SISTEMA DE MONITORIZACIÓN DE UNA PLANTA FOTOVOLTAICA CON ACCESO REMOTO UTILIZANDO INTERNET

Daniel Barroso Tristán

Proyecto de Fin de Carrera

Presentado en el Departamento de Ingeniería Electrónica (Área Tecnología Electrónica) de la Universidad de Sevilla, como parte de los requisitos para obtener el título de

INGENIERO INDUSTRIAL

ESPECIALIDAD ELÉCTRICA

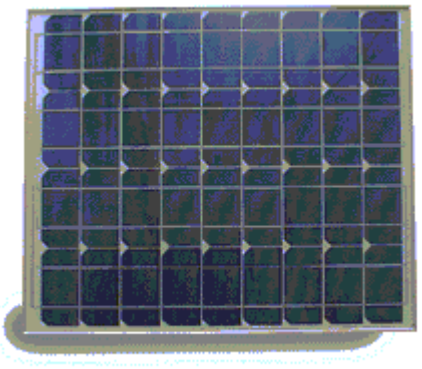
Director del Proyecto: DR. D. JUAN MANUEL CARRASCO SOLÍS

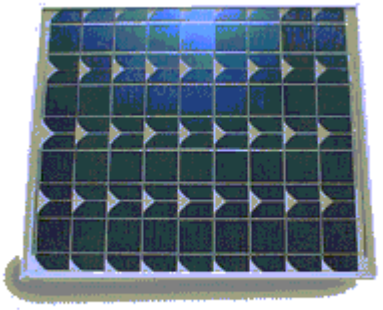
Dpto. de Ingeniería Electrónica
Escuela Superior de Ingenieros. Universidad de Sevilla.

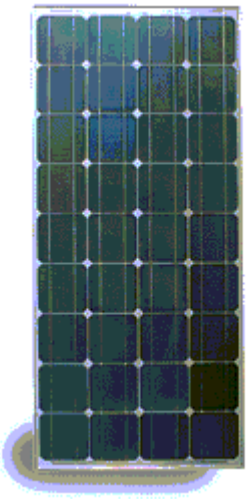
Índice

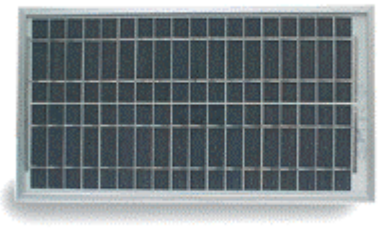
"We should treat all the trivial things of life seriously and all the serious things of life with sincere and studied trivialty"

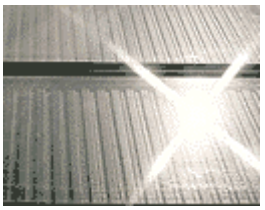
Oscar Wilde sobre *"The Importance of Being Earnest"*

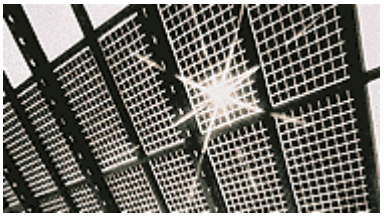


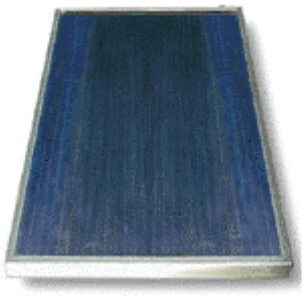


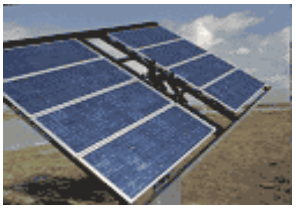












1. Memoria Descriptiva

1. OBJETIVO Y MOTIVACIÓN DEL PROYECTO.

1.1. Contexto.

El presente trabajo se realiza como Proyecto de Fin de Carrera para la obtención del título de Ingeniero Industrial, impartido en la Escuela Superior de Ingenieros de Sevilla (Universidad de Sevilla). La dirección del proyecto ha sido llevada a cabo por el profesor D. Juan Manuel Carrasco Solís, perteneciente al Departamento de Ingeniería Electrónica (Área de Tecnología Electrónica) de la Universidad de Sevilla.

1.2. Marco. Descripción.

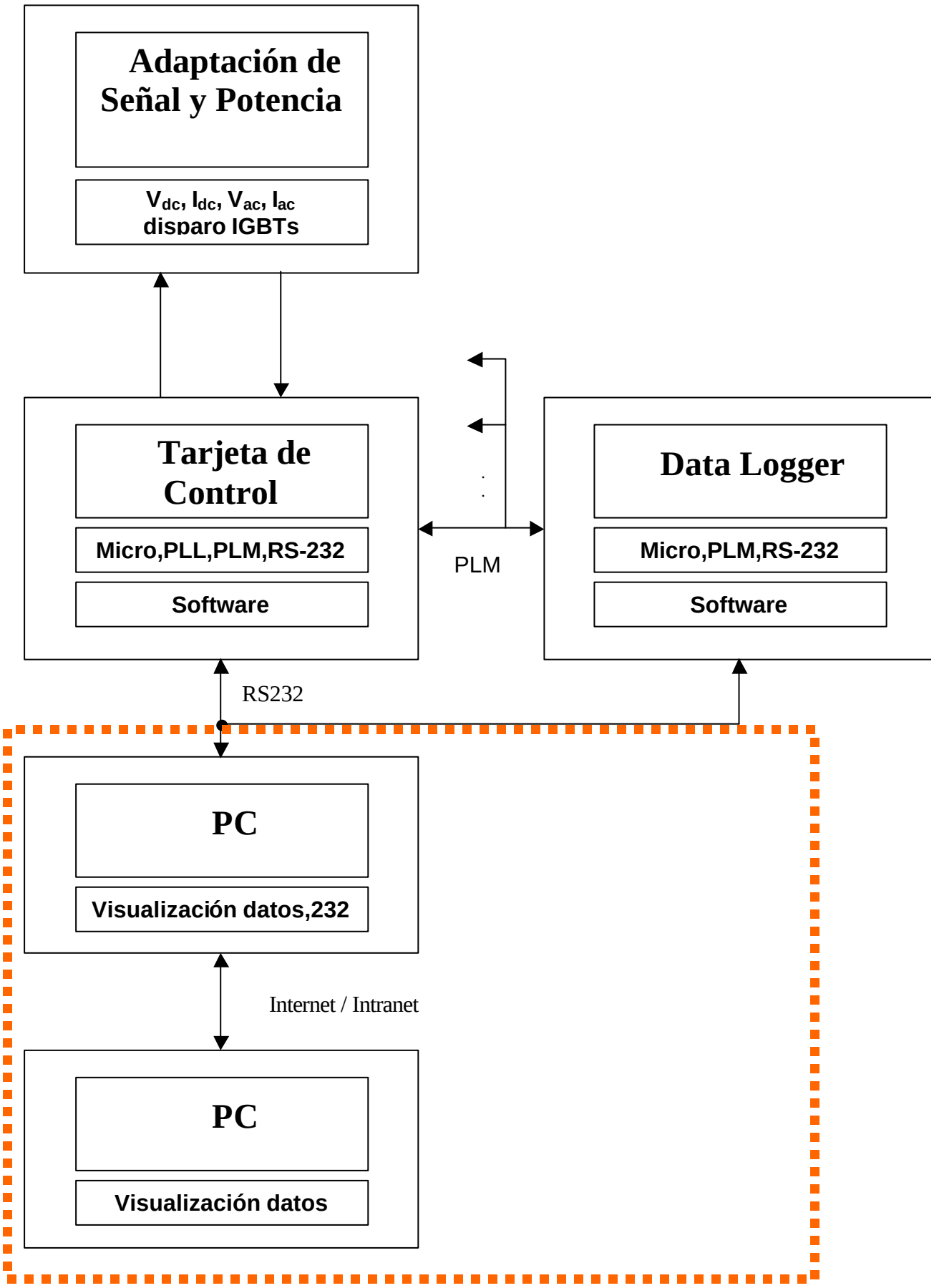
Este proyecto se encuentra inmerso dentro de uno de mayor tamaño: el control de la cesión de la energía eléctrica generada por un panel fotovoltaico a la red mediante un inversor basado en IGBT's.

El fin del mismo consiste en la creación de una base de datos en la cual ir almacenando los diferentes puntos de trabajo en que se desarrolle la conversión de corriente continua a corriente alterna, para su monitorización y un posible estudio posterior de los mismos, comprobando el funcionamiento del inversor frente a varios algoritmos de control y optimizándolo según los criterios pertinentes. Para ello se ha realizado un protocolo mediante el cual comunicar a través del puerto serie el servidor que aloja a la base de datos con el *data logger* o, en su caso, con la tarjeta de control encargada de manejar los diversos inversores dispuestos en paralelo. Se ha facilitado la visualización de estos datos en la misma aplicación que gestiona la base de datos así como su posterior tratamiento mediante herramientas matemáticas como *MATLAB*

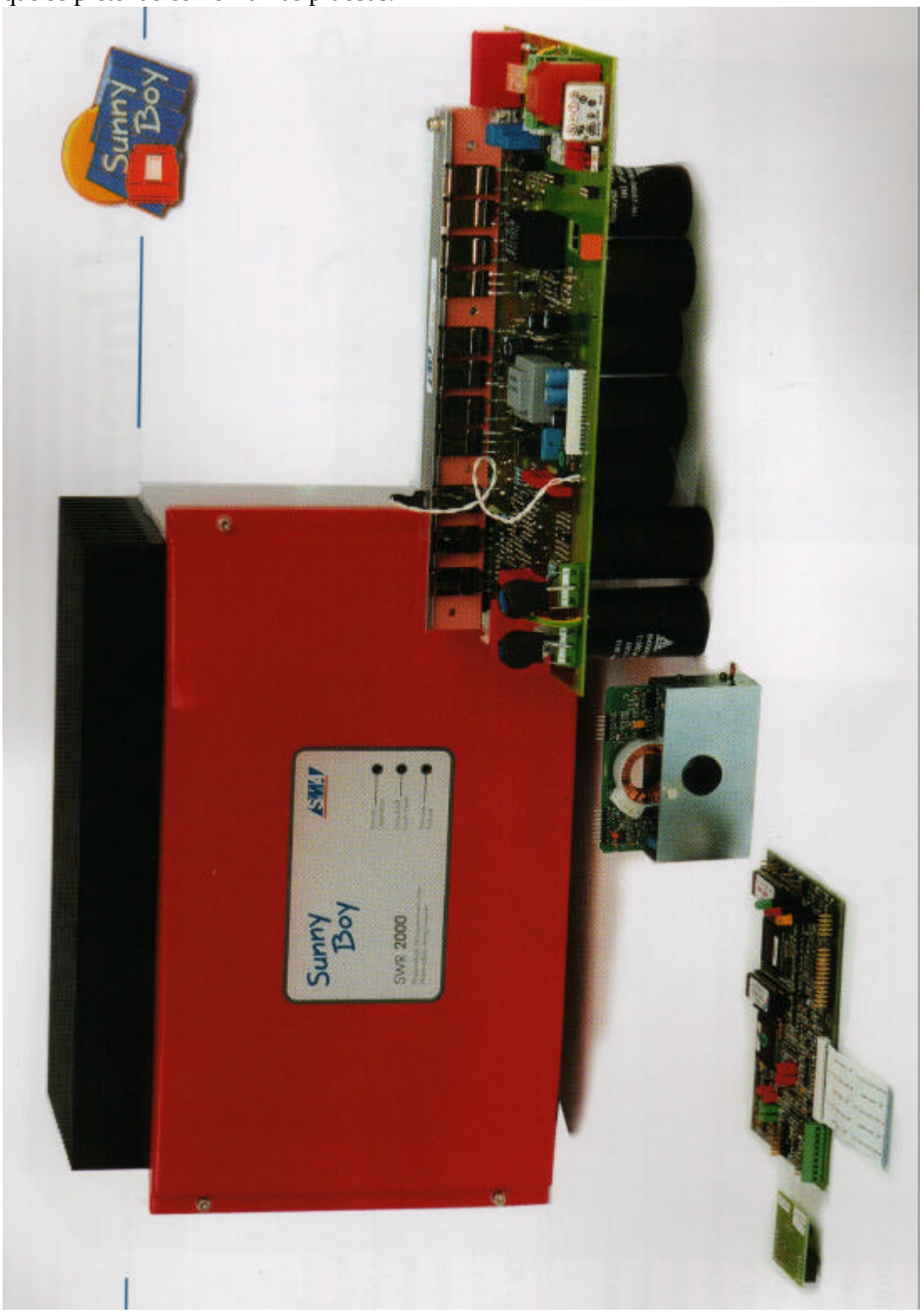
La base de datos se ha realizado utilizando una arquitectura cliente / servidor, de forma que sea accesible remotamente, ya sea en la red de área local implementada en el mismo laboratorio o bien desde cualquier punto del mundo empleando Internet, evitando de esta manera los posibles conflictos que pudieran surgir del intento de acceder simultáneamente a un mismo punto de la base desde varios puestos.

Por otra parte, también es posible enviar comandos al *data logger* desde la misma aplicación que gestiona la base de datos, de forma que entren en funcionamiento los inversores deseados y con un cierto límite de potencia.

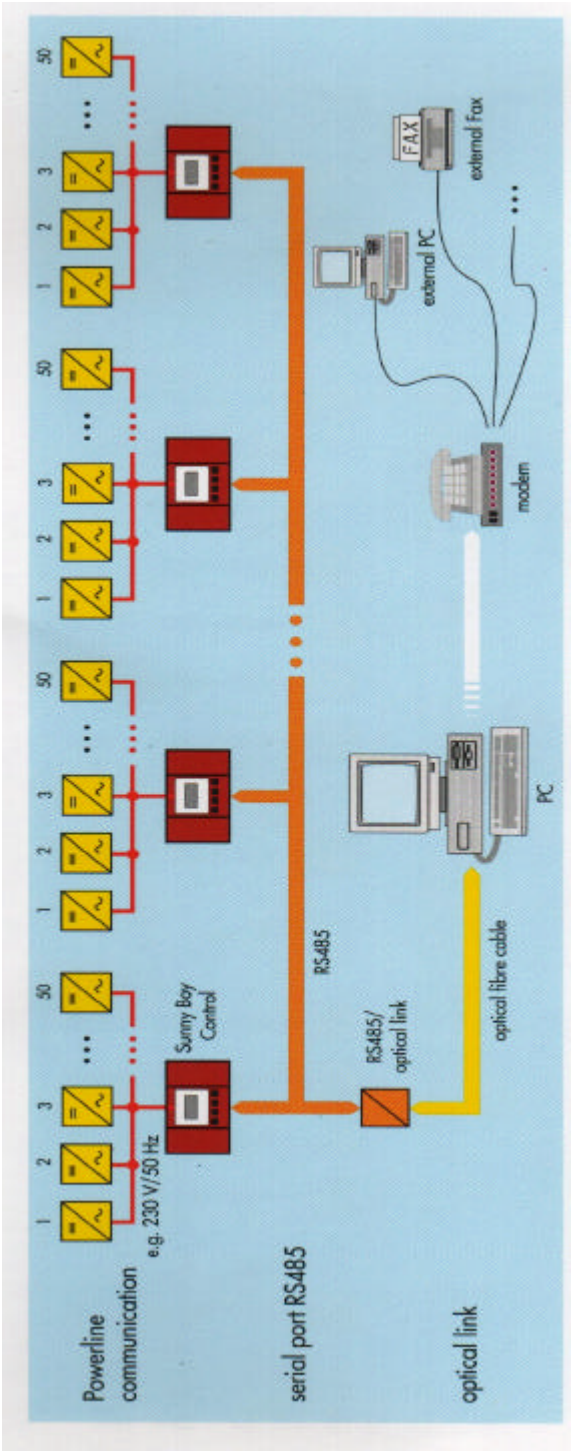
A continuación se presenta un esquema del proyecto global y se señala dónde se encuentra situado el presente trabajo.


Adaptación de Señal y Potencia
V_{dc}, I_{dc}, V_{ac}, I_{ac}
disparo IGBTs
Tarjeta de Control
Micro,PLL,PLM,RS-232
Software
PLM
Data Logger
Micro,PLM,RS-232
Software
RS232
PC
Visualización datos,232
Internet / Intranet
PC
Visualización datos

En la siguiente imagen podemos apreciar un detalle del *SunnyBoy*, inversor con el que se pretende comenzar las pruebas:



Aquí se presenta un diagrama con el funcionamiento de la planta de generación de energía eléctrica mediante los paneles fotovoltaicos:



2. ELECCIÓN DEL SOFTWARE DE DESARROLLO

2.1. ¿En qué lenguaje de programación debería desarrollarse el proyecto?

Aunque aparentemente la primera pregunta que deberíamos hacernos a la hora de desarrollar una aplicación de bases de datos debería ser en qué lenguaje realizarla, este asunto es relativamente secundario para el éxito del proyecto, siempre que este lenguaje no tenga limitaciones intrínsecas. Tras conversaciones con diversos integrantes del departamento y varias inmersiones en la *World Wide Web*, estudiando la disponibilidad de software la elección se decantó a favor del producto C++BUILDER de la compañía *Borland*, cuyos entornos de programación están particularmente bien preparados para el desarrollo de aplicaciones cliente / servidor y de múltiples capas.

A continuación se presenta una comparativa de este producto con el análogo de Microsoft (VisualC++):

	C++Builder 5	Visual C++ 6
Real Visual Development	Yes	No
Extensible OO Component Framework	Yes	No
Highest Level of ANSI C++ Language Support	Yes	No
Runtime memory debugging and leak detection	New! CodeGuard	No
Integrated XML Web Application Development	New!	No
Integrated Language Translation Tools	New!	No
Database Enabled ISAPI, NSAPI, and CGI Development	Yes	No
Drop-in Internet Protocol Components	Yes	No
ADO Express for fast connections to any Data	Yes	No
CORBA ORB	VisiBroker	No
Integrated CORBA Development	Yes	No
Multi-tier Database Services	Yes	No
Project Manager Format	New! XML	MS Proprietary
Distributed Object Interfaces	COM+ and CORBA	COM+
High-performance RDBMS Drivers	Both Native and ODBC	ODBC
Class Browsing	Yes	Yes
Delay Load DLLs	New!	Yes
Background Compilation	New!	Yes
Just-in-time Debugging	New!	Yes
ActiveX/ATL Control Creation	Yes - One Step	Yes
Microsoft Transaction Server Support	Yes	Yes
RDBMS SQL Server Included	Yes	Yes
Integrated Team Development	Yes	Yes
Visual Database Design	Yes	Yes
Microsoft Windows SDK	Yes	Yes
MFC Included	Yes	Yes

En las siguientes tablas podemos observar que las características de este entorno de programación son óptimas para trabajar con bases de datos y desarrollo de aplicaciones multicapa.

Database	BCC	Std	Pro	Ent
Open Database Architecture – Connect to all corporate data				
Enhanced! Borland Database Engine v5.10 for easy access to any database system			x	x
New! Support for autorefresh			x	x
New! TcustomConnection class to easily integrate 3rd party database solutions			x	x
Access, FoxPro, Paradox, and dBASE drivers for high-speed access to desktop and LAN databases systems			x	x
Complete ODBC connectivity			x	x
BDE API for direct open access to any database engine			x	x
SQL Links native drivers, with unlimited deployment license, for InterBase, Oracle, Sybase, Informix, MS SQL Server, and DB2				x

Advanced InterBase Support				
New! InterBase v5.6 support			x	x
Local InterBase (1 user license) for off-line scalable SQL development			x	x
InterBase event alerter component			x	x
InterBase NT (5 user license) to develop and test multi-user SQL Applications				x

InterBase Express High Performance Direct InterBase Data Access Components				
New! True native InterBase interface for tight integration			x	x
New! Unlimited deployment of InterBase Express			x	x
New! Advanced Transaction control			x	x
New! Comprehensive Database information			x	x
New! Transparent updates of complex datasets			x	x
New! Powerful SQL Component for the ultimate in speed			x	x
New! SQL Monitor component for advanced debugging			x	x
New! MIDAS provider for scalable distributed applications			x	x
New! Upgrade path to InterBase 6.0			x	x

High Performance MS SQL Server Support				
New! MS SQL Server 7 support				x
New! ADOExpress: Complete support for MS ADO for fast access to all types of information			*	x
New! ADO OLE DB and RDS Connection commponents for complete control over all aspects of information retieval			*	x
New! ADO Command component for controling data manipulation and definition			*	x
New! ADO DataSet component for maintaining complete control while navigating and updating data			*	x
New! ADO support for Tables, Queries and Stored Procedures to get up to speed quickly and migrate your existing C++Builder applications to ADO			*	x
* Borland ADOExpress and Borland TeamSource can be purchased separately for Borland C++Builder 5 Professional users on http://shop.borland.com/ (US only) or call your local Borland office for more details.				

Advanced Oracle 8I Support				
New! Oracle8i support				x
Nested Fields (ADT)				x
Array Fields				x
Reference fields				x

Multi-Tier Database Development Services - MIDAS Development Kit				
New! MIDAS 3 development license for developing and testing multi-tier database applications				x
New! XML data packet support				x
New! Stateless DataBroker for more control in mobile and low-bandwidth situations				x
New! WebConnection component to securely move your applications to the outside of the firewall				x
New! Server object pooling for enhanced scalability				x
New! CommandText property for building client side queries				x
New! Provider Options increase control over how and what information is transmitted				x
Master/Detail Provider and Resolver support				x
Business Object Broker for 24x7 fail-over support and load balancing				x
Exclusive: Remote DataBroker for distributed database connectivity and thin client database applications				x
Constraint Broker for data validation at the client				x
On Demand Blobs				x
End user login support				x
Transaction Resolver for transaction conflict resolution				x

BuisnessInsight - Data Visualization components				
Exclusive: Decision Cube CrossTab for multi-dimensional analysis of data				x
New! Decision Cube source code to learn from and extend your decision support applications				x
QuickReports to easily create, preview and print embedded reports			x	x
TeeCharts to visually turn data into information			x	x

También podemos valorar las posibilidades de integración en redes de área local e Internet:

Get to the Net	BCC	Std	Pro	Ent
Seamless Internet/Intranet Wizards and Components				
MS Internet Explorer		x	x	x
FastNet Native Internet Client Components (HTTP, FTP, SMTP, NNTP, TCPIP, POP3, and more)			x	x
FastNet TCP/IP Server Components			x	x
Exclusive: ActiveXForms for building Web applications			x	x
BDE CAB File for easy distribution of database apps over the web			x	x
New! Active Server Object Wizard for creating high-performance ASP Server Side Web Applications			x	x

WebBroker – Deliver the fastest web database applications				
WebServer Applications for high-speed, high-throughput web-delivered data applications			New!	x
WebBridge for a open solution that supports CGI, WinCGI, Netscape NSAPI and Microsoft ISAPI			New!	x
WebModules for centralized information publishing with live internet applications			New!	x
Enhanced: Preview with HTML4 support			New!	x
WebDispatch for seamless responses to web client requests			New!	x
WebDeploy for deploying thin-client, zero-configuration, applications using the Web			New!	x
Web Application Wizard			New!	x

New! Internet Express				
New! XML data from MIDAS Servers for flexible Web data exchange from any backend database				x
New! XML Broker for quickly providing XML data for web server applications from any backend database				x
New! MIDAS PageProducer wizard to migrate MIDAS applications to the Internet with XML and HTML4				x
New! Web Page Editor for instantly designing dynamic XML data driven HTML4 web documents				x

CORBA Distributed Object Development				
Enhanced! Visual TypeLibrary Editor – CORBA IDL Emitter				x
New! VisiBroker ORB v4.00				x
New! Allow building of CORBA 2.3 compliant servers and clients with full Valuetypes support				x
New! Portable Object Adapter (POA), providing sophisticated management of server object lifetimes, and allows server source code portability across ORB™ products				x
New! Objects By Value (OBV), for passing of arbitrary complex objects across processes, machines, and languages				x
New! Allow inter-operating with Borland Application Server 4, the leading Java J2EE deployment platform				x
Enhanced! Interface Repository (IR) for CORBA Object Interface and Type info storage and retrieval				x
Enhanced! ANSI/ISO CORBA C++-compliant interfaces for maximum source portability				x
Enhanced! CORBA Wizards for CORBA client and server development				x
Enhanced! Server object persistence model for greater scalability				x
VisiBroker Event Services				x
VisiBroker Naming Services				x
IDL integrated stub and skeleton compilation				x
CORBA Object Wizards				x
Two-Way CORBA IDL updates				x

BCC = Borland C++ 5.5 Compiler | Std = C++Builder 5 Standard | Pro = C++Builder 5 Professional | Ent = C++Builder 5 Enterprise

En las siguientes páginas se presentan las características más resaltables de InterBase y sus puntos fuertes:

InterBase Overview

InterBase® is the open source relational database that combines ease of use, low maintenance costs, and enterprise power. Since 1985 InterBase has provided the strength of a powerful, high-performance, proven architecture with the sophisticated technology that applications need to be successful. Today, InterBase leads the industry again, offering free development and distribution rights and moving
database use to the center of all application development.

Versioning Architecture for ultimate concurrency—readers never block writers.
Active database, including the most full featured trigger and stored procedure implementation.
Event Alerters—React to database changes without polling.
Exceptional ANSI SQL-92 compliance and full UNICODE support.
Rich data types—Blobs, multi-dimension-al arrays.
InterClient—all-Java JDBC driver for low maintenance.
Designed for business critical distributed database environments, InterBase provides power and flexibility for Internet, mobile, and embedded database applications.
Scalable from Windows 95/98, Linux, HP/UX, Solaris, and other UNIX systems.

In January 2000, Borland announced that the new version of the InterBase database—InterBase 6.0—would be open source. This is your chance to read about what InterBase is, and what it
does. You can download the latest version of the software and then use and test the 6.0 release to see if it meets your requirements.

Business-critical tasks like stock trading, pharmaceuticals, network management, and aerospace demand performance.
InterBase delivers their performance and remains practical and easy to deploy for small applications. Developers who want a sophisticated database with a small foot-print, low maintenance cost, and high reli-ability choose InterBase. They depend on InterBase to manage their customers' data
without the services of a highly trained and compensated DBA.
InterBase offers the best features of the database world—triggers, stored proce-dures, Blobs, event alerters, user defined functions, multi-dimensional arrays, two-phase commit, referential integrity, con-straints, and a flexible set of transaction options. These features reduce develop-ment
time and improve reliability.

Sophisticated Architecture Delivers Performance

The InterBase server implements a Multi-Generational Architecture [MGA]. This MGA provides unique 'versioning' capabilities that result in high data avail-ability for transaction-processing users and decision-support users simultaneously. Traditional database servers support the On-Line Transaction Processing (OLTP) model of database interaction, with many short, simple transactions. The InterBase MGA engine performs well on short OLTP-style transactions, but it excels in real world applications, outperforming other databases

because concurrent long-duration, decision-support transactions do not degrade its performance.

"InterBase . . . outperformed all other SQL databases that we tested; nothing else came close. Its peed and small foot-print, combined with its ability to handle data-intensive client/server applications, made it an easy choice for us."

–Russ LeMaster, Dover Elevator Systems

The versioning engine eliminates the need for transactions to lock the records they read, making them contention-free—

readers never block writers. Unlike other databases, InterBase provides a time-con-sistent, repeatable result for every query, without special programming. Because long and short duration transactions can coexist, the InterBase versioning engine maximizes throughput for all transactions.

Multi-threaded Architecture

The InterBase server adds a multi-threaded architecture to the MGA, improving per-formance

and optimizing the use of system resources, especially for large numbers of users. A shared server supports more clients on the same hardware and still improves the system responsiveness.

The multi-threaded architecture provides a shared data cache, reducing the amount of disk I/O for each application request. The server's shared metadata cache reduces the compilation costs for requests and makes procedures and triggers more efficient.

User and database statistics kept by the server are useful for diagnosing application hot spots.

Java Enablement

Java® and InterBase are a natural pair. Features that make Java intriguing—sim-plicity, robustness, portability, and flexibil-ity— are also characteristic of InterBase. Java applications access InterBase through InterClient, a high-performance JDBC driver. InterClient is an all-Java driver, which can be an applet—nothing is installed on the client. Deployment is simple, since configuring machines with client libraries is unnecessary. InterClient makes upgrading easier as well, because database upgrades on the server never make your client obsolete.

Easy to Manage and Maintain

"The database we picked had to perform searches on millions of records. It had to scale to meet our projected requirements well into the future. Since our staffing lev-els are such that no one person can act as a full-time database administrator, we also needed an application that was largely self-maintaining. Finally, as a city agency, we were under strict budget and management constraints. Taking all of this into consideration, and after looking at every major database on the market, there was only one clear choice: InterBase."

–Al Porco, Division of Disease Intervention, City of New York

Most SQL database server products require expensive MIS staffs to install, tune, and manage them. The InterBase design doesn't require hours of maintenance or a PhD in InterBase tuning. When your applications must run without constant supervision or when your desktop database runs out of steam, InterBase is the clear choice.

Installs in Minutes

InterBase is simple to install. It has a small footprint, so you do not need a lot of free disk space. InterBase is self-tuning, so you won't be required to set hundreds of incomprehensible parameters. InterBase optimizes itself for you.

"As powerful as [InterBase] is, it's not hard to learn, and it's easy to set up and install." –Peter Miller, Hospitality Data Systems

Lower Life Cycle Costs

When estimating all the costs over a product life cycle, remember the following:

• InterBase requires less memory than most database systems, so you will buy less RAM for high performance.

• Because of the InterBase tight code with its small footprint, you need less disk space.
• You will be productive quickly because InterBase adheres to industry standards.
• If you upgrade to a new operating system such as UNIX, you can redeploy without rewriting any of your database objects by backing up and restoring your database.
The life cycle of a product made with InterBase—**from conception to end-of-life**—has minimal cost, no matter how your application changes.

High Reliability for All of Your Applications

InterBase pioneered the concept of an active database, building advanced automation technology into the server's kernel. InterBase active database features include our patented event alerters, stored
procedures, triggers, User-Defined Func-tions, and Binary Large Object (Blob ™) filters. Together they move data processing steps to the server—where they are fastest and most reliable. Complementing this strong support for built-in business rules, InterBase ensures data reliability with
declarative referential integrity, including cascading operations.

Event Alerters Automate Your Applications

Event alerters notify "interested parties" when specific changes occur in the database. An application registers interest in an event, then waits without polling the database until it is notified that the event has occurred. By eliminating polling, event alerters save system resources and make applications scale better.

Real-World Example

Problem: Stock tracking database needs constant polling to make sure inventory does not drop too low before more is purchased.
Solution: With InterBase, event alerters can be set up to register specific events, and notify the parties who have expressed their interest in these events when they occur. So when stock on widgets drops below 500, an e-mail message is sent to the purchasing manager letting her know to buy more of that item.

Triggers: Reusable Business Objects

Triggers store and enforce a company's business rules on the server. The server itself guarantees that every application using corporate data adheres to these rules. InterBase triggers automated responses to events on the server, and validates input data whenever a row is inserted, updated, or deleted from a table.
"InterBase . . . has the best implementation of modular, optionally-ordered, pre- and post-operation triggers."
–DBMS, July 1996

Stored Procedures: Reusable Business Processes

Stored procedures off-load common business tasks from the client to the server, causing major performance gains. Any InterBase application can use stored procedures. They encourage modular design, and make reuse and maintenance easier.

User-Defined Functions: Reusable Custom Features

User-Defined Functions (UDFs) give developers a means of extending the analytical capabilities of InterBase. They are reusable code, accessed from the server, and ensure data reliability and integrity.
UDFs can process data themselves or call external services. InterBase provides a **default library** of User-Defined Functions that offers commonly used functions. This library includes:
• Math and trigonometry functions, including cos, sin, base, log, and more.
• String functions: difference, insert, substring, and more.

Declarative Referential Integrity Constraints

Declarative referential integrity constraints maintain relationships between records in your database efficiently and reliably.
InterBase supports four categories of constraints:
• **Unique and primary key:** No two rows in a table have the same value for the set of key columns.
• **Referential Integrity:** Parent-child relationships between tables are synchronized. The declaration can include cascading updates and deletes.
• **Check:** The associated condition will be valid for every row in the table.
• **Domain:** Create a new subtype and specify a range of acceptable values, enumerate a list of values, provide default values, and set a data type. Any column declaration may reference a defined domain as an alias for a more sophisticated data type.

Powerful Data Types Increase Flexibility

Unstructured data is an essential element for an increasing number of applications. InterBase meets this need with multi-dimensional arrays and Blobs. They make InterBase the best choice for multimedia and scientific applications.

Binary Large Objects (Blobs)

In 1986 InterBase set the industry standard by storing sound, image, graphic, and binary information directly in the database using its Blob data types. Internet and telephony applications use Blobs and Blob filters to store and manage multimedia data. Blob filters are custom routines that transform the contents of a Blob from one state to another. Filters are ideal for compression and translation and add nothing to the processing load on the client.

Multidimensional Arrays

InterBase provides direct support for the multidimensional arrays used in scientific and financial applications. A single field in the database can hold an array of sixteen dimensions. For data that is inherently organized as arrays, InterBase simplifies database design and increases performance.

Distributed Database: Application Flexibility

When you need to move your desktop database to something more sophisticated or enlarge a small workgroup application so that several departments can use it, InterBase is ideal. It was designed for
distributed database environments.

Multi-Database Access

InterBase is a truly distributed SQL database server that lets each database system query and return information to any other InterBase server.

Automatic Two-Phase Commit

Multi-database transactions require more than just the ability to connect to two data-bases. To be transactions, they must be consistent and automatic. InterBase includes a two-phase commit that ensures that distributed updates are consistent, automatically.

Distributed Two-Phase Commit Recovery

InterBase goes beyond a simple two-phase commit. It was the first database to provide distributed recovery from a failure during a two-phase commit. This ensures full recovery with no single point of failure, since the coordination of the commit is distributed among all the servers. A transaction that
cannot commit across all servers is automatically rolled back on all servers.

Database Shadows

InterBase allows you to keep an exact duplicate of the original database, called a database shadow. This copy is maintained in real time by the InterBase server, and provides an immediately available backup in case of hardware failure. The shadow runs automatically, and adds a minimal
performance penalty. Your control over the shadow includes its use of hard disk space

and distribution across available devices.

ANSI SQL-92

Training developers is expensive and time consuming. Because InterBase delivers exceptional SQL-92 compatibility, it reduces the learning curve dramatically for new programmers moving to InterBase. Because InterBase uses SQL in all its fea-tures— stored procedures, triggers, con-straints, and declarative Referential Integrity—you benefit from your develop-ers'
prior exposure to an industry-standard language. InterBase is SQL-92 entry-level compliant, with many intermediate level features and some SQL 3 features such as ROLES for group level security.

International Character Set Support: UNICODE

InterBase supports UNICODE, the universal character code, and many international character sets for data storage and manipu-lation. Columns in the same table can be created with different character sets,
allowing easy worldwide deployment. Languages supported by InterBase include Big 5 (Chinese), Korean, and all major European languages.

InterBase Can Grow With You

With InterBase, you not only have optimized performance across the most popular Windows, Linux, and UNIX platforms, but also across your company needs. InterBase crosses the spectrum from single-user databases to workgroups, to enterprise systems. As your business grows, InterBase grows with you.

2.2. ¿Qué sistema de bases de datos es el más apropiado a las necesidades del proyecto?

2.2.1. ¿Bases de datos locales o servidores SQL?

La primera gran división entre los sistemas de bases de datos existentes se produce entre los sistemas locales, o de escritorio, y las bases de datos SQL, o cliente / servidor.

A los sistemas de bases de datos locales se les llama de este modo porque comenzaron su existencia como soluciones para un solo usuario, ejecutándose en un solo ordenador. Sin embargo, su nombre no es muy apropiado, porque más adelante estos sistemas crecieron para permitir su explotación en red, incluso soportando lenguajes de consulta potentes. La esencia de las bases de datos de escritorio es el hecho de que la programación usual con las mismas se realiza en una sola capa. Todos estos sistemas utilizan como interfaz de aplicaciones un motor de datos que, en la era de la supremacía de Windows, se implementa como una DLL. En la época de MS-DOS, en cambio, eran funciones que se enlazaban estáticamente dentro del ejecutable. La aplicación y el motor de datos (BDE para Borland, DAO para Access) constituyen dos entes separados, pero un mismo ejecutable. En este tipo de bases no existe un software central que sirva de árbitro entre distintas aplicaciones para el acceso simultáneo a la base de datos física. Es como si deambularan a ciegas en un cuarto oscuro tratando de sentarse en un sitio libre. La única forma que tienen de actuar es hacerlo y confiar en que no haya nadie intentando hacer lo mismo.

Esta forma primitiva de resolver las inevitables colisiones, la implementación de la concurrencia, las transacciones y, en último término, la recuperación después de fallos, han sido tradicionalmente el punto débil de las bases de datos de escritorio.

Por esto si los datos van a ser atacados por decenas de usuarios simultáneamente será necesaria una base de datos cliente / servidor, caracterizadas por usar al menos dos capas de software. El par aplicación + motor local de datos ya no tiene acceso directo a los ficheros de la base de datos, pues hay un nuevo elemento que actúa de puente: el servidor SQL. En este nuevo diseño, el "cliente SQL" desempeña el papel del motor de datos. Esta es una denominación genérica. Para las aplicaciones desarrolladas con C++ Builder y el Motor de Datos de Borland, el cliente SQL consiste en la combinación del BDE propiamente dicho *más* alguna biblioteca dinámica o DLL suministrada por el fabricante de la base de datos. En cualquier caso, todas estas bibliotecas se funden junto a la aplicación dentro de una misma capa de software, compartiendo el mismo espacio de memoria y procesamiento.

La división entre bases de datos de escritorio y las bases de datos SQL no es una clasificación tajante, pues se basa en la combinación de una serie de características.

El hecho de que exista un árbitro en las aplicaciones cliente / servidor hace posible implementar una gran variedad de técnicas y recursos que están ausentes en la mayo-ría de los sistemas de bases de datos de escritorio. Por ejemplo, el control de concurrencia se hace más sencillo y fiable, pues el servidor puede llevar la cuenta de qué clientes están accediendo a qué registros durante todo el tiempo. También es más fácil implementar transacciones atómicas, esto es, agrupar operaciones de modificación de forma tal que, o se efectúen todas, o ninguna llegue a tener efecto.

Una de las principales características de las bases de datos cliente / servidor es la forma peculiar en que "conversan" los clientes con el servidor. Resulta que estas conversaciones tienen lugar en forma de petición de ejecución de comandos del lenguaje SQL (es por esto por lo que toman también este sobrenombre).

Un sistema SQL es inherentemente más lento que una base de datos local. Antes manipulábamos directamente un fichero. Ahora tenemos que pedírselo a alguien, con un protocolo y con reglas de cortesía. Ese alguien tiene que entender nuestra petición, es decir, compilar la instrucción. Luego debe ejecutarla y solamente entonces procederá a enviarnos el primer registro a través de la red.¿Dónde está entonces la ventaja de trabajar con sistemas cliente / servidor? Pues esta consiste en la posibilidad de meter código en el servidor, y si esto no se realizara estaríamos desaprovechando las mejores dotes de nuestra base de datos.

Hay dos formas principales de hacerlo: mediante procedimientos almacenados y mediante *triggers*. Los primeros son conjuntos de instrucciones que se almacenan dentro de la propia base de datos. Se activan mediante una petición explícita de un cliente, pero se ejecutan en el espacio de aplicación del servidor. Por descontado, estos procedimientos no deben incluir instrucciones de entrada y salida. Cualquier proceso en lote que no contenga este tipo de instrucciones es candidato a codificarse como un procedimiento almacenado. ¿La ventaja?, que evitamos que los registros procesados por el procedimiento tengan que atravesar la barrera del espacio de memoria del servidor y viajar a través de la red hasta el cliente. Los *triggers* son también secuencias de instrucciones, pero en vez de ser activados explícitamente, se ejecutan previa y posteriormente a las tres operaciones básicas de actualización de SQL: **update**, **insert** y **delete**. No importa la forma en que estas tres operaciones se ejecuten, si es a instancias de una aplicación o mediante alguna herramienta incluida

en el propio sistema; los *triggers* que se hayan programado se activarán en cualquier caso.

Tras el estudio anterior la conclusión es la de trabajar con una base de datos de arquitectura cliente / servidor.

2.2.2. Criterios para evaluar un servidor SQL

¿Cómo saber cuál es el sistema de bases de datos cliente / servidor más adecuado para satisfacer los objetivos del proyecto? En esta sección se trata de establecer criterios de comparación para estos sistemas.

- **Plataformas soportadas**

¿En qué sistema operativo debe ejecutarse el servidor? La respuesta es importante por dos razones.

La primera: nos da una medida de la flexibilidad que tenemos a la hora de seleccionar una configuración de la red. Si fuéramos a instalar la aplicación en una empresa que tiene toda su red basada en determinado servidor con determinado protocolo, no sería recomendable abandonar todo lo que tienen hecho para que el recién llegado pueda ejecutarse.

En segundo lugar, no todos los sistemas operativos se comportan igual de eficientemente actuando como servidores de bases de datos. Esto tiene que ver sobre todo con la implementación que realiza el SO de la concurrencia. El mejor en este sentido es UNIX: un InterBase, un Oracle, un Informix, ejecutándose sobre cualquier distribución de UNIX (HP-UX, AIX, Solaris).

Si el mantenimiento de la red no está en manos de un profesional dedicado a esta área, como es el caso del Laboratorio de Potencia, hay que reconocer que es mucho más sencillo administrar un Windows NT y más aún mantener una red punto a punto con el Windows 9x en cada puesto, opción elegida en nuestro caso.

- **Soporte de tipos de datos y restricciones**

En realidad, casi todos los sistemas SQL actuales tienen tipos de datos que incluyen a los especificados en el estándar de SQL, excepto determinados casos. De lo que se trata sobre todo es la implementación de los mismos. Por ejemplo, no todos los formatos de bases de datos implementan con la misma eficiencia el tipo *VARCHAR*, que almacena cadenas de longitud variable. La longitud máxima es uno de los parámetros que varían, y el formato de representación: si siempre se asigna un tamaño fijo (el máximo) para estos campos, o si la 0longitud total del registro puede variar.

Más importante es el soporte para validaciones declarativas, esto es, verificaciones para las cuales no hay que desarrollar código especial. Dentro de este apartado entran las claves primarias, claves alternativas, relaciones de integridad referencial, dominios, chequeos a nivel de fila, etc. Un elemento a tener en cuenta, por ejemplo, es si se permite o no la propagación en cascada de modificaciones en la tabla maestra de una relación de integridad referencial. En caso positivo, esto puede

ahorrarnos bastante código en *triggers* y permitirá mejores modelos de bases de datos.

- **Lenguaje de triggers y procedimientos almacenados**

Este es uno de los criterios a los que se debe conceder mayor importancia. El éxito de una aplicación, o un conjunto de aplicaciones, en un entorno C/S depende en gran medida de la forma en que dividamos la carga de la aplicación entre el servidor de datos y los clientes. En este punto es donde podemos encontrar mayores diferencias entre los sistemas SQL, pues no hay dos dialectos de este lenguaje que sean exactamente iguales.

Debemos fijarnos en si el lenguaje permite *triggers* a nivel de fila o de operación, o de ambos niveles. Un *trigger* a nivel de fila se dispara antes o después de modificar una fila individual. Por el contrario, un *trigger* a nivel de operación se dispara después de que se ejecute una operación completa que puede afectar a varias filas.

Si la base de datos en cuestión sólo ejecuta sus *triggers* al terminar las operaciones, será mucho más complicada la programación de los mismos, pues más trabajo costará restablecer los valores anteriores y posteriores de cada fila particular. También debe tenerse en cuenta las posibilidades de extensión del lenguaje, por ejemplo, incorporando funciones definidas en otros lenguajes, o el poder utilizar servidores de automatización COM o CORBA, si se permiten o no llamadas recursivas, etc.

- **Implementación de transacciones: recuperación y aislamiento**

Este es otro de los criterios de análisis fundamentales. Las transacciones ofrecen a las bases de datos la consistencia necesaria para que operaciones parciales no priven de sentido semántico a los datos almacenados. Al constituir las unidades básicas de procesamiento del servidor, el mecanismo que las soporta debe encargarse también de aislar a los usuarios entre sí, de modo que la secuencia exacta de pasos concurrentes que realicen no influya en el resultado final de sus acciones. Por lo tanto, hay dos puntos en los que centrar la atención: cómo se implementa la atomicidad (la forma en que el sistema deshace operaciones inconclusas), y el método empleado para aislar entre sí las transacciones concurrentes.

- **Segmentación**

Es conveniente poder distribuir los datos de un servidor en distintos dispositivos físicos. Al situar tablas en distintos discos, o segmentos de las propias tablas, podemos aprovechar la concurrencia inherente a la existencia de varios controladores físicos de estos medios de almacenamiento. Esta opción permite, además, superar las restricciones impuestas por el tamaño de los discos en el tamaño de la base de datos. En nuestro caso el tamaño de la base de datos es minúsculo comparado con el disponible en el disco duro de cualquier PC medianamente moderno, por lo que este criterio permanecerá en segundo plano.

- **Replicación**

Uno de los usos principales de la replicación consiste en aislar las aplicaciones que realizan mantenimientos (transacciones OLTP) de las aplicaciones para toma de decisiones (transacciones DSS). Las aplicaciones con fuerte base OLTP tienden a modificar en cada transacción un pequeño número de registros. Las aplicaciones DSS se caracterizan por leer grandes cantidades de información, siendo poco probable que escriban en la base de datos. En sistemas de bases de datos que implementan el aislamiento de transacciones mediante bloqueos, ambos tipos de transacciones tienen una coexistencia difícil. Por lo tanto, es conveniente disponer de réplicas de la base de datos sólo para lectura, y que sea a estos clones a quienes se refieran las aplicaciones DSS.

El último factor de la lista anterior es el precio. Todos queremos lo mejor, pero no siempre estamos dispuestos a pagar por eso. Así que muchas veces una decisión de compra representa un balance entre la calidad y el precio.

2.2.3. Bases de datos C/S disponibles

- **InterBase**

Quizá debido a algún fallo de Marketing no tiene la popularidad que merecería por sus muchas virtudes. Una de estas virtudes es que se puede instalar el servidor en muchos sistemas operativos: Windows NT y 9x, NetWare, Solaris, HP-UX, SCO, Linux... Otra es que en cualquiera de estos sistemas ocupa muy poco espacio, del orden de los 10 ó 20 MB. El mantenimiento de un servidor, una vez instalado, es también mínimo.

Cada base de datos se sitúa en uno o más ficheros, casi siempre de extensión *gdb*. Estos ficheros crecen automáticamente cuando hace falta más espacio, y no hay que preocuparse por reservar espacio adicional para registrar los cambios efectuados por las transacciones, como en otros sistemas. Se pueden realizar copias de seguridad de una base de datos “en caliente”, mientras que otros sistemas requieren que no existan usuarios activos para efectuar esta operación. Y la recuperación después de un fallo de hardware es sencilla e inmediata: vuelva a encender el servidor.

Tal sencillez tiene un precio, y es que actualmente InterBase no implementa directamente ciertas opciones avanzadas de administración, como la segmentación y la replicación. Esta última debe ser implementada manualmente, por medio de *triggers* definidos por el diseñador de la base de datos, y con la ayuda de un proceso en segundo plano que vaya grabando los datos del original a la réplica. No obstante, InterBase permite definir directamente copias en espejo (*mirrors*) de una base de datos, de forma tal que existan dos copias sincronizadas de una misma base de datos en discos duros diferentes del mismo servidor. De este modo, si se produce un fallo de hardware en uno de los discos o controladores, la explotación de la base de datos puede continuar con la segunda copia.

En lo que atañe a los tipos de datos, InterBase implementa todos los tipos del estándar SQL con una excepción. Este lenguaje define tres tipos de campos para la fecha y la hora: *DATE*, para fechas, *TIME*, para horas, y *TIMESTAMP* para

almacenar conjuntamente fechas y horas. InterBase solamente tiene *DATE*, pero como equivalente al *TIMESTAMP* del estándar. Esto en sí no conlleva problemas, a no ser que haya que escribir una aplicación que acceda indistintamente a bases de datos de InterBase y de cualquier otro sistema SQL. El uso del Diccionario de Datos de C++ Builder, permite resolver esta nimiedad.

InterBase tiene, hoy por hoy, una de las implementaciones más completas de las restricciones de integridad referencial, pues permite especificar declarativamente la propagación en cascada de borrados y modificaciones:

Por supuesto, tenemos todas las restricciones de unicidad, claves primarias, validaciones a nivel de registro (cláusulas **check**). Estas últimas son más potentes que en el resto de los sistemas, pues permiten realizar comprobaciones que involucren a registros de otras tablas.

Los *triggers* de InterBase se ejecutan antes o después de cada operación de actualización, fila por fila, que es lo que interesa. Por otra parte, tiene un lenguaje de procedimientos almacenados muy completo, que permite llamadas recursivas, y la definición de procedimientos de selección, que devuelven más de un registro de datos por demanda. Todo ello se integra con un mecanismo de excepciones muy elegante, que compagina muy bien con las técnicas transaccionales. Es posible, además, extender el conjunto de funciones del lenguaje mediante módulos dinámicos, DLLs en el caso de las versiones para Windows y Windows NT.

Pero la característica más destacada de InterBase es la forma en la que logra implementar el acceso concurrente a sus bases de datos garantizando que, en lo posible, cada usuario no se vea afectado por las acciones del resto. Casi todos los sistemas existentes utilizan, de una forma u otra, bloqueos para este fin. InterBase utiliza la denominada *arquitectura multigeneracional*, en la cual cada vez que una transacción modifica un registro se genera una nueva versión del mismo. Teóricamente, esta técnica permite la mayor cantidad de acciones concurrentes sobre las bases de datos. En la práctica, la implementación de InterBase es muy eficiente y confiable.

- **Microsoft SQL Server**

MS SQL Server comenzó como un derivado del servidor de Sybase, por lo que la arquitectura de ambos es muy parecida. De modo que gran parte de lo que se diga en esta sección sobre un sistema, vale para el otro.

Es muy fácil tropezarse por ahí con MS SQL Server. Hay incluso quienes lo tienen instalado y aún no se han dado cuenta. Microsoft tiene una política de distribución bastante agresiva para este producto, pues lo incluye en el paquete BackOffice, junto a su sistema operativo Windows NT y unos cuantos programas más. Como es de imaginar, SQL Server 6.5 sólo puede ejecutarse en Windows NT . Por fortuna, la versión 7.0 ofrece un servidor para Windows 9x ... aunque obliga al usuario a instalar primeramente Internet Explorer 4.

Lo primero que llama la atención es la cantidad de recursos del sistema que consume una instalación de SQL Server. La versión 6.5 ocupa de 70 a 90MB,

mientras que la versión 7 llega a los 180MB de espacio en disco. ¿La explicación? Asistentes, y más asistentes: en esto contrasta con la austeridad espartana de InterBase.

A pesar de que la instalación del servidor es relativamente sencilla, su mantenimiento es bastante complicado, sobre todo en la versión 6.5. Cada base de datos reside en uno o más *dispositivos físicos*, que en el fondo no son más que vulgares ficheros. Estos dispositivos ayudan a implementar la segmentación, pero no crecen automáticamente, por lo que el administrador del sistema deberá estar pendiente del momento en que los datos están a punto de producir un desbordamiento. A diferencia de InterBase, para cada base de datos hay que definir explícitamente un *log*, o registro de transacciones, que compite en espacio con los datos verdaderos, aunque este registro puede residir en otro dispositivo (lo cual se recomienda).

Aunque los mecanismos de *logging*, en combinación con los bloqueos, son los más frecuentes en las bases de datos relacionales como forma de implementar transacciones atómicas, presentan claras desventajas en comparación con la arquitectura multigeneracional.

En primer lugar, no se pueden realizar copias de seguridad con usuarios conectados a una base de datos. Los procesos que escriben bloquean a los procesos que se limitan a leer información, y viceversa. Si se desconecta físicamente el servidor, es muy probable que haya que examinar el registro de transacciones antes de volver a echar a andar las bases de datos. Por último, hay que estar pendientes del crecimiento de estos ficheros.

Esta situación mejora un poco con la versión 7, pues desaparece el concepto de dispositivo, siendo sustituido por el de fichero del propio sistema operativo: las bases de datos se sitúan en ficheros de extensión *mdf* y *ndf*, los registros de transacciones, en ficheros *ldf*. Este nuevo formato permite que los ficheros de datos y de transacciones crezcan dinámicamente, por demanda.

Otro grave problema de versiones anteriores que soluciona la nueva versión es la granularidad de los bloqueos. Antes, cada modificación de un registro imponía un bloqueo a toda la página en que éste se encontraba. Además, las páginas tenían un tamaño fijo de 2048 bytes, lo que limitaba a su vez el tamaño máximo de un registro.

En la versión 6.5 se introdujo el bloqueo a nivel de registro ... pero únicamente para las inserciones, que es cuando menos hacen falta. Finalmente, la versión 7 permite siempre bloqueos de registros, que pueden escalarse por demanda a bloqueos de página o a nivel de tablas, y aumenta el tamaño de página a 8192 bytes. No obstante, este tamaño sigue sin poder ajustarse.

Microsoft SQL Server ofrece extrañas extensiones a SQL que solamente sirven para complicar el lenguaje. Por ejemplo, aunque el SQL estándar dice que por omisión una columna admite valores nulos, esto depende en SQL Server del estado de un parámetro, ¡que por omisión produce el efecto contrario!

La implementación de la integridad referencial en este sistema es bastante pobre, pues solamente permite restringir las actualizaciones y borrados en la tabla maestra;

nada de propagación en cascada y otras alegrías. También es curioso que SQL Server no crea automáticamente índices secundarios sobre las tablas que contienen claves externas.

Otro de los aspectos negativos de SQL Server es su lenguaje de *triggers* y procedimientos almacenados, llamado TransactSQL, que es bastante excéntrico respecto al resto de los lenguajes existentes y a la propuesta de estándar. Uno puede acostumbrarse a soberanas tonterías tales como obligar a que todas las variables locales y parámetros comiencen con el carácter @. Pero es bastante difícil programar determinadas reglas de empresa cuando los *triggers* se disparan solamente después de instrucciones completas.

- **Oracle**

Oracle es uno de los abuelos en el negocio de las bases de datos relacionales; el otro es DB2, de IBM. Este sistema es otra de las apuestas seguras en el caso de tener que elegir un servidor de bases de datos. Su principal desventaja no es de carácter técnico, sino que tiene que ver con una política de precios altos, alto coste de la formación y del mantenimiento posterior del sistema... lo que restringe su uso a compañías con un elevado presupuesto destinado a la gestión de bases de datos.

