cambiar en alguna versión futura de Delphi. Por mi parte, queda advertido...

zación de elementos. Y si estamos de ánimo esotérico, hay un evento `OnDataStateChange` escondido por ahí, para tener en cuenta el cambio de "estado" de cada elemento de la lista virtual.

Como por algo tenemos que empezar, traiga a un formulario vacío un `TClientDataSet`, al que llamaremos `Emps`, para abreviar, y un `TListView`, al que con mucha originalidad llamaremos `LView`. Para cargar datos en el conjunto de datos, haga que su propiedad `FileName` apunte al fichero `employees.xml`, de las demos de Delphi; cuando lo haya hecho, cree y configure cada uno de sus seis objetos de acceso a campo. Vaya entonces a la sección **private** de la clase del formulario, y declare una variable como la siguiente:

```

private
  Clon: TClientDataSet;

```

Luego, dé el siguiente tratamiento al evento `OnCreate` del formulario, para inicializar el clon, y configurar el control de lista. Observe la asignación sobre la propiedad `OwnerData` de este último, casi al final del método:

```

procedure TMain.FormCreate(Sender: TObject);
const
  DF = 'DisplayFormat';
var
  I: Integer;
  C: TListColumn;
  F: TField;
begin
  Emps.Open;
  Clon := TClientDataSet.Create(Self);
  Clon.CloneCursor(Emps, False);
  for I := 0 to Emps.FieldCount - 1 do
    begin
      C := LView.Columns.Add;
      F := Emps.Fields[I];
      C.Caption := F.DisplayLabel;
      C.Alignment := F.Alignment;
      C.Width := LView.StringWidth('M') *
        F.Displaywidth;
      if TypInfo.IsPublishedProp(F,DF) then
        SetStrProp(Clon.Fields[I], DF,
          GetStrProp(F, DF));
    end;
  LView.ViewStyle := vsReport;
  LView.OwnerData := True;
  LView.Items.Count := Clon.RecordCount;
end;

```

Al haber activado `OwnerData`, estamos obligados a manejar el evento `OnData`:

```

procedure TMain.LViewData(Sender: TObject;
  Item: TListItem);
var
  I: Integer;
begin
  Clon.RecNo := Item.Index + 1;
  Item.Caption := Clon.Fields[0].DisplayText;
  for I:=1 to LView.Columns.Count-1 do
    Item.SubItems.Add(
      Clon.Fields[I].DisplayText);
end;

```

En pocas palabras, nos pasan un elemento como parámetro. Su única propiedad correctamente configurada es el índice, es decir,

su posición virtual dentro de la lista. Somos entonces responsables de suministrar cadenas para la primera columna (Caption) y para las restantes columnas, por medio de la propiedad SubItems.

C#: EL LENGUAJE

Tengo que reconocer que estoy muy entusiasmado con C#, más que con la famosa plataforma .NET. Quizás tendría que comenzar explicando las actuales características de C#, pero eso no es tema para un solo boletín. Además, muchos de vosotros probablemente estéis más al día respecto al mundo .NET que un servidor. Para aquellos pocos que no sepan de qué va el asunto, ahí va un resumen, muy personal, sobre cómo veo la nueva plataforma:

Básicamente se trata de que Microsoft ha querido crear su propio Java... pero en el proceso, las cosas han terminado tomando un cariz más prometedor; hay que reconocer que han sabido evitar la mayoría de los errores de la competencia, y al final, sólo hay semejanzas aparentes entre ambos entornos. Para empezar, el **.NET Framework**, que es como se conoce la plataforma micro-sófica, tiene una máquina virtual por debajo, que le sirve como base, al igual que Java tiene la suya (la JVM). No obstante, la de Microsoft está íntimamente ligada a un único sistema operativo (teóricamente puede transportarse a otros sistemas), mientras que los servicios básicos ofrecidos por la JVM necesariamente son menos concretos, al tener Java la portabilidad entre sus objetivos primordiales de diseño (que lamentablemente tampoco llegan a cumplirse). La máquina virtual de Microsoft, o CLR (*Common Language Runtime*), es más potente: está pensada para traducir a código binario las partes de las aplicaciones que más se ejecuten, de forma totalmente automática. Pero lo principal es que parte de unas operaciones básicas más pegadas al hardware. En Java, por el contrario, se decidió utilizar un modelo de ejecución basado en unas pocas primitivas: la decisión típica que hubiera tomado un chaval que prepara su trabajo de fin de carrera.

Java observa la regla: un lenguaje, muchos sistemas operativos. Por el contrario, en .NET son compatibles muchos lenguajes para un mismo sistema operativo... aunque en teoría sea posible la migración a otras plataformas. Cualquier lenguaje que respete las reglas básicas de la máquina virtual CLR puede compilar código que además de ejecutarse en esta plataforma, puede aprovechar componentes desarrollados *por* otros lenguajes, o desarrollar componentes *para* otros lenguajes. Ahí es donde Delphi tiene su oportunidad: no es secreto que Borland está preparando una versión de Delphi que producirá código para la CLR. De todos modos, el lenguaje estrella de la fiesta es C#, desarrollado por la misma persona que creó Delphi, y que abandonó Borland por Microsoft al salir Delphi 3 a la venta.

¿Cuál es mi interés personal en .NET? Sinceramente, por muy bueno que sea el entorno .NET, de momento no me convence demasiado trabajar sobre un entorno en el que haya un intérprete, aunque esté a cinco kilómetros de distancia. No obstante, puede que este entorno termine convirtiéndose, con el paso de los años y la aprobación del mercado, en el *núcleo* del propio Windows, por lo que ningún programador que trabaje en esta plataforma puede permitirse el lujo de perder de vista esta vía. En definitiva, .NET promete simplificar enormemente el desarrollo de aplicaciones, y por lo que he visto, la promesa se está cumpliendo. Además, a corto plazo hay ventajas: por ejemplo, hay una versión del entorno .NET que funciona en dispositivos de mano (PDA). La plataforma de desarrollo para bases de datos de .NET, el llamado ADO.NET, es una versión muy mejorada de Midas (aunque ni Microsoft ni Borland estén dispuestos a reconocerlo), y la programación para Internet con el nuevo ASP.NET parece atractiva y potente a la misma vez. Para rematar, el futuro de COM+ apunta en la misma dirección, aunque para su sustitución completa falte aún bastante tiempo.

¿Y qué hay con C#, el lenguaje que puede competir con Delphi por la lealtad de nuestros corazones? Debo confesar que cada vez

Si le sobra algo de tiempo, puede probar a interceptar el evento `OnColumnClick` para reordenar el conjunto de datos y `OnDataFind`, para la búsqueda incremental de registros.

me gusta más C#, por lo que es mejor que comience diciendo lo que no me gusta de éste: su críptica sintaxis al estilo C. Por lo demás, es un lenguaje orientado a objetos *puro*, al contrario que Delphi, que es un híbrido. La ventaja es que es más fácil de aprender... y de explicar. Y esa pureza de C#, al contrario de lo que sucede con Java, no implica debilidad, ni limitación en los recursos. Por ejemplo, C# ofrece "estructuras", para evitar tener que usar clases para todo, y existen mecanismos muy elegantes para forzar determinado formato en memoria de los tipos de datos que deseemos (por medio de atributos). Además, existen punteros a eventos, muy parecidos a los de Delphi, aunque más potentes, porque permiten el envío simultáneo del evento a más de un receptor. Recordemos que Java, por afán de ortodoxia, no ofrece esa característica.

En la misma línea de razonamiento, C# soporta directamente *propiedades*, casi iguales a las de Delphi. En Java, las propiedades no existen como construcciones del lenguaje. Eso hace que cualquier entorno de desarrollo se las pase canutas para poder habilitar un simple Inspector de Objetos. Quien diga que Java es un lenguaje apropiado para el "desarrollo rápido de aplicaciones" (RAD) miente como un bellaco, o no sabe de lo que está hablando. C# es a la vez un lenguaje orientado a objetos *puro*, y un lenguaje RAD muy práctico y funcional.

Todo este incienso que estoy quemando en honor de C# tiene una explicación: he descargado recientemente, de la página de C# (www.csharp.net) el anuncio de las novedades para la segunda versión del lenguaje. Estas se conocían desde noviembre del año pasado, pero al leer los detalles, no he podido evitar el entusiasmo, porque se trata de características que vengo rogando (en vano) a Borland que introduzca para Delphi desde hace mucho tiempo. La primera de las novedades se conoce técnicamente como *genericidad*, pero la pobre gente que sufre con C++ la conoce mejor como *plantillas*, o *templates*. Se acabaron los malditos *typecasts* sobre los objetos extraídos de una conexión, se acabó el tener que programar clases especiales cuando lo que hay que meter en una lista es un simple valor entero o flotante, y no una clase. La solución de implementación anunciada por el equipo de C# parece ser muy inteligente. Y, a diferencia de la propuesta similar de ampliación de Java, la máquina virtual CLR interviendrá en la implementación, por lo que el *runtime* no perderá información alguna sobre los objetos con los que trabaja.

En segundo lugar, la próxima versión introducirá la posibilidad de definir *iteradores*. En realidad, ya C# soporta una instrucción **foreach**, que simplifica el recorrido sobre determinados tipos de conjuntos y colecciones. Pero para que una clase soporte **foreach**, actualmente es necesario que dicha clase implemente la interfaz `IGetEnumerator`, para devolver otro objeto que a su vez debe implementar la interfaz `IEnumerator`. O dicho con pocas palabras, es un trabajo de chinos. Por el contrario, en la versión 2 de C# se podrá definir un método, con el nombre especial de **foreach**, que simule el recorrido sobre los elementos de la clase, de forma parecida a como funcionan los procedimientos de selección en InterBase con la instrucción `SUSPEND`, que en C# se convertirá en **yield**. En realidad, el primero lenguaje que popularizó el uso de iteradores fue CLU, el lenguaje de la mítica investigadora Barbara Liskov.

Las otras dos novedades tienen que ver más con la facilidad de uso del lenguaje. Por una parte, se podrán utilizar *métodos anónimos*, que simplificarán enormemente el código de interceptación de eventos, entre otras cosas. Y por último, C# soportará *tipos parciales*, que permitirán distribuir la declaración e implementación de una clase entre varios ficheros compilables por separado.

Son avances menores, en comparación, pero que también se agradecen.

Es muy triste constatar que desde la ya lejana versión 4 de Delphi no se han añadido mejoras significativas al propio lenguaje. No obstante, es muy probable que la necesidad de hacer Delphi lo más compatible posible con C# obligue a Borland a retocar algunas partes de Delphi que ya lo exigen a gritos. Estaremos MUY atentos...

NOTA: ¿Por qué C#? Además de lo obvio, que es la inspiración sintáctica en C++ (aunque reconozcamos que sea a través de

Java), hay dos lecturas adicionales del símbolo en inglés. Por una parte, C# significa Do Sostenido, en notación abreviada musical. La famosa *Moonlight Sonata* de Beethoven (lo de *moonlight* no fue cosa del Gran Sordo) está escrita en Do Sostenido Menor, lo mismo que algunos vales muy conocidos de Chopin. En inglés, al carácter # lo conocen como "hash", entre otras denominaciones. Pero bajo la denominación musical, se lee como "sharp", es decir, "sostenido". Por lo tanto, C# se pronuncia oficialmente "see sharp"... que significa "ver con agudeza", o "ver inteligentemente". Un nombre afortunado, ¿o no?

BUGS EN SERVICIOS WEB

Ultimamente, los *bugs* asolan la faz de la tierra délfica. Uno de los más molestos, y aún no resueltos, hace imposible el uso de servicios Web implementados como módulos ISAPI en Delphi 7. Al parecer, *algo* que hacen estos servicios provoca un error dentro de la aplicación, que el motor de IIS coloca en el buffer de salida HTTP. La primera vez que se ejecuta una petición todo marcha de maravillas, pero la segunda petición ya no funciona. No me extraña nada: estoy teniendo montones de problemas con aplicaciones más normales cuando Delphi alcanza el código de terminación de muchas unidades.

¿Solución? De momento ninguna. Si tiene mucha prisa por utilizar un servicio Web escrito en Delphi, tendrá que utilizar el modelo

CGI, y aceptar la pérdida de rendimiento. No sé si este problema afecta también o no a las DLL para Apache, pero en caso negativo, sería una salida para algunos. Me han comentado también la posibilidad de desactivar la característica conocida como *Keep-alive*, que mantiene abierto el *socket* entre cliente y servidor, pero no lo he ensayado todavía.

En el mismo orden de cosas, ya está disponible un parche no oficial del componente *ActionManager* en el Code Central de Borland (codecentral.borland.com). Con ese parche ya es posible desarrollar aplicaciones MDI con los ya no tan nuevos controles de Borland. Busque el artículo por su identificador: 19151. Y no olvide que se trata de un parche todavía "extraoficial".

EL TIPO MÁS RÁPIDO DE CURSOR EN ADO

Reconozco que, en estos momentos, prefiero desarrollar mis aplicaciones con SQL Server antes que con InterBase. Como el tan esperado controlador de DB Express para SQL Server sigue sin funcionar como debería, me veo obligado a acceder a las bases de datos a través de ADO Express.

Como es de sobra conocido, ADO ofrece una amplia variedad de tipos de cursores, que son algoritmos de evaluación de consultas y de recuperación de resultados. Cuando se trata de SQL Server, hay dos tipos principales de cursores (con algunas simplificaciones):

- 1 Cursores en el lado cliente:** Todos los registros de la consulta son traídos de golpe al lado cliente, y se almacenan en una estructura local en memoria dinámica. En este sentido, se parecen un poco a los conjuntos de datos cliente de DataSnap. Un cursor en el lado cliente puede filtrarse, reordenarse arbitrariamente y sus filas pueden ser modificadas directamente.
- 2 Cursores en el lado servidor:** La alternativa, para no traer todos los registros de golpe, es crear una estructura de datos en el servidor que permita recuperar uno a uno esos mismos registros. Hay tres variantes para este tipo de cursor:
 - **Unidireccionales:** Se asemejan a los conjuntos de datos de DB Express. No es posible navegar hacia atrás, y la única forma de llegar al final es recorrer todos los registros intermedios. Pero no exige un *buffer* en el lado cliente, y la implementación en el servidor es la menos costosa.
 - **Conjuntos de claves:** Permite la navegación en los dos sentidos. Se logra creando una lista con las claves primarias de los registros devueltos por la consulta. El precio que se paga es que no son visibles las inserciones sobre las tablas subyacentes... aunque esto no sea intrínsecamente malo, sino todo lo contrario.
 - **Dinámicos:** Es la implementación más compleja y costosa. Permite la navegación irrestricta, y se pueden ver los cambios hechos en paralelo sobre las tablas subyacentes. Pero su alto consumo de recursos los hacen desaconsejables en la mayoría de los casos.

Si tuviese que programar una aplicación en dos capas, sin poder utilizar conjuntos de datos, probablemente elegiría los cursores en el lado cliente, entre otros motivos por su mayor flexibilidad. También podría utilizar cursores de conjuntos de claves para mostrar registros en una rejilla, pero el coste en velocidad es notable. El primer registro se recibe con mayor rapidez, por lo general, pero el tiempo total de navegación sobre la consulta es considerablemente mayor. Por este motivo, incluso en las aplicaciones en las que utilizo DataSnap, hasta el momento, he utilizado cursores en el lado cliente para acceder a la base de datos.

APLICACIONES ISAPI: DEPURACIÓN

¿Ha intentado alguna vez depurar una DLL correspondiente a una aplicación ISAPI? Es un verdadero dolor de cabeza. Para empezar, como se trata de una DLL, debemos indicar al depurador de Delphi la aplicación ejecutable que cargará nuestra DLL. En la lejana época de Windows 95, esa aplicación "anfitriona" podía ser el Personal Web Server para W95, una aplicación pequeña, con las opciones de configuración indispensables. Pero a partir de W98, las cosas fueron complicándose.

Para no divagar demasiado, centrémonos en lo que sucede con Internet Information Services, sobre Windows 2000 y Windows XP

Sin embargo, esa decisión siempre me ha provocado cargos de conciencia. Un cursor de ADO en el lado cliente utiliza su propia caché de registros. DataSnap, a continuación, creará otra caché en el lado cliente, con la ayuda de `TClientDataSet`. ¿No estamos derrochando memoria? Efectivamente, y la analogía con DB Express nos sugiere que sería mejor utilizar los cursores unidireccionales de ADO si vamos a utilizar también DataSnap.

Por desgracia, mis pruebas iniciales me demostraban que, a pesar de todo, los cursores ADO en el lado cliente se portaban más eficientemente que los cursores unidireccionales, también de ADO, algo sorprendente, ¿no? Por lógica, seguí utilizando cursores en el cliente para alimentar mis `TDataSetProvider` en los módulos de capa intermedia. Y debo reconocer que la eficiencia, hasta el momento, ha sido más que aceptable.

Hace poco decidí repetir las pruebas... y encontré la causa de la paradoja: inunca se me había ocurrido configurar los conjuntos unidireccionales de ADO para que fuesen también de sólo lectura! Arrastre un componente `TADOQuery` a un módulo de datos o formulario, y configure estas propiedades:

- 1 CursorLocation:** `clUseServer`
- 2 CursorType:** `ctOpenForwardOnly`
- 3 LockType:** `ltReadOnly`

Los cursores así configurados son los más veloces para suministrar registros a un `TDataSetProvider` de DataSnap (por supuesto, tendrá que configurar la conexión a la base de datos y la consulta SQL para comprobarlo). No es que la diferencia en velocidad sea excesivamente grande, aunque pueden completar el recorrido en dos terceras partes del tiempo consumido por un cursor en el lado cliente. Pero es que aparte de la pequeña ventaja de la velocidad, reducimos de paso el consumo de memoria del servidor de capa intermedia.

¿Quiere decir esto que debemos utilizar *siempre* este tipo de cursor, cuando trabajemos con DataSnap? ¡NO, de ningún modo! Déjeme terminar: así sería en un mundo perfecto. Pero al parecer hay un *bug*, o una mala especificación en la conducta de `TDataSet` que hacen que DataSnap falle si intenta extraer datos de una relación maestro/detalles establecida entre dos cursores unidireccionales de ADO. Puede incluso que el problema guarde alguna relación con los errores del controlador DB Express para SQL Server.

En consecuencia, y hasta que no sepa más sobre este asunto, sigo creando mis relaciones maestro/detalles con cursores ADO en el lado cliente. Y eso sí, cuando tengo que devolver los registros de una sola consulta, sin detalles, configuro el conjunto de datos de ADO Express para que devuelva un cursor creado en el lado servidor, unidireccional y ... no lo olvide, que sea también de sólo lectura.

Professional. En estas plataformas, el servidor HTTP se ejecuta por lo regular como una aplicación de servicio. Si quisiéramos depurar una DLL ISAPI, tendríamos que detener el servidor HTTP, y configurar I.I.S. para que pueda ejecutarse como aplicación "normal". Esto implica cambiar muchas entradas en el registro de Windows, y dejar casi inútil a I.I.S., con el riesgo de no poder revertir siempre los cambios de configuración. Y están también todos los problemas relacionados con cuentas de usuarios y permisos... Para aliviar la situación, Borland introdujo en Delphi 6 un pequeño servidor HTTP con "trampas", el Web Application De-

bugger (WAD), que es el tipo de proyecto recomendado para el desarrollo de extensiones de servidor para Internet.

