Turbo C++ 2006

For Microsoft® Windows™
Professional

IntraWeb 8 for Borland Developer Studio 2006

Borland®   Registered
Made in Borland®. Copyright© 2005 Borland Software Corporation. All rights reserved.
Loading package dclindy100.bpl

Turbo C++ 2006
& C++Builder 2006
Database
Development

Bob Swart (aka Dr.Bob)
http://www.drbob42.com

Table of Contents

The contents of this Turbo C++ / C++Builder 2006 Database Development courseware manual is based on my articles about C++Builder 6 and Database Development that were published in the C++Builder Developer's Journal in 2005. I've migrated the articles to C++Builder 2006 and Turbo C++ by retaking all screenshots and verifying the code and steps against the new version of the C++Builder IDE.

If you like this manual, then you may also like a subscription to the C++Builder Developer's Journal – see http://www.bcbjournal.com for more details.

The free version of this Turbo C++ / C++Builder Database Development courseware manual is distributed in a PDF format that can only be read on a computer screen, and cannot be printed (nor can you select and copy text or code snippets from the document).

For 15 Euro or 21 US$ you can get a personal version of the PDF file that can be printed, including full source code with projects for C++Builder 2006.

To order this PDF file, just use PayPal to transfer 15 Euro or 21 US$ to my PayPal account paypal@drbob42.com. After I've received your payment, I will produce a personal version of the PDF file and send it to your e-mail address (the same one you used to make the PayPal payment). I regret that I cannot accept any other payment methods (note that you can use credit cards with PayPal without becoming a member).

For any questions, feedback or comments, feel free to contact me at Bob@drbob42.com. Thanks in advance!

0. Database Development

In this first section, I will give a brief introduction and list all available data access technologies that are available in C++Builder 6, C++Builder 2006 and Turbo C++ 2006. Most even work with C++Builder 5, and perhaps even older versions although I haven't verified that. In this courseware manual, I will use Borland C++Builder 2006 for all screenshots, but you can safely assume that everything also works in C++Builder 6 and Turbo C++ 2006. Therefore, from now on I will just refer to C++Builder when talking about the development environment.

The rest of the manual, I'll follow up with one section per "technology", working with similar data, showing similar techniques (how to work with a single table, do master-details, queries, queries with parameters, use lookup tables, insert/add records, edit/post, and delete records, etc.), which databases can be used, and what it takes to create the tables or database.

Data Access Technologies

The data access technologies that are present in C++Builder and Turbo C++ and will be covered in the following sections, are the following:
- Borland Database Engine (BDE)
- ADO (dbGo for ADO aka ADOExpress)
- stand-alone ClientDataSet
- cross-platform dbExpress

C++Builder Enterprise (or Architect) developers can also use DataSnap for building multi-tier database applications. But DataSnap is not available for Turbo C++ 2006 developers, and not covered in this courseware manual.

With that in mind, let's define the problem space - the data model if you wish - for this database development courseware manual. I wanted to cover something that was both fun, not too artificial, small enough to do in a training class, and yet even a bit useful.

CD/DVD Data Model

The answer is: my CD/DVD collection database. It contains enough fields and tables to make it worthwhile, and is in fact a real-world application (for all my family members who want to borrow DVDs - and for me keeping track of who has borrowed what, and since when).

The database consists of four tables:

CD / DVD	
Field	**Type**
CdDvdID	Integer (key)
CategoryID	Integer (foreign key)
Name	String
Description	String

Category (lookup table)	
Field	**Type**
CategoryID	Integer (key)
Name	String

Friend	
Field	**Type**
FriendID	Integer (key)
Name	String
Description	String
Contact (e-mail, MSN, phone)	String

Borrow	
Field	**Type**
CDDVDID	Integer (key)
FriendID	Integer (foreign key)
SinceDate	Date

Note that the Borrow table and the CD / DVD table actually have a 1-on-1 relationship. But since the Borrow fields will only be used if the CD / DVD is actually borrowed, it makes sense to put this information in a separate table. As a downside: it'll take longer to determine if a CD / DVD is actually available (but that will be one of our observations as of the next section).

On the other hand: in this data model, each CD or DVD can have exactly one category. In practice, this will be a little difficult at times to select the best category for a DVD. Extending the model to allow a DVD to be categorised in more than one category means adding another table with the CDDVDID and CategoryID. Not a big problem, but a solution that I didn't implement for the simple reason that my actual physical CD's and DVD's are currently kept on shelves in closets, sorted by category. And unless I own multiple copies of the same CD or DVD, they can only be found in one category...

Finally, note that in the proposed data model, the two description fields are optional for my examples, and will probably be expanded to several fields with interesting information (but totally irrelevant to the technologies I want to demonstrate).

Summary

In this section, I have introduced the data model for courseware manual on data access technologies available in C++Builder. In the next section, I'll start to cover the Borland Database Engine, using Paradox files to implement the data module. After that, we'll move on to ADO and an ADO database (like SQL Server MSDE or Access), followed by a section where we'll examine the stand-alone TClientDataSet as alternative. In the fourth section, I'll cover dbExpress- the cross-platform data access technologies that was branded as the replacement for SQL Links.

C++Builder Enterprise or Architect developers can extend all this to the multi-tier world, adding DataSnap to the mix (which can use any of the previously covered data access technologies as basis). But like I wrote in the introduction: DataSnap is not part of this courseware manual.

1. The Borland Database Engine

In the previous section, I've introduced the database development problem domain, which will take us through just about all different data access techniques in C++Builder. This first part will cover the Borland Database Engine (BDE) with the Paradox, dBASE, ForPro or ASCII local table formats. Note that I will not cover SQL Links – the big brother of the BDE that could be used to connect to DBMSs like InterBase, SQL Server, Oracle, etc. since SQL Links has been declared deprecated a while ago. The recommended replacement to talk to SQL DBMSs is dbExpress (or ADO if you wish), which both will be covered in the sections to come. This time, we'll focus only on local tables in the Paradox table format.

Local Tables

A local table supported by the Borland Database Engine is meant as the name suggests: a local table. The Borland Database Engine is not really equipped to handle client/server or multi-user situations, although you can try it in a small setting. I would recommend however to use dBASE, Paradox, ASCII or FoxPro files for desktop applications only.

Regarding the choice between dBASE, Paradox and FoxPro table formats, it's a matter of taste. There is little difference, but if I have a choice, I'll pick a Paradox 7 type table. Feel free to use the dBASE format for your own implementation of the example application, however, it shouldn't make much of a difference. Paradox uses a .db file extension, dBASE a .dbf, and ASCII files a .txt file extension (and I'm not sure what FoxPro uses – I've never used FoxPro, sorry).

Example DVD Application

We started in the previous section by defining a simple data model, using my CD and DVD collection at home, including categories and friends who borrow the DVDs. A four table solution. And the first step this section is deciding how to map this logical data model into physical tables.

Creating Tables using Database Desktop

The easiest way is to use the Database Desktop, This tool allows us to create dBASE and Paradox tables, where you can define your fields, their types, size (in case of a string or memo field) and whether or not this is a key field, and other relevant information like the fact if this is a required fields, etc., as shown in the next screenshot where I just created the CDDVD table as Paradox 7 Table (using Database Desktop on Windows 2000).
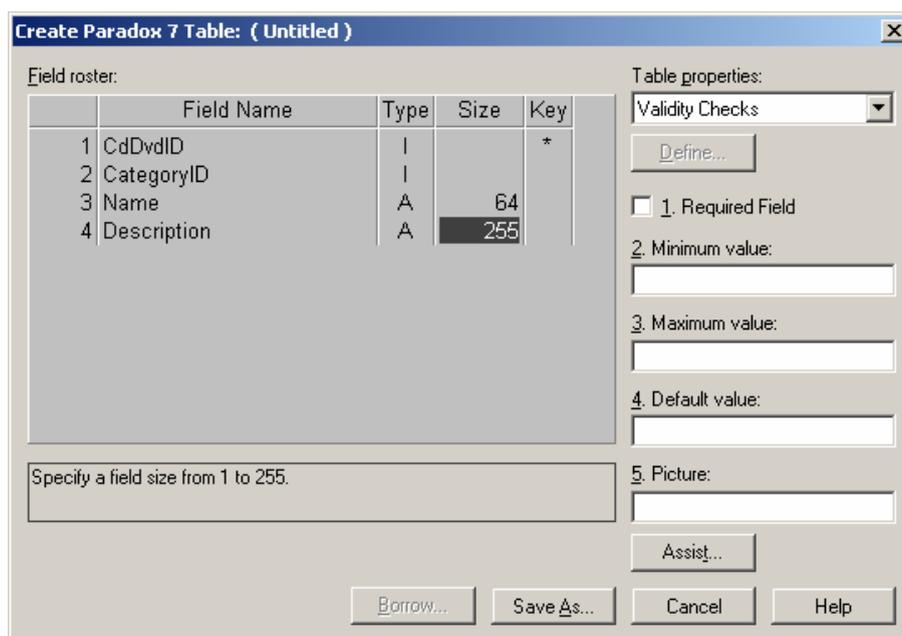

Figure 1. Database Desktop creating CdDvd.db

The CdDvdID and CategoryID fields are of type Integer, the Name field is a string of 64 characters, and the Description is a string field of 255 characters. These 255 are the maximum value we can specify for a string field in a Paradox table. If you want to store larger strings, you can use a Memo type field (which is in fact a BLOB field). In that case, you can use the Size column to specify how many characters of the BLOB will be stored in the table –the remainder will be stored in a special file with the .mb extension (for Paradox, the table file itself has a .db extension, and the index the .px extension).
When the definition is complete, you can click on Save As and save the table in an external file, such as CdDvd for the current example.

Note that the shortcut for Database Desktop is no longer created when installing C++Builder 2006 or Turbo C++ 2006, but it's part of the BDE and can often be found in the C:\Program Files\Common Files\Borland Shared\Database Desktop directory as dbd32.exe.

Creating Tables in C++

Apart from using the Database Desktop, it's sometime very useful (or necessary) to be able to create your local tables with just a few lines of C++ code. In the coming sections we'll see how we can also create database tables using SQL commands, but a local BDE table is a different matter. For those tables, we can use the TTable component, specify the TableType, TableName, the Field definitions, Index definitions, and finally physically create the table on disk. Although this can be done from any kind of application, I often use a simple Console application to produce the tables (without the need for a GUI).
Do *File | New – Other*, and in the Object Repository, and double-click on the Console Application icon in the C++Builder Projects category. This will produce the Console Wizard to create a new Console Application. Note that we should enable the use of the VCL (since the BDE requires the db and dbtables units, which are part of the VCL).
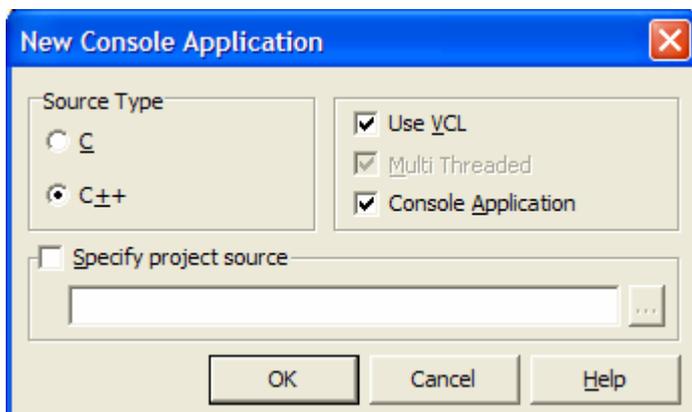
Figure 2. New Console Application Wizard

In the generated source file, add <db.hpp> and <dbtables.hpp> to the #include sections, and then write the code from listing 1 in the main part of the console application. The TableType can be set to ttParadox, ttDBase, ttFoxPro or ttASCII (or ttDefault, which then uses the extension of the tablename - .db, .dbf or .txt - to determine the table type).

```
//---------------------------------------------------------------------------

#include <vcl.h>
#include <db.hpp>
#include <dbtables.hpp>
#pragma hdrstop

//---------------------------------------------------------------------------

#pragma argsused
```

```cpp
int main(int argc, char* argv[])
{

  TTable* Table = new TTable(NULL);
  Table->Active = false;
  Table->TableType = ttParadox;
  Table->TableName = "Category.db";
  Table->FieldDefs->Clear();
  Table->FieldDefs->Add("CategoryID", ftInteger, 0, FALSE);
  Table->FieldDefs->Add("Name", ftString, 32, FALSE);
  Table->IndexDefs->Clear();
  Table->IndexDefs->Add("", "CategoryID", TIndexOptions() <<ixPrimary << ixUnique);
  Table->CreateTable();
  Table->Free();

  Table = new TTable(NULL);
  Table->TableType = ttParadox;
  // CD/DVD Table
  Table->TableName = "CdDvd.db";
  Table->FieldDefs->Clear();
  Table->FieldDefs->Add("CdDvdID", ftInteger, 0, false);
  Table->FieldDefs->Add("CategoryID", ftInteger, 0, false);
  Table->FieldDefs->Add("Name", ftString, 64, false);
  Table->FieldDefs->Add("Description", ftString, 255, false);
  Table->IndexDefs->Clear();
  Table->IndexDefs->Add("", "CdDvdID", TIndexOptions() <<ixPrimary << ixUnique);
  Table->CreateTable();
  Table->Free();

  Table = new TTable(NULL);
  Table->TableType = ttParadox;
  // Friend Table
  Table->TableName = "Friend.db";
  Table->FieldDefs->Clear();
  Table->FieldDefs->Add("FriendID", ftInteger, 0, false);
  Table->FieldDefs->Add("Name", ftString, 64, false);
  Table->FieldDefs->Add("Description", ftString, 255, false);
  Table->FieldDefs->Add("Contact", ftString, 128, false);
  Table->IndexDefs->Clear();
  Table->IndexDefs->Add("", "FriendID", TIndexOptions() <<ixPrimary << ixUnique);
  Table->CreateTable();
  Table->Free();

  Table = new TTable(NULL);
  Table->TableType = ttParadox;
  // Borrow Table
  Table->TableName = "Borrow.db";
  Table->FieldDefs->Clear();
  Table->FieldDefs->Add("CdDvdID", ftInteger, 0, false);
  Table->FieldDefs->Add("FriendID", ftInteger, 0, false);
  Table->FieldDefs->Add("Date", ftDate, 0, false);
  Table->IndexDefs->Clear();
  Table->IndexDefs->Add("", "CdDvdID;FriendID", TIndexOptions() <<ixPrimary << ixUnique);
  Table->CreateTable();
  Table->Free();

  return 0;
}
//---------------------------------------------------------------------------
```

                    Listing 1. Console Create Table

This will produce the Category, CdDvd, Friend, and Borrow tables, consisting of their fields
and the key fields added to the primary key index.

The ability to create local tables in code is more powerful than may seem at first, since it means that you can deploy your application without the need to deploy empty Paradox or dBASE tables. The application can create its own set of local tables when needed.

BDE Data Access

Once we have a set of local tables, we can use the components from the BDE tab of the component palette to access these tables. There are three different dataset components: TTable, TQuery, and TStoredProc. Each of the three has specific features. Note, however, that TStoredProc can only be used in combination with an SQL DBMS (and not with local dBASE and Paradox files that do not support stored procedures), so I'll cover TTable and TQuery components only this section.

TTable

A TTable component is the most basic of all data access components. With only three properties, you can access the contents of a local database file. First of all, the DatabaseName property should be set to BDE Alias, or to a directory. If you want to deploy an application to several machines, then the use of a BDE Alias means you need to define the BDE Alias on the other machines as well. However, you can always specify a (relative) path as DatabaseName property, such as .\ or .\Data to specify the data subdirectory of the current working directory.

Apart from the DatabaseName property, you also need to specify the TableName property. A drop-down list in the Object Inspector will help you to select a table (assuming the DatabaseName property points to a valid BDE Alias or directory containing database table files).