Oracle funciona en cualquier plataforma hardware. El servidor *Personal Oracle* para Windows 95 es muy estable, aunque a veces es algo lento para establecer la conexión a la base de datos y cuesta un poco instalarlo adecuadamente (sobre todo por las complicaciones típicas de TCP/IP). pero una vez en funcionamiento va de maravillas.

Oracle permite todas las funciones avanzadas de un servidor SQL: segmentación, replicación, etc. Incluso puede pensarse que tiene demasiados parámetros de configuración. La parte principal del control de transacciones se implementa mediante bloqueos y registros de transacciones, aunque el nivel de aislamiento superior se logra mediante copias sólo lectura de los datos. Por supuesto, el nivel mínimo de granularidad de estos bloqueos es a nivel de registro. ¿Tipos de datos? Todos los que usted desee. ¿Restricciones *check*? No tan generales como las de InterBase, pero quedan compensadas por la mayor abundancia de funciones predefinidas. Hasta la versión 7.3, Oracle implementaba solamente la propagación en cascada de borrados para la integridad referencial. Las extensiones procedimentales a SQL, denominadas PL/SQL, conforman un lenguaje potente, que permite programar *paquetes* (*packages*) para la implementación de tipos de datos abstractos. Con la versión 8, incluso, se pueden definir tipos de clases, u *objetos*. Esta última extensión no es, sin embargo, lo suficientemente general como para clasificar a este sistema como una base de datos orientada a objetos, en el sentido moderno de esta denominación. Uno de los puntos fuertes de la versión 4.0 de C++ Builder es la posibilidad de trabajar con las extensiones de objetos de Oracle 8.

Como pudiera esperarse, el lenguaje de *triggers* es muy completo, y permite especificarlos tanto a nivel de fila como de operación. Hay muchas funciones utilizables desde SQL, y curiosas extensiones al lenguaje consulta, como la posibilidad de realizar determinados tipos de clausuras transitivas.

- **Otros sistemas de uso frecuente**

Evidentemente, es imposible hablar acerca de todos los formatos de bases de datos existentes en el mercado, y las secciones anteriores se han limitado a presentar aquellos sistemas con los se trabajan con mayor frecuencia, pero pueden comentarse de pasada algunos. Por ejemplo, DB2, de IBM. Antes se mencionó que este sistema y Oracle eran los dos sistemas que más tiempo llevaban en este negocio, y los frutos de esta experiencia se dejan notar también en DB2. Existen actualmente versiones de DB2 para una amplia gama de sistemas operativos.

La arquitectura de DB2 es similar a la de Oracle, a la que se parece la de MS SQL Server, que es similar a la de Sybase SQL Server... En realidad, la concepción de estos sistemas está basada en un proyecto experimental de IBM, denominado System-R, que fue la primera implementación de un sistema relacional. En este proyecto se desarrollaron o perfeccionaron técnicas como los identificadores de registros, los mecanismos de bloqueos actuales, registros de transacciones, índices basados en árboles balanceados, los algoritmos de optimización de consultas, etc. Así que también podrá usted esperar de DB2 la posibilidad de dividir en segmentos sus bases de datos, de poder realizar réplicas y de disponer de transacciones atómicas y coherentes.

El mantenimiento de las bases de datos de DB2 puede ser todo lo simple que (sacrificando algo el rendimiento), o todo lo complicado que se desee (a costa de un buen esfuerzo). El lenguaje de *triggers* y procedimientos almacenados es muy completo, y similar al de Oracle e InterBase, como era de esperar. El único defecto de DB2 es que la instalación de clientes es bastante pesada, y para poder conectar una estación de trabajo hay que realizar manualmente un proceso conocido como *catalogación*. Pero esto mismo le sucede a Oracle con su SQL Net.

Otro sistema importante es Informix, que está bastante ligado al mundo de UNIX, aunque en estos momentos existen versiones del servidor para Windows NT. Su arquitectura es similar a la de los sistemas antes mencionados.

Finalmente, se tratarán ligeramente otras bases de datos “no BDE”.

Tenemos, por ejemplo, la posibilidad de trabajar con bases de datos de AS/400.

Aunque el motor de datos que viene con C++ Builder no permite el acceso directo a las mismas, podemos programar para estas bases de datos colocando una pasarela DB2 como interfaz. No obstante, el producto C++ Builder/400 sí que nos deja saltarnos las capas intermedias, logrando mayor eficiencia a expensas de la pérdida de portabilidad. También está muy difundido Btrieve, una base de datos que inicio su vida como un sistema navegacional, pero que en sus últimas versiones ha desarrollado el producto Pervasive SQL, que es un motor de datos cliente / servidor relacional.

Lamentablemente, tampoco está soportado directamente por el motor de datos de C++ Builder, por el momento.

2.3. Conclusión

Tras el estudio precedente y vistos los recursos del Departamento en que este trabajo se ve inmerso y las necesidades y objetivos del proyecto, se tomó la resolución de trabajar con el entorno de desarrollo C++BUILDER de Borland en la versión 5.0 sobre el sistema de bases de datos InterBase, también de Borland, esta vez en su versión 5.5.

A continuación mostramos las especificaciones técnicas de este último producto, de sobra aptas para nuestros propósitos:

Database Statistics (Upper Limits)

Maximum size of database: 32TB using multiple files; largest recorded InterBase database in production is over 200GB

Maximum size of one file: 4GB on most platforms; 2GB on some platforms

Maximum number of tables: 64K Tables

Maximum size of one table: 32TB

Maximum number of rows per table: 4G Rows

Maximum row size: 64KB

Maximum number of columns per table: Depends on the datatypes you use. (Example: 16,384 INTEGER (4 byte) values per row.)

Maximum number of indexes per table: 64K indexes

Maximum number of indexes per database: 4G indexes

InterBase Datatype Specifics

Name	Size	Range/Precision	Description
Varchar(n)	n chars	1 to 32767 bytes	Variable length char or text string
Smallint	16 bits	-2^15 to 2^15-1	Signed short (word)
Integer	32 bits	-2^31 to 2^31-1	Signed long (longword)
Float	32 bits	3.4 x 10^-38 to 3.4 x 10^38	7 digit precision
Double Precision	64 bits	1.7 x 10^-308 to 1.7 x 10^308	15 digit precision
*Timestamp	64 bits	1 Jan 100 a.d. to 28 Feb 32768 a.d.	Includes time and date
**Date	32 bits	1 Jan 100 a.d. to 29 Feb 32768 a.d.	
*Time	32 bits	0:00 AM to 23:59.9999 PM	
Blob	<32GB		Stores data of variable indeterminate size
***Numeric (*precision, scale*)	Variable (16, 32, or 64)	specifies exactly *precision* digits of precision	Example: **Numeric(10,3)** holds numbers accurately in the following format: **ppppppp.sss**
***Decimal (*precision, scale*)	Variable (16, 32, or 64)	specifies at least *precision* digits of precision	Example: **Decimal(10,3)** holds numbers accurately in the following format: **ppppppp.sss**

2. Manual del Programador

1. LA BASE DE DATOS

1.1. La estructura de SQL

Las instrucciones de SQL se pueden agrupar en dos grandes categorías: las instrucciones que trabajan con la estructura de los datos y las instrucciones que trabajan con los datos en sí. Al primer grupo de instrucciones se le denomina también el *Lenguaje de Definición de Dato*s, en inglés *Data Definition Languag*e, con las siglas DDL. Al segundo grupo se le denomina el *Lenguaje de Manipulación de Dato*s, en inglés *Data Manipulation Languag*e, o DML. A veces las instrucciones que modifican el acceso de los usuarios a los objetos de la base de datos, y que aquí se incluyen en el DDL, se consideran pertenecientes a un tercer conjunto: el *Lenguaje de Control de Dato*s, *Data Control Languag*e, ó DCL.

En estos momentos existen estándares aceptables para estos tres componentes del lenguaje. El primer estándar, realizado por la institución norteamericana ANSI y luego adoptada por la internacional ISO, se terminó de elaborar en 1987. El segundo, que es el que está actualmente en vigor, es del año 1992. En estos momentos está a punto de ser aprobado un tercer estándar, que ya es conocido como SQL-3.

Las condiciones en que se elaboró el estándar de 1987 fueron especiales, pues el mercado de las bases de datos relacionales estaba dominado por unas cuantas compañías, que trataban de imponer sus respectivos dialectos del lenguaje. El acuerdo final dejó solamente las construcciones que eran comunes a todas las implementaciones; de este modo, nadie estaba obligado a rescribir su sistema para no quedarse sin la certificación. También se definieron diferentes niveles de conformidad para facilitarles las cosas a los fabricantes. Este estándar dejó fuera cosas tan necesarias como las definiciones de integridad referencial. Sin embargo, introdujo el denominado *lenguaje de módulo*s, una especie de interfaz para desarrollar funciones en SQL que pudieran utilizarse en programas escritos en otros lenguajes.

El estándar del 92 se ocupó de la mayoría de las áreas que quedaron por cubrir en el 87. Sin embargo, no se hizo nada respecto a recursos tales como los procedimientos almacenados, los *triggers* o disparadores, y las excepciones, que permiten la especificación de reglas para mantener la integridad y consistencia desde un enfoque *imperativ*o, en contraste con el enfoque *declarativo* utilizado por el DDL, el DCL y el DML. Dedicaremos un capítulo al estudio de estas construcciones del lenguaje. En este preciso momento, cada fabricante tiene su propio dialecto para definir procedimientos almacenados y disparadores. El estándar conocido como SQL-3 se encarga precisamente de unificar el uso de estas construcciones del lenguaje.

InterBase viene acompañado por la aplicación Windows ISQL. Con esta utilidad podemos crear y borrar bases de datos de InterBase, conectarnos a bases de datos

existentes y ejecutar todo tipo de instrucciones SQL sobre ellas. Las instrucciones del lenguaje de manipulación de datos, y algunas del lenguaje de definición, pueden ejecutarse directamente tecleando en un cuadro de edición multilíneas y pulsando un botón. El resultado de la ejecución aparece en un control de visualización situado en la parte inferior de la ventana. Para las instrucciones DDL más complejas y la gestión de procedimientos, disparadores, dominios y excepciones es preferible utilizar *scripts* SQL. Un *script* es un fichero de texto, por lo general de extensión *sql*, que contiene una lista de instrucciones arbitrarias de este lenguaje separadas entre sí por puntos y comas. Este fichero debe ejecutarse mediante el comando de menú *File|Run an ISQL Script*. Las instrucciones SQL se van ejecutando secuencialmente, según el orden en que se han escrito. Por omisión, los resultados de la ejecución del *script* también aparecen en el control de visualización de la ventana de Windows ISQL.

También puede utilizarse la utilidad SQL Explorer del propio C++ Builder para ejecutar instrucciones individuales sobre cualquier base de datos a la que deseemos conectarnos.

1.2. La creación y conexión a la base de datos

Estamos en InterBase, utilizando Windows ISQL. ¿Cómo nos conectamos a una base de datos de InterBase para comenzar a trabajar? Basta con activar el comando de menú *File|Connect to databas*e. Si se ha instalado InterBase en un servidor remoto y tiene los SQL Links que vienen con C++ Builder cliente / servidor, se puede elegir la posibilidad de conectarse a ese servidor remoto. En cualquier caso, se puede elegir el servidor local. Cuando especificamos un servidor remoto, tenemos que teclear el nombre completo del fichero de base de datos en el servidor; no necesitamos tener acceso al fichero desde el sistema operativo, pues es el servidor de InterBase el que nos garantiza el acceso al mismo. Si estamos utilizando el servidor local, las cosas son más fáciles, pues contamos con un botón *Browse* para explorar el disco. Las bases de datos de InterBase se sitúan por lo general en un único fichero de extensión *gd*b; existe, no obstante, la posibilidad de distribuir información en ficheros secundarios, lo cual puede ser útil en servidores con varios discos duros. Los otros datos que tenemos que suministrar a la conexión son el nombre de usuario y la contraseña. El nombre de usuario inicial en InterBase es, por omisión, *SYSDB*A, y su contraseña es *masterke*y. Las mayúsculas y minúsculas deben respetarse. Una vez que acepte el cuadro de diálogo, se intentará la conexión. En cualquier momento de la sesión podemos saber a qué base de datos estamos conectados mirando la barra de estado de la ventana.

El mismo mecanismo puede utilizarse para crear una base de datos interactivamente. La diferencia está en que debemos utilizar el comando *File|Create databas*e. Sin embargo, necesitamos saber también cómo podemos crear una base de datos y establecer una conexión utilizando instrucciones SQL. La razón es que todo *script* SQL debe comenzar con una instrucción de creación de bases de datos o de conexión.

La más sencilla de estas instrucciones es la de conexión:

connect "C:\ InterBase\Examples\Prueba.GDB"

user "SYSDBA" **password** "masterkey";

Por supuesto, el fichero mencionado en la instrucción debe existir, y el nombre de usuario y contraseña deben ser válidos. Observe el punto y coma al final, para separar esta instrucción de la próxima en el *scrip*t.

La instrucción necesaria para crear una base de datos tiene una sintaxis similar. El siguiente ejemplo muestra el ejemplo más común de creación:

create database "C:\InterBase \Examples\Prueba.GDB"

user "SYSDBA" **password** "masterkey"

page_size 2048;

InterBase, y casi todos los sistemas SQL, almacenan los registros de las tablas en bloques de longitud fija, conocidos como *página*s. En la instrucción anterior estamos redefiniendo el tamaño de las páginas de la base de datos. Por omisión, InterBase utiliza páginas de 1024 bytes. En la mayoría de los casos, es conveniente utilizar un tamaño mayor de página; de este modo, el acceso a disco es más eficiente, entran más claves en las páginas de un índice con lo cual disminuye la profundidad de estos, y mejora también el almacenamiento de campos de longitud variable. Sin embargo, como nuestra aplicación trabaja con pocas filas de la base de datos, es más conveniente mantener un tamaño pequeño de página, pues la lectura de éstas tarda entonces menos tiempo, y el *buffer* puede realizar las operaciones de reemplazo de páginas en memoria más eficientemente. También podemos indicar el tamaño inicial de la base de datos en páginas. Normalmente esto no hace falta, pues InterBase hace crecer automáticamente el fichero *gdb* cuando es necesario. Pero si se tuviera en mente insertar grandes cantidades de datos sobre la base recién creada, puede ahorrar el tiempo de crecimiento utilizando la opción **length**. La siguiente instrucción crea una base de datos reservando un tamaño inicial de 1 MB:

create database "C:\ InterBase\Examples\Prueba.GDB"

user "SYSDBA" **password** "masterkey"

page_size 2048 **length** 512

default character set "ISO8859_1";

Esta instrucción muestra también cómo especificar el *conjunto de caracteres* utilizado por omisión en la base de datos. Más adelante, se pueden definir conjuntos especiales para cada tabla, de ser necesario. El conjunto de caracteres determina, fundamentalmente, de qué forma se ordenan alfabéticamente los valores alfanuméricos. Los primeros 127 caracteres de todos los conjuntos de datos coinciden; es en los restantes valores donde puede haber diferencias.

En nuestra pequeña aplicación para la inversión de corriente esta sección se reduce a la siguiente sentencia

```
CREATE DATABASE "c:\inversor\inversor.db" PAGE_SIZE 1024;
```

1.3. Tipos de datos en SQL

Antes de poder crear las tablas, tenemos que saber qué tipos de datos podemos emplear. SQL estándar define un conjunto de tipos de datos que todo sistema debe implementar. Ahora bien, la interpretación exacta de estos tipos no está completamente especificada. Cada sistema de bases de datos ofrece, además, sus propios tipos nativos.

Estos son los tipos de datos aceptados por InterBase:

Tipo de dato	**Tamaño**	**Observaciones**
char(n), varchar(n)	*n* bytes	Longitud fija; longitud variable
integer, int	32 bits	
smallint	16 bits	
float	4 bytes	Equivale al tipo homónimo de C
double precision	8 bytes	Equivale al *double* de C
numeric(*prec*, *esc*)	*prec*=1-15, *esc* <= *prec*	Variante “exacta” de *decimal*
decimal(*prec*, *esc*)	*prec*=1-15, *esc* <= *prec*	
date	64 bits	Almacena la fecha y la hora
blob	No hay límite	

La diferencia fundamental respecto al estándar SQL tiene que ver con el tipo **date**. SQL estándar ofrece los tipos **date**, **time** y **timestamp**, para representar fecha, hora y la combinación de fecha y hora. El tipo **date** de InterBase corresponde al tipo **timestamp** del SQL estándar. El tipo de dato **blob** *(Binary Large OBject* = Objeto Binario Grande) se utiliza para almacenar información de longitud variable, generalmente de gran tamaño. En principio, a InterBase no le preocupa qué formato tiene la información almacenada. Pero para cada tipo **blob** se define un *subtipo*, un valor entero que ofrece una pista acerca del formato del campo. InterBase interpreta el subtipo 0 como el formato por omisión: ningún formato. El subtipo 1 representa texto, como el tipo memo de otros sistemas. Se pueden especificar subtipos definidos por el programador; en este caso, los valores empleados deben ser negativos. La especificación de subtipos se realiza mediante la cláusula **sub_type**:

```
Comentarios blob sub_type 1
```

Una de las peculiaridades de InterBase como gestor de bases de datos es el soporte de *matrices multidimensionales*. Se pueden crear columnas que contengan matrices de tipos simples, con excepción del tipo **blob**, de hasta 16 dimensiones. Sin

embargo, C++ Builder no reconoce directamente este tipo de campos, y debemos trabajar con ellos como si fueran campos BLOB.

- **Representación de datos en InterBase**

¿Qué diferencias hay entre los tipos **char** y **varcha**r? Cuando una aplicación graba una cadena en una columna de tipo **varcha**r, InterBase almacena exactamente la misma cantidad de caracteres que ha especificado la aplicación, independientemente del ancho máximo de la columna. Cuando se recupera el valor más adelante, la cadena obtenida tiene la misma longitud que la original. Ahora bien, si la columna de que hablamos ha sido definida como **cha**r, en el proceso de grabación se le añaden automáticamente espacios en blanco al final del valor para completar la longitud de la cadena. Cuando se vuelve a leer la columna, la aplicación recibe estos espacios adicionales. ¿Quiere esto decir que ahorraremos espacio en la base de datos utilizando siempre el tipo **varcha**r? ¡Conclusión prematura! InterBase utiliza registros de longitud variable para representar las filas de una tabla, con el fin de empaquetar la mayor cantidad posible de registros en cada página de la base de datos. Como parte de la estrategia de disminución del tamaño, cuando se almacena una columna de tipo **char** se eliminan automáticamente los espacios en blanco que puede contener al final, y estos espacios se restauran cuando alguien recupera el valor de dicha columna. Más aún: para almacenar un **varchar** es necesario añadir a la propia representación de la cadena un valor entero con la longitud de la misma. Como resultado final, ¡una columna de tipo **varchar** consume más espacio que una de tipo **cha**r! ¿Para qué, entonces, utilizar el tipo **varcha**r?. Debe recuerdarse que si se utiliza el tipo **char** se recibirán valores con espacios en blanco adicionales al final de los mismos, y se tendrá que utilizar frecuentemente la función *TrimRight* para eliminarlos. El tipo **varchar** le ahorra este incordio.

También es útil conocer cómo InterBase representa los tipos **numeric** y **decima**l. El factor decisivo de la representación es el número de dígitos de la precisión. Si es menor que 5, **numeric** y **decimal** pueden almacenarse dentro de un tipo entero de 16 bits, o **smallin**t; si es menor que 10, en un **integer** de 32 bits; en caso contrario, se almacenan en columnas de tipo **double precisio**n.

1.4. Creación de tablas

La instrucción de creación de tablas tiene la siguiente sintaxis en InterBase:

```
create table NombreDeTabla [external file NombreFichero] (DefColumna [, DefColumna |
Restriccion … ] );
```

La opción **external file** es propia de InterBase e indica que los datos de la tabla deben residir en un fichero externo al principal de la base de datos. Aunque el formato de este fichero no es ASCII, es relativamente sencillo de comprender y puede utilizarse para importar y exportar datos de un modo fácil entre InterBase y otras aplicaciones. No haremos uso de esta cláusula.

Para crear una tabla tenemos que definir columnas y restricciones sobre los valores que pueden tomar estas columnas. La forma más sencilla de definición de columna es la que sigue: NombreColumna TipoDeDato

Por ejemplo:

```
create table Empleados (Codigo integer, Nombre varchar(30), Contrato date, Salario integer);
```

- **Campos Calculados**

Con InterBase tenemos la posibilidad de crear *columnas calculada*s, cuyos valores se derivan a partir de columnas existentes, sin necesidad de ser almacenados físicamente, para lo cual se utiliza la cláusula **computed by**. Aunque para este tipo de columnas podemos especificar explícitamente un tipo de datos, es innecesario, porque se puede deducir de la expresión que define la columna:

```
create table Empleados(Codigo integer, Nombre varchar, Apellidos varchar, Salario integer, NombreCompleto computed by (Nombre || " " || Apellidos),/* … */);
```

El operador || sirve para concatenar cadenas de caracteres en InterBase. En general, no es buena idea definir columnas calculadas en el servidor, sino que es preferible el uso de campos calculados en el cliente. Si utilizamos **computed by** hacemos que los valores de estas columnas viajen por la red con cada registro, aumentando el tráfico en la misma.

- **Valores por omisión**

Otra posibilidad es la de definir valores por omisión para las columnas. Durante la inserción de filas, es posible no mencionar una determinada columna, en cuyo caso se le asigna a esta columna el valor por omisión. Si no se especifica algo diferente, el valor por omisión de SQL es **nul**l, el valor desconocido. Con la cláusula **default** cambiamos este valor:

Salario integer **default** 0, FechaContrato date **default** "Now",

Observe en el ejemplo anterior el uso del literal "*Now*", para inicializar la columna con la fecha y hora actual en InterBase.

Si se mezclan las cláusulas **default** y **not null** la primera debe ir antes de la segunda.

- **Restricciones de integridad**

Durante la definición de una tabla podemos especificar condiciones que deben cumplirse para los valores almacenados en la misma. Por ejemplo, no nos basta saber que el salario de un empleado es un entero; hay que aclarar también que en circunstancias normales es también un entero positivo, y que no podemos dejar de especificar un salario a un trabajador. También nos puede interesar imponer condiciones más complejas, como que el salario de un empleado que lleva menos de un año con nosotros no puede sobrepasar cierta cantidad fija. Aquí veremos cómo expresar estas *restricciones de integrida*d.

La restricción más frecuente es pedir que el valor almacenado en una columna no pueda ser nulo.Esto quiere decir que hay que suministrar un valor para esta columna durante la inserción de un nuevo registro, pero también que no se puede modificar posteriormente esta columna de modo que tenga un valor nulo. Esta restricción es indispensable para poder declarar claves primarias y claves alternativas. Por ejemplo:

```
create table Empleados(
Codigo integer not null,
Nombre varchar(30) not null,
/* … */
);
```

Cuando la condición que se quiere verificar es más compleja, se puede utilizar la cláusula **check**. Por ejemplo, la siguiente restricción verifica que los códigos de provincias se escriban en mayúsculas:

Provincia varchar(2) **check** (Provincia = upper(Provincia))

Existen dos posibilidades con respecto a la ubicación de la mayoría de las restricciones: colocar la restricción a nivel de columna o a nivel de tabla. A nivel del columna, si la restricción afecta solamente a la columna en cuestión; a nivel de tabla si hay varias columnas involucradas. La cláusula **check** de InterBase permite incluso expresiones que involucran a otras tablas. Analicemos la siguiente restricción, expresada a nivel de tabla:

```
create table Detalles ( RefPedido int not null, NumLinea int not null, RefArticulo
int not null, Cantidad int default 1 not null, Descuento int default 0  not null,
check (Descuento between 0 and 50 or "Marteens Corporation"= (select Nombre from
Clientes where Codigo = (select RefCliente from Pedidos where Numero =
Detalles.RefPedido))), /* … */ );
```

Esta cláusula dice, en pocas palabras, que solamente el autor del libro que sirve de fuente para esta memoria y para gran parte del proyecto en si puede beneficiarse de descuentos superiores al 50%. ¡Algún privilegio tenía que corresponderle!

- **Claves primarias y alternativas**

Las restricciones **check** nos permiten con relativa facilidad imponer condiciones sobre las filas de una tabla que pueden verificarse examinando solamente el registro activo. Cuando las reglas de consistencia involucran a varias filas a la vez, la expresión de estas reglas puede complicarse bastante. En último caso, una combinación de cláusulas **check** y el uso de *triggers* o disparadores nos sirve para expresar *imperativamente* las reglas necesarias. Ahora bien, hay casos típicos de restricciones que afectan a varias filas a la vez que se pueden expresar *declarativament*e; estos casos incluyen a las restricciones de claves primarias y las de integridad referencial. Mediante una clave primaria indicamos que una columna, o una combinación de columnas, debe tener valores únicos para cada fila. Por ejemplo, en una tabla de clientes, el código de cliente no debe repetirse en dos filas diferentes. Esto se expresa de la siguiente forma:

```
create table Clientes( Codigo integer not null primary key, Nombre varchar(30) not
null, /* … */ );
```

Si una columna pertenece a la clave primaria, debe estar especificada como no nula. Observe que en este caso hemos utilizado la restricción a nivel de columna. También es posible tener claves primarias compuestas, en cuyo caso la restricción

hay que expresarla a nivel de tabla. Por ejemplo, en la tabla de detalles de pedidos, la clave primaria puede ser la combinación del número de pedido y el número de línea dentro de ese pedido:

```
create table Detalles( NumPedido integer not null, NumLinea integer not null, /* … */
primary key (NumPedido, NumLinea) );
```

Solamente puede haber una clave primaria por cada tabla. De este modo, la clave primaria representa la *identidad* de los registros almacenados en una tabla: la información necesaria para localizar unívocamente un objeto. No es imprescindible especificar una clave primaria al crear tablas, pero es recomendable como método de trabajo. Sin embargo, es posible especificar que otros grupos de columnas también poseen valores únicos dentro de las filas de una tabla. Estas restricciones son similares en sintaxis y semántica a las claves primarias, y utilizan la palabra reservada **unique**. En la jerga relacional, a estas columnas se le denominan *claves alternativa*s. Una buena razón para tener claves alternativas puede ser que la columna designada como clave primaria sea en realidad una *clave artificia*l. Se dice que una clave es artificial cuando no tiene un equivalente semántico en el sistema que se modela. Por ejemplo, el código de cliente no tiene una existencia *real*; nadie va por la calle con un 666 grabado en la frente. La verdadera clave de un cliente puede ser, además de su alma inmortal, su DNI. Pero el DNI debe almacenarse en una cadena de caracteres, y esto ocupa mucho más espacio que un código numérico. En este caso, el código numérico se utiliza en las referencias a clientes, pues al tener menor tamaño la clave, pueden existir más entradas en un bloque de índice, y el acceso por índices es más eficiente. Entonces, la tabla de clientes puede definirse del siguiente modo:

```
create table Clientes (Codigo integer not null, DNI varchar(9) not null, /* … */
primary key (Codigo), unique (DNI));
```

Por cada clave primaria o alternativa definida, InterBase crea un índice único para mantener la restricción. Este índice se bautiza según el patrón *rdb$primary*N, donde *N* es un número único asignado por el sistema.

- **Integridad referencial**

Un caso especial y frecuente de restricción de integridad es la conocida como restricción de *integridad referencia*l. También se le denomina restricción por *clave externa* o *foránea (foreign key*). Esta restricción especifica que el valor almacenado en un grupo de columnas de una tabla debe encontrarse en los valores de las columnas en alguna fila de otra tabla, o de sí misma. Por ejemplo, en una tabla de pedidos se almacena el código del cliente que realiza el pedido. Este código debe corresponder al código de algún cliente almacenado en la tabla de clientes. La restricción puede expresarse de la siguiente manera:

```
create table Pedidos( Codigo integer not null primary key, Cliente integer not null
references Clientes(Codigo), /* … */);
```

O, utilizando restricciones a nivel de tabla:

```
create table Pedidos(Codigo integer not null, Cliente integer not null, /* … */
primary key (Codigo), foreign key (Cliente) references Clientes(Codigo) );
```

La columna o grupo de columnas a la que se hace referencia en la tabla maestra, la columna *Codigo* de *Clientes* en este caso, debe ser la clave primaria de esta tabla o ser una clave alternativa, esto es, debe haber sido definida una restricción **unique** sobre la misma.

Si todo lo que pretendemos es que no se pueda introducir una referencia a cliente inválida, se puede sustituir la restricción declarativa de integridad referencial por esta cláusula:

```
create table Pedidos( /* … */ check (Cliente in (select Codigo from Clientes)));
```

Esta restricción **check** sencillamente comprueba que la referencia al cliente exista en la columna *Codigo* de la tabla *Cliente*s.

Con lo que llevamos visto hasta ahora estamos plenamente capacitados para crear las diminutas tablas objeto de nuestro proyecto. El código en el que esto se efectúa se presenta a continuación:

```
/* Table: TABLA1, Owner: INVERSOR */
CREATE TABLE TABLA1 (POTENCIA FLOAT,
        TENSION_DE_PANEL FLOAT,
        INTENSIDAD_DE_PANEL FLOAT,
        TENSION_DE_RED FLOAT,
        INTENSIDAD_DE_BOBINA FLOAT,
        FECHA_Y_HORA DATE DEFAULT "NOW",
        NUMERO SMALLINT);

/* Table: TABLA2, Owner: INVERSOR */
CREATE TABLE TABLA2 (NO_INV SMALLINT NOT NULL,
        LIMITE_DE_POTENCIA FLOAT,
        ESTADO VARCHAR(12) default "no" NOT NULL,
      PRIMARY KEY (NO_INV));
```

La primera tabla se limitará a recoger aquellas variables de los inversores fotovoltaicos transmitidas por el *data logger* a través del puerto serie que sean de nuestro interés, junto con el registro del instante en que se produjeron. En la segunda indicaremos qué aparatos están operativos y cuáles están encendidos. Posteriormente, mediante un *trigger* codificado en un *script* SQL, haremos que esta información se traduzca en la creación de registros de la tabla de variables. Asímismo, también dispondremos en esta última tabla de los límites de potencia permisibles en cada inversor, de forma que el *data logger* pueda mantenerlos en condiciones de operación.

- **Acciones referenciales**

Sin embargo, las restricciones de integridad referencial ofrecen más que esta simple comprobación. Cuando tenemos una de estas restricciones, el sistema toma las riendas cuando tratamos de eliminar una fila maestra que tiene filas dependientes asociadas, y cuando tratamos de modificar la clave primaria de una fila maestra con las mismas condiciones. El estándar SQL-3 dicta una serie de posibilidades y reglas, denominadas *acciones referenciale*s, que pueden aplicarse. Lo más sencillo es prohibir estas operaciones, y es la solución que adoptan MS SQL Server (incluyendo la versión 7) y las versiones de InterBase anteriores a la 5. En la sintaxis más completa de SQL-3, esta política puede expresarse mediante la siguiente cláusula:

```
create table Pedidos( Codigo integer not null primary key, Cliente integer not null,
/* … */ foreign key (Cliente) references Clientes(Codigo) on delete no action on
update no action );
```

Otra posibilidad es permitir que la acción sobre la tabla maestra se propague a las filas dependientes asociadas: eliminar un cliente puede provocar la desaparición de todos sus pedidos, y el cambio del código de un cliente modifica todas las referencias a este cliente. Por ejemplo:

```
create table Pedidos( Codigo integer not null primary key, Cliente integer not  null
references Clientes(Codigo) on delete no action on update cascade /* … */ );
```

Observe que puede indicarse un comportamiento diferente para los borrados y para las actualizaciones. En el caso de los borrados, puede indicarse que la eliminación de una fila maestra provoque que en la columna de referencia en las filas de detalles se asigne el valor nulo, o el valor por omisión de la columna de referencia:

```
insert into Empleados(Codigo, Nombre, Apellidos) values(-1, "D'Arche", "Jeanne");
create table Pedidos( Codigo integer not null primary key, Cliente integer not null
references Clientes(Codigo) on delete no action on update cascade, Empleado integer
default -1 not null references Empleados(Codigo) on delete set default on update
cascade /* … */ );
```

InterBase 5 implementa todas estas estrategias, para lo cual necesita crear índices que le ayuden a verificar las restricciones de integridad referencial. La comprobación de la existencia de la referencia en la tabla maestra se realiza con facilidad, pues se trata en definitiva de una búsqueda en el índice único que ya ha sido creado para la gestión de la clave. Para prohibir o propagar los borrados y actualizaciones que afectarían a filas dependientes, la tabla que contiene la cláusula **foreign key** crea automáticamente un índice sobre las columnas que realizan la referencia. De este modo, cuando sucede una actualización en la tabla maestra, se pueden localizar con rapidez las posibles filas afectadas por la operación. Este índice nos ayuda en C++ Builder en la especificación de relaciones *master*/*detail* entre tablas. Los índices creados automáticamente para las relaciones de integridad referencial reciben nombres con el formato *rdb$foreign*N, donde *N* es un número generado automáticamente.

Para la planta fotovoltáica elegiremos la tabla 2 como maestra y elegiremos la columna numero de la tabla 1 para referenciar a la columna primaria de dicha tabla:

```
ALTER TABLE TABLA1 ADD FOREIGN KEY (NUMERO) REFERENCES TABLA2(NO_INV);
```

- **Nombres para las restricciones**

Cuando se define una restricción sobre una tabla, sea una verificación por condición o una clave primaria, alternativa o externa, es posible asignarle un nombre a la restricción. Este nombre es utilizado por InterBase en el mensaje de error que se produce al violarse la restricción, pero su uso fundamental es la manipulación posterior por parte de instrucciones como **alter table**. Por ejemplo:

```
create table Empleados( /* … */ Salario integer default 0, constraint SalarioPositivo
check(Salario >= 0)/* … */ constraint NombreUnico unique(Apellidos, Nombre));
```

También es posible utilizar nombres para las restricciones cuando éstas se expresan a nivel de columna. Las restricciones a las cuales no asignamos nombre reciben uno automáticamente por parte del sistema, siendo este el caso de nuestro

proyecto, ya que no se creyó necesaria la adición de nombres a las restricciones empleadas.

· Definición y uso de dominios

SQL permite definir algo similar a los tipos de datos de los lenguajes tradicionales. Si estamos utilizando cierto tipo de datos con frecuencia, podemos definir un dominio para ese tipo de columna y utilizarlo consistentemente durante la definición del esquema de la base de datos. Un dominio, sin embargo, va más allá de la simple definición del tipo, pues permite expresar restricciones sobre la columna y valores por omisión. La sintaxis de una definición de dominio en InterBase es la siguiente:

```
create domain NombreDominio [as] TipoDeDato [ValorPorOmisión] [not null]
[check(Condición)] [collate Criterio];
```

Cuando sobre un dominio se define una restricción de chequeo, no contamos con el nombre de la columna. Si antes expresábamos la restricción de que los códigos de provincia estuvieran en mayúsculas de esta forma: Provincia varchar(2) **chec**k(Provincia = upper(Provincia)) ahora necesitamos la palabra reservada **value** para referirnos al nombre de la columna:

```
create domain CodProv as varchar(2) check(value = upper(value));
```

El dominio definido, *CodPro*v, puede utilizarse ahora para definir columnas: `Provincia CodProv`

Las cláusulas **check** de las definiciones de dominio no pueden hacer referencia a columnas de otras tablas.

Es aconsejable definir dominios en InterBase por una razón adicional: el Diccionario de Datos de C++ Builder los reconoce y asocia automáticamente a conjuntos de atributos *(attribute set*s). De esta forma, se ahorra mucho tiempo en la configuración de los objetos de acceso a campos.

A pesar de lo anterior, no se estimó oportuno el uso de dominios en este proyecto.

· Creación de índices

Como ya he explicado, InterBase crea índices de forma automática para mantener las restricciones de clave primaria, unicidad y de integridad referencial. En la mayoría de los casos, estos índices bastan para que el sistema funcione eficientemente. No obstante, es necesario en ocasiones definir índices sobre otras columnas. Esta decisión depende de la frecuencia con que se realicen consultas según valores almacenados en estas columnas, o de la posibilidad de pedir que una tabla se ordene de acuerdo al valor de las mismas. Por ejemplo, en la tabla de empleados es sensato pensar que el usuario de la base de datos deseará ver a los empleados listados por orden alfabético, o que querrá realizar búsquedas según un nombre y unos apellidos.

La sintaxis para crear un índice es la siguiente:

```
create [unique] [asc[ending] | desc[ending]] index Indice
on Tabla (Columna [, Columna …])
```

Por ejemplo, para crear un índice sobre los apellidos y el nombre de los empleados necesitamos la siguiente instrucción:

create index NombreEmpleado **on** Empleados(Apellidos, Nombre)

Los índices creados por InterBase son todos sensibles a mayúsculas y minúsculas, y todos son mantenidos por omisión. El concepto de índice definido por expresiones y con condición de filtro es ajeno a la filosofía de SQL; este tipo de índices no se adapta fácilmente a la optimización automática de consultas. InterBase no permite tampoco crear índices sobre columnas definidas con la cláusula **computed by**.

Aunque definamos índices descendentes sobre una tabla en una base de datos SQL, el Motor de Datos de Borland no lo utilizará para ordenar tablas. Exactamente lo que sucede es que el BDE no permite que una tabla (no una consulta) pueda estar ordenada descendentemente por alguna de sus columnas, aunque la tabla mencione un índice descendente en su propiedad *IndexNam*e. En tal caso, el orden que se establece utiliza las mismas columnas del índice, pero ascendentemente.

Hay otro problema relacionado con los índices de InterBase. Al parecer, estos índices solamente pueden recorrerse en un sentido. Si definimos un índice ascendente sobre determinada columna de una tabla, y realizamos una consulta sobre la tabla con los resultados ordenados descendentemente por el valor de esa columna, InterBase no podrá aprovechar el índice creado.

Para la tabla2 pareció interesante la creación de un índice que permitiera ordenar los inversores según la máxima potencia de que disponían.

- **Modificación de tablas e índices**

Es bastante común que una vez terminado el diseño de una base de datos surja la necesidad de añadir nuevas columnas a las tablas para almacenar información imprevista, o que tengamos que modificar el tipo o las restricciones activas sobre una columna determinada. La forma más simple de la instrucción de modificación de tablas es la que elimina una columna de la misma:

```
alter table Tabla drop Columna [, Columna …]
```

También se puede eliminar una restricción si conocemos su nombre. Por ejemplo,

esta instrucción puede originar graves disturbios sociales:

```
alter table Empleados drop constraint SalarioPositivo;
```

Se pueden añadir nuevas columnas o nuevas restricciones sobre una tabla existente:

```
alter table Empleados add EstadoCivil varchar(8);
alter table Empleados
add check (EstadoCivil in ("Soltero","Casado","Polígamo"));
```

Para los índices existen también instrucciones de modificación. En este caso, el único parámetro que se puede configurar es si el índice está activo o no:

```
alter index Indice (active | inactive);
```

Si un índice está inactivo, las modificaciones realizadas sobre la tabla no se propagan al índice, por lo cual necesitan menos tiempo para su ejecución. Si se va a efectuar una entrada masiva de datos, quizás sea conveniente desactivar algunos de los índices secundarios, para mejorar el rendimiento de la operación. Luego, al activar el índice, éste se reconstruye dando como resultado una estructura de datos perfectamente balanceada. Estas instrucciones pueden ejecutarse periódicamente, para garantizar índices con tiempo de acceso óptimo:

```
alter index NombreEmpleado inactive;
alter index NombreEmpleado active;
```

Tras crear las tablas, pareció conveniente añadir una nueva restricción a la tabla2, en la cual imponer que la columna estado sólo admitiera tres valores: encendido, apagado o inexistente:

```
ALTER TABLE TABLA2 ADD
        check(estado in ("on","off","no"));
```

Otra instrucción que puede mejorar el rendimiento del sistema y que está relacionada con los índices es **set statistic**s. Este comando calcula las estadísticas de uso de las claves dentro de un índice. El valor obtenido, conocido como *selectividad* del índice, es utilizado por InterBase para elaborar el plan de implementación de consultas. Normalmente no hay que invocar a esta función explícitamente, pero si las estadísticas de uso del índice han variado mucho es quizás apropiado utilizar la instrucción:

```
set statistics index NombreEmpleado;
```

Por último, las instrucciones **drop** nos permiten borrar objetos definidos en la base de datos, tanto tablas como índices:

```
drop table Tabla;
drop index Indice;
```

· Creación de vistas

Uno de los recursos más potentes de SQL, y de las bases de datos relacionales en general, es la posibilidad de definir tablas "virtuales" a partir de los datos almacenados en tablas "físicas". Para definir una de estas tablas virtuales hay que definir qué operaciones relacionales se aplican a qué tablas bases. Este tipo de tabla recibe el nombre de *vist*a.

Aunque en principio se estudió la posibilidad de incluir vistas en el proyecto según los privilegios de que dispusiera el usuario del programa, finalmente se optó por restringir las posibilidades al público en general de otras formas.

· Creación de usuarios

InterBase soporta el concepto de usuarios a nivel del servidor, no de las bases de datos. Inicialmente, todos los servidores definen un único usuario especial: *SYSDB*A. Este usuario tiene los derechos necesarios para crear otros usuarios y asignarles contraseñas. Toda esa información se almacena en la base de datos *isc4.gd*b, que se

instala automáticamente con InterBase. La gestión de los nombres de usuarios y sus contraseñas se realiza mediante la utilidad *Server Manage*r.

Dentro de esta aplicación, hay que ejecutar el comando de menú *Tasks*|*User securit*y, para llegar al diálogo con el que podemos añadir, modificar o eliminar usuarios.

El nombre del usuario *SYSDBA* no puede cambiarse, pero es casi una obligación cambiar su contraseña en cuanto termina la instalación de InterBase. Sin embargo, podemos eliminar al administrador de la lista de usuarios del sistema. Si esto sucede, ya no será posible añadir o modificar nuevos usuarios en ese servidor. Así que debe tenerse cuidado con lo que se hace.

El sistema de seguridad explicado tiene un par de aparentes "fallos". En primer lugar, cualquier usuario con acceso al disco duro puede sustituir el fichero *isc4.gdb* con uno suyo. Más grave aún: si copiamos el fichero *gdb* de la base de datos en un servidor en el cual conozcamos la contraseña del administrador, tendremos acceso total a los datos, aunque este acceso nos hubiera estado vedado en el servidor original.

En realidad, el fallo consiste en permitir que cualquiera pueda acceder a nuestras apreciadas bases de datos. Así que, antes de planear la protección del sistema de gestión de base de datos (ya sea InterBase o cualquier otro), debería tratarse de controlar el acceso al servidor de la gente indeseable.

En principio se usará un único usuario INVERSOR en nuestro proyecto, con la sencilla clave “i” para facilitar el desarrollo de la aplicación en *Builder*, aunque por supuesto se recomienda el posterior cambio de la misma.

· Asignación de usuarios

Una vez creados los objetos de la base de datos, es necesario asignar derechos sobre los mismos a los demás usuarios. Inicialmente, el dueño de una tabla es el usuario que la crea, y tiene todos los derechos de acceso sobre la tabla. Los derechos de acceso indican qué operaciones pueden realizarse con la tabla. Naturalmente, los nombres de estos derechos o privilegios coinciden con los nombres de las operaciones correspondientes:

Privilegio	Operación
select	Lectura de datos
update	Modificación de datos existentes
insert	Creación de nuevos registros
delete	Eliminación de registros
all	Los cuatro privilegios anteriores
execute	Ejecución (para procedimientos almacenados)

La instrucción que otorga derechos sobre una tabla es la siguiente:

```
grant Privilegios on Tabla to Usuarios [with grant option]
```

Por ejemplo:

```
/* Derecho de sólo-lectura al público en general */
grant select on Articulos to public;

/* Todos los derechos a un par de usuarios */
grant all privileges on Clientes to Spade, Marlowe;

/* Monsieur Poirot sólo puede modificar salarios (¡qué peligro!) */
grant update(Salario) on Empleados to Poirot;

/* Privilegio de inserción y borrado, con opción de concesión */
grant insert, delete on Empleados to Vance with grant option;
```

Se han mostrado unas cuantas posibilidades de la instrucción. En primer lugar, podemos utilizar la palabra clave **public** cuando queremos conceder ciertos derechos a todos los usuarios. En caso contrario, podemos especificar uno o más usuarios como destinatarios del privilegio. Luego, podemos ver que el privilegio **update** puede llevar entre paréntesis la lista de columnas que pueden ser modificadas. Por último, vemos que a Mr. Philo Vance no solamente le permiten contratar y despedir empleados, sino que también, gracias a la cláusula **with grant option**, puede conceder estos derechos a otros usuarios, aún no siendo el creador de la tabla. Esta opción debe utilizarse con cuidado, pues puede provocar una propagación descontrolada de privilegios entre usuarios indeseables.

¿Y qué pasa si otorgamos privilegios y luego nos arrepentimos? No hay problema, pues para esto tenemos la instrucción **revoke**:

```
revoke [grant option for] Privilegios on Tabla from Usuarios
```

Hay que tener cuidado con los privilegios asignados al público. La siguiente instrucción no afecta a los privilegios de Sam Spade sobre la tabla de artículos, porque antes se le ha concedido al público en general el derecho de lectura sobre la misma:

```
/* Spade se ríe de este ridículo intento */
revoke all on Articulos from Spade;
```

Existen variantes de las instrucciones **grant** y **revoke** pensadas para asignar y retirar privilegios sobre tablas a procedimientos almacenados, y para asignar y retirar derechos de ejecución de procedimientos a usuarios.

· Roles

Los roles son una especificación del SQL-3 que InterBase 5 ha implementado. Si los usuarios se almacenan y administran a nivel de servidor, los roles, en cambio, se definen a nivel de cada base de datos. De este modo, podríamos trasladar con más facilidad una base de datos desarrollada en determinado servidor, con sus usuarios particulares, a otro servidor, en el cual existiera históricamente otro conjunto de usuarios.

El sujeto de la validación por contraseña sigue siendo el usuario. La relación entre usuarios y roles es la siguiente: un usuario puede *asumir* un rol al conectarse a la base de datos; actualmente, InterBase no permite asumir roles después de este momento. A un rol se le pueden otorgar privilegios exactamente igual que a un

usuario, utilizando **grant** y **revoke**. Cuando un usuario asume un rol, los privilegios del rol se suman a los privilegios que se le han concedido como usuario. Por supuesto, un usuario debe contar con la autorización para asumir un rol, lo cual se hace también mediante **grant** y **revoke**.

¿Un ejemplo? Primero necesitamos crear los roles adecuados en la base de datos:

```
create role Domador;
create role Payaso;
create role Mago;
```

Ahora debemos asignar los permisos sobre tablas y otros objetos a los roles. Esto no impide que, en general, se puedan también asignar permisos específicos a usuarios puntuales:

```
grant all privileges on Animales to Domador, Mago;
grant select on Animales to Payaso;
```

Hasta aquí no hemos mencionado a los usuarios, por lo que los resultados de estas instrucciones son válidos de servidor a servidor. Finalmente, debemos asignar los usuarios en sus respectivos roles, y esta operación sí depende del conjunto de usuarios de un servidor:

```
grant Payaso to Bill, Steve, RonaldMcDonald;
grant Domador to Ian with admin option;
```

La opción **with admin option** permitiría asignar el rol de domador a otros usuarios. De este modo, siempre habría quien se ocupara de los animales cuando se ausentara del circo por vacaciones.

La pregunta importante es: ¿cómo puede el usuario indicar a InterBase que desea asumir determinado rol en una conexión? Si nos conectamos con las herramientas del propio InterBase, comprobaremos que existe un cuadro de edición que nos pregunta por el rol a asumir. Por supuesto, podemos dejarlo vacío, si nos queremos atener exclusivamente a nuestros privilegios como usuarios. Pero si la conexión la realizamos desde C++ Builder tendremos que programar un poco. Los roles se añaden al soporte de InterBase del BDE en la versión 5.0.1.23. El BDE que acompañaba a Delphi 4, por ejemplo, no contaba con esta posibilidad. El nuevo SQL Link de InterBase introduce el parámetro *ROLE NAM*E, y es aquí donde debemos indicar el rol del usuario. Lamentablemente, el diálogo de conexión del componente *TDatabase* tampoco considera a los roles, por lo que si necesitamos que el usuario dicte dinámicamente los parámetros de conexión tendremos que interceptar el evento *OnLogin* del componente *TDatabas*e.

En un principio, no se tenía muy clara la utilidad que podrían tener los roles en el proyecto, por lo que se incluyeron en él para dar los privilegios de acceso oportunos en un futuro a los becarios que se encargaron de ayudar al autor del proyecto en el desarrollo del mismo, mientras que al público en general sólo se le permitía seleccionar:

```
/* Grant role for this database */

CREATE ROLE JUANANTONIO;
CREATE ROLE LUIS;
```

```
/* Grant permissions for this database */

GRANT DELETE, INSERT, SELECT, UPDATE, REFERENCES ON TABLA1 TO DANIEL;
GRANT DELETE, INSERT, SELECT, UPDATE, REFERENCES ON TABLA1 TO JUANANTONIO;
GRANT DELETE, INSERT, SELECT, UPDATE, REFERENCES ON TABLA1 TO LUIS;
GRANT SELECT ON TABLA1 TO PUBLIC;
GRANT DELETE, INSERT, SELECT, UPDATE, REFERENCES ON TABLA2 TO DANIEL;
GRANT DELETE, INSERT, SELECT, UPDATE, REFERENCES ON TABLA2 TO JUANANTONIO;
GRANT DELETE, INSERT, SELECT, UPDATE, REFERENCES ON TABLA2 TO LUIS;
GRANT SELECT ON TABLA2 TO PUBLIC;
```

1.5. Procedimientos almacenados y triggers

1.5.1. Qué es un procedimiento almacenado. Ventajas e inconvenientes.

En este capítulo se completa la presentación de los sublenguajes de SQL, mostrando el lenguaje de definición de procedimientos de InterBase.

Desgraciadamente, los lenguajes de procedimientos de los distintos sistemas de bases de datos difieren entre ellos, al no existir todavía un estándar al respecto. El dialecto de InterBase para procedimientos es el que más se asemeja al propuesto en el borrador del estándar SQL-3. De cualquier manera, las diferencias entre dialectos no son demasiadas.

¿Para qué usar procedimientos almacenados? Un *procedimiento almacenado (stored procedur*e) es, sencillamente, un algoritmo cuya definición reside en la base de datos, y que es ejecutado por el servidor del sistema. Aunque SQL-3 define formalmente un lenguaje de programación para procedimientos almacenados, cada uno de los sistemas de bases de datos importantes a nivel comercial implementa su propio lenguaje para estos recursos. InterBase ofrece un dialecto parecido a la propuesta de SQL-3; Oracle tiene un lenguaje llamado PL-SQL; Microsoft SQL Server ofrece el denominado Transact-SQL. No obstante, las diferencias entre estos lenguajes son mínimas, principalmente sintácticas, siendo casi idénticas las capacidades expresivas.

El uso de procedimientos almacenados ofrece las siguientes ventajas:

- Los procedimientos almacenados ayudan a mantener la consistencia de la base de datos. Las instrucciones básicas de actualización, **update**, **insert** y **delete**, pueden combinarse arbitrariamente si dejamos que el usuario tenga acceso ilimitado a las mismas. No toda combinación de actualizaciones cumplirá con las reglas de consistencia de la base de datos. Hemos visto que algunas de estas reglas se pueden expresar declarativamente durante la definición del esquema relacional. El mejor ejemplo son las restricciones de integridad referencial. Pero, ¿cómo expresar declarativamente que para cada artículo presente en un pedido, debe existir un registro correspondiente en la tabla de movimientos de un almacén? Una posible solución es prohibir el uso directo de las instrucciones de actualización, revocando permisos de acceso al público, y permitir la modificación de datos solamente a partir de procedimientos almacenados.

- Los procedimientos almacenados permiten superar las limitaciones del lenguaje de consultas. SQL no es un lenguaje completo. Un problema típico en que falla es en

la definición de *clausuras relacionale*s. Tomemos como ejemplo una tabla con dos columnas: *Objeto* y *Part*e. Esta tabla contiene pares como los siguientes:

Objeto	**Parte**
Cuerpo humano	Cabeza
Cuerpo humano	Tronco
Cabeza	Ojos
Cabeza	Boca
Boca	Dientes

¿Puede el lector indicar una consulta que liste todas las partes incluidas en la cabeza? Lo que falla es la posibilidad de expresar algoritmos recursivos. Para resolver esta situación, los procedimientos almacenados pueden implementarse de forma tal que devuelvan conjuntos de datos, en vez de valores escalares. En el cuerpo de estos procedimientos se pueden realizar, entonces, llamadas recursivas.

- Los procedimientos almacenados pueden reducir el tráfico en la red. Un procedimiento almacenado se ejecuta en el servidor, que es precisamente donde se encuentran los datos. Por lo tanto, no tenemos que explorar una tabla de arriba a abajo desde un ordenador cliente para extraer el promedio de ventas por empleado durante el mes pasado. Además, por regla general el servidor es una máquina más potente que las estaciones de trabajo, por lo que puede que ahorremos tiempo de ejecución para una petición de información. No conviene, sin embargo, abusar de esta última posibilidad, porque una de las ventajas de una red consiste en distribuir el tiempo de procesador.

- Con los procedimientos almacenados se puede ahorrar tiempo de desarrollo. Siempre que existe una información, a alguien se le puede ocurrir un nuevo modo de aprovecharla. En un entorno cliente/servidor es típico que varias aplicaciones diferentes trabajen con las mismas bases de datos. Si centralizamos en la propia base de datos la imposición de las reglas de consistencia, no tendremos que volverlas a programar de una aplicación a otra. Además, evitamos los riesgos de una mala codificación de estas reglas, con la consiguiente pérdida de consistencia.

Como todas las cosas de esta vida, los procedimientos almacenados también tienen sus inconvenientes. Ya he mencionado uno de ellos: si se centraliza todo el tratamiento de las reglas de consistencia en el servidor, corremos el riesgo de saturar los procesadores del mismo. El otro inconveniente es la poca portabilidad de las definiciones de procedimientos almacenados entre distintos sistemas de bases de datos. Si hemos desarrollado procedimientos almacenados en InterBase y queremos migrar nuestra base de datos a Oracle (o viceversa), estaremos obligados a partir "casi" de cero; algo se puede aprovechar, de todos modos.