No obstante, Web App Debugger tiene sus propios problemas. El menor de ellos es que obliga a registrar clases COM en la máquina de desarrollo... con lo que el registro de Windows termina hecho un asco al cabo de un par de semanas de trabajo. Por otra parte, para generar la aplicación final, hay que crear un proyecto vacío ISAPI en paralelo, y añadir a éste los módulos y unidades del proyecto original para el Web App Debugger. Si no tenemos cuidado, podemos terminar cargando en la DLL con código de registro de clases particular del WAD. Y lo verdaderamente preocupante, en mi opinión, es que la sintaxis de las peticiones HTTP a una aplicación WAD es bastante diferente a la sintaxis habitual que aceptará la DLL ISAPI. Esto significa que, si no prestamos suficiente atención, la DLL final puede tener algún que otro problema si tiene que analizar o descomponer la URL que recibe con cada petición...

Hace pocos días, di con la página www.msdelphi.com, de Alessandro Federici, que recomiendo sinceramente. Uno de los artículos describe una técnica para depurar extensiones ISAPI con IIS, que en mi opinión es bastante sencilla. El artículo se llama *Debugging IIS5 the easy way*. Aquí incluyo un resumen de la técnica, pero no incluyo las imágenes, que podéis consultar en el artículo original:

Supongamos que tenemos una DLL ISAPI en un directorio virtual de I.I.S. Para precisar, utilizaré como ejemplo el proyecto en que estoy trabajando ahora mismo, la versión 2 de Classique. En mi ordenador, que ejecuta Windows XP Professional, hay un directorio llamado *classique*, dentro del directorio principal de IIS (*inetpub*). He otorgado el permiso para ejecutar DLLs dentro de este directorio, utilizando el Administrador de IIS, y puedo acceder a mi aplicación, de forma local, utilizando la siguiente URL:

```
http://localhost/classique/classique.dll
```

Estos son los pasos necesarios para depurar la aplicación:

- 1 Abrimos la aplicación *Servicios de Internet Information Server*, dentro del menú *Herramientas administrativas*, o desde el Panel de Control de Windows.
- 2 En el árbol de directorios virtuales, seleccionamos el nodo que corresponde al directorio virtual de la aplicación en la que estamos interesados, y activamos su diálogo de propiedades.
- 3 En la primera página de ese diálogo (*Directorio virtual*), en la parte inferior de la misma, hay un combo titulado *Protección de la aplicación*, que por omisión debe mostrar el valor *Media (agrupada)*. Hay que cambiar su valor a *Alta (aislada)*, y cerrar el cuadro de diálogo.

Tenga en cuenta que estoy describiendo la técnica sobre WinXP Pro, pero no hay grandes diferencias respecto a Windows 2000. El siguiente paso nos llevará a COM+:

- 4 Abra la aplicación *Servicios de componentes*, en el mismo menú de *Herramientas administrativas*, o en el Panel de Control.
- 5 Al aumentar el nivel de aislamiento del directorio ISAPI, hemos forzado a Windows para que cree o registre una "aplicación" dentro del catálogo de COM+. Debe localizarla en la siguiente rama del catálogo:

```
Raíz de consola // 0
Servicios de componentes // 1
Equipos // 2
```

APLICACIONES ISAPI: MANTENIMIENTO

Otra de las pesadillas asociadas a las aplicaciones ISAPI es su mantenimiento y actualización, cuando finalmente las instalamos en un servidor "real". Una de las ventajas de utilizar ISAPIs es que IIS carga el código en memoria, para acelerar la generación

```
Mi PC // 3
Aplicaciones COM+ // 4
```

6 En mi caso, la aplicación creada automáticamente se llama *IIS-{Sitio Web predeterminado}/Root/classique*. En general, debe ser fácil identificarla.

7 Active su diálogo de propiedades. En la primera página (*General*) encontrará una etiqueta con el texto *Id. de aplicación*, y un GUID a continuación, como el siguiente:

```
{C6309979-38A1-4FE7-A24D-032E76BCE6CF}
```

8 ¡Anote ese número!

9 A continuación, seleccione la tercera página: *Identidad*, que indica la cuenta de usuario bajo la cual se ejecutará la aplicación. Por omisión, Windows utilizará una cuenta concreta predefinida, que corresponde al usuario virtual de IIS. Tenemos que cambiarla a *Usuario interactivo*, dentro de un grupo de opciones titulado *Cuenta del sistema*.

Los últimos pasos los daremos dentro de Delphi:

10 Dentro de Delphi, cargue el proyecto ISAPI, y ejecute el comando de menú *Run/Parameters*.

11 Dentro de este diálogo, seleccione el cuadro de edición *Host application*, y seleccione, con la ayuda del botón *Browse*, el siguiente fichero ejecutable:

```
c:\windows\system32\dllhost.exe
```

Esta aplicación es la verdadera anfitriona de las aplicaciones COM+. Recuerde que estoy utilizando WinXP, y que el directorio de sistema en un Windows 2000 será ligeramente diferente.

12 Por último, hay que indicar a *dllhost.exe* cuál es la aplicación que debe cargar para que nosotros la depuremos. Esto se hace modificando el cuadro de edición *Parameters*, en el mismo cuadro de diálogo, con el siguiente valor:

```
/ProcessID:{C6309979-38A1-4FE7-A24D-032E76BCE6CF}
```

Por supuesto, el GUID concreto que debe teclear como identificador de proceso es el que ha anotado en el paso 8.

Una vez que hayamos completado todos estos pasos, podemos pulsar F9. La aplicación que se ejecutará será *dllhost.exe*, que no mostrará nada en pantalla. Sin embargo, si abre la consola de los Servicios de Componentes, podrá comprobar que la aplicación COM+ correspondiente se ha activado, y que el icono asociado debe estar girando. Podemos entonces poner puntos de ruptura dentro del código fuente de nuestra DLL, como si se tratase de una aplicación normal. Claro, para alcanzar uno de esos puntos, necesitaremos realizar peticiones HTTP a la extensión ISAPI. Para ello, abra su navegador de Internet y teclee la URL local de acceso a la aplicación. Verá entonces que...

... bueno, no hay nada perfecto en este mundo. Mis primeras pruebas con la técnica descrita fallaron estrepitosamente: o no se alcanzaban los puntos de ruptura, o el navegador recibía un mensaje de error proveniente de IIS. Sólo pude hacerla funcionar a golpe de prueba y error. Aunque sigo sin comprender completamente la causa de mis dificultades, parece que la culpa la tiene el hecho de que estemos depurando una DLL con múltiples hilos. Para "resolver" el problema, abrí la ventana de hilos del depurador de Delphi, mediante el comando de menú *View/Debug windows/Threads*. Al hacerlo, de forma mágica, Delphi fue capaz de interceptar el hilo concreto que estaba tratando la petición del navegador, y a partir de ese momento, todo fue de maravillas.

de respuestas. Como efecto secundario, no podemos sobrescribir el fichero DLL mientras esté en uso por IIS. Eso sí, podemos "descargar" la DLL desde el diálogo de propiedades del directorio virtual... siempre que seamos más rápidos que nuestros usuarios,

porque la siguiente petición volverá a cargar en memoria la DLL. Y siempre que tengamos el permiso o la posibilidad de acceder a la herramienta de administración de IIS. Si estamos utilizando un servidor de Internet "externo", ubicado fuera de la empresa, no se puede ni siquiera soñar con este pedazo de lujo. Ya es bastante que nuestro ISP haya condescendido generosamente a que instalemos nuestras "peligrosas" DLLs en su precioso servidor (aunque nos permita quitar y poner aplicaciones ASP que presenten los mismos problemas y para colmo consumen más recursos).

Nuevamente, la solución está en la propia Internet, en la "eggcéntrica" página de William Egge (www.eggcentric.com). Este señor ha encontrado una elegante salida al problema: una DLL ISAPI "universal", cuya función es cargar la verdadera DLL programada por nosotros en memoria, y vigilar los cambios que podamos hacer sobre la misma. La idea es sencilla... después de que alguien la haya inventado, por supuesto:

- 1 William Egge ofrece el código fuente (shareware) de una aplicación ISAPI, o el proyecto ya compilado.
- 2 Debemos cambiar el nombre de la DLL compilada para que coincida con el de la nuestra. Por ejemplo, si mi DLL se llama

classique.dll, debo renombrar la DLL de William con ese nombre.

- 3 ¿Y qué pasa con nuestra propia DLL? Pues que tenemos que modificar su extensión, para que sea *.run*. Es decir, que mi DLL pasaría a llamarse *classique.run*.
- 4 Hay que situar ambos ficheros en el mismo directorio virtual. Cuando IIS reciba una petición dirigida a nuestra ISAPI, la interceptará la DLL impostora. Esta buscará nuestra DLL, con extensión *.run*, la cargará en memoria y delegará la respuesta en ella.
- 5 Periódicamente (por omisión, cada 10 segundos) la DLL impostora comprobará si existe un fichero con el mismo nombre, pero con extensión *.update*. Volviendo a nuestro ejemplo, se comprueba si hemos copiado un fichero llamado *classique.update* al directorio ISAPI.
- 6 Si encuentra tal fichero, espera a que terminen las peticiones en curso, y pone en espera las nuevas peticiones que puedan arribar mientras tanto. Una vez que no hay respuestas en curso, se descarga la DLL original, se renombran los ficheros y se carga la nueva versión de la aplicación.

COMPONENTES, RECOMENDACIONES, RUMORES...

Ya no es un secreto: **Borland** ha anunciado, oficial y públicamente, que está preparando un entorno de desarrollo parecido al Visual Studio.NET desde el que podrá programarse inicialmente en **C#**, y posteriormente en **Delphi.NET**.

La noticia ha caído como un bombazo, y muchos programadores han manifestado su preocupación sobre el futuro de Delphi. Para tranquilidad de todos, no creo que Delphi esté amenazado de forma alguna, al menos a mediano plazo. Hay que darse cuenta de algo muy importante: la plataforma .NET es una buena idea, pero por el momento no deja de ser un experimento, a estas alturas más comercial que técnico (porque la viabilidad técnica está sobradamente asegurada). ¿Cree usted que, de repente, todos los contratos de desarrollo van a exigir la compatibilidad con .NET? Por amor de Dios, si es que el proyecto entregado por IntSight la semana pasada ha terminado ejecutándose isobre Windows NT4!

En mi humilde opinión, durante un tiempo bastante largo, coexistirán el desarrollo para el API nativo de Windows y para la nueva plataforma, al menos mientras este API de Windows siga estando disponible. La verdadera amenaza se cierne sobre el mundillo de la programación en Java, que sí sufrirá la competencia de .NET. Y si me guío por las pruebas preliminares, Java lo tendrá muy difícil para enfrentarse a lo que se avecina.

En IntSight seguimos trabajando en varios frentes. En primer lugar, vamos a incorporar C# a nuestra oferta de cursos. Y he comenzado a escribir un par de libros para este lenguaje (porque no quiero que ni me hablen sobre Visual Basic.NET y el resto de la cuadrilla). Digo "un par de libros" porque estoy escribiendo uno sobre el lenguaje, las técnicas de programación orientada a objetos, y las clases básicas, en plan introductorio. Y en un segundo libro, más parecido a las "caras ocultas", me ocupo de la programación para bases de datos con **ADO.NET** y la programación para Internet con **ASP.NET**. Por supuesto, cuando salga Delphi 8 tendrá su correspondiente Cara Oculta... a no ser que se repita la decepción provocada por Delphi 7.

Estamos también en mitad de varios proyectos de desarrollo. Estamos a punto de cerrar la segunda versión de **Classique**, nuestro sistema de comercio electrónico, en la que hemos utilizado la segunda revisión de **Sonata**, nuestra extensión particular de WebBroker. Me ha sorprendido lo fácil que ha sido terminar la aplicación de compras en Internet, a pesar de que la funcionalidad del nuevo sistema se ha ampliado considerablemente. Para comprobar esta impresión, he reescrito el software de autorización de descargas que utilizamos para los cursos a distancia, que

en estos momentos está desarrollado con WebSnap. Y efectivamente, ha sido una agradable sorpresa ver cuánto se simplifica el desarrollo cuando se utiliza un sistema del tipo de Sonata. Como resultado, estamos pensando cómo amplia, mejorar o simplificar las características de Sonata para que realmente pueda ser utilizado fuera de **Intuitive Sight**.

Donde sí hemos tropezado con más obstáculos ha sido al desarrollar la aplicación GUI de gestión de tiendas en Internet, principalmente por lo obsoletos que se han vuelto muchos de los controles "de serie" de Delphi. La rejilla de datos, por mencionar un ejemplo, fue un control muy bueno en su momento inicial, pero a estas alturas se echan de menos algunas características de otros controles comerciales... y los *bugs* comienzan a hacerse notar.

Tras ensayar y descartar varias rejillas de terceros (en algunos casos encontramos errores y carencias, y en la mayoría, la interacción del usuario con las mismas era complicada y confusa) decidimos lanzarnos de cabeza al río, y crear nuestro propio control.

Como en este proyecto no existen grandes demandas de edición sobre la rejilla, estoy encapsulando el control *list view* nativo de Microsoft (sin pasar por el *TListView* de Delphi), en su variante "virtual". En mi opinión, el resultado estético es muy superior, y lo mismo sucede con el lado funcional. Menciono este asunto porque me lo estoy pasando de miedo viendo las rarezas del estilo de programación de Microsoft, y de la propia Borland. Por ejemplo, no sé si se habrá fijado en que la primera columna de un *TListView* siempre va alineada a la izquierda, y que no se puede forzar su alineación a la derecha. Esto no es un capricho de Borland, sino una restricción del control original de Microsoft.

Sin embargo, existe una forma de superar la limitación: crear una primera columna de mentirijillas, y eliminarla al terminar la creación de las restantes. No obstante, no lo intente directamente con el *TListView* encapsulado por Borland, porque la forma en que se ha implementado el mantenimiento de la información de columnas hace imposible el truco.

¿Quiere más? Abra la unidad `ComCtrls`, del código fuente de la VCL, y busque la línea 12.382 (Delphi 7; la 12.187 en Delphi 6.02):

```
for I:=0 to Collection.Count-1 do
  if TListColumn(Collection.Items[I]).
    WidthType<=ColumnTextWidth then Break;
  Changed(I<>Collection.Count);
```

En mi época de profesor de Pascal, esta metedura de pata podía haberle costado un suspenso a alguien. El disparate consiste en utilizar la variable de control del bucle **for**, en este caso *I*, fuera del ámbito de la propia instrucción. Una de las reglas sagradas, primero de Pascal, y luego de Delphi, es que al terminar un bucle **for**, la variable de control no tiene un valor definido; el compilador puede haber generado el código que haya considerado más eficiente. En palabras de la propia Borland, en la referencia del lenguaje:

... After the **for** statement terminates (provided this was not forced by a break or an exit procedure), the value of counter is undefined...

Es cierto que hay un **break** en el ejemplo. Pero el programador considera la posibilidad de que nunca se llegue a ejecutar. ¡Por ese motivo es que compara el valor de *I*, al terminar el bucle, con el número de elementos en la colección! La suerte del programador ha sido que, en este caso particular, el compilador se ha comportado de forma predecible, y no se ha atrevido a modificar de manera sustancial el algoritmo de recorrido del vector. A veces el crimen *sí* paga...

INTRAWEB

Una de las (pocas) novedades de Delphi 7 es IntraWeb, un producto desarrollado por **AtoZed** (www.atozedsoftware.com). El principal objetivo de diseño de IntraWeb es que la programación para Internet sea similar a la programación con Delphi de aplicaciones GUI. Es decir: que baste con colocar componentes sobre una superficie de diseño, modificar propiedades e interceptar algún que otro evento. ¿Logra IntraWeb este objetivo? Veremos que sí: es mucho más fácil programar con IntraWeb que WebBroker o WebSnap. Sin embargo, cuando se trata de aplicaciones para Internet, hay una gran diversidad de necesidades, y en consecuencia, hay muchos estilos de programación válidos. Voy a dar mis impresiones sobre algunas de las características de IntraWeb, y a qué tipo de aplicaciones beneficia o perjudica cada una de ellas.

- **No necesita cookies**

WebSnap utiliza *cookies* para dar continuidad a la secuencia de peticiones de un mismo usuario. Si el usuario de la aplicación ha desactivado las *cookies* en su navegador, adiós WebSnap. Supongamos que usted tiene un comercio en Internet. ¿Obligaré a sus clientes potenciales para que activen las *cookies*... si es que quieren comprar en su página?

IntraWeb, por el contrario, no las necesita. Por omisión, el identificador de sesión, que es el valor que conecta las peticiones entre sí, se transmite como parte de la URL de la petición. Este es un ejemplo de URL de IntraWeb (en modo de depuración, por cierto):

```
http://127.0.0.1:4956/EXEC/  
/6/2CFAA80007485C0FDE6AE240
```

Como ve, al final de la URL, como parte del *path info*, o información de ruta, se incluye un valor generado aleatoriamente en el lado servidor, que se mantendrá constante durante la sesión. Como curiosidad, observe que antes de ese valor hay un número "pequeño". IntraWeb lo utiliza para indicar la posición de esta página en la secuencia de peticiones, y así evitar que un usuario tramposo intente enviar dos veces los datos de una misma página, o trate de recuperar de la caché del navegador una página generada por la petición de otro usuario.

- **Continuidad de sesiones**

Casi todas las aplicaciones para Internet deben implementar un sistema para agrupar las peticiones de una sesión. Una vez resuelto este problema, con mayor o menor éxito, se suele añadir un mecanismo basado en el anterior para asociar variables de estado a la sesión. En ASP y en WebSnap, podemos utilizar una cadena con un nombre de variable como "índice" dentro de un objeto global llamado *Session*, como si se tratase de un vector asociativo. Naturalmente, el valor obtenido es de tipo *Variant*.

En cambio, IntraWeb es mucho más potente y seguro. En cada aplicación IntraWeb se define una clase, llamada *TUserSession*. Para cada sesión se crea una instancia de esta clase. Si declaramos una variable dentro de la clase, obtendremos una copia independiente de la misma para cada sesión de usuario. No hace falta utilizar variantes: si necesita un valor entero, declare un *Integer*.

También se pueden crear módulos de datos, que "persisten" de forma automática en el lado servidor. Sin embargo, hay que tener mucho cuidado con esta característica. ¿Qué pasaría si las peticiones dejasen conexiones abiertas, o conjuntos de datos sin cerrar al terminar? El modelo de acceso a datos y de módulos de WebBroker y WebSnap nos obliga a programar con cautela, pero el resultado son aplicaciones que se pueden cambiar de escala sin

mayor problema. En cambio, con IntraWeb podemos obtener una aplicación que funcione bien... sólo en condiciones de poca carga de usuarios.

Por supuesto, es posible superar esta dificultad. Es irónico que sea ADO Express quien menos trabajo nos dé con IntraWeb, gracias a su sistema de caché de conexiones.

- **Diseño sobre formularios**

Para diseñar una página, se crea un formulario derivado de la clase *TIWAppForm*, y se colocan sobre su superficie los controles especializados de la paleta de IntraWeb. Así de sencillo. Claro, hay un motor muy bien sintonizado detrás que transforma estos componentes en código HTML, mezclado con JavaScript. De esta manera, en IntraWeb, un formulario es igual a una página... y viceversa.

¿Y cómo se tratan los saltos a otras páginas de la misma aplicación? Por mucho que intentemos disfrazar una aplicación HTML, hay un límite en lo que podemos (y merece la pena) simular. En HTML no existen diálogos modales, ni aplicaciones SDI o MDI, ni pila de llamadas. Lo que sucede es que IntraWeb mantiene una pila con los formularios activos de una aplicación, y sólo el formulario que se encuentra en el tope de la pila es visible.

Cuando se aplica el método *Show* sobre un formulario existente, IntraWeb lo mueve al tope de la pila, y cuando se ejecuta *Hide*, se envía al final de la misma, mientras que el segundo formulario de la pila se hace visible. Por supuesto, esta estructura de formularios solamente existe en el lado servidor. En el navegador, el cliente solamente recibe el código HTML de la página que ha solicitado.