The third property is the Active property. With this property, you can show live data at design-time. However, if you want to ship your application with the ability to produce its own (empty) tables as bootstrapping, then it's better to set Active to false at design-time (just before the final compilation), and opening the tables at run-time using a few lines of C++ code (we'll see this in a minute).

TQuery

A TQuery component offers the ability to execute a SQL command, most often a SQL SELECT command, retrieving records from one or more tables. The SQL is a subset of the SQL that real SQL DBMSs support, but good enough for more uses. Simple SQL commands can be made more flexible by adding parameters to them, so you can re-execute the same SQL command by changing a parameter value (which is faster than re-executing an entirely new SQL command).

Like the TTable, the TQuery also has a DatabaseName property, but instead of a TableName you get a SQL property of type TStringList that can hold the SQL command. There's also a ParamData property that contains the parameter definitions, as we'll see shortly.

BDE Example Application

Create a new C++ VCL Application to start the new BDE example application, ready to work with the BDE tables.

Although you can place your data access components on the same Form as your visual controls, it's generally recommended to place the data access components in a special form, called the Data Module. This not only prevents the visual form from being cluttered with all kinds of non-visual components, it also helps you to structure your data model independent of your user interface, and to re-use the data module in multiple forms (if you wish), again resulting in code which is easier to maintain.

Data Module

A data module is a special module where you can concentrate your non-visual data access components such as TTables and TQueries. You can even combine a data module with the ability to create local tables at runtime by making sure the tables are created on disk before the data module opens them.

For the current example at hand, we can create a new data module using File | New – Other, finding the Data Module icon in the Object Repository.

On the new data module we can place TTable and TQuery components. Let's place three TTables, one for the Category.db table, one for the CdDvd.db table and one for the Friend.db table. I call these components tbCategory, tbCdDvd and tbFriend. I also want to place one TQuery component, to look for the DVDs that have been borrowed by a friend (and you could also add another TQuery to look for the Friend who has borrowed a specific CD or DVD), these can be called qBorrowFriend (what did the Friend borrow) and qBorrowCdDvd (who borrowed this CD or DVD).
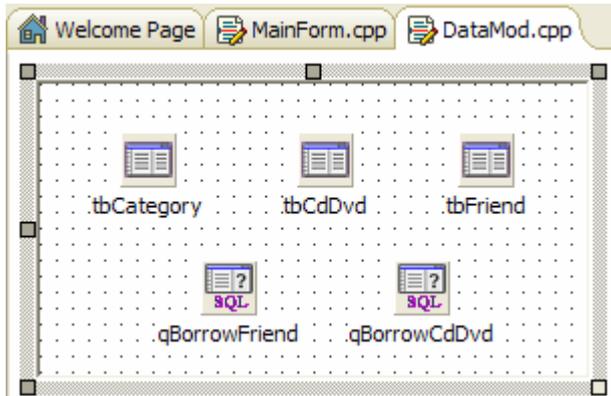

Figure 3. Data Module with TTables and TQueries

For the three TTables and two TQueries, we can specify a DatabaseName property value of .\ meaning that we want to use tables from the current project directory. Note that this directory may be different at design-time and at runtime (since the executable may end up in a different directory).

We can either use the resulting tables from the previous example, or use the next listing to produce them. Assuming we use the existing tables (which must then be placed in the Data subdirectory of the project directory), we can connect the TableName properties to Category.db, CdDvd.db and Friend.db respectively.

For the two TQuery components, I want to produce a parameterized SQL SELECT Command. The first one, qBorrowFriend, is used to locate the records from the Borrow table where FriendID is equal to a specific FriendID. This can be coded as follows:
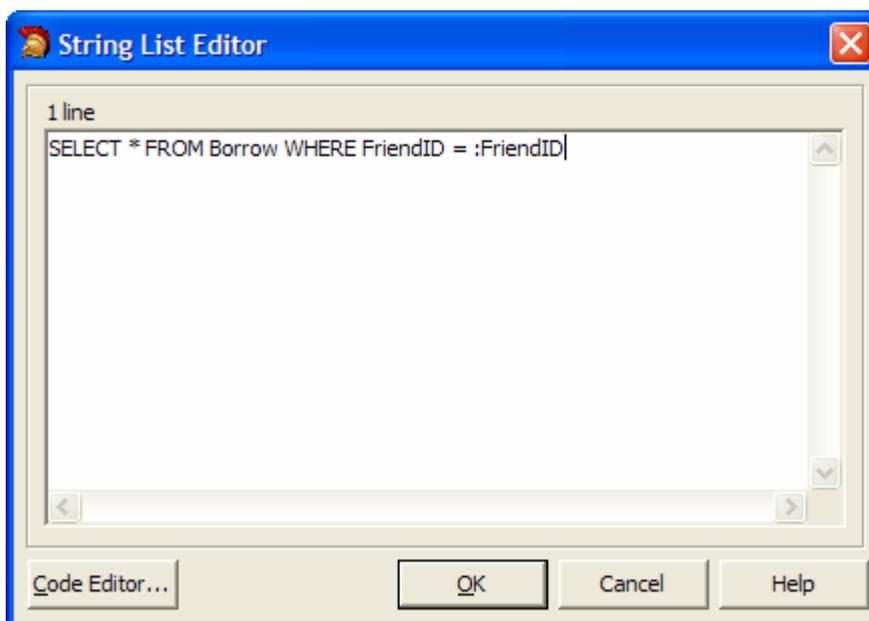

Figure 4. TQuery SQL Text Editor

Note the syntax for the parameter: a colon specifies that this is a parameter name (usually, but not necessarily with the same name as the column it needs to connect to).

After we've specified the SQL contents, we need to specify the DataType and ParamType properties of the parameter. Double-click on the Params property of the qBorrowFriend TQuery command to get the Parameters Editor.
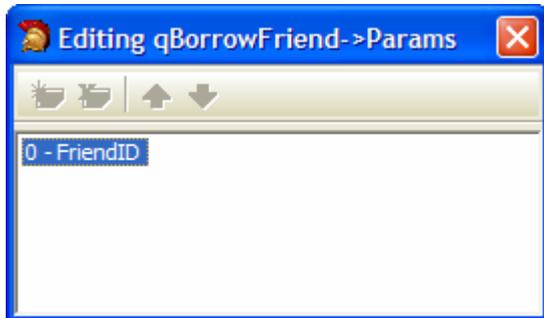

Figure 5. TQuery Parameters Editors

For each parameter – in this example only one of course – we must set the ParamType to input or output (ptInput in this case) and the DataType to the corresponding column type (ftInteger in this case). It may also be useful to specify a default value and type (set to Integer and 0 for example), so you can open the qBorrowFriend at design-time if you wish.
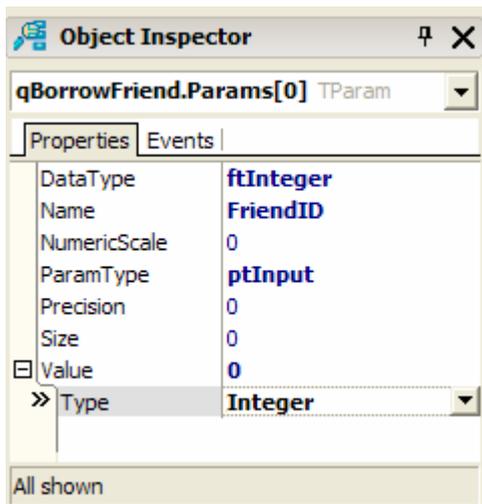

Figure 6. TQuery Parameter Properties

Note that we can access this parameter by index as qBorrowFriend->Params->Items[0] as well as by name as qBorrowFriend->Params->ParamByName("FriendID").

We can set the SQL property of the qBorrowCdDvd to "SELECT * FROM Borrow WHERE CdDvdID = CdDvdID" and add a parameter called CdDvdID of DataType frInteger, ParamType ptInput and default type Integer and value 0.

Creating Tables for Data Module

Although it's easy to define the TTables and TQueries on the data module, if you start from scratch is sometimes handy to be able to deploy the application without empty tables, and let the application produce the tables by itself.

The Data Module has a Create event, where we can check if the local database files exist, and if not, create them on the fly. This code will usually only be executed once, but in my view it adds a great sense of purpose and usefulness to your applications if they are capable of creating their own set of local tables to work with.

The code is basically the same as shown in Listing 1, but this time all four tables are checked (to see if they already exist) and only created when required.

```cpp
void __fastcall TDataModuleBDE::DataModuleCreate(TObject *Sender)
{
  if (!FileExists("Category.db"))
  {
    TTable* Table = new TTable(NULL);
    Table->TableType = ttParadox;
    Table->TableName = "Category.db"; // Category Table
    Table->FieldDefs->Clear();
    Table->FieldDefs->Add("CategoryID", ftInteger, 0, false);
    Table->FieldDefs->Add("Name", ftString, 32, false);
    Table->IndexDefs->Clear();
    Table->IndexDefs->Add("", "CategoryID", TIndexOptions() <<ixPrimary << ixUnique);
    Table->CreateTable();
    Table->Free();
  }
  if (!FileExists("CdDvd.db"))
  {
    TTable* Table = new TTable(NULL);
    Table->TableType = ttParadox;
    Table->TableName = "CdDvd.db"; // CD/DVD Table
    Table->FieldDefs->Clear();
    Table->FieldDefs->Add("CdDvdID", ftInteger, 0, false);
    Table->FieldDefs->Add("CategoryID", ftInteger, 0, false);
    Table->FieldDefs->Add("Name", ftString, 64, false);
    Table->FieldDefs->Add("Description", ftString, 255, false);
    Table->IndexDefs->Clear();
    Table->IndexDefs->Add("", "CdDvdID", TIndexOptions() <<ixPrimary << ixUnique);
    Table->CreateTable();
    Table->Free();
  }
  if (!FileExists("Friend.db"))
  {
    TTable* Table = new TTable(NULL);
    Table->TableType = ttParadox;
    Table->TableName = "Friend.db"; // Friend Table
    Table->FieldDefs->Clear();
    Table->FieldDefs->Add("FriendID", ftInteger, 0, false);
    Table->FieldDefs->Add("Name", ftString, 64, false);
    Table->FieldDefs->Add("Description", ftString, 255, false);
    Table->FieldDefs->Add("Contact", ftString, 128, false);
    Table->IndexDefs->Clear();
    Table->IndexDefs->Add("", "FriendID", TIndexOptions() <<ixPrimary << ixUnique);
    Table->CreateTable();
    Table->Free();
  }
  if (!FileExists("Borrow.db"))
  {
    TTable* Table = new TTable(NULL);
    Table->TableType = ttParadox;
    Table->TableName = "Borrow.db"; // Borrow Table
    Table->FieldDefs->Clear();
    Table->FieldDefs->Add("CdDvdID", ftInteger, 0, false);
    Table->FieldDefs->Add("FriendID", ftInteger, 0, false);
    Table->FieldDefs->Add("Date", ftDate, 0, false);
    Table->IndexDefs->Clear();
    Table->IndexDefs->Add("", "CdDvdID;FriendID", TIndexOptions() <<ixPrimary << ixUnique);
    Table->CreateTable();
    Table->Free();
  }
}
```

<center>Listing 2. DataModuleCreate</center>

At the end of the DataModuleCreate, we can add some code to open the TTables and TQueries that were placed on the data module at design-time.

Since the two TQueries require a parameter value, let's skip those (until we have a parameter value to specify), and just open the three TTables, adding the following three lines of code to the end of the DataModuleCreate method:

```
tbCategory->Active = true;
tbCdDvd->Active = true;
tbFriend->Active = true;
```

This is another reason why it's a good idea to set the Active property of your datasets to false at design-time.

Data-Aware Controls

C++Builder contains a number of different data access technologies and components, but only one set of data-aware controls. The connecting component – one of the most important VCL controls if you ask me – is the TDataSource component. It links the datasets on one hand (like TTable and TQuery from the BDE) and the data-aware controls on the other hand. There are several data-aware controls, like the TDBGrid, TDBNavigator, TDBEdit and TDBLookupCombobox to name but a few.

The TDataSource connects its DataSet property to the DataSet, and the data-aware controls connect their DataSource property to the TDataSource component. It's that easy. And sometimes it can be done for you, as we'll see in a minute.

Persistent Fields

Data-aware controls can be connected to fields from datasets. However, there can be two different kinds of fields: persistent fields and default fields. As the name indicates, default fields are present by default. As an example, if we use a TTable component to open the CDDVD.db table, then by default we get all fields from this table. However, we cannot directly assign code to the individual fields. We have to call a method like FieldsByName to find the field and operate on it.

If you want to work with field components, setting individual property values for example, then you can consider adding persistent fields to your VCL dataset components. You can do this – adding fields and working with field properties – with the Fields Editor. Note that the Fields Editor or Persistent Fields are not BDE specific, so we'll continue to use them in the coming sections.

Fields Editor

One of the great benefits of the C++Builder IDE when working with dataset components, is the Fields Editor. The best place to add all fields you want to make visible to your application, add new fields, define lookup or calculated fields. You can even use the Fields Editor to drag and drop fields on your Form.

For our example, double-click on the tbCdDvd table on the data module, which will show the Fields Editor. By default, no fields are shown. If you add one or more explicit fields (also called persistent fields because their definition is streamed to the .DFM file), then only the explicit fields will be shown. You can right-click in the Fields Editor and add fields. You can also Add All Fields, with the following result:
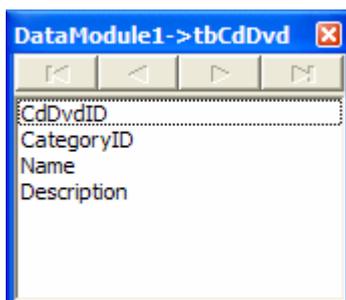

Figure 7. Fields Editor

Apart from the regular persistent fields that "connect" to physical database table fields, which are called data fields, there are two other kinds of explicit fields: lookup fields and calculated fields.

Lookup Fields

Lookup fields are fields that actually contain data from another dataset. In order to determine which data the lookup field contains, two linking fields are used: one from the table that gets the lookup field, and one from the table that holds the field to lookup. You can create lookup fields by right-clicking in the Fields Editor, and selecting the "New Field" option.

See the following screenshot for the definition of the Category string field in the CdDvD table, which is the result of matching the CategoryID field from the CdDvd table to the CategoryID field from the Category table, returning the Name field from the Category table.
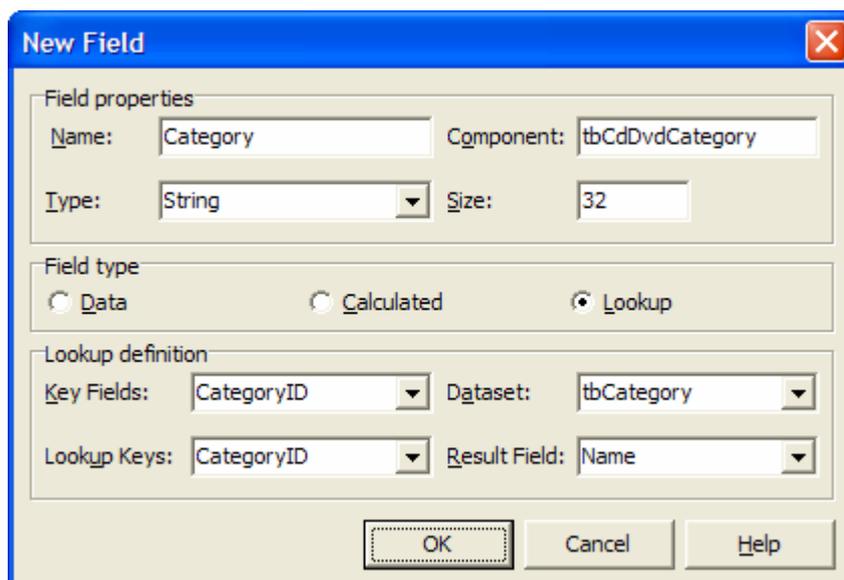

Figure 8. New Lookup Field

Once you've finished the definition of the lookup field, we can use the complete set of fields from the Fields Editor and drag them to the Form. Using C++Builder 6 you only have to make sure both the Form and the Data Module are visible at design-time, so you can select and drag all fields easily. Using C++Builder 2006 or Turbo C++ 2006, the Form and Data Module are not both visible at the same time (in the new tabbed IDE). However, just move the Fields Editor to a convenient place on your screen (so it overlaps the upper-right corner of the IDE) and then click on the tab for your Form. As you will notice, the Fields Editor will stat on top, so you can now select and drag all fields from the Fields Editor to the Form.