1.5.2. Cómo se utiliza un procedimiento almacenado

Un procedimiento almacenado puede utilizarse desde una aplicación cliente, desarrollada en cualquier lenguaje de programación que pueda acceder a la interfaz

de programación de la base de datos, o desde las propias utilidades interactivas del sistema.

En la VCL tenemos el componente *TStoredProc*, diseñado para la ejecución de estos procedimientos. Posteriormente veremos cómo suministrar parámetros, ejecutar procedimientos y recibir información utilizando este componente.

En el caso de InterBase, también es posible ejecutar un procedimiento almacenado directamente desde la aplicación *Windows ISQL*, mediante la siguiente instrucción:

```
execute procedure NombreProcedimiento [ListaParámetros];
```

La misma instrucción puede utilizarse en el lenguaje de definición de procedimientos y *triggers* para llamar a un procedimiento dentro de la definición de otro. Es posible también definir procedimientos recursivos. InterBase permite hasta un máximo de 1000 llamadas recursivas por procedimiento.

•El carácter de terminación

Los procedimientos almacenados de InterBase deben necesariamente escribirse en un fichero *script* de SQL. Más tarde, este fichero debe ser ejecutado desde la utilidad *Windows ISQL* para que los procedimientos sean incorporados a la base de datos. Hemos visto las reglas generales del uso de *scripts* en InterBase en el capítulo de introducción a SQL. Ahora tenemos que estudiar una característica de estos *scripts* que anteriormente hemos tratado superficialmente: el carácter de terminación.

Por regla general, cada instrucción presente en un *script* es leída y ejecutada de forma individual y secuencial. Esto quiere decir que el intérprete de *scripts* lee del fichero hasta que detecta el fin de instrucción, ejecuta la instrucción recuperada, y sigue así hasta llegar al final del mismo. El problema es que este proceso de extracción de instrucciones independientes se basa en la detección de un carácter especial de terminación. Por omisión, este carácter es el punto y coma; el lector habrá observado que todos los ejemplos de instrucciones SQL que deben colocarse en *scripts* han sido, hasta el momento, terminados con este carácter. Ahora bien, al tratar con el lenguaje de procedimientos y *triggers* encontraremos instrucciones y cláusulas que deben terminar con puntos y comas. Si el intérprete de *scripts* tropieza con uno de estos puntos y comas pensará que se encuentra frente al fin de la instrucción, e intentará ejecutar lo que ha leído hasta el momento; casi siempre, una instrucción incompleta. Por lo tanto, debemos cambiar el carácter de terminación de *Windows ISQL* cuando estamos definiendo *triggers* o procedimientos almacenados. La instrucción que nos ayuda para esto es la siguiente:

```
set term Terminador
```

Como carácter de terminación podemos escoger cualquier carácter o combinación de los mismos lo suficientemente rara como para que no aparezca dentro de una instrucción del lenguaje de procedimientos. Por ejemplo, podemos utilizar el acento circunflejo:

```
set term ^;
```

Observe cómo la instrucción que cambia el carácter de terminación debe terminar ella misma con el carácter antiguo. Al finalizar la creación de todos los procedimientos que necesitamos, debemos restaurar el antiguo carácter de terminación:

```
set term ;^
```

En lo sucesivo asumiremos que el carácter de terminación ha sido cambiado al acento circunflejo.

- Procedimientos almacenados **en InterBase**

La sintaxis para la creación de un procedimiento almacenado en InterBase es la siguiente:

```
create procedure Nombre
[ ( ParámetrosDeEntrada ) ]
[ returns ( ParámetrosDeSalida ) ]
as CuerpoDeProcedimiento
```

Las cláusulas *ParámetrosDeEntrada* y *ParámetrosDeSalida* representan listas de declaraciones de parámetros. Los parámetros de salida pueden ser más de uno; esto significa que el procedimiento almacenado que retorna valores no se utiliza como si fuese una función de un lenguaje de programación tradicional. El siguiente es un ejemplo de cabecera de procedimiento:

```
create procedure TotalPiezas(PiezaPrincipal char(15))
returns (Total integer)
as
/* ... Aquí va el cuerpo ... */
```

El cuerpo del procedimiento, a su vez, se divide en dos secciones, siendo opcional la primera de ellas: la sección de declaración de variables locales, y una instrucción compuesta, **begin...en**d, que agrupa las instrucciones del procedimiento. Las variables se declaran en este verboso estilo, *á la 197*0:

```
declare variable V1 integer;
declare variable V2 char(50);
```

Estas son las instrucciones permitidas por los procedimientos almacenados de InterBase:

· Asignaciones:

Variable = Expresión

Las variables pueden ser las declaradas en el propio procedimiento, parámetros de entrada o parámetros de salida.

· Llamadas a procedimientos:

```
execute procedure NombreProc [ParsEntrada]
[returning_values ParsSalida]
```

No se admiten expresiones en los parámetros de entrada; mucho menos en los de salida.

· Condicionales:

```
if (Condición) then Instrucción [else Instrucción]
```

· Bucles controlados por condiciones:

```
while (Condición) do Instrucción
```

· Instrucciones SQL:

Cualquier instrucción de manipulación, **insert**, **update** ó **delete**, puede incluirse en un procedimiento almacenado. Estas instrucciones pueden utilizar variables locales y parámetros, siempre que estas variables estén precedidas de dos puntos, para distinguirlas de los nombres de columnas. Por ejemplo, si *Minimo* y *Aumento* son variables o parámetros, puede ejecutarse la siguiente instrucción:

```
update Empleados
set Salario = Salario * :Aumento
where Salario < :Minimo;
```

Se permite el uso directo de instrucciones **select** si devuelven una sola fila; para consultas más generales se utiliza la instrucción **for** que veremos dentro de poco. Estas selecciones únicas van acompañadas por una cláusula **into** para transferir valores a variables o parámetros:

```
select Empresa
from Clientes
where Codigo = 1984
into :NombreEmpresa;
```

· Iteración sobre consultas:

```
for InstrucciónSelect into Variables do Instrucción
```

Esta instrucción recorre el conjunto de filas definido por la instrucción **select**. Para cada fila, transfiere los valores a las variables de la cláusula **into**, de forma similar a lo que sucede con las selecciones únicas, y ejecuta entonces la instrucción de la sección **do**.

· Lanzamiento de excepciones:

```
exception NombreDeExcepción
```

Similar a la instrucción **throw** de C++.

· Captura de excepciones:

```
when ListaDeErrores do Instrucción
```

Similar a la cláusula **catch** de la instrucción **try...catch** de C++. Los errores capturados pueden ser excepciones propiamente dichas o errores reportados con la variable SQLCODE. Estos últimos errores se producen al ejecutarse instrucciones SQL. Las instrucciones **when** deben colocarse al final de los procedimientos.

· Instrucciones de control:

```
exit;
suspend;
```

La instrucción **exit** termina la ejecución del procedimiento actual, y es similar a la instrucción **return** de C++. Por su parte, **suspend** se utiliza en procedimientos que devuelven un conjunto de filas para retornar valores a la rutina que llama a este procedimiento. Con esta última instrucción, se interrumpe temporalmente el procedimiento, hasta que la rutina que lo llama haya procesado los valores retornados.

· Instrucciones compuestas:

```
begin ListaDeInstrucciones end
```

La sintaxis de los procedimientos de InterBase es similar a la de Pascal. A diferencia de este último lenguaje, la palabra **end** no puede tener un punto y coma a continuación.

Se muestran ahora un par de procedimientos sencillos, que ejemplifiquen el uso de estas instrucciones. El siguiente procedimiento sirve para recalcular la suma total de un pedido, si se suministra el número de pedido correspondiente:

```
create procedure RecalcularTotal(NumPed int) as
declare variable Total integer;
begin
       select sum(Cantidad * PVP * (100 - Descuento) / 100)
       from Detalles, Articulos
       where Detalles.RefArticulo = Articulos.Codigo
       and Detalles.RefPedido = :NumPed
       into :Total;
       if (Total is null) then
       Total = 0;
       update Pedidos
       set Total = :Total
       where Numero = :NumPed;
end ^
```

El procedimiento consiste básicamente en una instrucción **select** que calcula la suma de los totales de todas las líneas de detalles asociadas al pedido; esta instrucción necesita mezclar datos provenientes de las líneas de detalles y de la tabla de artículos. Si el valor total es nulo, se cambia a cero. Esto puede suceder si el pedido no tiene líneas de detalles; en este caso, la instrucción **select** retorna el valor nulo. Finalmente, se localiza el pedido indicado y se le actualiza el valor a la columna *Total*, utilizando el valor depositado en la variable local del mismo nombre.

El procedimiento que definimos a continuación se basa en el anterior, y permite recalcular los totales de todas las filas almacenadas en la tabla de pedidos; de este modo ilustramos el uso de las instrucciones **for select ... do** y **execute procedure**:

```
create procedure RecalcularPedidos as
declare variable Pedido integer;
begin
       for select Numero from Pedidos into :Pedido do
       execute procedure RecalcularTotal :Pedido;
end ^
```

- **Procedimientos que devuelven un conjunto de datos**

Antes se mencionó la posibilidad de superar las restricciones de las expresiones **select** del modelo relacional mediante el uso de procedimientos almacenados. Un procedimiento puede diseñarse de modo que devuelva un conjunto de filas; para esto hay que utilizar la instrucción **suspend**, que transfiere el control temporalmente a la rutina que llama al procedimiento, para que ésta pueda hacer algo con los valores asignados a los parámetros de salida. Esta técnica es poco habitual en los lenguajes de programación más extendidos; si se quiere encontrar algo parecido, pueden desenterrarse los *iteradores* del lenguaje CLU, diseñado por Barbara Liskov a mediados de los setenta.

Supongamos que necesitamos obtener la lista de los primeros veinte, treinta o mil cuatrocientos números primos. Comencemos por algo fácil, con la función que analiza un número y dice si es primo o no:

```
create procedure EsPrimo(Numero integer)
       returns (Respuesta integer) as
declare variable I integer;
begin
       I = 2;
       while (I < Numero) do
       begin
              if (cast((Numero / I) as integer) * I = Numero) then
              begin
                     Respuesta = 0;
                     exit;
              end
              I = I + 1;
       end
       Respuesta = 1;
end ^
```

Hay implementaciones más eficientes, pero no se quería complicar mucho el ejemplo. Observe, de paso, la pirueta que ha tenido que realizarse para ver si el número es divisible por el candidato a divisor. Se ha utilizado el criterio del lenguaje C para las expresiones lógicas: devuelvo 1 si el número es primo, y 0 si no lo es. Recuerde que InterBase no tiene un tipo *Boolean*.

Ahora, en base al procedimiento anterior, implementamos el nuevo procedimiento *Primos*:

```
create procedure Primos(Total integer)
       returns (Primo Integer) as
declare variable I integer;
declare variable Respuesta integer;
begin
       I = 0;
       Primo = 2;
       while (I < Total) do
       begin
              execute procedure EsPrimo Primo
              returning_values Respuesta;
              if (Respuesta = 1) then
              begin
                     I = I + 1;
                     suspend; /* ¡¡¡ Nuevo !!! */
              end
       Primo = Primo + 1;
       end
end ^
```

Este procedimiento puede ejecutarse en dos contextos diferentes: como un procedimiento normal, o como procedimiento de selección. Como procedimiento normal, utilizamos la instrucción **execute procedure**, como hasta ahora:

```
execute procedure Primos(100);
```

No obstante, no van a resultar tan sencillas las cosas. Esta llamada, si se realiza desde Windows ISQL, solamente devuelve el primer número primo (era el 2, ¿o no?). El problema es que, en ese contexto, la primera llamada a **suspend** termina completamente el algoritmo.

La segunda posibilidad es utilizar el procedimiento como si fuera una tabla o vista. Desde Windows ISQL podemos lanzar la siguiente instrucción, que nos mostrará los primeros cien números primos:

select * **from** Primos(100);

Por supuesto, el ejemplo anterior se refiere a una secuencia aritmética. En la práctica, un procedimiento de selección se implementa casi siempre llamando a **suspend** dentro de una instrucción **for...do**, que recorre las filas de una consulta.

- **Recorriendo un conjunto de datos**

En esta sección se mostraran un par de ejemplos más complicados de procedimientos que utilizan la instrucción **for...select** de InterBase. El primero tiene que ver con un sistema de entrada de pedidos. Supongamos que queremos actualizar las existencias en el inventario después de haber grabado un pedido. Tenemos dos posibilidades, en realidad: realizar esta actualización mediante un trigger que se dispare cada vez que se guarda una línea de detalles, o ejecutar un procedimiento almacenado al finalizar la grabación de todas las líneas del pedido.

La primera técnica será explicada en breve, pero adelanto en estos momentos que tiene un defecto. Pongamos como ejemplo que dos usuarios diferentes están pasando por el cajero, simultáneamente. El primero saca un pack de Coca-Colas de la cesta de la compra, mientras el segundo pone Pepsis sobre el mostrador. Si, como es de esperar, la grabación del pedido tiene lugar mediante una transacción, al dispararse el trigger se han modificado las filas de estas dos marcas de bebidas, y se han bloqueado hasta el final de la transacción. Ahora, inesperadamente, el primer usuario saca Pepsis mientras el segundo nos sorprende con Coca-Colas; son unos fanáticos de las bebidas americanas estos individuos. El problema es que el primero tiene que esperar a que el segundo termine para poder modificar la fila de las Pepsis, mientras que el segundo se halla en una situación similar.

Esta situación se denomina abrazo mortal (deadlock) y realmente no es problema alguno para InterBase, en el cual los procesos fallan inmediatamente cuando se les niega un bloqueo (Realmente, es el BDE quien configura a InterBase de este modo. En la versión 5.0.1.24 se introduce el nuevo parámetro WAIT ON LOCKS, para modificar este comportamiento.) Pero puede ser un peligro en otros sistemas con distinta estrategia de espera. La solución más común consiste en que cuando un proceso necesita bloquear ciertos recursos, lo haga siempre en el mismo orden. Si nuestros dos consumidores de líquidos oscuros con burbujas hubieran facturado sus compras en orden alfabético, no se hubiera producido este conflicto. Por supuesto,

esto descarta el uso de un trigger para actualizar el inventario, pues hay que esperar a que estén todos los productos antes de ordenar y realizar entonces la actualización. El siguiente procedimiento se encarga de implementar el algoritmo explicado:

```
create procedure ActualizarInventario(Pedido integer) as
declare variable CodArt integer;
declare variable Cant integer;
begin
       for select RefArticulo, Cantidad
       from Detalles
       where RefPedido = :Pedido
       order by RefArticulo
       into :CodArt, :Cant do
       update Articulos
       set Pedidos = Pedidos + :Cant
       where Codigo = :CodArt;
end ^
```

Otro ejemplo: necesitamos conocer los diez mejores clientes de nuestra tienda. Pero sólo los diez primeros, y no vale mirar hacia otro lado cuando aparezca el decimoprimero. Algunos sistemas SQL tienen extensiones con este propósito **(top** en SQL Server; **fetch first** en DB2), pero no InterBase. Este procedimiento, que devuelve un conjunto de datos, nos servirá de ayuda:

```
create procedure MejoresClientes(Rango integer)
returns (Codigo int, Nombre varchar(30), Total int) as
begin
       for select Codigo, Nombre, sum(Total)
       from Clientes, Pedidos
       where Clientes.Codigo = Pedidos.Cliente
       order by 3 desc
       into :Codigo, :Nombre, :Total do
       begin
              suspend;
              Rango = Rango - 1;
              if (Rango = 0) then
              exit;
       end
end ^
```

Entonces podremos realizar consultas como la siguiente:

```
select *
from MejoresClientes(10)
```

1.5.3. Triggers, o disparadores

Una de las posibilidades más interesantes de los sistemas de bases de datos relacionales son los *trigger*s, o disparadores; en adelante, se utilizará preferentemente la palabra inglesa original. Se trata de un tipo de procedimiento almacenado que se activa automáticamente al efectuar operaciones de modificación sobre ciertas tablas de la base de datos.

La sintaxis de la declaración de un *trigger* es la siguiente:

```
create trigger NombreTrigger for Tabla [active | inactive]
{before | after} {delete | insert | update}
[position Posición]
as CuerpoDeProcedimiento
```

El cuerpo de procedimiento tiene la misma sintaxis que los cuerpos de los procedimientos almacenados. Las restantes cláusulas del encabezamiento de esta instrucción tienen el siguiente significado:

Cláusula	Significado
NombreTrigger	El nombre que se le va a asignar al *trigger*
Tabla	El nombre de la tabla a la cual está asociado
active | **inactive**	Puede crearse inactivo, y activarse después
before | **after**	Se activa antes o después de la operación
delete | **insert** | **update**	Qué operación provoca el disparo del *trigger*
position	Orden de disparo para la misma operación

A diferencia de otros sistemas de bases de datos, los *triggers* de InterBase se definen para una sola operación sobre una sola tabla. Si queremos compartir código para eventos de actualización de una o varias tablas, podemos situar este código en un procedimiento almacenado y llamarlo desde los diferentes *triggers* definidos.

Un parámetro interesante es el especificado por **position**. Para una operación sobre una tabla pueden definirse varios *trigger*s. El número indicado en **position** determina el orden en que se disparan los diferentes sucesos; mientras más bajo sea el número, mayor será la prioridad. Si dos *triggers* han sido definidos con la misma prioridad, el orden de disparo entre ellos será aleatorio.

Hay una instrucción similar que permite modificar algunos parámetros de la definición de un *trigger*, como su orden de disparo, si está activo o no, o incluso su propio cuerpo:

```
alter trigger NombreTrigger [active | inactive]
[{before | after} {delete | insert | update}]
[position Posición]
[as CuerpoProcedimiento]
```

Podemos eliminar completamente la definición de un *trigger* de la base de datos mediante la instrucción:

```
drop trigger NombreTrigger
```

• Las variables *new* y *old*

Dentro del cuerpo de un *trigger* pueden utilizarse las variables predefinidas *new* y *old*. Estas variables hacen referencia a los valores nuevos y anteriores de las filas involucradas en la operación que dispara el *trigger*. Por ejemplo, en una operación de modificación **update**, *old* se refiere a los valores de la fila antes de la modificación y *new* a los valores después de modificados. Para una inserción, solamente tiene sentido la variable *new*, mientras que para un borrado, solamente tiene sentido *old*.

El siguiente *trigger* hace uso de la variable *new*, para acceder a los valores del nuevo registro después de una inserción:

```
create trigger UltimaFactura for Pedidos
active after insert position 0 as
declare variable UltimaFecha date;
begin
       select UltimoPedido
       from Clientes
       where Codigo = new.RefCliente
       into :UltimaFecha;
       if (UltimaFecha < new.FechaVenta) then
       update Clientes
       set UltimoPedido = new.FechaVenta
       where Codigo = new.RefCliente;
end ^
```

Este *trigger* sirve de contraejemplo a un error muy frecuente en la programación SQL. La primera instrucción busca una fila particular de la tabla de clientes y, una vez encontrada, extrae el valor de la columna *UltimoPedido* para asignarlo a la variable local *UltimaFech*a. El error consiste en pensar que esta instrucción, a la vez, deja a la fila encontrada como "fila activa". El lenguaje de *triggers* y procedimientos almacenados de InterBase, y la mayor parte de los restantes sistemas, no utiliza "filas activas".

Es por eso que en la instrucción **update** hay que incluir una cláusula **where** para volver a localizar el registro del cliente. De no incluirse esta cláusula, cambiaríamos la fecha para *todos* los clientes. Es posible cambiar el valor de una columna correspondiente a la variable *ne*w, pero solamente si el *trigger* se define "antes" de la operación de modificación. En cualquier caso, el nuevo valor de la columna se hace efectivo después de que la operación tenga lugar.

- **Más ejemplos de *triggers***

Para mostrar el uso de *trigger*s, las variables *new* y *old* y los procedimientos almacenados, se mostrará cómo se puede actualizar automáticamente el inventario de artículos y el total almacenado en la tabla de pedidos en la medida en que se realizan actualizaciones en la tabla que contiene las líneas de detalles.

Necesitaremos un par de procedimientos auxiliares para lograr una implementación más modular. Uno de estos procedimientos, *RecalcularTota*l, debe actualizar el total de venta de un pedido determinado, y se programó antes. Repito aquí su código, para mayor comodidad:

```
create procedure RecalcularTotal(NumPed int) as
declare variable Total integer;
begin
       select sum(Cantidad * PVP * (100 - Descuento) / 100)
       from Detalles, Articulos
       where Detalles.RefArticulo = Articulos.Codigo
       and Detalles.RefPedido = :NumPed
       into :Total;
       if (Total is null) then
       Total = 0;
       update Pedidos
       set Total = :Total
       where Numero = :NumPed;
end ^
```

El otro procedimiento debe modificar el inventario de artículos. Su implementación es muy simple:

```
create procedure ActInventario(CodArt integer, Cant Integer) as
begin
update Articulos
set Pedidos = Pedidos + :Cant
```

```
where Codigo = :CodArt;
end ^
```

Ahora le toca el turno a los *trigger*s. Los más sencillos son los relacionados con la inserción y borrado; en el primero utilizaremos la variable *ne*w, y en el segundo, *ol*d:

```
create trigger NuevoDetalle for Detalles
active after insert position 1 as
begin
       execute procedure RecalcularTotal new.RefPedido;
       execute procedure ActInventario
              new.RefArticulo, new.Cantidad;
end ^

create trigger EliminarDetalle for Detalles
active after delete position 1 as
declare variable Decremento integer;
begin
       Decremento = - old.Cantidad;
       execute procedure RecalcularTotal old.RefPedido;
       execute procedure ActInventario
              old.RefArticulo, :Decremento;
end ^
```

Es curiosa la forma en que se pasan los parámetros a los procedimientos almacenados. Debe tomarse nota, en particular, de que hemos utilizado una variable local, *Decremen*to, en el *trigger* de eliminación. Esto es así porque no se puede pasar expresiones como parámetros a los procedimientos almacenados, ni siquiera para los parámetros de entrada.

Finalmente, nos queda el *trigger* de modificación:

```
create trigger ModificarDetalle for Detalles
active after update position 1 as
declare variable Decremento integer;
begin
       execute procedure RecalcularTotal new.RefPedido;
       if (new.RefArticulo <> old.RefArticulo) then
       begin
              Decremento = -old.Cantidad;
              execute procedure ActInventario
              old.RefArticulo, :Decremento;
              execute procedure ActInventario
                     new.RefArticulo, new.Cantidad;
       end
       else
       begin
              Decremento = new.Cantidad - old.Cantidad;
              execute procedure ActInventario
                     new.RefArticulo, :Decremento;
       end
end ^
```

Observe cómo comparamos el valor del código del artículo antes y después de la operación. Si solamente se ha producido un cambio en la cantidad vendida, tenemos que actualizar un solo registro de inventario; en caso contrario, tenemos que actualizar dos registros. No hemos tenido en cuenta la posibilidad de modificar el pedido al cual pertenece la línea de detalles. Suponemos que esta operación no va a permitirse, por carecer de sentido, en las aplicaciones clientes.

1.5.4. Generadores

Los *generadores (generator*s) son un recurso de InterBase para poder disponer de valores secuenciales, que pueden utilizarse, entre otras cosas, para garantizar la unicidad de las claves artificiales. Un generador se crea, del mismo modo que los procedimientos almacenados y *trigger*s, en un fichero *script* de SQL. El siguiente ejemplo muestra cómo crear un generador:

```
create generator CodigoEmpleado;
```

Un generador define una variable interna persistente, cuyo tipo es un entero de 32 bits. Aunque esta variable se inicializa automáticamente a 0, tenemos una instrucción para cambiar el valor de un generador:

```
set generator CodigoEmpleado to 1000;
```

Por el contrario, no existe una instrucción específica que nos permita eliminar un generador. Esta operación debemos realizarla directamente en la tabla del sistema que contiene las definiciones y valores de todos los generadores:

```
delete from rdb$generators
where rdb$generator_name = 'CODIGOEMPLEADO'
```

Para utilizar un generador necesitamos la función *gen_i*d. Esta función utiliza dos parámetros. El primero es el nombre del generador, y el segundo debe ser la cantidad en la que se incrementa o decrementa la memoria del generador. La función retorna entonces el valor ya actualizado. Utilizaremos el generador anterior para suministrar automáticamente un código de empleado si la instrucción **insert** no lo hace:

```
create trigger NuevoEmpleado for Empleados
active before insert
as
begin
      if (new.Codigo is null) then
      new.Codigo = gen_id(CodigoEmpleado, 1);
end ^
```

Al preguntar primeramente si el código del nuevo empleado es nulo, estamos permitiendo la posibilidad de asignar manualmente un código de empleado durante la inserción.

Los programas escritos en C++ Builder tienen problemas cuando se asigna la clave primaria de una fila dentro de un *trigger* utilizando un generador, pues el registro recién insertado "desaparece" según el punto de vista de la tabla. Este problema se presenta sólo cuando estamos navegando simultáneamente sobre la tabla.

Para no tener que abandonar los generadores, una de las soluciones consiste en crear un procedimiento almacenado que obtenga el próximo valor del generador, y utilizar este valor para asignarlo a la clave primaria en el evento *BeforePost* de la tabla. En el lado del servidor se programaría algo parecido a lo siguiente:

```
create procedure ProximoCodigo returns (Cod integer) as
begin
Cod = gen_id(CodigoEmpleado);
end ^
```

En la aplicación crearíamos un componente *spProximoCodigo*, de la clase *TStoredProc*, y lo aprovecharíamos de esta forma en uno de los eventos *BeforePost* o *OnNewRecord* de la tabla de clientes:

```
void __fastcall TmodDatos::tbClientesBeforePost(TDataSet *DataSet)
{
spProximoCodigo->ExecProc();
tbClientesCodigo->Value =
       spProximoCodigo->ParamByName("COD")->AsInteger;
}
```

De todos modos, si la tabla cumple determinados requisitos, podemos ahorrarnos trabajo en la aplicación y seguir asignando la clave primaria en el *trigger*. Las condiciones necesarias son las siguientes:

· La tabla no debe tener columnas con valores **default**. Así evitamos que el BDE tenga que releer la fila después de su creación.

· Debe existir un índice único, o casi único, sobre alguna columna alternativa a la clave primaria. La columna de este índice se utilizará entonces como criterio de ordenación para la navegación.

· El valor de la clave primaria no nos importa realmente, como sucede con las claves artificiales. Las tablas de referencia que abundan en toda base de datos son un buen ejemplo de la clase de tablas anterior. Por ejemplo, si necesitamos una tabla para los diversos valores del estado civil, probablemente la definamos de este modo:

```
create table EstadoCivil (
Codigo integer not null primary key,
Descripcion varchar(15) not null unique,
EsperanzaVida integer not null
);

create generator EstadoCivilGen;

set term ^;
create trigger BIEstadoCivil for EstadoCivil
       active before insert as
begin
Codigo = gen_id(EstadoCivilGen, 1);
end ^
```

En C++ Builder asociaremos una tabla o consulta a la tabla anterior, pero ordenaremos las filas por su descripción, y ocultaremos el campo *Codigo*, que será asignado automáticamente en el servidor. Recuerde, en cualquier caso, que los problemas con la asignación de claves primarias en el servidor son realmente problemas de la navegación con el BDE, y nada tienen que ver con InterBase, en sí.

NOTA IMPORTANTE

En cualquier caso, si necesita valores únicos *y consecutivos* en alguna columna de una tabla, no utilice generadores (ni secuencias de Oracle, o identidades de MS SQL Server). El motivo es que los generadores no se bloquean durante las transacciones. Usted pide un valor dentro de una transacción, y le es concedido; todavía no ha terminado su transacción. A continuación, otro usuario pide el siguiente valor, y sus deseos se cumplen. Pero entonces usted aborta la transacción, por el motivo que sea. La consecuencia: se pierde el valor que recibió, y se produce un "hueco" en la secuencia.

1.5.5. Excepciones

Sin embargo, todavía no contamos con medios para detener una operación SQL; esta operación sería necesaria para simular imperativamente las restricciones a la propagación de cambios en cascada, en la integridad referencial. Lo que nos falta es poder lanzar excepciones desde un *trigger* o procedimiento almacenado. Las excepciones de InterBase se crean asociando una cadena de mensaje a un identificador:

```
create exception CLIENTE_CON_PEDIDOS
"No se puede modificar este cliente"
```

Es necesario confirmar la transacción actual para poder utilizar una excepción recién creada. Existen también instrucciones para modificar el mensaje asociado a una excepción **(alter exceptio**n), y para eliminar la definición de una excepción de la base de datos **(drop exceptio**n).

Una excepción se lanza desde un procedimiento almacenado o *trigger* mediante la instrucción **exception**:

```
create trigger CheckDetails for Clientes
active before delete
position 0
as
declare variable Numero int;
begin
select count(*)
from Pedidos
where RefCliente = old.Codigo
into :Numero;
if (:Numero > 0) then
exception CLIENTE_CON_PEDIDOS;
end ^
```

Las excepciones de InterBase determinan que cualquier cambio realizado dentro del cuerpo del *trigger* o procedimiento almacenado, sea directa o indirectamente, se anule automáticamente. De esta forma puede programarse algo parecido a las transacciones anidadas de otros sistemas de bases de datos.

Si la instrucción **exception** es similar a la instrucción **throw** de C++, el equivalente más cercano a **try...catch** es la instrucción **when** de InterBase. Esta instrucción tiene tres formas diferentes. La primera intercepta las excepciones lanzadas con **exception**:

```
when exception NombreExcepción do
BloqueInstrucciones;
```

Con la segunda variante, se detectan los errores producidos por las instrucciones SQL:

```
when sqlcode Numero do
BloqueInstrucciones;
```

Los números de error de SQL aparecen documentados en la ayuda en línea y en el manual *Language Reference* de InterBase. A grandes rasgos, la ejecución correcta de una instrucción devuelve un código igual a 0, cualquier valor negativo es un error propiamente dicho (-803, por ejemplo, es un intento de violación de una clave primaria), y los valores positivos son advertencias. En particular, 100 es el valor que se devuelve cuando una selección única no encuentra el registro buscado. Este

convenio es parte del estándar de SQL, aunque los códigos de error concreto varíen de un sistema a otro.

```
La tercera forma de la instrucción when es la siguiente:
when gdscode Numero do
BloqueInstrucciones;
```

En este caso, se están interceptando los mismos errores que con **sqlcod**e, pero se utilizan los códigos internos de InterBase, que ofrecen más detalles sobre la causa. Por ejemplo, los valores 335544349 y 35544665 corresponden a –803, la violación de unicidad, pero el primero se produce cuando se inserta un valor duplicado en cualquier índice único, mientras que el segundo se reserva para las violaciones específicas de clave primaria o alternativa.

En cualquier caso, las instrucciones **when** deben ser las últimas del bloque en que se incluyen, y pueden colocarse varias simultáneamente, para atender varios casos:

```
begin
/* Instrucciones */
/* … */
when sqlcode –803 do
Resultado = "Violación de unicidad";
when exception CLIENTE_CON_PEDIDOS do
Resultado = "Elimine primero los pedidos realizados";
end
```

No debe detenerse la propagación de una excepción, a no ser que se tenga una solución a su causa.

1.5.6. Alertadores de eventos

Los alertadores de eventos *(event alerter*s) son un recurso único, por el momento, de InterBase. Los procedimientos almacenados y *triggers* de InterBase pueden utilizar la instrucción siguiente:

```
post_event NombreDeEvento
```

El nombre de evento puede ser una constante de cadena o una variable del mismo tipo. Cuando se produce un evento, InterBase avisa a todos los clientes interesados de la ocurrencia del mismo.

Los alertadores de eventos son un recurso muy potente. En un entorno cliente / servidor donde se producen con frecuencia cambios en una base de datos, las estaciones de trabajo normalmente no reciben aviso de estos cambios, y los usuarios deben actualizar periódica y frecuentemente sus pantallas para reflejar los cambios realizados por otros usuarios, pues en caso contrario puede suceder que alguien tome una decisión equivocada en base a lo que está viendo en pantalla. Sin embargo, refrescar la pantalla toma tiempo, pues hay que traer cierta cantidad de información desde el servidor de bases de datos, y las estaciones de trabajo realizan esta operación periódicamente, colapsando la red. La solución pasa por crear *triggers* al estilo del siguiente:

```
create trigger AlertarCambioBolsa for Cotizaciones
active after update position 10
as
begin
post_event "CambioCotizacion";
end ^
```

Obsérvese que se ha definido una prioridad baja para el orden de disparo del *trigge*r.

Hay que aplicar la misma técnica para cada una de las operaciones de actualización de la tabla de cotizaciones. Luego, en el módulo de datos de la aplicación que se ejecuta en las estaciones de trabajo, hay que añadir el componente *TIBEventAlerter*, que se encuentra en la página *Samples* de la Paleta de Componentes. Este componente tiene las siguientes propiedades, métodos y eventos:

Nombre	**Tipo**	**Propósito**
Events	Propiedad	Los nombres de eventos que nos interesan.
Registered	Propiedad	Debe ser *True* para notificar, en tiempo de diseño, nuestro interés en los eventos almacenados en la propiedad anterior.
Database	Propiedad	La base de datos a la cual nos conectaremos.
RegisterEvents	Método	Notifica a la base de datos nuestro interés por los eventos de la propiedad *Event*s.
UnRegisterEvents	Método	El inverso del método anterior.
OnEventAlert	Evento	Se dispara cada vez que se produce el evento**.**

En nuestro caso, podemos editar la propiedad *Events* y teclear la cadena *CambioCotizacion*, que es el nombre del evento que necesitamos. Conectamos la propiedad *Database* del componente a nuestro componente de bases de datos y activamos la propiedad *Register*ed. Luego creamos un manejador para el evento *OnEventAlert* similar a éste:

```
void __fastcall TForm1::IBEventAlerter1EventAlert(TObject *Sender,
AnsiString EventName, long EventCount, bool &CancelAlerts)
{
tbCotizaciones->Refresh();
}
```

Cada vez que se modifique el contenido de la tabla *Cotizacione*s, el servidor de InterBase lanzará el evento identificado por la cadena *CambioCotizacio*n, y este evento será recibido por todas las aplicaciones interesadas. Cada aplicación realizará consecuentemente la actualización visual de la tabla en cuestión.

1.5.7. Funciones de usuario en InterBase

Aquí se muestra un ejemplo de cómo utilizar las DLL para extender la funcionalidad de un servidor de InterBase. Como forma de ampliar el conjunto de funciones disponibles en SQL, los servidores de InterBase basados en Windows 95 y Windows NT admiten la creación de *funciones definidas por el usuario (User Defined Function*s, ó *UDF*). Estas funciones se definen en DLLs que se deben registrar en el servidor, para poder ser ejecutadas desde consultas SQL, *triggers* y procedimientos almacenados.

Los pasos para crear una función de usuario son los siguientes:

· Programar la DLL, exportando las funciones deseadas.

· Copiar la DLL resultante al directorio *bin* del servidor de InterBase. Si se trata de un servidor local, o si tenemos acceso al disco duro del servidor remoto, esto puede realizarse cambiando el directorio de salida en las opciones del proyecto.

· Utilice la instrucción **declare external function** de InterBase para registrar la función en la base de datos correspondiente. Para facilitar el uso de la extensión programada, puede acompañar a la DLL con las declaraciones correspondientes almacenadas en un *script* SQL.

Para ilustrar la técnica, crearemos una función que devuelva el nombre del día de la semana de una fecha determinada. La declaración de la función, en la sintaxis de InterBase, será la siguiente:

```
declare external function DiaSemana(DATE)
returns cstring(15)
entry_point "DiaSemana"
module_name "MisUdfs.dll";
```

Aunque podemos comenzar declarando la función, pues InterBase cargará la DLL sólo cuando sea necesario, es preferible comenzar creando la DLL, así que cree un nuevo proyecto DLL, con el nombre MisUdfs.

Las funciones de usuario de InterBase deben implementarse con el atributo **__cdec**l. Hay que tener en cuenta que todos los parámetros se pasan por referencia; incluso los valores de retorno de las funciones se pasan por referencia (se devuelve un puntero), si no se especifica la opción **by value** en la declaración de la función. La correspondencia entre tipos de datos de InterBase y de C++ Builder es sencilla: **int** equivale a **in**t, **smallint** a **short in**t, las cadenas de caracteres se pasan como punteros a caracteres, y así sucesivamente. En particular, las fechas se pasan en un tipo de registro con la siguiente declaración:

```
typedef struct {
int Days;
int Frac;
} TIBDate;
```

Days es la cantidad de días transcurridos a partir de una fecha determinada por InterBase, el 17 de noviembre de 1858. Frac es la cantidad de diezmilésimas de segundos transcurridas desde las doce de la noche. Con esta información en nuestras manos, es fácil programar la función DiaSemana:

```
__declspec(dllexport) char const * __cdecl DiaSemana(TIBDate &fecha)
{
static char *dias[] = {
"Miércoles", "Jueves", "Viernes", "Sábado",
"Domingo", "Lunes", "Martes" };
return dias[fecha.Days % 7];
}
```

Para saber qué día de la semana corresponde al "día de la creación" de InterBase, tuvimos que realizar un proceso sencillo de prueba y error; parece que para alguien en este mundo los miércoles son importantes.

Una vez compilado el proyecto, debe asegurarse que la DLL generada está presente en el directorio bin del servidor de InterBase. Hay que activar entonces la utilidad WISQL, conectarse a una base de datos que contenga tablas con fechas, teclear la instrucción **declare external function** que hemos mostrado anteriormente y ejecutarla. A continuación, probar el resultado, con una consulta como la siguiente:

```
select DiaSemana(SaleDate), SaleDate,
cast("Now" as date), DiaSemana("Now")
from Orders
```

Tenga en cuenta que, una vez que el servidor cargue la DLL, ésta quedará en memoria hasta que el servidor se desconecte. De este modo, para sustituir la DLL (para añadir funciones o corregir errores) debe primero detener al servidor y volver a iniciarlo posteriormente.

Hay que tener cuidado, especialmente en InterBase 5, con las funciones que devuelven cadenas de caracteres generadas por la DLL. El problema es que estas funciones necesitan un *buffer* para devolver la cadena, que debe ser suministrado por la DLL.

No se puede utilizar una variable global con este propósito, como en versiones anteriores de InterBase, debido a la nueva arquitectura multihilos. Ahora todas las conexiones de clientes comparten un mismo proceso en el servidor, y si varias de ellas utilizan una misma UDF, están accediendo a la función desde distintos hilos. Si utilizáramos una variable global, podríamos sobrescribir su contenido con mucha facilidad. Por ejemplo, ésta es la implementación en C++ Builder de una función de usuario para convertir cadenas a minúsculas:

```
__declspec(dllexport) char * __cdecl Lower(char *s)
{
int len = strlen(s);
char *res = (char *) SysGetMem(len + 1);
// SysGetMem asigna memoria en el formato adecuado
strcpy(res, s);
CharLowerBuff(res, len);
return res;
}
```

Si queremos utilizar esta función desde InterBase, debemos declararla mediante la siguiente instrucción:

```
declare external function lower cstring(256)
returns cstring (256) free_it
entry_point "Lower" module_name "MisUdfs.dll"
```

Observe el uso de la nueva palabra reservada **free_it**, para indicar que la función reserva memoria que debe ser liberada por el servidor.

1.6. El procedimiento almacenado *ActVarInv*

Una de las acciones que deberá realizar nuestra aplicación consistirá en ir actualizando periódicamente las variables de los inversores en la tabla 1 mediante los valores que el servidor vaya adquiriendo a través del puerto serie. Para realizar esto es necesario ejecutar un procedimiento desde el programa, pasándole como parámetros dichar variables. Este procedimiento se ha codificado en el *script* ActVarInv.sql:

```
set term ^;

create procedure ActVarInv (num smallint, tenpan float, intpan float, tenred float,
intbob float)
as

begin
UPDATE tabla1
SET tension_de_panel = :tenpan, intensidad_de_panel = :intpan,
tension_de_red = :tenred, intensidad_de_bobina= :intbob, potencia=
(:intpan)*(:tenpan)
WHERE numero= :num;

end ^
set term ;^
```

1.7. El *trigger AddInvON*

Mediante este disparador conseguimos la interacción deseada entre las tablas 1 y 2: al manipular la columna *estado* de esta última tabla queremos que los inversores encendidos tengan de forma automática su correspondiente registro en la tabla 1.

```
set term ^;

create trigger AddInvON for tabla2
       active after update position 0 as
declare variable auxiliar integer;
begin

for select no_inv from tabla2 where estado="on" into :auxiliar do
insert into tabla1(numero)
values(:auxiliar);

end ^
set term ;^
```

2. LA APLICACIÓN

2.1. Una vista preliminar

Mediante esta aplicación satisfaceremos varios fines:

-Dar la bienvenida al usuario y facilitarle "*online*" una breve introducción a los fines del programa y sus usos.

En el caso de la aplicación servidora:

-Adquisición de los datos suministrados por el *data logger* a través de la comunicación con el puerto serie, mediante el protocolo adecuado.

-Introducción de dichos datos en la base de datos previamente comentada en el anterior apartado, así como su manipulación. Concretamente, desde la ventana principal tenemos varias opciones:

**Actualizar base de datos*: se añaden o eliminan registros de la tabla de variables según se elija en la tabla de estados.

**Imprimir*: reproduce en papel el informe generado a raíz de la tabla 1.

**Salvar*: crea una copia de las variables almacenada en ese momento en la tabla 1.

**Cargar histórico*: presenta datos anteriormente salvados con la opción anterior.

**Gráficas:* dibuja la evolución actual de los primeros cuatro inversores.

Por otra parte, los datos que sirven para la representación de estas gráficas son salvados en los oportunos ficheros *.m* para su posterior estudio y tratamiento matemático con *MATLAB*.

-Servir a las aplicaciones *clientes* de puente hacia la base de datos.

Se ha tratado de presentar toda la aplicación con el aspecto más cuidado posible. Para ello se han empleado texturas del programa *WordPerfect* y se ha dotado de "banda sonora" en aquellas partes en las que se estimó adecuado.

El objeto de este manual no es otro que el de presentar los cimientos sobre los que se basó el desarrollo del proyecto y permitir de esta forma su adecuación a futuras metas. De forma progresiva se explicarán aquí los pasos dados para la consecución de los objetivos alcanzados por el momento, así como las herramientas que el *Builder* nos proporcionó para ello. Gran parte de este manual y de la memoria descriptiva se basa en el libro *"The Dark Side of C++ Builder"* de *Ian Marteens*, sin el cual no hubiera sido posible la realización del proyecto, por lo que desde aquí quisiera agradecérselo profundamente, así como recomendárselo a todo aquel que algún día tuviera que continuar mi labor y a cualquier aficionado a la programación.

2.2. La presentación

Antes de acceder a las opciones que comentamos con anterioridad, se realiza la presentación del programa mediante una fotografía de la planta fotovoltáica en la que este proyecto está integrada, con música de fondo. Como dicha presentación sólo aparecerá una única vez en toda la ejecución del programa, será menester crearla de forma dinámica para proceder acto seguido a su destrucción, de manera que quede optimizado el empleo de la memoria RAM por parte de la aplicación. Es por ello que en el proyecto no aparece disponible inicialmente un objeto heredado de la clase *TForm* que corresponda a esta ventana, sino que su constructor es llamado explícitamente en el evento *OnCreate* de la ventana principal:

```
Formpresentacion = new TForm (this);
```

Del mismo modo, todos los componentes contenidos en dicho objeto (léase *TMediaPlayer* y *TImage*) han sido creados dinámicamente y en la misma ubicación.

```
Foto = new TImage (Formpresentacion);
MediaPlayer2 = new TMediaPlayer (Formpresentacion);
```

Se ha cuidado el hecho de que la fotografía se ajuste exactamente al tamaño del marco de la ficha (*Client*):

```
Formpresentacion->ClientHeight=Foto->Picture->Height;
Formpresentacion->ClientWidth=Foto->Picture->Width;
Foto->Parent=Formpresentacion;
Foto->AutoSize=true;
Foto->Visible=true;
```

y se han eliminado los iconos (y con ellos sus acciones) de maximizar y minimizar el tamaño de la ficha, además de la posibilidad de variarlo con el ratón:

```
TBorderIcons tempBI = Formpresentacion->BorderIcons;
tempBI >> biMaximize;
tempBI >> biMinimize;
Formpresentacion->BorderIcons = tempBI;
Formpresentacion->BorderStyle = bsSingle;
```

En cuanto al componente de audio lo hemos configurado de forma que su ejecución no dependa de la máquina que soporte la aplicación, sino que se efectúe de forma automática:

```
MediaPlayer2->DeviceType=dtAutoSelect;
MediaPlayer2->FileName="C:\\AINV\\Beautiful Way.asf";
MediaPlayer2->AutoEnable=true;
```

A continuación se presenta todo el código relativo a la presentación:

```
void __fastcall TFormppal::FormCreate(TObject *Sender)
{
//.......creación del hilo responsable de la comunicación serie
TForm * Formpresentacion;
TImage * Foto;
TMediaPlayer * MediaPlayer2;
Formpresentacion = new TForm (this);
Formpresentacion->Position = poScreenCenter;

Foto = new TImage (Formpresentacion);
Foto->Picture->LoadFromFile("C:/MyFiles/image008.jpg");

Formpresentacion->ClientHeight=Foto->Picture->Height;
Formpresentacion->ClientWidth=Foto->Picture->Width;
Foto->Parent=Formpresentacion;
Foto->AutoSize=true;
```

```
Foto->Visible=true;
TBorderIcons tempBI = Formpresentacion->BorderIcons;
tempBI >> biMaximize;
tempBI >> biMinimize;
Formpresentacion->BorderIcons = tempBI;
Formpresentacion->BorderStyle = bsSingle;

MediaPlayer2 = new TMediaPlayer (Formpresentacion);
MediaPlayer2->Parent=Formpresentacion;
MediaPlayer2->DeviceType=dtAutoSelect;
MediaPlayer2->FileName="C:\\AINV\\Beautiful Way.asf";
MediaPlayer2->AutoEnable=true;
MediaPlayer2->Visible=false;
MediaPlayer2->Open();
MediaPlayer2->Play();

Formpresentacion->ShowModal();

//.......visualización de la ventana principal
}
```

2.3. La Base de Datos y la Aplicación

2.3.1. Conjuntos de datos: tablas

Un conjunto de datos, para C++ Builder, es cualquier fuente de información estructurada en filas y columnas. Este concepto abarca tanto a las tablas "reales" y las consultas SQL como a ciertos tipos de procedimientos almacenados. Pero también son conjuntos de datos los *conjuntos de datos cliente*s, que obtienen su contenido por medio de automatización OLE remota, o a partir de un fichero "plano" local, y que son una de las piezas claves de Midas. Y también lo son las tablas anidadas de Oracle 8, y los conjuntos de datos a la medida que desarrollan otras empresas para acceder a formatos de bases de datos no reconocidos por el Motor de Datos de Borland. Todos estos objetos tienen muchas propiedades, métodos y eventos en común.

En este apartado estudiaremos los conjuntos de datos en general, pero haremos énfasis en las propiedades específicas de las tablas.

· La jerarquía de los conjuntos de datos

La clase *TDataSet* representa una mayor abstracción del concepto de conjunto de datos, sin importar en absoluto su implementación física. Esta clase define características y comportamientos comunes que son heredados por clases especializadas.

Estas características son, entre otras:

· *Métodos de navegació*n: Un conjunto de datos es una colección de registros homogéneos. De estos registros, siempre hay un *registro activ*o, que es el único con el que podemos trabajar directamente. Los métodos de navegación permiten gestionar la posición de este registro activo.

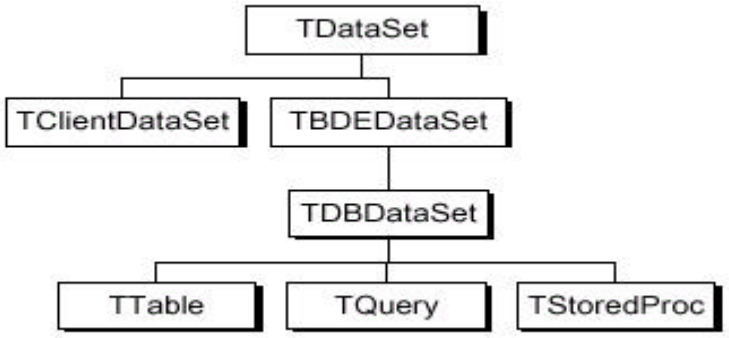
· *Acceso a campo*s: Todos los registros de un conjunto de datos están estructurados a su vez en campos. Existen mecanismos para descomponer la información almacenada en el registro activo de acuerdo a los campos que forman la estructura del conjunto de datos.

· *Estados del conjunto de datos*: Los conjuntos de datos implementan una propiedad *State*, que los transforman en simples autómatas finitos. Las transiciones entre estados se utilizan para permitir las altas y modificaciones dentro de los conjuntos de datos.

· Notificaciones a componentes visuales: Uno de los subsistemas más importantes asociados a los conjuntos de datos envía avisos a todos los componentes que se conectan a los mismos, cada vez que cambia la posición de la fila activa, o cuando se realizan modificaciones en ésta. Gracias a esta técnica es posible asociar controles de edición y visualización directamente a las tablas y consultas.

· Control de errores: Cuando detectan un error, los conjuntos de datos disparan eventos que permiten corregir y reintentar la operación, personalizar el mensaje de error o tomar otras medidas apropiadas.

En la versión 3 de la *VCL* la jerarquía de clases era la siguiente:



De esta forma, TDataSet pasó a ser totalmente independiente del BDE. La definición de esta clase reside ahora en la unidad DB, mientras que las clases que dependen del BDE (derivadas de TBDEDataSet) se han movido a la unidad DBTables.

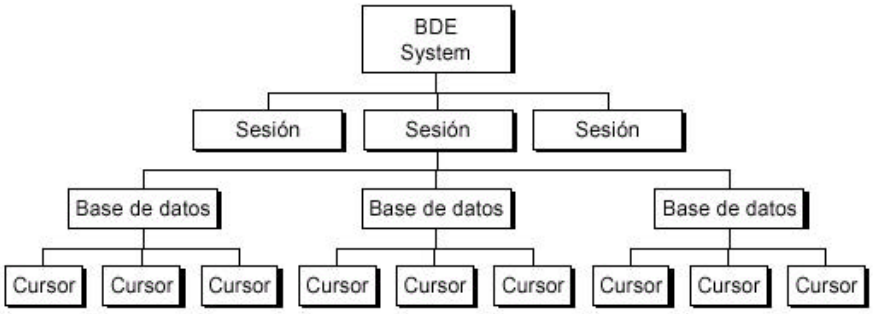
Si hubiéramos desarrollado un programa que no contuviera referencias a la unidad DBTables (ni a la unidad BDE de bajo nivel, por supuesto) no necesitaríamos incluir al Motor de Datos en la posterior instalación de la aplicación. Esto es particularmente cierto para las aplicaciones clientes de Midas, que se basan en la clase TClientDataSet. Como esta clase desciende directamente de TDataSet, no necesita la presencia del BDE para su funcionamiento.

La interacción de los subsistemas de los conjuntos de datos con los demás componentes de la VCL es bastante compleja, y existen muchas dependencias circulares. Para explicar el funcionamiento de los campos, necesitamos saber cómo funcionan las tablas, y viceversa.

En vez de comenzar la explicación con la clase abstracta *TDataSet*, lo cual haría imposible mostrar ejemplos, utilizaremos la clase *TTable*, que es además el componente de acceso a datos utilizado con mayor frecuencia.

· La arquitectura de objetos del Motor de Datos

Cuando utilizamos el BDE para acceder a bases de datos, nuestras peticiones pasan por toda una jerarquía de objetos. El siguiente esquema muestra los tipos de objetos con los que trabaja el BDE, y la relación que existe entre ellos:



El nivel superior se ocupa de la configuración global, inicialización y finalización del *sistem*a: el conjunto de instancias del BDE que se ejecutan en una misma máquina.

Las *sesiones* representan las diferentes aplicaciones y usuarios que acceden concurrentemente al sistema; en una aplicación de 32 bits pueden existir varias sesiones por aplicación, especialmente si la aplicación soporta concurrencia mediante hilos múltiples como es nuestro caso.

Cada sesión puede trabajar con varias bases de datos. Estas bases de datos pueden estar en distintos formatos físicos y en diferentes ubicaciones en una red. Su función es controlar la conexión a bases de datos protegidas por contraseñas, la gestión de transacciones y, en general, las operaciones que afectan a varias tablas simultáneamente.

Por último, una vez que hemos accedido a una base de datos, estamos preparados para trabajar con los cursores. Un *cursor* es una colección de registros, de los cuales tenemos acceso a uno solo a la vez, por lo que puede representarse mediante los conjuntos de datos de la VCL. Existen funciones y procedimientos para cambiar la posición del registro activo del cursor, y obtener y modificar los valores asociados a este registro. El concepto de cursor nos permite trabajar con tablas, consultas SQL y con el resultado de ciertos procedimientos almacenados de manera uniforme, ignorando las diferencias entre estas técnicas de obtención de datos.

Cada uno de los tipos de objetos internos del BDE descritos en el párrafo anterior tiene un equivalente directo en la VCL de C++ Guilder. La excepción es el nivel principal, el de sistema, algunas de cuyas funciones son asumidas por la clase de sesiones:

Objeto del BDE	**Clase de la VCL**
Sesiones	*TSession*
Bases de datos	*TDatabase*
Cursores	*TBDEDataSet*

Un programa escrito en C++ Builder no necesita utilizar explícitamente los objetos superiores en la jerarquía a los cursores para acceder a bases de datos. Los componentes de sesión y las bases de datos, por ejemplo, pueden ser creados internamente por la VCL, aunque el programador puede acceder a los mismos en tiempo de ejecución. Es por esto que podemos postergar el estudio de casi todos estos objetos.

· ¿Tabla o consulta?

Bien, nos disponemos a comenzar a desarrollar nuestra gran aplicación de bases de datos. ¿Qué componente debemos utilizar para acceder a los datos: tablas o consultas? La respuesta depende de qué operaciones vamos a permitir sobre los datos, del formato y tamaño de los mismos, y de la cantidad de tiempo que se quiera invertir en el proyecto. Cuando lleguemos al capítulo final de esta parte tendremos elementos suficientes para tomar una decisión, pero podemos adelantar algo ahora.