- **Plantillas externas**

En mi opinión, si un sistema de desarrollo de aplicaciones para Internet exige el trabajo de un programador para cambios estéticos en una página, pierde muchos puntos en mi valoración. ¿Qué tal se comporta IntraWeb en este sentido? A pesar de que la mayoría de los ejemplos no lo dan a entender, IntraWeb SI permite el uso de plantillas externas para establecer el formato de cada una de sus páginas. De hecho, existe un ejemplo en Delphi 7 (*Phonetics Customer Profiler*), que demuestra cómo se puede utilizar un componente *TIWTemplateProcessorHTML* para este propósito.

Sin embargo, y hasta donde puedo ver, sigue siendo necesario abrir la aplicación para añadir una nueva página... quiero decir, un nuevo formulario, al proyecto. Puede que se pueda "forzar" algún truco para ello, quizás mediante el uso de *plug-ins* o formularios residentes en DLLs, pero de entrada no he encontrado nada parecido. También es rígido el flujo de páginas, porque se establece en el código fuente, pero si se resolviese el problema anterior, éste también desaparecería. Hay que darle un poco de tiempo a IntraWeb...

- **JavaScript en el lado cliente**

El único problema que me preocupa seriamente es que IntraWeb debe generar código muy optimizado para mantener el diseño de página en los distintos navegadores en funcionamiento. Las extensiones a HTML, e incluso las características de JavaScript, de los navegadores actuales difieren mucho entre sí. Como consecuencia, IntraWeb debe detectar qué tipo de navegador está utilizando cada petición para generar y enviar el código apropiado. En estos momentos, IntraWeb soporta Internet Explorer, Netscape Navigator y Opera; de los dos últimos mencionados, sólo las versiones más recientes (Netscape Navigator ha sido histórica-

mente un desastre, y sólo las últimas versiones han logrado cierta estabilidad y compatibilidad, no con Microsoft, sino con los estándares).

- **Conclusiones**

IntraWeb representa un esfuerzo de desarrollo impresionante, que afortunadamente ya se encuentra en su sexta versión; quiero decir, que no vamos a ser los conejillos de Indias del fabricante, a diferencia de lo que nos ha sucedido a muchos con WebSnap, DB Express y tantas otras "tecnologías punteras". Considero que es una opción muy a tener en cuenta antes de iniciar un nuevo proyecto de desarrollo para Internet. Como comprenderá, en este breve resumen he dejado fuera muchas características importan-

SERVICIOS ENDEMONIADOS

Esta aplicación de servicio, en particular, vigila la aparición de registros en una tabla llamada Mensajes, para componer una serie de mensajes de correo electrónico y enviarlos a través de un servidor SMTP que le indiquemos en su configuración.

La aplicación forma parte de nuestro sistema de comercio electrónico, Classique, por lo que es perfectamente inútil... a no ser que usted duplique la estructura de la base de datos de Classique, lo que sería excesivo. Por consiguiente, en vez de adjuntar un código fuente que sería de difícil manejo y actualización, he preferido detallar los pasos para su creación. En vez de regalar pescado, es mejor enseñar a pescar, ¿no?

1 Crear una aplicación de servicio

Para ello, vamos al *Object Repository* de Delphi, y en la primera página hacemos doble clic sobre el icono *Service Application*. Delphi crea un nuevo proyecto ejecutable, y un módulo de datos derivado de la clase *TService*, definida en la unidad *SvcMgr*. Cambiamos el nombre al módulo, para que sea *TMailDemon*, y declaramos una variable en la sección privada de la clase:

```
type
  TMailDemon = class(TService)
  private
    FDemonThread: TDemonThread;
  end;
```

Con *TDemonThread* estamos haciendo referencia a una clase derivada de *TThread*, que implementará un hilo adicional para el servicio y que definiremos más adelante.

¿Para qué necesitamos un hilo adicional? En realidad, Delphi lanza un hilo independiente para cada servicio que definamos dentro de una aplicación de servicio, y aparentemente, esto sería suficiente. Pero ese hilo "principal" tiene una misión: implementar un bucle de mensajes para responder a los mensajes de control que le enviará el sistema operativo al servicio. Estos mensajes de control son los que iniciarán, pausarán, reanudarán y detendrán el servicio. En principio, podríamos utilizar ese mismo hilo para implementar el algoritmo que consulta la base de datos y envía correos electrónicos, pero...

...pero el nuestro es un hilo algo especial: no queremos estar vigilando continuamente la base de datos, porque podríamos gastar demasiado tiempo de CPU en esta tarea. De modo que el algoritmo de consulta de registros y generación de mensajes de correo debe utilizar algún sistema para activarse periódicamente. Podríamos intentarlo con un *timer*, pero he preferido utilizar objetos de *eventos* del sistema operativo (más detalles, enseguida), que entre otras cosas, van a bloquear el hilo en cuestión y a despertarlo cada cierto tiempo. Una vez que hemos tomado esta decisión de diseño, está claro que no podemos utilizar el hilo principal para este propósito, bajo pena de que el servicio no responda fluidamente a los mensajes de control del sistema operativo.

tes, como la forma de depuración y las distintas formas de instalación y explotación. Es cierto que existen algunos problemas, como sucede con todo software. En este caso, los más importantes son la compatibilidad con los navegadores, el mayor o menor dinamismo del conjunto de páginas y el cuidado que hay que poner en la configuración del acceso a datos. Pero todos ellos pueden resolverse con un poco de dedicación y pequeñas dosis de sentido común.

En la página de **AToZed** encontrará más información sobre licencias, preguntas técnicas y sus respuestas, y enlaces a foros de usuarios sobre esta herramienta. Existe también una actualización gratuita para la versión incluida en Delphi 7 que, previo registro, podrá descargar desde allí.

2 Implementar el bucle de mensajes

El mencionado bucle de mensajes se implementa como respuesta al evento *OnExecute* del módulo de servicio, y el código necesario es muy sencillo:

```
procedure TMailDemon.ServiceExecute(
  Sender: TService);
begin
  while not Terminated do
    ServiceThread.ProcessRequests(True);
end;
```

Perdóneme que insista, pero debo recalcar que también podríamos implementar nuestro servicio mediante un temporizador. No lo he hecho simplemente porque he considerado que la técnica que mostraré es más segura e interesante (dentro de lo "interesante" que pueden llegar a ser estas cosas).

3 Tratar los mensajes de control

Los mensajes de control que recibe el servicio son automáticamente dirigidos a cuatro eventos del módulo, que serán interceptados en la siguiente manera:

```
procedure TMailDemon.ServiceStart(
  Sender: TService;
  var Started: Boolean);
begin
  FDemonThread :=
    TDemonThread.Create(Self);
  Started := True;
end;

procedure TMailDemon.ServicePause(
  Sender: TService;
  var Paused: Boolean);
begin
  FDemonThread.Suspend;
  Paused := True;
end;

procedure TMailDemon.ServiceContinue(
  Sender: TService;
  var Continued: Boolean);
begin
  FDemonThread.Resume;
  Continued := True;
end;

procedure TMailDemon.ServiceStop(
  Sender: TService;
  var Stopped: Boolean);
begin
  FDemonThread.Signal;
  FDemonThread := nil;
  Stopped := True;
end;
```

Como puede comprobar, nos limitamos a delegar las acciones al hilo paralelo. Los métodos *Suspend* y *Resume* son métodos ya definidos en la clase *TThread*. Observe cómo el hilo se crea en la respuesta a *OnStart*, y se destruye en la respuesta a *OnStop*.

Por último, el método `Signal` será implementado por nosotros mismos, en la clase `TDemonThread`.

4 Definir un hilo paralelo

La clase `TDemonThread` hará referencia a la clase del módulo, `TMailDemon...` que a su vez hace referencia a la clase de hilo. Como se trata de una referencia circular, debemos implementar ambas clases en una misma unidad. Esta es la declaración de `TDemonThread`:

```
type
  TDemonThread = class(TThread)
  protected
    FEventHandle: THandle;
    FInterval: Cardinal;
    FOwner: TMailDemon;
  public
    constructor Create(
      AOwner: TMailDemon);
    destructor Destroy; override;

    procedure Execute; override;
    procedure Signal;
  end;
```

Como ve, hemos declarado un constructor, hemos redefinido el destructor y el método virtual `Execute`, y hemos añadido un método llamado `Signal`. Comencemos por el constructor:

```
constructor TDemonThread.Create(
  AOwner: TMailDemon);
begin
  inherited Create(True);
  FOwner := AOwner;
  FEventHandle := CreateEvent(
    nil, False, False, nil);
  FInterval := DEFAULT_INTERVAL;
  FreeOnTerminate := True;
  Resume;
end;

destructor TDemonThread.Destroy;
begin
  CloseHandle(FEventHandle);
  FEventHandle := 0;
  inherited Destroy;
end;
```

La llamada al constructor heredado pasa `True` como parámetro: eso significa que el hilo se va a crear en estado suspendido, para que nos dé tiempo a configurar sus variables de instancia. Almacenamos la referencia al módulo de servicio que nos pasan como parámetro; luego explicaré para qué la necesitamos. También creamos el famoso objeto de *evento*, asignamos un valor predefinido para el intervalo entre activaciones del demonio, indicamos que el hilo debe destruirse automáticamente al terminar (`FreeOnTerminate`), e iniciamos el funcionamiento del mismo llamando a su método `Resume`.

¿Y el evento, de qué va? Un evento es un objeto implementado por Windows, que forma parte del API de sincronización de hilos de este sistema operativo. Funciona como un interruptor eléctrico, pero en vez de detener el flujo de electrones, puede bloquear la ejecución de uno o más hilos. Este objeto de evento es destruido por el destructor de la clase de hilo.

Veamos ahora cómo se produce el bloqueo del hilo mediante el evento:

```
procedure TDemonThread.Execute;
begin
  CoInitialize(nil);
  try
    while not Terminated and
      (WaitForSingleObject(FEventHandle,
        FInterval) = WAIT_TIMEOUT) and
      not Terminated do
      FOwner.ProcessEMails;
  finally
    CoUninitialize;
  end;
end;
```

La responsabilidad del bloqueo corresponde a la llamada al procedimiento `WaitForSingleObject`, del API de Windows. Hemos creado el evento en el estado que deniega la petición del hilo, pasando `False` en el tercer parámetro de `CreateEvent`. Por lo tanto, cuando el hilo llama a `WaitForSingleObject`, queda bloqueado hasta que "alguien" ponga el evento en el estado que permite continuar la ejecución... o hasta que el hilo se harte y decida continuar por su cuenta. Esto se logra pasando un valor positivo en el segundo parámetro de `WaitForSingleObject`, para indicar un tiempo máximo de espera. Normalmente, la llamada a este procedimiento terminará por agotamiento del tiempo de espera, y el hilo seguirá ejecutando el bucle definido en `Execute`. Se enviará un grupo de mensajes de correo electrónico llamando al método `ProcessEMails` del módulo de servicio (¡para esto es que nos quedamos con la referencia al mismo!) y volverá a esperar por la señal de paso del evento. Este mecanismo es el que permite activar el envío de mensajes periódicamente.

Y usted se preguntará para qué he utilizado un sistema tan enrevesado. ¿No sería más simple bloquear el hilo durante un tiempo llamando al procedimiento `Sleep`, también del API de Windows? La respuesta es un rotundo NO. Piense en lo que puede suceder si activamos `ProcessEMails` cada 20 segundos y tenemos que detener el servicio. No podemos parar el hilo secundario "a lo bestia", porque podemos dañar la base de datos, o dejar la conexión a la misma en un estado inestable, o perder memoria y otros recursos. Por lo tanto, tendríamos que dejar que el hilo terminase por voluntad propia en el siguiente paso del bucle. En el peor de los casos, eso implicaría esperar 20 segundos, si es que el hilo acaba de llamar a `Sleep`. Si Windows no puede detener un servicio en un tiempo dado, considerará que la operación ha fallado.

Es cierto que no podemos salir de `Sleep` una vez iniciada la espera, pero sí que podemos terminar la espera provocada por `WaitForSingleObject`, si pasamos el objeto de evento al estado que permite la continuación del hilo. Esa es la función del método `Signal` que vamos a definir. Ya hemos visto cómo se utiliza `Signal` en la respuesta al evento `OnStop` del módulo de servicio. Ahora veremos su implementación:

```
procedure TDemonThread.Signal;
begin
  if not Suspended then
  begin
    Terminate;
    PulseEvent(FEventHandle);
  end;
end;
```

Por una parte, el método `Terminate`, heredado de `TThread`, asigna `True` en la propiedad `Terminated` del hilo. Con esto, ya estaríamos garantizando que el hilo terminará en la siguiente iteración. Para salir de la espera, ejecutamos el procedimiento `PulseEvent`, del API de Windows, pasando el objeto de evento como parámetro. `PulseEvent` *abre* o *dispara* el evento, para desbloquear los hilos pendientes del mismo, y a continuación vuelve a desactivar el evento.

5 Propiedades del servicio

Regresemos al servicio, porque tenemos que configurar un par de propiedades del mismo. En primer lugar, debemos darle un nombre "legible" al servicio, utilizando su propiedad `DisplayName`. En mi aplicación, por ejemplo, esta propiedad contiene la cadena "Classique v2.0 Mail Demon". Podríamos también modificar la propiedad `ServiceStartName`, para indicar a Windows en qué cuenta del sistema queremos que se ejecute el servicio. Si dejamos la propiedad en blanco, que es lo que sucede en mi caso, Windows utilizará la cuenta local del sistema. Por último, podemos también retocar `Dependencies`. Por ejemplo, mi servicio utiliza ADO para comunicarse con un servidor de SQL Server. Si se instala el servicio en el mismo ordenador donde se ejecuta localmente SQL Server, podemos añadir una dependencia, con el nombre de `MSSQLServer`, que es el nombre interno del servicio de SQL Server. Pero esto es sólo un detalle elegante...

6 ¿Y el envío de mensajes?

Para el envío de mensajes utilicé directamente los componentes Indy, que se incluyen en Delphi 6 y 7. Los detalles son sencillos,

DESCONEXIÓN EN ADO

Analice la implementación del método `ProcessEMails`, de la clase del servicio (he omitido algunos detalles, para simplificar):

```
procedure TClMailDemon.ProcessEMails;
begin
  try
    Messages.Open;
    try
      Messages.Recordset.
        Set_ActiveConnection(nil);
      ADOConn.Close;
      if not Messages.IsEmpty then
        begin
          SMTP.Connect;
          try
            Messages.First;
            while not Messages.Eof do
              begin
                ProcessEMail;
                Messages.Next;
              end;
            finally
              SMTP.Disconnect;
            end;
          end;
        finally
          Messages.Close;
        end;
      except
        end;
    end;
  end;
end;
```

`Messages` es un componente de tipo `TADOQuery`, conectado mediante un componente de conexión llamado `ADOConn`. Su instrucción SQL es la siguiente:

```
set rowcount :limit
select m.*
from   dbo.MESSAGES m
where  m.Active = 1
order by Created desc
set rowcount 0
```

Probablemente usted ya sabe que podemos limitar el número de registros devuelto por una consulta mediante una cláusula **top** en la cláusula **select**. Sin embargo, para esta consulta he utilizado la variable **rowcount** de Transact SQL, que data de la época de SQL Server 6. ¿Nostalgia?, ¡no! El problema es que con la cláusula **top** no podemos pasar el número de filas como parámetro de la consulta. En cambio, con **rowcount** sí podemos utilizar un parámetro para esta importante restricción.

LA ESQUINA DE LAS MALAS LENGUAS...

José Rodríguez, desde Venezuela, me escribe para recomendarme un producto: un proveedor de datos para FireBird... que funciona en la plataforma .NET. Tiene muy buena pinta, pero no he tenido tiempo para evaluarlo. Según me comenta, el autor es Carlos Guzmán Alvarez, de Vigo. El enlace para la descarga está en la siguiente URL:

```
http://www.ibphoenix.com/main.nfs?
a=ibphoenix&page=ibp_download#NET
```

Hablando sobre C#, intentaré en las próximas semanas habilitar un formulario o encuesta en mi página para recoger opiniones y sugerencias sobre C#. Por ejemplo, tengo material suficiente para dos libros: uno de programación orientada a objetos, más centrado sobre el lenguaje, y otro al estilo de La Cara Oculta, que cubra principalmente ADO.NET y ASP.NET. Pero me gustaría conocer otros puntos de vista sobre la distribución de temas, estilo de presentación y ese tipo de cosas.

aunque la explicación es larga, por lo que voy a omitirla. No obstante, lea el siguiente truco, que está también relacionado con esta aplicación.

Pero la parte interesante del método está aún por llegar. Abrimos la consulta, con lo que se activa la conexión a la base de datos, si es que estaba inactiva. Al hacerlo, obtenemos un grupo de registros, a partir de los cuales debemos generar mensajes de correo y enviarlos. ¿Cuánto tiempo puede consumir esta operación? No me atrevería a adivinar, porque en estas cosas es imprescindible experimentar. Entonces adoptaremos la peor hipótesis; supondremos que la operación puede consumir bastante tiempo. ¿Podemos dejar activa la conexión a la base de datos durante todo ese tiempo? Sería arriesgado, ¿verdad? Por este motivo es que ejecutamos las siguientes instrucciones:

```
Messages.Recordset.
  Set_ActiveConnection(nil);
ADOConn.Close;
```

El método `Set_ActiveConnection` pertenece al API de bajo nivel de ADO; note que lo aplicamos a través del puntero de interfaz que el componente de consulta almacena en su propiedad `Recordset`. Al ejecutarlo con el parámetro `nil`, estamos utilizando un truco común entre los desarrolladores que utilizan ASP. Esta llamada "desconecta" el resultado de la consulta de la base de datos. Esto es, los registros se almacenan en una caché local gestionada por ADO en el lado cliente. A partir de ese momento, podemos incluso cerrar la conexión, como hacemos en el ejemplo, que no se pierden los registros que hemos traído a la parte cliente. La técnica es más o menos similar a lo que sucede cuando traemos registros a un componente `TClientDataSet`, para cortar luego la conexión a la base de datos.

¿Y no será más costoso establecer nuevamente la conexión en el siguiente paso del bucle? Recuerde que estamos utilizando ADO, por lo que la conexión cerrada en realidad irá a parar al *pool* o depósito de conexiones, para ser reutilizada en el momento en que volvamos a activar la conexión.

El resto del código es sencillo. Si hay mensajes para enviar, se establece una conexión TCP con un servidor SMTP. Esto es responsabilidad del componente SMTP, de la clase `TIdSMTP` de Indy. Luego, se recorre la consulta, y para cada registro se ejecuta el método `ProcessEMail`, que configura un objeto de tipo `TIdMessage` a partir de los campos de la consulta, para enviarlo por medio de SMTP:

```
SMTP.Send(IDMessage);
```

Por último, hay una secuela del artículo del boletín anterior sobre los cursores de ADO. En el número de diciembre del 2001 de la **Delphi Magazine**, Martin Lanza publicó un artículo comparando la velocidad de acceso a una tabla de Access, utilizando llamadas directas a ADO, utilizando variantes y de forma directa, y a través de los componentes de ADO Express. El resultado era preocupante, porque ADO Express quedaba mal parado. En aquel momento sospeché que el problema estaba en el tipo de cursor utilizado... pero como no sabía aún que al no configurar el cursor como sólo lectura se disminuía la velocidad de acceso, pensé que la causa estaba en otro lugar. Lo que quiero confirmar es que, efectivamente, al utilizar cursores unidireccionales, en el lado servidor y de sólo lectura, la velocidad de acceso desde Delphi volvía a las cifras esperadas: un tiempo de acceso igual al necesitado por el acceso directo mediante el API de ADO.