Before the "drop" happens to produce the actual display of persistent fields on the form, we first get a dialog that complains about the fact that the DataModule unit is not used by the Form, yet, which can be remedied by clicking on Yes.
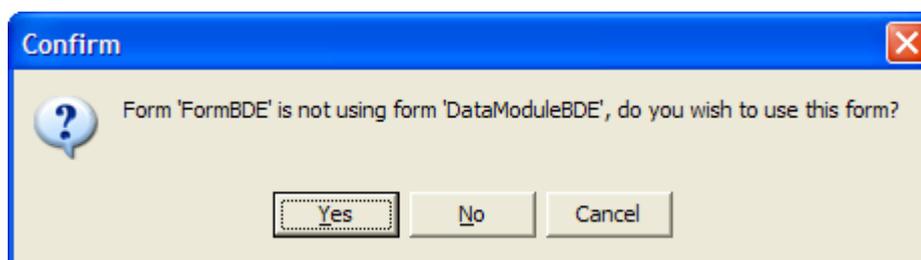

Figure 9. Dialog after Dragging Fields to Form

If you say No the nothing will happen; no components will be placed on the Form. So for best results, just say Yes.

Right after that, you'll see the fields that you dragged to the form being displayed. Note that the regular data fields are represented by TDBEdit controls, while the lookup field is represented by a TDBLookupComboBox (showing the result field, but using the key field hooked to the actual data field in the dataset, just as we defined).
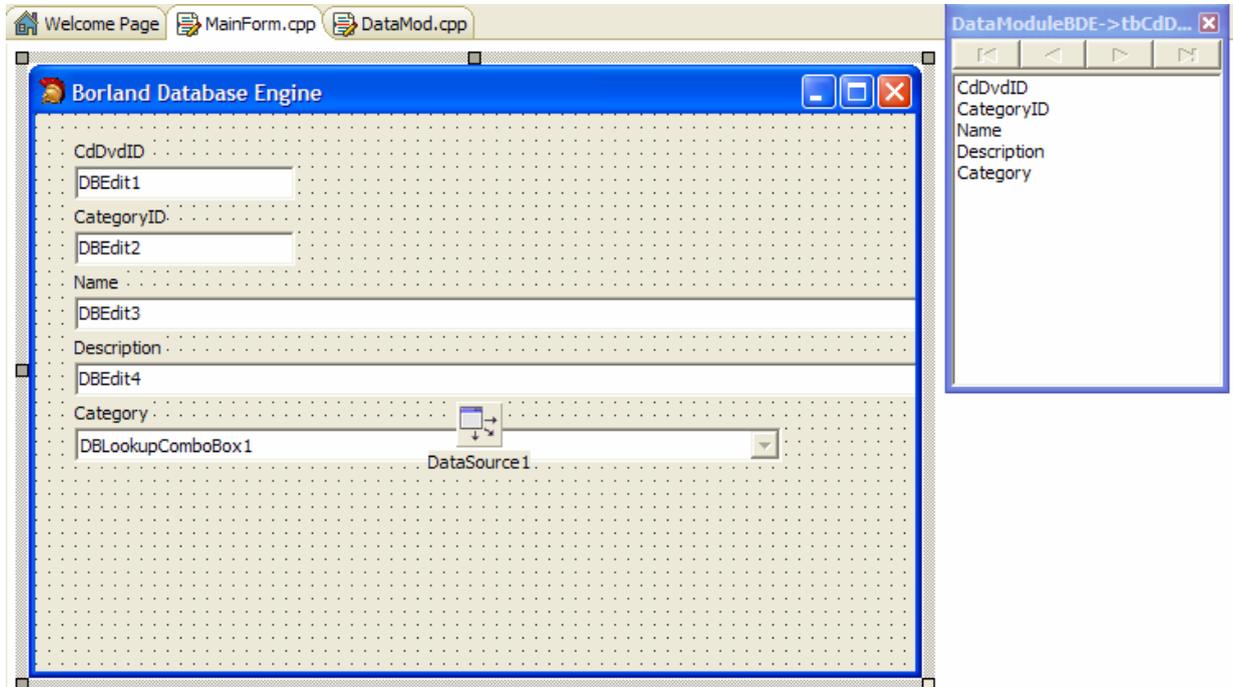
Figure 10. Result of Dragging Data Module TTable Fields to Form

Note that the Name and Description TDBEdit controls that were automatically added have a great width: much larger than the form's width. We have to manually correct that to make sure their width is less than the form's width.

Calculated Fields

Apart from data fields and lookup fields, we can also define calculated fields. A calculated field is shown just like a data field and a lookup field, but is not stored inside the dataset, nor collected from another dataset. As the name suggests, it is calculated on a "when needed" basis: only when the record is displayed or the value is required in some other way. We'll see the trigger for this in a moment, but let's first take a look at the next screenshot which defines the calculated field "Available" of type Boolean in the CdDvd dataset.

Figure 11. New Calculated Field

This field will be used to specify if the corresponding CD or DVD is still available to be borrow (i.e. if I should still be able to find it in my archive, or if I should look for the friend who borrowed it).

Calculated fields are calculated based on a trigger, which is the special OnCalcFields event of the dataset (in this case the tbCdDvd). This event gets a DataSet as input argument, positioned at the record for which we should calculate the calculated field values. Note that there can be more than one calculated field, but yet the OnCalcFields event will only fire once for each record (so we should calculate all fields at once here).

For our example, the value for the Available calculated field can be determined by running a SQL SELECT command to select a record from the Borrow table, specifying the CdDvdID as value for the CdDvdID column. Since each CD or DVD will occur only once (when borrowed) or not at all (when not borrowed) in the Borrow table, the result of the SELECT command will hold zero or one records. If it's one record, then the CD or DVD is borrowed, otherwise it's still available. This code can be seen in Listing 3.

```
void __fastcall TDataModuleBDE::tbCdDvdCalcFields(TDataSet *DataSet)
{
  TQuery* Query = new TQuery(NULL);
  Query->SQL->Text = "SELECT * FROM Borrow WHERE CdDvdID = " +
    DataSet->FieldByName("CdDvdID")->AsString;
  Query->Active = true;
  if (Query->RecordCount == 0)
    DataSet->FieldByName("Available")->AsBoolean = true;
  else
    DataSet->FieldByName("Available")->AsBoolean = false;
  Query->Close();
  Query->Free();
}
```
<center>Listing 3. Calculated Field Available</center>

Note that the TQuery is created, the SQL SELECT command added, executed and freed inside the OnCalcFields event. This is not exactly the most efficient way to determine the availability of the CD or DVD.

A faster way is to use the qBorrowCdDvd TQuery that we defined earlier. Remember that this is a parameterized SQL statement, which means that once prepared, it will execute very fast – even for different values of the parameter. There is no overhead in creating of freeing the TQuery, and it can be closed, new parameter value specified and re-opened quite efficiently. So for a faster implementation of the OnCalcFields event, see Listing 4.

```
void __fastcall TDataModuleBDE::tbCdDvdCalcFields(TDataSet *DataSet)
{
  qBorrowCdDvd->Active = false;
  qBorrowCdDvd->ParamByName("CdDvdID")->AsInteger =
    DataSet->FieldByName("CdDvdID")->AsInteger;
  qBorrowCdDvd->Active = true;
  if (qBorrowCdDvd->RecordCount == 0)
    DataSet->FieldByName("Available")->AsBoolean = true;
  else
    DataSet->FieldByName("Available")->AsBoolean = false;
  qBorrowCdDvd->Active = false;
}
```
<center>Listing 4. Calculated Field using Parameterized Query</center>

With this as last code for our example this section, we have enough to build the first example CD / DVD borrow application. Without the borrow part actually (that's left for the next section, when we also migrate to dbExpress).

The last screenshot of this section shows the application (full source code is available in the accompanying archive with the registered version of this manual), showing the three TTables in three TDBGrids, as well as the result of the qBorrowCdDvd in the Available column of the big datagrid at the bottom of the form.

Note the calculated field Category, which is represented by a TDBLookupComboBox above the grid, or a drop-down combobox in the TDBGrid control itself.
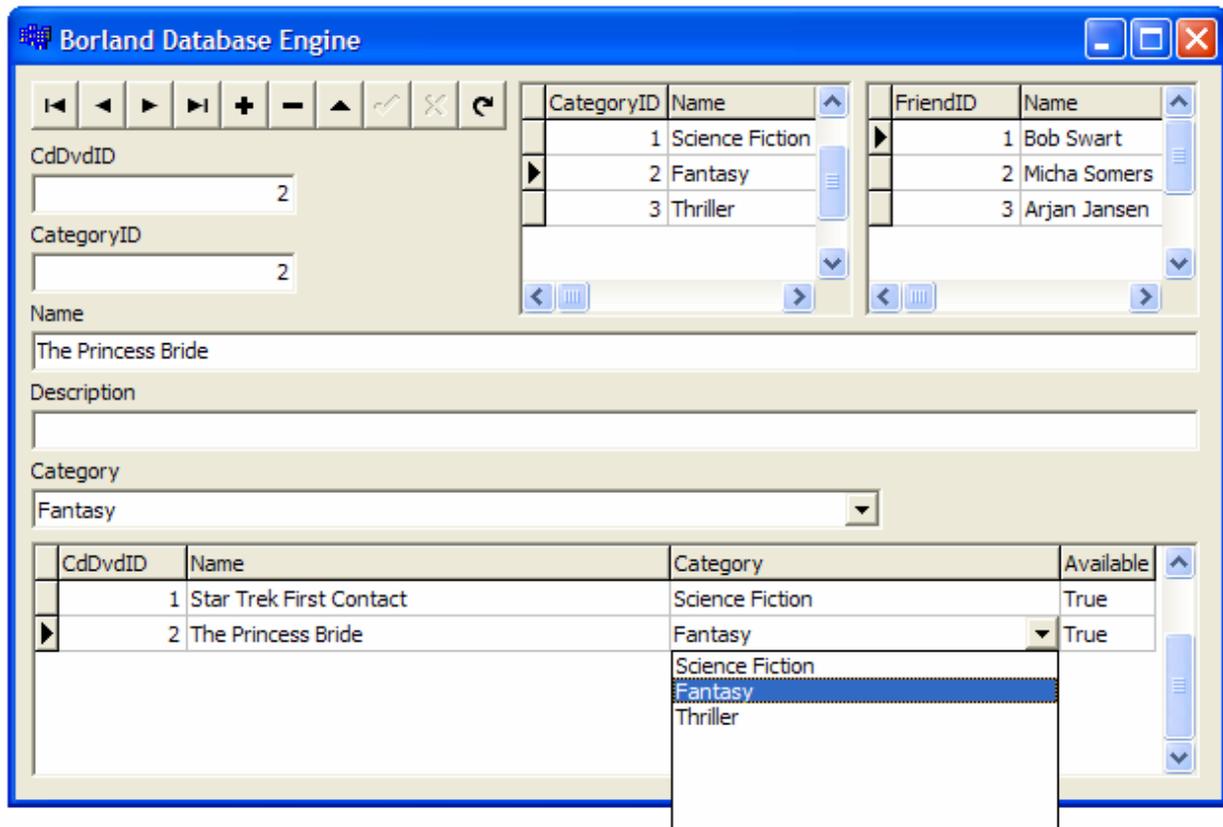

Figure 12. Final BDE Demo in Action

Using C++Builder 2006 or Turbo C++ 2006 you may experience a little problem trying to connect your TDataSource on the Main Form to a TTable or TQuery on the Data Module, This is caused by a bug in C++Builder 2006 / Turbo C++ 2006, but there is an easy workaround. You just have to drag one field from the Fields Editors (associated with the particular TTable or TQuery you want a TDataSource for) which will produce the TDataSource on the Form for you, along with a data-aware control and label for the field that you dragged but you can remove those again. Interestingly enough, it will also produce the error message of the data module not being used again, telling us that the Form is not using the Data Module (when in fact it is already), which clarifies the problem: the C++ VCL Designer is just unable to find the link or reference at design-time. Anyway, once created, the TDataSources on the Form will remain connected to the Data Module.

Summary

In this section, I have covered the Borland Database Engine and the data access as well as data aware controls that can be used with C++Builder. We've seen how to define persistent fields, lookup fields and calculated fields, and how to create dynamic tables using C++ code.
Local BDE tables are wonderful for small desktop database applications. However, they were never built to scale well in a multi-user (or multi-tier) environment, let alone the internet. Also, the BDE is available on Windows only (and not migrated to Linux for example). Finally, the BDE technology is frozen by Borland, and the big brother SQL Links is even declared deprecated, so should no longer be used at all.
In the next section, we'll examine ADO as one of the alternatives for connecting to SQL Server and Access databases (or generally any data source that can be accessed using ADO).

2. dbGo for ADO

Before we take a closer look at dbGo for ADO (previously known as ADOExpress), let's first explain ADO itself. ADO is the last Win32 incarnation of Microsoft's data access technologies, that started with ODBC and also included OLEDB (a COM-based set of interfaces designer to replace ODBC). ODBC was not a database engine, but more an access layer on top of a native database driver. Using ODBC you can connect to Access, SQL Server and just about anything that had an ODBC driver written for it. Microsoft introduced OLEDB to replace ODBC and work with the web as well. ADO (ActiveX Data Objects) is a higher level COM wrapper around OLEDB to make all this accessible for mere developers. In effect, ADO is still a data access layer that needs the physical database drivers to access the data. But from a developer point of view, using ADO you see an identical (programmer) interface whether you're connecting to Access, SQL 7, Oracle or any other database that has an ADO connection.

If you want the latest edition of Microsoft Data Access Components, then you can download MDAC 2.8 (or later) from Microsoft's Universal Data Access (UDA) website at http://www.microsoft.com/data/download.htm.

To quote the Microsoft website: *"Microsoft Data Access Components (MDAC) 2.8 contains core Data Access components such as the Microsoft SQL Server OLE DB provider and ODBC driver. This redistributable installer for the MDAC 2.8 release installs the same Data Access components as Microsoft Windows Server 2003."*

dbGo for ADO

The good thing about the VCL is that you seldom need to know the "nitty gritty details under the hood". In the case of ADO, it's no different, as Borland provides us with a set of components that look-and-feel almost lke other data access components. I say almost, because there are some differences and gotchas, so let's examine the dbGo for ADO components in some more detail. C++Builder contains seven dbGo for ADO components in the dbGo category of the Tool Palette.
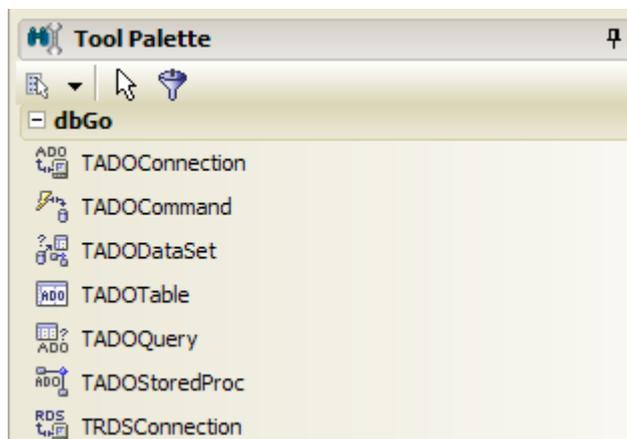

Figure 1. ADO on Component Palette

From top top bottom, we have TADOConnection, TADOCommand, TADODataSet, TADOTable, TADOQuery, TADOStoredProc and TRDSConnection. The last one can be used for multi-tier ADO applications, but will not be covered here. The others can be "mapped" to the regular BDE data access components we saw in the previous section, which makes migration of the components very easy. Migration of the data is another story, and will be covered later in this section.