Si los datos estuvieran representados en una base de datos de escritorio, lo más indicado es utilizar las tablas del BDE, mediante el componente *TTable* de la VCL. Este componente accede de forma más o menos directa al fichero donde se almacenan los datos. Los registros se leen por demanda, solamente cuando son necesarios. Si tuviéramos una tabla de 100.000 clientes y quisiéramos buscar el último, el BDE realizará un par de operaciones aritméticas y le traerá precisamente el registro deseado. Para las bases de datos de escritorio, el uso de consultas *(TQuery*) es una forma indirecta de acceder a los datos. La ventaja de las consultas es que hay que programar menos, pero lo pagamos en velocidad.

Sin embargo, las consideraciones de eficiencia cambian diametralmente cuando se trata de sistemas cliente / servidor. El enemigo fundamental de estos sistemas es la limitada capacidad de transmisión de datos de las redes actuales. Montamos una red local con 100 megabits de tráfico por segundo (tecnología avanzada) esa será la máxima velocidad de transmisión, independientemente de si tiene conectados 10 o 100 ordenadores a la red. Se pueden utilizar trucos: servidores con varios puertos de red, por ejemplo, pero el cuello de botella seguirá localizado en la red. En tales condiciones, hay operaciones peligrosas para la eficiencia de la red, y la principal de ellas es la navegación indiscriminada.

En una aplicación cliente / servidor con grandes demandas de tráfico debe evitarse en la medida de lo posible operaciones de navegación libre sobre grandes conjuntos de datos. Este no es nuestro caso, ya que almacenaremos tan solo 6 valores para cada uno de los inversores (32 como máximo, pero 4 en el más común de los casos), por lo que es apto el uso de rejillas en nuestro programa, además de su conveniencia meramente estética.

· Tablas: el componente *TTable*

Mediante este componente podríamos conectarnos a tablas en cualquiera de los formatos reconocidos por el BDE. El componente *TTable* también permite conectarnos a una *vista* definida en una bases de datos SQL. Las vistas son tablas virtuales, definidas mediante una instrucción **select**, cuyos valores se extraen de otras tablas y vistas. La configuración de una tabla es muy sencilla. Primero hay que asignar el valor de la propiedad *DatabaseNam*e. En esta propiedad se indica el nombre del alias del BDE donde reside la tabla. Este alias puede ser un alias persistente, como los que creamos con la utilidad de configuración del BDE, o un alias local a la aplicación, creado con el componente *TDatabas*e.

-El concepto de alias:

Para "aplanar" las diferencias entre tantos formatos diferentes de bases de datos y métodos de acceso, BDE introduce los *alia*s. Un alias es, sencillamente, un nombre simbólico para referirnos a un base de datos. Cuando un programa que utiliza el BDE quiere, por ejemplo, abrir una tabla, sólo tiene que especificar un alias y la tabla que quiere abrir. Entonces el Motor de Datos examina su lista de alias y puede saber en qué formato se encuentra la base de datos, y cuál es su ubicación. Existen dos tipos diferentes de alias: los alias *persistentes* y los alias *locale*s, o de *sesión*.

Los alias persistentes se crean por lo general con el Administrador del BDE, y pueden utilizarse por cualquier aplicación que se ejecute en la misma máquina. Los alias locales son creados mediante llamadas al BDE realizadas desde una aplicación, y son visibles solamente dentro de la misma. La VCL facilita esta tarea mediante componentes de alto nivel, en concreto mediante la clase *TDatabase*.

Nosotros hemos especificado un alias persistente para nuestra base de datos mediante el *Alias Manager* de la utilidad *Database Desktop* incluida en el menú *Tools*.

Los alias ofrecen a la aplicación que los utiliza independencia con respecto al formato de los datos y su ubicación. Esto vale sobre todo para los alias persistentes, creados con el BDE. Puedo estar desarrollando una aplicación en casa, que trabaje con tablas del alias *dato*s. En mi ordenador, este alias está basado en el controlador de Paradox, y las tablas encontrarse en un determinado directorio de mi disco local. El destino final de la aplicación, en cambio, puede ser un ordenador en que el alias *datos* haya sido definido como una base de datos de Oracle, situada en tal servidor y con tal protocolo de conexión. A nivel de aplicación no necesitaremos cambio alguno para que se ejecute en las nuevas condiciones.

A partir de la versión 4.0 del BDE, se introducen los alias *virtuale*s, que corresponden a los nombres de fuentes de datos de ODBC, conocidos como DSN *(data source name*s). Una aplicación puede utilizar entonces directamente el nombre de un DSN como si fuera un alias nativo del BDE.

Una vez que tenemos asignado el nombre del alias o del directorio, podemos especificar el nombre de la tabla dentro de esa base de datos mediante la propiedad *Table-Nam*e.

Para poder extraer, modificar o insertar datos dentro de la tabla, necesitamos que la tabla esté abierta o activa. Esto se controla mediante la propiedad *Active* de la clase. También tenemos los métodos *Open* y *Clos*e, que realizan asignaciones a *Activ*e; el uso de estos métodos hace más legible nuestros programas.

Active es una propiedad que está disponible en tiempo de diseño. Esto quiere decir que si le asignamos el valor *True* durante el diseño, la tabla se abrirá

automáticamente al cargarse desde el fichero *df*m. También significa que mientras programamos la aplicación, podemos ver directamente los datos tal y como van a quedar en tiempo de ejecución; esta importante característica no está presente en ciertos sistemas RAD. No obstante, nunca está de más abrir explícitamente las tablas durante la inicialización del formulario o módulo de datos donde se ha definido. La propiedad *Active* puede, por accidente, quedarse en *False* por culpa de un error en tiempo de diseño, y de este modo garantizamos que en tiempo de ejecución las tablas estén abiertas. Por ello las dos últimas líneas del evento *OnCreate* de la ventana principal:

```
Table2->Active=true;
Table1->Active=true;
```

Además, aplicar el método *Open* sobre una tabla abierta no tiene efectos negativos, pues la llamada se ignora. En cuanto a cerrar la tabla, el destructor del componente llama automáticamente a *Clos*e, de modo que durante la destrucción del formulario o módulo donde se encuentra la tabla, ésta se cierra antes de ser destruida.

Una propiedad relacionada con el modo de apertura de una tabla es *ReadOnly*, que permite abrir la tabla en el modo sólo lectura.

· Conexión con componentes visuales

Un conjunto de datos, sea una tabla o una consulta, no puede visualizar directamente los datos con los que trabaja. Para comunicarse con los controles visuales, el conjunto de datos debe tener asociado un componente auxiliar, perteneciente a la clase TDataSource. Traduciremos esta palabra como fuente de datos, pero será utilizada lo menos posible, pues el parecido con "conjunto de datos" puede dar lugar a confusiones.

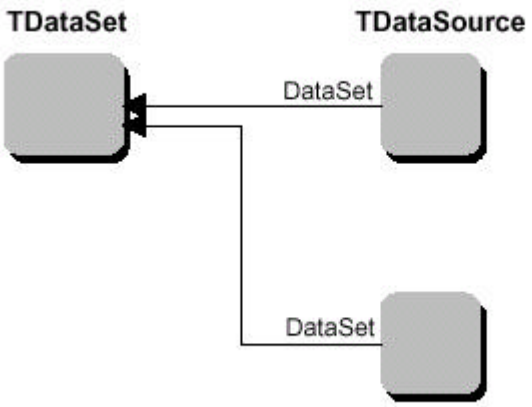
Un objeto TDataSource es, en esencia, un "notificador". Los objetos que se conectan a este componente son avisados de los cambios de estado y de contenido del conjunto de datos controlado por TDataSource. Las dos propiedades principales de TdataSource son:

· DataSet: Es un puntero, de tipo TDataSet, al conjunto de datos que se controla.

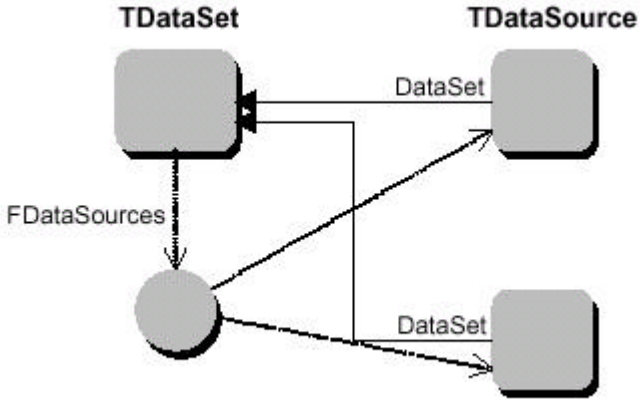
· AutoEdit: Cuando es True, el valor por omisión, permite editar directamente sobre los controles de datos asociados, sin tener que activar explícitamente el modo de edición en la tabla. El modo de edición se explica más adelante.

A la fuente de datos se conectan entonces todos los controles de datos que deseemos. Estos controles de datos se encuentran en la página Data Controls de la Paleta de Componentes, y todos tienen una propiedad DataSource para indicar a qué fuente de datos, y por lo tanto, a qué conjunto de datos indirectamente se conectan. Es posible acoplar más de una fuente de datos a un conjunto de datos. Estas fuentes de datos pueden incluso encontrarse en formularios o módulos de datos diferentes al del conjunto de datos. El propósito de esta técnica es establecer canales de notificación separados.

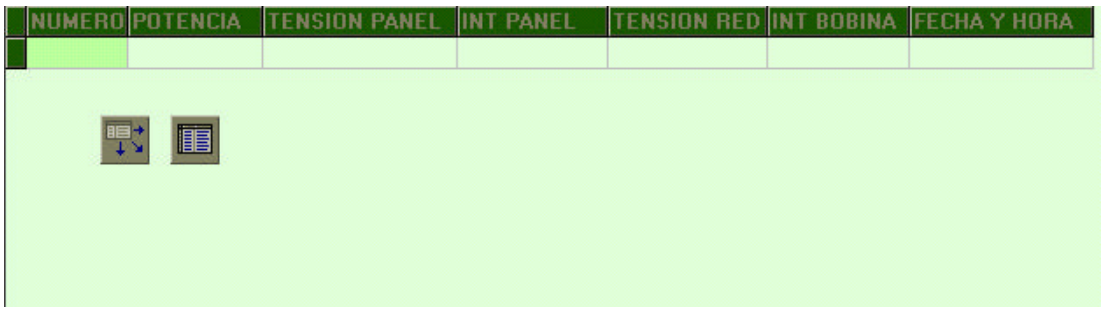
La técnica de utilizar varias fuentes de datos es posible gracias al mecanismo oculto que emplea C++ Builder. Cuando se engancha un data source a un conjunto de datos esto es lo que se ve:



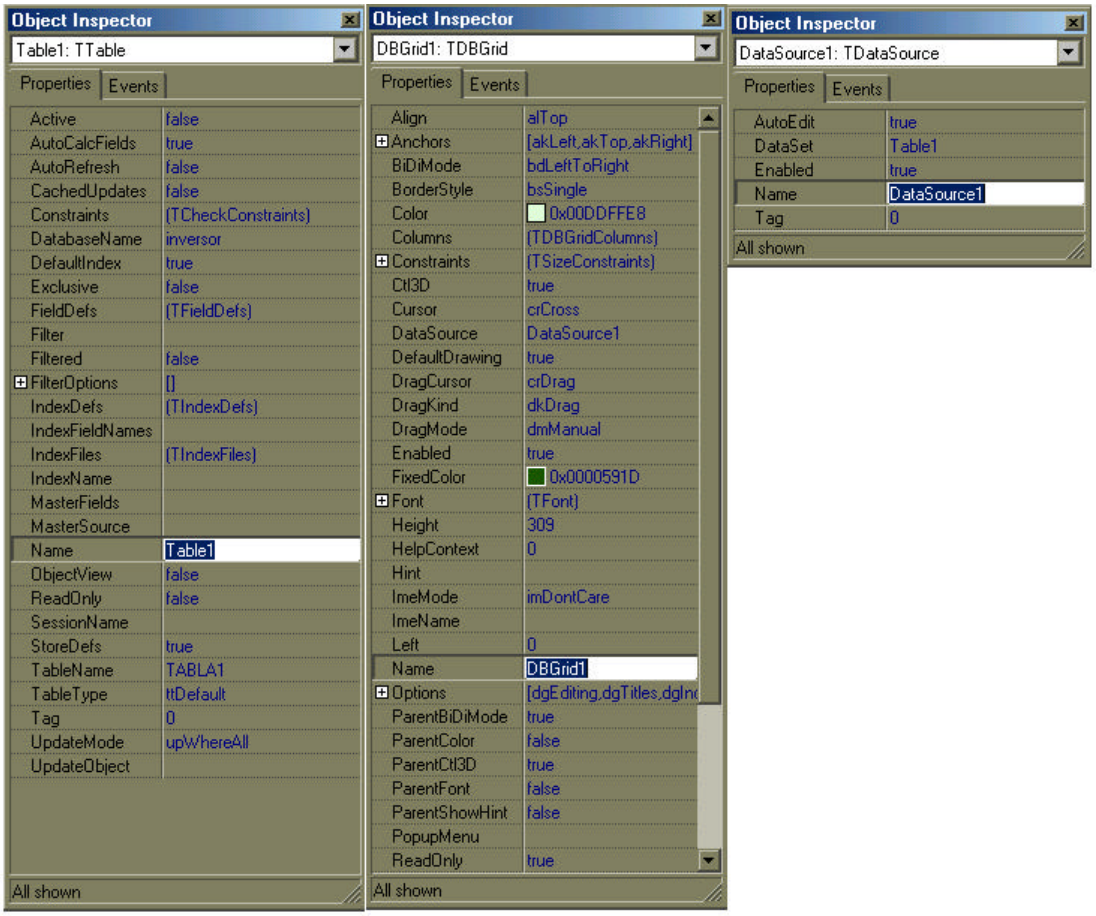
Sin embargo, existe un objeto oculto que pertenece al conjunto de datos, que es una lista de fuentes de datos y que completa el cuadro real:



Veamos la conexión de componentes utilizando una *rejilla de datos (TDBGrid*) para mostrar el contenido de una tabla o consulta utilizando como ejemplo la visualización de las variables de los inversores en la tabla 1:



A continuación se presentan mediante el Inspector de Objetos las propiedades de los distintos componentes utilizados.


Object Inspector
Table1: TTable
Properties
Events
Active false
AutoCalcFields true
AutoRefresh false
CachedUpdates false
Constraints (TCheckConstraints)
DatabaseName inversor
DefaultIndex true
Exclusive false
FieldDefs (TFieldDefs)
Filter
Filtered false
FilterOptions []
IndexDefs (TIndexDefs)
IndexFieldNames
IndexFiles (TIndexFiles)
IndexName
MasterFields
MasterSource
Name Table1
ObjectView false
ReadOnly false
SessionName
StoreDefs true
TableName TABLA1
TableType ttDefault
Tag 0
UpdateMode upWhereAll
UpdateObject
All shown
Object Inspector
DBGrid1: TDBGrid
Properties
Events
Align alTop
Anchors [akLeft,akTop,akRight]
BiDiMode bdLeftToRight
BorderStyle bsSingle
Color 0x00DDFFE8
Columns (TDBGridColumns)
Constraints (TSizeConstraints)
Ctl3D true
Cursor crCross
DataSource DataSource1
DefaultDrawing true
DragCursor crDrag
DragKind dkDrag
DragMode dmManual
Enabled true
FixedColor 0x0000591D
Font (TFont)
Height 309
HelpContext 0
Hint
ImeMode imDontCare
ImeName
Left 0
Name DBGrid1
Options [dgEditing,dgTitles,dgIn
ParentBiDiMode true
ParentColor false
ParentCtl3D true
ParentFont false
ParentShowHint false
PopupMenu
ReadOnly true
All shown
Object Inspector
DataSource1: TDataSource
Properties
Events
AutoEdit true
DataSet Table1
Enabled true
Name DataSource1
Tag 0
All shown

· Navegando por las filas

La mayor parte de las operaciones que se realizan sobre un conjunto de datos se aplican sobre la *fila activa* de éste, en particular aquellas operaciones que recuperan o modifican datos. Los valores correspondientes a las columnas de la fila activa se almacenan internamente en un *buffe*r, del cual los campos extraen sus valores.

Al abrir un conjunto de datos, inicialmente se activa su primera fila. Qué fila es ésta depende de si el conjunto está ordenado o no. Si se trata de una tabla con un índice activo, o una consulta SQL con una cláusula de ordenación **order b**y, el criterio de ordenación es, por supuesto, el indicado. Pero el orden de los registros no queda tan claro cuando abrimos una tabla sin índices activos. Para una tabla SQL el orden de las filas es aún más impredecible que para una tabla de Paradox o dBase. De hecho, el concepto de posición de registro no existe para las bases de datos SQL, debido a la forma en que generalmente se almacenan los registros. Este es el motivo por el cual las barras de desplazamientos vertical de las rejillas de datos de C++ Builder tienen sólo tres posiciones cuando se conectan a un conjunto de datos SQL: al principio de la tabla, al final o en el medio.

Los métodos de movimiento son los siguientes:

Método	Objetivo
First	Ir al primer registro
Prior	Ir al registro anterior
Next	Ir al registro siguiente
Last	Ir al último registro
MoveBy	Moverse la cantidad de filas indicada en el parámetro

Hay dos propiedades que nos avisan cuando hemos llegado a los extremos de la tabla:

Función	Significado
BOF	¿Estamos en el principio de la tabla?
EOF	¿Estamos al final de a tabla?

Combinando estas funciones con los métodos de posicionamiento, podemos crear los siguientes algoritmos de recorrido de tablas:

Hacia delante:

```
Table1->First();
while (! Table1->Eof)
{// Acción
Table1->Next();
}
```

Hacia atrás:

```
Table1->Last();
while (! Table1->Bof)
{// Acción
Table1->Prior();
}
```

Estas técnicas serán usadas en el evento *OnTimer* de la unidad principal para enviar caracteres al *data logger* según la instrucción en curso (que comentaremos en el momento oportuno):

```
if(inst==2){
Table2->First();
coninvON=0;
    while(!Table2->Eof){
     if(Table2ESTADO->AsString=="on"){
         TransmitCommChar(hComm,Table2NO_INV->AsVariant);
         invON[coninvON++]= Table2NO_INV->AsVariant;
         checks = checks+Table2NO_INV->AsVariant;
      }
    Table2->Next();
    }

  }

if(inst==3){
  Table2->First();
  conpot=0;
  while(!Table2->Eof){
    if(Table2ESTADO->AsString=="on"){
     TransmitCommChar(hComm,Table2LIMITE_DE_POTENCIA->AsVariant);
     checks=checks+Table2LIMITE_DE_POTENCIA->AsVariant;
     pot[conpot++] = Table2LIMITE_DE_POTENCIA->AsVariant;
    }
    Table2->Next();
   }
}
```

Si se tiene un bucle de navegación en cuyo interior no existe interacción alguna con el usuario u otro elemento (en nuestro caso el *data logger*), debe convertirse lo antes posible en un procedimiento almacenado que se ejecute en el servidor. Así se descarga de tareas al hilo correspondiente de la aplicación y el programa se ejecuta más rápido. Esto es lo que se ha preferido en el módulo *Surmain.cpp*, desde donde se ejecuta previo paso de los parámetros pertinentes el procedimiento almacenado *ActVarInv* que ya vimos:

```
StoredProc1->ParamByName("num")->AsSmallInt = trama[0];
  StoredProc1->ParamByName("tenpan")->AsFloat= trama[1];
  StoredProc1->ParamByName("intpan")->AsFloat= trama[2];
  StoredProc1->ParamByName("tenred")->AsFloat= trama[3];
  StoredProc1->ParamByName("intbob")->AsFloat= trama[4];
  StoredProc1->ExecProc();
```

- **El estado de un conjunto de datos**

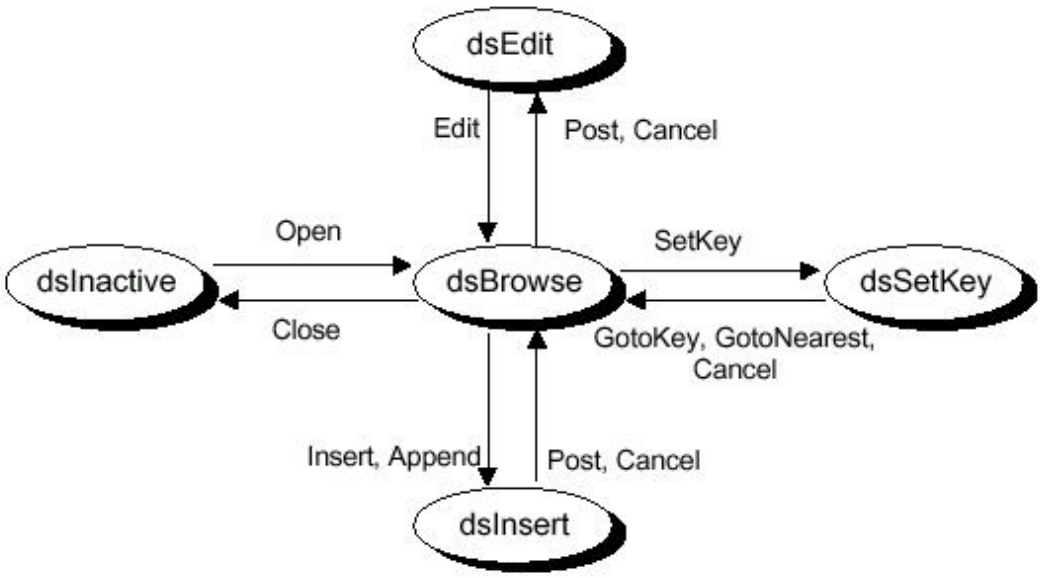
Una de las propiedades más importantes de los conjuntos de datos es *State*, cuya declaración es la siguiente:

```
enum TDataSetState = {dsInactive, dsBrowse, dsEdit, dsInsert,
dsSetKey, dsCalcFields, dsFilter, dsNewValue, dsOldValue,
dsCurValue, dsBlockRead, dsInternalCalc};
__property TDataSetState State;
```

El estado de un conjunto de datos determina qué operaciones se pueden realizar sobre el mismo. Por ejemplo, en el estado de exploración, *dsBrowse*, no se pueden realizar asignaciones a campos. Algunas operaciones cambian su semántica de acuerdo al estado en que se encuentre el conjunto de datos. El método *Post*, por ejemplo, graba una nueva fila si el estado es *dsInsert* en el momento de su aplicación, pero modifica la fila activa si el estado es *dsEdit*.

La propiedad *State* es una propiedad de sólo lectura, por lo que no podemos cambiar de estado simplemente asignando un valor a ésta. Incluso hay estados a los

cuales el programador no puede llegar explícitamente. Tal es el caso del estado *dsCalcFields*, al cual se pasa automáticamente cuando existen campos calculados en la tabla. Las transiciones de estado que puede realizar el programador se logran mediante llamadas a métodos. El siguiente diagrama muestra los diferentes estados de un conjunto de datos a los que puede pasar explícitamente el programador, y las transiciones entre los mismos:



En el diagrama no se han representado los estados internos e inaccesibles. Los estados *dsUpdateNew* y *dsUpdateOl*d, *dsOldValu*e, *dsNewValue* y *dsCurValue* son estados utilizados internamente por la VCL, y el programador nunca encontrará que una rutina programada por él se está ejecutando con una tabla en uno de estos estados.

En cambio, aunque el programador nunca coloca una tabla de forma explícita en los estados *dsCalcFields* y *dsFilte*r, aprovecha estos estados durante la respuestas a un par de eventos, *OnCalcFields* y *OnFilterRecor*d. El primero de estos eventos se utiliza para asignar valores a campos calculados; el segundo evento permite trabajar con un subconjunto de filas de una tabla.

La comprensión de los distintos estados de un conjunto de datos, los métodos y los eventos de transición son fundamentales para poder realizar actualizaciones en bases de datos.

2.3.2. Acceso a campos

Los componentes de acceso a campos son parte fundamental de la estructura de la VCL. Estos objetos permiten manipular los valores de los campos, definir formatos de visualización y edición, y realizar ciertas validaciones básicas. Sin ellos, nuestros programas tendrían que trabajar directamente con la imagen física del *buffer* del registro activo en un conjunto de datos. Afortunadamente, C++ Builder crea campos aún cuando a nosotros se nos olvida hacerlo. Vamos a estudiar las clases de campos y sus propiedades, concentrándonos en los tipos de campos "simples", y en el uso del Diccionario de Datos para acelerar la configuración de campos en tiempo de diseño.

· Creación de componentes de campos

Por mucho que busquemos, nunca encontraremos los componentes de acceso a campos en la Paleta de Componentes. El quid está en que estos componentes se vinculan al conjunto de datos (tabla o consulta) al cual pertenecen, del mismo modo en que los ítems de menú se vinculan al objeto de tipo *TMainMenu* ó *TPopupMenu* que los contiene. Siguiendo la analogía con los menús, para crear componentes de campos necesitamos realizar una doble pulsación sobre una tabla para invocar al *Editor de Campos* de C++ Builder. Este Editor se encuentra también disponible en el menú local de las tablas como el comando *Fields editor*.

Antes de explicar el proceso de creación de campos necesitamos aclarar una situación: podemos, como ya hemos hecho, colocar una tabla en un formulario, asociarle una fuente de datos y una rejilla, y echar a andar la aplicación resultante. ¿Para qué queremos campos entonces?

Aún cuando no se han definido componentes de acceso a campos explícitamente para una tabla, estos objetos están ahí, pues han sido creados automáticamente por C++ Builder. Si durante la apertura de una tabla se detecta que el usuario no ha definido campos en tiempo de diseño, la VCL crea objetos de acceso de forma implícita. Por supuesto, estos objetos reciben valores por omisión para sus propiedades, que quizás no sean los que deseamos. Precisamente por eso creamos componentes de campos en tiempo de diseño: para poder controlar las propiedades y eventos relacionados con los mismos. La creación en tiempo de diseño no nos hace malgastar memoria adicional en tiempo de ejecución, pues los componentes se van a crear de una forma u otra. Pero sí tenemos que contar con el aumento de tamaño del fichero *dfm*, que es donde se va a grabar la configuración persistente de los valores iniciales de las propiedades de los campos.

El Editor de Campos es una ventana de ejecución no modal; esto quiere decir que podemos tener a la vez en pantalla distintos conjuntos de campos, correspondiendo a distintas tablas, y que podemos pasar sin dificultad de un Editor a cualquier otra ventana, en particular, al Inspector de Objetos. Para realizar casi cualquier acción en el Editor de Campos hay que pulsar el botón derecho del ratón y seleccionar el comando de menú adecuado. Tenemos además una pequeña barra de navegación en la parte superior del Editor. Esta barra no está relacionada en absoluto con la edición de campos, sino que es un medio conveniente de mover, en tiempo de diseño, el cursor o fila activa de la tabla asociada.

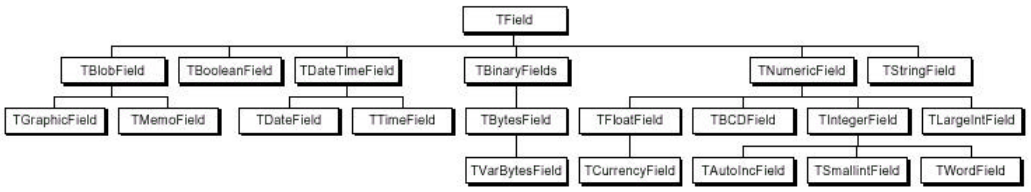
Añadir componentes de campos es muy fácil, pues basta con ejecutar el comando *Add fields* del menú local. Se presenta entonces un cuadro de diálogo con una lista de los campos físicos existentes en la tabla y que todavía no tienen componentes asociados. Esta lista es de selección múltiple, y selecciona por omisión todos los campos. Es aconsejable crear componentes para todos los campos, aún cuando no tengamos en mente utilizar algunos campos por el momento. La explicación tiene que ver también con el proceso mediante el cual la VCL crea los campos. Si al abrir la tabla se detecta la presencia de al menos un componente de campo definido en tiempo de diseño, C++ Builder no intenta crear objetos de campo automáticamente. El resultado es que estos campos que dejamos sin crear durante el diseño *no existen* en lo que concierne a C++ Builder.

Esta operación puede repetirse más adelante, si añadimos nuevos campos durante una reestructuración de la tabla, o si modificamos la definición de un campo. En este último caso, es necesario destruir primeramente el viejo componente antes de añadir el nuevo. Para destruir un componente de campo, sólo es necesario seleccionarlo en el Editor de Campos y pulsar la tecla SUPR. El comando *Add all field*s, del menú local del Editor de Campos, es una novedad de C++ Builder 4 para acelerar la configuración de campos y es muy recomendable por lo visto con anterioridad.

· Clases de campos

Una vez creados los componentes de campo, podemos seleccionarlos en el Inspector de Objetos a través de la lista de objetos, o mediante el propio Editor de Campos. Lo primero que llama la atención es que, a diferencia de los menús donde todos los comandos pertenecen a la misma clase, *TMenuIte*m, aquí cada componente de acceso a campo puede pertenecer a una clase distinta. En realidad, todos los componentes de acceso a campos pertenecen a una jerarquía de clases derivada por herencia de una clase común, la clase *TField*.

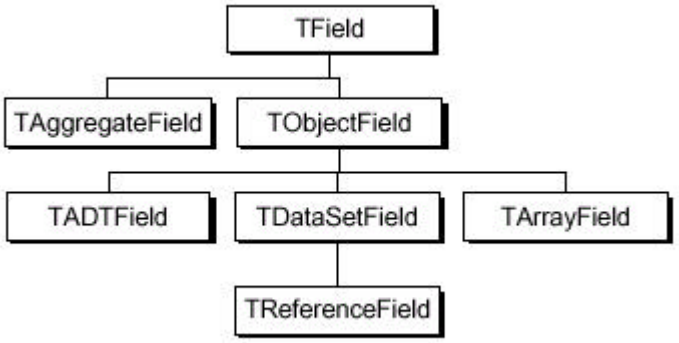
El siguiente diagrama muestra esta jerarquía:



De todos estos tipos, *TFiel*d, *TNumericField* y *TBinaryField* nunca se utilizan directamente para crear instancias de objetos de campos; su papel es servir de ancestro a campos de tipo especializado. La correspondencia entre los tipos de las variables de campos y los tipos de las columnas es la siguiente:

Tipo de campo	**InterBase**
TStringField	char, varchar
TIntegerField	int, long
TSmallintField	short
TFloatField	float, double
TDateTimeField	timestamp date
TBlobField	blob
TMemoField	text blob

A la jerarquía de clases que hemos mostrado antes, C++ Builder 4 añade un par de ramas:



La clase *TAggregateField* permite definir campos agregados con cálculo automático en conjuntos de datos clientes: sumas, medias, máximos, mínimos, etc. Tendremos que esperar un poco para examinarlos. vectores *(TArrayFiel*d) y campos de tablas anidadas *(TDataSetFiel*d).

Aunque InterBase permite definir campos que contienen matrices de valores, las versiones actuales de la VCL y del BDE no permiten tratarlos como tales directamente.

· Nombre del campo y etiqueta de visualización

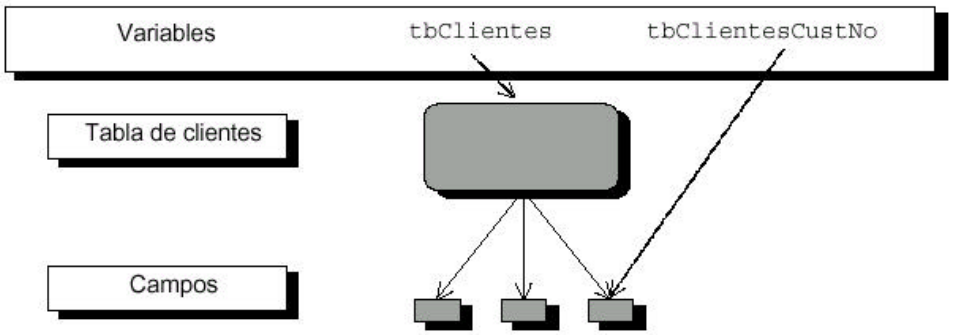
Existen tres propiedades de los campos que muchas veces son confundidas entre sí por el programador. Son las propiedades *Nam*e, *FieldName* y *DisplayLabel*. La primera es, como sucede con casi todos los componentes, el nombre de la variable de campo, o sea, del puntero al objeto. *FieldName* es el nombre de la columna de la tabla a la que se refiere el objeto de campo. Y *DisplayLabel* es un texto descriptivo del campo, que se utiliza, entre otras cosas, como encabezamiento de columna cuando el campo se muestra en una rejilla de datos.

De estas propiedades, *FieldName* es la que menos posibilidades nos deja: contiene el nombre de la columna, y punto. Por el contrario, *Name* se deduce inicialmente a partir del nombre de la tabla y del nombre de la columna. Si el nombre de tabla es *tabla1*y el nombre del campo *(FieldNam*e) es *INTENSIDAD_DE_BOBINA*, el nombre que C++ Builder le asigna a *Nam*e, y por consiguiente a la variable que apunta al campo, es *tabla1INTENSIDAD_DE_BOBINA*, la concatenación de ambos nombres.

Esto propicia un error bastante común entre los programadores, pues muchas veces escribimos por inercia, pensando en un esquema *tabl*a.*campo*:

tabla1->INTENSIDAD_DE_BOBINA // ¡¡¡INCORRECTO!!!

El siguiente gráfico puede ayudar a comprender mejor la relación entre los nombres de variables y los componentes de tablas y de campos:



La asignación automática de nombres de componentes de campos nos plantea un problema práctico: el tamaño del fichero *dfm* crece desmesuradamente. Tomemos por ejemplo una aplicación pequeña que trabajara con diez tablas, y supongamos que cada tuviera diez campos. Entonces, tendríamos cien componentes de campos, y cada componente tendría un nombre kilométrico que estaría ocupando espacio en el fichero *dfm* y luego en la memoria, en tiempo de ejecución. Es por eso que buscando un menor tiempo de carga de la aplicación, pues la memoria no es una consideración primordial en estos días, es una buena costumbre renombrar los componentes de campos con el propósito de disminuir la longitud de los nombres en lo posible, sin caer en ambigüedades. Por ejemplo, el nombre de componente *tbClientesCustNo* puede abreviarse a algo así como *tbClCustNo*; son sólo seis letras menos, pero multiplíquelas por cien y verá. En nuestro caso sólo tenemos 10 componentes de campos, por lo que esto no representa un problema.

· Acceso a los campos por medio de la tabla

Aunque la forma más directa, segura y eficiente de acceder a un campo es crear el componente en tiempo de diseño y hacer uso de la variable asociada, es también posible llegar indirectamente al campo a través de la tabla a la cual pertenece. Estas son las funciones y propiedades necesarias:

```
TField* __fastcall TDataSet::FieldByName(const AnsiString Nombre);
__property TFields* TDataSet::Fields;
```

La clase *TFields*, por su parte, tiene las siguientes propiedades principales:

```
__property int TFields::Count;
__property TField* TFields::Fields[int Index];
```

Con *FieldByName* podemos obtener el componente de campo dado su nombre, mientras que con *Fields* lo obtenemos si conocemos su posición. Está claro que esta última propiedad debe utilizarse con cautela, pues si la tabla se reestructura cambian las posiciones de las columnas. Mediante *FieldByName* y *Fields* obtenemos un objeto de tipo *TField*, la clase base de la jerarquía de campos. Por lo tanto, no se pueden utilizar directamente las propiedades específicas de los tipos de campos más

concretos sin realizar una conversión de tipo. A esto volveremos a referirnos. Mientras tanto, he aquí una muestra de cómo se accede a un campo dada su posición en C++ Builder 4:

```
Table1->Fields->Fields[0]->FieldName
```

Si a la función *FieldByName* le pasamos un nombre inexistente de campo, se produce una excepción, por lo cual no debemos utilizar esta función si lo que queremos es saber si el campo existe o no. Para esto último contamos con la función *FindField*, que devuelve el puntero al objeto si éste existe, o el puntero vacío si no:

```
TField* __fastcall TDataSet::FindField(const AnsiString Nombre);
```

Recuerde que el componente de campo puede haber sido creado explícitamente por usted en tiempo de diseño, pero que si no ha realizado esta acción, C++ Builder construye automáticamente estos objetos al abrir el conjunto de datos.

· Extrayendo información de los campos

Un componente de campo contiene los datos correspondientes al valor almacenado en la columna asociada de la fila activa de la tabla, y la operación más frecuente con un campo es extraer o modificar este valor. La forma más segura y eficiente es, una vez creados los campos persistentes con la ayuda del Editor de Campos, utilizar las variables generadas y la propiedad *Value* de las mismas. Esta propiedad se define del tipo apropiado para cada clase concreta de campo. Si el campo es de tipo *TStringFiel*d, su propiedad *Value* es de tipo *AnsiStrin*g; si el campo es de tipo *TBooleanFiel*d, el tipo de *Value* es **boo**l.

```
ShowMessage(Format("%d-%s", ARRAYOFCONST(
(tbClientesCodigo->Value, // Un valor entero
tbClientesNombre->Value)))); // Una cadena de caracteres
```

Si la referencia al campo es del tipo genérico *TField*, como las que se obtienen con la propiedad *Fields* y la función *FieldByNam*e, es necesario utilizar propiedades con nombres como *AsString*, *AsInteger*, *AsFloat*, etc., que aclaran el tipo de datos que queremos recuperar.

```
ShowMessage(
IntToStr(tbClientes->FieldByName("Codigo")->AsInteger)
+ "-" + tbClientes->FieldByName("Nombre")->AsString);
```

Las propiedades mencionadas intentan siempre hacer la conversión del valor almacenado realmente al tipo especificado; cuando no es posible, se produce una excepción. Por ejemplo, en el caso anterior hubiéramos podido utilizar también la propiedad *AsString* aplicada al campo entero *Codig*o. Ahora bien, existe un camino alternativo para manipular los datos de un campo: la propiedad *FieldValues* de *TDataSe*t. La declaración de esta propiedad es la siguiente:

```
__property Variant TDataSet::FieldValues[AnsiString FieldName];
```

Como la propiedad devuelve valores variantes, no es necesario preocuparse demasiado por el tipo del campo, pues la conversión transcurre automáticamente:

```
ShowMessage(tbClientes->FieldValues["Codigo"]
+ "-" + tbClientes->FieldValues["Nombre"]);
```

También puede utilizarse *FieldValues* con una lista de nombres de campos separados por puntos y comas. En este caso se devuelve una matriz variante formada por los valores de los campos individuales:

```
System::Variant V = tbClientes->FieldValues["Codigo;Nombre"];
ShowMessage(V.GetElement(0) + "-" + V.GetElement(1));
```

Debemos utilizar esta propiedad lo menos posible, pues tiene dos posibles puntos de fallo: puede que nos equivoquemos al teclear el nombre del campo (no se detecta el error sino en ejecución), y puede que nos equivoquemos en el tipo de retorno esperado.

Intencionalmente, todos los ejemplos que he mostrado leen valores desde las componentes de campos, pero no modifican este valor. El problema es que las asignaciones a campos sólo pueden efectuarse estando la tabla en alguno de los estados especiales de edición; en caso contrario, provocaremos una excepción.

Es útil saber también cuándo es nulo o no el valor almacenado en un campo. Para esto se utiliza la función *IsNull*, que retorna un valor de tipo **bool**. Por último, si el campo es de tipo memo, gráfico o BLOB, no existe una propiedad simple que nos proporcione acceso al contenido del mismo. Más adelante explicaremos cómo extraer información de los campos de estas clases.

· Creación de tablas

Aunque es posible utilizar propiedades y métodos del componente *TTable* para crear tablas, es preferible hacerlo mediante instrucciones SQL. La razón es que la VCL no ofrece mecanismos para la creación directa de restricciones de rango, de integridad referencial y valores por omisión para columnas; para esto, tenemos que utilizar llamadas directas al BDE, si tenemos que tratar con tablas Paradox, o utilizar SQL en las bases de datos que lo permiten.

2.3.3. El Diccionario de Datos

El Diccionario de Datos es una ayuda para el diseño que se administra desde la herramienta *Database Explorer*. El Diccionario nos ayuda a:

· Definir conjuntos de propiedades o *conjuntos de atributos (attribute set*s), que pueden después asociarse manual o automáticamente a los componentes de acceso a campos que se crean en C++ Builder. Por ejemplo, si nuestra aplicación trabaja con varios campos de porcentajes, puede sernos útil definir que los campos de este tipo se muestren con la propiedad *DisplayFormat* igual a *"%#0*", y que sus valores oscilen entre 0 y 100.

· Propagar a las aplicaciones clientes restricciones establecidas en el servidor, ya sea a nivel de campos o de tablas, valores por omisión, definiciones de dominios, etcétera.

· Organizar de forma centralizada los criterios de formato y validación entre varios programadores de un mismo equipo, y de una aplicación a otra.

C++ Builder crea al instalarse un Diccionario de Datos por omisión, que se almacena como una tabla Paradox, cuyo nombre por omisión es *bdesdd* y que reside en el alias predefinido *DefaultDD*. Debe recuerdarse que esta herramienta es una utilidad que funciona en tiempo de diseño, de modo que la tabla anterior no tiene (no *debe*, más bien) que estar presente junto a nuestro producto final.

Ahora bien, podemos trabajar con otro diccionario, que puede estar almacenado en cualquier otro formato de los reconocidos por el BDE. Este diccionario puede incluso almacenarse en un servidor SQL y ser compartido por todos los miembros de un equipo de programación. Mediante el comando de menú *Dictionary|New* puede crearse una nueva tabla para un nuevo diccionario, indicando el alias donde residirá, el nombre de la tabla y una descripción para el diccionario:

Después, mediante el comando de menú *Dictionary|Register* cualquier miembro del equipo puede registrar ese diccionario desde *otra* máquina, para utilizarlo con *su* C++ Builder. También es útil el comando *Dictionary|Select*, para activar alguno de los diccionarios registrados en determinado ordenador.

· Conjuntos de atributos

Son dos los objetos almacenados en un Diccionario de Datos: los conjuntos de atributos e información sobre bases de datos completas. Un conjunto de atributos indica valores para propiedades comunes de campos. Existen dos formas de crear esta definición: guardando los atributos asociados a un campo determinado, o introduciéndolos directamente en el Diccionario de Datos. Para salvar las modificaciones realizadas a un campo desde C++ Builder, hay que seleccionarlo en el Editor de Campos, pulsar el botón derecho del ratón y ejecutar el comando *Save attributes as*, que nos pedirá un nombre para el conjunto de atributos.

La otra vía es crear directamente el conjunto de atributos en el propio Diccionario. Digamos que nuestra aplicación debe trabajar con precios en dólares, mientras que nuestra moneda local es diferente. Nos situamos entonces sobre el nodo *Attribute sets* y ejecutamos *Object|New*. Renombramos el nuevo nodo como *Dollar*, y modificamos las siguientes propiedades, en el panel de la derecha:

Propiedad	**Valor**
Currency	*False*
DisplayFormat	*$#0,.00*

¿Qué se puede hacer con este conjunto de atributos, una vez creado? Asociarlo a campos, por supuesto. Nuestra hipotética aplicación maneja una tabla *Articulos* con un campo *PrecioDolares*, que ha sido definido con el tipo *money* de Paradox o de MS SQL Server. C++ Builder, por omisión, trae un campo de tipo *TCurrencyField*, cuya propiedad *Currency* aparece como *True*. El resultado: nuestros dólares se transforman mágicamente en pesetas (y pierden todo su valor). Pero seleccionamos el menú local del campo, y ejecutamos el comando *Associate attributes*, seleccionando el conjunto de atributos *Dollar* definido hace poco. Entonces C++ Builder lee los valores de las propiedades *Currency* y *DisplayFormat* desde el Diccionario de Datos y los copia en las propiedades del campo deseado. Y todo vuelve a la normalidad.

Dos de los atributos más importantes que se pueden definir en el Diccionario son *TFieldClass* y *TControlClass*. Mediante el primero podemos establecer explícitamente qué tipo de objeto de acceso queremos asociar a determinado campo; esto es útil sobre todo con campos de tipo BLOB. *TControlClass*, por su parte, determina qué tipo de control debe crearse cuando se arrastra y suelta un componente de campo sobre un formulario en tiempo de diseño.

Si un conjunto de atributos se modifica después de haber sido asociado a un campo, los cambios no se propagarán automáticamente a las propiedades de dicho campo. Habrá entonces que ejecutar el comando del menú local *Retrieve attribute*s.

· Importando bases de datos

Pero si nos limitamos a las técnicas descritas en la sección anterior, tendremos que configurar atributos campo por campo. Una alternativa consiste en *importar* el esquema de la base de datos dentro del Diccionario, mediante el comando de menú *Dictionary|Import from databas*e. Aparece un cuadro de diálogo para que seleccionemos uno de los alias disponibles; nos debemos armar de paciencia, porque la operación puede tardar un poco.

Muchas aplicaciones trabajan con un alias de sesión, definido mediante un componente *TDatabase* que se sitúa dentro de un módulo de datos, en vez de utilizar alias persistentes. Si ejecutamos Database Explorer como una aplicación independiente no podremos importar esa base de datos, al no estar disponible el alias en cuestión. La solución es invocar al Database Explorer desde el Entorno de Desarrollo, con la aplicación cargada y la base de datos conectada. Todos los alias de sesión activos en la aplicación podrán utilizarse entonces desde esta utilidad.

Cuando ha finalizado la importación, aparece un nuevo nodo para nuestra base de datos bajo el nodo *Database*s. El nuevo nodo contiene todas las tablas y los campos existentes en la base de datos, y a cada campo se le asocia automáticamente un conjunto de atributos. Por omisión, el Diccionario crea un conjunto de atributos por cada campo que tenga propiedades dignas de mención: una restricción (espere a la próxima sección), un valor por omisión, etc. Esto es demasiado, por supuesto. Después de la importación, debemos sacar factor común de los conjuntos de atributos similares y asociarlos adecuadamente a los campos.

La labor se facilita en InterBase si hemos creado dominios. Supongamos que definimos el dominio Dollar en la base de datos mediante la siguiente instrucción:

```
create domain Dollar as

numeric(15, 2) default 0 not null;
```

A partir de esta definición podremos definir columnas de tablas cuyo tipo de datos sea Dollar. Entonces el Diccionario de Datos, al importar la base de datos, creará automáticamente un conjunto de atributos denominado Dollar, y asociará correctamente los campos que pertenecen a ese conjunto.

¿Para qué todo este trabajo? Ahora, cuando traigamos una tabla a la aplicación y utilicemos el comando Add fields para crear objetos persistentes de campos, C++ Builder podrá asignar de forma automática los conjuntos de atributos a estos campos.

· Evaluando restricciones en el cliente

Los sistemas de bases de datos cliente / servidor, y algunas bases de datos de escritorio, permiten definir restricciones en las tablas que, normalmente, son verificadas por el propio servidor de datos. Por ejemplo, que el nombre del cliente no puede estar vacío, que la fecha de venta debe ser igual o anterior a la fecha de envío de un pedido, que un descuento debe ser mayor que cero, pero menor que cien...

Un día de verano, un usuario de nuestra aplicación (¡ah, los usuarios!) se enfrasca en rellenar un pedido, y vende un raro disco de Hendrix con un descuento del 125%. Al enviar los datos al servidor, éste detecta el error y notifica a la aplicación acerca del mismo. Un registro pasó a través de la red, un error nos fue devuelto; al parecer, poca cosa. Pero multiplique este tráfico por cincuenta puestos de venta, y lo poco se convierte en mucho. Además, ¿por qué hacer esperar tanto al usuario para preguntarle si es tonto, o si lo ha hecho a posta? Este tipo de validación sencilla puede ser ejecutada perfectamente por nuestra aplicación, pues solamente afecta a columnas del registro activo. Otra cosa muy diferente sería, claro está, intentar verificar por duplicado en el cliente una restricción de unicidad.

Hay cuatro propiedades, de tipo AnsiString, para los componentes de acceso a campos:

Propiedad	**Significado**
DefaultExpression	Valor por omisión del campo
CustomConstraint	Restricciones importadas desde el servidor
ImportedConstraint	Restricciones adicionales impuestas en el cliente
ConstraintErrorMessage	Mensaje de error cuando se violan restricciones

DefaultExpression es una expresión SQL que sirve para inicializar un campo durante las inserciones. No puede contener nombres de campos.

Hay un pequeño *bug* en la VCL: cuando se mezclan inicializaciones en el evento *OnNewRecord* con valores en las propiedades *DefaultExpressio*n, se producen comportamientos anómalos. Evite las mezclas, que no son buenas para la salud.

Podemos asignar directamente un valor constante en *DefaultExpression* ... pero si hemos asociado un conjunto de atributos al campo, C++ Builder puede leer automáticamente el valor por omisión asociado y asignarlo. Este conjunto de atributos puede haber sido configurado de forma manual, pero también puede haberse creado al importar la base de datos dentro del Diccionario. En este último caso, el Diccionario de Datos ha actuado como eslabón intermedio en la propagación de reglas de empresa desde el servidor al cliente:

Lo mismo se aplica a la propiedad *ImportedConstraint*. Esta propiedad recibe su valor desde el Diccionario de Datos, y debe contener una expresión SQL evaluable en SQL Local; léase, en el dialecto de Paradox y dBase. ¿Por qué permitir que esta propiedad pueda modificarse en tiempo de diseño? Precisamente porque la expresión importada puede ser incorrecta en el dialecto local. En ese caso, podemos eliminar toda o parte de la expresión. Normalmente, el Diccionario de Datos extrae las restricciones para los conjuntos de atributos de las cláusulas check de SQL definidas a nivel de columna.

Si, por el contrario, lo que se desea es añadir nuevas restricciones, debemos asignarlas en la propiedad *CustomConstraint*. Se puede suministrar cualquier expresión lógica de SQL Local, y para representar el valor actual del campo hay que utilizar un identificador cualquiera que no sea utilizado por SQL. Por ejemplo, esta puede ser una expresión aplicada sobre un campo de tipo cadena de caracteres:

```
x <> '' and x not like '% %'
```

Esta expresión verifica que el campo no esté vacío y que no contenga espacios en blanco. Cuando se viola cualquiera de las restricciones anteriores, *ImportedConstraint* ó *CustomConstraint*, por omisión se muestra una excepción con el mensaje "*Record or field constraint failed*", más la expresión que ha fallado en la segunda línea del mensaje. Si queremos mostrar un mensaje personalizado, debemos asignarlo a la propiedad *ConstraintErrorMessag*e.

Sin embargo, no todas las restricciones **check** se definen a nivel de columna, sino que algunas se crean a nivel de tabla, casi siempre cuando involucran a dos o más campos a la vez. Por ejemplo:

```
create table Pedidos ( /* Restricción a nivel de columna */

FormaPago varchar(10)

check (FormaPago in ('EFECTIVO', 'TARJETA')), /* Restricción a nivel de tabla */

FechaVenta date not null,

FechaEnvio date,

check (FechaVenta <= FechaEnvio), /* … */

);
```

Las restricciones a nivel de tabla se propagan a una propiedad de *TTable* denominada *Constraint*s, que contiene tanto las restricciones importadas como alguna restricción personalizada añadida por el programador.

2.3.4. Controles de datos y fuentes de datos

A continuación trataremos los controles que ofrece la VCL para visualizar y editar información procedente de bases de datos, la filosofía general en que se apoyan y las particularidades de cada uno de ellos. Es el momento de ampliar, además, nuestros conocimientos acerca de un componente esencial para la sincronización de estos controles, el componente *TDataSourc*e. Veremos cómo un control "normal" puede convertirse, gracias a una sencilla técnica basada en este componente, en un flamante control de acceso a datos. Por último, estudiaremos cómo manejar campos BLOB, que pueden contener grandes volúmenes de información, dejando su interpretación a nuestro cargo.

· Controles *data-aware*

Los controles de bases de datos son conocidos en la jerga de C++ Builder como controles *data-aware*. Estos controles son, generalmente, versiones especializadas de controles "normales". Por ejemplo, *TDBMemo* es una versión orientada a bases de datos del conocido *TMem*o. Al tener por separado los controles de bases de datos y los controles tradicionales, C++ Builder evita que una aplicación que no haga uso de bases de datos tenga que cargar con todo el código necesario para estas operaciones.

No sucede así con Visual Basic, lenguaje en que todos los controles pueden potencialmente conectarse a una base de datos.

Los controles de acceso a datos de C++ Builder se pueden dividir en dos grandes grupos:

· Controles asociados a campos

· Controles asociados a conjuntos de datos

Los controles asociados a campos visualizan y editan una columna particular de una tabla. Los componentes *TDBEdit* (cuadros de edición) y *TDBImage* (imágenes almacenadas en campos gráficos) pertenecen a este tipo de controles. Los controles asociados a conjuntos de datos, en cambio, trabajan con la tabla o consulta como un todo. Las rejillas de datos y las barras de navegación son los ejemplos más conocidos de este segundo grupo. Todos los controles de acceso a datos orientados a campos tienen un par de propiedades para indicar con qué datos trabajan: *DataSource* y *Data-Field*; estas propiedades pueden utilizarse en tiempo de diseño. Por el contrario, los controles orientados al conjunto de datos solamente disponen de la propiedad *Data-Source*. La conexión con la fuente de datos es fundamental, pues es este componente quien notifica al control de datos de cualquier alteración en la información que debe visualizar.

Casi todos los controles de datos permiten, de una forma u otra, la edición de su contenido. Hace falta que la tabla esté en uno de los modos *dsEdit* ó *dsInsert* para poder modificar el contenido de un campo. Pues bien, todos los controles de

datos son capaces de colocar a la tabla asociada en este modo, de forma automática, cuando se realizan modificaciones en su interior. Este comportamiento se puede modificar con la propiedad *AutoEdit* del *data source* al que se conectan. Cada control, además, dispone de una propiedad *ReadOnly*, que permite utilizar el control únicamente para visualización.

Actualmente, los controles de datos de C++ Builder son los siguientes:

Nombre de la clase	Explicación
Controles orientados a campos	
TDBText	Textos no modificables (no consumen recursos)
TDBEdit	Cuadros de edición
TDBMemo	Textos sin formato con múltiples líneas
TDBImage	Imágenes BMP y WMF
TDBListBox	Cuadros de lista (contenido fijo)
TDBComboBox	Cuadros de combinación (contenido fijo)
TDBCheckBox	Casillas de verificación (dos estados)
TDBRadioGroup	Grupos de botones (varios valores fijos)
TDBLookupListBox	Valores extraídos desde otra tabla asociada
TDBLookupComboBox	Valores extraídos desde otra tabla asociada
TDBRichEdit	Textos en formato RTF

Nombre de la clase	Explicación
Controles orientados a conjuntos de datos	
TDBGrid	Rejillas de datos para la exploración
TDBNavigator	Control de navegación y estado
TDBCtrlGrid	Rejilla que permite incluir controles

Nos ocuparemos principalmente de los controles orientados a campos. El resto de los controles serán estudiados posteriormente.