PÉRDIDAS DE MEMORIA CON DLLS

Tendréis que disculparme: voy a explicar un problema relacionado con la carga dinámica de DLLs en Delphi... y no voy a incluir el código que lo resuelve. El código necesario es demasiado extenso, y la mejor solución, en mi opinión, es la que ofrece Roy Nelson en un artículo de mi siempre recomendada **Delphi Magazine** (número 75, Noviembre 2001). No obstante, el fallo que explicaré, aunque sólo aparece en circunstancias muy específicas y poco frecuentes, merece ser tenido en cuenta... porque en esas condiciones puede producir una pérdida de memoria importante.

Para que aparezca el problema deben darse las siguientes condiciones:

- 1 El fallo lo producen las DLLs escritas en Delphi.
- 2 La DLL debe crear objetos derivados de `TwinControl`, pero es suficiente que incluya la unidad `Controls` en alguna de sus cláusulas **uses**.
- 3 La DLL debe ser cargada y descargada dinámicamente por una aplicación, o por otra DLL.

Cuando se cumplen todos estos requisitos, se produce el fallo:

- 4 Cada vez que se carga y descarga la DLL, se pierden aproximadamente 4KB de memoria.

La raíz del mal está profundamente enterrada, en los mismísimos cimientos de la VCL. Quien haya trabajado alguna vez con el API de Windows de manejo de ventanas, al más bajo nivel, sabrá que cuando se crea una ventana o un control, hay que especificar una *clase* de ventana, y que esa *clase*, en el sentido que Windows le da a esa palabra, está asociada a un *procedimiento de ventana*: una función "normal", no un método. Cada vez que a la ventana llega un mensaje, es su procedimiento de ventana el encargado de ejecutar la respuesta apropiada. Sin embargo, sabemos que las clases (en el verdadero sentido del término) de la VCL manejan los mensajes que reciben los controles por medio de métodos. El paso que convierte una llamada al procedimiento de ventana en una llamada a método, de acuerdo a la programación orientada a objetos, es responsabilidad de un pequeño fragmento de código... ique se genera en tiempo de ejecución!

El encargado de generar el código que sirve de puente a las dos tecnologías es el procedimiento `MakeObjectInstance`, cuya historia se remonta a la época del primer Turbo Pascal para Windows. El procedimiento reserva una pequeña zona de memoria, modifica sus atributos para permitir la ejecución de código, y escribe en ella los códigos binarios en lenguaje nativo Intel para que pase el control de los mensajes de Windows a la tabla de mensajes de la VCL. Como cada control que se crea en Delphi necesita este sistema, se podría presentar un grave problema de fragmentación de memoria, si Borland no hubiese tenido cuidado. Para evitarlo, existe una lista de bloques enlazados, y cada uno de estos bloques ocupa, precisamente, 4KB: ¡la memoria que se pierde!

Como puede imaginar, el problema consiste en que basta que se cree un solo control para que la llamada a `MakeObjectInstance` reserve un bloque de 4KB. Y ese bloque, ¡no se libera

cuando se destruye el control asociado! El problema es que el bloque no es "propiedad" exclusiva del control.

Normalmente, esto no causa mayores problemas, porque la pérdida de memoria no es grande: si estamos dentro de un ejecutable, los bloques libres pueden ser reutilizados. Al terminar la ejecución, el sistema operativo se encarga automáticamente de recuperar toda la memoria utilizada por el ejecutable, incluyendo esos bloques, por supuesto. Una DLL que no utilice la unidad `Controls` tampoco tendría problemas, porque no se generarían esos bloques. ¿Y si es una DLL con enlace estático? Tampoco habría motivo de preocupación: la DLL se carga una sola vez, cuando comienza a ejecutarse la aplicación principal, y se descarga al terminar la aplicación.

Curiosamente, tampoco hay problemas con cargar y descargar *packages* dinámicos. En ese caso, el puntero a la lista de bloques libres es compartido por los restantes *packages* y por la propia aplicación. Lo que sí sería problemático es cargar y descargar una DLL que crease controles: en cada carga y descarga se perderían, como mínimo, los 4KB mencionados.

¿Qué soluciones existen? Al parecer, cualquier solución es complicada, por necesidad. Una DLL puede crear un control, y ser descargada sin que el control tenga necesariamente que desaparecer. Por lo tanto, el problema no se resuelve destruyendo la estructura de datos al "finalizar" la DLL.

He visto, además, varias propuestas de solución. En la Borland Community, en su Code Central, hay dos contribuciones con código incluido:

Dejan Maksimovic
codecentral.borland.com/codecentral/ccweb.exe/listing?id=16380
Dmitri Burov
codecentral.borland.com/codecentral/ccweb.exe/listing?id=18735

No he leído estos dos artículos por completo, pero hasta donde he llegado, lo que proponen son cambios en el código fuente de las unidades implicadas. Puede que este tipo de solución sea lo más sencillo, pero tiene un inconveniente: a partir de ese momento no podrá utilizar los *runtime packages*, que no pueden ser parcheados porque Borland no ofrece el código fuente necesario para reconstruirlos.

La solución de Roy Nelson es bastante enrevesada, porque se basa en suposiciones sobre el formato de los bloques de memoria reservados por `MakeObjectInstance`. Pero es probablemente la más fácil de aplicar, porque todo el código queda incluido en una unidad, que debe situarse en determinado orden en la cláusula **uses** del proyecto DLL. Hasta donde sé, Roy sólo ha publicado el código en la revista mencionada, y por evidente respeto a los derechos de copia, no debo reproducir el código aquí. De todos modos, creo que es bueno conocer que *existe* el problema, evitarlo siempre que esté en nuestras posibilidades... y si no puede esquivarlo, bueno, ya sabe donde puede encontrar una solución al mismo.

PROYECTO SNOWBALL

Snowball, por supuesto, es "bola de nieve" en inglés. Pero en este caso, se trata de uno de los personajes porcinos de *Animal Farm*, la divertida novela de Orwell. En realidad, hay tres cerdos revolucionarios: Napoleón, el tirano cruel que al final se hace con todo el control; Squealer, su servil ayudante, y Snowball, inicialmente uno de los fundadores del Animalismo, obsesionado con la electri-

cidad, que al final es perseguido y exilado por el dictatorial Napoleón. ¿A qué viene todo esto? Bueno, a que por una parte se acerca el Día de San Juan, "glorioso" aniversario de la Rebelión de la Granja, a que este año es el centenario de Orwell... pero principalmente, he recordado al trágico Snowball gracias al único Man-

damiento del Animalismo que logra sobrevivir a las muchas revisiones durante la larga dictadura napoleónica:

Todos los animales son iguales
pero algunos son más iguales que otros

Sustituya la palabra *animal* por *conjunto de datos*: ¿son iguales todos los conjuntos de datos? No, pero esa es la idea que Borland nos estuvo vendiendo desde la aparición del BDE: al acceder a distintos servicios de datos con una misma interfaz, podríamos crear aplicaciones a las que no le importasen el tipo de servidor utilizado. Crasa mentira, porque montones de detalles lo impiden: desde el formato de nombres de parámetros, pasando por las propias capacidades de cada servidor... hasta las diferentes clases de los objetos de acceso a columnas.

Se suponía que la aparición y posterior evolución de Midas, ahora llamado DataSnap, nos ayudaría a crear aplicaciones que trabajasen indistintamente con diferentes motores de datos (una vez que hayamos renunciado a incluir bases de datos de escritorio, por cierto). La compatibilidad se logra a un precio: podemos, teóricamente, usar una misma aplicación visual para acceder a servicios alternativos en la capa intermedia, específicos para el formato de los datos SQL que manejan. Es decir, tendríamos que seguir manteniendo proyectos paralelos para la capa intermedia, pero al menos, la capa de presentación estaría a cargo de una sola aplicación, que podría ignorar qué sistema de bases de datos se esconde al final de la cadena.

Pero en la práctica, de momento, no se cumple tal promesa. Parte de la culpa la tiene el nuevo DB Express: las clases tradicionales de campos, como `TBCDFIELD` y `TDATEFIELD` no ofrecían suficiente precisión, por lo que introdujo nuevas clases, como `TSQLTIMESTAMPFIELD` y `TFMTBCDFIELD`, que además de tener nombres largos y horribles, no son utilizadas por las interfaces alternativas de acceso a datos, como ADO Express. Si DB Express funcionase como debería funcionar, la solución sería *no* utilizar interfaces alternativas. Pero tal como están las cosas, si un programador no utiliza ADO Express para acceder a SQL Server, es porque está loco (o porque ha asistido a un curso de "deformación" impartido por mi presunta competencia).

Sin embargo, una parte de la culpa tiene raíces más profundas. SQL Server soporta columnas de tipo `bit`, que contienen exactamente un bit de información, y son el mejor modo de crear columnas de tipo "lógico". El tipo `bit` es muy eficiente, porque SQL Server agrupa todas las columnas de este tipo, de modo que ocho columnas `bit` pueden ocupar solamente un byte dentro de un registro (no se admiten nulos). Cuando Delphi crea un objeto de campo para manejar una de estas columnas, utiliza la clase `TBooleanField`. Hasta aquí, estupendo.

Por desgracia, Oracle no tiene un tipo de datos para representar columnas lógicas; tampoco InterBase, hasta la versión 7, e incluso con esta versión no podemos aún utilizar el nuevo tipo `boolean`, porque DB Express no lo reconoce (al parecer, la actualización de IB Express, sí, pero me entran escalofríos ante la perspectiva de tener que usarlo). ¿Debemos renunciar entonces al bit de SQL Server? Poder, podemos, pero deber... ni hablar. Suponga que decidimos "traducir" las columnas lógicas a Oracle utilizando el tipo `char`. El problema consiste en que Delphi se empeña entonces en crear objetos de tipo `TStringField` para las columnas lógicas. Y no conozco forma alguna de forzar a Delphi para que utilice `TBooleanField`, por compatibilidad con SQL Server.

Hace ya un tiempo, pensando en estos problemas, se me ocurrió una solución que creo que es bastante sencilla: ¿por qué no utilizar un *adaptador*, como los que se utilizan en las clavijas de toma de corriente, para transformar los tipos de campos? Para explicarme, comenzaré por definir el escenario: quiero una solución que sirve para desarrollar módulos de capa intermedia con interfaces, de cara a la capa de presentación, que sean completamente compatibles entre sí. Mi solución no valdrá para aplica-

ciones en dos capas, a no ser que estén internamente estructuradas en tres capas, como explico en los cursos de **ADO Express/DataSnap** y **DB Express/DataSnap**. Si conoce el modelo de programación de DataSnap, sabrá que en la capa intermedia se utilizan, preferentemente, conjuntos de datos unidireccionales de sólo lectura. Las restantes características que pueda implementar determinada colección de componentes de acceso a datos, son desaprovechadas, porque DataSnap las sustituye por su propia funcionalidad. Con estos requisitos más claro, ya puedo explicar mi propuesta:

- Definir una clase derivada directamente de `TDataSet`.
- Los conjuntos de datos de la nueva clase tendrán una propiedad llamada `DataSet`, que apuntará a otro objeto de la clase `TDataSet`. No importará si este segundo conjunto de datos pertenece a ADO Express, DB Express o a la clase concreta que sea. En cualquier caso, nuestro nuevo *dataset* extraerá sus datos de este conjunto de datos asociado, como un vampiro chupa la sangre de sus víctimas (o como un jefe se apodera de las ideas de sus subordinados y las vende como propias).
- La nueva clase tendrá una propiedad llamada `Mappings`, o algo parecido, que será una colección o lista de "traducciones" de tipos de campos. Un elemento de esta colección podría indicar, por ejemplo, que el campo `EscCliente` del conjunto de datos base, que es de tipo `char(1)`, debe traducirse al tipo `boolean`, considerando que su valor es `True` cuando contiene los valores reales 'S' o 's', y que es `False` en caso contrario.
- El ejemplo de traducción anterior es algo complicado, pero existen traducciones más sencillas: por ejemplo, todos los campos de la clase `TDateTimeField` del conjunto de datos subyacente, podrían transformarse en `TSQLTimeStampField`... o viceversa, en dependencia de cuál es la clase con la que prefiera tratar.
- En un módulo remoto típico, conectaríamos los proveedores (`TDataSetProvider`) a los conjuntos de datos de las nuevas clases, y estos últimos, a conjuntos de datos de las clases que realmente manejan datos SQL.
- Para evitar la proliferación de componentes, podría plantearse la posibilidad de mezclar la nueva clase de conjuntos de datos con los componentes `TDataSetProvider`, o por el contrario, crear clases adaptadoras específicas para DB Express, ADO Express, IB Express, etc.

En teoría, tendríamos otra posibilidad: en vez de crear clases adaptadoras para los conjuntos de datos, ¿por qué no crear directamente una nueva clase de proveedor? La respuesta es que el código fuente de `TDataSetProvider` es un excelente ejemplo de *cómo no se debe programar*. Hay características muy importantes del funcionamiento de los proveedores que están implementadas por objetos delegados sobre los que no tenemos control alguno. Tomemos como ejemplo el evento `AfterUpdateRecord`: en vez de ser disparado por un método virtual del propio `TDataSetProvider` o de alguno de sus ancestros, se dispara dentro de un bloque de código monolítico implementado dentro de la clase `TCustomResolver`. El individuo que lo programó así merecería un suspenso en Programación Orientada a Objetos en cualquier universidad...

Este será un proyecto freeware, con disposición total del código. El núcleo del componente es suficientemente pequeño e imbricado como para que sea preferible que lo desarrolle una sola persona. Pero voy a necesitar ayuda, si la cosa llega a buen puerto, sobre todo con las pruebas. Seguiremos informando...

ACUMULADORES DE CADENAS

Es muy importante, tanto para una aplicación en Internet como para un servidor de capa intermedia, que las respuestas a las peticiones que reciben se elaboren en el menor tiempo posible. El motivo principal no es, como puede parecer, que un usuario tenga que esperar menos tiempo, porque en ese caso, un milisegundo de más o de menos no tendría importancia. La verdadera urgencia de minimizar los tiempos de respuestas se debe a que, mientras más peticiones por segundo se puedan atender, será menor la cantidad de módulos o instancias necesarias para atender el mismo número de peticiones concurrentes. Y esto último significa un mejor aprovechamiento de los recursos del servidor.

Con esta idea en la cabeza, me puse hace un par de semanas a revisar el código fuente de Sonata, nuestra suite de componentes para el desarrollo en Internet. El objetivo: arañar cada milisegundo posible. La mina de oro que encontré: las funciones de manejo de cadenas. ¿Son tan malas las rutinas que ofrece Delphi? Realmente, no, hacen su trabajo decentemente... aunque sólo eso. Estas rutinas tienen que lidiar con varios problemas, y el principal es el formato de las cadenas "largas" que se introdujeron en Delphi 2. Estas cadenas, que son el núcleo de la implementación del actual Delphi, utilizan memoria dinámica para almacenar sus caracteres. Si no tenemos cuidado, podríamos tener problemas con el manejo de esa memoria en sistemas con más de un procesador... o con procesadores como el Pentium IV, si se activa la característica conocida como *hyperthreading*. Por lo tanto, Delphi se ve obligado a utilizar instrucciones de sincronización a nivel de hardware (LOCK) para evitar los conflictos de concurrencia. Claro, el precio que se paga es la eficiencia.

Como consecuencia, no es mala idea tomar el mando en los casos en que sabemos que Delphi no ofrece la solución óptima. Uno de estos casos es la búsqueda de un carácter dentro de una cadena. El programador descuidado, o el que tiene prisa, utiliza la conocida función Pos:

```
if Pos('/', S) <> 0 then
  // ...
```

Pero Pos tiene dos problemas. El primero: su primer argumento es una cadena. En el ejemplo anterior, estamos pasando una constante, que puede que Delphi interprete directamente como una cadena de longitud uno. Pero si pasamos el carácter a buscar dentro de una variable, Delphi tendrá que convertirlo a cadena (iuna petición de memoria dinámica!) en tiempo de ejecución. El segundo problema es una consecuencia del primero: internamente, Pos ejecutará un algoritmo de búsqueda general de cadenas. En estos casos, es preferible perder un poco de tiempo de desarrollo, y crear (¡o copiar!) una función especial de búsqueda de caracteres escrita en ensamblador:

```
function CharPos(const Source: string;
  C: Char): Integer; assembler;
asm
    PUSH    EDI
    TEST   EAX, EAX
    JZ     @@false
    MOV   EDI, EAX
    XCHG EAX, EDX
    MOV   ECX, [EDI-4]
    REPNE SCASB
    JNE  @@false
    MOV   EAX, EDI
    SUB  EAX, EDX
    POP  EDI
    RET
@@false:
    XOR  EAX, EAX
    POP  EDI
end;
```

Observe que la función utiliza internamente la instrucción SCASB, por lo que puede ser bastante rápida, en comparación con una función similar escrita en Pascal "puro".

Este otro truco puede sorprenderle. En muchas ocasiones, tenemos que convertir una cadena de Delphi al tipo PChar, casi siempre para llamar a rutinas del API de Windows. Por ejemplo:

```
var
  S: string;
begin
  S := 'Son las ' + TimeToStr(Now);
  MessageBox(0, PChar(S), 'Hora', MB_OK);
end;
```

Como sabemos, si una cadena de Delphi no está vacía, la variable apunta a un área de memoria que tiene el mismo formato que las cadenas de C/C++. Por lo tanto, nos bastaría una conversión de tipo estática, como la que acabo de mostrar, para que la función de Windows acepte el parámetro. Aquí no es importante la función en sí; he usado MessageBox por ser una de las funciones más conocidas del API de Windows.

Sin embargo, si ejecuta el código anterior y activa la ventana de la CPU en Delphi, verá algo inesperado: Delphi llama a una función interna para convertir un string en PChar. Es decir, lo que suponíamos que iba a ser una conversión sin "coste", provoca una llamada a un procedimiento auxiliar.

La culpa la tienen las cadenas vacías, que Delphi representa mediante un puntero nulo. Sin embargo, no todas las funciones del API de Windows aceptan un puntero vacío en los parámetros que necesitan una cadena con formato C/C++. Por este motivo, la función de conversión de Delphi comprueba si la cadena que recibe está vacía. En caso negativo, el mismo puntero inicial sirve como resultado. De lo contrario, la función devuelve un puntero a una zona de memoria reservada por Delphi, de manera que el puntero apunta directamente a un carácter nulo. Y así se evita tener que pasar un puntero nulo...

Naturalmente, en el ejemplo que mostré, es *imposible* que la cadena que se va a pasar a MessageBox esté vacía. ¿Qué necesidad tenemos entonces de pagar ese peaje? Para evitar la llamada innecesaria a la función de conversión, utilice el siguiente truco:

```
MessageBox(0, PChar(Pointer(S)),
  'Hora', MB_OK);
```

El primer *typecast*, al tipo Pointer despista a Delphi, y evita que se llame a la función de conversión para la conversión final a PChar.

Por último, una de las principales optimizaciones que pude aplicar al código de Sonata, tiene que ver con la forma en que se construían las respuestas HTML. Originalmente, se iban concatenando textos y etiquetas a una cadena de resultado, que era la que se devolvía al cliente. Esto es todo un patrón, común a las aplicaciones del tipo WebBroker, WebSnap, etc. El problema es que en cada concatenación, Delphi tiene que crear una nueva cadena y liberar, al menos, la memoria ocupada por uno de los dos operandos de la concatenación. El programador que conoce Java o C# sabe, sin embargo, que en estos lenguajes las cadenas son objetos *inmutables*, y la solución para evitar estas concatenaciones infinitas, que por causa de la "inmutabilidad" son aún peores desde el punto de vista de la eficiencia, es usar clases auxiliares: *string builders*, o acumuladores de cadenas.