TADOConnection has the same purpose as the TDataBase component. TADOTable is similar to a TTable, TADOQuery resembles a TQuery and TADOStoredProc is equivalent to a TStoredProc. The TADODataSet has no real BDE counterpart, but can be seen as a polymorphic equivalent to both a TTable and TQuery. Finally, the TADOCommand lacks a direct BDE equivalent, although the TUpdateSQL may come closest.

TADOConnection

Whenever you want to work with ADO in your application, it always starts with an ADOConnection component. This is the place where you "point" your connection to a specific OLE DB datasource, like Access or SQL Server (or any other OLE DB or ODBC datasource). Don't worry if you don't have those on your machine, because C++Builder includes an ADO-edition of the BCBDEMOS data, inside BCDEMOS.udl.

Place a TADOConnection component on a form or data module, and click on the ellipsis next to the ConnectionString property. This will start the ConnectionString "editor" (or just dialog) where you can either specify that you want to use a Data Link File (like BCDEMOS.udl as installed by C++Builder), or you wish to actually build a new Connection String yourself. In the next section, we'll connect to a real SQL Server database, this time we can start with the BCDEMOS.udl file.
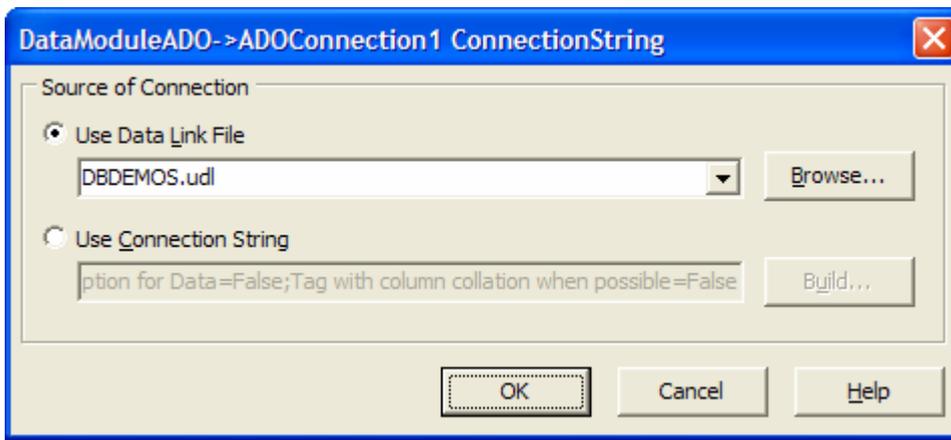
Figure 2. ConnectionString Editor

After you click on the OK-button to close the ConnectionString dialog, you can test the connection by assigning true to the Connected property. Since the LoginPrompt property is also set to true (by default), this will result in a login dialog. For the DBDEMOS.udl, Just leave the User Name and Password fields empty, and click on OK to open the connection. Obviously, you want to set the LoginPrompt property of the ADOConnection to false for further use.

Figure 3. Login Dialog

Once the connection has been verified, you can use any other the other dbGo for ADO data access components to actually work with the data. Although every ADO component has its own ConnectionString property as well, I recommend using a single ADOConnection component to assign to the Connection properties instead.

TADOTable

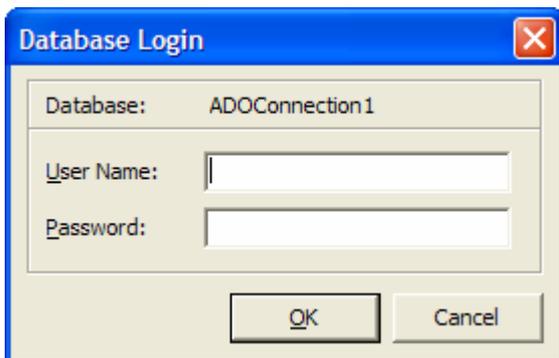An TADOTable component is very similar to a regular TTable component. Just place one on the form, and like anything using dbGo for ADO, assign the Connection property to the ADOConnection component from the previous example. This is equivalent to setting the Alias or DatabaseName of a regular Table component.

Once you've set the Connection property, you only need to select the actual table, which is done by - obviously - the TableName property. The following tablenames should be available in BCDEMOS: country, customer, employee, items, nextcust, nextitem, orders, parts and vendors. Note that the notorious biolife is not among them, but customer will do just fine for now.

Having set the Connection and TableName property, all that's left is the Active property. Once that has been set to true, the data is available, and can be displayed in the usual way using a TDataSource component and any data-aware component such as a TDBGrid or TDBEdits. As you can see, apart from the Connection property, working with ADOTables is very similar to working with regular Table components.

TADOQuery

Now, suppose you want to use a Query instead. Using ADO, that means placing a TADOQuery component. And as you would expect by now, the first thing you need to do is assign the Connection property. Next on the list is the SQL property, and after you've entered a query (like "SELECT * FROM ORDERS"), you can set the Active property to true to get results. Again, apart from the Connection property, this is very much like using a regular TQuery component.

TADODataSet

The TADODataSet component can act as either a Table or a Query (or even a Stored Procedure), depending on the value of its CommandType property. This property can be cmdFile, cmdStoredProc, cmdTable, cmdTableDirect cmdText or cmdUnknown. And based on the value of this CommandType property, the CommandText property will behave differently. If you set the CommandType to cmdUnknown, then ADO must actually parse the contents of CommandText to determine what kind of action to perform. For cmdTable or cmdTableDirect, you should enter a tablename inside the CommandText property, while for cmdText you should enter an SQL query in CommandText. In fact, the TADODataSet will display a Query builder, as can be seen in the following screenshot:
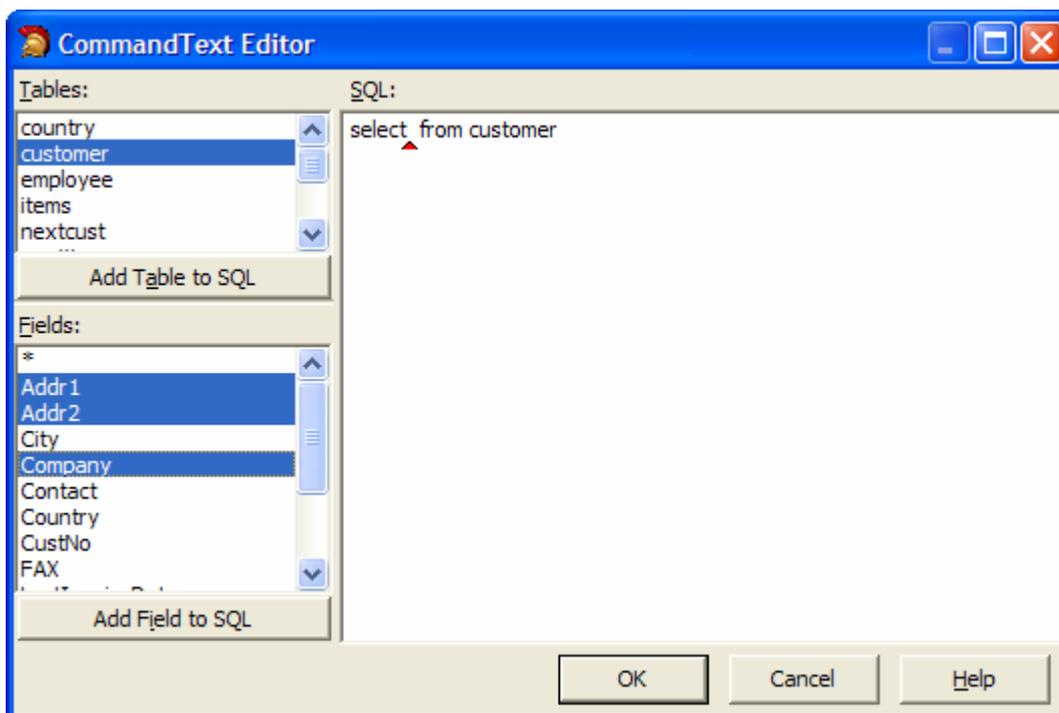

Figure 4. TADODataSet Query Editor

The difference in design-time support for the CommandText property become clear when you set CommandType to cmdText and click on the ellipsis for CommentText. This will show an SQL CommandText Editor.

If, on the other hand, you set the CommandType property to cmdTable (or cmdTableDirect) then the CommandText property will have a arrow (instead of an ellipsis) which will show a dropdown list of available tables.

Like any other ADO component, you can connect the ADODataSet component to all existing data-aware components, which makes it easy for you to "port" or migrate existing BDE applications to ADO.

Migration to ADO

ADO is Microsoft's way to access datasources in the Win32 world (for the .NET world, the name is ADO.NET, although this has very little in common with ADO). In this section, I've introduced the core dbGo for ADO components and explained how they can replace the BDE data access components.

Now, let's take that knowledge and use it to migrate the DVD application from using the BDE (local Paradox tables) to ADO. The process consists of two steps: first, need to migrate the data from the Paradox tables to MS Access or a real DBMS like SQL Server (which can be connected to using ADO). And then we need to modify the DVD application from using BDE to using ADO instead.

SQL Server / MSDE

Although MS Access is an easy to use table format that can be used with ADO, I've seldom used it (since in my view it offers little over the Paradox or dBASE local table formats). If we want to migrate to ADO, then why not make use of a real DBMS like SQL Server? Best of all, the SQL Server Desktop Engine (also called MSDE) is available for free from Microsoft.

Assuming you have SQL Server installed, we must first create a database with the table structure based on the data model. Unlike the BDE, there is no Database Desktop that you can use to define your tables. And while creating tables can be done in code, like we did for the Paradox BDE tables, using a real DBMS it's often more convenient to write a few SQL commands as follows to create your tables.

```
IF EXISTS (SELECT name
  FROM master.dbo.sysdatabases
  WHERE name = N'MyDVDs')
  DROP DATABASE [MyDVDs]
GO

CREATE DATABASE [MyDVDs]
GO

USE [MyDVDs]
GO

CREATE TABLE CDDVD (
  CdDvdID INTEGER NOT NULL,
  CategoryID INTEGER NOT NULL,
  Name VARCHAR(64) NOT NULL,
  Description VARCHAR(255),
    PRIMARY KEY (CdDvdID)
  )
GO

CREATE TABLE Category (
  CategoryID INTEGER NOT NULL,
  Name VARCHAR(32) NOT NULL ,
    PRIMARY KEY (CategoryID)
  )
GO
```

```
CREATE TABLE Friend (
  FriendID INTEGER NOT NULL,
  Name VARCHAR(64) NOT NULL,
  Description VARCHAR(255),
  Contact VARCHAR(10),
    PRIMARY KEY (FriendID)
  )
GO


CREATE TABLE Borrow (
  CdDvdID INTEGER NOT NULL,
  FriendID INTEGER NOT NULL,
    PRIMARY KEY (CdDvdID, FriendID)
  )
GO
```

Note that the above SQL script drops the database when run again, so you can use it to "start over" at any time if you wish. Also note, perhaps even more important, that you have to "use" the right database before you create the tables, otherwise they'll be created in the default database (which might not be the database that you've just created).

Given a file MyDVDs.sql containing the above SQL script, you can run it using the command-line tool osql as follows:

```
osql –E . –i myDVDs.sql
```

ADO Connection

Once the empty MyDVDs database is created, we can write a little application to migrate the actual data from the Paradox tables to the SQL Server database. We already know how to access the Paradox tables. So let's see how we can get to the MyDVDs SQL Server database.

For this, we need the TADOConnection component covered previously. This time we need to build a ConnectionString to the MyDVDs database. When you click on the Build button in the ConnectionString Editor, you'll get a new dialog where you must first select the data we want to connect to (which determine the drivers that are used), as shown below.
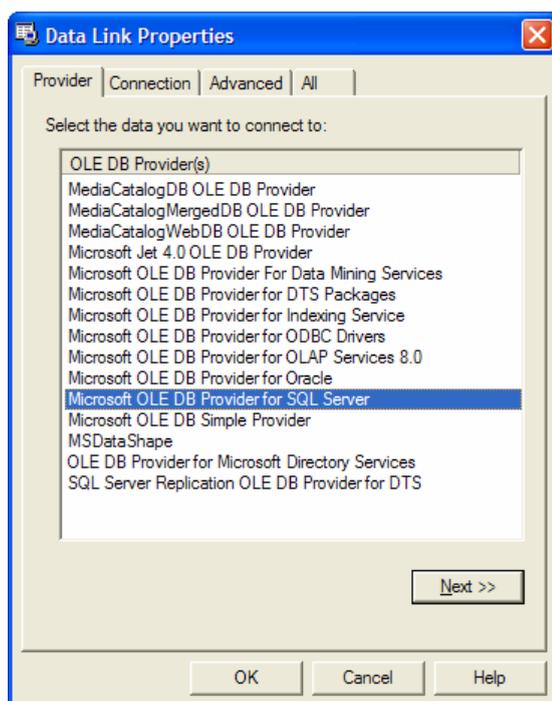

Figure 5. Data Link Properties Dialog - Provider

We must select the Microsoft OLE DB Provider for SQL Server here. On the Connection tab, we can define the Connection properties, like the name of the server (which can be a dot or local for the server on the local machine), the username (like sa) and corresponding password, or the option to use Windows integrated security. Note that the latter will work as long as your ADO connection is used in a GUI application, i.e. an application that is started by someone logged into Windows. When you want to use the same as a web application, you might encounter problems (but that's a story for another day).

The final option is the name of the database, which should be MyDVDs (this is one of the available databases that you can select in the drop-down combobox below).
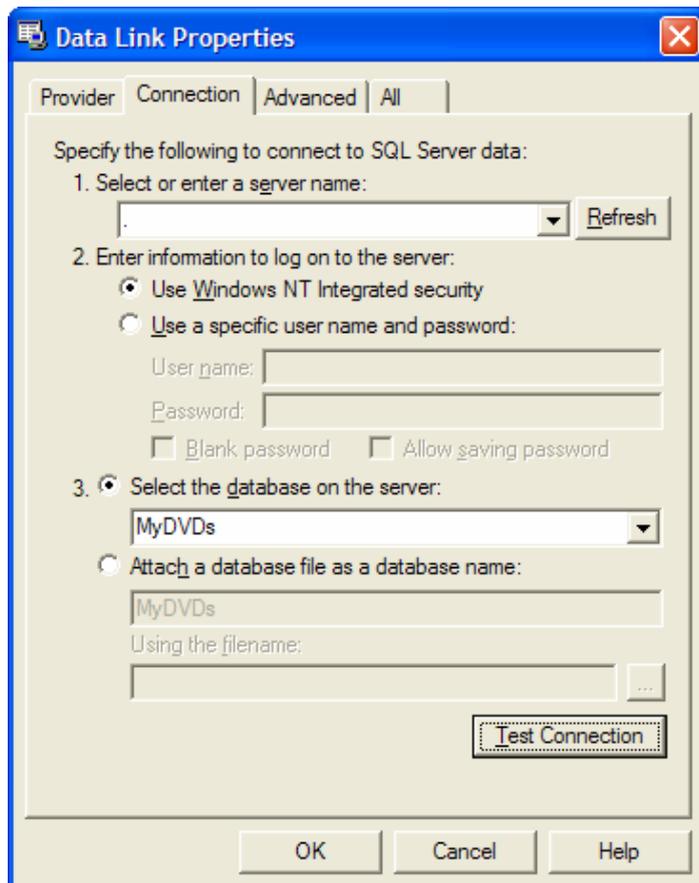
Figure 6. Data Link Properties Dialog - Connection

Note that it's always a good idea to test the connection while you're still in the dialog, so you can make necessary changes here if needed. Finally, set the LoginPrompt property of the TADOConnection component to false, so you won't have to click OK on the dialog each time.