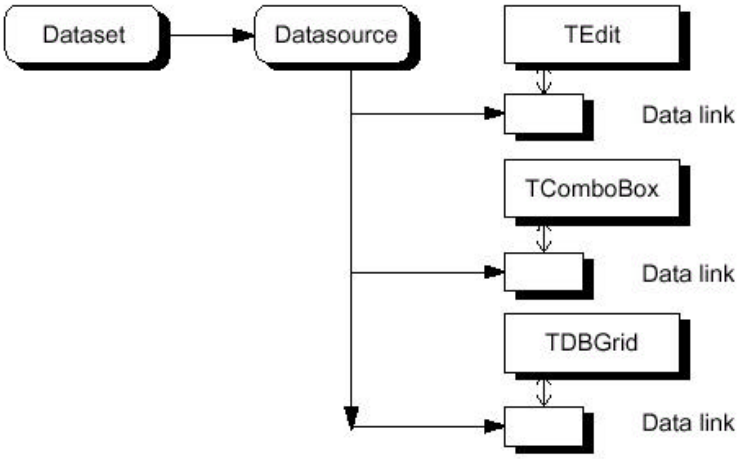
· Los enlaces de datos.

Según lo explicado, todos los controles de datos deben ser capaces de reconocer y participar en el juego de las notificaciones, y esto supone la existencia de un montón de código común a todos ellos. Pero, si observamos el diagrama de herencia de la VCL, notaremos que no existe un ancestro compartido propio para los controles *data-aware*. ¿Qué solución se ha utilizado en la VCL para evitar la duplicación de código?

La respuesta la tiene un objeto generalmente ignorado por el programador de C++ Builder: *TDataLink*, y su descendiente *TFieldDataLink*. Este desconocimiento es comprensible, pues no es un componente visual, y sólo es imprescindible para el desarrollador de componentes. Cada control de datos crea durante su inicialización un componente interno perteneciente a una de estas clases. Es este componente interno el que se conecta a la fuente de datos, y es también a éste a quien la fuente de datos envía las notificaciones acerca del movimiento y modificaciones que ocurren en el conjunto de datos subyacente. Todo el tratamiento de las notificaciones se produce entonces, al menos de forma inicial, en el *data link*. Esta técnica es conocida

como *delegación*, y nos evita el uso de la herencia múltiple, recurso no soportado por la VCL hasta el momento.

El siguiente esquema muestra la relación entre los componentes de acceso y edición de datos:



· Creación de controles de datos

Podemos crear controles de datos en un formulario trayendo uno a uno los componentes deseados desde la Paleta e inicializando sus propiedades *DataSource* y *Data-Field*. Esta es una tarea tediosa. Muchos programadores utilizaban en las primeras versiones de C++ Builder el Experto de Datos *(Database form exper*t) para crear un formulario con los controles deseados, y luego modificar este diseño inicial. Este experto, sin embargo, tiene un par de limitaciones importantes: en primer lugar, trabaja con un tamaño fijo de la ventana, lo cual nos obliga a realizar desplazamientos cuando no hay espacio para los controles, aún cuando aumentando el tamaño de la ventana se pudiera resolver el problema. El otro inconveniente es que siempre genera un nuevo componente *TTable* ó *TQuery*, no permitiendo utilizar componentes existentes de estos tipos. Esto es un problema, pues lo usual es definir primero los conjuntos de datos en un módulo aparte, para poder programar reglas de empresa de forma centralizada.

Podemos arrastrar campos desde el Editor de Campos sobre un formulario. Cuando lo hacemos, se crea automáticamente un control de datos asociado al campo. Junto con el control, se crea también una etiqueta, de clase *TLabe*l, con el título extraído de la propiedad *DisplayLabel* del campo. Recuerde que esta propiedad coincide inicialmente con el nombre del campo, de modo que si las columnas de sus tablas tienen nombre crípticos como *NomCl*i, es conveniente modificar primero las etiquetas de visualización en los campos antes de crear los controles. Adicionalmente, C++ Builder asigna a la propiedad *FocusControl* de los componentes *TLabel* creados el puntero al control asociado. De este modo, si la etiqueta tiene un carácter subrayado, podemos hacer uso de este carácter como abreviatura para la selección del control.

En cuanto al tipo de control creado, C++ Builder tiene sus reglas por omisión: casi

siempre se crea un *TDBEdit*. Si el campo es un campo de búsqueda, crea un componente *TDBLookupComboBo*x. Si es un memo o un campo gráfico, se crea un control *TDBMemo* o un *TDBImag*e. Si el campo es lógico, se crea un *TDBCheckBo*x. Estas reglas implícitas, sin embargo, pueden modificarse por medio del Diccionario de Datos. Si el campo que se arrastra tiene definido un conjunto de atributos, el control a crear se determina por el valor almacenado en la propiedad *TControlClass* del conjunto de atributos en el Diccionario de Datos.

· Los cuadros de edición

La forma más general de editar un campo simple es utilizar un control TDBEdit. Este componente puede utilizarse con campos de cadenas de caracteres, numéricos, de fecha y de cualquier tipo en general que permita conversiones desde y hacia cadenas de caracteres. Los cuadros de edición con conexión a bases de datos se derivan de la clase TCustomMaskEdit. Esto es así para poder utilizar la propiedad EditMask, perteneciente a la clase TField, durante la edición de un campo. Sin embargo, Edit-Mask es una propiedad protegida de TDBEdit; el motivo es permitir que la máscara de edición se asigne directamente desde el campo asociado, dentro de la VCL, y que el programador no pueda interferir en esta asignación. Si el campo tiene una máscara de edición definida, la propiedad IsMasked del cuadro de edición se vuelve verdadera. Windows no permite definir alineación para el texto de un cuadro de edición, a no ser que el estilo del cuadro sea multilíneas. Si nos fijamos un poco, el control TEdit estándar no tiene una propiedad Alignment. Sin embargo, es común mostrar los campos numéricos de una tabla justificados a la derecha. Es por eso que, en el código fuente de la VCL, se realiza un truco para permitir los distintos tipos de alineación en los componentes TDBEdit. Es importante comprender que Alignment solamente funciona durante la visualización, no durante la edición. La alineación se extrae de la propiedad correspondiente del componente de acceso al campo. Los eventos de este control que se utilizan con mayor frecuencia son OnChange, OnKeyPress, OnKeyDown y OnExit. OnChange se produce cada vez que el texto en el control es modificado. Puede causar confusión el que los componentes de campos tengan un evento también nombrado OnChange. Este último evento se dispara cuando se modifica el contenido del campo, lo cual sucede cuando se realiza una asignación al componente de campo. Si el campo se está editando en un TDBEdit, esto sucede al abandonar el control, o al pulsar INTRO estando el control seleccionado. En cambio, el evento OnChange del control de edición se dispara cada vez que se cambia algo en el control. El siguiente evento muestra cómo pasar al control siguiente cuando se alcanza la longitud máxima admitida por el editor. Este comportamiento era frecuente en programas realizados para MS-DOS:

```
void __fastcall TForm1::DBEdit1Change(TObject *Sender)
{   if (Visible)
    {      TDBEdit &edit = dynamic_cast<TDBEdit&>(*Sender);
    if (edit.Text.Length() >= edit.MaxLength)
    SelectNext(&edit, True, True);
    }
}
```

· Editores de texto

Cuando el campo a editar contiene varias líneas de texto, debemos utilizar un componente TDBMemo. Si queremos además que el texto tenga formato y permita el uso de negritas, cursivas, diferentes colores y alineaciones podemos utilizar el componente TDBRichEdit.

TDBMemo está limitado a un máximo de 32KB de texto, además de permitir un solo tipo de letra y de alineación para todo su contenido. Las siguientes propiedades han sido heredadas del componente TMemo y determinan la apariencia y forma de interacción con el usuario:

Propiedad	**Significado**
Alignment	La alineación del texto dentro del control.
Lines	El contenido del control, como una lista de cadenas de caracteres.
ScrollBars	Determina qué barras de desplazamiento se muestran.
WantTabs	Si está activa, el editor interpreta las tabulaciones como tales; en caso contrario, sirven para seleccionar el próximo control.
WordWrap	Las líneas pueden dividirse si sobrepasan el extremo derecho del editor.

La propiedad AutoDisplay es específica de este tipo de controles. Como la carga y visualización de un texto con múltiples líneas puede consumir bastante tiempo, se puede asignar *False* a esta propiedad para que el control aparezca vacío al mover la fila activa. Luego se puede cargar el contenido en el control pulsando INTRO sobre el mismo, o llamando al método *LoadMemo*.

El componente *TDBRichEdit*, por su parte, es similar a *TDBMemo*, excepto por la mayor cantidad de eventos y las propiedades de formato.

· Textos no editables

El tipo *TLabel* tiene un equivalente *data-aware*: el tipo *TDBText*. Mediante este componente, se puede mostrar información como textos no editables. Gracias a que este control desciende por herencia de la clase *TGraphicControl*, desde el punto de vista de Windows no es una ventana y no consume recursos. Si utilizamos un *TDBEdit* con la propiedad *ReadOnly* igual a *True*, consumiremos un *handle* de ventana. En compensación, con el *TDBEdit* podemos seleccionar texto y copiarlo al Portapapeles, cosa imposible de realizar con un *TDBText*.

Además de las propiedades usuales en los controles de bases de datos, *DataSource* y *DataField*, la otra propiedad interesante es *AutoSize*, que indica si el ancho del control se ajusta al tamaño del texto o si se muestra el texto en un área fija, truncándolo si la sobrepasa.

· Combos y listas con contenido fijo

Los componentes *TDBComboBox* y *TDBListBox* son las versiones *data-aware* de *TComboBox* y *TListBo*x, respectivamente. Se utilizan, preferentemente, cuando hay un número bien determinado de valores que puede aceptar un campo, y queremos ayudar al usuario para que no tenga que teclear el valor, sino que pueda seleccionarlo con el ratón o el teclado. En ambos casos, la lista de valores predeterminados se indica en la propiedad *Item*s, de tipo *TString*s.

De estos dos componentes, el cuadro de lista es el menos utilizado. No permite introducir valores diferentes a los especificados; si el campo asociado del registro activo tiene ya un valor no especificado, no se selecciona ninguna de las líneas. Tampoco permite búsquedas incrementales sobre listas ordenadas. Si las opciones posibles en el campo son pocas, la mayoría de los usuarios y programadores prefieren utilizar el componente *TDBRadioGrou*p, que estudiaremos en breve.

En cambio, *TDBComboBox* es más flexible. En primer lugar, nos deja utilizar tres estilos diferentes de interacción mediante la propiedad *Styl*e; en realidad son cinco estilos, pero dos de ellos tienen que ver más con la apariencia del control que con la interfaz con el usuario:

Estilo	**Significado**
csSimple	La lista siempre está desplegada. Se pueden teclear valores que no se encuentran en la lista.
csDropDown	La lista está inicialmente recogida. Se pueden teclear valores que no se encuentran en la lista.
csDropDownList	La lista está inicialmente recogida. No se pueden teclear valores que no se encuentren en la lista.
csOwnerDrawFixed	El contenido de la lista lo dibuja el programador. Líneas de igual altura.
csOwnerDrawVariable	El contenido de la lista lo dibuja el programador. La altura de las líneas la determina el programador.

Por otra parte, la propiedad *Sorted* permite ordenar dinámicamente los elementos de la lista desplegable de los combos. Los combos con el valor *csDropDownList* en la propiedad *Styl*e, y cuya propiedad *Sorted* es igual a *Tru*e, permiten realizar búsquedas incrementales. Si, por ejemplo, un combo está mostrando nombres de países, al teclear la letra *A* nos situaremos en *Abisini*a, luego una *N* nos llevará hasta la *Antártid*a, y así en lo adelante. Si queremos iniciar la búsqueda desde el principio, o sea, que la *N* nos sitúe sobre *Nepal*, podemos pulsar ESC o RETROCESO … o esperar dos segundos.

Esto último es curioso, pues la duración de ese intervalo está incrustada en el código de la VCL y no puede modificarse fácilmente.

Cuando el estilo de un combo es *csOwnerDrawFixed* ó *csOwnerDrawVariabl*e, es posible dibujar el contenido de la lista desplegable; para los cuadros de lista, *Style* debe valer *lbOwnerDrawFixed* ó *lbOwnerDrawVariabl*e. Si utilizamos alguno de estos estilos, tenemos que crear una respuesta al evento *OnDrawItem* y, si el estilo de dibujo es con alturas variables, el evento *OnMeasureIte*m. Estos eventos tienen los siguientes parámetros:

```
void __fastcall TForm1::DBComboBox1DrawItem(TWinControl *Control,
int Index, TRect Rect, TOwnerDrawState State)
{
}
void __fastcall TForm1::DBComboBox1MeasureItem(TWinControl *Control,
int Index, int &Altura)
{
}
```

Una propiedad poco conocida de *TDBComboBo*x, que éste hereda de *TComboBo*x, es *DroppedDow*n, de tipo lógico. *DroppedDown* es una propiedad de tiempo de ejecución, y permite saber si la lista está desplegada o no. Pero también permite asignaciones, para ocultar o desplegar la lista.

· Combos y listas de búsqueda

En cualquier diseño de bases de datos abundan los campos sobre los que se han definido restricciones de integridad referencial. Estos campos hacen referencia a valores almacenados como claves primarias de otras tablas. Pero casi siempre estos enlaces se realizan mediante claves artificiales, como es el caso de los códigos. ¿Qué me importa a mí que la Coca-Cola sea el artículo de código 4897 de mi base de datos particular? Soy un enemigo declarado de los códigos: en la mayoría de los casos se utilizan para permitir relaciones más eficientes entre tablas, por lo que deben ser internos a la aplicación. El usuario, en mi opinión, debe trabajar lo menos posible con códigos. ¿Qué es preferible, dejar que el usuario teclee cuatro dígitos, 4897, o que teclee el prefijo del nombre del producto?

Por estas razones, cuando tenemos que editar un campo de este tipo es preferible utilizar la clave verdadera a la clave artificial. En el caso del artículo, es mejor que el usuario pueda seleccionar el nombre del artículo que obligarle a introducir un código.

Esta traducción, de código a descripción, puede efectuarse a la hora de visualizar mediante los campos de búsqueda, que ya hemos estudiado. Estos campos, no obstante, son sólo para lectura; si queremos editar el campo original, debemos utilizar los controles *TDBLookupListBox* y *TDBLookupComboBo*x.

Las siguientes propiedades son comunes a las listas y combos de búsqueda, y determinan el algoritmo de traducción de código a descripción:

Propiedad	**Significado**
DataSource	La fuente de datos original. Es la que se modifica.
DataField	El campo original. Es el campo que contiene la referencia.
ListSource	La fuente de datos a la que se hace referencia.
KeyField	El campo al que se hace referencia.
ListField	Los campos que se muestran como descripción.

Cuando se arrastra un campo de búsqueda desde el Editor de Campos hasta la superficie de un formulario, el componente creado es de tipo *TDBLookupComboBox*. En este caso especial, solamente hace falta dar el nombre del campo de búsqueda en la propiedad *DataField* del combo o la rejilla, pues el resto del algoritmo de búsqueda es deducible a partir de las propiedades del campo base.

Los combos de búsquedas funcionan con el estilo equivalente al de un *TDBCombo-Box* ordenado y con el estilo *csDropDownList*. Esto quiere decir que no se pueden introducir valores que no se encuentren en la tabla de referencia. Pero también significa que podemos utilizar las búsquedas incrementales por teclado. Y esto es también válido para los componentes *TDBLookupListBox*.

Más adelante, cuando estudiemos la comunicación entre el BDE y las bases de datos cliente/servidor, veremos que la búsqueda incremental en combos de búsqueda es una característica *muy* peligrosa. La forma en que el BDE implementa por omisión las búsquedas incrementales insensibles a mayúsculas es un despilfarro.

¿Una solución? Utilice ventanas emergentes para la selección de valores, implementando usted mismo el mecanismo de búsqueda incremental, o diseñe su propio combo de búsqueda. Recuerde que esta advertencia solamente vale para bases de datos cliente / servidor.

Tanto para el combo como para las listas pueden especificarse varios campos en *ListField*; en este caso, los nombres de campos se deben separar por puntos y comas:

```
DBLookupListBox1->ListField = "Nombre;Apellidos";
```

Si son varios los campos a visualizar, en el cuadro de lista de *TDBLookupListBox*, y en la lista desplegable de *TDBLookupComboBox* se muestran en columnas separadas. En el caso del combo, en el cuadro de edición solamente se muestra uno de los campos: aquel cuya posición está indicada por la propiedad *ListFieldIndex*. Como por omisión esta propiedad vale 0, inicialmente se muestra el primer campo de la lista. *ListField-Index* determina, en cualquier caso, cuál de los campos se utiliza para realizar la búsqueda incremental en el control.

2.3.5. Rejillas y barras de navegación

Las rejillas de datos nos permiten visualizar de una forma cómoda y general cualquier conjunto de datos. Muchas veces se utiliza una rejilla como punto de partida para realizar el mantenimiento de tablas. Desde la rejilla se pueden realizar búsquedas, modificaciones, inserciones y borrados. Las respuestas a consultas *ad hoc* realizadas por el usuario pueden también visualizarse en rejillas. Por otra parte, las barras de navegación son un útil auxiliar para la navegación sobre conjuntos de datos, estén representados sobre rejillas o sobre cualquier otro conjunto de controles. Aquí estudiaremos este par de componentes, sus propiedades y eventos básicos, y la forma de personalizarlos.

Sin embargo, es fácil utilizar incorrectamente las rejillas de datos. Pongamos por caso que una aplicación deba manejar una tabla de clientes de 1.000.000 de registros. El programador medio coloca una rejilla y ¡hala, a navegar! Si la aplicación está basada en tablas de Paradox o dBase no hay muchos problemas. Pero si tratamos con

una base de datos SQL es casi seguro que se nos ahogue la red. Está claro que mostrar 25 filas en pantalla simultáneamente es más costoso que mostrar, por ejemplo, sólo un registro a la vez. Además, es peligroso dejar en manos de un usuario desaprensivo la posibilidad de moverse libremente a lo largo y ancho de un conjunto de datos. Si utilizamos el componente *TQuery*, en vez de *TTable*, para recuperar los datos y el usuario intenta ir al último registro del conjunto resultado, veremos cómo la red se pone literalmente al rojo vivo, mientras va trayendo al cliente cada uno de los 999.998 registros intermedios.

Afortunadamente, en nuestro caso sólo tendremos 32 registros a lo sumo, y en principio no tendremos este tipo de problemas, por lo que se ha preferido usar este tipo de visualización de las tablas de la base de datos(cuadros de edición, combos, imágenes, etc) que otros posiblemente más eficaces pero menos cómodos.

Para que una rejilla de datos "funcione", basta con asignarle una fuente de datos a su propiedad *DataSource*. Es todo. Quizás por causa de la sencillez de uso de estos componentes, hay muchos detalles del uso y programación de rejillas de datos que el desarrollador normalmente pasa por alto, o descuida explicar en su documentación para usuarios. Uno de estos descuidos es asumir que el usuario conoce todos los detalles de la interfaz de teclado y ratón de este control. Y es que esta interfaz es rica y compleja. Las teclas de movimiento son las de uso más evidente. Las flechas nos permiten movernos una fila o un carácter a la vez, podemos utilizar el avance y retroceso de página; las tabulaciones nos llevan de columna en columna, y es posible usar la tabulación inversa.

La tecla INS pone la tabla asociada a la rejilla en modo de inserción. Aparentemente, se crea un nuevo registro con los valores por omisión, y el usuario debe llenar el mismo. Para grabar el nuevo registro tenemos que movernos a otra fila. Por supuesto, si tenemos una barra de navegación asociada a la tabla, el botón *Post* produce el mismo efecto sin necesidad de cambiar la fila activa. Un poco más adelante estudiaremos las barras de navegación.

Pulsando F2, el usuario pone a la rejilla en modo de edición; C++ Builder crea automáticamente un cuadro de edición del tamaño de la celda activa para poder modificar el contenido de ésta. Esta acción también se logra automáticamente cuando el usuario comienza a teclear sobre una celda. La *edición automática* se controla desde la propiedad *AutoEdit* de la fuente de datos *(data source*) a la cual se conecta la rejilla. Para grabar los cambios realizados hacemos lo mismo que con las inserciones: pulsamos el botón *Post* de una barra de navegación asociada o nos cambiamos de fila.

Otra combinación útil es CTRL+SUPR, mediante la cual se puede borrar el registro activo en la rejilla. Cuando hacemos esto, se nos pide una confirmación. Es posible suprimir este mensaje, que es lanzado por la rejilla, y pedir una confirmación personalizada para cada tabla interceptando el evento *BeforeDelete* de la propia tabla.

La rejilla de datos tiene una columna fija, en su extremo izquierdo, que no se mueve de lugar aún cuando nos desplazamos a columnas que se encuentran fuera del área de visualización. En esta columna, la fila activa aparece marcada, y la marca depende del estado en que se encuentre la tabla base. En el estado *dsBrowse*, la

marca es una punta de flecha; cuando estamos en modo de edición, una viga I (*i-bea*m), la forma del cursor del ratón cuando se sitúa sobre un cuadro de edición; en modo de inserción, la marca es un asterisco. Como veremos, esta columna puede ocultarse manipulando las opciones de la rejilla.

Por otra parte, con el ratón podemos cambiar en tiempo de ejecución la disposición de las columnas de una rejilla, manipulando la barra de títulos. Por ejemplo, arrastrando una cabecera de columna se cambia el orden de las columnas; arrastrando la división entre columnas, se cambia el tamaño de las mismas.

· Opciones de rejillas

Muchas de las características visuales y funcionales de las rejillas pueden cambiarse mediante la propiedad *Option*s. Aunque las rejillas de datos, *TDbGrid* y las rejillas *TDrawGrid* y *TStringGrid* están relacionadas entre sí, las opciones de estas clases son diferentes. He aquí las opciones de las rejillas de datos y sus valores por omisión:

Opción	PO	Significado
dgEditing	Sí	Permite la edición de datos sobre la rejilla
dgAlwaysShowEditor	No	Activa siempre el editor de celdas
dgTitles	Sí	Muestra los títulos de las columnas
dgIndicator	Sí	La primera columna muestra el estado de la tabla
dgColumnResize	Sí	Cambiar el tamaño y posición de las columnas
dgColLines	Sí	Dibuja líneas entre las columnas
dgRowLines	Sí	Dibuja líneas entre las filas
dgTabs	Sí	Utilizar tabulaciones para moverse entre columnas
dgRowSelect	No	Seleccionar filas completas, en vez de celdas
dgAlwaysShowSelection	No	Dejar siempre visible la selección
dgConfirmDelete	Sí	Permite confirmar los borrados
dgCancelOnExit	Sí	Cancela inserciones vacías al perder el foco

dgMultiSelect No Permite seleccionar varias filas a la vez.

Muchas veces, es conveniente cambiar las opciones de una rejilla en coordinación con otras opciones o propiedades. Por ejemplo, cuando queremos que una rejilla se utilice sólo en modo de lectura, además de cambiar la propiedad *ReadOnly* es aconsejable eliminar la opción *dgEditing*. De este modo, cuando el usuario seleccione una celda, no se creará el editor sobre la celda y no se llevará la impresión de que la rejilla iba a permitir la modificación. Es esto lo que se escogió en la rejilla que permite ver los datos de la tabla 1, ya que estos no serán modificados por el usuario, sino adquiridos a través de una línea RS-232. Como ejemplo adicional, cuando preparamos una rejilla para seleccionar múltiples filas con la opción *dgMultiSelec*t, es bueno activar también la opción *dgRowSelec*t, para que la barra de selección se dibuje sobre toda la fila, en vez de sobre celdas individuales.

· Columnas a la medida

La configuración de una rejilla de datos va más allá de las posibilidades de la propiedad *Option*s. Por ejemplo, es necesario indicar el orden inicial de las columnas, los títulos, la alineación de los textos dentro de las columnas...

La propiedad *Columns* permite modificar el diseño de las columnas de una rejilla de datos. El tipo de esta propiedad es *TDBGridColumn*s, y es una colección de objetos de tipo *TColum*n. Para editar esta propiedad podemos hacer un doble clic en el valor de la propiedad en el Inspector de Objetos, o realizar el doble clic directamente sobre la propia rejilla.

La propiedades que nos interesan de las columnas son:

Propiedad	**Significado**
Alignment	Alineación de la columna
ButtonStyle	Permite desplegar una lista desde una celda, o mostrar un botón de edición.
Color	Color de fondo de la columna.
DropDownRows	Número de filas desplegables.
FieldName	El nombre del campo que se visualiza.
Font	El tipo de letra de la columna.
PickList	Lista opcional de valores a desplegar.
ReadOnly	Desactiva la edición en la columna.
Width	El ancho de la columna, en píxeles.
Title.Alignment	Alineación del título de la columna.
Title.Caption	Texto de la cabecera de columna.
Title.Color	Color de fondo del título de columna.
Title.Font	Tipo de letra del título de la columna.

Si el programador no especifica columnas en tiempo de diseño, éstas se crean en tiempo de ejecución y se llenan a partir de los valores extraídos de los campos de la tabla; observe que algunas propiedades de las columnas se corresponden a propiedades de los componentes de campo. Si existen columnas definidas en tiempo de diseño, son éstas las que se utilizan para el formato de la rejilla.

En la mayoría de las situaciones, las columnas se configuran en tiempo de diseño, pero es también posible modificar propiedades de columnas en tiempo de ejecución.

El siguiente método muestra como se pueden mostrar de forma automática en color azul las columnas de una rejilla que pertenezcan a los campos que forman parte del índice activo.

```
void __fastcall TForm1::FormCreate(TObject *Sender)

{

for (int i = 0; i < DBGrid1->Columns->Count; i++)

{

TColumn *C = DBGrid1->Columns->Items[i];

if (C->Field->IsIndexField)

C->Font->Color = clBlue;

}

}
```

En este otro ejemplo tenemos una rejilla con dos columnas, que ocupa todo el espacio interior de un formulario. Queremos que la segunda columna de la rejilla ocupe todo el área que deja libre la primera columna, aún cuando cambie el tamaño de la ventana. Se puede utilizar la siguiente instrucción:

```
void __fastcall TForm1::FormResize(TObject *Sender)

{

DBGrid1->Columns->Items[1]-> Width = DBGrid1->ClientWidth -

DBGrid1->Columns->Items[0]->Width - IndicatorWidth - 2;

}
```

El valor que se resta en la fórmula, *IndicatorWidt*h, corresponde a una variable global declarada en la unidad *DBGrid*s, y corresponde al ancho de la columna de indicadores.

Se han restado 2 píxeles para tener en cuenta las líneas de separación. Si la rejilla cambia sus opciones de visualización, cambiará el valor a restar, por supuesto. Para saber qué columna está activa en una rejilla, utilizamos la propiedad *SelectedIndex*, que nos dice su posición. *SelectedField* nos da acceso al componente de campo asociado a la columna activa. Por otra parte, si lo que queremos es la lista de campos de una rejilla, podemos utilizar la propiedad vectorial *Fields* y la propiedad entera *Field-Coun*t. Un objeto de tipo *TColumn* tiene también, en tiempo de ejecución, una propiedad *Field* para trabajar directamente con el campo asociado.

· Guardar y restaurar los anchos de columnas.

Para los usuarios de nuestras aplicaciones puede ser conveniente poder mantener el formato de una rejilla de una sesión a otra, especialmente los anchos de las columnas. Voy a mostrar una forma sencilla de lograrlo, suponiendo que cada columna de la rejilla pueda identificarse de forma única por el nombre de su campo asociado. El ejemplo utiliza ficheros de configuración, pero puede adaptarse fácilmente para hacer uso del registro de Windows.

Supongamos que el formulario *Form1* tiene una rejilla *DBGrid1* en su interior. Entonces necesitamos la siguiente respuesta al evento *OnCreate* del formulario para restaurar los anchos de columnas de la sesión anterior:

```
const AnsiString SClaveApp = "Software\\MiEmpresa\\MiAplicacion\\";

void __fastcall TForm1::FormCreate(TObject *Sender)

{

std::auto_ptr<TRegIniFile> ini(new TRegIniFile(

SClaveApp + "Rejillas\\" + Name + "." + DBGrid1->Name));

for (int i = 0; i < DBGrid1->Columns->Count; i++)

{

TColumn c = DBGrid1->Columns->Items[i];

c->Width = ini->ReadInteger("Width", c->FieldName, c->Width);

}

}
```

Estamos almacenando los datos de la rejilla *DBGrid1* en la siguiente clave del registro de Windows:

[HKEY_CURRENT_USER\MiEmpresa\MiAplicacion\Rejillas\Form1.DBGrid1]

El tercer parámetro de ReadInteger es el valor que se debe devolver si no se encuentra la clave dentro de la sección. Este valor se utiliza la primera vez que se ejecuta el programa, cuando aún no existe el fichero de configuraciones. Este fichero se debe actualizar cada vez que se termina la sesión, durante el evento OnClose del formulario:

```
void __fastcall TForm1::FormClose(TObject *Sender,

TCloseAction &Action)

{

std::auto_ptr<TRegIniFile> ini(new TRegIniFile(

SClaveApp + "Rejillas\\" + Name + "." + DBGrid1->Name));

for (int i = 0; i < DBGrid1->Columns->Count; i++)

{

TColumn *C = DBGrid1->Columns->Items[i];

ini->WriteInteger("Width", C->FieldName, C->Width);

}

}
```

Sobre la base de estos procedimientos simples, el lector puede incorporar mejoras, como la lectura de la configuración por secciones completas, y el almacenamiento de más información, como el orden de las columnas.

· Listas desplegables y botones de edición

Las rejillas de datos permiten que una columna despliegue una lista de valores para que el usuario seleccione uno de ellos. Esto puede suceder en dos contextos diferentes:

· Cuando el campo visualizado en una columna es un campo de búsqueda (lookup field).

· Cuando el programador especifica una lista de valores en la propiedad PickList de una columna.

En el primer caso, el usuario puede elegir un valor de una lista que se extrae de otra tabla. El estilo de interacción es similar al que utilizan los controles TDBLookupComboBox, que estudiaremos más adelante; no se permite, en contraste, la búsqueda incremental mediante teclado. En el segundo caso, la lista que se despliega contiene exactamente los valores tecleados por el programador en la propiedad PickList de la columna. Esto es útil en situaciones en las que los valores más frecuentes de una columna se reducen a un conjunto pequeño de posibilidades: formas de pagos, fórmulas de tratamiento (Señor, Señora, Señorita), y ese tipo de cosas. En nuestro proyecto se ha dispuesto una de estas listas en la columna *estado* de la rejilla DBGrid2, donde podemos elegir entre on, off o no (valor por defecto del campo de la tabla correspondiente).

En cualquiera de estos casos, la altura de la lista desplegable se determina por la propiedad DropDown-Rows: el número máximo de filas a desplegar.

La propiedad *ButtonStyle* determina si se utiliza el mecanismo de lista desplegable o no. Si vale *bsAut*o, la lista se despliega si se cumple alguna de las condiciones anteriores.

Si la propiedad vale *bsNon*e, no se despliega nunca la lista. Esto puede ser útil en ocasiones: supóngase que hemos definido, en la tabla que contiene las líneas de detalles de un pedido, el precio del artículo que se vende como un campo de búsqueda, que partiendo del código almacenado en la línea de detalles, extrae el precio de venta de la tabla de artículos. En este ejemplo, no nos interesa que se despliegue la lista con todos los precios existentes, y debemos hacer que *ButtonStyle* valga *bsNone* para esa columna.

Por otra parte, si asignamos el valor *bsEllipsis* a la propiedad *ButtonStyle* de alguna columna, cuando ponemos alguna celda de la misma en modo de edición aparece en el extremo derecho de su interior un pequeño botón con tres puntos suspensivos. Lo único que hace este botón es lanzar el evento *OnEditButtonClick* cuando es pulsado:

```
void __fastcall TForm1::DBGrid1EditButtonClick(TObject *Sender)
```

La columna en la cual se produjo la pulsación es la columna activa de la rejilla, que se puede identificar por su posición, *SelectedIndex*, o por el campo asociado, *SelectedField*.

Este botón también puede activarse pulsando CTRL+INTRO. El nuevo método *ShowPopupEditor* permite invocar al editor desde el programa.

La propiedad *Columns* nos permite especificar un color para cada columna por separado. ¿Qué sucede si deseamos, por el contrario, colores diferentes por fila, o incluso por celdas? Y puestos a pedir, ¿se pueden dibujar gráficos en una rejilla? Claro que sí: para eso existe el evento *OnDrawColumnCell*… pero esto excede los fines estéticos de mi trabajo, y queda reservado a la posibilidad de que algún día el depto. llegara a comercializar algún tipo de producto que incluyera alguna modificación de este programa.

*OnCellClic*k, se dispara cuando el usuario pulsa el ratón sobre una celda de datos, y *OnTitleClic*k, cuando la pulsación ocurre en alguno de los títulos de columnas.

Aunque estos eventos pueden detectarse teóricamente directamente con los eventos de ratón, *OnCellClick* y *OnTitleClick* nos facilitan las cosas al pasar, como parámetro del evento, el puntero al objeto de columna donde se produjo la pulsación.

Para aprovechar el uso de estos eventos, se realizó el siguiente fragmento de código:

```
 void __fastcall TFormppal::DBGrid1TitleClick(TColumn *Column)

{

                    try

        {

            Table1->IndexFieldNames = Column->FieldName;

                 }

                        catch(Exception&)

                 {

                 }

}
```

La propiedad *IndexFieldNames* de las tablas se utiliza para indicar por qué campo, o combinación de campos, queremos que la tabla esté ordenada. Para una tabla SQL este campo puede ser arbitrario, cosa que no ocurre para las tablas locales. Nuestra aplicación, por lo tanto, permite cambiar el criterio de ordenación de la tabla que se muestra con sólo pulsar con el ratón sobre el título de la columna por la cual se quiere ordenar.

· La barra de desplazamiento de la rejilla

En las versiones 1 y 2 de la VCL, para desesperación de muchos programadores habituados a trabajar con bases de datos locales, la barra solamente asume tres posiciones: al principio, en el centro y al final. ¿Por qué? La culpa la tienen las tablas SQL: para saber cuántas filas tiene una tabla residente en un servidor remoto necesitamos cargar todas las filas en el ordenador cliente. ¿Y todo esto sólo para que el cursor de la barra de desplazamiento aparezca en una posición proporcional?

Afortunadamente, C++ Builder 3 corrigió esta situación para las tablas locales aunque las cosas siguen funcionando igual para las bases de datos SQL.

2.3.6. Comunicación cliente / servidor

Toda la comunicación entre el Motor de Datos de Borland y los servidores SQL tiene lugar mediante sentencias SQL, incluso cuando el programador trabaja con tablas, en vez de con consultas. Para los desarrolladores en entornos cliente / servidor es de primordial importancia comprender cómo tiene lugar esta comunicación. En la mayoría de los casos, el BDE realiza su tarea eficientemente, pero hay ocasiones en las que tendremos que echarle una mano.

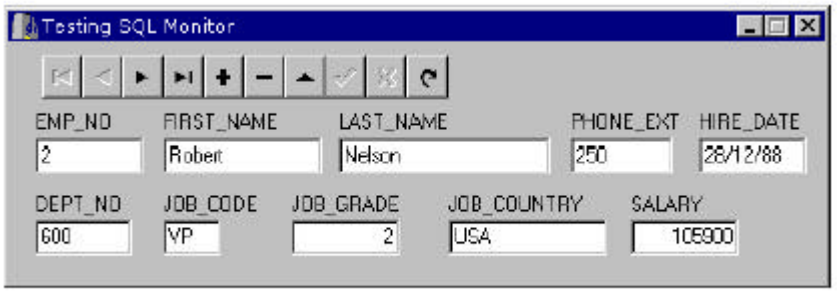
El propósito de este apartado es mostrar cómo detectar estas situaciones. Para lograrlo, veremos cómo el BDE traduce a instrucciones SQL las instrucciones de navegación y búsqueda sobre tablas y consultas. La forma en que se manejan las actualizaciones será estudiada más adelante.

· SQL Monitor

Necesitamos un espía que nos cuente qué está pasando entre el servidor y nuestro cliente, y para esta labor contamos con el SQL Monitor. Esta utilidad puede lanzarse desde el menú de programas de Windows, o directamente desde el menú *Databas*e, opción *SQL Monito*r, del propio C++ Builder. La ventana principal de este programa muestra las distintas instrucciones enviadas por el BDE y las respuestas que éste recibe del servidor. Podemos especificar qué tipos de instrucciones o de respuestas queremos mostrar en esta ventana mediante el comando de menú *Options*|*Trace option*s. En ese mismo cuadro de diálogo se ajusta el tamaño del *buffer* que albergará las instrucciones. En principio, dejaremos las opciones tal y como vienen de la fábrica. Más adelante podrá desactivar algunas de ellas para mostrar una salida más compacta y legible.

También necesitaremos una aplicación de prueba, que utilice una base de datos cliente / servidor. Para simplificar, utilizaremos un *TTable* asociado a la tabla *employee* del alias de InterBase *iblocal*. Crearemos los objetos de acceso a campos, los arrastraremos a la superficie del formulario, y añadiremos una barra de navegación. Importante: no se le ocurra utilizar una rejilla por el momento, pues se complicará la lectura e interpretación de los resultados de SQL Monitor.

La siguiente imagen muestra nuestra aplicación de pruebas en funcionamiento:



· Apertura de tablas y consultas

Estando "apagada" la aplicación, lance SQL Monitor, mediante el método que más le guste. Luego, ejecute la aplicación y vea la salida generada:

¡Más de 150 entradas, solamente por abrir una tabla y leer el primer registro! Hagamos una prueba. Sustituya la tabla con una consulta que tenga la siguiente instrucción:

```
select * from employee
```

Repita ahora el experimento, ¡y verá que con sólo 15 entradas se puede comenzar a ejecutar la aplicación!

Ahora bien, esto no significa que haya que utilizar siempre consultas, en vez de tablas, si se está programando en un entorno cliente/servidor.

Y es que hay trampa en el asunto. ¿Se ha fijado que la consulta tiene la propiedad *RequestLive* igual a *Fals*e? Cámbiela a *True* y repita la prueba, para que vea cómo vuelve a dispararse el contador de entradas en el monitor. Y pruebe después ir al último registro, tanto con la consulta como con la tabla, para que vea que la ventaja inicial de la consulta desaparece en este caso.

¿Qué está pasando? ¿Cómo podemos orientarnos entre la maraña de instrucciones del SQL Monitor? La primera regla de supervivencia es: *"Concéntrese en las instrucciones SQL Prepare y SQL Execute"*

La razón es que éstas son las órdenes que envía el BDE al servidor. Repitiendo la apertura de la tabla, ¿con qué sentencias SQL nos tropezamos? Helas aquí:

```
select rdb$owner_name, rdb$relation_name, rdb$system_flag,

rdb$view_blr, rdb$relation_id

from rdb$relations

where rdb$relation_name = 'employee'
```

El propósito de la sentencia anterior es comprobar si existe o no la tabla *employee*. Si esta instrucción diera como resultado un conjunto de filas vacío, fallaría la apertura de la tabla.

```
select r.rdb$field_name, f.rdb$field_type, f.rdb$field_sub_type,
f.rdb$dimensions, f.rdb$field_length, f.rdb$field_scale,
f.rdb$validation_blr, f.rdb$computed_blr,
r.rdb$default_value, f.rdb$default_value, r.rdb$null_flag
from rdb$relation_fields r, rdb$fields f
where r.rdb$field_source = f.rdb$field_name and
r.rdb$relation_name = 'employee'
order by r.rdb$field_position asc
select i.rdb$index_name, i.rdb$unique_flag, i.rdb$index_type,
f.rdb$field_name
from rdb$indices i, rdb$index_segments f
where i.rdb$relation_name = 'employee' and
i.rdb$index_name = f.rdb$index_name
order by i.rdb$index_id, f.rdb$field_position asc
select r.rdb$field_name, f.rdb$validation_blr, f.rdb$computed_blr,
r.rdb$default_value, f.rdb$default_value, r.rdb$null_flag
from rdb$relation_fields r, rdb$fields f
where r.rdb$field_source = f.rdb$field_name and
r.rdb$relation_name = 'employee'
order by r.rdb$field_position asc
```

No hace falta ser un especialista en InterBase para darse cuenta de lo que está pasando. El BDE está extrayendo de las tablas del sistema la información sobre qué campos, índices y restricciones están definidas para esta tabla. Estos datos se almacenan dentro de las propiedades *FieldDefs* e *IndexDefs* de la tabla.

Finalmente, se abre la consulta básica para extraer datos de la tabla:

```
select emp_no, first_name, last_name, phone_ext, hire_date, dept_no,
job_code, job_grade, job_country, salary, full_name
from employee
order by emp_no asc
```

Anote como detalle el que la consulta ordene las filas ascendentemente por el campo *Emp_N*o, que es la clave primaria de esta tabla. Dentro de poco comprenderemos por qué.

· La caché de esquemas

Traiga un botón al formulario y asocie el siguiente método con su evento *OnClic*k:

```
void __fastcall TForm1::Button1Click(TObject *Sender)

{

Table1->Close();

Table1->Open();

}
```

Si pulsamos este botón durante la ejecución de la aplicación, veremos que la segunda vez que se abre la misma tabla no se produce la misma retahíla de sentencias que al inicio, sino que directamente se pasa a leer el primer registro del cursor. La explicación es que la tabla *ya* tiene la información de su esquema almacenada en las propiedades *FieldDefs* e *IndexDef*s. Esta es una buena optimización, porque disminuye el tráfico de datos en la red. Sin embargo, cada vez que se vuelve a ejecutar la aplicación partimos de cero, y hay que traer otra vez todos los datos del catálogo. Imagine una empresa con cincuenta empleados, todos conectando su ordenador a las 9:15 de la mañana y arrancando su aplicación...

Este problema es el que resuelve la opción *ENABLE SCHEMA CACHE* del BDE, que vimos en el capítulo sobre la configuración del Motor de Datos. Así que ya sabe por qué es recomendable activar siempre la caché de esquemas.

Este no es el caso de la monitorización de la planta fotovoltaica, ya que no es probable que el servidor tenga una gran demanda de datos.

· Operaciones de navegación simple

Volvemos a las pruebas con la aplicación. Limpie el *buffer* de SQL Monitor, y pulse el botón de la barra de navegación de la aplicación que lo lleva al último registro de la tabla. Esta es la sentencia generada por el BDE, después de cerrar el cursor activo:

```
select emp_no, first_name, last_name, phone_ext, hire_date, dept_no,

job_code, job_grade, job_country, salary, full_name

from employee

order by emp_no desc
```

¡Ha cambiado el criterio de ordenación! Por supuesto, cuando el BDE ejecute la siguiente sentencia **fetch** el registro leído será el último de la tabla. Si seguimos navegando hacia atrás, con el botón *Prior*, el BDE solamente necesita ejecutar más instrucciones **fetch**, parece que nos movemos hacia atrás, pero en realidad nos movemos hacia delante. Este truco ha sido posible gracias a la existencia de una clave primaria única sobre la tabla. Algunos sistemas SQL admiten que una clave primaria sea nula, permitiendo un solo valor nulo en esa columna, por supuesto. El

problema es que, según el estándar SQL, al ordenar una secuencia de valores siendo algunos de ellos nulos, estos últimos siempre aparecerán en la misma posición: al principio o al final. Por lo que invertir el criterio de ordenación en el **select** no nos proporcionará los mismos datos en sentido inverso. El BDE no podrá practicar sus habilidades, y nos obligará a leer el millón de registros de la tabla a través de nuestra frágil y delicada red.

Precisamente eso es lo que sucede con un *TQuery*, sea actualizable o no. Cuando vamos al final del cursor, siempre se leen todos los registros intermedios.

Cuando se indica un criterio de ordenación para la tabla, ya sea mediante *IndexName* o *IndexFieldNames*, se cambia la cláusula **order by** del cursor del BDE. Sin embargo, sucede algo curioso cuando el criterio se especifica en *IndexName*: el BDE extrae los campos del índice para crear la sentencia SQL. Si la tabla es de InterBase, aunque el índice sea descendente la cláusula **order by** indicará el orden ascendente. Esto, evidentemente, es un *bug*, pues nos fuerza a utilizar una consulta si queremos ver las filas de una tabla ordenadas en forma descendente por determinada columna.

2.3.7. Actualizaciones

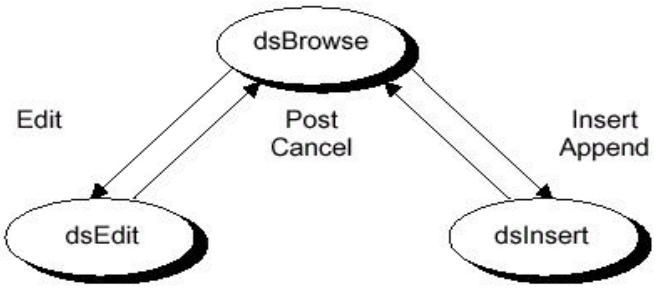
Después de crear las tablas, hay que alimentarlas con datos. En este apartado iniciaremos el estudio de los métodos de actualización. Se trata de un tema extenso, por lo cual nos limitaremos al principio a considerar actualizaciones sobre tablas aisladas, en entornos de un solo usuario, dejando para más adelante las actualizaciones coordinadas y el control de transacciones y concurrencia.

· Los estados de edición y los métodos de transición

Como ya hemos explicado, cuando un conjunto de datos se encuentra en el estado *dsBrowse*, no es posible asignar valores a los campos, pues se produce una excepción. Para realizar modificaciones, hay que cambiar el estado del conjunto de datos a uno de los estados *dsEdit* ó *dsInsert*. Para realizar la transición a estos estados, debemos utilizar el método *Edit*, si es que deseamos realizar modificaciones sobre el registro activo; si queremos añadir registros debemos utilizar *Insert* ó *Append*.

```
// Para modificar
Table1->Edit();
// … secuencia de modificación …
```

```
// Para añadir un nuevo registro
Table1->Append();
// … secuencia de inserción …
```



Cuando estudiamos los controles de datos, vimos que se podía teclear directamente valores sobre los mismos, y modificar de este modo los campos, sin necesidad de llamar explícitamente al método *Edit*. Este comportamiento se controla desde la fuente de datos, el objeto *DataSource*, al cual se conectan los controles de datos. Si este componente tiene el valor *True* en su propiedad *AutoEdit*, una modificación en el control provocará el paso del conjunto de datos al estado *Edit*. Puede ser deseable desactivar la edición automática cuando existe el riesgo de que el usuario modifique inadvertidamente los datos que está visualizando.

El método *Edit* debe releer siempre el registro actual, porque es posible que haya sido modificado desde otro puesto desde el momento en que lo leímos por primera vez.

Se puede aprovechar el comportamiento de *Edit*, que es válido tanto para bases de datos de escritorio como para bases de datos cliente / servidor, para releer el registro actual sin necesidad de llamar al método *Refresh*, que es potencialmente más costoso. Para asegurarse de que el contenido de la fila activa esté al día, ejecute el siguiente par de instrucciones:

```
Table1>Edit();

Table1>Cancel();
```

Es necesario llamar a *Cancel* para devolver la tabla al estado original *dsBrowse*.

· Asignaciones a campos

Bien, ya tenemos los campos de la tabla en un modo receptivo a nuestras asignaciones. Ahora tenemos que realizar dichas asignaciones, y ya hemos visto cómo realizar asignaciones a un campo al estudiar la implementación de los campos calculados.

Recordemos las posibilidades:

· Asignación por medio de variables de campos creadas mediante el Editor de Campos. Es la forma más eficiente de asignación, y la más segura, pues se comprueba su validez en tiempo de compilación.

```
tbClientes->Edit();

tbClientesLastInvoiceDate->Value = Date(); // Ultima factura

// …
```

· Asignación mediante los objetos de campo creados en tiempo de ejecución, por medio de la función *FieldByName* o la propiedad *Fields*. *FieldByName* es menos estable frente a cambios de nombres de columnas, y respecto a errores tipográficos.

Tampoco suministra información en tiempo de compilación acerca del tipo de campo. La propiedad *Fields* debe utilizarse solamente en casos especiales, pues nos hace depender además del orden de definición de las columnas.

```
tbClientes->Edit();

tbClientes->FieldByName("LastInvoiceDate")->AsDateTime = Date();

// …
```

· Asignación mediante la propiedad *FieldValue*s. Es la forma menos eficiente de todas, pues realiza la búsqueda de la columna y se asigna a una propiedad *Varian*t, pero es quizás la más flexible.

```
tbClientes->Edit();

tbClientes->FieldValues["LastInvoiceDate"] = Date();
```

En Delphi tiene más atractivo utilizar *FieldValue*s, pues ésta es la propiedad vectorial por omisión de la clase *TDataSet*. La siguiente instrucción sería correcta:

```
tbClientes['LastInvoiceDate'] := Date; // ¡Esto es Delphi!
```

Lamentablemente, C++ Builder no ofrece un equivalente para las propiedades por omisión, aunque hubiera sido fácil conseguirlo mediante la sobrecarga de operadores. La asignación de valores a campos mediante la propiedad *FieldValues* nos permite asignar el valor **null** de SQL a un campo. Para esto, utilizamos la variable variante especial *Null*, definida en la unidad *Syste*m:

```
tbClientes->FieldValues["LastInvoiceDate"] = Null;
```

Pero es más cómodo utilizar el método *Clear* del campo:

```
tbClientesLastInvoiceDate->Clear();
```

Es necesario tener bien claro que, aunque en Paradox y dBase un valor nulo se representa mediante una cadena vacía, esto no es así para tablas SQL.

Otra posibilidad es utilizar el método *Assign* para copiar el contenido de un campo en otro. Por ejemplo, si *Table1* y *Table2* son tablas con la misma estructura de campos, el siguiente bucle copia el contenido del registro activo de *Table2* en el registro activo de *Table*1; se asume que antes de este código, *Table1* se ha colocado en alguno de los estados de edición:

```
for (int i = 0; i < Table1->FieldCount; i++)

Table1->Fields->Fields[i]->Assign(Table2->Fields->Fields[i]);
```

Cuando se copia directamente el contenido de un campo a otro con *Assign* deben coincidir los tipos de los campos y sus tamaños. Sin embargo, si los campos son campos BLOB, esta restricción se relaja. Incluso puede asignarse a estos campos el contenido de un memo o de una imagen:

```
Table1Foto->Assign(Image1->Picture);
```

Por último, la propiedad Modified nos indica si se han realizado asignaciones sobre campos del registro activo que no hayan sido enviadas aún a la base de datos:

```
void GrabarOCancelar(TDataSet* ADataSet);

{
```

```
if (ADataSet->State == dsEdit || ADataSet->State == dsInsert)

if (ADataSet->Modified)

ADataSet->Post();

else

ADataSet->Cancel();

}
```

· Confirmando las actualizaciones

Una vez realizadas las asignaciones sobre los campos, podemos elegir entre confirmar los cambios o descartar las modificaciones, regresando en ambos casos al estado inicial, dsBrowse. Para confirmar los cambios se utiliza el método Post, indistintamente de si el conjunto de datos se encontraba en el estado dsInsert o en el dsEdit. El método Post corresponde, como ya hemos explicado, al botón que tiene la marca de verificación de las barras de navegación. Como también hemos dcho, Post es llamado implícitamente por los métodos que cambian la fila activa de un conjunto de datos, pero esta técnica es recomendada sólo para acciones inducidas por el usuario, nunca como recurso de programación. Siempre es preferible un Post explícito.

Si la tabla o consulta se encontraba inicialmente en el modo dsInsert, los valores actuales de los campos se utilizan para crear un nuevo registro en la tabla base. Si por el contrario, el estado inicial es dsEdit, los valores asignados modifican el registro activo.

En ambos casos, y esto es importante, si la operación es exitosa la fila activa de la tabla corresponde al registro nuevo o al registro modificado.

Para salir de los estados de edición sin modificar el conjunto de datos, se utiliza el método Cancel, que corresponde al botón con la "X" en la barra de navegación. Cancel restaura el registro modificado, si el estado es dsEdit, o regresa al registro previo a la llamada a Insert ó Append, si el estado inicial es dsInsert. Una característica interesante de Cancel es que si la tabla se encuentra en un estado diferente a dsInsert ó dsEdit no pasa nada, pues se ignora la llamada.

Por el contrario, es una precondición de Post que el conjunto de datos se encuentre alguno de los estados de edición; de no ser así, se produce una excepción. Hay un método poco documentado, llamado CheckBrowseMode, que se encarga de asegurar que, tras su llamada, el conjunto de datos quede en el modo dsBrowse. Si la tabla o la consulta se encuentra en alguno de los modos de edición, se intenta una llamada a *Pos*t. Si el conjunto de datos está inactivo, se lanza entonces una excepción. Esto nos ahorra repetir una y otra vez la siguiente instrucción:

```
if (Table1->State == dsEdit || Table1->State == dsInsert)

Table1->Post();
```

Es muy importante, sobre todo cuando trabajamos con bases de datos locales, garantizar que una tabla siempre abandone el estado *Edit*. La razón, como veremos más adelante, es que para tablas locales *Edit* pide un bloqueo, que no es devuelto

hasta que se llame a *Cancel* ó *Pos*t. Una secuencia correcta de edición por programa puede ser la siguiente:

```
Table1->Edit();
try
{
// Asignaciones a campos …
Table1->Post();
}
catch(Exception&)
{
Table1->Cancel();
throw;
}
```

· Diferencias entre *Insert* y *Append*

¿Por qué *Insert* y también *Appen*d? Cuando se trata de bases de datos SQL, los conceptos de inserción *in situ* y de inserción al final carecen de sentido, pues en este tipo de sistemas no existe el concepto de posición de registro. Por otra parte, el formato de tablas de dBase no permite una implementación eficiente de la inserción *in sit*u, por lo cual la llamada al método *Insert* es siempre equivalente a una llamada a *Appen*d. En realidad, en el único sistema en que estos dos métodos tienen un comportamiento diferente es en Paradox, en el caso especial de las tablas definidas sin índice primario.