No voy a copiar el código aquí, porque es algo extenso y está muy dirigido a la forma de trabajo de Sonata, pero la idea es sencilla: en vez de ir acumulando el texto en una variable de cadena, definí una nueva clase TStrOutput. Internamente, se mantiene un *buffer* de unos 8.192 bytes; experimentando con diferentes tamaños, encontré que un *buffer* mayor no mejoraba significativamente el rendimiento. Cuando quiero añadir una cadena al resultado, normalmente al final del texto ya construido, lo que hago realmente es copiar el nuevo sufijo dentro del *buffer* y

avanzar un puntero interno. Por supuesto, cuando el *buffer* se desborda hay que "vaciarlo" en una cadena interna. Esto significa

volver a las andadas... pero el número total de concatenaciones disminuye, y la mejora en velocidad es notable.

INTERBASE 7.1, CURSOS PARA FIREBIRD

Noticia con segunda intención: acabo de empezar, oficialmente, la escritura de **La Cara Oculta de Delphi 8**; tendréis que imaginar el motivo. En esta primera fase, la herramienta estrella son las tijeras: todo lo que tenía que ver con el BDE, se queda fuera. Los capítulos sobre SQL Server, Oracle y DB2 dejarán de hacer referencia a las versiones ya obsoletas de estos productos. E intentaré condensar los capítulos teóricos, para disminuir su extensión sin perder contenido. Y es que habrá que abrir espacio para muchas novedades...

Por otra parte, Borland acaba de anunciar la disponibilidad de InterBase 7.1. Entre las novedades anunciadas, está la compatibilidad con Windows Server 2003, un proveedor de datos para ADO.NET, un controlador JDBC mejorado, y soporte para *hyper-threading*, la técnica de Pentium IV para simular multiprocesamiento a nivel de hardware. Se ha mejorado el mantenimiento en segundo plano de los índices. Y al parecer, IB Console permite ahora monitorizar el funcionamiento de los servidores, aprovechando las tablas especiales en memoria introducidas por InterBase 7.

También se ha introducido la instrucción `drop generator`: ya no hará falta recurrir a las tablas del sistema para eliminar un generador. Y se anuncia el soporte de los puntos de control, o `savepoints`, en la jerga SQL.

En cuanto el producto esté disponible en España, Intuitive Sight anunciará los precios para las distintas variantes de licencia.

Nuestro objetivo, al igual que hicimos con InterBase 6.5, es ofrecer precios asequibles, sobre todo para instalaciones con muchos puestos. Además, en cuanto llegue el producto a nuestras manos, daremos más detalles técnicos, y probablemente efectuemos algunas pruebas comparativas.

En relación con InterBase, en estos momentos estoy preparando la actualización del curso de DB Express. Entre las sugerencias que he recibido y que vamos a aplicar, estará el soporte para FireBird. En estos momentos, hay poca diferencia entre InterBase y FireBird, pero la grieta puede ensancharse en cualquier momento...

Libros: estoy leyendo ahora uno bastante bueno sobre cómo crear controles y componentes para ASP.NET:

Developing Microsoft ASP.NET Server Controls and Components
Autores: Nikhil Kothari, Vandana Datye
Editorial: Microsoft Press
ISBN: 0735615829

No es el libro adecuado para empezar a aprender ASP.NET, porque asume que ya se domina esta materia. Pero es el mejor material para aprender a extender la funcionalidad de ASP.NET. Le advierto, como en ocasiones anteriores: sé que existen traducciones al castellano de la serie. No sé si ya existe la de este libro en concreto. Si está interesado, eche un vistazo antes de comprarlo en inglés, porque puede que esté traducido.

C# Y LAS EXCEPCIONES

Sísifo fue un buen señor griego al que, probablemente por algo relacionado con Hacienda, lo condenaron a arrastrar una enorme piedra por la ladera de una montaña... con el bonito detalle de que al llegar a la cima, la piedra caía, y todo volvía a empezar. Digo que debe haber sido un problema con Hacienda, porque si Sísifo hubiera sido un matón histérico como Aquiles y su compadre Patroclo, o un liante de cuidado como el astuto Odiseo (fecundo en ardides), en vez de una multa, le habrían dedicado un libro o un monumento. O ambos. Para ese monumento es para lo que hacía falta el dinero requisado a Sísifo, entre otras cosas...

Recordé a Sísifo hace un par de días, leyendo un libro "avanzado" sobre C#. Eche un vistazo al siguiente fragmento de código... y lloré:

```
SqlDataReader rd = cmd.ExecuteReader();
while (rd.Read())
    // ... hacer algo ...
rd.Close();
```

Bueno, vale, es mentira lo del llanto, pero quedaba más dramático. La variable `cmd`, cuya declaración e inicialización no se muestra en el fragmento, pertenece a la clase `SqlCommand`, que ADO.NET utiliza para enviar todo tipo de instrucciones a SQL Server. En este ejemplo, supondremos que el comando contiene una instrucción **select**.

Por su parte, la clase `SqlDataReader` sirve para manejar un cursor unidireccional de sólo lectura... más o menos como nuestros conjuntos de datos de DB Express, pero sin los *bugs* de este último. El cursor se obtiene cuando se ejecuta la sentencia del comando, por medio del método `ExecuteReader`. A propósito, y como detalle curioso, la clase `SqlDataReader` no tiene constructores públicos, y la única forma de obtener estos objetos es utilizar `ExecuteReader`.

Para recorrer el cursor, ejecutamos el método `Read` en un bucle; observe que hay que llamar a `Read` antes de poder acceder al primer registro. Cuando terminamos, es OBLIGATORIO cerrar el cursor, ejecutando el método `Close`. Bueno, puede decir un programador de Delphi, lo más probable es que al destruir el objeto, se llame internamente a `Close`, ¿o no?

Pero si esto le parece sensato, vaya haciéndose a la idea de que las cosas son muy diferentes en C#: no hay destrucción explícita en los lenguajes .NET, sino *garbage collection*. Es cierto que existen *finalizadores* de instancias, y que es probable que el finalizador de `SqlDataReader` llame internamente a `Close`... pero no existe ningún tipo de certidumbre sobre el momento en que tiene lugar la recogida de basura y la ejecución del finalizador.

¿Por qué es tan importante llamar a `Close`? La culpa la tiene una característica de SQL Server (y de MySQL, y de algún servidor SQL más, para ser justos): en una conexión con una base de datos, sólo puede haber una instrucción activa en cada momento. En ADO clásico, este problema se evita abriendo conexiones adicionales de forma transparente al programador, y lo mismo hace DB Express (aunque con menos éxito, viendo los resultados). ¿Quiere hacer la prueba? Probemos con un fragmento con más detalles:

```
SqlCommand cmd = new SqlCommand();
cmd.CommandText =
    "select au_lname from authors";
SqlDataReader rd = cmd.ExecuteReader();
while (rd.Read())
    /* instrucciones */;
cmd.CommandText =
    "select au_fname from authors";
```

```
SqlDataReader rd1 = cmd.ExecuteReader();
while (rd1.Read())
    /* instrucciones */;
rd1.Close();
```

La metedura de pata: no "cerramos" `rd`, el primer cursor, antes de abrir el segundo. La conexión sigue ocupada cuando intentamos obtener el segundo cursor, y se produce una excepción. La solución parece sencilla: tengamos cuidado, y no olvidemos que hay que llamar a `Close`. Es cierto que esto no es la maravilla que nos prometía la recogida automática de basura. Tenga en cuenta que aquí es muy fácil descubrir el motivo por el que falla la segunda consulta... porque el código culpable está a la vista, inmediatamente antes. Pero puede suceder que ejecutemos un cursor en un método y olvidemos cerrarlo, y que el error se manifieste en un método que se ejecuta mucho más adelante, cuando aún no ha tenido lugar la pasada de liberación de basura.

Sin embargo, en TODOS los libros de ADO.NET que he leído hasta el momento, el recorrido de cursores está incorrectamente programado, como en este ejemplo:

```
SqlDataReader rd = cmd.ExecuteReader();
while (rd.Read())
    // ... hacer algo ...
rd.Close();
```

Ejem... sí, es el código del principio... y sí, llamamos a `Close`... pero está mal programado. ¿Qué pasa si ocurre un error al procesar alguna de las filas del cursor? Pues lo mismo que en Delphi: se dispararía una excepción. Y como las excepciones interrumpen el flujo normal de ejecución, no se llegaría a ejecutar la línea donde llamamos a `Close`. Oh, sí, en algún momento se disparará el *garbage collector*, pero no tenemos seguridad alguna sobre cuándo llegará ese momento. Y mientras tanto, la conexión estará ocupada...

Evidentemente, deberíamos programar el bucle de recorrido con una instrucción de "protección de recursos":

```
SqlDataReader rd = cmd.ExecuteReader();
try {
    while (rd.Read())
        // ... hacer algo ...
}
finally {
    rd.Close();
}
```

Puedo contar con los dedos de la mano los ejemplos en libros, no sólo de ADO.NET, que utilizan alguna instrucción de manejo de excepciones. Lo peor es que, cuando incluyen alguna, casi siempre se trata de un **try/catch**, el equivalente del **try/except** de Delphi. ¿La causa? Probablemente se deba a que la mayoría de estos autores vienen del naufragio de Visual Basic. O aún, peor, son gente que trabajan con Java, e intentan sacar tajada de la situación pensando en las falsas analogías. Mi experiencia con Java, sin embargo, me dice que el programador de este lenguaje se confía pensando en que la recolección de basura resolverá todos sus problemas de recursos, y descuida la correcta devolución de recursos que no son precisamente la memoria dinámica.

En este ejemplo, existe una técnica más elegante, que nos ahorrará un par de líneas de código. Podemos sustituir el código anterior por esta alternativa:

```
using (SqlDataReader rd = cmd.ExecuteReader())
{
    while (rd.Read())
        // ... hacer algo ...
}
```

La instrucción **using** delimita un bloque, y la variable `rd`, declarada e inicializada en dicha instrucción, solamente está accesible dentro del bloque. La clase a la que pertenece la variable debe implementar el tipo de interfaz `IDisposable`, que solamente exige que se implemente un método llamado `Dispose`. Lo que la clase debe hacer exactamente en ese método, depende de la propia clase, pero se espera que este método devuelva todos los recursos adquiridos por la instancia. Excepto la memoria, se sobrentiende. `SqlReader`, efectivamente, implementa `IDisposable`, y su método `Dispose` se ocupa de llamar a `Close`, si no hemos cerrado el cursor.

Recapitulando, el bloque **using** anterior se puede traducir así:

```
{ // Nuevo bloque
  SqlDataReader rd = cmd.ExecuteReader();
  try {
    while (rd.Read())
```

AUTOMATIZAR CAMBIOS DE CURSOR

En casi cualquier libro o página sobre Delphi, encontrará un fragmento de código como el siguiente:

```
Screen.Cursor := crHourglass;
try
  // operación larga
finally
  Screen.Cursor := crDefault;
end;
```

Si usamos este patrón literalmente, tendremos problemas: ¿qué sucedería si la operación de larga duración llamase, a su vez, a otra operación escrita en Delphi... que también cambiase la forma del cursor? Evidentemente, al terminar la operación anidada, la forma del cursor cambiaría al cursor por omisión, aunque a la operación externa le faltase algún tiempo todavía para terminar.

No es difícil corregir el problema, modificando el patrón:

```
var
  C: TCursor;
begin
  C := Screen.Cursor;
  Screen.Cursor := crHourglass;
  try
    // Operación larga
  finally
    Screen.Cursor := C;
  end;
end;
```

Lo malo es que ahora tenemos más instrucciones, y es fácil hartarse si hay que cambiar el cursor con cierta frecuencia. Aún peor es que la mayor parte del código lo consume la instrucción **try/finally**, que no podemos encapsular; podemos, eso sí, definir métodos `SalvarCursor` y `RestaurarCursor`, pero nos ahorrarían sólo una pequeña parte de lo que tenemos que teclear.

Este caso pide a gritos una solución basada en la *Programación Orientada a Aspectos* (AOP)... que Delphi no soporta con naturalidad en su estado actual. En la AOP, estos problemas se resolverían añadiendo atributos declarativos, como en COM+ y en los lenguajes .NET. En este caso particular, añadiríamos un atributo `LongDuration`, o algo parecido, para marcar el método, en su declaración. Luego, en tiempo de ejecución, el entorno de ejecución debería realizar la operación conocida con el nombre de *aspect weaving* (tejido, o entramado de aspectos) para que el atributo añadido tuviese algún efecto real. En COM+, el *aspect weaving* se implementa mediante intercepción de llamadas, interponiendo *proxies* especializados entre el objeto y sus clientes, en el momento en que se instancia el objeto. Y algo parecido podría lograrse con un lenguaje .NET.

A pesar de lo interesante que puede ser la AOP, entre otras cosas por su novedad, nos conformaremos con una solución mucho más simple, y casi tan efectiva. ¿Cuál es el problema que tenemos? Hay un patrón de código en el que se ejecutan ciertas instruccio-

```
// ... hacer algo ...
}
finally {
  ((IDisposable) rd).Close();
}
}
```

Compare este fragmento con el basado en **using**, y decida si merece la pena o no usarlo...

¿Y qué tiene que ver Sísifo con todo esto? Pues que estoy revisando lo que sucedió cuando Delphi salió a la calle: en muy pocos libros se explicaban correctamente las reglas para el manejo de excepciones. La piedra ha caído desde la cima de la montaña, y con mucha paciencia, hay que volver a llevarla a donde tiene que estar.

nes antes y después de una operación central. ¿Qué tal si aprovechamos un tipo de interfaz para ahorrarnos código?

Vamos a crear una unidad, con dos funciones globales declaradas en su sección de interfaz:

```
unit Cursores;

interface
uses Controls, Forms;

function SaveCursor: IUnknown; overload;
function SaveCursor(C: TCursor):
  IUnknown; overload;

implementation
// ...
end.
```

Un adelanto: sólo tendremos que incluir una llamada a una de estas funciones antes de iniciar una operación lenta.

Ahora vamos a declarar una clase dentro de la implementación de la unidad:

```
type
  TSaveCursor = class(
    TInterfacedObject, IUnknown)
  private
    OldCursor: TCursor;
  constructor Create(C: TCursor);
  destructor Destroy; override;
end;
```

Esta sería la implementación de `Create`:

```
constructor TSaveCursor.Create(C: TCursor);
begin
  OldCursor := Screen.Cursor;
  Screen.Cursor := C;
end;
```

Al crearse un objeto de la clase `TSaveCursor`, recordaríamos la forma actual del cursor dentro de uno de sus campos, y cambiaríamos el cursor de acuerdo al tipo de cursor pasado como parámetro al constructor. Esto se complementa con lo que sucede al destruir el objeto:

```
destructor TSaveCursor.Destroy;
begin
  Screen.Cursor := OldCursor;
  inherited Destroy;
end;
```

Al destruir el objeto, por lo tanto, se restaura la forma inicial del cursor. Para terminar con la unidad, veamos lo que hacen las dos versiones sobrecargadas de la función global `SaveCursor`:

```
function SaveCursor(C: TCursor): IUnknown;
begin
  Result := TSaveCursor.Create(C);
end;
```

```
function SaveCursor: IUnknown;
begin
    Result := SaveCursor(crHourglass);
end;
```

Observe atentamente que ambos métodos crean un objeto de la clase `SaveCursor...` pero lo que devuelven al contexto que los llama es un puntero de interfaz, no el "puntero de clase" normal.

Y todo este embrollo, ¿para qué nos sirve? El truco está en el tiempo de vida de los punteros de interfaz. Para verlo con más claridad, veamos cómo tendríamos que utilizar la función `SaveCursor` para cambiar el cursor dentro de un método que ejecuta una operación lenta:

```
procedure OperacionLenta;
begin
    SaveCursor;
    // ... instrucciones lentas ...
end;
```

Cuando llamamos a `SaveCursor`, éste nos devuelve un puntero de interfaz instanciado... y cambia el cursor, por supuesto. Los objetos apuntados por tipos de interfaz se destruyen cuando no quedan referencias a él. En este caso, ese momento llega al

terminar el método. En ese momento se ejecutaría el destructor de `TSaveCursor`, y restauraríamos el cursor a su estado inicial.

¿Y qué pasa con las operaciones anidadas? Hagamos una prueba: ponga un botón sobre un formulario e intercepte su evento `OnClick`.

```
begin
    SaveCursor(crHandPoint);
    Sleep(1000);
    SaveCursor;
    Sleep(1000);
end;
```

Primero activamos el cursor de la mano que señala con un dedo, y esperamos un segundo. Luego, cambiamos al reloj de arena, y esperamos otro segundo. Visualmente, podemos comprobar que al terminar los dos segundos, el cursor vuelve a tomar su forma habitual. Y si traceamos la ejecución del evento, instrucción por instrucción, veremos que la restauración del cursor tiene lugar en el orden lógico que todos esperamos: primero se cambia el reloj de arena por la mano, e inmediatamente después, la mano se cambia por la flecha. Nos hemos ahorrado el **try/finally**, y unas cuantas líneas de código más...

CURSORES DE PANTALLA EN .NET

¿Cómo podríamos resolver este mismo problema del cambio de cursor en .NET... sin meternos en las honduras del manejo de atributos?. Pudimos aprovechar los tipos de interfaz de Delphi gracias a que estos implementan un conteo de referencias para su destrucción automática. Sin embargo, en C# se utiliza recolección de basura para estos menesteres, y la destrucción de objetos es no determinista...

Afortunadamente, podemos echar mano de la misma instrucción **using** que hemos explicado antes. Dije que **using** debía utilizarse con clases que implementasen la interfaz `IDisposable`. Pues bien, vamos a declarar una clase `SaveCursor` que implemente dicha interfaz:

```
class SaveCursor: IDisposable
{
    private Cursor oldCursor;

    // ...
}
```

Al igual que en la implementación para Delphi, hemos creado un campo privado, `oldCursor`, para guardar el tipo original de cursor. Hay una diferencia, no obstante: el tipo `Cursor` de C# es una clase, y por lo tanto, es un valor manejado siempre por referencia.

Nuestra clase tendrá dos constructores:

```
public SaveCursor(Cursor newCursor) {
    oldCursor = Cursor.Current;
    Cursor.Current = newCursor;
}

public SaveCursor() :
    this(Cursors.WaitCursor) {
}
```

En la primera variante sobrecargada del constructor, pasamos un nuevo tipo de cursor explícitamente. El segundo constructor no tiene parámetros, y lo utilizaremos cuando el nuevo cursor deba ser el típico reloj de arena. Observe la sintaxis de C# para ejecutar el código de otro constructor dentro de la misma clase.

Un poco más complicado es el siguiente método, que sirve de implementación al único método de la interfaz `IDisposable`:

```
void IDisposable.Dispose()
{
    if (oldCursor != null)
        Cursor.Current = oldCursor;
    oldCursor = null;
}
```

```
GC.SuppressFinalize(this);
}
```

¿Se ha fijado en el extraño nombre que le hemos dado? La técnica más frecuente de implementación de interfaces consiste en declarar, dentro de la clase, métodos que tengan el mismo nombre y prototipo que los métodos del tipo de interfaz; en este ejemplo, `SaveCursor` debería implementar un solo método llamado `Dispose`. Ahora bien, ese método de la clase puede declararse como público, privado, protegido, etc. Si quisiéramos que el método no pudiera ser ejecutado directamente, a través de la referencia de clase, deberíamos "esconderlo". La técnica que he utilizado esconde el método con más facilidad. Primero, note que no hay un modificador de visibilidad (**public**, **private**), sino que la declaración del método arranca directamente con el tipo de valor de retorno (**void**). Luego, el "nombre" que recibe el método es una combinación del nombre de la interfaz y del nombre del método de la interfaz a implementar. Con esto logramos crear un método privado, con un nombre especial generado por el compilador, y al cuál sólo tendremos acceso si convertimos la referencia a un objeto de esta clase en un puntero de interfaz.