Migrating Data

Once we have a connection, it's time to write some code to open a BDE TTable and move the records over to an ADO TADODataSet.

This can be done with the following code that I've used in a little migration application:

```
ADOConnection1->Open();
TADODataSet* ADODataSet = new TADODataSet(NULL);
ADODataSet->CommandType = cmdTable;
ADODataSet->Connection = ADOConnection1;
TTable* Table = new TTable(NULL);
Table->TableType = ttParadox;

const int NumTables = 4;
String TableNames[NumTables] = {"Category", "CDDVD", "Friend", "Borrow"};
```

```
  for (int tableno=0; tableno<NumTables; tableno++)
  {
    Table->TableName = TableNames[tableno] +  ".db";
    Table->Open();
    ADODataSet->CommandText = TableNames[tableno];
    ADODataSet->Open();
    while (!Table->Eof)
    {
      ADODataSet->Append();
      for (int i=0; i<Table->FieldCount; i++)
        ADODataSet->FieldByName(Table->FieldDefs->Items[i]->Name)->Value =
          Table->FieldByName(Table->FieldDefs->Items[i]->Name)->Value;
      ADODataSet->Post();
      Table->Next();
    }
    Table->Close();
    ADODataSet->Close();
  }
  Table->Free();
  ADODataSet->Free();
  ADOConnection1->Close();
```
<center>Listing 1. Migrating BDE to ADO</center>

Note that I've used a fixed array of the four tablenames to migrate, using the tablename plus the .db extension to load the Paradox file, and just the tablename to open the ADODataSet (with the CommandType set to cmdTable). For each record in the BDE table, we do an Append of the TADODataSet, copy all fields (using the FieldByName and Value properties), and finally posting the new records. Since ADO is also using a "connected" approach (just like the BDE), any update that we perform on the TADODataSet is immediately reflected in the database itself as well.

Migrating Data Module

When the data is migrated, we no longer have empty tables but can continue with the data that has been inserted using the BDE application.

The BDE application itself was built with migration in mind: we used a main form with the TDataSource and data-aware controls, but the actual (BDE) data access was done in a data module. So, in order to migrate from the BDE to dbGo for ADO, we need to start by adding another data module to the application. This one will be called DataModuleADO.

Now, place one TADOConnection and five TADODataSet components on the new data module. Make sure the TADOConnection points to the MyDVDs SQL Server database, and give the five TADODataSets the same name as the three TTables and two TQueries from the BDE application.
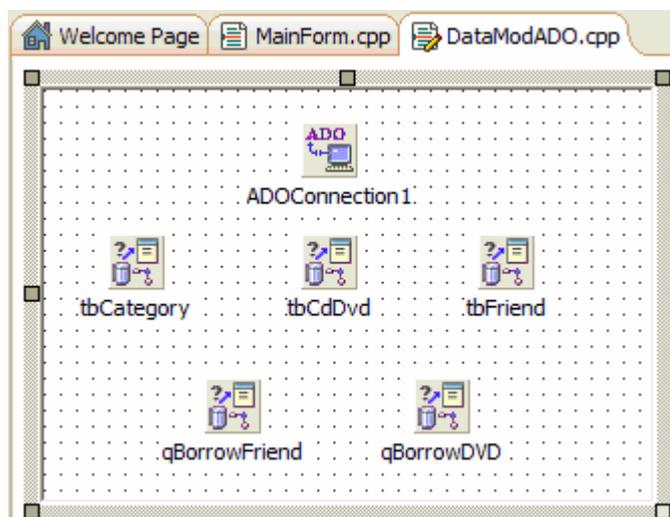

Figure 7. DataModuleADO

Note that although I've said earlier in this section that the TADOTable and TADOQuery are provided for easy migration from the BDE to ADO, I still prefer to use the TADODataSet whenever possible (which can replace both TTable and TQuery).

All TADODataSets have their Connection property pointing to the TADOConnection. And the first three (tbCategory, tbCdDvd and tbFriend) can have their CommandType set to cmdTable and their CommandText to the Category, CdDvd and the Friend table respectively. However, this is still using the old paradigm of a "table" component, which is really something from the (local) BDE world that we should avoid. When using a real DBMS, we should be using SQL all the way. Including in situations where we "only" want to see all fields from all records from a table. This can simply be translated to a "SELECT * FROM TABLE" query (which is what will happen behind the scenes if you open a table using the CommandType cmdTable, but I'd rather to it explicitly).

Using the TADODataSet component, you have a choice: either use the CommandType cmdTable and specify a tablename in the CommandText property, of use the CommandType cmdText and specify an SQL query. For the latter, we can even use the SQL Command Builder. This dialog has the benefit of showing all tables and fields but is only available for the TADODataSet, not for the TADOQuery (the latter only offers a TStrings editor for entering a SQL string, in which case you need to know the table names and field names yourself – another reason why I always prefer the TADODataSet over the TADOQuery).
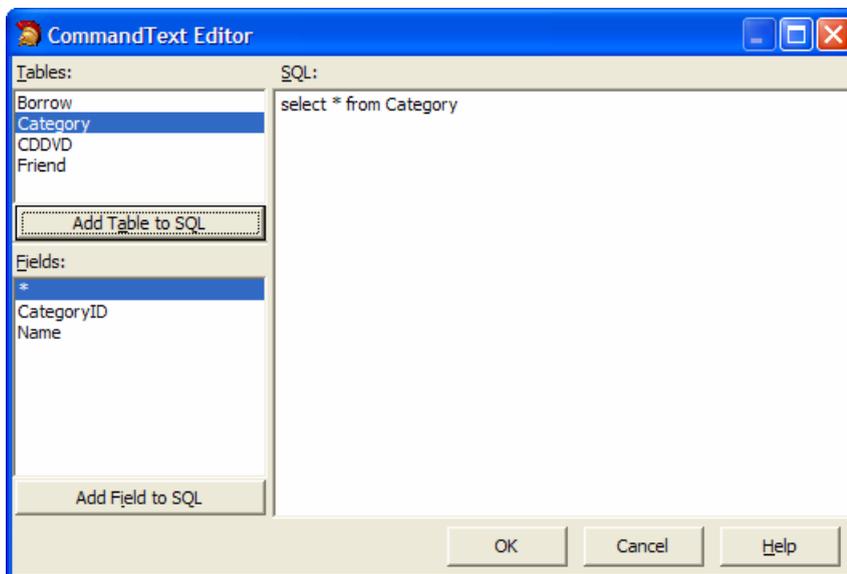

Figure 8. CommandText Editor

Both the qBorrowFriend and qBorrowCdDvd TADODataSets have their CommandType set to cmdText, so they act as a SQL query. Their CommandText can be copied from the SQL property of the corresponding TQuery components on the BDE data module. There is a little difference when we define the parameters: the dialogs for the ADO TParameter looks a little bit different than the one for the BDE TParam. This should be a first indication that ADO query parameters need some attention. Stay tuned for more details on this.

Once the five TADODataSets have been configured, we can double-click on them to start the Fields Editor and add all fields as explicit persistent fields – just like we did with the BDE TTables and TQueries on the BDE data module.

For the tbCdDvd TADODataSet, we also need to add two additional fields: one for the Category Name, and another called Available. The Category Name is a simple lookup field, connecting the CategoryID of the CdDvd table to the CategoryID of the Category table, showing the Category string field.

The Available field is a calculated field. For this one, we need to write some code in the OnCalcFields event of the tbCdDvd dataset. For the BDE application, we can assign a value to a parameter by using the Table->ParamByName->AsInteger approach, but that doesn't work for ADO parameters. For one thing, the AsInteger isn't known. But more importantly, the ParamByName doesn't exist as part of the ADO DataSet anymore.

To allow the code to compile, we have to change the original left-hand side of the expression from the BDE version:

```
qBorrowCdDvd->ParamByName("CdDvdID")->AsInteger
```

to the longer dbGo for ADO version:

```
qBorrowCdDvd->Parameters->ParamByName("CdDvdID")->Value
```

This is caused by the fact that the ParamByName method is introduced in the TDBDataSet class, defined in the dbtables unit, which is BDE specific. The original TDataSet class doesn't know about the ParamByName, and hence neither does our ADO data module. The actual parameter type for the BDE is a TParam (defined in the DB unit), while ADO uses a TParameter class (defined in ADODB) Apart from the parameter types, everything else works just the same, including the normal TDataSet FieldByName method, so the complete code in the OnCalcFields event handler is as follows:

```
void __fastcall TDataMod-uleADO::tbCdDvdCalcFields(TDataSet *DataSet)
{
  qBorrowCdDvd->Active = false;
  qBorrowCdDvd->Parameters->ParamByName("CdDvdID")->Value =
    DataSet->FieldByName("CdDvdID")->AsInteger;

  qBorrowCdDvd->Active = true;
  if (qBorrowCdDvd->RecordCount == 0)
    DataSet->FieldByName("Available")->AsBoolean = true;
  else
    DataSet->FieldByName("Available")->AsBoolean = false;
  qBorrowCdDvd->Active = false;
}
```
                    Listing 2. CdDvd Available Calculated Field

Once the ADO Data Module compiles, we can replace the original (BDE) data module with the ADO Data Module. In the main form, remove the reference to the original data module, and add the ADO Data Module. Then, we only have to make sure the three TDataSources are now pointing to the Category, Friend and CdDvd datasets from the ADO data module. The result will look the same as the local BDE version:
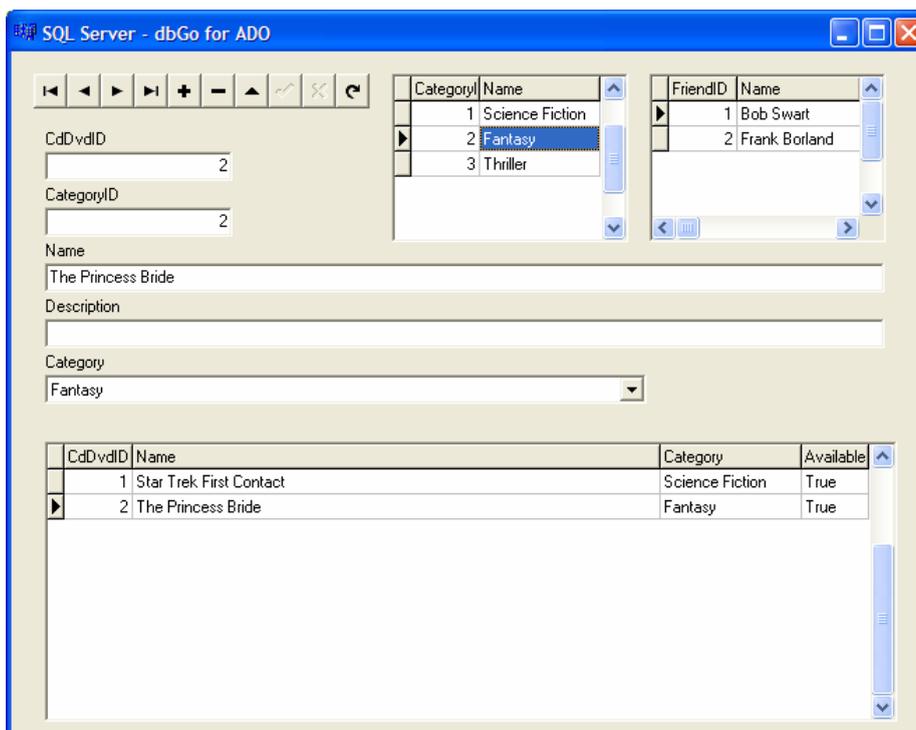

Figure 9. ADO Version of the DVD Application

Stored Procedures

I promised you an example using a Stored Procedure. A stored procedure is like an SQL command that is running in the DBMS itself, where you only have to call it by name, passing arguments (if any) and using the result. I often use Stored Procedures to insert new records, which enables the stored procedure to deal with new (unique) key values, etc. But you can also use a Stored Procedure to perform tasks which are better down by the DBMS, or just anything that you don't want to code inside your C++ application, but want to leave in the DBMS itself.

Creating a Stored Procedure is similar to creating a new table: you can run an SQL script, for example the following one to count the number of friends who have borrowed one or more DVDs.

```
USE [MyDVDs]
GO


DROP PROCEDURE FriendDVDs
GO


CREATE PROCEDURE FriendDVDs
  @FriendID int AS

SELECT COUNT(*)
FROM Borrow
WHERE FriendID = @FriendID


GO
```

As you can see, the definition of a Stored Procedure starts with the name and the list of parameters and their types. The implementation contains of a SQL command that uses the parameters, and optionally returns a result. In this case, the result is a single value, but you can also return one or more records (if the stored procedure contains a SELECT command), or return nothing (for example if the stored procedure performs an INSERT, DELETE or UPDATE).

Running this script to create the stored procedure in the SQL Server database can be done with OSQL just like we did with the first script to create the database and the four tables.

We can call (or "invoke") this Stored Procedure by either using a TADOStoredProc component, or by using the TADODataSet acting as a Stored Procedure. It should be no surprise by now that I'll select the TADODataSet with the CommandType property set to cmdStoredProc. I've called it spFriendDVDs and placed it on the DataModuleADO.
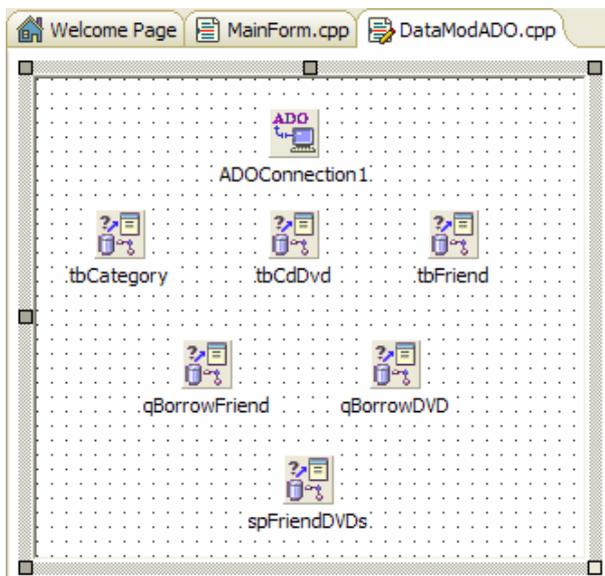

Figure 10. DataModule ADO

Once the CommandType is set, the CommandText will turn into a drop-down combobox. Not to show you the available tables in the database (which will be the result when CommandType is set to cmdTable), but to show the available Stored Procedures. Which should be just one now, FriendDVDs. Apart from that name, the drop-down combobox will also show the number of expected parameter values, which turns out to be 1 (not a big surprise). It gets even better when you click on the ellipsis for the Parameters property to open up the Parameters Editor. This will show two parameters: one for the RETURN_VALUE, and the other with the FriendID. That's very helpful, resulting in the Stored Procedure being ready to use right away.
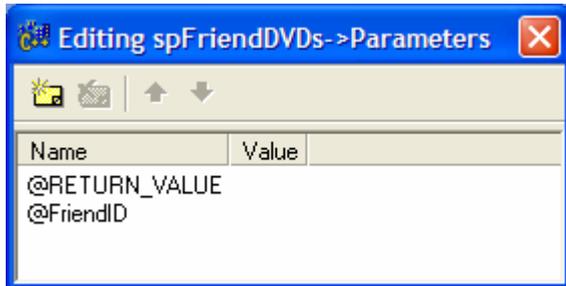

Figure 11. Stored Procedure Parameters

Calling the Stored Procedure is now easy: just assign a value to the @FriendID parameter and then execute the Stored Procedure (by setting the Active property to true) and reading the @RETURN_VALUE parameter. This is demonstrated with the following code, using the current selected Friend from the tbFriend dataset to pass the FriendID value as value for the @FriendID parameter.

```
void __fastcall TForm1::btnBorrowedClick(TObject *Sender)
{
  DataModuleADO->spFriendDVDs->Parameters->ParamByName("@FriendID")->Value =
      DataModuleADO->tbFriend->FieldByName("FriendID")->AsInteger;
  DataModuleADO->spFriendDVDs->Active = true;
  ShowMessage(String("This friend has ") +
    DataModuleADO->spFriendDVDs->Parameters->ParamByName("@RETURN_VALUE")->Value
    + " borrowed DVDs");
  DataModuleADO->spFriendDVDs->Active = false;
}
                            Listing 3. Borrowed DVD?
```

Obviously, with no way to actually borrow DVDs, the result will be that all friends have no DVDs borrowed at this time. And all DVDs are still available (the result of the calculated field), but that will both change in the next section, when we add the ability to actually Borrow DVDs.