Pero la explicación anterior se refiere solamente al resultado final de ambas operaciones. Si estamos trabajando con una base de datos cliente/servidor, existe una pequeña diferencia entre Insert y Append, a tener en cuenta especialmente si paralelamente estamos visualizando los datos de la tabla en una rejilla. Cuando se realiza un Append, la fila activa se desplaza al final de la tabla, por lo que el BDE necesita leer los últimos registros de la misma. Luego, cuando se grabe el registro, la fila activa volverá a desplazarse, esta vez a la posición que le corresponde de acuerdo al criterio de ordenación activo. En el peor de los casos, esto significa releer dos veces la cantidad de registros que pueden aparecer simultáneamente en pantalla. Por el contrario, si se trata de Insert, solamente se produce el segundo desplazamiento, pues inicialmente la fila activa crea un "hueco" en la posición en que se encontraba antes de la inserción. Esta diferencia puede resultar o no significativa. Si la tabla en que se está insertando contiene registros ordenados por algún campo secuencial, o por la fecha de inserción, y está ordenada por ese campo, es preferible utilizar Append, pues lo normal es que el registro quede definitivamente al final del cursor.

Un pequeño ejemplo: Vamos a generar aleatoriamente filas para una tabla; esta operación es a veces útil para comprobar el funcionamiento de ciertas técnicas de C++ Builder. La tabla para la cual generaremos datos tendrá una estructura sencilla:

una columna Cadena, de tipo cadena de caracteres, y una columna Entero, de tipo numérico. Supondremos que la clave primaria de esta tabla consiste en el campo Cadena; para el ejemplo actual es indiferente qué clave está definida. Lo primero será crear una función que nos devuelva una cadena alfabética aleatoria de longitud fija:

```
AnsiString RandomString(int Longitud)

{

char* Vocales = "AEIOU";

char LastChar = 'A';

AnsiString Rslt;

Rslt.SetLength(Longitud);

for (int i = 1; i <= Longitud; i++)

{

LastChar = strchr("AEIOUNS", LastChar) ?

random(26) + 'A' : Vocales[random(5)];

Rslt[i] = LastChar;

}

return Rslt;

}
```

Me he tomado incluso la molestia de favorecer las secuencias consonante/vocal. El procedimiento que se encarga de llenar la tabla es el siguiente:

```
void LlenarTabla(TTable *Tabla, int CantRegistros)

{

randomize();

int Intentos = 3;

while (CantRegistros > 0)

try

{

Tabla->Append();

Tabla->FieldValues["Cadena"] = RandomString(

Tabla->FieldByName("Cadena")->Size);

Tabla->FieldValues["Entero"] = random(MAXINT);

Tabla->Post();

Intentos = 3;
```

```
CantRegistros--;

}

catch(Exception&)

{

if (--Intentos == 0) throw;

}

}
```

La mayoría de las excepciones se producirán por violaciones de la unicidad de la clave primaria. En definitiva, las excepciones son ignoradas, a no ser que sobrepasemos el número predefinido de intentos; esto nos asegura contra el desbordamiento de la capacidad de un disco y otros factores imprevisibles. El número de registros se decrementa solamente cuando se produce una grabación exitosa.

NOTA IMPORTANTE

Cuando se trata de una base de datos SQL, el método de inserción masivo anterior es *muy* ineficiente. En primer lugar: cada grabación *(Post*) abre y cierra una transacción, así que es conveniente agrupar varias grabaciones en una transacción.

En segundo lugar, este algoritmo asume que estamos navegando sobre la tabla en la que insertamos, por lo que utiliza un *TTable*. Eso no es eficiente; más adelante veremos cómo utilizar un *TQuery* con parámetros para lograr más rapidez.

· Métodos abreviados de inserción

Del mismo modo que *FindKey* y *FindNearest* son formas abreviadas para la búsqueda basada en índices, existen métodos para simplificar la inserción de registros en tablas y consultas. Estos son los métodos *InsertRecord* y *AppendRecor*d:

```
void __fastcall TDataSet::InsertRecord(

const System::TVarRec *Values, const int Values_Size);

void __fastcall TDataSet::AppendRecord(

const System::TVarRec *Values, const int Values_Size);
```

En principio, por cada columna del conjunto de datos donde se realiza la inserción hay que suministrar un elemento en el vector de valores. El primer valor se asigna a la primera columna, y así sucesivamente. Pero también puede utilizarse como parámetro un vector con menos elementos que la cantidad de columnas de la tabla. En ese caso, las columnas que se quedan fueran se inicializan con el valor por omisión.

El valor por omisión depende de la definición de la columna; si no se ha especificado otra cosa, se utiliza el valor nulo de SQL.

Si una tabla tiene tres columnas, y queremos insertar un registro tal que la primera y tercera columna tengan valores no nulos, mientras que la segunda columna sea nula, podemos pasar la constante *Null* en la posición correspondiente:

```
Table1->InsertRecord(ARRAYOFCONST(("Valor1", Null, "Valor3")));
// …
Table1->AppendRecord(ARRAYOFCONST((
RandomString(Tabla->FieldByName("Cadena")->Size),
random(MAXINT))));
```

C++ Builder también ofrece el método *SetFields*, que asigna valores a los campos de una tabla a partir de un vector de valores:

```
Table1->Edit();
Table1->SetFields(ARRAYOFCONST(("Valor1", Null, "Valor3")));
Table1->Post();
```

El inconveniente de estos métodos abreviados se ve fácilmente: nos hacen dependientes del orden de definición de las columnas de la tabla. Se reestructura la tabla y ¡adiós inserciones!

Hasta el momento hemos asumido que solamente nuestra aplicación tiene acceso, desde un solo puesto, a los datos con los que trabaja. Aún sin trabajar con bases de datos SQL en entornos cliente/servidor, esta suposición es irreal, pues casi cualquier escenario de trabajo actual cuenta con varios ordenadores conectados en una red puesto a puesto; a una aplicación para los formatos de datos más sencillos, como Paradox y dBase, se le exigirá que permita el acceso concurrente a éstos. ¿Qué sucede cuando varias aplicaciones intentan modificar simultáneamente el mismo registro? En vez de especular sobre la respuesta, lo más sensato es realizar un sencillo experimento que nos aclare el desarrollo de los acontecimientos. Aunque el experimento ideal debería involucrar al menos dos ordenadores, podremos arreglárnosla ejecutando dos veces la misma aplicación en la misma máquina. Lo más importante que descubriremos es que sucederán cosas diferentes cuando utilicemos bases de datos de escritorio y servidores SQL. La aplicación en sí será muy sencilla: una tabla, una fuente de datos *(TDataSourc*e), una rejilla de datos *(TDBGri*d) y una barra de navegación *(TDBNavigato*r), esta última para facilitarnos las operaciones sobre la tabla.

· Tablas locales

En su primera versión, la tabla debe referirse a una base de datos local, a una tabla en formato Paradox ó dBase. Para lograr que la misma aplicación se ejecute dos veces sin necesidad de utilizar el Explorador de Windows o el menú Inicio, cree el siguiente método en respuesta al evento *OnCreate* del formulario principal:

```
void __fastcall TForm1::FormCreate(TObject *Sender)
{
Top = 0;
Width = Screen->Width / 2;
Height = Screen->Height;
if (CreateMutex(NULL, False,
ExtractFileName(Application->ExeName).c_str()) != 0 &&
GetLastError() == ERROR_ALREADY_EXISTS)
Left = Screen->Width / 2;
else
{
Left = 0;
WinExec(Application->ExeName.c_str(), SW_NORMAL);
}
}
```

Para detectar la presencia de la aplicación he utilizado un semáforo binario soportado por Windows, llamado *mute*x, que se crea con la función *CreateMute*x. Para que se reconozca globalmente el semáforo hay que asignarle un nombre, que

estamos formando a partir del nombre de la aplicación. Ejecute dos instancias de la aplicación y efectúe entonces la siguiente secuencia de operaciones:

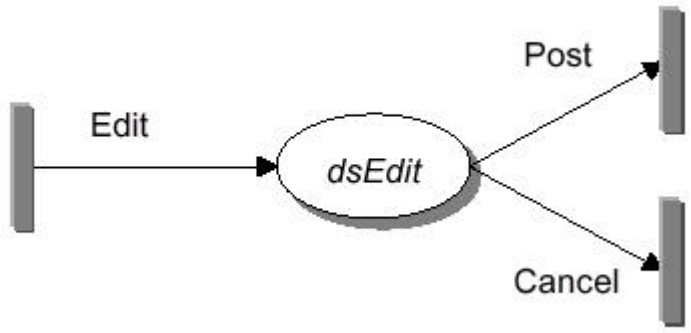
· Sitúese sobre un registro cualquiera de la rejilla, en la primera aplicación. Digamos, por conveniencia, que éste sea el primer registro.

· Ponga la tabla en estado de edición, pulsando el botón del triángulo *(Edi*t) de la barra de navegación. El mismo efecto puede obtenerlo tecleando cualquier cosa en la rejilla, pues la fuente de datos tiene la propiedad *AutoEdit* con el valor por omisión, *Tru*e. En cualquiera de estos dos casos, tenga cuidado de no cambiar la fila activa, pues se llamaría automáticamente a *Pos*t, volviendo la tabla al modo *dsBrows*e.

· Deje las cosas tal como están en la primera aplicación, y pase a la segunda.

· Sitúese en el mismo registro que escogió para la primera aplicación e intente poner la tabla en modo de edición, pulsando el correspondiente botón de la barra de navegación, o tecleando algo.

He escrito "intentar", porque el resultado de esta acción es un mensaje de error: "Registro bloqueado por otro usuario". Si tratamos de modificar una fila diferente no encontraremos problemas, lo que quiere decir que el bloqueo se aplica solamente al registro que se está editando en la otra aplicación, no a la tabla completa. También puede comprobar que el registro vuelve a estar disponible en cuanto guardamos las modificaciones realizadas en la primera aplicación, utilizando el botón de la marca de verificación (√) o moviéndonos a otra fila. Lo mismo sucede si se cancela la operación.



La conclusión a extraer de este experimento es que, para las tablas locales, el método *Edi*t, que se llama automáticamente al comenzar alguna modificación sobre la tabla, intenta colocar un bloqueo sobre la fila activa de la tabla. Este bloqueo puede eliminarse de dos formas: con el método *Pos*t, al confirmar los cambios, y con el método *Cance*l, al descartarlos.

· Tablas SQL

Repetiremos ahora el experimento, pero cambiando el formato de la tabla sobre la cual trabajamos. Esta vez conecte con una tabla de InterBase, da lo mismo una que otra. Ejecute nuevamente la aplicación dos veces y siga estos pasos:

· En la primera aplicación, modifique el primer registro de la tabla, pero no confirme la grabación, dejando la tabla en modo de inserción.

· En la segunda aplicación, ponga la tabla en modo de edición y modifique el primer registro de la tabla. Esta vez no debe ocurrir error alguno. Grabe los cambios en el disco.

· Regrese a la primera aplicación e intente grabar los cambios efectuados en esta ventana.

En este momento, *sí* que tenemos un problema. La excepción se presenta con el siguiente mensaje: "No se pudo realizar la edición, porque otro usuario cambió el

registro". Hay que cancelar los cambios realizados, y entonces se releen los datos de la tabla, pues los valores introducidos por la segunda aplicación aparecen en la primera.

La explicación es, en este caso, más complicada. Acabamos de ver en acción un mecanismo *optimista* de control de edición. En contraste, a la técnica utilizada con las bases de datos locales se le denomina *pesimist*a. Ya hemos visto que un sistema de control pesimista asume, al intentar editar un registro, que cualquier otro usuario puede estar editando este registro, por lo que para iniciar la edición "pide permiso" para hacerlo. Pedir permiso es el símil de colocar un bloqueo *(loc*k): si hay realmente otro usuario editando esta fila se nos denegará dicho permiso.

C++ Builder transforma la negación del bloqueo en una excepción. Antes de lanzar la excepción, se nos avisa mediante el evento *OnEditError* de la tabla. En la respuesta a este evento tenemos la posibilidad de reintentar la operación, fallar con la excepción o fallar silenciosamente, con una excepción *EAbor*t. En la lejana época en la que los Flintstones hacían de las suyas y la mayor parte de las aplicaciones funcionaban en modo *batc*h, era de suma importancia decidir cuándo la aplicación que no obtenía un bloqueo se cansaba de pedirlo. Ahora, sencillamente, se le puede dejar la decisión al usuario. He aquí una simple respuesta al evento *OnEditError*, que puede ser compartido por todas las tablas locales de una aplicación:

```
void __fastcall TForm1::Table1EditError(TDataSet *DataSet,
EDatabaseError *E, TDataAction &Action)
{
if (MessageDlg(E->Message, mtWarning,
TMsgDlgButton() << mbRetry << mbAbort, 0) == mrRetry)
Action = daRetry;
else
Action = daAbort;
}
```

Otra posibilidad es programar un bucle infinito de reintentos. En este caso, es recomendable reintentar la operación transcurrido un intervalo de tiempo prudencial; cuando llamamos por teléfono y está la línea ocupada no marcamos frenéticamente el mismo número una y otra vez, sino que esperamos a que la otra persona termine su llamada. En este código muestro además cómo esperar un intervalo de tiempo aleatorio:

```
void __fastcall TForm1::Table1EditError(TDataSet *TDataSet,
EDatabaseError *E, TDataAction &Action)
{
// Esperar entre 1 y 2 segundos
Sleep(1000 + random(1000));
// Reintentar
Action = daRetry;
}
```

El sistema pesimista es el más adecuado para bases de datos locales, pero para bases de datos SQL en entornos cliente/servidor, donde toda la comunicación transcurre a través de la red, y en la que se trata de maximizar la cantidad de usuarios que pueden acceder a las bases de datos, no es la mejor política. En este tipo de sistemas, el método *Edi*t, que marca el comienzo de la operación de modificación, no intenta colocar el bloqueo sobre la fila que va a cambiar. Es por eso que no se produce una excepción al poner la misma fila en modo de edición por dos aplicaciones simultáneas.

Las dos aplicaciones pueden realizar las asignaciones al *buffer* de registro sin ningún tipo de problemas. Este *buffer* reside en el ordenador cliente. La primera de ellas en terminar puede enviar sin mayores dificultades la petición de actualización al servidor. Sin embargo, cuando la segunda intenta hacer lo mismo, descubre que el registro que había leído originalmente ha desaparecido, y en ese momento se aborta la operación mediante una excepción. Esta excepción pasa primero por el evento

OnPostError, aunque en este caso lo mejor es releer el registro, si no se ha modificado la clave primaria, y volver a repetir la operación.

La suposición básica tras esta aparentemente absurda filosofía es que es poco probable que dos aplicaciones realmente traten de modificar el mismo registro a la vez.

Piense, por ejemplo, en un cajero automático. ¿Qué posibilidad existe de que tratemos de sacar dinero al mismo tiempo que un buen samaritano aumenta nuestras existencias económicas? Siendo de breve duración este tipo de operaciones, ¿cuán probable es que la compañía de teléfonos y la de electricidad se topen de narices al saquearnos a principios de mes? Sin embargo, la razón de más peso para adoptar una filosofía optimista con respecto al control de concurrencia es que de esta manera disminuye el tiempo en que el bloqueo está activo sobre un registro, disminuyendo por consiguiente las restricciones de acceso sobre el mismo. Ahora este tiempo depende del rendimiento del sistema, no de la velocidad con que el usuario teclea sus datos después de la activación del modo de edición.

Cuando una tabla está utilizando el modo optimista de control de concurrencia, este mecanismo se configura de acuerdo a la propiedad *UpdateMode* de la tabla en cuestión. Esta propiedad nos dice qué algoritmo utilizarán C++ Builder y el BDE para localizar el registro original correspondiente al registro modificado. Los posibles valores son los siguientes:

Valor	**Significado**
upWhereAll	Todas las columnas se utilizan para buscar el registro a modificar.
upWhereChanged	Solamente se utilizan las columnas que pertenecen a la clave primaria, más las columnas modificadas.
upWhereKeyOnly	Solamente se utilizan las columnas de la clave primaria.

El valor por omisión es *upWhereAll*. Este valor es nuestro seguro de vida, pues es el más restrictivo de los tres. Es, en cambio, el menos eficiente, porque la petición de búsqueda del registro debe incluir más columnas y valores de columnas.

Aunque *upWhereKeyOnly* parezca una alternativa más atractiva, el emplear solamente las columnas de la clave puede llevarnos en el caso más general a situaciones en que dos aplicaciones entran en conflicto durante la modificación de un registro. Piense, por ejemplo, que estamos modificando el salario de un empleado; la clave primaria de la tabla de empleados es su código de empleado. Por lo tanto, si alguien está modificando en otro lugar alguna otra columna, como la fecha de contratación, las actualizaciones realizadas por nuestra aplicación pueden sobrescribir las actualizaciones realizadas por la otra aplicación. Si nuestra aplicación es la primera que escribe, tendremos un empleado con una fecha de contratación correcta y el salario sin corregir; si somos los últimos, el salario será el correcto, pero la fecha de contratación no estará al día.

Sin embargo, el valor *upWhereChanged* puede aplicarse cuando queremos permitir actualizaciones simultáneas en diferentes filas de un mismo registro, que no modifiquen la clave primaria; esta situación se conoce como *actualizaciones ortogonales*, y volveremos a mencionarlas en el capítulo sobre bases de datos remotas.

· Utilizando procedimientos almacenados

Para ejecutar un procedimiento almacenado desde una aplicación escrita en C++Builder debemos utilizar el componente *TStoredProc*, de la página *Data Access*

de la Paleta de Componentes. Esta clase hereda, al igual que *TTable* y *TQuery*, de la clase *TDBDataSet*. Por lo tanto, técnicamente es un conjunto de datos, y esto quiere decir que se le puede asociar un *TDataSource* para mostrar la información que contiene en controles de datos. Ahora bien, esto solamente puede hacerse en ciertos casos, en particular, para los procedimientos de selección de Sybase.

¿Recuerda el lector los procedimientos de selección de InterBase, que se explicaron en el capítulo sobre *triggers* y procedimientos almacenados? Resulta que para utilizar estos procedimientos necesitamos una instrucción SQL, por lo cual la forma de utilizarlos desde C++ Builder es por medio de un componente *TQuery*.

En casi todos los casos, la secuencia de pasos para utilizar un *TStoredProc* es la siguiente:

· Asigne el nombre de la base de datos en la propiedad *DatabaseName*, y el nombre del procedimiento almacenado en *StoredProcName*.

· Edite la propiedad *Params*. *TStoredProc* puede asignar automáticamente los tipos a los parámetros, por lo que el objetivo de este paso es asignar opcionalmente valores iniciales a los parámetros de entrada.

· Si el procedimiento va a ejecutarse varias veces, prepare el procedimiento, de forma similar a lo que hacemos con las consultas paramétricas.

· Asigne, de ser necesario, valores a los parámetros de entrada utilizando la propiedad *Params* o la función *ParamByName*.

· Ejecute el procedimiento mediante el método *ExecProc*.

· Si el procedimiento tiene parámetros de salida, después de su ejecución pueden extraerse los valores de retorno desde la propiedad *Params* o por medio de la función *ParamByName*.

Pongamos por caso que queremos estadísticas acerca de cierto producto, del que conocemos su código. Necesitamos saber cuántos pedidos se han realizado, qué cantidad se ha vendido, por qué valor y el total de clientes interesados. Toda esa información puede obtenerse mediante el siguiente procedimiento almacenado:

```
create procedure EstadisticasProducto(CodProd int)
returns (TotalPedidos int, CantidadTotal int,
TotalVentas int, TotalClientes int)
as
declare variable Precio int;
begin
select Precio
from Articulos
where Codigo = :CodProd
into :Precio;
select count(Numero), count(distinct RefCliente)
from Pedidos
where :CodProd in
(select RefArticulo from Detalles
where RefPedido = Numero)
into :TotalPedidos, :TotalClientes;
select sum(Cantidad),
sum(Cantidad*:Precio*(100-Descuento)/100)
from Detalles
where Detalles.RefArticulo = :CodProd
```

```
into :CantidadTotal, :TotalVentas;
end ^
```

Para llamar al procedimiento desde C++ Builder, configuramos en el módulo de datos un componente *TStoredProc* con los siguientes valores:

Propiedad	**Valor**
DatabaseName	El alias de la base de datos
StoredProcName	*EstadisticasProducto*

Después, para ejecutar el procedimiento y recibir la información de vuelta, utilizamos el siguiente código:

```
void __fastcall TForm1::MostrarInfo(TObject *Sender)
{
AnsiString S;
if (! InputQuery("Información", "Código del producto", S)
|| Trim(S) == "") return;
TStoredProc *sp = modDatos->StoredProc1;
sp->ParamByName("CodProd")->AsString = S;
sp->ExecProc();
ShowMessage(Format(
"Pedidos: %d\nClientes: %d\nCantidad: %d\nTotal: %m",
ARRAYOFCONST((
sp->ParamByName("TotalPedidos")->AsInteger,
sp->ParamByName("TotalClientes")->AsInteger,
sp->ParamByName("CantidadTotal")->AsInteger,
sp->ParamByName("TotalVentas")->AsFloat))));
}
```

Al total de ventas se le ha dado formato con la secuencia de caracteres %m, que traduce un valor real al formato nacional de moneda.

A la hora de desarrollar la monitorización de la planta, se hizo necesario el uso de procedimientos almacenados para la inclusión de los datos adquiridos a través del puerto serie en la tabla de la base de datos, ya que esto se hacía mediante un hilo corriendo en paralelo con el hilo principal, y en este caso no se podía acceder de forma directa a los componentes de bases de datos (para los demás componentes VCL hubiera sido suficiente el uso del método `synchronize`). El procedimiento en cuestión es el que ya vimos en el apdo. relativo a la base de datos, encontrándose en el archivo “actualizarvariables.sql”:

```
   set term ^;

   create procedure ActVarInv (num smallint, tenpan float, intpan float, tenred
float, intbob float)

   as

   begin

   UPDATE tabla1

   SET tension_de_panel = :tenpan, intensidad_de_panel = :intpan,

   tension_de_red   =   :tenred,   intensidad_de_bobina=   :intbob,   potencia=
(:intpan)*(:tenpan),fecha_y_hora  =      'now'

   WHERE numero= :num;

   end ^

   set term ;^
```

Este procedimiento se encuentra en DataModule1, contenedor correspondiente al módulo de código SesionProtocolo.cpp, y el uso que se hace de él en el hilo inmerso en el módulo Surmain.cpp es el siguiente:

```
     DataModule1->StoredProc1->ParamByName("num")->AsSmallInt = trama[0];

                DataModule1->StoredProc1->ParamByName("tenpan")->AsFloat= trama[1];

                DataModule1->StoredProc1->ParamByName("intpan")->AsFloat= trama[2];

                DataModule1->StoredProc1->ParamByName("tenred")->AsFloat= trama[3];

                DataModule1->StoredProc1->ParamByName("intbob")->AsFloat= trama[4];

   DataModule1->StoredProc1->ExecProc();
```

2.3.8. Transacciones

En la jerarquía de objetos manejada por el BDE, las bases de datos y las sesiones ocupan los puestos más altos. En este capítulo estudiaremos la forma en que los componentes *TDatabase* controlan las conexiones a las bases de datos y la activación de transacciones. Dejaremos las posibilidades de los componentes de la clase *TSession* para el siguiente apartado.

· El componente *TDatabase*

Los componentes *TDatabase* de C++ Builder representan y administran las conexiones del BDE a sus bases de datos. Por ejemplo, este componente lleva la cuenta de las tablas y consultas activas en un instante determinado. Recíprocamente, las tablas y consultas están conectadas en tiempo de ejecución a un objeto *TDatabas*e, que puede haber sido definido explícitamente por el programador, utilizando en tiempo de diseño el componente *TDatabase* de la paleta de componentes, o haber sido creado implícitamente por C++ Builder en tiempo de ejecución. Para saber si una base de datos determinada ha sido creada por el programador en tiempo de diseño o es una base de datos temporal creada por C++

Builder, tenemos la propiedad de tiempo de ejecución *Temporary*, en la clase *TDatabase*. Con las bases de datos se produce la misma situación que con los componentes de acceso a campos: que pueden definirse en tiempo de diseño o crearse en tiempo de ejecución con propiedades por omisión.

Como veremos, las diferencias entre componentes *TDatabase* persistentes y dinámicos son mayores que las existentes entre ambos tipos de componentes de campos.

Las propiedades de un objeto *TDatabase* pueden editarse también mediante un cuadro de diálogo que aparece al realizar una doble pulsación sobre el componente.

Un objeto *TDatabase* creado explícitamente define siempre un alias local a la sesión a la cual pertenece. Básicamente, existen dos formas de configurar tal conexión:

· Crear un alias a partir de cero, siguiendo casi los mismos pasos que en la configuración del BDE, especificando el nombre del alias, el controlador y sus parámetros.

· Tomar como punto de partida un alias ya existente. En este caso, también se pueden alterar los parámetros de la conexión.

En cualquiera de los dos casos, la propiedad *IsSQLBased* nos dice si la base de datos está conectada a un servidor SQL o un controlador ODBC, o a una base de datos local.

Haya sido creado por la VCL en tiempo de ejecución, o por el programador en tiempo de diseño, un objeto de base de datos nos sirve para:

· Modificar los parámetros de conexión de la base de datos: contraseñas, conexiones establecidas, conjuntos de datos activos, etc.

· Controlar transacciones y actualizaciones en caché.

· Objetos de bases de datos persistentes

Comenzaremos con los objetos de bases de datos que el programador incluye en tiempo de diseño. La propiedad fundamental de estos objetos es *DatabaseName*, que corresponde al cuadro de edición *Name* del editor de propiedades. El valor almacenado en *DatabaseName* se utiliza para definir un alias local a la aplicación. La forma en que se define este alias local depende de cuál de las dos propiedades, *AliasName* ó *DriverName*, sea utilizada por el programador. *AliasName* y *DriverName* son propiedades de uso mutuamente excluyente: si se le asigna algo a una, desaparece el valor almacenado en la otra. En este sentido se parecen al par *IndexName* e *IndexFieldNames* de las tablas. O al Yang y el Yin de los taoístas.

Si utilizamos *AliasName* estaremos definiendo un alias basado en otro alias existente.

El objeto de base de datos puede utilizarse entonces para controlar las tablas pertenecientes al alias original. ¿Qué sentido tiene esto? La respuesta es que es posible modificar los parámetros de conexión del alias original. Esto quiere decir que podemos añadir parámetros nuevos en la propiedad *Param*s. Esta propiedad, declarada de tipo *TString*s, está inicialmente vacía para los objetos *TDatabas*e. Se puede, por ejemplo, modificar un parámetro de conexión existente:

```
ENABLE SCHEMA CACHE=TRUE
```

El último parámetro permite acelerar las operaciones de apertura de tablas, y puede activarse cuando la aplicación no modifica dinámicamente el esquema relacional de la base de datos creando, destruyendo o modificando la estructura de las tablas.

Otro motivo para utilizar un alias local que se superponga sobre un alias persistente es la intercepción del evento de conexión a la base de datos *(login*). En el caso de la monitorización se eligieron en principio "inversor" como nombre de usuario y "i" como contraseña, y dado que al servidor se suponía un acceso reservado, se eliminó el *login* eligiendo como parámetros del componente DataBase:

```
USER NAME=inversor
PASSWORD=i
```

No ocurrirá lo mismo en el cliente, al que deberemos restringir el acceso a personal autorizado.

Pero muchas veces los programadores utilizan el componente *TDatabase* sólo para declarar una variable de este tipo que controle a las tablas pertinentes. Si está utilizando esta técnica con este único propósito, existen mejores opciones, como veremos dentro de poco.

La otra posibilidad es utilizar *DriverNam*e. ¿Recuerda cómo se define un alias con la configuración del BDE? Es el mismo proceso: *DatabaseName* indica el nombre del nuevo alias, mientras que *DriverName* especifica qué controlador, de los disponibles, queremos utilizar. Para configurar correctamente el alias, hay que introducir los parámetros requeridos por el controlador, y para esto utilizamos también la propiedad *Param*s. De este modo, no necesitamos configurar alias persistentes para acceder a una base de datos desde un programa escrito en C++ Builder. Esto no es de aplicación en el presente trabajo, por lo que no entraremos a profundizar en este aspecto.

2.3.9. Sesiones

Si seguimos ascendiendo en la jerarquía de organización de objetos del DE, pasaremos de las bases de datos a las sesiones. El estudio de estos componentes es de especial importancia cuando necesitamos utilizar varios hilos en una misma aplicación.

· ¿Para qué sirven las sesiones?

El uso de sesiones en C++ Builder nos permite lograr los siguientes objetivos:

· Cada sesión define un usuario diferente que accede al BDE. Si dentro de una aplicación queremos sincronizar acciones entre procesos, sean realmente concurrentes o no, necesitamos sesiones.

· Las sesiones nos permiten administrar desde un mismo sitio las conexiones a bases de datos de la aplicación. Esto incluye la posibilidad de asignar valores por omisión a las propiedades de las bases de datos.

· Las sesiones nos dan acceso a la configuración del BDE. De este modo podemos administrar los alias del BDE y extraer información de esquemas de las bases de datos.

· Mediante las sesiones, podemos controlar el proceso de conexión a tablas Paradox protegidas por contraseñas.

En las versiones de 16 bits del BDE, anteriores a C++ Builder, sólo era necesaria una sesión por aplicación, porque podía ejecutarse un solo hilo por aplicación. En la primera versión de la VCL, por ejemplo, la clase *TSession* no estaba disponible como componente en la Paleta, y para tener acceso a la única sesión del programa teníamos la variable global *Session*. En estos momentos, además de poder crear sesiones adicionales en tiempo de diseño, seguimos teniendo la variable *Session*, que esta vez se refiere a la sesión por omisión. También se ha añadido la variable global *Sessions*, de la clase *TSessionList*, que permite el acceso a todas las sesiones existentes en la aplicación.

Tanto *Session*s, como *Sessio*n, están declaradas en la unidad *DBTable*s, y son creadas automáticamente por el código de inicialización de esta unidad en el caso de que sea mencionada por alguna de las unidades de su proyecto.

· Especificando la sesión

Los componentes *TSession* tienen una propiedad llamada *SessionNam*e, de tipo *AnsiStrin*g, que le sirve al BDE para identificar las sesiones activas. No pueden existir simultáneamente dos sesiones que compartan el mismo nombre dentro de un mismo proceso. La sesión por omisión, a la que se refiere la variable global *Session*, se define con el nombre de *Default*.

Cada componente *TDatabase* posee también una propiedad *SessionName* que debe coincidir con el nombre de alguna de las sesiones del proyecto. Por supuesto, el valor de esta propiedad para un *TDatabase* recién creado es *Default*, con lo que se indica que la conexión se realiza mediante la sesión por omisión. Como explicamos en el apartado anterior, todo *TDatabase* crea un alias "de sesión" dentro de la aplicación a la cual pertenece.

Ahora podemos ser más precisos: este alias temporal realmente pertenece a la sesión a la cual se asocia el componente. Esto quiere decir que podemos tener dos componentes *TDatabase* dentro de la misma aplicación con el mismo valor en sus propiedades *DatabaseNam*e, aún cuando *HandleShared* sea *False* para ambos.

En consecuencia, todos los conjuntos de datos derivados de *TDBDataSet* también contienen una propiedad *SessionNam*e. Cuando desplegamos en el Inspector de

Objetos la lista de valores asociada a la propiedad *DatabaseName* de una tabla o consulta, solamente veremos los alias persistentes *más* los alias de sesión definidos para la sesión a la cual está asociado el conjunto de datos.

· Cada sesión es un usuario

Hemos explicado que cada sesión define un acceso diferente al BDE, como si fuera un usuario distinto. La consecuencia más importante de esto es que el BDE levanta barreras de contención entre estos diferentes usuarios. Por ejemplo, si en una sesión se abre una tabla en modo exclusivo, dentro de la misma sesión se puede volver a abrir la tabla en este modo, pues la sesión no se bloquea a sí misma. Lo mismo ocurre con cualquier posible bloqueo a nivel de registro.

· El inicio de sesión y la inicialización del BDE

La inicialización del Motor de Datos de Borland por la VCL corre a cargo del componente *TSessio*n. Cada vez que se va a ejecutar un método de esta clase, la implementación verifica en primer lugar que la propiedad *Active* sea *Tru*e, es decir, que la sesión esté iniciada. Si no lo está, se inicializa el BDE, lo cual quiere decir que se cargan las estructuras de las DLLs del Motor de Datos en memoria. Si es la primera vez que se inicializa una sesión en esa máquina en particular, las DLLs necesarias se cargan en memoria por primera vez. En caso contrario, se inicializa un nuevo cliente o instancia de las DLLs.

Comprender cómo funciona la inicialización del BDE es importante, pues durante la misma la VCL especifica en qué idioma deben estar los mensajes del Motor de Datos.

· Sesiones e hilos paralelos

La principal aplicación de estas propiedades de la sesión es poder realizar operaciones de bases de datos en distintos hilos de la misma aplicación; cada hilo enchufa sus componentes de bases de datos por medio de una sesión diferente. Los servidores de automatización y las extensiones ISAPI/NSAPI para servidores de Internet permiten que varios clientes se conecten a la misma instancia de la aplicación. A cada cliente se le asigna un hilo diferente, por lo que es esencial utilizar sesiones para evitar conflictos entre las peticiones y modificaciones de datos. Todo esto lo veremos en su momento.

Ahora bien, ¿es apropiado utilizar hilos en aplicaciones clientes “normales”? El ejemplo más socorrido en los libros de C++ Builder es el de una aplicación MDI que abre una ventana hija basada en el resultado de la ejecución de una consulta. Como la ejecución de la consulta por el servidor (o por el intérprete local) puede tardar, para que el usuario no pierda el control de la interfaz de la aplicación, la apertura de la consulta se efectúa en un hilo separado. La técnica es correcta, y los motivos impecables. Pero yo nunca haría tal disparate en una aplicación real, pues cada ventana lanzada de esta manera consumiría una conexión a la base de datos, que es un recurso generalmente limitado y costoso. Aún después de haber terminado la ejecución del hilo que abre la consulta, el objeto *TQuery* sigue conectado a una sesión separada, como si hubiera un Dr. Jekyll y un Mr. Hide dentro de mi ordenador personal.

Se me ocurre, sin embargo, un ejemplo ligeramente diferente en el que sí es recomendable utilizar un hilo en paralelo. Sustituya en el párrafo anterior la palabra "consulta" por "procedimiento almacenado". Supongamos que nuestra base de datos cliente/servidor tiene definido un procedimiento que realiza una operación de mantenimiento larga y costosa, y que ese procedimiento debemos lanzarlo desde la aplicación.

En principio, el procedimiento no devuelve nada importante. Si utilizamos la técnica de lanzamiento convencional, tenemos que esperar a que el servidor termine para poder continuar con la aplicación. Y esta espera es la que podemos evitar utilizando hilos y sesiones. Así, la adquisición de datos por el puerto serie en paralelo con la monitorización en sí queda enmarcada en un hilo donde se ejecuta el preceptivo procedimiento almacenado albergado por la correspondiente sesión:

Creamos el módulo de datos *DataModule1*, y colocamos en él los siguientes tres objetos:

El objeto *Session1* solamente es necesario asignar un nombre en *SessionNam*e, por ejemplo *S*1. Este mismo nombre debe copiarse en la propiedad homónima del *TDatabas*e. Configuramos, además, este componente para que podamos conectarnos a la misma base de datos que estamos explorando en el hilo principal. Finalmente, cambiamos también *SessionName* en el procedimiento almacenado, y lo enganchamos al procedimiento almacenado en cuestión. Y muy importante: ¡dejamos inactivos a todos los objetos! No queremos que esta sesión esté abierta desde que arranque la aplicación.

Vamos ahora a programar el hilo que se encargará de ejecutar el procedimiento. Ejecutamos *File*|*New* para obtener el diálogo del Depósito de Objetos, y realizamos una doble pulsación en el icono *Thread object*.

Este experto crea, en una unidad aparte, un esqueleto de clase que debemos modificar del siguiente modo:

```
class TRead : public TThread

{

protected:

void __fastcall Execute();

public:

__fastcall TRead(bool CreateSuspended);

};
```

Hemos añadido un constructor al objeto. He aquí el cuerpo de los métodos:

```
_fastcall TRead::TRead(bool CreateSuspended)

    : TThread(CreateSuspended)

{

}
```

```
//---------------------------------------------------------------------------

void __fastcall TRead::Execute()

{

//Destruimos automáticamente el objeto thread cuando el proceso acaba

    FreeOnTerminate = true;

    while(1)

    {

    if(Terminated)

            {

            DataModule1->Database1->Close();

           DataModule1->Session1->Close();

            return;

             }

    //Protocolo de comunicación e inserción de datos en la tabla de variables

    //a través del correspondiente procedimiento almacenado.

    }

}
```

La creación y el posterior lanzamiento del hilo, la activación de la sesión y la apertura de la base de datos se realizan desde la unidad principal:

```
//Evento OnCreate de la ficha principal

ReadThread = new TRead(true);

DataModule1->Session1->Active=true;

DataModule1->Database1->Open();

....

//Primer evento OnTimer

if(!lanzado)

ReadThread->Resume();

lanzado=true;
```

El hilo en estado “suspendido”, es decir, no comienza inmediatamente su ejecución. Al lanzarlo, asigna *True* a la propiedad *FreeOnTerminat*e, para que la memoria del objeto *TRead* sea liberado al finalizar la ejecución del hilo. Lo que hace el hilo está determinado por el contenido del método *Execut*e. En éste se accede a los objetos necesarios (sesión, base de datos, procedimiento almacenado) mediante la variable global *DataModule1*; recuerde que los hilos comparten el mismo espacio de memoria dentro de una aplicación. He abierto explícitamente la base de datos antes

de ejecutar el procedimiento, y después me he asegurado de que se cierra la sesión (y con ella la base de datos).

2.3.10. Actualizaciones en caché

Las actualizaciones en caché son un recurso de las versiones de 32 its del BDE para aumentar el rendimiento de las transacciones en entornos cliente/servidor. Los conjuntos de datos de C++ Builder vienen equipados con una propiedad, *CachedUpdate*s, que decide si los cambios efectuados en el conjunto de datos son grabados inmediatamente en la base de datos o si se almacenan en memoria del ordenador cliente y se envían en bloque al servidor, a petición del programa cliente, en un momento dado.

Aquí estudiaremos las características básicas de las actualizaciones en caché, y cómo se pueden aplicar a la automatización de los procesos de entrada de datos.

· ¿Caché para qué?

¿Qué nos aporta este intrincado mecanismo? En primer lugar, mediante este recurso una transacción que requiera interacción con el usuario puede hacerse más corta. Y una transacción mientras más breve, mejor. Por otra parte, las actualizaciones en caché pueden disminuir drásticamente el número de paquetes enviados por la red. Cuando no están activas las actualizaciones en caché, cada registro grabado provoca el envío de un paquete de datos. Cada paquete va precedido de cierta información de control, que se repite para cada envío. Además, estos paquetes tienen un tamaño fijo, y lo más probable es que se desaproveche parte de su capacidad. También se benefician aquellos sistemas SQL que utilizan internamente técnicas pesimistas de bloqueos para garantizar las lecturas repetibles. En este caso, los bloqueos impuestos están activos mucho menos tiempo, durante la ejecución del método *ApplyUpdate*s. De este modo, se puede lograr en cierto modo la simulación de un mecanismo optimista de control de concurrencia.

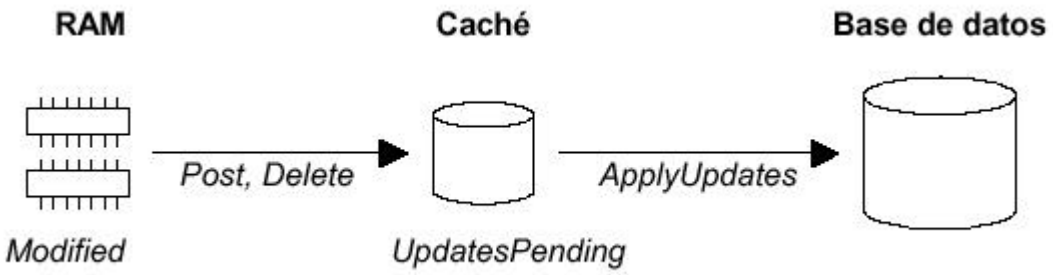
Las ventajas mencionadas se aplican fundamentalmente a los entornos cliente/servidor como el nuestro, pero también es correcto el uso de actualizaciones en caché para bases de datos locales. Esta vez la razón es de simple conveniencia para el programador, y tiene que ver nuevamente con la modificación e inserción de objetos complejos, representados en varias tablas. Se pueden utilizar las transacciones para lograr la atomicidad de estas operaciones. Pero para programar una transacción necesitamos iniciarla y confirmarla o deshacerla explícitamente, mientras que, como veremos, una actualización en caché no necesita ser iniciada: en cada momento existe sencillamente un conjunto de actualizaciones realizadas desde la última operación de confirmación. Todo lo cual significa menos trabajo para el programador.

La historia, sin embargo, no acaba aquí. Al seguir estando disponibles los datos originales de un registro después de una modificación o borrado, tenemos a nuestro alcance nuevas posibilidades: la selección de registros de acuerdo a su estado de actualización y la cancelación de actualizaciones registro a registro.

· Activación de las actualizaciones en caché

La activación del mecanismo se logra asignando el valor *True* a la propiedad *Cached-Updates* de las tablas o consultas. El valor de esta propiedad puede cambiarse incluso estando la tabla activa. Como las actualizaciones en caché utilizan internamente transacciones para su confirmación, cuando se efectúan sobre tablas locales la propiedad *TransIsolation* de la correspondiente base de datos debe valer *tiDirtyRea*d.

Se puede conocer si existen actualizaciones en la caché, pendientes de su confirmación definitiva, utilizando la propiedad de tipo lógico *UpdatesPending* para cada tabla o consulta. Observe que la propiedad *UpdatesPending* solamente informa acerca de las actualizaciones realizadas con *Post* y *Delet*e; si la tabla se encuentra en alguno de los modos de edición *dsEditModes* y se han realizado asignaciones a los campos, esto no se refleja en *UpdatesPendin*g, sino en la propiedad *Modifie*d, como siempre.



La activación de la caché de actualizaciones es válida únicamente para el conjunto de datos implicado. Si activamos *CachedUpdates* para un objeto *TTabl*e, y creamos otro objeto *TTable* que se refiera a la misma tabla física, los cambios realizados en la primera tabla no son visibles desde la segunda hasta que no se realice la confirmación de los mismos.

Una vez que las actualizaciones en caché han sido activadas, los registros del conjunto de datos se van cargando en el cliente en la medida en que el usuario va leyendo y realizando modificaciones. Es posible, sin embargo, leer el conjunto de datos completo desde un servidor utilizando el método *FetchAll*:

```
void __fastcall TDBDataSet::FetchAll();
```

De esta forma, se logra replicar el conjunto de datos en el cliente. No obstante, este método debe usarse con cuidado, debido al gran volumen de datos que puede duplicar.

· Confirmación de las actualizaciones

Existen varios métodos para la confirmación definitiva de las actualizaciones en caché.

El más sencillo es el método *ApplyUpdate*s, que se aplica a objetos de tipo *TDatabas*e.

ApplyUpdates necesita, como parámetro, la lista de tablas en las cuales se graban, de forma simultánea, las actualizaciones acumuladas en la caché:

```
void __fastcall TDatabase::ApplyUpdates(

TDBDataSet* const * DataSets,

const int DataSets_size);
```

Un detalle interesante, que nos puede ahorrar código: si la tabla a la cual se aplica el método *ApplyUpdates* se encuentra en alguno de los estados de edición, se llama de forma automática al método *Post* sobre la misma. Esto implica también que *ApplyUpdates* graba, o intenta grabar, las modificaciones pendientes que todavía residen en el *buffer* de registro, antes de confirmar la operación de actualización.

A un nivel más bajo, los conjuntos de datos tienen implementados los métodos *ApplyUpdates* y *CommitUpdate*s; la igualdad de nombres entre los métodos de los conjuntos de datos y de las bases de datos puede confundir al programador nuevo en la orientación a objetos. Estos son métodos sin parámetros:

```
void __fastcall TDataSet::ApplyUpdates();

void __fastcall TDataSet::CommitUpdates();
```

*ApplyUpdate*s, cuando se aplica a una tabla o consulta, realiza la primera fase de un protocolo en dos etapas; este método es el encargado de grabar físicamente los cambios de la caché en la base de datos. La segunda fase es responsabilidad de *Commit-Update*s.

Este método limpia las actualizaciones ya aplicadas que aún se encuentran en la caché. ¿Por qué necesitamos un protocolo de dos fases? El problema es que, si realizamos actualizaciones sobre varias tablas, y pretendemos grabarlas atómicamente, tenemos que enfrentarnos a la posibilidad de errores de grabación, ya sean provocados por el control de concurrencia o por restricciones de integridad. Por lo tanto, en el algoritmo de confirmación se han desplazado las operaciones falibles a la primera fase, la llamada a *ApplyUpdate*s; por el contrario, *CommitUpdates* no debe fallar nunca, a pesar de Murphy.

La división en dos fases la aprovecha el método *ApplyUpdates* de la clase *TDatabas*e.

Para aplicar las actualizaciones pendientes de una lsta de tablas, la base de datos inicia una transacción e intenta llamar a los métodos *ApplyUpdates* individuales de cada conjunto de datos. Si falla alguno de éstos, no pasa nada, pues la transacción se deshace y los cambios siguen residiendo en la memoria caché. Si la grabación es exitosa en conjunto, se confirma la transacción y se llama sucesivamente a *CommitUpdates* para cada conjunto de datos. El esquema de la implementación de *ApplyUpdates* es el siguiente:

```
StartTransaction(); // this = la base de datos
try
{
for (int i = 0; i <= DataSets_size; i++)
DataSets[i]->ApplyUpdates(); // Pueden fallar
Commit();
}
catch(Exception&)
Rollback();
throw; // Propagar la excepción
}
for (int i = 0; i <= DataSets_size; i++)
DataSets[i]->CommitUpdates(); // Nunca fallan
```

Es recomendable llamar siempre al método *ApplyUpdates* de la base de datos para confirmar las actualizaciones, en vez de utilizar los métodos de los conjuntos de datos, aún en el caso de una sola tabla o consulta. No obstante, es posible aprovechar estos procedimientos de más bajo nivel en circunstancias especiales, como puede suceder cuando queremos coordinar actualizaciones en caché sobre dos bases de datos diferentes.

Por último, una advertencia: como se puede deducir de la implementación del método *ApplyUpdates* aplicable a las bases de datos, las actualizaciones pendientes se graban en el orden en que se pasan las tablas dentro de la lista de tablas. Por lo tanto, si estamos aplicando cambios para tablas en relación *master/detai*l, hay que pasar primero la tabla maestra y después la de detalles. De este modo, las filas maestras se graban antes que las filas dependientes. Por ejemplo:

```
Database1->ApplyUpdates(
OPENARRAY(TDBDataSet*, (tbPedidos, tbDetalles)));
```

Debemos tener en cuenta que, en bases de datos cliente/servidor, los *triggers* asociados a una tabla se disparan cuando se graba la caché, no en el momento en que se ejecuta el *Post* sobre la tabla correspondiente. Lo mismo sucede con las restricciones de unicidad y de integridad referencial. Las restricciones **chec**k, sin embargo, pueden duplicarse en el cliente mediante las propiedades *Constraint*s, del conjunto de datos, y *CustomConstraint* ó *ImportedConstraint* a nivel de campos.

· Marcha atrás

En contraste, no existe un método predefinido que descarte las actualizaciones pendientes en todas las tablas de una base de datos. Para descartar las actualizaciones pendientes en caché, se utiliza el método *CancelUpdate*s, aplicable a objetos de tipo *TDBDataSe*t. Del mismo modo que *ApplyUpdates* llama automáticamente a *Pos*t, si

el conjunto de datos se encuentra en algún estado de edición, *CancelUpdates* llama implícitamente a *Cancel* antes de descartar los cambios no confirmados.

El siguiente procedimiento muestra una forma sencilla de descartar cambios en una lista de conjuntos de datos:

```
void DescartarCambios(TDBDataSet* const* DataSets,
int DataSets_size)
{
for (int i = 0; i <= DataSets_size; i++)
DataSets[i]->CancelUpdates();
}
```

También se pueden cancelar las actualizaciones para registros individuales. Esto se consigue con el método *RevertRecord*, que devuelve el registro a su estado original.

2.3.11. Conjuntos de datos clientes

Ser organizados tiene su recompensa, y no hay que esperar a morirse e r al Cielo para disfrutar de ella. Al parecer, los programadores de Borland hicieron sus deberes sobresalientemente al diseñar la biblioteca dinámica *dbclient.dll*, y el tipo de datos *TClientDataSet*, que se comunica con ella.

Al aislar el código de manejo de bases de datos en memoria en esta biblioteca, han podido mejorar ostensiblemente las posibilidades de la misma al desarrollar la versión 4 de la VCL.

Los *conjuntos de datos cliente*s, o *client datasets*, son objetos que pertenecen a una clase derivada de *TDataSet*: la clase *TClientDataSet*. Estos objetos almacenan sus datos en memoria RAM local, pero la característica que los hizo famosos en C++ Builder 3 es que pueden leer estos datos desde un servidor de datos remoto mediante automatización DCOM.

Aquí nos concentraremos en las características de *TClientDataSet* que nos permiten gestionar bases de datos en memoria, utilizando como origen de datos a ficheros “planos” del sistema operativo. ¿Qué haría usted si tuviera que implementar una aplicación de bases de datos, pero cuyo volumen de información esté en el orden de 1.000 a 10.000 registros? En vez de utilizar Paradox, dBase o Access, que requieren la presencia de un voluminoso y complicado motor de datos, puede elegir los conjuntos de datos en memoria. En C++ Builder 4 estos conjuntos soportan características avanzadas como tablas anidadas, valores agregados mantenidos, índices dinámicos, etc. Para más adelante dejaremos las propiedades, métodos y eventos que hacen posible la comunicación con servidores Midas.

· **Creación de conjuntos de datos**

El componente *TClientDataSet* está situado en la página *Midas* de la Paleta de Componentes. Si se trae uno de ellos a un formulario y se pregunta: ¿cuál es el esquema de los datos almacenados por el componente? Si el origen de nuestros datos es un servidor remoto, la definición de campos, índices y restricciones del conjunto de datos se leen desde el servidor, por lo cual no tenemos que preocuparnos en ese sentido.

- **Cómo el *TClientDataSet* obtiene sus datos**

Un componente *TClientDataSet* siempre almacena sus datos en dos vectores situados en la memoria del ordenador donde se ejecuta la aplicación. Al primero de estos vectores se accede mediante la propiedad *Data*, de tipo *OleVariant*, y contiene los datos leídos inicialmente por el componente; después veremos desde dónde se leen estos datos. La segunda propiedad, del mismo tipo que la anterior, se denomina *Delta*, y contiene los cambios realizados sobre los datos iniciales. Los datos reales de un *TClientDataSet* son el resultado de mezclar el contenido de las propiedades *Data* y *Delta.*

Hay varias formas de obtener estos datos:

· A partir de ficheros del sistema operativo.

· Copiando los datos almacenados en otro conjunto de datos.

· Mediante una conexión a una interfaz *IProvider* suministrada por un servidor de datos remoto.

Para obtener datos a partir de un fichero, se utiliza el método *LoadFromFile*. El fichero debe haber sido creado por un conjunto de datos (quizás éste mismo) mediante una llamada al método *SaveToFil*e, que también está disponible en el menú local del componente. Los ficheros, cuya extensión por omisión es *cd*s, no almacenan los registros en formato texto, sino en un formato que es propio de estos componentes.

Podemos leer un conjunto de datos cliente a partir de un fichero aunque el componente no contenga definiciones de campos, ya sea en *FieldDefs* o en *Field*s.

Sin embargo, la forma más común de trabajar con ficheros es asignar un nombre de fichero en la propiedad *FileName* del componente. Si el fichero existe en el momento de la apertura del conjunto de datos, se lee su contenido. En el momento en que se cierra el conjunto de datos, se sobrescribe el contenido del fichero con los datos actuales.

- **Navegación, búsqueda y selección**

Como todo buen conjunto de datos, los *TClientDataSet* permiten las ya conocidas operaciones de navegación y búsqueda: *First*, *Next*, *Last*, *Prior*, *Locate*, rangos, filtros, etc. Permiten también especificar un criterio de ordenación mediante las propiedades *IndexName* e *IndexFieldName*s. En esta última propiedad, al igual que

sucede con las tablas SQL, podemos indicar cualquier combinación de columnas, pues el conjunto de datos cliente puede generar dinámicamente un índice de ser necesario.

- **Edición de datos**

Como hemos explicado, los datos originales de un *TClientDataSet* se almacenan en su propiedad *Data*, mientras que los cambios realizados van a parar a la propiedad *Delta.*

Cada vez que un registro se modifica, se guardan los cambios en esta última propiedad, pero sin descartar o mezclar con posibles cambios anteriores. Esto permite que el programador ofrezca al usuario comandos sofisticados para deshacer los cambios realizados, en comparación con las posibilidades que ofrece un conjunto de datos del BDE.

En primer lugar, tenemos el método *UndoLastChange*:

```
void __fastcall TClientDataSet::UndoLastChange(bool FollowChanges);
```

Este método deshace el último cambio realizado sobre el conjunto de datos. Cuando indicamos *True* en su parámetro, el cursor del conjunto de datos se desplaza a la posición donde ocurrió la acción. Con *UndoLastChange* podemos implementar fácilmente el típico comando *Deshacer* de los procesadores de texto. Claro, nos interesa también saber si hay cambios, para activar o desactivar el comando de menú, y en esto nos ayuda la siguiente propiedad:

```
__property int ChangeCount;
```

Tenemos también la propiedad *SavePoint*, de tipo entero. El valor de esta propiedad indica una posición dentro del vector de modificaciones, y sirve para establecer un punto de control dentro del mismo. Por ejemplo, supongamos que usted desea imitar una transacción sobre un conjunto de datos cliente para determinada operación larga. Esta pudiera ser una solución sencilla:

```
void __fastcall TwndMain::OperacionLarga()

{

int SP = ClientDataSet1->SavePoint;

try

{

// Aquí se realizan las distintas modificaciones …

}

catch(Exception&)

{

ClientDataSet1->SavePoint = SP;
```

```
throw;
}
}
```

A diferencia de lo que sucede con una transacción verdadera, podemos anidar varios puntos de verificación.

Cuando se guardan los datos de un *TClientDataSet* en un fichero, se almacenan por separado los valores de las propiedades *Delta* y *Data*. De este modo, cuando se reanuda una sesión de edición que ha sido guardada, el usuario sigue teniendo a su disposición todos los comandos de control de cambios. Sin embargo, puede que nos interese mezclar definitivamente el contenido de estas dos propiedades, para lo cual debemos utilizar el método siguiente:

```
void __fastcall TClientDataSet::MergeChangeLog();
```

Como resultado, el espacio necesario para almacenar el conjunto de datos disminuye, pero las modificaciones realizadas hasta el momento se hacen irreversibles. Si le interesa que todos los cambios se guarden directamente en *Data*, sin utilizar el *log*, debe asignar *False* a la propiedad *LogChange*s.