Pasemos al interior de `Dispose`. Las dos primeras instrucciones se comprenden sin mayor problema: restauramos el cursor original, y limpiamos la referencia, para evitar cualquier posible caso de ejecución redundante... que ahora veremos que será imposible por otra razón. La última instrucción es la más interesante: como este objeto ya ha "liberado" todos los "recursos" que tenía (ninguno, por cierto), no hace falta "finalizar" la instancia en el momento en que el recolector de basura se libere del objeto. O en otras palabras: no hará falta llamar al destructor de la clase sobre el objeto. GC es la clase que agrupa los métodos de control del recolector de basura, y `SuppressFinalize` es, evidentemente, un método estático que marca como "destruido" el objeto que recibe como parámetro.

Sólo nos queda ver cómo se utilizaría `SaveCursor` para cambiar, y posteriormente restaurar, la forma del cursor:

```
using (new SaveCursor()) {
    // operaciones de larga duración
}
```

Como puede ver, el nuevo código es mucho más pequeño y elegante que el código alternativo que tendríamos que teclear sin este truco. Dentro de la cabecera de la instrucción **using** creamos una instancia de la clase que hemos declarado; no hace falta asignar esta instancia en una variable, como se muestra en el ejemplo. Luego, ya dentro del bloque de instrucciones asociado a

using, podemos escribir las instrucciones de la operación. Cuando el programa termine con estas instrucciones, se ejecutará el método `Dispose` del objeto creado, a través de un puntero de

interfaz. Y esta finalización tendrá lugar incluso si se producen excepciones dentro del bloque de instrucciones. Asunto concluido.

DESDE EL FRENTE DE BATALLA

Noticia con tercera intención: la escritura de La Cara Oculta de Delphi 8, que comencé a principio de julio, se ha retrasado, por causas ajenas a mi voluntad (podéis adivinar la causa nuevamente). Las cosas en el frente Delphi se mueven muy lentamente. La novedad más significativa de julio y agosto ha sido una actualización del controlador de **DB Express para DB2**, en Delphi 7.

Lo que sí es ya una realidad es el **C#Builder**: el entorno de desarrollo de Borland para C#. Entre las características de este producto, puede interesarle cierta semejanza con los entornos de desarrollo para Delphi y C++ Builder, que pueden facilitar en cierto grado el aprendizaje del nuevo lenguaje a programadores curtidos de Delphi, como un servidor. De todos modos, la filosofía de diseño es más parecida a la de JBuilder que a la de Delphi. También hay que destacar que la versión más limitada del producto, la Personal, es gratuita. C#Builder incluye un proveedor .NET para InterBase, probado al parecer con InterBase 7.1, y ofrece algunas clases especializadas para el acceso a datos, que permiten mostrar datos reales en los controles durante el diseño. A estas alturas, no se trata de algo de vida o muerte, pero estas pequeñas mejoras se agradecen.

En realidad, una de las bazas con la que cuenta Borland es toda una retahíla de "herramientas de diseño" que acompañan a la criatura. Reconozco que soy bastante escéptico respecto a las metodologías que prometen convertir a cualquier programador cenutrio en una estrella del rock. Utilizo UML regularmente, pero

como herramienta de análisis, diseño y documentación; jamás se me ha pasado por la cabeza que se pueda "programar" en UML... y no creo que sea una buena idea.

Otro "punto fuerte" de C#Builder, destacado por la propia Borland es la interacción con Java, CORBA, y otros sistemas pecaminosos de este estilo. Si a usted le preocupa algo de esto, échele un vistazo a C#Builder. porque puede que sea de lo mejor en ese ramo...

En otro orden de asuntos, vamos a lanzar próximamente un programa de colaboración con IntSight para *webmasters*. Vamos a habilitar puntos de enlace en nuestra red para la venta de cursos a distancia desde referencias ubicadas en otros sitios. Con este sistema, quien tenga una página en Internet y se afilie a nuestro programa, podrá enviar usuarios a las páginas de IntSight, y recibir una comisión por ventas de cursos a distancia. Hay bastante trabajo técnico en marcha, y próximamente anunciaremos más detalles sobre este plan.

Por último, ya está listo para su inminente comercialización el **IntSight's Server Explorer**, un asistente de desarrollo para DB Express y DataSnap, con versiones para Delphi 6 y 7. Hay un tutorial en mi página (www.marteens.com/isse/isse00.html) sobre el uso de este producto, y en breve habilitaremos los formularios de pedido. El producto costará 35 euros, impuestos incluidos, pero será gratuito para quienes hayan adquirido cualquiera de nuestros cursos a distancia.

CONMOCIÓN EN EL UNIVERSO DÉLFICO

El 22 de octubre 1993, los sismógrafos del US Geological Survey captaron dos potentes vibraciones, aparentemente no relacionadas. La primera fuente de ondas tenía la Antártida como epicentro; siete décimas de segundo más tarde, recibían una señal similar procedente del sur de la India. Treinta y tres días más tarde, algo parecido volvió a ocurrir: un suceso sísmico iniciado en el sur de Australia, propagándose en línea recta a través de la Tierra, y saliendo de nuevo a la superficie por la Antártida. Estas señales fueron confirmadas por estaciones de India y Australia, pero también se sintieron en Bolivia y Turquía.

El nexo entre los sucesos descritos fue descubierto, pocos meses más tarde, por tres científicos de la Southern Methodist University, de los Estados Unidos. La única explicación plausible que se ha encontrado es que las ondas de choque fueron producidas por una entidad formada por quarks del tipo "*strange*", conocida como "*strangelet*", cuya existencia teórica había sido predicha años antes. La materia extraña formada exclusivamente por quarks tendría una densidad superior a la de la materia que forma las estrellas neutrónicas, y una sola partícula de esta materia sería capaz de provocar ondas sísmicas detectables con instrumentos convencionales.

La conmoción causada por la última "*carta abierta*" o epístola de San Borland a los Délficos sí que era de prever. Primero, le contaré mi versión resumida de la carta, por si no ha podido leerla: básicamente, la cosa va de que en Borland están muy excitados... perdón, entusiasmados, por el próximo producto que van a venderos (claro, si no se entusiasman ellos mismos, qué va a quedar para el resto de los mortales), que las cosas han cambiado mucho, y que han decidido, por medio de encuestas, que la gente que programamos en Delphi ya somos bastante mayorcitos como para comprometernos (esto suena a matrimonio) *a full* (como en el anuncio de Burn que están pasando en España), con la plataforma .NET. Vaya, hombre, siempre pensé que cuando ocurrían estas cosas, siempre era uno el primero en darse cuenta...

La introducción del anuncio es bastante desafortunada; es muy probable que incluso sea verdad, pero la redacción deja mucho que desear en las formas. Aunque lo más interesante viene entonces: la próxima versión de Delphi será Delphi.NET... ¿qué que hay de extraño en esto?

- 1 Las promesas se hacen para ser rotas, claro. Pero en la versión anterior de esta misma carta abierta, de mediados de abril de 2003, se decía que la próxima versión iba a incluir una actualización de Delphi para Win32, además de la versión final, completamente funcional, de Delphi.NET.
- 2 Delphi 7 está lleno de errores y problemas, y es lamentable que en todo el tiempo transcurrido desde su aparición, Borland no se haya molestado en sacar siquiera un mísero *Update Pack* (léase *parche*) para corregir al menos los problemas más sangrantes.

Lo que ha sucedido es comprensible: Borland se ha dado cuenta de que no podía dar la batalla en los dos frentes, Win32 y .NET, al mismo tiempo, y ha decidido concentrarse primero en Delphi.NET. Por supuesto, me hubiese gustado que la próxima actualización de Delphi incluyese de entrada un verdadero Delphi 8 para Win32 *más* el nuevo Delphi.NET. Pero también es cierto que prefiero un Delphi.NET (o un Delphi para Win32) estable, antes que dos productos con problemas. Hasta aquí, suspiro de resignación, aunque en parte de alivio...

Las cosas se liaron un poco más, sin embargo: la versión del anuncio que ahora puede leer ha sido retocada a toda prisa. En la versión original, no quedaban claros algunos flecos: ¿seguirá

desarrollándose Delphi para Win32? ¿se considerarán Delphi para Win32 y para .NET productos separados, pagando en consecuencia dos veces para tener un entorno multiplataforma? Finalmente, ¿se va a abandonar el proyecto de migrar la VCL a .NET?

Las primeras interpretaciones por parte de los programadores no tardaron en aparecer... y como era esperable, se dio la peor interpretación posible a cada una de las preguntas abiertas. "Piensa mal, y acertarás", ya sabe. De ahí, los retoques inmediatos a la redacción. Y aquí viene mi parte de interpretación personal, porque la famosa carta sigue siendo tan ambigua como un violín: todo el mundo puede tocar con él la melodía que prefiera oír. Lo que he entendido ha sido lo siguiente:

- Sí, Delphi para Win32 seguirá existiendo... al menos mientras haya gente programando en Win32 a secas. Pregunta interesante: ¿cómo se puede saber esto último con exactitud?
- Habrá una actualización de Delphi para Win32, que saldrá algún tiempo después de Delphi 8. Delphi para .NET seguirá llamándose Delphi, y se mantendrá la actual numeración de versiones.
- Borland tiene una modalidad de compra lejanamente parecida a la garantía de cursos a distancia de IntSight, en la que te aseguran la actualización gratuita... dentro de un intervalo de tiempo prefijado. Borland quiere tranquilizarnos sobre la posibilidad de tener que pagar dos veces con esta garantía. Sin embargo, mi mayorista de productos Borland no me ha mencionado este asunto para nada. Si la garantía no se establece en el momento del registro, no sé cómo va a funcionar en los países como España que no tenemos una representación directa de Borland. Por supuesto, intentaré averiguar todo lo posible sobre este punto, y os informaré sobre lo que averigüe en el próximo boletín.
- Efectivamente, se va a traducir la VCL a .NET... pero dentro de ciertos límites (era previsible). No habrá actualización automática del código actual a Delphi.NET. Es más, personalmente, no recomiendo en absoluto la dependencia respecto a lo que pueda inventar Borland y bautizarlo como VCL.NET. Es de sentido común: si hay que programar en .NET, ¿para qué inventarse una capa adicional que sólo introduciría dependencias innecesarias? El API de Win32 era lo suficientemente primitivo como para que la VCL tuviese sentido; y las clases MFC no eran clases orientadas al desarrollo RAD. Eso no sucede con las clases de .NET.

¿Qué oportunidad tiene Delphi.NET de competir con Visual Studio... o con el propio Borland C# Builder? Todo depende de cuán productivo llegue a ser el entorno de desarrollo. Borland, gracias a su éxito con Java, se ha formado una imagen de sí misma como proveedora de herramientas para desarrolladores; en definitiva, el compilador de Java es, como el de Microsoft, un producto gratuito. Claro, otra cosa sería si Sun ofreciese un entorno de desarrollo decente para Java; ahí se rompe la analogía. Pero no es muy prometedora la actual ausencia, en C# Builder, de muchos de los asistentes de Visual Studio. La versión 2 de C# Builder mejorará ese aspecto, pero estamos hablando del futuro.

¿Y mi consejo cuál es? Pues que haga lo que yo *hago*, no lo que *digo* (lo normal es lo contrario). Creo que es lo más honesto por mi parte. Como esto se veía venir desde hace unos meses, en IntSight congelamos los proyectos de desarrollo a mediano y a largo plazo. Ahora, con el panorama un poco más claro, vamos a reanudar la mayoría de ellos... en C#. Esto no quiere decir que abandonemos Delphi, ni mucho menos. Y más adelante, en dependencia de lo bueno o malo que resulte ser Delphi.NET y la próxima actualización de Delphi para Win32, probablemente

retomemos algunos de los proyectos pendientes con estas herramientas.

- Nuestros cursos a distancia para Delphi seguirán actualizándose regularmente. El próximo en actualizarse será el curso de Internet, probablemente en la primera mitad de noviembre.
- Por supuesto, aparecerán nuevos cursos sobre C#. El primero de ellos, sobre programación con ADO.NET y .NET Remoting... que como comprenderá, es el equivalente funcional del curso actual de ADO Express/DataSnap.
- Desde septiembre estamos ofreciendo cursos de C# a la medida, para empresas; ya sea en aulas de la propia empresa, con desplazamiento del profesor, o en aulas de IntSight. Puede consultar un temario típico en la siguiente página:

PROPIEDADES DINÁMICAS

Todos sabemos que los formularios de Delphi se inicializan con la ayuda de un recurso especial que se genera a partir del fichero DFM asociado. Los datos del fichero DFM, una vez convertidos a una conveniente representación binaria, se almacenan dentro del fichero ejecutable. Casi todo el mundo ha pensado en algún momento: ¿y si pudiésemos "sacar" el recurso fuera del ejecutable? En el momento de crear el formulario, la aplicación buscaría el fichero externo, y extraería de éste las propiedades de los componentes del formulario. De esta manera, sería posible modificar la apariencia de la aplicación sin necesidad de recompilar el programa. Por supuesto, ésta sería un arma de doble filo, especialmente si sacásemos todo el DFM a la luz. Puede que usted haya codificado alguna que otra regla de negocio dentro de las propiedades de algunos controles, y no le interese que alguien pueda saltársela...

Mi objetivo actual no es mostrar cómo se podría implementar tal técnica en Delphi, sino mostrar una técnica equivalente que sí soporta la plataforma .NET, y que tiene en cuenta los peligros potenciales que acabo de mencionar. La técnica se conoce con el nombre de *propiedades dinámicas* y, como no podía ser de otro modo, podremos usarla en el futuro Delphi.NET. Voy a mostrar los ejemplos en C#, pero la traducción a Delphi.NET es casi inmediata.

Comencemos por lo que veríamos en un entorno de desarrollo como Visual Studio o C# Builder. Suponga que tenemos un botón sobre un formulario. La propiedad equivalente al `Caption` de Delphi, para el texto que mostraría el botón, se llama `Text` en .NET. Si asignamos la cadena `Botón` como valor de esta propiedad, se generaría la siguiente instrucción de inicialización:

```
this.button1.Text = "Botón";
```

Para quien no haya trabajado todavía con estos entornos de desarrollo, la instrucción anterior se incluye dentro del método `InitializeComponents`, generado y mantenido por el diseñador de formularios. Este método se ejecuta desde el constructor del formulario.

Si seleccionásemos el botón y activásemos el Inspector de Objetos (que en VS se llama *Inspector de Propiedades*), veríamos una seudo propiedad en el Inspector, con el nombre de (*Dynamic Properties*), paréntesis incluidos. La falsa propiedad tiene un botón de expansión a su izquierda, como las propiedades de tipo clase o conjunto en Delphi, y al expandir la propiedad, inicialmente sólo se muestra una línea titulada (*Advanced*), con un pequeño botón que muestra, al ser pulsado, un diálogo modal. En este diálogo se nos presenta una lista con las propiedades del objeto, en nuestro caso, del botón. Seleccionamos la propiedad `Text` y cerramos el cuadro de diálogo.

Al hacer esto, veremos un par de cambios en el Inspector de Objetos. El primero de ellos: dentro de (*Dynamic Properties*)

<http://www.marteens.com/csh001.htm>

- El próximo libro que IntSight editará será "**La Cara Oculta de C#**", para mediados de noviembre. Para la actualización de "La Cara Oculta de Delphi 6", que es un libro para Win32, esperaremos a la correspondiente actualización de Delphi/Win32. Es posible que entre tanto publiquemos algo sobre Delphi.NET como lenguaje, en una colaboración entre autores.
- Y, cómo no, seguiremos con los trucos, tanto para Delphi en Win32, como para Delphi.NET y C#.

¿En menos palabras? Sigo teniéndole cariño a Delphi (y una parte importante de mi empresa se basa en Delphi), y creo que Borland se merece una oportunidad. Pero no voy a jugarme la vida en una apuesta tan incierta. En definitiva, .NET es una plataforma sólida y muy interesante, y no quiero perderme la ocasión.

aparecerá ahora una segunda línea de propiedad. El nombre de propiedad será `Text`, y como valor, se mostrará `button1.Text`; en un momento explicaré qué significa. Además, en la "verdadera" propiedad `Text` del botón, seguirá mostrándose el valor que asignamos inicialmente, es decir, `Botón`, pero en la mitad izquierda de la línea, donde se muestra el nombre de la propiedad, aparecerá un pequeño icono: un rectángulo azul que, al menos para mí, se parece al icono que utiliza Windows para los ficheros de configuración.

Pero donde hay cambios notables es en el código de inicialización de los componentes del formulario. La primera línea de `InitializeComponents` declarará y creará un objeto local:

```
AppSettingsReader configAppSettings=  
new AppSettingsReader();
```

En realidad, he abreviado la instrucción quitando el espacio de nombres al que pertenece la clase `AppSettingsReader`, que es `System.Configuration`. La clase en cuestión sirve para manejar un fichero de configuración, que debe tener el mismo nombre que la aplicación, incluyendo la extensión, *más* la extensión `.config`, y que debe estar ubicado en el mismo directorio que la aplicación. Por ejemplo, si la aplicación se llama `app.exe`, el fichero de configuración debe llamarse `app.exe.config`. En este ejemplo, el contenido del fichero sería el siguiente:

```
<?xml version="1.0"  
encoding="windows-1252"?>  
<configuration>  
<appSettings>  
<add key="button1.Text"  
value="Botón"/>  
</appSettings>  
</configuration>
```

Más adelante, encontrará que la propiedad `Text` del botón se configurará mediante esta instrucción:

```
this.button1.Text = ((string)  
(configAppSettings.GetValue(  
"button1.Text", typeof(string))));
```

Como comprenderá, se utiliza el objeto lector de configuración creado al inicio del método para localizar una clave llamada `button1.Text`, leer su valor y asignarlo en la propiedad que hemos configurado como propiedad dinámica.

Gracias a esta técnica, cuando la aplicación ya esté en funcionamiento, nos bastaría modificar valores dentro del fichero de configuración para cambiar las propiedades que hayamos marcado como dinámicas de los objetos de la aplicación. Todo, sin tener que recompilar nada. ¿Problemas y peligros? Por supuesto que los hay: usuarios muy listos que puedan tocar lo que no deben, y programadores tontos que pongamos información confidencial o peligrosa en estas propiedades.

Uno de los usos más frecuentes de las propiedades dinámicas es almacenar la propiedad `ConnectionString`, de un componente de conexión de bases de datos, en el fichero de configuraciones. ¿Es correcto hacerlo así, o no? Si nos conectamos a un servidor SQL Server que soporte la seguridad integrada, no habrá pro-

blema alguno en usar esta propiedad dinámica, porque no necesitaremos incluir en la cadena de conexión ni el nombre del usuario ni su contraseña. En caso contrario, sería un suicidio confiarse de este modo. Ya sabe: un gran poder conlleva una gran responsabilidad...

EXTENSORES DE PROPIEDADES

Otra de propiedades: cuando quise escribir mi primera aplicación con formularios en C#, tropecé con una sorpresa. Traje un botón a un formulario, le cambié el título mediante la propiedad `Text`, le asigné un manejador para el evento `Click`... pero por mucho que busqué, no encontré una propiedad equivalente al `Hint` de Delphi, que me permitiese mostrar la típica ayuda flotante con fondo amarillo a la que ya estamos acostumbrados.

Después de buscar un poco en la paleta de componentes, encontré un componente llamado `ToolTip`, y naturalmente, lo solté sobre el formulario. Tanto en Visual Studio como en el Borland Developer Studio, los componentes que no son controles no se sitúan directamente sobre el formulario, como en Delphi, sino en una zona rectangular, casi siempre ubicada en la parte inferior del diseñador, llamada la Bandeja de Componentes (o *Component Tray*).