Summary
In this section, I've completed the dbGo for ADO coverage by migrating the data from Paradox to SQL Server, and the BDE data module to an dbGo for ADO data module. We've also seen how to use Stored Procedures, a feature only available in a real DBMS like Microsoft SQL Server / MSDE (and not in local BDE Paradox or dBASE tables).
In the next section, we'll take a little side step by examining the TClientDataSet component as a means to work with local data in the MyBase format. This will be a first step towards the dbExpress (and DataSnap) topic, which is using a disconnected approach to the database. Compared to the BDE and dbGo for ADO, which both use the connected approach, this is a huge difference that will force us to approach data access in a new way.

3. TClientDataSet

In this section we'll continue with the database develement topic with the TClientDataSet. Previously, I've covered the BDE (Borland Database Engine) as well as ADO and the dbGo for ADO components in C++Builder, using SQL Server / MSDE as our database. This time, we take a little step back and examine the TClientDataSet component. This is actually quite a powerful component that will be used for the next few parts in this courseware manual! First, this section, we'll cover the TClientDataSet in a stand-alone scenario, using a local file format (binary or XML). Then, next section, we'll see how the TClientDataSet is also used in a dbExpress scenario, connecting to SQL Server / MSDE again (or any of the other dbExpress-compatible databases for that matter), and after that C++Builder Enterprise developers can even move on to DataSnap (not covered in this manual), building multi-tier database applications with the TClientDataSet in the thin (or smart) clients.

Standalone TClientDataSet
Before we start, let me first explain the consequence of using a stand-alone TClientDataSet component. There are some compelling advantages, like: it's free, and it's fast, and no DBMS required. However, it's also essentially single user only, since the data is stored in local files and loaded in memory. If this is no problem for you, then come along as we turn the DVD project into a stand-alone application. Otherwise, just wait until the next section when we'll introduce dbExpress which will include database drivers again (and multi-user capabilities).
The TClientDataSet data format is also called the MyBase format, and can be binary or XML For a local use of TClientDataSet, the first step consists of migrating the data from the original database to the TClientDataSet format. The original database can be anything, from the BDE to our SQL Server / MSDE database (that we used with ADO and dbGo for ADO). And this time, we don't even have to write a single line of code to migrate the data!

Migrating Data
Open the application from the previous section, which still uses the dbGo for ADO components on the data module. First of all, save the project as LocalCDS. The original project still contains two data modules: DataModuleBDE and DataModuleADO. Since we've added some more features to the application, we should use the DataModuleADO as starting point now. The DataModuleADO contains six TADODataSets: three acting as table, two as queries, and one stored procedure. We need to place a TClientDataSet component next to them, to help migrate the data to the MyBase format.
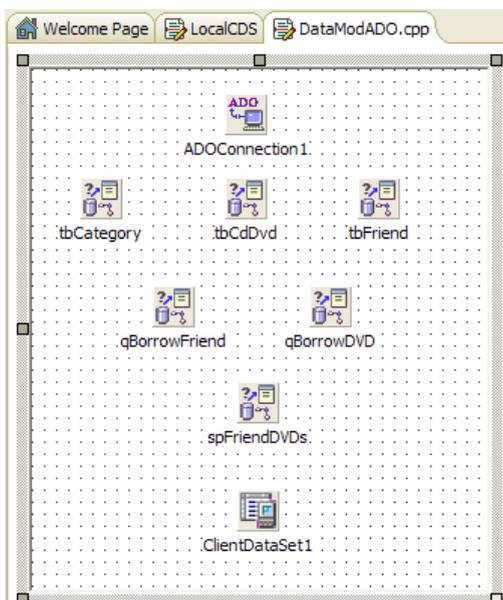

Figure 1. DataModule ADO

Right-click on the TClientDataSet component, and select the "Assign Local Data" option. This will give you a dialog where you can feed the TClientDataSet with the local data found in the other datasets:
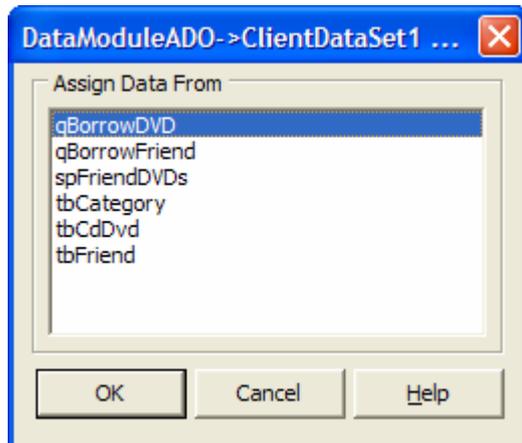

Figure 2. Assign Local Data

We need to ignore the stored procedure and two queries at this time, so start with the TTable called tbCategory; select this one and click on OK. As a result, the contents of the Category table will now be inside the TClientDataSet. Right-click on the TClientDataSet component again, but this time we need to save the data. We can save it as a binary MyBase file, or as an XML file in "normal" or the UTF-8 format. The latter is ideal when you're working with data containing special characters like umlauts. The binary format is the smallest of all, but can only be read by the TClientDataSet component (of C++Builder, Kylix and Delphi). And if the contents gets corrupted somehow, then you've lost your data. As a result, I only use the binary format when sending data from one place to another, but prefer to store the data in the XML format. It's bigger than the binary format, but at least anyone can read it. To cut a long story short: I've saved the contents of the TClientDataSet in an XML UTF-8 file called Category.xml. After this is done, we can assign local data again, this time to the tbCdDvd table, saving the result in CdDvd.xml, and finally do the same with the tbFriend table.

No Queries!?
A TClientDataSet is an in-memory dataset, which can result in great speed. However, it's equivalent to a Table component, and has no direct correspondence to a Query component. So in order to "translate" the two queries, we need to find another solution. Looking at the SQL commands, both queries select fields from the Borrow table, passing either a FriendID or a CdDvdID. SO instead of migrating the two queries to a TClientDataSet, we should migrate the Borrow table to the MyBase format instead. And even if no CDs or DVDs are borrowed at this time, the result will at least contain the meta information for the table.
In order to do this, we need to temporarily modify one of the query components to ommit the WHERE part, so we only perform a "SELECT * FROM Borrow". After that, we can again assign local data to the TClientDataSet component, and save the result in MyBase file Borrow.xml.
Finally, the Stored Procedure is also using the Borrow table, so we can try to reconstruct that functionality in a minute as well.

New Data Module
Remove the TClientDataSet from the DataModuleADO, as we no longer need it. We should now have four XML files with the contents of the SQL Server database. Time to reconstruct the data access components on an all-TClientDataSet data module.
Add a new Data Module to the LocalCDS project. Set its name to DataModuleCDS, and save it in DataModCDS.cpp.
We should now add four new TClientDataSet compoenents to the new data module, and call them cdsCategory, cdsCdDvd, cdsFriend, and cdsBorrow.
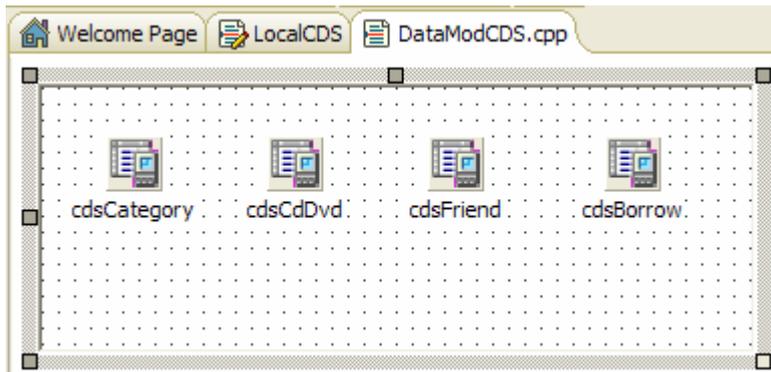
Figure 3. DataModule CDS

Instead of using the "Assign Local Data", we can now point these four components to the actual MyBase files, using their FilenNme property set to the four XML files (I'm sure you can figure out which file belongs to which component).

Note that once you've assigned a value to the FileName property using the "Open MyBase table" property editor (that you get when you click on the ellipsis for the FileName property), the FileName will be stored as fully qualified filename. This is not something that I fancy, to be honest, since it means the application is now using hardcoded paths to the database files. I prefer to use relative paths, so I can deploy the application to another machine as well. So, I always recommend to remove the path part from the FileName property, leaving only the local XML filename itself (in the current directory).

Adding Lookup Field

Before we can continue, we first need to add persisten fields to the TClientDataSet components, as well as additional lookup fields. This was first done in the section using the BDE, and is no different when using stand-alone TClientDataSets. We should add all fields, as well as a field called Category to the CdDvd dataset (see the BDE section for more details).
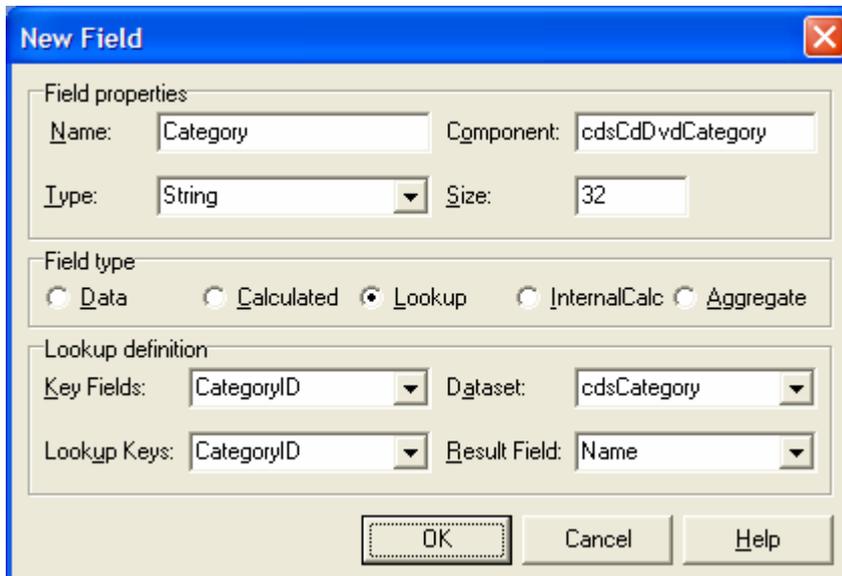

Figure 4. New Lookup Field

Note that if you compare the above screenshot with Figure 8 on Page 11, there are two additional field types available for a TClientDataSet (compared to a BDE TTable), namely the InternalCalc and Aggregate field type. These new types are only available for the in-memory TClientDataSet (but will be covered some other time, I'm afraid).

Adding Calculated Field

Apart from the lookup field Category, we should also add a calculated field called Available of type Boolean to the cdsCdDvd TClientDataSet.
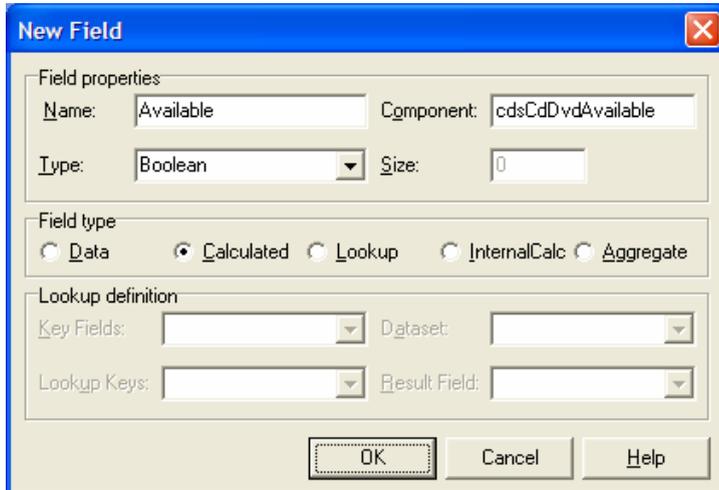

Figure 5. New Calculated Field

The original implementation of this calculated field used the query "SELECT * FROM Borrow WHERE CdDvdId = " followed by the actual CdDvdId, to determine if this DVD was still available. Since a TClientDataSet is an in-memory table only, with no SQL capabilities, this solution is no longer possible. Fortunately, we can still find out wether or not a CD or DVD is borrowed, by using the Filter property of a TClientDataSet component. In this Filter property, we can add an expression, which corresponds to the WHERE clause of a regular SQL SELECT command. This means that the implementation of the OnCalcFields event for the cdsCdDvd can be implemented as follows:

```
void __fastcall TDataMod-uleCDS::cdsCdDvdCalcFields(TDataSet *DataSet)
{
  cdsBorrow->Filtered = false; // just in case
  cdsBorrow->Filter ="CdDvdID = " + DataSet->FieldByName("CdDvdID")->AsString;
  cdsBorrow->Filtered = true;
  if (cdsBorrow->RecordCount == 0)
    DataSet->FieldByName("Available")->AsBoolean = true;
  else
    DataSet->FieldByName("Available")->AsBoolean = false;
  cdsBorrow->Filtered = false;
}
                    Listing 1. Available Calculated Field
```

Note that this code expects the cdsBorrow to be open in order to use the filter. Something which is not automatically the case. We could set the Active property of the TClientDataSets to true, which will open them by default, but not necessarily in the right order. The cdsBorrow *must* be active before we activate the cdsCdDvd. The only way to enforce that, is to do this in code, for example in the OnCreate event handler of the DataModuleCDS, as follows:

```
void __fastcall TDataModuleCDS::DataModuleCreate(TObject *Sender)
{
  cdsBorrow->Active = true;
  cdsCategory->Active = true;
  cdsCdDvd->Active = true;
  cdsFriend->Active = true;
}
                    Listing 2. Activating ClientDataSets
```

We should also close the TClientDataSets in the OnDestroy event handler.

Updates and Undo

A TClientDataSet is an in-memory dataset, which means that as long as the application using the TClientDataSet is up and running, all changes and modifications are made in memory only. If you accidently pull the plug of your computer, then these changes will be lost. Fortunately, if you close a TClientDataSet, then it will automatically save its contents, but only if the FileName property has been assigned (otherwise it wouldn't know where to save its contents).

The updates and changes to the clientdataset stored back in the MyBase file are not just simple updates, however. In fact, if you examine the contents of the XML files after you've made some changes, you'll notice that the XML files actually contain the original version, as well as all changes (inserts, updates and deletes). Step by step, so you can undo the changes as well, if you wish. In fact, the reason why the contents of the TClientDataSet is saved back to disk in the first place is only done if the ChangeCount property contains a value bigger than 0.

The fact that a MyBase file contains both the data and the delta (the changes) has advantages as well as disadvantages. The disadvantage is that the MyBase file will grow (with all subsequent updates), and will take increasingly longer to load.