Estos otros dos métodos también permiten deshacer cambios, pero de forma radical:

```
void __fastcall TClientDataSet::CancelUpdates();

void __fastcall TClientDataSet::RevertRecord();
```

Cancelar todos los cambios significa, en este contexto, vaciar la propiedad *Delta*, y que si no se han mezclado los cambios mediante *MergeChangeLog* en ningún momento, se encontrará de repente con un conjunto de datos vacío.

2.4. Midas

En este apartado nos ocuparemos del modelo de desarrollo de aplicaciones en múltiples capas, y en particular de los servicios que ofrece la tecnología Midas de Inprise (o Borland, como se prefiera). Enfocaremos el desarrollo de servidores de datos en la capa intermedia, y veremos qué componentes nos proporciona C++ Builder para desarrollar las partes del servidor y del cliente de la aplicación, estudiando los mecanismos particulares de este modelo para resolver los problemas que ocasiona el acceso concurrente.

• ¿Qué es Midas?

Las siglas Midas quieren decir, con un poco de buena voluntad, *Multitiered Distributed Application Service*s, que traducido viene a ser, poco más o menos, Servicios para Aplicaciones Distribuidas en Múltiples Capas. Midas no es una aplicación, ni un componente, sino una serie de servicios o mecanismos que permiten transmitir conjuntos de datos entre dos aplicaciones. En la primera versión de Midas, que apareció con Delphi 3.0, el vehículo de transmisión era DCOM, aunque podíamos utilizar un sustituto, también basado en COM, denominado OLEnterprise. Con C++ Builder 5 podemos transmitir conjuntos de datos entre aplicaciones utilizando indistintamente COM/DCOM, OLEnterprise y TCP/IP.

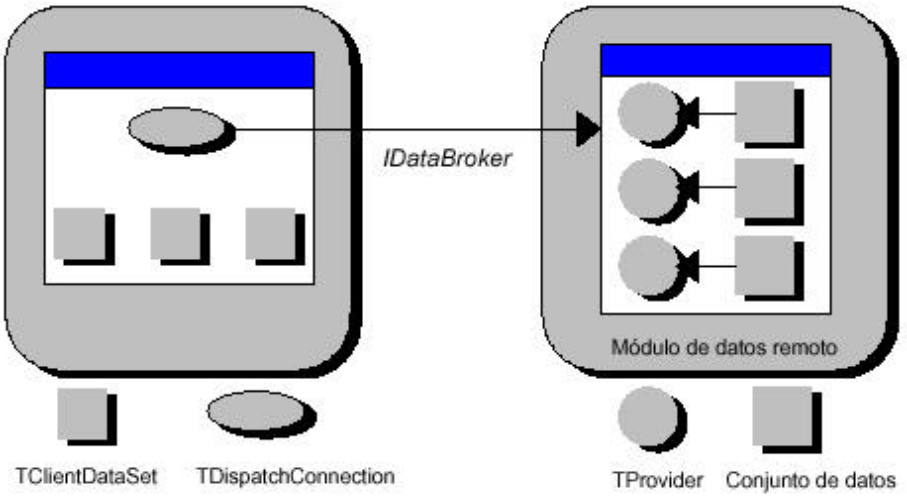
En el mecanismo básico de comunicación mediante Midas intervienen dos aplicaciones. Una actúa como servidora de datos y la otra actúa como cliente. Lo normal es que ambas aplicaciones estén situadas en diferentes ordenadores, aunque en ocasiones es conveniente que estén en la misma máquina, como veremos más adelante.

También es habitual que el servidor sea un ejecutable, aunque si vamos a colocar el cliente y el servidor en el mismo puesto es posible, y preferible, programar un servidor DLL dentro del proceso.

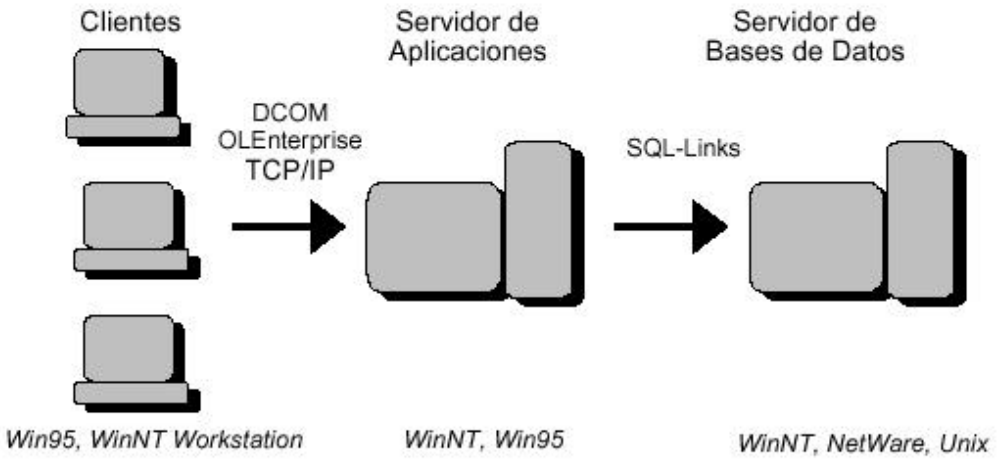
A grandes rasgos, la comunicación cliente/servidor se establece a través de una interfaz COM nombrada *IDataBroker*, que es una interfaz de automatización. Esta interfaz permite el acceso a un conjunto de *proveedores*; cada proveedor es un objeto que soporta la interfaz *IProvider*. Los proveedores se sitúan en la aplicación servidora, y cada uno de ellos proporciona acceso al contenido de un conjunto de datos diferente.

Esta estructura tiene su paralelo en la parte cliente. Los componentes derivados de la clase abstracta *TDispatchConnection* utilizan la interfaz *IDataBroker* de la aplicación servidora, y ponen a disposición del resto de la aplicación cliente los punteros a las interfaces *IProvider*. Cada clase concreta derivada de *TDispatchConnection* implementa la comunicación mediante un protocolo determinado: *TDCOMConnection*, *TSocketConnection* y *TOLEnterpriseConnection*.

Nuestro viejo conocido, el componente *TClientDataSet*, puede utilizar una interfaz *IProvider* extraída de un *TDispatchConnection* como fuente de datos. A partir de estos componentes, la arquitectura de la aplicación es similar a la tradicional, con conjuntos de datos basados en el BDE. El siguiente esquema muestra los detalles de la comunicación entre los clientes y el servidor de aplicaciones:



Las más popular de las posibles configuraciones de un sistema de este tipo es la clásica aplicación en tres capas, cuya estructura se muestra en el siguiente diagrama:



En esta configuración los datos se almacenan en un servidor dedicado de bases de datos. El sistema operativo que se ejecuta en este ordenador no tiene por qué ser Windows: UNIX, en cualquiera de sus mutaciones, o NetWare pueden ser alternativas mejores, pues la concurrencia está mejor diseñada y, en mi opinión, son más estables, aunque también más difíciles de administrar correctamente. Hay un segundo ordenador, el servidor de aplicaciones, que actúa de capa intermedia. En esta máquina está instalado el BDE y los SQL Links, con el propósito de acceder a los datos del servidor SQL. El sistema operativo, por lo tanto, debe ser Windows NT Server, preferentemente, ó Windows NT Workstation e incluso Windows 95/98. La aplicación escrita en C++ Builder (o incluso Delphi) que se ejecuta aquí no tiene necesidad de presentar una interfaz visual; puede ejecutarse en segundo plano, aunque es conveniente disponer de algún monitor de control. Finalmente, los ordenadores clientes son los que se encargan de la interfaz visual con los datos; éstos son terminales relativamente baratas, que ejecutan preferentemente Windows 95/98 ó NT Workstation.

En estos ordenadores no se instala el Motor de Datos de Borland, pues la comunicación entre ellos y el servidor intermedio se realiza a través de DCOM, OLEnterprise o TCP/IP.

- **Cuándo utilizar y cuándo no utilizar Midas**

Las bondades de Midas, pero sobre todo la propaganda acerca de la nueva técnica, ha llevado a muchos equipos de programación a lanzarse indiscriminadamente al desarrollo utilizando el modelo que acabo de presentar. Desde mi punto de vista, muchas de las aplicaciones planteadas no justifican el uso de este modelo. De lo que se trata no es de la conveniencia indiscutible de estratificar los distintos niveles de tratamiento de datos en una aplicación de bases de datos, sino de si es rentable o no que esta división se exprese físicamente. Es algo de sentido común el hecho de que al añadir una capa adicional de software o de hardware, cuya única función es la de servir de correa de transmisión, solamente logramos ralentizar la ejecución de la aplicación.

Olvidémonos por un momento de la estratificación metodológica y concentrémonos en el análisis de la eficiencia. ¿Cuál es la ventaja del modelo de dos

capas, más conocido como modelo cliente/servidor? La principal es que gran parte de las reglas de empresa pueden implementarse en el ordenador que almacena los datos. Por lo tanto, para su evaluación los datos no necesitan viajar por la red hasta alcanzar el nodo en que residen las reglas. ¿Qué sucede cuando se añade una capa intermedia? Pues que en la mayoría de las aplicaciones no existen reglas de empresa lo suficientemente complejas como para justificar un nivel intermedio: casi todo lo que puede hacer un servidor Midas puede implementarse en el propio servidor SQL. Existe una excepción importante para este razonamiento: si nuestra aplicación requiriere reglas de empresa que involucren simultáneamente a varias bases de datos. Por ejemplo, el inventario de nuestra empresa reside en una base de datos Oracle, pero el sistema de facturación utiliza InterBase. Ninguno de los servidores SQL puede asumir por sí mismo la ejecución de las reglas correspondientes, por lo que la responsabilidad debe descargarse en un servidor Midas.

Existe, sin embargo, una técnica conocida como *balance de carg*a, que en C++ Builder 3 solamente podía implementarse con OLEnterprise, pero que en la versión 4 también puede aplicarse a otros protocolos. La técnica consiste en disponer de una batería de servidores de capa intermedia similares, que ejecuten la misma aplicación servidora y que se conecten al mismo servidor SQL. Los clientes, o estaciones de trabajo, se conectan a estos servidores de forma balanceada, de forma tal que cada servidor de capa intermedia proporcione datos aproximadamente al mismo número de clientes.

Se deben dar dos condiciones para que esta configuración disminuya el tráfico de red y permita una mayor eficiencia:

· El segmento de red que comunica al servidor SQL con los servidores Midas y el que comunica a los servidores Midas con los clientes deben ser diferentes físicamente.

· Los servidores Midas deben asumir parte del procesamiento de las reglas de empresa, liberando del mismo al servidor SQL.

Otra desventaja del uso de Midas consiste en que las aplicaciones clientes deben emplear conjuntos de datos clientes, y estos componentes siguen una filosofía optimista respecto al control de cambios. Ya es bastante difícil convencer a un usuario típico de las ventajas de los bloqueos optimistas, como para además llevar este modo de acción a su mayor grado. Los conjuntos de datos clientes obligan a realizar las grabaciones mediante una acción explícita *(ApplyUpdate*s), y es bastante complicado disfrazar esta acción de modo que pase desapercibida para el usuario.

De todos modos, existen otras consideraciones aparte de la eficiencia que pueden llevarnos a utilizar servidores de capa intermedia. La principal es que los ordenadores en los que se ejecutan las aplicaciones clientes no necesitan el Motor de Datos de Borland ni, lo que es más importante, el cliente del sistema cliente/servidor. Así, después de instalar el sistema operativo, no nos veremos obligados a instalar y configurar el BDE y el cliente de InterBase en cada uno de los ordenadores donde vaya a usarse el programa. Todo resulta ser más fácil con los clientes de Midas, que solamente necesitan para su funcionamiento una DLL de 206 KB.

- **Módulos de datos remotos**

Una vez sopesados los pros y los contras de las aplicaciones Midas, veamos como crearlas. Para desarrollar una aplicación en tres capas es necesario, en primer lugar, crear la aplicación intermedia que actuará como servidor de aplicaciones. De esta manera, la aplicación cliente conocerá la estructura de los datos con que va a trabajar.

Es parecido a lo que ocurre en las aplicaciones en una o dos capas: necesitamos que las tablas estén creadas, y de ser posible con datos de prueba, para poder trabajar sobre su estructura.

Primero creamos un módulo de datos remoto, por medio del Depósito de Objetos. El icono necesario se encuentra en la página *Multitier* del Depósito, bajo el nombre de *Remote data module*. Al seleccionar el icono y pulsar el botón *Ok*, C++ Builder nos pide un nombre de clase y un modelo de concurrencia, al igual que para los objetos de automatización que hemos visto en el capítulo anterior:

Como nombre de clase utilizaremos *MastSQL*. Como modelo de instanciación, utilizaremos *Multiple instances*, que es el utilizado por omisión por la ATL; el modelo de instanciación puede cambiarse en la página *ATL* de las opciones del proyecto. De esta forma, cuando se conecten varios clientes al servidor, la aplicación se ejecutará una sola vez. Por cada cliente, sin embargo, habrá un módulo remoto diferente, ejecutándose en su propio hilo. Y como estamos creando un servidor remoto, podemos ignorar el modelo de concurrencia.

A primera vista, el módulo incorporado al proyecto tiene el mismo aspecto que un módulo normal, pero no es así. Si abrimos el Administrador de Proyectos, veremos que C++ Builder ha incluido una biblioteca de tipos en el proyecto. Esto se debe a que el módulo de datos remoto implementa una interfaz, denominada *IDataBroker*, que Borland define como descendiente de la interfaz de automatización *IDispatch*. En la biblioteca se define una nueva interfaz y una clase de componentes asociada:

```
[
uuid(03B79CE1-F03A-11D2-B67D-0000E8D7F7B2),
version(1.0),
helpstring("Dispatch interface for MastSQL Object"),
dual,
oleautomation
]
interface IMastSQL: IDataBroker
{
};
[
uuid(03B79CE3-F03A-11D2-B67D-0000E8D7F7B2),
```

```
version(1.0),
helpstring("MastSQL Object")
]
coclass MastSQL
{
[default] interface IMastSQL;
};
```

No debemos modificar directamente los ficheros importados a partir de la biblioteca, sino a través del Editor de la Biblioteca de Tipos.

En el formulario principal añadimos un nuevo atributo de tipo *TCriticalSection.*

```
class TFormppal : public TForm
{
__published: // IDE-managed Components
private: // User declarations
TCriticalSection *csection;
public: // User declarations
__fastcall TFormppal(TComponent* Owner);
};
```

La sección crítica se crea y se destruye explícitamente en el constructor del formulario, y durante la respuesta a *OnClos*e, respectivamente:

```
__fastcall TFormppal::TFormppa1(TComponent* Owner) : TForm(Owner)
{
csection = new TCriticalSection;
}
void __fastcall TFormppal::FormClose(TObject *Sender,
TCloseAction &Action)
{
delete csection;
}
```

La sentencia csection->Enter(); permitirá la entrada a la sección crítica, mientras que csection->Leave() proporcina la salida.

- **Proveedores**

Las aplicaciones clientes deben extraer sus datos mediante una interfaz remota cuyo nombre es *IProvider*. El propósito del módulo de datos remoto, además de contener los componentes de acceso a datos, es exportar un conjunto de proveedores por medio de la interfaz *IDataBroker*. Por lo tanto, nuestro próximo paso es crear estos proveedores. Comenzaremos con el caso más sencillo: una sola tabla. Más adelante veremos como manejar relaciones *master/detai*l, y cómo pueden manejarse transacciones de forma explícita.

Colocamos sendas tablas en el módulo de datos remotos, cambiamos sus nombres a *table1 y 2*, su base de datos a *inversor* y su nombre de tabla a *Tabla1 y 2*; después la abrimos mediante su propiedad *Activ*e. Pulsamos el botón derecho del ratón sobre la tabla y elegimos el comando *Export table1 from data module*.

Con esta acción hemos provocado que una nueva propiedad, *table1* de sólo lectura, aparezca en la interfaz del objeto de automatización. La declaración IDL de la interfaz se ha modificado de la siguiente forma:

```
[
uuid(03B79CE1-F03A-11D2-B67D-0000E8D7F7B2),
version(1.0),
helpstring("Dispatch interface for MastSQL Object"),
dual,
oleautomation
]
interface IMastSQL: IDataBroker
{
[propget, id(0x00000001)]
HRESULT _stdcall table1 ([out, retval] IProvider ** Value);
};
```

Como se trata de una interfaz sólo lectura, se implementa mediante un método de nombre *get_table1*, generado automáticamente por C++ Builder:

```
STDMETHODIMP TMastSQLImpl::get_tbClientes(IProvider** Value)
{
try
{
_di_IProvider IProv = m_DataModule->table1->Provider;
IProv->AddRef();
```

```
*Value = IProv;

}

catch(Exception &e)

{

return Error(e.Message.c_str(), IID_IMastSQL);

}

return S_OK;

}
```

En primer lugar, observe que la implementación de la interfaz está a cargo de la clase *TMastSQLImpl*, mientras que el módulo de datos está asociado a la clase *TMastSQL*.

No importa: la primera clase desciende por herencia de la segunda, y es la que será utilizada por la fábrica de clases para la creación de objetos.

Los conjuntos de datos basados en el BDE tienen una propiedad *Provider*, del tipo *IProvider*, que ha hecho posible exportar directamente sus datos desde el módulo remoto. Ahora bien, es preferible realizar la exportación incluyendo explícitamente un componente *TDataSetProvider*, de la página *Midas* de la Paleta de Componentes, conectándolo a un conjunto de datos y exportándolo mediante su menú de contexto.

La ventaja de utilizar este componente es la posibilidad de configurar opciones y de crear manejadores para los eventos que ofrece, ganando control en la comunicación entre el cliente, el servidor de aplicaciones y el servidor de bases de datos.

Para deshacer la exportación de *table1*, deberíamos buscar en primer lugar la Biblioteca de Tipos *(View|Type library*), seleccionar la propiedad *table1* de la interfaz *IMastSQL* y eliminarla. Después, en la unidad del módulo de datos, hay que eliminar la función *get_table1*. Traemos entonces un componente *TDataSetProvider*, de la página *Midas*, y lo situamos sobre el módulo de datos. Cambiamos las siguientes propiedades:

Propiedad	**Valor**
Name	*tabla1provider*
DataSet	*table1*
Options	Añadir *poIncFieldProps*

La opción *poIncFieldProps* hace que el proveedor incluya en los paquetes que envía al cliente información sobre propiedades de los campos, tales como *DisplayLabel*, *DisplayFormat*, *Alignment*, etc. De esta forma, podemos realizar la configuración de los campos en el servidor y ahorrarnos repetir esta operación en sus clientes. Más adelante estudiaremos otras propiedades de *TDataSetProvider*, además de sus eventos. Finalmente, pulsamos el botón izquierdo del ratón sobre el componente y seleccionamos *true* en su propiedad *exported*.

Como no estamos utilizando una tabla perteneciente a una base de datos de escritorio, no es necesario activar la opción *ResolveToDataSet* del *TDataSetProvider*. Si esta opción no está activa, el proveedor genera sentencias SQL que envía directamente a la base de datos, saltando por encima del conjunto de datos asociado. Esto es lo más conveniente para las bases de datos SQL a las cuales se accede por medio del BDE, pero no vale para Paradox, dBase y Access.

Una vez que hemos exportado una interfaz *IProvider*, de una forma u otra, podemos guardar el proyecto y ejecutarlo la primera vez; así se registra el objeto de automatización dentro del sistema operativo.

Para eliminar las entradas del registro, se debe ejecutar la aplicación con el parámetro */unregserver* en la línea de comandos.

- **Servidores remotos y conjuntos de datos clientes**

Es el momento de programar la aplicación cliente. Si no se dispone de una red con DCOM configurado de algún modo, podemos probar el cliente y el servidor dentro de la misma máquina.

Iniciamos una aplicación nueva, con un formulario vacío, y añadimos un módulo de datos de los de siempre. Sobre este módulo de datos colocamos un componente *TsocketConnectio*n, de la página *Mida*s, cuyas propiedades configuramos de este modo:

Propiedad	**Valor**	**Significado**
Computer		Nombre del ordenador donde reside el servidor. Se deja vacío si es local.
ServerName	*invServidor.MastSQL*	Identificador de clase del objeto de automatización
ServerGUID	Lo llena automáticamente C++ Builder	
Connected	*True*	Al activarla, se ejecuta el servidor de aplicaciones

Por cada proveedor que exporte el módulo de datos remoto y que nos interese, traemos al módulo local un componente *TClientDataSet*, de la página *Midas* de la Paleta de Componentes. Ya hemos visto en funcionamiento a este componente, pero trabajando con datos extraídos de ficheros locales. Ahora veremos qué propiedades, métodos y eventos tenemos que utilizar para que funcione con una interfaz *IProvider* como origen de datos. En concreto, para este proyecto traemos un *TClientDataSet* para cada tabla al módulo local, y asignamos las siguientes propiedades; después traemos un *TDataSource* para poder visualizar los datos.

Propiedad	**Valor**
RemoteServer	*SocketConnection1*

ProviderName	*tabla2provider*
Active	*True*

Hay un par de propiedades que controlan la forma en que el conjunto de datos cliente extrae los registros del servidor de aplicaciones. *FetchOnDeman*d, por ejemplo, debe ser *True* (valor por omisión) para que los registros se lean cuando sea necesario; si es *Fals*e, hay que llamar explícitamente al método *GetNextPacke*t, por lo que se recomienda dejar activa esta propiedad. *PacketRecords* indica el número de registros que se transfiere en cada pedido. En el caso en que es -1, el valor por omisión, todos los registros se transfieren en la primera operación, lo cual solamente es aconsejable para tablas pequeñas.

A partir de ahí, podemos crear y configurar los objetos de acceso a campos igual que si estuviésemos tratando con una tabla o una consulta. Si me hizo caso e incluyó la opción *poIncFieldProps* en el proveedor, se podrá ahorrar la configuración de los campos.

- **Grabación de datos**

Si realizamos modificaciones en los datos por medio de la aplicación cliente, salimos de la misma y volvemos a ejecutarla, nos encontraremos que hemos perdido las actualizaciones.

Las aplicaciones que utilizan *TClientDataSet* son parecidas a las que aprovechan las actualizaciones en caché: tenemos que efectuar una operación explícita de actualización del servidor para que éste reconozca los cambios producidos en los datos.

Ya hemos visto que *TClientDataSet* guarda los cambios realizados desde que se cargan los datos en la propiedad *Delta*, y que *ChangeCount* almacena el número de cambios.

En las aplicaciones basadas en ficheros planos, los cambios se aplicaban mediante el método *MergeChangeLog*. Ahora, utilizado proveedores como origen de datos, la primera regla del juego dice:

"Prohibido utilizar MergeChangeLog con proveedores"

Para grabar los cambios en el proveedor, debemos enviar el contenido de *Delta* utilizando la misma interfaz *IProvider* con la que rellenamos *Data*. El siguiente método es la forma más fácil de enviar las actualizaciones al servidor:

```
int __fastcall TClientDataSet::ApplyUpdates(int MaxErrors);
```

El parámetro *MaxErrors* representa el número máximo de errores tolerables, antes de abortar la operación:

· Si indicamos -1, la operación tolera cualquier número de errores.

· Si indicamos 0, se detiene al producirse el primer error.

· Si indicamos *n* > 0, se admite ese número de errores antes de abortar todo el proceso.

¿Cuándo debemos llamar al método *ApplyUpdate*s? Depende del modo de edición y de los convenios de interfaz que usted establezca con sus usuarios. Si el usuario utiliza cuadros de diálogo para insertar o modificar registros, es relativamente sencillo asociar la grabación al cierre del diálogo.

En nuestro caso, al tratarse de una modificación continua de la base de datos, se ha elegido el evento *OnTimer* de una temporización de 6 segundos, intervalo suficiente para que el cliente esté suficientemente actualizado y además el usuario pueda variar tranquilamente el estado de los inversores.

- **Resolución**

El algoritmo empleado para grabar los cambios efectuados en el cliente mediante el servidor remoto es un ballet muy bien sincronizado y ensayado, cuyas *primas ballerinas* son el cliente y el proveedor, pero en el que interviene todo un coro de componentes.

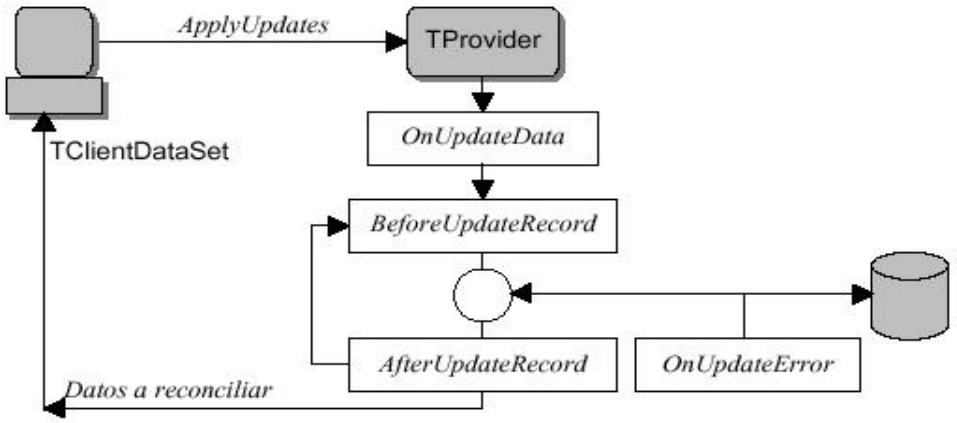
La obra comienza cuando el cliente pide la grabación de sus cambios mediante el método *ApplyUpdate*s:

```
ClientDataSet1->ApplyUpdates(0);
```

Internamente, el conjunto de datos echa mano de su interfaz *Provide*r, ejecuta el siguiente método remoto, y se queda a la espera del resultado para efectuar una operación que estudiaremos más adelante, llamada *reconciliación*:

```
Provider->ApplyUpdates(Delta, MaxErrors, Result);
```

Como vemos, al servidor Midas se le envía *Delta*: el vector que contiene los cambios diferenciales con respecto a los datos originales. Y ahora cambia el escenario de la danza, pasando la acción al servidor:



El proveedor utiliza un pequeño componente auxiliar, derivado de la clase *TCustom-Resolver*, que se encarga del proceso de grabar las diferencias en la base de

datos. En la jerga de Borland/Inprise, este proceso recibe el nombre de *resolución*. El algoritmo de resolución puede concretarse de muchas formas diferentes. El componente *TProvider*, por ejemplo, aplica las modificaciones mediante sentencias SQL que lanza directamente a la base de datos, saltándose el conjunto de datos que tiene conectado en su propiedad *DataSet*. Así se logra mayor eficiencia, pero se asume que *DataSet* apunta a un conjunto de datos del BDE. El componente *TDataSetProvider*, que también se encuentra en la página *Midas*, aplica las actualizaciones utilizando como intermediario a su conjunto de datos asociado. Es más lento, pero puede utilizarse con cualquier tipo de conjunto de datos. Y si se tiene la suficiente paciencia, puede crearse su proveedor personalizado, derivando clases a partir de *TBaseProvider* y de *TCustomResolver*.

- **Tipos de conexión**

Veamos ahora los distintos tipos de conexiones que pueden establecerse y los problemas de configuración de cada uno de ellos. El primer tipo de conexión se basa en COM/DCOM. Ya hemos visto que cuando el cliente y el servidor residen en la misma máquina es éste el tipo de protocolo que utilizan para comunicarse, y no hace falta ninguna configuración especial. Así que vamos directamente a DCOM. Para poder utilizar este protocolo, debe tener los siguientes elementos:

· Una red Windows/Windows NT en la que los ordenadores controlen el acceso a los recursos compartidos mediante el nombre de usuarios.

· Si las estaciones de trabajo tienen instalado Windows 95/98, el requisito anterior obliga a que estén conectadas a un dominio NT. Así que, en cualquier caso, necesitará un ordenador que actúe como controlador de dominio de Windows NT.

· Los ordenadores que tienen Windows 95 necesitan un parche de Microsoft para activar DCOM, tanto si van a actuar como clientes o como servidores. Si se trata de Windows 98, NT Server o Workstation, el soporte DCOM viene incorporado.

Una de las rarezas de DCOM es que no se configura, como sería de esperar, mediante el Panel de Control. Hay que ejecutar, desde la línea de comandos o desde *Inicio|Ejecutar*, el programa *dcomcnfg.exe*. La versión para Windows 95 permite activar o desactivar DCOM en el ordenador local.

La versión de *dcomcnfg* para Windows NT se utiliza para controlar la seguridad del acceso a las aplicaciones situadas en el servidor. El control de acceso es la principal ventaja que ofrece DCOM respecto a otros protocolos. Sin embargo, a veces es un poco complicado echar a andar DCOM en una red existente, configurada de acuerdo a las prácticas viciosas y perversas de algunos "administradores de redes".

En tales casos, la forma más sencilla de comunicación es utilizar el propio protocolo TCP/IP, instalando en el ordenador que contiene el servidor Midas una aplicación que actúa como *proxy*, o delegada, y que traduce las peticiones TCP/IP en llamadas a métodos COM dentro del servidor. Esta aplicación se llama *scktsrvr.exe*, y se encuentra en el directorio *bin* de C++ Builder:

La aplicación al ejecutarse se coloca en la bandeja de iconos. También hay una versión, *scktsrvc.exe*, que se ejecuta como un servicio en Windows NT. Si queremos utilizar TCP/IP con Midas, debemos ejecutar cualquiera de estas dos versiones antes de conectar el primer cliente al servidor. Da lo mismo que la máquina que contiene el servidor ejecute Windows 95, 98 o NT. ¿Qué modificaciones debemos realizar en las aplicaciones clientes y servidoras para que se conecten vía TCP/IP? En el servidor, ninguna. En el cliente debemos sustituir el componente *TDCOMConnection* por un *TSocketConnection*. El uso de este tipo de conexión puede disminuir el tráfico en la red. DCOM envía periódicamente desde el servidor a su cliente notificaciones, que le permiten saber si sus clientes siguen activos y no se ha producido una desconexión.

Así pueden ahorrarse recursos en caso de fallos, pero las notificaciones deben viajar por la red. *TSocketConnection* evita este tráfico, pero un servidor puede perder a un cliente y no darse cuenta. Este tipo de conexión tampoco soporta el control de acceso basado en roles de DCOM.

OLEnterprise es el tercer mecanismo de comunicación. Está basado en RPC, funciona en cualquiera de las variantes de Windows, y permite implementar de forma fácil estas tres deseables características:

· **Transparencia de la ubicación**: El cliente no tiene por qué conocer el ordenador exacto donde está situado su servidor. Todo lo que tiene que especificar es el nombre del servicio *(ServerGUID*) y el nombre del ordenador donde se ejecuta el Agente de Objetos *(Object Broker*). Este último es el software central de OLEnterprise, que lleva un directorio de servicios para toda la red.

· **Balance de carga**: Gracias a su arquitectura, el Agente de Objetos puede decidir a qué servidor se conecta cada cliente. Al principio de este capítulo mencionábamos la posibilidad de habilitar una batería de servidores Midas redundante para reducir el tráfico en red. Bien, el balance de carga de OLEnterprise es la técnica más completa y segura para aprovechar esta configuración.

· **Seguridad contra fallo**s: El uso de una batería de servidores redundantes permite que, cuando se cae un servidor, sus clientes puedan deshacer las transacciones pendientes, si es que existen, y conectarse a otro servidor de la lista.

La gran desventaja de OLEnterprise es que debemos instalarlo tanto en los servidores como en los clientes, y se trata de una instalación bastante pesada

Tras un estudio de los puntos anteriores se vio como mejor optativa la de emplear el protocolo TCP/IP, debido en gran parte a su robustez.

2.5. Impresión de informes con QuickReport

Toda aplicación de bases de datos debe permitir imprimir la información con la que trabaja. Sistemas de creación e impresión de informes para C++ Builder hay muchos, quizás demasiados. Inicialmente, junto con Delphi 1 se suministraba un producto de Borland, llamado ReportSmith. Era un generador de informes bastante bueno, con posibilidades de impresión de informes por columnas, *master/detail*, tabulares, gráficos, etc. Sin embargo, no tuvo la acogida deseable por parte de los

programadores. ¿La razón?: un *runtime* bastante grande e incómodo que distribuir, demasiado ineficiente en tiempo y espacio (estamos hablando de los tiempos de Delphi 1, sobre Windows 3.1, cuando 16 MB en una máquina era bastante memoria) y, sobre todo, una molesta pantalla de presentación con los créditos de Borland, que aparecía cada vez que se imprimía un informe.

Posteriormente se descubrió que era muy fácil ocultar esta pantalla, pero estoy seguro de que ésta fue una de las causas no confesadas que motivaron la aparición de toda una variedad de sistemas de informes alternativos.

En junio de 1995, un programador noruego llamado Allan Lochert colocó en la Internet un pequeño y eficiente sistema de impresión de informes de libre distribución, al que bautizó QuickReport. Era un producto simple y elegante, basado en el principio "lo pequeño es bello"; el código fuente solamente ocupaba 5000 líneas. El sistema fue creciendo y depurándose, llamando la atención de Borland, que lo incluyó como alternativa a ReportSmith en Delphi 2 y C++ Builder 1. Adicionalmente, era posible comprar aparte la versión profesional de QuickReport, que contenía el código fuente y los componentes para 16 y 32 bits.

Cuando apareció la siguiente versión de QuickReport, éste había sido reprogramado de arriba abajo, cambiando totalmente su apariencia y añadiéndose más potencia. La filosofía del producto seguía siendo la misma, y era fácil, para alguien familiarizado con la versión anterior, crear un informe con la nueva. También era posible convertir de forma automática un informe de la versión 1 a la 2, pero en ciertos casos especiales, la conversión debía complementarse manualmente.

Por desgracia, junto a la inclusión de nuevas características aparecieron visitantes no deseados: me refiero a *bug*s. Durante el año y poco que duró C++ Builder 3, QuSoft (la empresa responsable de QuickReport) sacó parches designados alfabéticamente, desde la A hasta la K. A veces un parche no sólo corregía errores, sino que introducía otros nuevos. Para colmo de males, Borland se deshizo de ReportSmith, pasando el producto a la compañía Strategic Reporting, aunque continuó incluyendo en la Paleta de Componentes el componente *TReport*, que permite establecer una conexión con el motor de impresión de ReportSmith. Este componente está inicialmente escondido, y hay que utilizar el diálogo de propiedades de la Paleta para mostrarlo.

Ahora con C++ Builder 4, tenemos la versión 3 de QuickReport. Solamente el tiempo y la experiencia nos dirá si es una versión estable y fiable. Esperemos que sí (presunción de inocencia).

QuickReport es un sistema de informes basado en *banda*s. Esto quiere decir que durante el diseño del informe no se ve realmente la apariencia final de la impresión, sino un simple esquema, aunque bastante realista. Tomemos un listado simple de una base de datos: la mayor parte de una página estará ocupada con las líneas procedente de los registros de la tabla. Pues bien, todas estas líneas tienen la misma función y formato y, en la terminología de QuickReport se dice que proceden de una misma banda: la banda de *detalle*s. En ese mismo listado se pueden identificar otras bandas: una correspondiente a la cabecera de páginas, la de pie de páginas, etc. Son estas bandas las que se editan y configuran en QuickReport. Durante la edición, la banda

de detalles no se repite, como sucederá durante la impresión. Pero en cualquier momento podemos ver el aspecto final del informe mediante el comando *Previe*w.

La otra característica singular es que el proceso de diseño tiene lugar dentro de C++ Builder, utilizando las propias herramientas de diseño del Entorno de Desarrollo. Los componentes de QuickReport se colocan en un formulario, aunque este formulario solamente sirve como contenedor, y nunca es mostrado al usuario de la aplicación.

Como consecuencia, todo el motor de impresión reside dentro del mismo espacio de la aplicación; no es un programa externo con carga independiente. Con esto evitamos las demoras relacionadas con la carga en memoria de un programa de impresión externo. Por supuesto, en C++ Builder 3 y 4 el código del motor puede utilizase desde un paquete; *qrpt30.dpl* ó *qrpt40.bpl*, según la versión.

Otra ventaja de QuickReport es que los datos a imprimir se extraen directamente de conjuntos de datos de C++ Builder. Una tabla que se está visualizando en una ventana de exploración, sobre la cual hemos aplicado filtros y criterios de ordenación, puede también utilizarse de forma directa para la impresión de un informe. En el listado solamente aparecerán las filas aceptadas por el filtro, en el orden especificado para la tabla. El hecho de que los datos salgan directamente de la aplicación implica que no son necesarias conexiones adicionales durante la impresión del informe. Y esto significa muchas veces, en dependencia del servidor, ahorrar en el número de licencias.

Claro está, también existen inconvenientes para este estilo de creación de informes. Ya hemos mencionado el primero: no hay una retroalimentación inmediata de las acciones de edición. El segundo inconveniente es más sutil. Con un sistema de informes independiente se pueden incluir *a posteriori* informes adicionales, que puede diseñar el propio usuario de la aplicación. Esto no puede realizarse directamente con QuickReport, a no ser que montemos un sofisticado mecanismo basado en DLLs o en Automatización OLE o un formato de intercambio diseñado desde cero.

- **Plantillas y expertos para QuickReport**

QuickReport trae plantillas de formularios para acelerar la creación de informes, y las podemos encontrar en la página *Forms* del Depósito de Objetos. Las plantillas son tres: una para listados de una sola tabla, una para informes *master/detail*, y otra para la impresión de etiquetas. No entraremos en detalles acerca del trabajo con estas plantillas pues, a partir de la explicación que se hará de los componentes de impresión, puede deducirse fácilmente qué se puede hacer con ellas.

C++ Builder también ofrece un experto para la generación de listados simples, en la página Business del Depósito. Cuando ejecutamos el experto, en la primera página debemos indicar el tipo de informe que queremos generar. En la versión de QuickReport que viene con C++ Builder, solamente tenemos una posibilidad: List Report, es decir, un listado simple. En la siguiente pantalla, debemos seleccionar la tabla cuyos datos vamos a imprimir, ya sea mediante un alias o un directorio, si se trata de una base de datos local. Además, debemos elegir qué campos de la tabla seleccionada queremos mostrar.

El resultado es un formulario con una tabla y con los componentes necesarios de QuickReport. Si quiere visualizar el informe generado, pulse el botón derecho del ratón sobre el componente TQuickRep y ejecute el comando Preview.

El generador de informes de la versión anterior fallaba constantemente. La secuencia de pantallas era ligeramente diferente a la del experto de C++ Builder 4, y la tabla que creaba la dejaba cerrada, por lo que el comando Preview no funcionaba inmediatamente.

- **El corazón de un informe**

Ahora estudiaremos cómo montar manualmente los componentes necesarios para un informe. Para definir un informe, traemos a un formulario vacío un componente TQuickRep, que ocupará un área dentro del formulario en representación de una página del listado. Luego hay que asignar la propiedad fundamental e imprescindible, que indica de qué tabla principal se extraen los datos que se van a imprimir: DataSet. Esta propiedad, como su nombre indica apunta a un conjunto de datos, no a una fuente de datos. Si se trata de un informe master/detail, la tabla que se asigna en DataSet es la maestra.

La tarea principal de un componente TQuickRep es la de iniciar el proceso de impresión, cuando se le aplica uno de los métodos Print ó Preview. También podemos utilizar el método PrintBackground, que realiza la impresión en un proceso en segundo plano. Por ejemplo, el informe generado en la sección anterior se puede imprimir en respuesta a un comando de menú de la ventana principal utilizando el siguiente código:

```
void __fastcall TForm1::Imprimir1Click(TObject *Sender)
{
Form2->QuickRep1->Print();
}
```

La propiedad *ReportTitle* del informe se utiliza como título de la ventana de previsualización predefinida.

Para obtener la vista preliminar de un informe en tiempo de diseño, utilice el comando *Preview* del menú local del componente. Tenga en cuenta que en tiempo de diseño no se ejecutan los métodos asociados a eventos, así que para ver cualquier opción que haya implementado por código, tendrá ejecutar su aplicación.

La configuración de un informe comienza normalmente por definir las características de la página, que se almacenan dentro de la propiedad *Page*. Esta es una clase con las siguientes propiedades anidadas:

Propiedad	**Significado**
PaperSize, Length, Width	Tamaño del papel
Orientation	Orientación de la página
LeftMargin, RightMargin	Ancho de los márgenes horizontales
TopMargin, BottomMargin	Ancho de los márgenes verticales
Ruler	Mostrar regla en tiempo de diseño

Columns	Número de columnas
ColumnSpace	Espacio entre columnas

Sin embargo, es más cómodo realizar una doble pulsación sobre el componente *TQuickRe*p, para cambiar las propiedades anteriores mediante el cuadro de edición.

La configuración del tamaño de papel es una de las mayores frustraciones de los programadores de QuickReport, aunque la culpa no es achacable por completo a este producto. En primer lugar, no todos los tamaños de papel son aceptados por todas las impresoras. En segundo lugar, el controlador para Windows de la mayoría de las impresoras no permite definir tamaños personalizados de papel.

Como puede verse en el editor del componente, podemos asignar un tipo de letra por omisión, global a todo el informe. Esta corresponde a la propiedad *Font* del componente *TQuickRe*p, y por omisión se emplea la Arial de 10 puntos, que es bastante legible. En el mismo recuadro, a la derecha, se establece en qué unidades se indican las medidas (propiedad *Unit*s); inicialmente se utilizan milímetros. En el recuadro siguiente se muestran las subpropiedades de la propiedad *Fram*e, que sirve para trazar un recuadro alrededor de la página. Pueden seleccionarse las líneas que se van a dibujar, el color, el ancho y su estilo.

- **Las bandas**

Una *banda* es un componente sobre el cual se colocan los componentes "imprimibles"; en este sentido, una banda actúa como un panel. Sobre las bandas se colocan los *componentes de impresió*n, en la posición aproximada en la que queremos que aparezcan impresos. Para ayudarnos en esta tarea, el componente *TQuickRep* muestra un par de reglas en los bordes de la página, que se muestran y ocultan mediante el atributo *Ruler* de la propiedad *Pag*e.

La propiedad más importante de una banda es *BandTyp*e, y estos son sus posibles valores:

Tipo de banda	**Objetivo**
rbTitle	Se imprime una sola vez, al principio del informe
rbSummary	Se imprime una sola vez, al terminar el informe
rbPageHeader	Se imprime en cada página, en la cabecera
rbPageFooter	Se imprime en cada página, al final de la misma
rbColumnHeader	Si la página se divide en columnas, al comienzo de cada una
rbDetail	Se imprime para cada registro de la tabla principal
rbSubDetail	Se imprime para cada registro de una tabla dependiente
rbGroupHeader	Se imprime cuando se detecta un cambio de grupo
rbGroupFooter	Se imprime cuando termina la impresión de un grupo
rbChild	Banda hija: se imprime siempre después de su banda madre
rbOverlay	Se sobreimprime sobre cada página

El orden de impresión de los diferentes tipos de bandas, para un listado simple, es el siguiente:

*rbPageHeade*r: En todas las páginas

*rbTitl*e: En la primera página

*rbColumnHeade*r: En cada columna, si las hay

*rbDetai*l: Una banda por cada fila de la tabla

*rbSummar*y: Al final del informe

*rbPageFoote*r: Al final de cada página

Las bandas de tipo *rbChild* se imprimen siempre a continuación de la banda madre, pero podemos ejercer más control sobre ellas mediante eventos. En la siguiente sección veremos un ejemplo.

Las bandas pueden añadirse manualmente al informe. Traemos un componente *TQRBand* desde la Paleta de Componentes, e indicamos el tipo de banda en su propiedad *BandTyp*e. Sin embargo, es más fácil indicar las bandas que necesitamos en el Editor de Componente de *TQuickRep*, en el panel inferior, o mediante la propiedad *Bands* del componente. Tenga en cuenta que los componentes *TQRGroup* y *TQRSub-Detail*, que estudiaremos más adelante, son componentes visuales y traen incorporadas sus respectivas bandas. En la primera versión de QuickReport, estos componentes venían separados de sus bandas.

La propiedad *Options* del componente *TQuickRep* permite omitir la impresión de la cabecera en la primera página del listado, y la del pie de página en la última. Esto también puede especificarse en el Editor del componente, en el panel inferior, mediante dos casillas de verificación. De este modo, si definimos una banda de título *(rbTitl*e) en el informe, se logra el efecto de tener una cabecera de página diferente para la primera página y para las restantes. Sin embargo, las bandas de resumen *(rbSummar*y) se imprimen por omisión justo a continuación de la última banda de detalles. Si queremos que aparezca como si fuera un pie de página, debemos asignar *True* a su propiedad *AlignToBottom*.

· Componentes de impresión

Una vez que tenemos bandas, podemos colocar componentes de impresión sobre las mismas. Los componentes de impresión de QuickReport son:

Componente	**Imprime...**
TQRLabel	Un texto fijo de una línea
TQRMemo	Un texto fijo con varias líneas
TQRRichEdit	Un texto fijo en formato RTF
TQRImage	Una imagen fija, en uno de los formatos de Delphi
TQRShape	Una figura geométrica simple

TQRSysData	Fecha actual, número de página, número de registro, etc.
TQRDBText	Un texto extraído de una columna
TQRDBRichEdit	Un texto RTF extraído de una columna
TQRExpr	Una expresión que puede referirse a columnas
TQRDBImage	Una imagen extraída de una columna

Todos estos objetos, aunque pertenecen a clases diferentes, tienen rasgos comunes. La propiedad *AutoSiz*e, por ejemplo, controla el área de impresión en la dimensión horizontal. Normalmente, esta propiedad debe valer *Fals*e, para evitar que se superpongan entre sí los diferentes componentes de impresión. En tal caso, es necesario agrandar el componente hasta su área máxima de impresión. La posición del texto a imprimir dentro del área asignada se controla mediante la propiedad *Alignment*: a la derecha, al centro o a la izquierda. Ahora bien, el significado de esta propiedad puede verse afectado por el valor de *AlignToBand*. Cuando *AlignToBand* es *Tru*e, *Alignment* indica en qué posición horizontal, con respecto a la banda, se va a imprimir el componente.

Da lo mismo, entonces, la posición en que situemos el componente. Todos los componentes de impresión tienen también la propiedad *AutoStretch* que, cuando es *Tru*e, permite que la impresión del componente continúe en varias líneas cuando no cabe en el área de impresión vertical definida.

El componente más frecuentemente empleado es *TQRDBTex*t, que imprime el contenido de un campo de un conjunto de datos. Hay que configurar sus propiedades *DataSet* y *DataField*. Tiene la ventaja de que aprovecha automáticamente el formato de visualización del campo asociado, algo que no hace el componente alternativo *TQRExp*r, que veremos a continuación. Este componente, además, es capaz de mostrar el contenido de un campo memo, sin mayores complicaciones.

2.6. Análisis gráfico

Entre los componentes introducidos por la versión 3 de la VCL, destacan los que se encuentran la página *Decision Cub*e, que nos permiten calcular y mostrar gráficos y rejillas de decisión. Con los componentes de esta página podemos visualizar en pantalla informes al estilo de *tablas cruzadas (crosstab*s). En este tipo de informes se muestran estadísticas acerca de ciertos datos: totales, cantidades y promedios, con la particularidad de que el criterio de agregación puede ser multidimensional. Por ejemplo, nos interesa saber los totales de ventas por delegaciones, pero a la vez queremos subdividir estos totales por meses o trimestres. Además, queremos ocultar o expandir dinámicamente las dimensiones de análisis, que los resultados se muestren en rejillas, o en gráficos de barras. Todo esto es tarea de Decision Cube. Y ya que estamos hablando de gráficos, explicaremos también como aprovechar los componentes *TChart* y *TDBChar*t, que permiten mostrar series de datos generadas mediante código o provenientes de tablas y consultas.

Para ilustrar el empleo de Decision Cube y del componente *TDBChart* utilizaremos una base de datos que tratará sobre la gestión de libretas de ahorro. Para mayor facilidad.

· Gráficos

Mostraremos cómo puede utilizarse el componente *Tchart*. Este componente se encuentra en la página *Additional* de la Paleta de Componentes. Pero si exploramos un poco la Paleta, encontraremos también los componentes *TDBChart* y *TQRDBChar*t. En principio, *TChart* y *TDBChart* tienen casi la misma funcionalidad, pero el segundo puede ser llenado a partir de un conjunto de datos, especificando determinados campos del mismo. El conjunto de datos puede ser indistintamente una tabla, una consulta o el componente derivado de *TDataSet* que se le ocurra. Por su parte, *TQRDBChart* puede incluirse dentro de un informe generado con QuickReport.

Un gráfico debe mostrar valores de una colección de datos simples o puntuales. La colección de datos de un gráfico de tarta, por ejemplo, debe contener pares del tipo: (etiqueta, proporción)

La proporción determina el ángulo que ocupa cada trozo de la tarta, y la etiqueta sirve ... pues para eso ... para etiquetar el trozo. En cambio, un gráfico lineal puede contener tripletas en vez de pares: (etiqueta, valor X, valor Y)

Ahora se ha incluido un valor X para calcular distancias en el eje horizontal. Si un gráfico lineal contiene tres puntos, uno para enero, uno para febrero y otro para diciembre, se espera que los puntos de enero y febrero estén más alejados del punto correspondiente a diciembre.

A estas colecciones de puntos se les denomina *serie*s. Un componente *TChart* contiene una o más series, que son objetos derivados de la clase *TSerie*s. Esto quiere decir que puede mostrar simultáneamente más de un gráfico, en el sentido usual de la palabra, lo cual puede valer para realizar comparaciones. He dicho antes que los tipos de series concretas se derivan de la clase abstracta *TSerie*s, y es que la estructura de una serie correspondiente a un gráfico lineal es diferente a la de una serie que contenga los datos de un gráfico de tarta.

Las series deben ser creadas por el programador, casi siempre en tiempo de diseño. Tras pulsar dos veces el botón del ratón sobre el componente, aparecerá un cuadro de diálogo. A continuación pulse el botón *Add*, para que aparezca la galería de estilos disponible.

Podemos optar por series simples, o por funciones, que se basan en datos de otras series para crear series calculadas. Escogeremos un gráfico lineal, por simplicidad y conveniencia.

Cuando creamos una serie para un gráfico, estamos creando explícitamente un componente con nombre y una variable asociada dentro del formulario. Por omisión, la serie creada se llamará *Series*1. Repita dos veces más la operación anterior para tener también las series *Series2* y *Series*3. Las tres variables apuntan a objetos de la clase *TLineSerie*s, que contiene el siguiente método:

```
int __fastcall Add(const double YValue, const AnsiString ALabel,
TColor AColor);
```

Utilizaremos este método, en vez de *AddX*Y, porque nuestros puntos irán espaciados uniformemente. En el tercer parámetro podemos utilizar la constante especial de color *clTeeColor*, para que el componente decida qué color utilizar.

En el hilo de adquisición de datos sito en el módulo *Surmain*, hemos ido almacenando las diversas variables de cuatro inversores en 16 vectores de 100 elementos cada uno. Son estos vectores los que dibujamos al pulsar el botón "GRÁFICOS" de la ventana principal mediante sentencias del tipo:

```
Series7->Add(tp2[i],i,clTeeColor);
```

3. LA COMUNICACIÓN POR EL PUERTO SERIE

El protocolo de comunicación que se ha usado ha sido uno ya implementado previamente con éxito por miembros del departamento, y se basa en los siguientes elementos:

-Dirección de destino

-Número de instrucción

-Datos

-Dirección de destino

-Checksum

-Fin de trama

Cada uno de ellos con sus *nibbles* identificativos.

Este protocolo se ha introducido en un hilo independiente, de forma que transcurriera en paralelo con la ejecución normal del programa. Se ha tenido cuidado de que toda interacción con la base de datos en este hilo fuera a través de procedimientos almacenados. A continuación se presenta el diagrama de flujo que sintetiza el código del módulo *Surmain* donde dicho hilo se desarrolla:

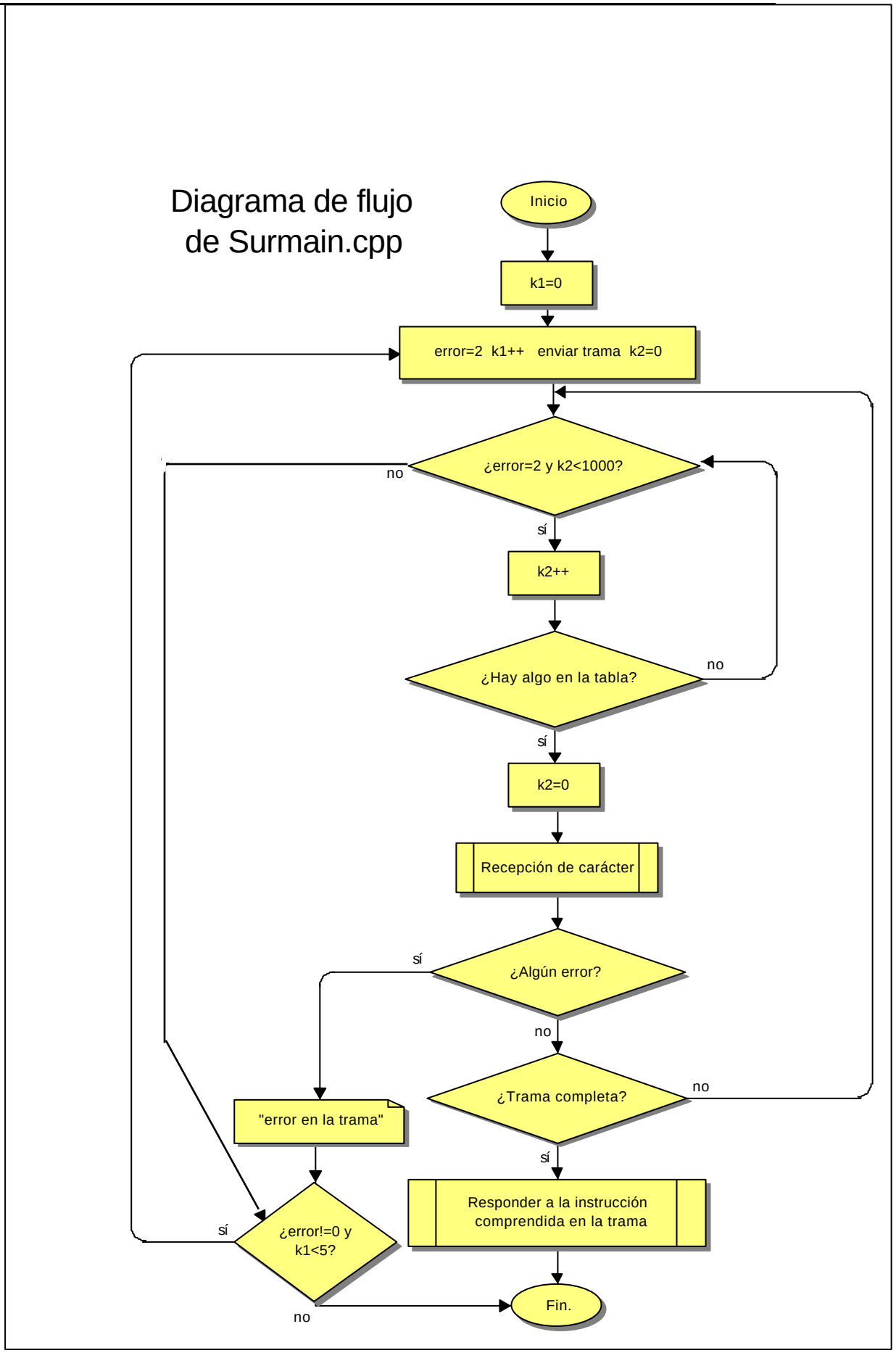


Diagrama de flujo de Surmain.cpp
Inicio
k1=0
error=2 k1++ enviar trama k2=0
¿error=2 y k2<1000?
no
sí
k2++
¿Hay algo en la tabla?
no
sí
k2=0
Recepción de carácter
¿Algún error?
sí
no
¿Trama completa?
no
sí
"error en la trama"
Responder a la instrucción comprendida en la trama
¿error!=0 y k1<5?
sí
no
Fin.