¿Un solo componente para proveer de textos de ayuda a todos los controles de un formulario? Las propiedades del componente `ToolTip`, además, sólo controlan los detalles globales del sistema de indicaciones flotantes: el tiempo que tarda la ayuda en aparecer, el tiempo que tarda antes de desaparecer, si el sistema está activo o no... Pero nada, aparentemente, que diga cuál mensaje debe mostrarse para cada control.

Después de un rato de devanarme la mollera, ¡bingo!, se me ocurre seleccionar de nuevo el botón: ¡y veo una nueva propiedad en el control, llamada *"ToolTip in toolTip1"*! No sé si se trata de una novedad en .NET, o si algo parecido estaba ya presente en Visual Studio 6, pero para un programador de Delphi como yo, es todo un descubrimiento: un componente que hace brotar propiedades y eventos en otros componentes. ¿Cómo se logra?

Lo que sucede es que la clase `ToolTip` implementa el tipo de interfaz `IExtenderProvider`:

```
interface IExtenderProvider
{
    bool CanExtend(object objeto);
}
```

Este método debe decidir si determinado componente situado en el mismo formulario que el extensor de propiedades puede recibir la propiedad, o propiedades, definidas por el extensor. Aparte de esto, la clase del componente extensor debe marcarse con el atributo `ProvideProperty`. Por ejemplo:

```
[ProvideProperty("ToolTip", typeof(Control))]
public class ToolTip:
    Component, IExtenderProvider
{
    // ...
}
```

Finalmente, y es aquí donde está el truco principal, la propiedad que va a añadirse a otros componentes se implementa mediante un par de métodos, que se nombran utilizando el conocido patrón `GetXxx/SetXxx`. Por ejemplo, el componente `ToolTip` implementa estos métodos:

```
public string GetToolTip(Control ctrl);
public void SetToolTip(Control ctrl,
    string tip);
```

Como puede ver, todo consiste en que los métodos que implementan la nueva propiedad reciben como parámetro adicional, el objeto que han "extendido". Como consecuencia, el extensor debe implementar internamente una estructura similar a un diccionario, que asocie los objetos extendidos con los valores de la propiedad adicional para cada uno de ellos. En .NET, este diccionario se puede implementar fácilmente mediante un objeto de la clase `HashTable`.

Volviendo al componente `ToolTip`, cuando se asigna una cadena en la "propiedad" que se ha añadido al botón, el diseñador de formularios añade la siguiente instrucción para la inicialización:

```
this.toolTip1.SetToolTip(
    this.button1, "Ayuda para el botón");
```

Hay que comprender que estas extensiones pueden utilizarse con sentido en .NET gracias a otra característica que lo diferencia de Delphi, y del antiguo Visual Basic: los eventos con multidifusión. Si el componente al que se le añaden propiedades debe comportarse pasivamente, no tendríamos problemas con un sistema de eventos como el de Delphi tradicional, pero si tuviésemos que escuchar eventos del componente que se ha ampliado, las opciones serían pocas y complicadas. Encadenar eventos, por ejemplo, es poco seguro: piense en lo que podría suceder si los eventos se "desregistrasen" en un orden que no fuese el orden inverso al del registro. Si quisiéramos resolverlo interceptando mensajes de Windows, o creando una subclase dinámicamente, tendríamos el mismo problema, de todos modos. El hecho de que en un evento .NET puedan registrarse y quitarse del registro más de un manejador, sin importar el orden relativo entre ellos, es una bendición a la que tendremos que acostumbrarnos.

CUERNO LARGO

...no, no mire la cabeza de su compañero de trabajo. Lo de "cuerno largo" es por Longhorn, el nombre en clave de la próxima versión de Windows. Hace ahora un mes, Microsoft celebró su PDC 2003, es decir, su Conferencia de Desarrolladores Profesionales. En contraste con el secretismo habitual de las reuniones de Borland (la BorCon 2003 tuvo lugar, sospechosamente, a la semana siguiente), que si no te vistes de pagano y te acercas, no te enteras de nada, las principales noticias de la conferencia de Microsoft tuvieron una excelente cobertura en Internet, comenzando por la propia página de la MSDN.

WHIDBEY

Whidbey es un término comodín, que se aplica como adjetivo a varias tecnologías. Tenemos una CLR Whidbey, con soporte nativo para plantillas genéricas, existe un C# Whidbey, que añade esas plantillas al lenguaje, además de métodos anónimos para manejadores de eventos, tipos parciales e iteradores. Y el Visual Studio Whidbey cierra el ciclo, con un entorno de desarrollo más productivo, con mejoras importantes en IntelliSense y con una interesante implementación de unas cuantas operaciones de *refactorización*.

Una de las sorpresas que se están cocinando en Visual Studio es lo que Microsoft llama *Service-Oriented Application Designer*. Al parecer, es un producto de ciclo cerrado que permite diseñar, representar e implementar sistemas que trabajan con "servicios". La idea es que la especificación del sistema sea parte del sistema, que los requisitos sean algo más que un frase en un papel. Suena confuso, pero el tiempo dirá...

AVALON

Avalon es un nuevo subsistema para la interfaz gráfica de usuarios en .NET, que estará disponible a partir de Longhorn. Una de las características que más llama la atención en Avalon es el lenguaje XAML (pronunciado "zamel", con la "z" sibilante inglesa de "zoom"). ¿Encuentra usted algo bueno a HTML? A pesar de su pobreza expresiva (iseamos sinceros de una vez!), la posibilidad de cambiar el aspecto de una aplicación para Internet retocando el código HTML es algo que a veces causa envidia. Es cierto que en el viejo Windows solíamos usar una antigualla llamada "ficheros de recursos"... pero la funcionalidad era muy limitada.

En Avalon, se separa la funcionalidad de los controles de su diseño, utilizando una técnica parecida a la "code behind" de ASP.NET. El diseño se controla con un fichero XAML que se puede modificar en cualquier momento. XAML permite incluso controlar determinados eventos mediante *triggers* basados en cambios de valores de propiedades. También se soporta una técnica similar a las hojas de estilo en cascada. De esta manera, el aspecto de una aplicación puede cambiarse radicalmente con muy pocas líneas de código.

INDIGO

¿Le suena a algo la frase "programación orientada a servicios"? A mí me parece un término antagonista de "programación orientada a objetos". ¿Es que alguien quiere sostener que el deslizamiento por arrastre es mejor que la rueda? Bueno, no sería tan tonto: pregúntele a un esquiador. Todo depende de las circunstancias...

Al parecer, el problema de la P.O.O. está en la escala. Cuando hay que cruzar ciertas fronteras, el mantenimiento de un estado interno, y la composición algebraica de primitivas se vuelven operaciones sumamente onerosas. Y no se trata de una novedad: es la lección que hemos aprendido a costa de los fracasos de J2EE, y de las primeras intentonas de DCOM. Pero ocurre que en

Microsoft es donde primero han mostrado estar dispuestos a aceptar las consecuencias.

Indigo es el sistema que sustituirá al actual revoltijo de técnicas de acceso remoto y de servicios corporativos de Windows. Estará basado en SOAP y en mensajería XML. Tanto DCOM como CORBA, Java RMI y .NET Remoting, están basados en llamadas remotas: un método, aunque no devuelva un tipo de retorno, nos obliga a esperar una respuesta. En Indigo, el patrón de interacción básico es el mensaje: no sólo se pueden enviar mensajes sin retorno, sino que podremos recibir mensajes a modo de eventos. Más aún, se incluirán patrones más complejos basados en el enrutamiento de mensajes, a la manera de **WS-Routing**.

WINFS

WinFS es el nuevo sistema de ficheros de Longhorn. Es un sistema que permitirá transacciones, algo que se echa de menos en el actual sistema de ficheros. Además, el sistema incorporará conocimiento extensible sobre tipos de datos. Al parecer, parte del motor de SQL Server estará involucrado en la implementación de WinFS.

YUKON

Esta vez, no se trata de un nombre en clave del todo desconocido: **Yukon** quiere decir SQL Server 2003. Y aquí tenemos mucho de que hablar. Una de las novedades más interesantes es la inclusión de la **CLR**, el entorno de ejecución de .NET, dentro del servidor. Gracias a esta inclusión podremos programar, utilizando cualquiera de los lenguajes .NET, funciones escalares, funciones que devuelven tablas, procedimientos almacenados, incluyendo aquellos que devuelven conjuntos de resultados al proceso que los ejecuta, triggers... e incluso funciones agregadas definidas por el usuario. Tenga presente que la CLR compila a código nativo, por lo que todo este código funcionaría a velocidad crucero. Imagine lo que significará poder escribir cláusulas **check** en una tabla que utilicen expresiones regulares para validar columnas que almacenan direcciones de correo electrónico. O poder extender la funcionalidad de una base de datos que implementa la gestión de productos financieros (préstamos, hipotecas y esas cosas) simplemente registrando un nuevo conjunto de clases escritas en C#. Hubiera matado por tener algo así hace cuatro años...

Pero también hay nueva funcionalidad en Transact SQL. ¿Ha trabajado alguna vez con DB2? Una de las características más interesantes de este sistema es la forma en que se definen consultas recursivas, mediante una extensión del lenguaje SQL. Ahora SQL Server introduce un tipo de consultas recursivas muy parecido, las llamadas **CTE's**, o *common table expressions*. Eche un vistazo a la siguiente consulta:

```
with Emps(EmpID, Nombre) as (  
  select EmpID, Nombre  
  from Empleados  
  where EmpID = 12345  
  union all  
  select e.EmpID, e.Nombre  
  from Empleados e join Emps em  
  on e.MgrID = em.EmpID )  
select *  
from Emps
```

El objetivo de la consulta es, a partir de una tabla de empleados, en la que cada uno de los registros hace referencia a otro registro de la misma tabla (el "jefe" del empleado), averiguar cuáles empleados trabajan directa o indirectamente a las órdenes del empleado número 12345. Primero se define una "expresión" que devuelve una tabla, por medio de la cláusula **with**, y luego se escribe una consulta usando esa expresión recién definida. Esta

última es muy sencilla, pero la complicación está en la expresión, bautizada Emps en el ejemplo, porque utilizamos una definición recursiva. Primero definimos una base o "semilla": los registros cuyo identificador vale 12345; naturalmente, hay un solo registro con este identificador. La siguiente subconsulta obtiene una nueva generación de registros, a partir de los registros ya encontrados para Emps. Observe que esa segunda subconsulta es la que contiene la referencia recursiva.

Pero la noticia que más gracia me ha hecho, es que SQL Server 2003 añade un nuevo nivel, o más bien modalidad, de aislamiento de transacciones: el modo snapshot, es decir, *instantánea*. La técnica consiste en crear versiones de registros para evitar bloqueos de lectura en transacciones de larga duración... lo que suena sospechosamente a InterBase. No es mala idea, por supuesto, sino todo lo contrario. De hecho, Oracle hacía algo parecido para las transacciones serializables creadas con el BDE,

aunque con el inconveniente de que la transacción debía ser de sólo lectura, y que era la única forma de lograr ese nivel de aislamiento, al menos con el BDE.

NOTA: Para los aficionados a la conspiranoia y los que se entretienen pintando a Billy con cuernos, cola y tridente, ahí va la última: ¿por qué la obsesión de Microsoft con las siglas XP? ¿Qué significa XP? Buah, a un conspiranoico nadie le podrá convencer de que XP viene de "eXPerience". Sin embargo, la "X" en griego equivale a nuestra "J", al igual que en cirílico, y la "P" significa realmente una "R", en estos dos alfabetos. Juntando las dos letras, tenemos los dos primeros caracteres de "Cristo" en griego: *xi-ro*. Este símbolo, junto con el pez, fue muy utilizado por las primitivas comunidades cristianas como señal de identidad. ¿Estará Guillermito jugando a ser el Anticristo? Además, eso de "Longhorn", cuerno largo, aunque dicen que es el nombre de un bar entre montañas, suena a aquelarre. ¿No será que...?

DATA SNAP VERSUS ADO.NET

Lo prometido es deuda: vamos a comparar las características de DataSnap (para Win32) y ADO.NET. El objetivo principal no será decidir cuál sistema es el mejor... aunque le servirá a usted para hacerse una mejor idea, sino facilitar la migración de aplicaciones desde DataSnap a ADO.NET. Por supuesto, no podremos ver de golpe todas las técnicas en juego: necesitaríamos evaluar también las técnicas de enlace a datos (*data binding*), por ejemplo, o las técnicas de acceso remoto. Pero incluso el viaje más largo comienza con un simple paso... ¡siempre que no sea un traspie!

DATA SET VS. TCLIENTDATASET

Una de las diferencias es que TClientDataSet representa una sola tabla, mientras que DataSet contiene una o más tablas. Para representar una relación maestro/detalles, DataSnap utiliza conjuntos de datos anidados, mientras que ADO.NET utiliza tablas, dentro de un mismo DataSet, relacionadas entre sí mediante objetos DataRelation. Resulta que el sistema de ADO.NET es mucho más eficiente cuando se trata de leer registros para estas estructuras complejas: la recuperación de los registros maestros y de detalles tiene lugar a través de dos consultas independientes: la relación se establece una vez que todos los datos están en memoria. En cambio, para alimentar un conjunto de datos anidado de DataSnap hay que utilizar, como origen de la información, conjuntos de datos relacionados como maestro y detalles: cada vez que se recupera una fila maestra, es necesario abrir y cerrar la consulta de detalles. Como comprenderá, esto es sumamente ineficiente. La solución que se le ha dado tradicionalmente a este problema de DataSnap es simular el sistema implementado por ADO.NET: leer dos consultas separadas y montar la relación sólo en el lado cliente. El problema con este truco es que sólo es recomendable para conjuntos de datos de sólo lectura. Al violentar la filosofía de diseño de DataSnap, las actualizaciones sobre estas relaciones ensambladas en el cliente se complican extraordinariamente: hay que crear métodos de grabación *ad hoc* en el servidor de capa intermedia, y tomar el mando sobre el control transaccional, el orden de grabación, la conciliación de errores, etcétera. DataSnap, por desgracia, no cede el control con facilidad en estos casos.

Pero la principal diferencia está en que DataSet no es el encargado de mantener la posición del registro activo, como sí sucede en Delphi. Se trata de un viejo error de diseño: un dataset en Delphi representa, simultáneamente, una colección de registros y a la vez, un registro: el registro activo. No existe en Delphi una forma sencilla de guardar un registro para su posterior utilización... porque en Delphi no existe el propio concepto de registro como entidad independiente. Tengo un componente que implementa un control *list view* virtual, que en vez de almacenar la información sobre su contenido como objetos redundantes en memoria, extrae de un conjunto de datos la información necesaria para dibujarse en pantalla. Como en pantalla se muestran varios registros a la vez, puede ser que el control solicite información

sobre el registro de la primera línea y el de la última línea. Esto implica que la posición del cursor activo del conjunto de datos debe dar saltos constantemente (disparando los eventos correspondientes). La única forma que tuve para resolver el problema fue clonar el conjunto de datos, y utilizar la versión clonada para extraer la información necesaria para el control.

En ADO.NET las cosas son muy diferentes. Las tablas se almacenan en objetos DataTable, y esta clase tiene una propiedad Rows, que contiene una colección de filas, donde cada fila es un objeto DataRow. Como prueba, para comprobar lo fácil o difícil que pueda ser la programación para .NET en general, me he lanzado a escribir un componente de rejilla sencillo, por el momento de sólo lectura. Aunque comencé desde cero, en dos días tenía implementado un control bastante decente y con toda la funcionalidad que me había puesto como objetivo. Cuando necesitaba los valores de las columnas de la fila cincuenta y cuatro, indexaba la propiedad Rows y punto.

¿Dónde entonces se guarda la posición activa del cursor? En .NET, esta posición es responsabilidad de los controles de Windows Forms. El mecanismo puede parecer complicado a alguien como yo, que lleva años trabajando con Delphi. Y realmente es complicado. Pero ofrece una ventaja, como compensación: todo el sistema de enlace a datos no está limitado a utilizar conjuntos de datos como origen de datos. .NET permite, por ejemplo, que utilicemos vectores con objetos pertenecientes a clases definidas por el programador como origen de datos. Hay una serie de tipos de interfaz que son la base de la técnica: si implementas los más sencillos, tendrás orígenes de datos de sólo lectura; implementando interfaces más avanzadas, se logra que estas estructuras puedan ordenarse o incluso editarse.

.NET DATA PROVIDERS VS. DB EXPRESS

Aunque la versión 2 de .NET añadirá funcionalidad a algunos proveedores .NET, en estos momentos tanto los proveedores .NET como DB Express tienen características similares. Ambos sistemas permiten ejecutar sentencias SQL arbitrarias en un servidor, y manejar cursores unidireccionales de sólo lectura.

La principal diferencia es la forma en que se programa un nuevo proveedor .NET o un nuevo controlador DB Express. En el primer caso, el modelo de programación es muy sencillo, pues se trata de implementar ciertos tipos de interfaz mediante clases, y en muchos casos se nos permite heredar de clases que ya contienen código útil. Además, existe suficiente documentación al respecto, incluso en la ayuda del SDK. Por el contrario, los detalles sobre cómo implementar un controlador DB Express son bastante escasos, y el modelo de programación de estos controladores es mucho más primitivo.

Este punto que voy a comentar puede resultar más polémico: una pequeña diferencia entre ambos sistemas consiste en que .NET no

ofrece nada parecido a la propiedad `Active` de las conexiones y conjuntos de datos de DB Express. Y mi opinión es que resulta más sencillo trabajar sin esta propiedad... aunque una de las quejas más frecuentes de las personas que llegan a .NET desde Delphi es ésta, precisamente. Yo mismo pertenecí a este grupo durante mucho tiempo. Pero al trabajar con proyectos grandes se hace difícil controlar que todas las conexiones y conjuntos de datos queden cerrados en tiempo de diseño, y al final terminas maldiciendo la existencia de esta propiedad. Y no menos importante es el hecho de que la implementación de `Active` en DB Express es bastante chapucera.

IDBAdapter vs. TDataSetProvider

A riesgo de simplificar demasiado, se puede decir que ADO.NET es un sistema más abierto y flexible que DataSnap, y es en este área donde se ve mejor esta afirmación.

Para empezar, DataSnap no permite controlar el orden en que se graban las filas actualizadas desde la capa intermedia. Hay un algoritmo enterrado en la unidad `Providers` que, a partir de un `TClientDataSet` anidado, crea un árbol de entidades que se recorre durante la actualización. Esto es necesario para tener en cuenta las posibles relaciones de integridad referencial entre registros. Por ejemplo, cuando hay nuevos registros en una relación maestro/detalles, es necesario insertar primero los registros maestros, antes de insertar los registros de detalles. En cambio, habría que eliminar primero los registros de detalles antes de intentar eliminar un registro maestro. En DataSnap, todo este algoritmo es automático... y rígido, porque está tan mal diseñado que es casi imposible realizar cualquier cambio en una clase derivada de `TDataSetProvider`. ADO.NET, por el contrario, exige al programador que se ocupe del orden de grabación de registros. ¿Complicado? Resulta que no: se puede lograr con un par de instrucciones. Existen métodos para devolver los registros de una tabla que están en determinado estado dentro de un vector de registros. Con la ayuda de estos métodos, es muy sencillo indicar el orden de grabación que deseamos. La mayor dificultad está en saber que es esto, precisamente, lo que se espera que hagamos.