The advantage is that it's now really easy to implement an "undo" feaure, which can be implemented using three different methods: UndoLastChange, RevertRecord and CancelUpdates. Calling UndoLastChange will – as the name indicates – undo the last change you've done. It has one argument, called FollowChange, which can be set to true in order to position the cursor in the dataset to the place where the change was un-done (which is quite handy when undoing changes in a TDBGrid for example). The RevertRecord on the other hand will revert the current record to its original value – this is the kind of undo that most customers will prefer. Finally, the CancelUpdates will cancel all updates that are currently stored in the delta.

Fortunately, there is a way to "merge" the delta with the data, so all changes are "accepted" and made final. This will shrink the MyBase file, and also remove the changes from the undo list. This can be done by calling the MergeChangeLog method of the TClientDataSet.

For my own applications, I prefer to be able to undo my changes as long as the application is up and running. So I use the delta at all times. However, as soon as I close the application, I want to store the smallest version of the MyBase file to disk, merging the delta and data. As a consequence, I can undo changes, until I close the application, but when I reopen it, the changes will no longer be undo-able. This can be done by writing some special code in the DataModuleCDS' OnDestroy method, as follows:

```
void __fastcall TDataMod-uleCDS::DataModuleDestroy(TObject *Sender)
{
  if (cdsFriend->ChangeCount)
  {
    cdsFriend->MergeChangeLog();
    cdsFriend->SaveToFile();
    cdsFriend->Active = false;
  }

  if (cdsCdDvd->ChangeCount)
  {
    cdsCdDvd->MergeChangeLog();
    cdsCdDvd->SaveToFile();
    cdsCdDvd->Active = false;
  }

  if (cdsCategory->ChangeCount)
  {
    cdsCategory->MergeChangeLog();
    cdsCategory->SaveToFile();
    cdsCategory->Active = false;
  }

  if (cdsBorrow->ChangeCount)
```

```
          {
            cdsBorrow->MergeChangeLog();
            cdsBorrow->SaveToFile();
            cdsBorrow->Active = false;
          }
        }
```
<p align="center">Listing 3. Merging/Saving/Closing ClientDataSets</p>

Note that after calling the MergeChangeLog method, we explicitly have to call SaveToFile, since the ChangeCount will return 0, so closing the TClientDataSet will no longer result in the file being saved automatically.

Connecting the GUI

When the CDS data module is fully configured, we can switch over to the MainForm.cpp to connect the GUI to the data module. There are three TDataSource components, currently connecting to the dbGo for ADO components, namely dsCategory, dsFriend and dsCdDvd. These should point to the cdsCategory, cdsFriend and cdsCDDvd from the CDS data module (note that you have to press Alt+F11 to use unit DataModCDS.

Figure 6. Local CDS at design-time

We can now add an Undo button, to undo changes made to the CdDvd TClientDataSet, for example, which can be implemented as mentioned before.

```
      void __fastcall TForm1::btnUndoClick(TObject *Sender)
      {
      //DataModuleCDS->cdsCdDvd->UndoLastChange(true);
        DataModuleCDS->cdsCdDvd->RevertRecord();
      //DataModuleCDS->cdsCdDvd->CancelUpdates();
      }
```
<p align="center">Listing 4. Undo options</p>

The RevertRecord method is the most convenient in my opinion.

Borrow DVDs

Time to add a button to borrow a DVD. This can be done by adding a record to the Borrow table, using the CdDvdId field from the current record in the cdsCdDvd dataset, and the FriendId from the current record in the cdsFriend dataset. The OnClick event handler of the Borrow button is implemented as follows:

```
void __fastcall TForm1::btnBorrowClick(TObject *Sender)
{
  DataModuleCDS->cdsBorrow->Append();
  DataModuleCDS->cdsBorrowCdDvdID->Value =  DataModuleCDS->cdsCdDvdCdDvdID->Value;
  DataModuleCDS->cdsBorrowFriendID->Value = DataModuleCDS->cdsFriendFriendID->Value;
  DataModuleCDS->cdsBorrow->Post();
}
```
<div align="center">Listing 5. Borrow CD or DVD</div>

Obviously, once a DVD has been borrowed, the avability field should return false, which is easy to test and verify. Unfortunately, the Available field in the TDBGrid is not immediately updated! The solution for this little issue is left as exercise for the reader, and will be discussed in the next section (when we also move on to dbExpress).

Figure 7. Local CDS in Action!

Local XML Files

There's one other issue, which will become clear once you try to run the application. At design-time, the XML files are in the current directory (since I specified the FileName property of the TClientDataSet components using the FileName only, and not the path information). However, at run-time, the executable is loaded from another directory, like the Debug_Build directory. And the XML files will not be in that directory, of course.

So before you run or deploy the application, make sure you either use a fixed path to the XML files, or just make sure the XML files are in the same directory as the executable (or use some kind of configuration or .ini file to solve the issue). Just something to keep in mind...

Summary

In this section, I've migrated the SQL Server data to the TClientDataSet MyBase local table format. We've seen how we can mimic the behaviour of queries by using the TClientDataSet's Filter property, and experimented with the data and delta (changes) of the TClientDataSets. Finally, we've added the ability to actually borrow DVDs in our example application.

In the next section, we'll continue the coverage of the TClientDataSet component, but this time in combination with the dbExpress data access components. We'll see the same in-memory capabilities, but using a DBMS backend, which means a different way to handle updates and a possibility for multiple users to update the database at the same time!

4. dbExpress

In the previous sections, I've covered the BDE (Borland Database Engine) as well as ADO and the dbGo for ADO components in C++Builder, using SQL Server / MSDE as our database. Last time, I introduced the TClientDataSet in a stand-alone setting, and this time we'll again use the TClientDataSet component, but in combination with dbExpress.

Disconnected Data

The most important issue that dbExpress introduces to this courseware manual on data access, is the concept of disconnected data. This is a fundamental difference compared to the connected approach that we've used so far, where the dataset was talking directly to the physical table. Any change made to the dataset (like Post or Delete) would result in a modification of the actual data in the table as well. The only exception was the TClientDataSet, which acted as an in-memory dataset. With dbExpress, we never work directly on the database tables themselves, but will use a TClientDataSet as "cache" for our data.

dbExpress Components

If you look at the dbExpress tab of the Component Palette, you see a number of components that – at first sight - are not unlike we've seen and worked with before. There's the TSQLConnection component, which is used to connect to the database. Using dbExpress, you can connect to InterBase, SQL Server / MSDE, Oracle, DB2, Informix or MySQL. And more if you're willing to purchase a third-party dbExpress driver.

The TSQLConnection connects to the database, and all other dbExpress components connect to the TSQLConnection component. We have the TSQLTable, TSQLQuery and TSQLStoredProc components, for which the name should be enough to define what they're doing. However, even in my dbExpress applications, I must admit that I seldom use any of these three components. Instead, I prefer to use the TSQLDataSet component, which can act as either a Table, Query or Stored Procedure of its own. The TSQLTable, TSQLQuery and TSQLStoredProc are probably only available to help migrate existing BDE (or ADO) applications to dbExpress. There's also a TSQLClientDataSet component, which is meant for use in small applications only. We get back to that one later. Finally, there's a TSQLMonitor component that can be used to monitor some of the traffic between the TSQLConnection and the actual DBMS. This can sometimes be helpful to see what SQL query is going on behind the scenes.

Migrating to dbExpress

As a practical example, let's migrate one of the previous versions of the DVD project over to use dbExpress. We may not have to migrate the data, since we already moved the data from local BDE tables to SQL Server in the ADO part.

Migrating DVDs

The only downside about using the SQL Server / MSDE database is that I had just entered a few hundred DVDs in my local TClientDataSet solution from last time, while the SQL Server / MSDE database is still practically empty. So, let's build one small data migration routine, to go from local TClientDataSet MyBase format to dbExpress.

Since we need to do this migration only once, let's just add a new form to the project and save it as CDS2DBX.cpp. From the dbExpress category on the Component Palette, place a TSQLConnection component on this new form, and double-click on the TSQLConnection to start the dbExpress Connections editor. The amount of available drivers that you see will depend on the version of C++Builder (Professional only has drivers for InterBase, MSSQL and MySQL, if I'm correct, while Enterprise adds Oracle, DB2 and Informix).

Select the MSSQLConnection (or add a new connection for MSSQL), enter a dot as HostName, MyDVDs as DataBase, and specify the User_Name and Password, or set OS Authentication to True, so you do not have to specify a named User_Name and Password (using MSDE, the default option is to use OS Authentication by the way).
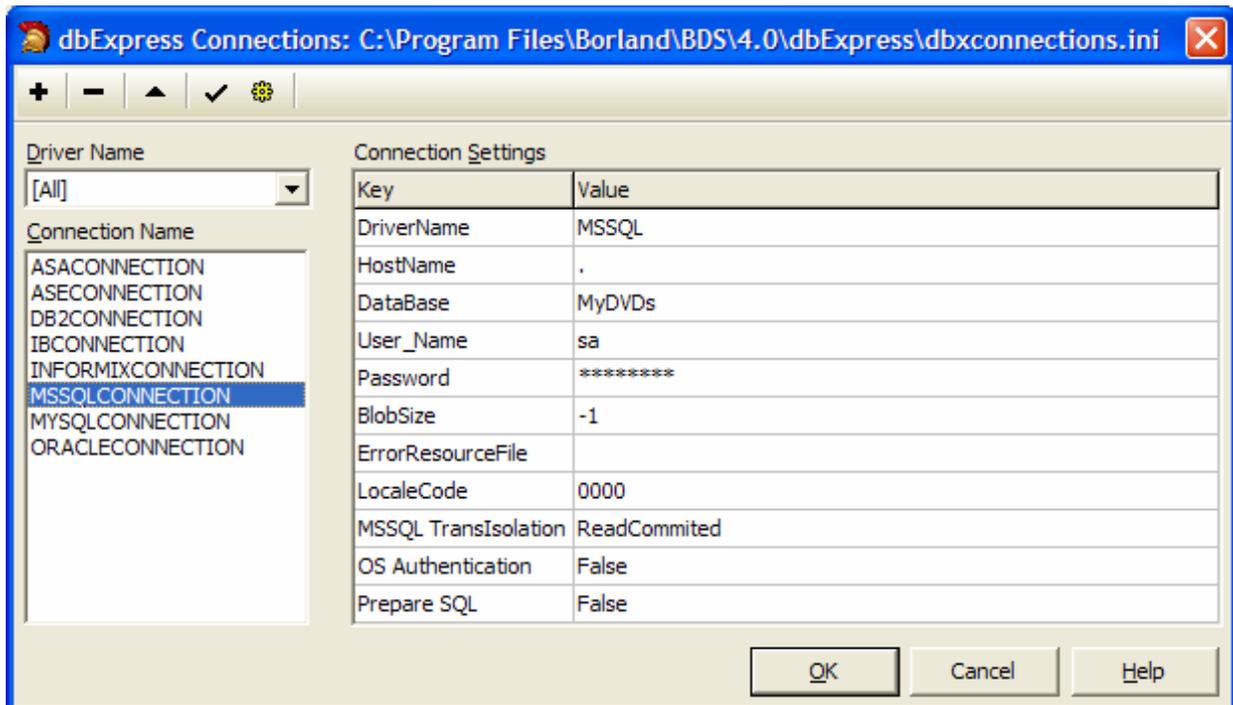
Figure 1. dbExpress Connections Editor

You can test the connection here, or using the Connected property of the TSQLConnection component itself (once you've closed the dbExpress Connections dialog). Make sure to set the LoginPrompt property of the TSQLConnection component to false, otherwise you will still get the login dialog, even if the User_Name and Password are already defined.

Once we can connect to the MyDVDs database, we need a dataset to connect to it. As I mentioned before, my personal preference is the TSQLDataSet, so let's place that one next to the TSQLConnection component, and call it sqlCDDVD. All dbExpress components – except for the TSQLConnection – have a SQLConnection property that needs to point to a TSQLConnection component. After you've assigned the SQLConnection property, you can set other properties, like the CommandText. The value of the CommandText property will be interpreted based on the value selected for the CommandType property. CommandType is set to ctQuery by default, but can also be ctStoredProc or ctTable. Using ctStoredProc or ctTable, the CommandText property will offer you a drop-down combobox where you can select the name of a Stored Procedure or Table. Using ctQuery, the CommandText property offers a Query CommandText Editor, showing the available tables and their fields.

By the way, for the DBMS, there isn't much difference to a ctTable or ctQuery, since using ctTable, it will convert the specified CommandText table name into a "select * from <table>" query, where <table> is the value of the specified CommandText table name (you can verify this using the TSQLMonitor component, which we'll do near later).

CommandText Problem

For our example, we need to talk to the CDDVD table, so let's set CommandType to ctTable, and open up the drop-down combobox for the CommandText property. This will show you the list of available tables, from which we can set CommandText to CDDVD. Set the Active property to true to verify that we can open this table using dbExpress.

Read-Only & Unidirectional

Once the TSQLDataSet is active, we can investigate the current contents of the CDDVD table. This may be the time to tell you that all dbExpress datasets (TSQLDataSet, TSQLTable, TSQLQuery and TSQLStoredProc – if returning a dataset) are very special, in that their contents is read-only and unidirectional. This means that you can read the records in a TSQLTable, but only from the first one to the last one (you cannot navigate back), and you cannot make any changes to this data.

These special features are a direct result of the disconnected nature of dbExpress that I briefly mentioned at the start of this section. Since the dbExpress datasets are disconnected from the database, it means that as soon as we activate them, a SQL command (or name of the Stored Procedure) will be sent to through the TSQLConnection to the actual DBMS, and the result of this command will then be received back and placed inside the dbExpress dataset. And nothing more. The connection (through the TSQLConnection) can then be broken, and the entire DBMS can disappear from the face of the earth. It's like a snapshot. Or in fact, it's more like SQL the way it was always intended: execute a query and give me the results. The whole notion of a live-connected dataset., where you can walk back and forth through your dataset and make changes where ever you wanted is perhaps a bit bloated (and also outdated).

As a little demonstration of the nature of the dbExpress datasets, place a TDataSource (called dsCDDVD) and TDBGrid (called dgCDDVD) on the new form, connect the TDataSource to the TSQLDataSet, and try to connect the TDBGrid to the TDataSource. That will give you an "Operation not allowed on a unidirectional dataset" exception (see Figure B).

Figure 2. Unidirectional Error

Since a TDBGrid can show more than one record at a time, it allows you to navigate from one record to another (and back), which is not allowed directly on a dbExpress dataset.

TClientDataSet to the Rescue

So, what good does a read-only, unidirectional dbExpress dataset do, if all we can do is read it from top to bottom? Well, that's where the TClientDataSet comes in – the smart and powerful in-memory dataset. If we can transfer the contents of the dbExpress dataset to a TClientDataSet, then we can view and navigate all we want.

In order to transfer the contents from the TSQLDataSet to the TClientDataSet, we must use a TDataSetProvider component (from the Data Access category of the Component Palette, where you can also find the TClientDataSet component). We need to connect the DataSet property of the TDataSetProvider to the TSQLDataSet, and then the ProviderName property of the TClientDataSet to the TDataSetProvider. Finally, we should make sure the TDataSource is now pointing to the TClientDataSet instead of the TSQLDataSet, and then we can connect the TDBGrid to the TDataSource without getting exceptions. In order to view data, we must also set the Active property of the TClientDataSet to true.

CDS to DBX

We now have one way of the equation: the connection to the CDDVD table inside the SQL Server / MSDE database. We should now place another TClientDataSet on the form, called cdsLocalCDDVD, pointing to the local MyBase file on disk that I already filled with hundreds of my DVDs. This TClientDataSet component only has to connect its FileName property to the CdDvd.xml file from last time.