3. Manual de Usuario

1. INSTALACIÓN DEL SERVIDOR

Para la instalación del servidor MPF será necesario tener el BDE y configurar un alias para la base de datos INVERSOR.db. Una vez se tenga esto, se procederá a salvar el contenido del directorio SERVIDOR MPF en el disco duro, abrir el proyecto InvServidor.bpr con el C++Builder y seguir los siguientes pasos:

SELECCIONAR new>>multitier>>remote data module

ESCRIBIR:

MastSQL

Apartment

Servidor Midas

SALVAR LA APLICACIÓN

CERRARLA

COPIAR LOS ARCHIVOS MastSQLImpl.* EN EL MISMO DIRECTORIO QUE EL bpr.

EJECUTAR EL PROGRAMA.

2. INSTALACIÓN DEL CLIENTE

Guardar la carpeta CLIENTE MPF en el disco duro.

Abrir el proyecto.

Entrar en DataModule1 y escribir el *Adress* del servidor al que se quiera acceder en el componente SocketConnection1.

Ejecutar el programa.

3. MANEJO DEL SERVIDOR

Para activar la adquisición de datos por el puerto serie no hay más que pulsar el botón "COMUNICACIÓN" y seleccionar *informativa*. Acto seguido, el PC enviará al *Data Logger* las listas de inversores encendidos y sus correspondientes límites de potencia. Inmediatamente pasará al modo de comunicación *activa*, en el cual recibirá las variables y las irá colocando en sus posiciones dentro de la base de datos.

Para variar el estado de un inversor no habrá más que pulsar con el ratón en la casilla correspondiente de la rejilla inferior y seleccionar de la lista desplegable. De forma similar se podrá introducir el límite de potencia deseado. Posteriormente a la acción deberá pulsarse el botón "ACTUALIZACIÓN DE LA BASE DE DATOS".

Si se quiere ordenar alguna de las tablas por una de sus columnas en particular no habrá más que pinchar en su título.

Si en algún momento resulta de interés guardar las variables no hay más que pulsar en "SALVAR", con lo que se generará un archivo de MATLAB que posteriormente permitiría tratar estos datos. Simultáneamente se produce un informe con el estado instantáneo de la planta, que podrá consultarse en el futuro mediante la opción "CARGAR HISTÓRICO" o sacarse en papel pinchando en "IMPRIMIR".

De forma inmediata, podemos ver una representación gráfica de la evolución de los cuatro primeros inversores pulsando en el botón "GRÁFICAS". Para salir de esta pantalla o de la de presentación o de bienvenida basta con pulsar un botón.

4. MANEJO DEL CLIENTE

La aplicación cliente consta de un subconjunto de las opciones del servidor. Todas son equivalentes a sus análogas, salvo la denominada "ACTUALIZACIÓN DE LA BASE DE DATOS", con la que ahora se realizan las actualizaciones hechas de forma remota en la tabla de estados.

APÉNDICE: Las fichas y el código más trascendental.

BASE DE DATOS MPF

Inversor.db

```
CREATE DATABASE "C:\inversor\INVERSOR.DB" PAGE_SIZE 1024
;
/* Table: TABLA1, Owner: INVERSOR */
CREATE TABLE TABLA1 (POTENCIA FLOAT,
        TENSION_DE_PANEL FLOAT,
        INTENSIDAD_DE_PANEL FLOAT,
        TENSION_DE_RED FLOAT,
        INTENSIDAD_DE_BOBINA FLOAT,
        FECHA_Y_HORA DATE DEFAULT "NOW",
        NUMERO SMALLINT);
/* Table: TABLA2, Owner: INVERSOR */
CREATE TABLE TABLA2 (NO_INV SMALLINT NOT NULL,
        LIMITE_DE_POTENCIA FLOAT,
        ESTADO VARCHAR(12) default "no"
 NOT NULL,
PRIMARY KEY (NO_INV));
/*  Index definitions for all user tables */
CREATE INDEX POTENCIA ON TABLA2(LIMITE_DE_POTENCIA);
ALTER TABLE TABLA1 ADD FOREIGN KEY (NUMERO) REFERENCES TABLA2(NO_INV);
ALTER TABLE TABLA2 ADD
        check(estado in ("on","off","no"));
COMMIT WORK;
SET AUTODDL OFF;
SET TERM ^ ;
```

```
/* Stored procedures */

CREATE PROCEDURE ACTVARINV AS BEGIN EXIT; END ^


ALTER PROCEDURE ACTVARINV (NUM SMALLINT,

TENPAN FLOAT,

INTPAN FLOAT,

TENRED FLOAT,

INTBOB FLOAT)

AS

begin

UPDATE tabla1

SET tension_de_panel = :tenpan, intensidad_de_panel = :intpan,

tension_de_red = :tenred, intensidad_de_bobina= :intbob, potencia=
(:intpan)*(:tenpan),

fecha_y_hora = 'now'

WHERE numero= :num;

end ^

SET TERM ; ^

COMMIT WORK ;

SET AUTODDL ON;

SET TERM ^ ;

/* Triggers only will work for SQL triggers */

CREATE TRIGGER ADDINVON FOR TABLA2

ACTIVE AFTER UPDATE POSITION 0

as

declare variable auxiliar integer;

begin

delete from tabla1

where 1=1;

for select no_inv from tabla2 where estado="on" into :auxiliar do

insert into tabla1(numero)

values(:auxiliar);

end
```

```
 ^
COMMIT WORK ^
SET TERM ; ^

/* Grant role for this database */
/* Role: JUANANTONIO, Owner: INVERSOR */
CREATE ROLE JUANANTONIO;
CREATE ROLE LUIS;
/* Grant permissions for this database */
GRANT DELETE, INSERT, SELECT, UPDATE, REFERENCES ON TABLA1 TO DANIEL;
GRANT DELETE, INSERT, SELECT, UPDATE, REFERENCES ON TABLA1 TO JUANANTONIO;
GRANT DELETE, INSERT, SELECT, UPDATE, REFERENCES ON TABLA1 TO LUIS;
GRANT SELECT ON TABLA1 TO PUBLIC;
GRANT DELETE, INSERT, SELECT, UPDATE, REFERENCES ON TABLA2 TO DANIEL;
GRANT DELETE, INSERT, SELECT, UPDATE, REFERENCES ON TABLA2 TO JUANANTONIO;
GRANT DELETE, INSERT, SELECT, UPDATE, REFERENCES ON TABLA2 TO LUIS;
GRANT SELECT ON TABLA2 TO PUBLIC;
```

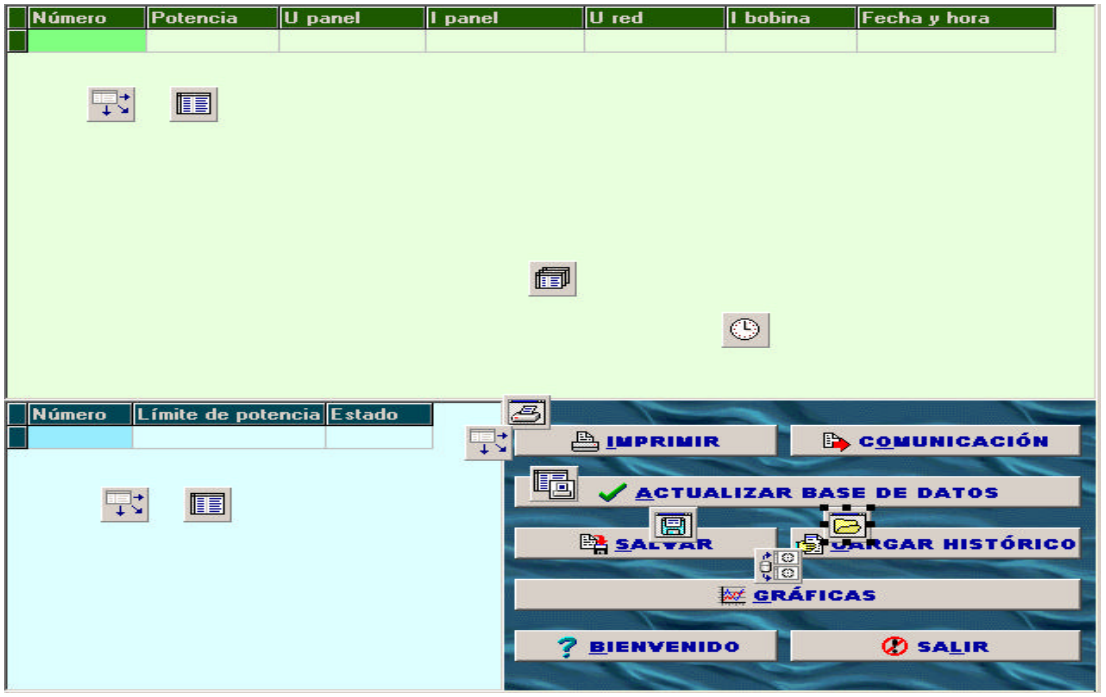
SERVIDOR MPF

InvServidor.cpp

```
#include <vcl.h>
#pragma hdrstop
USERES("invServidor.res");
USEFORM("Unidadppal.cpp", Formppal);
USEFORM("Unidad1.cpp", Form1);
USEFORM("Unidad2.cpp", Form2);
USEFORM("Unidadayuda.cpp", Formayuda);
USEFORM("Unidadopciones.cpp", Formopciones);
USEFORM("Unidadgraficos.cpp", Formgraficos);
USEUNIT("Surmain.cpp");
```

```
USEFORM("SesionProtocolo.cpp", DataModule1); /* TDataModule: File Type */
USEFORM("Unidadpresentacion.cpp", Formpresentacion);
//---------------------------------------------------------------------------
WINAPI WinMain(HINSTANCE, HINSTANCE, LPSTR, int)
{
        try
        {
                Application->Initialize();
                Application->Title = "Servidor MPF";
                Application->CreateForm(__classid(TDataModule1), &DataModule1);
                Application->CreateForm(__classid(TFormppal), &Formppal);
                Application->CreateForm(__classid(TForm1), &Form1);
                Application->CreateForm(__classid(TForm2), &Form2);
                Application->CreateForm(__classid(TFormayuda), &Formayuda);
                Application->CreateForm(__classid(TFormopciones), &Formopciones);
                Application->CreateForm(__classid(TFormgraficos), &Formgraficos);
                Application->Run();
        }
        catch (Exception &exception)
        {
                Application->ShowException(&exception);
        }
        return 0;
}
```

Formppal



Unidadppal.cpp

```
#include <vcl.h>
#include <conio.h>
#pragma hdrstop
#include "Printers.hpp"
#include "Unidadppal.h"
#include "Unidadpresentacion.h"
#include "Unidad1.h"
#include "Unidad2.h"
#include "Unidadopciones.h"
#include "Unidadgraficos.h"
#include "Unidadayuda.h"
#include "Surmain.h"
```

```
#include "SesionProtocolo.h"
//---------------------------------------------------------------------------
#pragma package(smart_init)
#pragma resource "*.dfm"


TFormppal *Formppal;
HANDLE hComm=NULL;
TRead *ReadThread;
COMMTIMEOUTS ctmoNew = {0}, ctmoOld;
extern unsigned char *tabla;
unsigned char invON[32];
unsigned char pot[32];
unsigned char coninvON;
unsigned char conpot;
unsigned char confilas=0;
extern unsigned char inst=0;
bool lanzado=false;
extern unsigned char ip1[100];
extern unsigned char tp1[100];
extern unsigned char ib1[100];
extern unsigned char tr1[100];
extern unsigned char ip2[100];
extern unsigned char tp2[100];
extern unsigned char ib2[100];
extern unsigned char tr2[100];
extern unsigned char ip3[100];
extern unsigned char tp3[100];
extern unsigned char ib3[100];
extern unsigned char tr3[100];
extern unsigned char ip4[100];
extern unsigned char tp4[100];
extern unsigned char ib4[100];
extern unsigned char tr4[100];
```

```
short recibido=0;
//---------------------------------------------------------------------------
__fastcall TFormppal::TFormppal(TComponent* Owner)
        : TForm(Owner)
{
}
//---------------------------------------------------------------------------
void __fastcall TFormppal::FormCreate(TObject *Sender)
{//PROTOCOLO
          DCB dcbCommPort;
          AnsiString aux;
          hComm = CreateFile("COM1",
                      GENERIC_READ | GENERIC_WRITE,
                      0,
                      0,
                      OPEN_EXISTING,
                      0,
                      0);
/*Si no puede abrirse el puerto, denunciamos el error, y
detenemos la aplicación.*/
         if(hComm == INVALID_HANDLE_VALUE)
  {
    Application->MessageBoxA("No se pudo abrir el puerto COM1","Error",0);
    Application->Terminate();
  }
  else{
/*Configurar los tiempos del puerto COMM*/
  GetCommTimeouts(hComm,&ctmoOld);
  ctmoNew.ReadTotalTimeoutConstant = 100;
  ctmoNew.ReadTotalTimeoutMultiplier = 0;
  ctmoNew.WriteTotalTimeoutMultiplier = 0;
  ctmoNew.WriteTotalTimeoutConstant = 0;
  SetCommTimeouts(hComm, &ctmoNew);
```

```
/*Configurar el número de baudios, la paridad, el tamaño de los datos
y los bits de stop. Hay otras formas de realizarlo, pero esta es la
más sencilla. */
  dcbCommPort.DCBlength = sizeof(DCB);
  GetCommState(hComm, &dcbCommPort);
  BuildCommDCB("38400,N,8,1", &dcbCommPort);
//Deshabilitamos el uso de RTS para la comunicación de Windows vía Hardware
  dcbCommPort.fRtsControl=RTS_CONTROL_DISABLE;
//Cargamos la configuración del puerto serie
  SetCommState(hComm, &dcbCommPort);
//Activamos REQUEST TO SEND para nuestra comunicación
  if(EscapeCommFunction(hComm,SETRTS)==false)
   ShowMessage("No se pudo activar el RTS");
  }
/*Activamos el proceso en paralelo. El argumento "false" simplemente
significa que el resto permanece ejecutándose, no suspendido*/
 ReadThread = new TRead(true);
//Sección crítica
 csection = new TCriticalSection;
//PRESENTACION
/*Creamos dinámicamente la ficha de la presentación para
poder liberar memoria posteriormente*/
TFormpresentacion * Formpresentacion;
TMediaPlayer * MediaPlayer2;
Formpresentacion = new TFormpresentacion(this);

/*Descomentar y elegir archivo sonoro si se quiere amenizar la presentación
MediaPlayer2 = new TMediaPlayer (Formpresentacion);
MediaPlayer2->Parent=Formpresentacion;
MediaPlayer2->DeviceType=dtAutoSelect;
MediaPlayer2->FileName="C:\\AINV\\Beautiful Way.asf";
MediaPlayer2->AutoEnable=true;
MediaPlayer2->Visible=false;
```

```
MediaPlayer2->Open();
MediaPlayer2->Play();
*/
Formpresentacion->ShowModal();
Formppal->Position = poScreenCenter;
Formppal->Visible=true;
//Activamos las tablas para actualizar las rejillas
Table2->Active=true;
Table1->Active=true;
Timer1->Enabled=true;
   DataModule1->Session1->Active=true;
   DataModule1->Database1->Open();
}
//---------------------------------------------------------------------------
/*Elección del índice de la tabla mediante la pulsación
del título de una de sus columnas*/
void __fastcall TFormppal::DBGrid1TitleClick(TColumn *Column)
{
                    try
        {
            Table1->IndexFieldNames = Column->FieldName;
                 }
                        catch(Exception&)
                 {
                 }
}
//---------------------------------------------------------------------------
void __fastcall TFormppal::DBGrid2TitleClick(TColumn *Column)
{
        try
        {
            Table2->IndexFieldNames = Column->FieldName;
                 }
```

```
                        catch(Exception&)
                    {
                    }
}
//---------------------------------------------------------------------------
/*Visualización de la ayuda*/
void __fastcall TFormppal::BitBtn1Click(TObject *Sender)
{
Formayuda->Visible=true;
}
//---------------------------------------------------------------------------
/*Actualización de la Base de Datos*/
void __fastcall TFormppal::BitBtn2Click(TObject *Sender)
{
/*Se pretende elegir los registros de la tabla 1 mediante
la columna Estado de la tabla 2*/
if(Table2->State == dsEdit)
   {
/*Pero si la tabla 2 se encuentra en estado de edición
en este momento, también debe actualizarse*/
Table2->Post();
   }
Table2->Close();
Table2->Open();
//}
Table1->Close();
Table1->Open();
}
//---------------------------------------------------------------------------
/*Salvar los datos en la presente Tabla 1*/
void __fastcall TFormppal::BitBtn3Click(TObject *Sender)
{
/*Para guardar los históricos nos serviremos de la tecnólogía MIDAS
```

```
creada por Borland para el desarrollo de aplicaciones multicapa.
Esto nos permite usar un "conjunto de datos del cliente" con el que
realizar una copia de los datos que en ese momento alberguemos en la
tabla.*/
        /*Primeramente debemos actualizar este conjunto, y para ello
proceder a activarlo y desactivarlo*/
        ClientDataSet1->Active = false;
        ClientDataSet1->Active = true;
        if(SaveDialog1->Execute()){
        /*Una vez actualizado con la información relevante, lo
utilizamos como trampolín para salvarla a un archivo*/
ClientDataSet1->SaveToFile(SaveDialog1->FileName+".db",dfBinary);
FILE *out;
if ((out = fopen((SaveDialog1->FileName+".m").c_str(), "wt"))
       == NULL)
   {
      fprintf(stderr, "Cannot open output file.\n");
   }
   fprintf(out,"ip1=[\n%f\n",ip1[0]);
   for(int i=1;i<100;i++)
       fprintf(out,"%f\n",ip1[i]);
   fprintf(out,"];\n");

   fprintf(out,"ip2=[\n%f\n",ip2[0]);
   for(int i=1;i<100;i++)
       fprintf(out,"%f\n",ip2[i]);
   fprintf(out,"];\n");

   fprintf(out,"ip3=[\n%f\n",ip3[0]);
   for(int i=1;i<100;i++)
       fprintf(out,"%f\n",ip3[i]);
   fprintf(out,"];\n");
```

```
fprintf(out,"ip4=[\n%f\n",ip4[0]);
for(int i=1;i<100;i++)
    fprintf(out,"%f\n",ip4[i]);
fprintf(out,"];\n");

fprintf(out,"tp1=[\n%f\n",tp1[0]);
for(int i=1;i<100;i++)
    fprintf(out,"%f\n",tp1[i]);
fprintf(out,"];\n");

fprintf(out,"tp2=[\n%f\n",tp2[0]);
for(int i=1;i<100;i++)
    fprintf(out,"%f\n",tp2[i]);
fprintf(out,"];\n");

fprintf(out,"tp3=[\n%f\n",tp3[0]);
for(int i=1;i<100;i++)
    fprintf(out,"%f\n",tp3[i]);
fprintf(out,"];\n");

fprintf(out,"tp4=[\n%f\n",tp4[0]);
for(int i=1;i<100;i++)
    fprintf(out,"%f\n",tp4[i]);
fprintf(out,"];\n");

fprintf(out,"ib1=[\n%f\n",ib1[0]);
for(int i=1;i<100;i++)
    fprintf(out,"%f\n",ib1[i]);
fprintf(out,"];\n");

fprintf(out,"ib2=[\n%f\n",ib2[0]);
for(int i=1;i<100;i++)
    fprintf(out,"%f\n",ib2[i]);
```

```
    fprintf(out,"];\n");


    fprintf(out,"ib3=[\n%f\n",ib3[0]);
    for(int i=1;i<100;i++)
        fprintf(out,"%f\n",ib3[i]);
    fprintf(out,"];\n");


    fprintf(out,"ib4=[\n%f\n",ib4[0]);
    for(int i=1;i<100;i++)
        fprintf(out,"%f\n",ib4[i]);
    fprintf(out,"];\n");


    fprintf(out,"tr1=[\n%f\n",tr1[0]);
    for(int i=1;i<100;i++)
        fprintf(out,"%f\n",tr1[i]);
    fprintf(out,"];\n");


    fprintf(out,"tr2=[\n%f\n",tr2[0]);
    for(int i=1;i<100;i++)
        fprintf(out,"%f\n",tr2[i]);
    fprintf(out,"];\n");


    fprintf(out,"tr3=[\n%f\n",tr3[0]);
    for(int i=1;i<100;i++)
        fprintf(out,"%f\n",tr3[i]);
    fprintf(out,"];\n");


    fprintf(out,"tr4=[\n%f\n",tr4[0]);
    for(int i=1;i<100;i++)
        fprintf(out,"%f\n",tr4[i]);
    fprintf(out,"];\n");
}
}
```

```
//---------------------------------------------------------------------------

/*Impresión de la tabla 1 mediante "Quick Report"*/
void __fastcall TFormppal::BitBtn4Click(TObject *Sender)
{
/*El siguiente código es necesario si tenemos varias
impresoras disponibles*/
        /*Configuración de la impresora*/
        PrintDialog1->PrintToFile = false;
       if( PrintDialog1->Execute()){
        /*Comunicar a Quick Report cuál es la impresora seleccionada*/
        Form1->QuickRep1->PrinterSettings->PrinterIndex = Printer()
        ->PrinterIndex;
        /*y el resto de parámetros de interés*/
        Form1->QuickRep1->PrinterSettings->FirstPage = PrintDialog1->
        FromPage;
        Form1->QuickRep1->PrinterSettings->LastPage = PrintDialog1->
        ToPage;
        Form1->QuickRep1->PrinterSettings->Copies = PrintDialog1->
        Copies;
        /*Dado el elevado número de columnas que se maneja conviene
imprimir de forma apaisada*/
        Form1->QuickRep1->PrinterSettings->Orientation = poLandscape;
        /*De esta forma no hemos tenido en cuenta el resto del
"PrintDialog1", por lo que es inútil pulsar el botón de propiedades
de esta ventana*/
        //En el laboratorio este comando no surte efecto
        //¿debido quizás a algún error en el entorno de desarrollo?
/*Imprimir el informe*/
        Form1->QuickRep1->Print();}
}
//---------------------------------------------------------------------------
/*Cargar histórico previamente salvado*/
```

```
   void __fastcall TFormppal::BitBtn5Click(TObject *Sender)
   {
   /*Apoyándonos en el código de la función anterior es muy sencillo
   desarrollar esta: no es necesaria la conexión de nuevos componentes*/
   if(OpenDialog1->Execute()){
   ClientDataSet1->LoadFromFile(OpenDialog1->FileName);
   //ejercicio de estética
   Form2->Caption = "Histórico " + OpenDialog1->FileName;
   Form2->Position = poScreenCenter;
   Form2->WindowState = wsMaximized;
   Form2->DBGrid1->DataSource = DataSource3;
   Form2->DBGrid1->Left = 0;
   Form2->DBGrid1->Width = Form2->ClientWidth;
   Form2->DBGrid1->Height = Form2->ClientHeight;
   Form2->Visible = true;
   }
   }
   //---------------------------------------------------------------------------
   void __fastcall TFormppal::Timer1Timer(TObject *Sender)
   {
      recibido=0;
      static short reintento=4;
      bool anton;
      Formopciones->Enabled=false;


     while((recibido==0)&&(reintento<5))
     {
        unsigned char checks=0;
       anton=true;
       while(anton)
       {
         anton=!TransmitCommChar(hComm,129);//128+dirección del esclavo(1 para el
Datalogger)
```

```
}
 anton=true;
while(anton)
{
anton=!TransmitCommChar(hComm,inst);//número de instrucción
}
checks=129+inst;
//El párrafo que viene a continuación es sólo para probar el programa
//En la versión definitiva deberá eliminarse
if(inst==1){
anton=true;
while(anton)
{
  anton=!TransmitCommChar(hComm,(2&240)>>4);
}
checks+=((2&240)>>4);
anton=true;
while(anton)
{
  anton=!TransmitCommChar(hComm,2&15);
}
checks+=(2&15);
          }
if(inst==2){
Table2->First();
coninvON=0;
    while(!Table2->Eof){
     if(Table2ESTADO->AsString=="on"){
       unsigned char aux=Table2NO_INV->AsVariant;
       anton=true;
       while(anton)
               {
        anton=!TransmitCommChar(hComm,(aux&240)>>4);
```

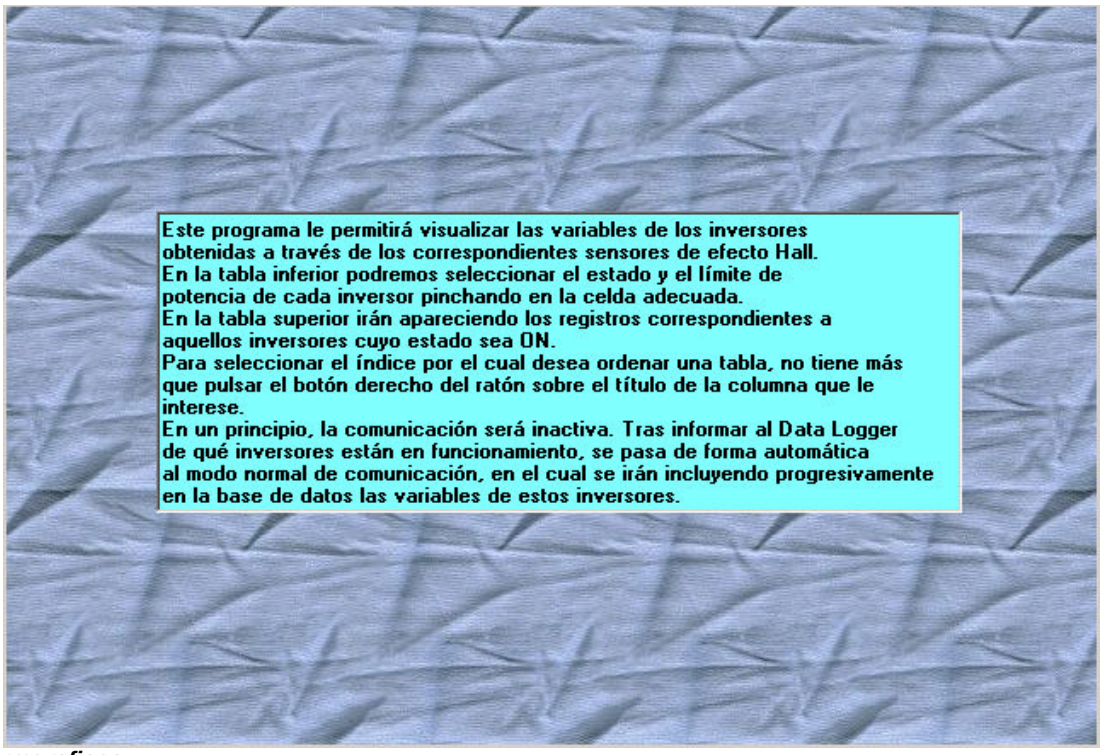
```
                    }
             checks+=(aux&240)>>4;
            anton=true;
            while(anton)
                    {
             anton=!TransmitCommChar(hComm,aux&15);
                    }
              invON[coninvON++]= aux;
              checks+=aux&15;
                                        }
      Table2->Next();
                           }
             }

if(inst==3){
  Table2->First();
  conpot=0;
  while(!Table2->Eof){
    if(Table2ESTADO->AsString=="on"){
    unsigned char aux=Table2LIMITE_DE_POTENCIA->AsVariant;
       anton=true;
       while(anton)
               {
        anton=!TransmitCommChar(hComm,(aux&240)>>4);
                }
        checks+=(aux&240)>>4;
       anton=true;
       while(anton)
               {
        anton=!TransmitCommChar(hComm,aux&15);
               }
         checks+=aux&15;
     pot[conpot++] = aux;
```

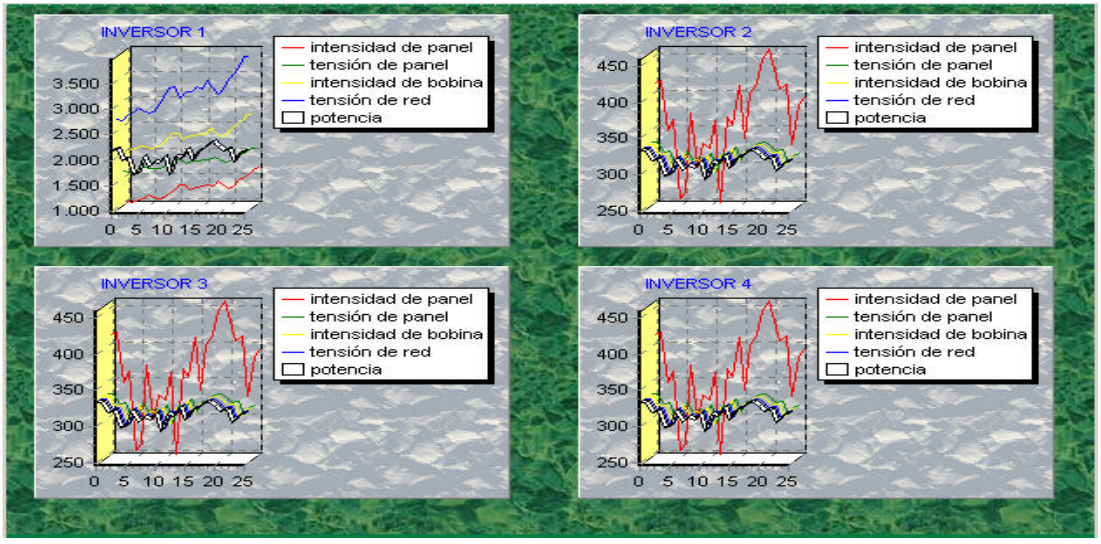
```
                                        }
          Table2->Next();
                              }
                }
anton=true;
    while(anton)
    {
     anton=!TransmitCommChar(hComm,((checks&240)>>4)|32);//nibble alto de checksum
    }
anton=true;
    while(anton)
    {
     anton=!TransmitCommChar(hComm,(checks&15)|16);//nibble  bajo de checksum
    }
    anton=true;
    while(anton)
    {
     anton=!TransmitCommChar(hComm,64);//fin de trama
    }
if(!lanzado)
ReadThread->Resume();
lanzado=true;
reintento++;
  }
Formopciones->Enabled=true;
reintento=4;
}
//---------------------------------------------------------------------------
void __fastcall TFormppal::FormClose(TObject *Sender, TCloseAction &Action)
{
//Destruir la sección crítica
    delete csection;
    if(hComm) {//Si el puerto se abrió correctamente
```

```
//Terminar el hilo de adquisición de datos
        ReadThread->Terminate();
//Esperar a que finalice el hilo
        Sleep(250);
//Desactivar el RTS
        if(EscapeCommFunction(hComm,CLRRTS)==false)
                ShowMessage("No se pudo desactivar el RTS");
//Restaurar la configuración primitiva del puerto serie
        SetCommTimeouts(hComm, &ctmoOld);
//Cerrar el puerto
        CloseHandle(hComm);
              }
}
//---------------------------------------------------------------------------
void __fastcall TFormppal::BitBtn6Click(TObject *Sender)
{
Formopciones->Visible=true;
}
void __fastcall TFormppal::BitBtn8Click(TObject *Sender)
{
Formgraficos->Visible=true;
}
/*La visualización de gráficas es la acción prioritaria*/
void __fastcall TFormppal::FormActivate(TObject *Sender)
{
BitBtn8->SetFocus();
}
//---------------------------------------------------------------------------
void __fastcall TFormppal::SALIRClick(TObject *Sender)
{
Close();
}
```

Formayuda



Formgraficos



Unidadgraficos.cpp

```
#include <vcl.h>
#pragma hdrstop
#include "Unidadppal.h"
#include "Unidadgraficos.h"
//---------------------------------------------------------------------------
#pragma package(smart_init)
#pragma resource "*.dfm"
TFormgraficos *Formgraficos;
extern unsigned char ip1[100];
extern unsigned char tp1[100];
extern unsigned char ib1[100];
extern unsigned char tr1[100];
extern unsigned char ip2[100];
extern unsigned char tp2[100];
extern unsigned char ib2[100];
extern unsigned char tr2[100];
extern unsigned char ip3[100];
extern unsigned char tp3[100];
extern unsigned char ib3[100];
extern unsigned char tr3[100];
extern unsigned char ip4[100];
extern unsigned char tp4[100];
extern unsigned char ib4[100];
extern unsigned char tr4[100];
//---------------------------------------------------------------------------
__fastcall TFormgraficos::TFormgraficos(TComponent* Owner)
   : TForm(Owner)
{
}
//---------------------------------------------------------------------------
void __fastcall TFormgraficos::FormShow(TObject *Sender)
```

```
{
Formppal->Enabled=false;
for(int i=0; i<100;i++)
   {
Series1->Add(ip1[i],i,clTeeColor);
Series2->Add(tp1[i],i,clTeeColor);
Series3->Add(ib1[i],i,clTeeColor);
Series4->Add(tr1[i],i,clTeeColor);
Series5->Add(ip1[i]*tp1[i],i,clTeeColor);
Series6->Add(ip2[i],i,clTeeColor);
Series7->Add(tp2[i],i,clTeeColor);
Series8->Add(ib2[i],i,clTeeColor);
Series9->Add(tr2[i],i,clTeeColor);
Series10->Add(ip2[i]*tp2[i],i,clTeeColor);
Series11->Add(ip3[i],i,clTeeColor);
Series12->Add(tp3[i],i,clTeeColor);
Series13->Add(ib3[i],i,clTeeColor);
Series14->Add(tr3[i],i,clTeeColor);
Series15->Add(ip3[i]*tp3[i],i,clTeeColor);
Series16->Add(ip4[i],i,clTeeColor);
Series17->Add(tp4[i],i,clTeeColor);
Series18->Add(ib4[i],i,clTeeColor);
Series19->Add(tr4[i],i,clTeeColor);
Series20->Add(ip4[i]*tp4[i],i,clTeeColor);
   }
}
//---------------------------------------------------------------------------
void __fastcall TFormgraficos::FormHide(TObject *Sender)
{
Formppal->Enabled=true;
}
//---------------------------------------------------------------------------
void __fastcall TFormgraficos::FormCreate(TObject *Sender)
```

```
{
Formgraficos->Position=poScreenCenter;
}
//---------------------------------------------------------------------------
void __fastcall TFormgraficos::FormKeyPress(TObject *Sender, char &Key)
{
Close();
}
```

Formopciones

Unidadopciones.cpp

```
#include <vcl.h>
#pragma hdrstop
#include "Unidadppal.h"
#include "Unidadopciones.h"
//---------------------------------------------------------------------------
#pragma package(smart_init)
#pragma resource "*.dfm"
TFormopciones *Formopciones;
extern unsigned char inst;
//---------------------------------------------------------------------------
__fastcall TFormopciones::TFormopciones(TComponent* Owner)
        : TForm(Owner)
{
}
//---------------------------------------------------------------------------
void __fastcall TFormopciones::BitBtn1Click(TObject *Sender)
{
```

```
      if(INACTIVA->Checked) inst=0;
      if(NORMAL->Checked) inst=1;
      if(INFORMATIVA->Checked) {
      inst=2;
      NORMAL->Enabled=true;
      /*Para pasar al modo de comunicación "normal"
      antes debemos informar al Data Logger de qué inversores están
      encendidos*/
      }
      Formppal->Enabled=true;
      Formopciones->Visible=false;
   }
   //---------------------------------------------------------------------------
   void __fastcall TFormopciones::FormShow(TObject *Sender)
   {  Formppal->Enabled=false;
    if (inst==0) INACTIVA->Checked=true;
    if (inst==1) NORMAL->Checked=true;
    if (inst==2||inst==3) INFORMATIVA->Checked=true;
   }
```

Surmain.cpp

```
   #include <vcl\vcl.h>
   #include "Unidadppal.h"
   #include "SesionProtocolo.h"
   #pragma hdrstop
   #include "Main.h"
   #include "SurMain.h"
   extern   BOOL Conectado;
   unsigned char tabla[200];
   extern int pinicial;
   extern short recibido;
           extern TTable *Table1;
           extern TSmallintField *Table1NUMERO;
```

```
        extern TFloatField *Table1POTENCIA;
        extern TFloatField *Table1TENSION_DE_PANEL;
        extern TFloatField *Table1INTENSIDAD_DE_PANEL;
        extern TFloatField *Table1TENSION_DE_RED;
        extern TFloatField *Table1INTENSIDAD_DE_BOBINA;
extern HANDLE hComm;
unsigned char InBuff[100];
unsigned char error;
unsigned long int k1;
int pinicial=0;
int pfinal=0;
extern unsigned char inst;
unsigned char ip1[100];
unsigned char tp1[100];
unsigned char ib1[100];
unsigned char tr1[100];
unsigned char ip2[100];
unsigned char tp2[100];
unsigned char ib2[100];
unsigned char tr2[100];
unsigned char ip3[100];
unsigned char tp3[100];
unsigned char ib3[100];
unsigned char tr3[100];
unsigned char ip4[100];
unsigned char tp4[100];
unsigned char ib4[100];
unsigned char tr4[100];
short int ktp1=0;
short int kip1=0;
short int ktr1=0;
short int kib1=0;
short int ktp2=0;
```

```
short int kip2=0;
short int ktr2=0;
short int kib2=0;
short int ktp3=0;
short int kip3=0;
short int ktr3=0;
short int kib3=0;
short int ktp4=0;
short int kip4=0;
short int ktr4=0;
short int kib4=0;
__fastcall TRead::TRead(bool CreateSuspended)
    : TThread(CreateSuspended)
{
}
//---------------------------------------------------------------------------
void __fastcall TRead::Execute()
{
//Contador de bytes leídos
   DWORD dwBytesRead=0;
//Destruimos automáticamente el objeto thread cuando el proceso acaba
   FreeOnTerminate = true;
   while(1)
  {
//Mira a ver si hay algún carácter en el búfer interno de recepción. Si lo hay, lo
//lee. Si no, espera a que llegue alguno o acaba si el hilo está terminado.
      if(Terminated)
        {
         DataModule1->Database1->Close();
         DataModule1->Session1->Close();
         return;
         }
      ReadFile(hComm, InBuff, 50, &dwBytesRead, NULL);
```

```
        if(dwBytesRead)
        {
        recibido=1;
          for (int h=0; h<int(dwBytesRead) ;h++)
          {
           tabla[pinicial++]=InBuff[h];
           if(pinicial==200) pinicial=0;
          }
//RECEPCION DE TRAMAS
unsigned int j;
unsigned long int k;
unsigned char i, car=0;
unsigned char contador=0, contrama=0;
unsigned char check, checksum, remite;
unsigned char trama[20];
    error=2;
    k1++;
    int k2=0;
    while(error==2 && k2<10000)
    //Mientras no se produzca ningún error ni se sobrepase el tiempo de espera
     {
        k2++;
        if(pinicial!=pfinal)
        {
        k2=0;
        car=tabla[pfinal++];
        if (pfinal==200) pfinal=0;
        switch(contador)
                {
                case 0:
                //comprobación de que el mensaje es para el PC
                    if(((car&192)==128)&&  //se trata de una dirección
```

```
        ((car&63)==0))   //nuestra dirección es 0
        {
        check=car;
        /*elaboramos la suma de los caracteres enviados para
        posteriormente comprobar que coincide con el checksum*/
        contador++;
        }
        else
        {
        contador=0;
        contrama=0;
        error=1;
        }
        break;
case 1:
if((car&240)==0)//se trata de una instrucción o un dato
{
      if(inst==car)
        {
        check+=car;
        contador++;
        }
        else
        {
        contador=0;
        contrama=0;
        error=1;
        } }
else
{
contador=0;
contrama=0;
error=1;
```

```
}
break;
case 2:
	switch(car&240)
	{
		case 0://Nibble alto de datos
			check +=car;
			trama[contrama] = car<<4;
			contador++;
			break;
		case 32://Nibble alto de checksum
			checksum = car<<4;
			contador=4;
			break;
		default:
			contador = 0;
			contrama = 0;
			error=1;
	}
	break;
case 3:
	switch(car&240)
	{
		case 0://Nibble bajo de datos
			check+=car;
			trama[contrama]+=car;
			contador=2;
			contrama ++;
			break;
		default:
			contador=0;
			contrama=0;
			error=1;
```

```
                }
                break;
        case 4:
                if ((car&240)==16)
                {               //Nibble bajo de checksum
                        checksum+=car&15;
                        if(check!=checksum)
                        {
                        contador=0;
                        contrama=0;
                        error=1;
                        }
                        else
                        {
                        contador++;
                        }
                }
                else
                {
                        contador=0;
                        contrama=0;
                        error=1;
                }
                break;
        case 5://Comprobación de fin de trama
                if ((car&192)==64)
                {
                        remite=car&63;
                        if(remite==1)//El remitente debe ser el datalogger
                        {
                        contador++;
                        }
                        else
```

```
                                {
                                contador=0;
                                contrama=0;
                                error=1;
                                }
                        }
                        if(contador==6)
                        {
switch(inst)
 {
   case  0:
                         break;
   case  1:             if (contrama != 5)
                                {
                                error=1;
                                break;
                                }
      DataModule1->StoredProc1->ParamByName("num")->AsSmallInt = trama[0];
      DataModule1->StoredProc1->ParamByName("tenpan")->AsFloat= trama[1];
      DataModule1->StoredProc1->ParamByName("intpan")->AsFloat= trama[2];
      DataModule1->StoredProc1->ParamByName("tenred")->AsFloat= trama[3];
      DataModule1->StoredProc1->ParamByName("intbob")->AsFloat= trama[4];
      DataModule1->StoredProc1->ExecProc();
             Formppal->Table1->Refresh();
           switch(trama[0])
             {
             case 1:
             tp1[ktp1++] = trama[1];
             ip1[kip1++] = trama[2];
             tr1[ktr1++] = trama[3];
             ib1[kib1++] = trama[4];
             if(ktp1==100) ktp1=0;
             if(kip1==100) kip1=0;
```

```
if(ktr1==100) ktr1=0;
if(kib1==100) kib1=0;
break;
case 2:
tp2[ktp2++] = trama[1];
ip2[kip2++] = trama[2];
tr2[ktr2++] = trama[3];
ib2[kib2++] = trama[4];
if(ktp2==100) ktp1=0;
if(kip2==100) kip2=0;
if(ktr2==100) ktr2=0;
if(kib2==100) kib2=0;
break;
case 3:
tp3[ktp3++] = trama[1];
ip3[kip3++] = trama[2];
tr3[ktr3++] = trama[3];
ib3[kib3++] = trama[4];
if(ktp3==100) ktp3=0;
if(kip3==100) kip3=0;
if(ktr3==100) ktr3=0;
if(kib3==100) kib3=0;
break;
case 4:
tp4[ktp4++] = trama[1];
ip4[kip4++] = trama[2];
tr4[ktr4++] = trama[3];
ib4[kib4++] = trama[4];
if(ktp4==100) ktp4=0;
if(kip4==100) kip4=0;
if(ktr4==100) ktr4=0;
if(kib4==100) kib4=0;
break;
```

```
                        }
                         break;
          case  2:
                  inst=3;
                         break;
          case  3:
                    Sleep(500);
                  inst=1;
                         break;
        default: ShowMessage("instrucción del Data Logger desconocida");
                         }
                         }
                         contador=0;
                         contrama=0;
                         if (error==2) error=0;
                         break;
                         default:
                                  contador=0;
                                  contrama=0;
                                  error=1;
                                 }
                      }
                      }
                      if(error==1) Sleep(200);
        }
  }
  }
```

CLIENTE MPF

InvCliente.cpp

```
#include <vcl.h>

#pragma hdrstop

USERES("InvCliente.res");

USEFORM("modulodatos.cpp", DataModule1); /* TDataModule: File Type */

USEFORM("Unidadppal.cpp", Formppal);

USEFORM("Unidad1.cpp", Form1);

USEFORM("Unidad2.cpp", Form2);

USEFORM("Unidadayuda.cpp", Formayuda);

USEFORM("Unidadpresentacion.cpp", Formpresentacion);

//---------------------------------------------------------------------------

WINAPI WinMain(HINSTANCE, HINSTANCE, LPSTR, int)

{

        try

        {

                Application->Initialize();

                Application->Title = "Cliente MPF";

                Application->CreateForm(__classid(TFormppal), &Formppal);

                Application->CreateForm(__classid(TForm1), &Form1);

                Application->CreateForm(__classid(TForm2), &Form2);

                Application->CreateForm(__classid(TFormayuda), &Formayuda);

                Application->CreateForm(__classid(TDataModule1), &DataModule1);

                Application->Run();

        }

        catch (Exception &exception)

        {

                Application->ShowException(&exception);

        }

        return 0;

}
```

Unidadppal.cpp

```
#include <vcl.h>

#include <conio.h>

#pragma hdrstop

#include "Printers.hpp"

#include "Unidadppal.h"

#include "Unidad1.h"

#include "Unidad2.h"

#include "Unidadpresentacion.h"

#include "modulodatos.h"

#include "Unidadayuda.h"

//---------------------------------------------------------------------------

#pragma package(smart_init)

#pragma resource "*.dfm"

TFormppal *Formppal;

//---------------------------------------------------------------------------

__fastcall TFormppal::TFormppal(TComponent* Owner)

        : TForm(Owner)

{

}

//---------------------------------------------------------------------------

void __fastcall TFormppal::FormCreate(TObject *Sender)

{

//PRESENTACION

/*Creamos dinámicamente la ficha de la presentación para

poder liberar memoria posteriormente*/

TFormpresentacion * Formpresentacion;

//TImage * Foto;

Formpresentacion = new TFormpresentacion(this);

Formpresentacion->Position = poScreenCenter;

/*MediaPlayer2 = new TMediaPlayer (Formpresentacion);

MediaPlayer2->Parent=Formpresentacion;

MediaPlayer2->DeviceType=dtAutoSelect;
```

```
MediaPlayer2->FileName="C:\\Download\\mp3\\apollo.mp3";
MediaPlayer2->AutoEnable=true;
MediaPlayer2->Visible=false;
MediaPlayer2->Open();
MediaPlayer2->Play();*/
Formpresentacion->ShowModal();
/*Centrar la ficha para facilitar la visualización*/
Formppal->Position = poScreenCenter;
Formppal->Visible=true;
Timer1->Enabled=false;
}
//---------------------------------------------------------------------------
/*Elección del índice de la tabla mediante la pulsación
del título de una de sus columnas*/
void __fastcall TFormppal::DBGrid1TitleClick(TColumn *Column)
{
                    try
        {
            DataModule1->ClientDataSet1->IndexFieldNames = Column->FieldName;
                 }
                        catch(Exception&)
                 {
                 }
}
//---------------------------------------------------------------------------
void __fastcall TFormppal::DBGrid2TitleClick(TColumn *Column)
{
        try
        {
            DataModule1->ClientDataSet2->IndexFieldNames = Column->FieldName;
                 }
                        catch(Exception&)
                 {
```

```
                    }
}
//---------------------------------------------------------------------------
/*Visualización de la ayuda*/
void __fastcall TFormppal::BitBtn1Click(TObject *Sender)
{
Formayuda->Visible=true;
}
//---------------------------------------------------------------------------
/*Actualización de la Base de Datos*/
void __fastcall TFormppal::BitBtn2Click(TObject *Sender)
{
/*Se pretende elegir los registros de la tabla 1 mediante
la columna Estado de la tabla 2*/
if (DataModule1->ClientDataSet2->State == dsEdit) {
/*Pero si la tabla 2 se encuentra en estado de edición
en este momento, también debe actualizarse*/
DataModule1->ClientDataSet2->Post();
}
DataModule1->ClientDataSet2->Open();
DataModule1->ClientDataSet2->ApplyUpdates(-1);
Sleep(200);
DataModule1->ClientDataSet1->Close();
DataModule1->ClientDataSet1->Open();
DataModule1->ClientDataSet1->ApplyUpdates(-1);
Timer1->Enabled=true;
}
//---------------------------------------------------------------------------
/*Salvar los datos presentes en la Tabla 1*/
void __fastcall TFormppal::BitBtn3Click(TObject *Sender)
{
/*Para guardar los históricos nos serviremos de la tecnólogía MIDAS
creada por Borland para el desarrollo de aplicaciones multicapa.
```

```
Esto nos permite usar un "conjunto de datos del cliente" con el que
realizar una copia de los datos que en ese momento alberguemos en la
tabla.*/
        /*Primeramente debemos actualizar este conjunto, y para ello
proceder a activarlo y desactivarlo*/
        ClientDataSet1->Active = false;
        ClientDataSet1->Active = true;
        if(SaveDialog1->Execute()){
        /*Una vez actualizado con la información relevante, lo
utilizamos como trampolín para salvarla a un archivo*/
ClientDataSet1->SaveToFile(SaveDialog1->FileName,dfBinary);
                                    }
}
//---------------------------------------------------------------------------
/*Impresión de la tabla 1 mediante "Quick Report"*/
void __fastcall TFormppal::BitBtn4Click(TObject *Sender)
{
/*El siguiente código es necesario si tenemos varias
impresoras disponibles*/
        /*Configuración de la impresora*/
       PrintDialog1->PrintToFile = false;
       if( PrintDialog1->Execute()){
        /*Comunicar a Quick Report cuál es la impresora seleccionada*/
        Form1->QuickRep1->PrinterSettings->PrinterIndex = Printer()
        ->PrinterIndex;
        /*y el resto de parámetros de interés*/
        Form1->QuickRep1->PrinterSettings->FirstPage = PrintDialog1->
        FromPage;
        Form1->QuickRep1->PrinterSettings->LastPage = PrintDialog1->
        ToPage;
        Form1->QuickRep1->PrinterSettings->Copies = PrintDialog1->
        Copies;
        /*Dado el elevado número de columnas que se maneja conviene
```

```
imprimir de forma apaisada*/
        Form1->QuickRep1->PrinterSettings->Orientation = poLandscape;
        /*De esta forma no hemos tenido en cuenta el resto del
"PrintDialog1", por lo que es inútil pulsar el botón de propiedades
de esta ventana*/
        //En el laboratorio este comando no surte efecto
        //debido a algún error en los drivers, del Builder o quizás
        //del propio código del programa
        /*Imprimir el informe*/
        Form1->QuickRep1->Print();}
}
//---------------------------------------------------------------------------
/*Cargar histórico previamente salvado*/
void __fastcall TFormppal::BitBtn5Click(TObject *Sender)
{
/*Apoyándonos en el código de la función anterior es muy sencillo
desarrollar esta: no es necesaria la conexión de nuevos componentes*/
if(OpenDialog1->Execute()){
ClientDataSet1->LoadFromFile(OpenDialog1->FileName);
//ejercicio de estética
Form2->Caption = "Histórico " + OpenDialog1->FileName;
Form2->Position = poScreenCenter;
Form2->WindowState = wsMaximized;
Form2->DBGrid1->DataSource = DataSource3;
Form2->DBGrid1->Left = 0;
Form2->DBGrid1->Width = Form2->ClientWidth;
Form2->DBGrid1->Height = Form2->ClientHeight;
Form2->Visible = true;
}}
//---------------------------------------------------------------------------
//Cada 6 segundos se actualiza la información recibida del servidor
//de forma que el cliente tenga suficiente tiempo como para actuar
//sobre la tabla 2
```

```
void __fastcall TFormppal::Timer1Timer(TObject *Sender)
{
                    DataModule1->ClientDataSet1->Close();
                    DataModule1->ClientDataSet1->Open();
                    DataModule1->ClientDataSet2->Close();
                    DataModule1->ClientDataSet2->Open();
}
//---------------------------------------------------------------------------
void __fastcall TFormppal::BitBtn6Click(TObject *Sender)
{
Close();
}
```

BIBLIOGRAFÍA:

[1] La cara oculta de C++ Builder. *Ian Marteens*

[2] Programación con C++ Builder 4. *Francisco Charte Ojeda*

[3] C++:Manual de Referencia con anotaciones. *M.A. Ellis. B.Stroustrup*

[4] C++ for C programmers.*I.Pohl*

[5] Introducción a los sistemas de bases de datos. *C.J.Date*

[6] Diseño de bases de datos. *G.Wiederwhold*

[7] Creación de bases de datos en Internet. *J.Sinclair. C. McCullogh*

[8] Gestión de bases de datos en Internet: JDBC. *J.M. Framiñán Torres. J.M. León Blanco*

[9] Proyecto de Fin de Carrera P1996: "Estudio de posibilidades de acceso a sistemas gestores de bases de datos a través de Internet. Encuesta calidad de enseñanza mediante JDBC.". *J.M.León Blanco.*

[10] A fondo. Microsoft SQL Server. *R.Soukup. K.Delaney*

[11] *InterBase Tutorial.*

[12] *Developing Database Applications help.*

[13] *C++ Builder Language Guide help.*

[14] *Visual Component Library Reference help.*

[15] *Programming with C++ Builder help.*

[16] *MS Windows System Development Kit help.*