Algo parecido sucede cuando se trata de la grabación de un registro individual. DataSnap tiene un complicado algoritmo para crear una instrucción SQL a partir de un registro que contiene su estado original. ADO.NET, en cambio, asume que seremos nosotros quienes le digamos cómo se inserta, elimina o modifica un registro... aunque existen clases, los famosos *command builders*, que hacen más o menos lo mismo que DataSnap (generan la instrucción). Estoy seguro de que mucha gente diría, sin pensarlo, que prefieren el sistema de DataSnap. Lo malo es cuando te ves en una situación como la que padecí en carne propia cuando intenté utilizar DB Express para Oracle 9 por primera vez. DataSnap generaba instrucciones disparatadas al confundir el tratamiento del separador decimal en el servidor (aparte, el propio controlador DB Express estaba plagado de errores).

¿Más? Suponga que vamos a insertar registros en una tabla de SQL Server, y que la clave primaria de ésta ha sido marcada con el atributo `identity`. En DataSnap tendríamos que interceptar el evento `AfterUpdateRecord` para, una vez que el propio DataSnap hubiese insertado el registro, recuperar el valor asignado a la identidad. Dos envíos de instrucciones al servidor. En ADO.NET, en cambio, podríamos agrupar la inserción y la recuperación de la

DATA SNAP VERSUS ADO.NET

Y seguimos con las comparaciones. Pero esta vez en la balanza no estarán Delphi y .NET, sino las distintas técnicas de acceso remoto en esta última plataforma. Tenga en cuenta que, más que cuestiones de lenguaje, vamos a tratar con características que otros tiempos hubieran encajado mejor dentro de una explicación del sistema operativo.

identidad en un mismo lote de consultas. Resultado: mayor velocidad. En ADO.NET v2, incluso existen opciones para agrupar todas las instrucciones de modificación en un mismo lote de consultas, aunque se trate de registros diferentes. Y esta filosofía de diseño flexible se deja notar en muchas otras técnicas, como la conciliación de grabaciones o el tratamiento de errores del bloque optimista.

En DataSnap, el mismo proveedor que se utiliza para leer datos es el que se utiliza para la grabación. En realidad no se trata del mismo objeto, porque la lectura y la escritura pueden ser manejadas por módulos diferentes, al reciclarse los módulos remotos. Pero el `TDataSetProvider` que se utiliza para la grabación tiene la misma configuración (¡el mismo nombre!) que el utilizado para la lectura. ADO.NET se aparta radicalmente de este diseño, y mientras el programador no se dé cuenta, no estará aprovechando todas las posibilidades de la herramienta. Imagine, por ejemplo, que tiene que leer un registro de cliente con todos sus pedidos asociados. Al estilo "DataSnap", necesitaría dos adaptadores de datos, tanto para la lectura como para la escritura. Supongamos ahora que el servidor relacional es SQL Server. En el mejor estilo .NET, en cambio, podríamos utilizar un solo adaptador para las lecturas, y dos adaptadores más para las escrituras. El adaptador de lectura contendría, en vez de una sola consulta, que es lo tradicional, las dos consultas: la de clientes y la de pedidos! Al tratarse de un lote de instrucciones, esto nos daría mucha más velocidad. Por supuesto, al grabar tendríamos que regresar al sistema de dos adaptadores: uno para las grabaciones de clientes y otro para las de pedidos.

RESUMIENDO...

En pocas palabras: ADO.NET ofrece mayor flexibilidad y más posibilidades de configuración a cambio de ser algo más complejo que DataSnap. No obstante, mi impresión es que, si hay que partir de cero en los dos casos, es más sencillo aprender ADO.NET que DataSnap. El motivo es que con ADO.NET usted puede ver lo que está pasando en cada momento; DataSnap, por el contrario, tiene grandes secciones de código que son cajas negras. Sólo la lectura del código fuente de DataSnap podría aportar algo de información sobre lo que sucede en esas zonas. ¿Un ejemplo? El componente `TDataSetProvider`, de DataSnap, tiene una propiedad `Options`, y una de las opciones que ésta contiene se llama `poAutoRefresh`. ¿Qué hace esta opción? Resulta que la ayuda no se cansa de repetir, desde hace no sé cuántas versiones, que sirve para leer la fila recién grabada; en ADO.NET eso se ve claramente en las instrucciones SQL generadas por el asistente de los adaptadores de datos. Pero si entra usted en el código fuente de la unidad `Provider` y busca `poAutoRefresh` en el texto... ¡verá que la opción nunca ha sido implementada!

Por supuesto, queda mucho por comparar. Por ejemplo, me he dado cuenta de que la mayoría de las preguntas de los programadores que pasan de Delphi a .NET es ¿dónde están los módulos de datos? Pues bien, ¡no necesitamos módulos de datos en .NET! No estoy recomendando que mezclemos código de presentación y de manejo de datos. Dios me guarde de ello. Pero las técnicas que se utilizan en .NET son incluso más "ortodoxas", más "orientadas a objetos" que los módulos de datos globales de Delphi. En el próximo boletín seguiremos con este tema.

Comencemos por las técnicas que, propiamente hablando, se ocupan del acceso remoto:

- **.NET REMOTING**

Es la técnica más rápida y la más expresiva, aunque para esto debe sacrificar la portabilidad con otros sistemas. Las conexiones predefinidas en .NET Remoting pueden estable-

cerse a través de zócalos TCP/IP, y por medio de HTTP; está claro que el primer tipo de canal es mucho más eficiente. Sí, es cierto que HTTP utiliza zócalos TCP a través del puerto 80, pero impone reglas a un nivel superior para el protocolo, que implican la transmisión de cabeceras y otras estructuras adicionales. Sobre estos *canales* (terminología oficial) se pueden enviar objetos y referencias a objetos en dos formatos predefinidos: un formato binario propio, o SOAP. Canales y formatos pueden combinarse a placer: es posible, por ejemplo, transmitir un objeto serializado en formato binario a través de un canal HTTP. No obstante, la técnica más eficiente es combinar un canal TCP con el formato de serialización binario. Y por supuesto: podemos definir nuestros propios canales y formatos (aunque la cercanía de **Indigo** aconseje prudencia).

Físicamente, un servidor de .NET Remoting puede residir en una aplicación de servicio, o alojarse dentro de Internet Information Services. En este último caso, estaríamos obligados a usar un canal HTTP, aunque mantendríamos la libertad de elección del formato de serialización, y ganaríamos las funciones de seguridad de I.I.S.

- **XML WEB SERVICES**

Los servicios Web permiten la comunicación con otras plataformas, siempre que no intentemos salirnos demasiado del guión. En determinados casos, son más fáciles de programar, pero a cambio de ser mucho más lentos, por lo general, y de permitir menos alegrías con el sistema de tipos: menos fidelidad en la transmisión de tipos, un modelo de objetos muy limitado, el protocolo HTTP impide definir *callbacks*...

En el lado positivo, los servicios Web implementados sobre I.I.S. disfrutan automáticamente de todas las funciones de seguridad admitidas por este servidor. También pueden aprovechar la implementación de SSL, para cifrar el tráfico entre el servidor y sus clientes. En este caso, servicios Web implementados vía ASP.NET, podemos incluso definir servidores "con estado": al igual que ocurre con una aplicación Web "normal", podríamos utilizar *cookies* con el servicio, e implementar variables de sesión en el lado servidor. Con el uso de *cookies*, por ejemplo, podríamos simplificar la recuperación de consultas grandes mediante grupos de registros, sin complicarle la vida al cliente (esta técnica se explica en **La Cara Oculta de C#**). Sin embargo, debemos saber que no todos los clientes reconocerán el uso de *cookies* por parte de un servidor Web. En el caso de clientes creados con .NET, el *proxy* con el que trabajan los clientes nos echa una mano en este sentido.

Una forma más estandarizada de aplicar trucos como los de las *cookies* son las cabeceras SOAP: información adicional que se permite enviar y recibir en paralelo con la información principal del servicio (también se tratan en el libro). Algunos servicios utilizan las cabeceras SOAP adicionales como un elemento opcional, aunque podemos exigir que el cliente las comprenda y utilice. Más adelante mencionaré algunas otras extensiones soportadas por los servicios Web en la plataforma .NET.

NOTA: .NET Remoting ofrece una técnica parecida a las *cookies* y cabeceras SOAP. Por medio de los métodos estáticos *GetData* y *SetData*, de la clase *CallContext*, podemos incluir información que viajará en las dos direcciones en cada llamada remota.

Resumiendo: en un proyecto nuevo, sobre usted caerá toda la presión social para que la capa intermedia se implemente como un servicio Web. ¿Motivo?, principalmente porque están de moda.. aunque la funcionalidad de un servicio Web es bastante limitada, y su rendimiento no puede compararse al de .NET Remoting. Pero si puede, convenga a su cliente (o a su jefe) para que la aplicación pueda utilizar los dos mecanismos de comunicación remota. Esto no es tan complicado como puede parecer: en el lado cliente, sólo hace falta definir una interfaz común a los dos

tipos de servicios, y encapsular los respectivos *proxies* dentro de clases que implementen la funcionalidad común.

¿Y qué sucede en el lado servidor? Está claro que podríamos aplicar la misma técnica: crear una clase común que implementase la funcionalidad requerida, y utilizar sencillos adaptadores para enchufar esta clase en las clases concretas que actuarían como clases remotas. Pero es interesante ver cómo **COM+** puede ayudar en esta dirección. Aunque no se recomienda el uso de COM+ como mecanismo de acceso remoto (para eso están .NET Remoting y Web Services), sí podemos aprovechar los restantes *servicios* de COM+, implementando la funcionalidad común como un *servicio corporativo*:

- **ENTERPRISE SERVICES**

Las clases derivadas de *ServiceComponent*, una clase residente en el espacio de nombre *EnterpriseServices* pueden registrarse dentro del Catálogo de COM+, para aprovechar los servicios de este subsistema. De todos modos, no se recomienda que estos objetos se utilicen directamente como objetos "remotos" (aunque COM+ permite habilitar SOAP para estos objetos de forma automática). Entre los servicios que nos interesa aprovechar tenemos la **caché de objetos**, para limitar el número mínimo y máximo de instancias del objeto en el servidor, las **transacciones declarativas**, que es la forma más sencilla de trabajar con operaciones que afectan a más de una base de datos o servidor, la **activación puntual**, las **cadena de construcción**, etc.

El diseño recomendado, cuando se utilizan los servicios corporativos de COM+, es implementar la funcionalidad remota dentro de estas clases, para aprovechar el manejo eficiente de recursos, y utilizar servicios Web o .NET Remoting como fachada para la comunicación remota con estas clases.

Otro componente del sistema operativo que puede resultar interesante es el conocido como **Message Queuing**, o *colas de mensajes*, que antes se identificaba mediante las siglas **MSMQ**. El uso de colas de mensajes internamente dentro del servidor sirve para "aplanar" el gráfico de carga de trabajo del servidor. Ciertas operaciones que no requieren una respuesta o confirmación inmediata, pueden diferir su ejecución mediante una cola de mensajes.

Por último, si va a trabajar con servicios Web, es necesario tener presente ciertas extensiones, que de momento son de tipo experimental:

- **WEB SERVICES ENHANCEMENTS**

Esta es una propuesta de estándar formulada por un equipo compuesto por IBM y Microsoft. La parte más interesante, en mi opinión, es la especificación **WS-Security**, que propone técnicas para la autenticación del usuario y para el cifrado del tráfico. Como propuesta, WS-Security no se inmiscuye en la forma concreta en que tiene lugar la autenticación en el lado servidor, sino que su función es indicar cómo se transmite la información sobre usuario y contraseñas en las peticiones SOAP.

Lamentablemente, mientras el estándar no sea aceptado e implementado por más partes, estas extensiones se limitan a sistemas en los que el cliente y el servidor se han programado dentro de .NET.

NOTA: Si tiene que empezar un proyecto de este tipo a corto plazo, puede que le interese el próximo proyecto de investigación de IntSight. Su propósito es crear lo que internamente hemos llamado un *servidor universal de capa intermedia*. Este es un software genérico que podría configurarse indistintamente como Web Service o como servicio de .NET Remoting, y debería permitir la recuperación de entidades y su posterior actualización mediante métodos remotos genéricos. La configuración de las entidades admitidas, y de las reglas de empresa a verificar, tendría lugar de forma declarativa, mediante ficheros de configuración XML. Ade-

más, el servidor facilitaría al cliente la paginación de resultados, o la recuperación de registros individuales con el propósito de obtener las versiones más recientes de estos. Finalmente, existiría la posibilidad de implementar diversas técnicas de caché para los resultados, con la ventaja de poder contar con un punto central para su implementación.

El objetivo final de este proyecto es acelerar significativamente el desarrollo y mantenimiento de este tipo de aplicaciones. En realidad, la idea original del proyecto surgió teniendo DataSnap en mente. Por supuesto, llegar a resultados prácticos con este tipo de proyectos exige un esfuerzo considerable en investigación. En

A NADIE LE PREOCUPA EL UNIVERSO...

En el último boletín incluí información real sobre un suceso sísmico preocupante, sin explicación hasta el momento... y, aunque recibí muchos mensajes preguntando sobre Delphi y .NET, inadie se preocupó por la salud del Universo! Es cierto que la historia apareció primero en la BBC... pero seamos justos: a veces la BBC dice la verdad, aunque ahora mismo no recuerde cuándo fue la última vez.

Borland ha anunciado ya la disponibilidad de Delphi 8. Y ya ha comenzado el bombardeo comercial para convencer a todos de que Delphi 8 no sólo es el mejor entorno de programación, sino que además cura a los enfermos y socorre a los desvalidos, sobre todo cuando se le dirigen oraciones en época de Luna Nueva. En particular, me causa mucha gracia la *feature matrix* oficial de Delphi.NET... porque la mayoría de las ventajas que enumera son las características genéricas de cualquier lenguaje que compile código IL; ventajas, por cierto, que salen más baratas al usar Visual Studio.

No debo, y no quiero, pronunciarme oficialmente sobre este tema mientras no exista una versión pública de evaluación de Delphi.NET; estas líneas están utilizando la información hecha pública por la propia Borland en sus páginas de Internet, el artículo sobre Delphi.NET aparecido en el número de diciembre de la Delphi Magazine, el *preview* (bastante incompleto) del compilador incluido en Delphi 7... y la mala experiencia con C#Builder. La publicidad, como era de esperar, habla de todas las cosas buenas y de ninguna de las "posibles" cosas malas. Pero como autodefensa contra los vendedores de aceite de serpiente, se me ocurren algunas preguntitas inocentes que podría plantear a estos señores:

- ¿Qué soporte ofrece Delphi.NET para la Compact Framework?
- En particular, ¿funcionaría la VCL sobre un PDA? ¿Y qué pasaría en ese caso con DB Express, y DataSnap?
- ¿Qué nuevos asistentes se han añadido desde la comercialización de C#Builder? Porque en esta versión, la ausencia de buenos asistentes que siquiera la igualaran a Visual Studio, era clamorosa.
- ¿Se han corregido todos los errores de DataSnap que existían en la versión 7? Que nadie le diga que no existen, porque mi enfado en gran parte es culpa de la enorme cantidad de *bugs* que he encontrado en Delphi 7, para los cuales Borland no se ha dignado a publicar un *Update Pack*. Y que nadie intente restarle importancia: muchos de estos *bugs* pueden agravarse al cambiar el modelo de manejo de memoria de Win32 a .NET.
- ¿Soporta Delphi.NET la instrucción **foreach** de C# y VB.NET? Si no la soporta, cualquier mínima iteración en C# se convertirían en líneas y líneas de código Delphi. Tomemos como punto de partida este fragmento de código:

la primera quincena de enero de 2004, anunciaremos la posibilidad de participar en este proyecto como *patrocinador*: por una inversión más o menos equivalente al coste de uno de nuestros cursos a distancia, su empresa podría tener acceso privilegiado e inmediato al código fuente, e influir en su desarrollo. La principal ventaja sería, como he dicho, disponer en poco tiempo de una sólida base o punto de partida para la clase de proyectos más demandada en un futuro cercano y previsible. Estamos barajando todavía algunas variantes y posibilidades de actuación, pero estamos abiertos a otras ideas o propuestas.

```
foreach (DataRow dr in tbClientes.Rows)
    Procesar(dr);
```

Si Delphi 8 no incluyese finalmente soporte para **foreach**, tendríamos que traducir la instrucción anterior de la siguiente manera:

```
var
    e: IEnumerator;
    d: DataRow;
begin
    e := tbClientes.Rows.GetEnumerator();
    try
        while e.MoveNext do
            begin
                d := e.Current as DataRow;
                Procesar(d);
            end;
        finally
            if e is IDisposable then
                (e as IDisposable).Dispose;
        end;
    end;
```

- ¿Soporta Delphi.NET algo parecido a la instrucción **using** de C#? ¿O hay que simularla también mediante **try/finally** y llamadas explícitas al método `Dispose`?
- ¿Cuánto dinero necesitará gastar para poder crear aplicaciones decentes? Porque un servidor, es decir *mois*, se las apaña muy bien con Visual C# Standard 2003, que se puede conseguir por algo menos de 150 euros. Y por otros 50 euros más, se puede conseguir una versión *completa* de SQL Server para desarrollo (sin claves que expiran a los tres meses o zarandajas por el estilo).

Advierto que estoy dispuesto a dejarme convencer; eso sí, con argumentos serios. Porque creo que la competencia es buena para todos, y Java no está en condiciones de competir con .NET (a no ser que se sometiese a una cirugía estética radical).

Por cierto, la pregunta que me está llegando con mayor frecuencia en estas fechas es, más o menos: "*¿me serviría La Cara Oculta de C# para programar con Delphi.NET?*". La respuesta: un 90% sí, un 10% no. La diferencia entre lenguajes es bastante grande, pero si está usted dispuesto a traducir mentalmente el código, el libro le puede ser muy útil. Tenga en cuenta que todos los lenguajes .NET comparten las mismas bibliotecas de clases. Lo que no encontraría en este libro es cómo migrar un proyecto desde Delphi 7 para que use la VCL.NET y el sucedáneo de DB Express/DataSnap implementado en Delphi 8. Pero tampoco le recomendaría esta vía... excepto en circunstancias extremas, y como medida heroica.

BOLETÍN TÉCNICO #1 - 1

Impresiones sobre Delphi 7 1
Un enlace de datos universal 1
Limitar el número de filas en Oracle 2
InterBase 7 2

BOLETÍN TÉCNICO #2 - 4

WebSnap y ADO 4
Módulos de datos transaccionales 4
Acciones de filas en WebSnap 5
Libros recomendados 6

BOLETÍN TÉCNICO #3 - 7

Propiedades de tipo interfaz 7
Controles de listas virtuales 8
C#: el lenguaje 9
Bugs en servicios Web 10

BOLETÍN TÉCNICO #4 - 11

El tipo más rápido de cursor en ADO 11
Aplicaciones ISAPI: depuración 11
Aplicaciones ISAPI: mantenimiento 12
Componentes, recomendaciones, rumores... 13

BOLETÍN TÉCNICO #5 - 15

IntraWeb 15
Servicios endemoniados 16

Desconexión en ADO 18
La esquina de las malas lenguas... 18

BOLETÍN TÉCNICO #6 - 19

Pérdidas de memoria con DLLs 19
Proyecto Snowball 19
Acumuladores de cadenas 21
InterBase 7.1, cursos para FireBird 22

BOLETÍN TÉCNICO #7 - 23

C# y las excepciones 23
Automatizar cambios de cursor 24
Cursores de pantalla en .NET 25
Desde el frente de batalla 26

BOLETÍN TÉCNICO #8 - 27

Conmoción en el Universo Delfico 27
Propiedades dinámicas 28
Extensores de propiedades 29

BOLETÍN TÉCNICO #9 - 30

Cuerno Largo 30
DataSnap versus ADO.NET 31
DataSnap versus ADO.NET 32
A nadie le preocupa el Universo... 34

INDICE - 35