Then, we need a Button, called btnMigrate, and a OnClick implementation which is as follows:

```
void __fastcall TForm2::btnMigrateClick(TObject *Sender)
{
  // First, clear entire CDDVD table in DBMS
  cdsCDDVD->Active = true;
  while (!cdsCDDVD->Eof) { cdsCDDVD->Delete(); }
```

```
      ShowMessage("Empty");

      // Open and copy local CDDVD data
      cdsLocalCDDVD->Active = true;
      cdsLocalCDDVD->First();
      while (!cdsLocalCDDVD->Eof)
      {
        cdsCDDVD->Append();
        cdsCDDVD->FieldByName("CdDvdID")->AsInteger =
        cdsLocalCDDVD->FieldByName("CdDvdID")->AsInteger;
        cdsCDDVD->FieldByName("CategoryID")->AsInteger =
        cdsLocalCDDVD->FieldByName("CategoryID")->AsInteger;
        cdsCDDVD->FieldByName("Name")->AsString =
        cdsLocalCDDVD->FieldByName("Name")->AsString;
        cdsCDDVD->FieldByName("Description")->AsString =
        cdsLocalCDDVD->FieldByName("Description")->AsString;
        cdsCDDVD->Post();
        cdsLocalCDDVD->Next();
      }
      ShowMessage("Full");

      // send data to the DBMS
      cdsCDDVD->ApplyUpdates(0);
    }
```
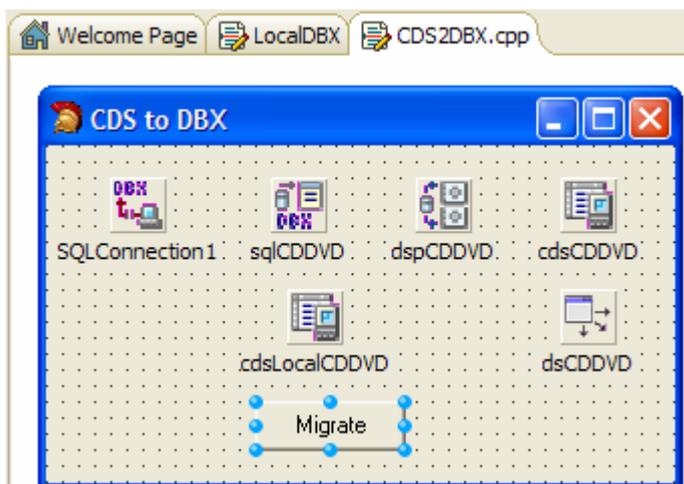   Listing 1. Migrate from CDS to dbExpress

Figure 3. CDS to DBX

The code in this event handler can be split into three parts. First, we need to clear the existing test data from the SQL Server / MSDE database, and we can do that by looping through the cdsCDDVD and calling the Delete() method until it's finally empty – or rather until we're at Eof. Note that calling Eof is more efficient than calling RecordCount, since the RecordCount will actually send a "select count(*) from <table>" to the DBMS, which is more costly than simply looking at the cursor of the TClientDataSet.,
The second part of the code consists of copying the contents of the cdsLocalCDDVD table to the cdsCDDVD table. This is again a loop, but this time using the cdsLocalCDDVD table as basis. Do not forget to call the Next() method, otherwise you may say in the loop until you get an out-of-memory exception or a key violation. Copying the actual record is done by copying the individual fields, using the names of the fields and AsInteger for the first two (Integer) fields, and AsString for the last two (String) fields.
The last part of the code may be new: this is where the cdsCDDVD TClientDataSet has to send its changes (both the deletion of the existing records and the addition of the new records) to the SQL Server / MSDE DBMS. Without that last line of code, the database would still be unchanged, and we would only have deleted and inserted records in memory.

The call to ApplyUpdates from the TClientDataSet will pass through the TDataSetProvider, which will generate a set of SQL DELETE and INSERT commands (and UPDATE commands when needed), and these will be passed right through the TSQLDataSet component to the TSQLConnection which will actually send them to the SQL Server / MSDE database. If all of these SQL INSERT, DELETE and/or UPDATE commands are accepted and executed without errors, then the call to ApplyUpdates has succeeded. Otherwise, the value of the parameter that we passed comes into play. A parameter value of 0 means that we will not tolerate any SQL error, and the update process should be prematurely terminated if an error occurs. This does not mean that all updates that succeeded will be rolled back, however, only that the updates will stop from the moment the first error occurs. The parameter of the ApplyUpdates method is called MaxErrors. If we specify a value bigger than zero – like 7 for example - for MaxErrors, then for each error the counter is decreased until the maximum number of allowed errors is reached, and the next error will again terminate all further updates.

ApplyUpdates and MaxErrors
In practice, I use only two different values for the MaxErrors parameter of the ApplyUpdates method: either 0 or -1. The former will stop as soon as one error is encountered, while the latter will ignore all errors, and will apply as many updates as possible.
If one or more errors are encountered, then a special event handler of the TClientDataSet will be called, namely the OnReconcileError event handler. Whether you handle it or not, the Delta property of the TClientDataSet – which holds all the pending insert, delete and updates – will still contain all non-applied updates and any errors that were encountered. Only the updates that succeeded are removed. As a consequence, you can always try to re-apply the updates, since they are never really gone (unless you explicitly remove them).

ChangeCount and Undo
This leads to another interesting issue: since dbExpress only works with disconnected data, we can treat the Delta property of the TClientDataSet as a cache that will hold our updates before they are actually applied to the DBMS. And in that cache, we should still be able to undo any changes that we've made to the data. This means that we can finally offer an undo capability (or several in fact), which would have been very hard to implement using a connected data access technology like the BDE or ADO (working directly on the database tables themselves). I'll get back to showing these in a minute, let's first create a dbExpress data module and migrate the main form.

dbExpress Data Module
Compared to a regular data module, you'll find that a dbExpress data module uses far more components. That's because each dbExpress dataset should also have a connected TDataSetProvider and TClientDataSet. If you don't want to use these three components, then there is a combined version called the TSQLClientDataSet component. This one should only be used for small-scale applications, however (at least that's what Borland keeps telling us). Feel free to experiment with the TSQLClientDataSet, however.
For our dbExpress data module, we need one TSQLConnection component, and then six TSQLDataSet components, called sqlCategory, sqlCdDvd, sqlFriend, sqlBorrowFriend, sqlBorrowDvd and finally sqlFriendDvd. They can all use CommandType set to ctQuery, with the exception of the last one, which should use a CommandType set to ctStoredProc.
We can now enter the SQL commands. They are as follows for the five queries:

TSQLDataSet	CommandText
sqlCategory	SELECT * FROM Category
sqlCdDvd	SELECT * FROM CdDvd
sqlFriend	SELECT * FROM Friend
sqlBorrowFriend	SELECT * FROM Borrow WHERE FriendID = :FriendID
sqlBorrowDvd	SELECT * FROM Borrow WHERE CdDvdID = :CdDvdID

The SQL commands are actually the same as we used with the ADO components.

The sqlBorrowFriend and sqlBorrowDvd need a parameter defined (the first one called FriendID and the second one called CdDvdID) of type ftInteger, with ParamType set to ptInput.

Once we have the first five TSQLDataSet components configured, we can place five TDataSetProvider components and five TClientDataSet components next to them on the data module. If you name them accordingly (with the dsp prefix for the TDataSetProvider and cds prefix for the TClientDataSet components), then the data module should look something like the following screenshot (a bit crowded).
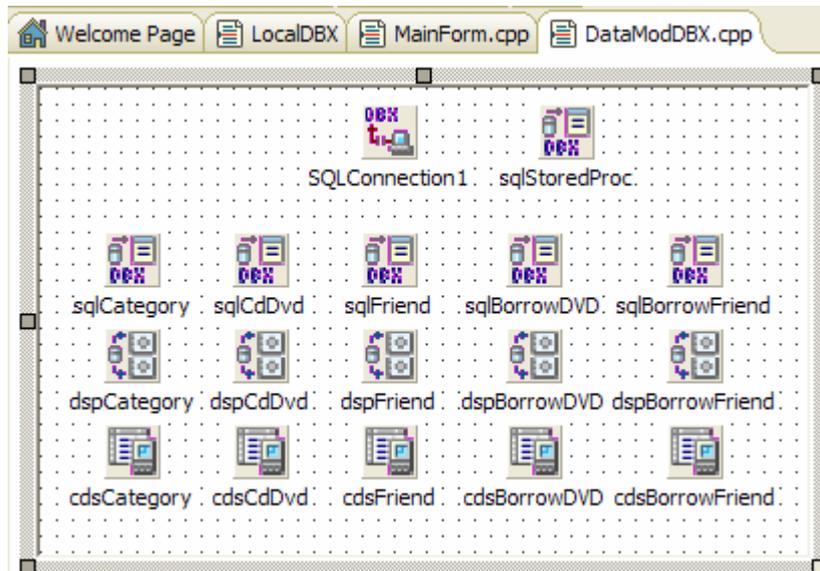

Figure 4. Data Module DBX

Additional Fields

Let's focus on the additional fields for the datasets. In the past few sections, we've been adding a lookup field (Category Name) and calculated fields (Available) to the CdDvd dataset. We should do the same thing using our dbExpress approach. However, as you may realize: there are two places where we can define these additional fields: at the dbExpress level (inside sqlCdDvd) or at the ClientDataSet level (inside cdsCdDvd). Since the lookup field will need to poke into another table, which I prefer to keep at the same level, it would be best to add the additional fields at the ClientDataSet level, where everything is cached and bi-directional (so we won't get any problems if we lookup a different category). So, double-click on the five TClientDataSet components to get the Fields Editor, and for each of them right-click in the Fields Editor and Select "Add all fields" to add all persistent fields.
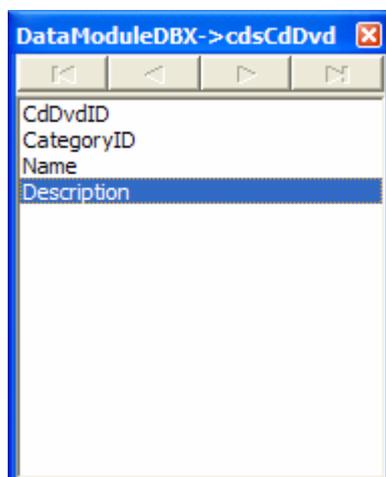

Figure 5. Fields Editor

This step will also verify that we can execute the SQL commands, by the way. Finally, for the cdsCdDvd, go back to the Fields Editor and add a new lookup field called Category, pointing to the Name field of the cdsCategory, using CategoryID as connecting foreign key field:

Figure 6. New Lookup Field

And we should also add a new calculated field called Available of type Boolean to the cdsCdDvd dataset, as follows:

Figure 7. New Calculated Field

We now need to write some code for the OnCalcFields event handler of the cdsCdDvd ClientDataSet, calculating the availability of the CD or DVD. This code is as follows:

```
void __fastcall TDataModuleDBX::cdsCdDvdCalcFields(TDataSet *DataSet)
{
  sqlBorrowDVD->Active = false;
  sqlBorrowDVD->Params->ParamByName("CdDvdID")->Value =
    DataSet->FieldByName("CdDvdID")->AsInteger;
  sqlBorrowDVD->Active = true;
  DataSet->FieldByName("Available")->AsBoolean = sqlBorrowDVD->Eof;
  sqlBorrowDVD->Active = false;
}
```
                     Listing 2. Available Calculated Field

Note that I'm using the dbExpress TSQLDataSet sqlBorrowDVD here, since that's the one with the query and the parameter. The corresponding TClientDataSet called cdsBorrowDVD doesn't know about the parameter, and only about the result of the query. Also note that compared to the earlier solutions using BDE, ADO or CDS that we've implementedbefore, I'm no longer checking the sqlBorrowDVD->RecordCount property to see if the query returns one record or more, but I'm checking the Eof property to see if we're at the end of the dataset. Since a dbExpress dataset is unidirectional, starting from the top, Eof right after we've activated a query would mean that the resultset is in fact empty. This is not only shorter, but also much faster (again, since the RecordCount could result in a "select count(*) from table" to be executed).

When we're done. Make sure to set the Active property of all TSQLDataSet components and all TClientDataSet components to false, as well as the Connected property of the TSQLConnection component. We'll activate everything at runtime, when the dbExpress data module is created, as follows:

```
__fastcall TDataModuleDBX::TDataModuleDBX(TComponent* Owner): TDataModule(Owner)
{
  cdsCategory->Active = true;
  cdsCdDvd->Active = true;
  cdsFriend->Active = true;
}
```
<div align="center">Listing 3. Activate DataSets</div>

Note that activating a TClientDataSet will force the underlying TSQLDataSet to be opened as well, which will force the TSQLConnection component to connect to the SQL Server / MSDE database. So there's a lot happening with one simple assignment here. Time to move to the main form, and reconnect the GUI.

Connecting the GUI
When the dbExpress data module is fully configured, we can switch over to the MainForm.cpp to connect the GUI to the data module. There are three TDataSource components that need to be connected to the TClientDataSet components on the data module, namely dsCategory, dsFriend and dsCdDvd. These should point to the cdsCategory, cdsFriend and cdsCdDvd from the dbExpress data module (note that you have to press Alt+F11 to use unit DataModDBX.
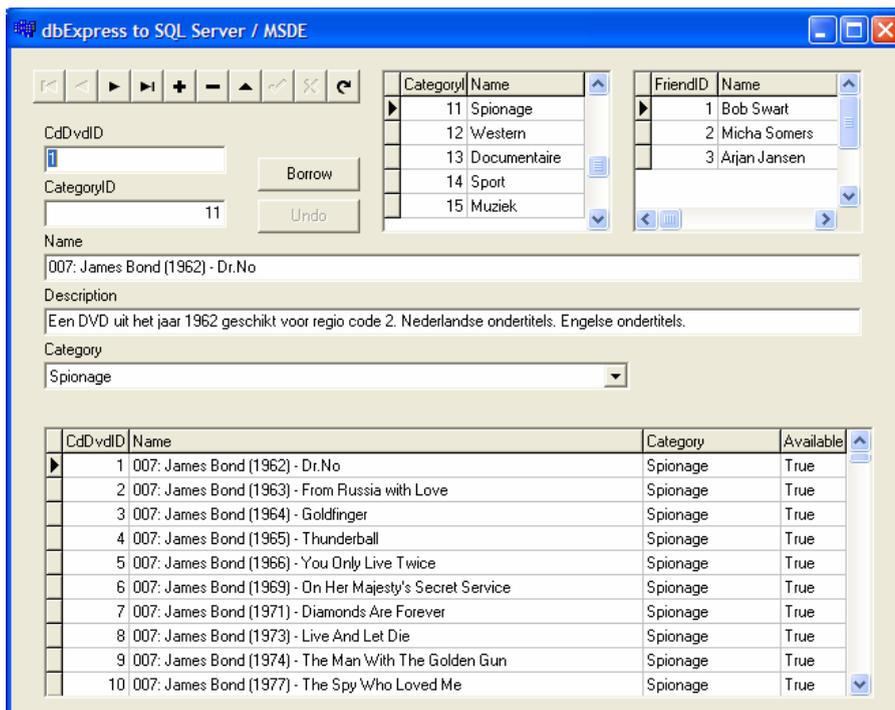

Figure 8. dbExpress to SQL Server / MSDE

Note that the Borrow and Undo buttons are already present, although we have to re-implement them using the dbExpress techniques.

Save Changes
First of all, however, we need to add a way to save all our changes made in the TlLientDataSet components back into the DBMS. Right now, if we make any change to the data using the application, the changes will be lost if we close the application (since the changes are only made to the in-memory TClientDataSet).
This has been discussed before, as has the solution: we must explicitly call ApplyUpdates to send the update, insert and/or delete commands to the DBMS. However, the question is: when do we call ApplyUpdates, and do we want to bother the enduser with this?

Update after Post / Delete
One possible solution would be to call ApplyUpdates after every change made in the TClientDataSet components. This means that we need to call ApplyUpdates in the OnAfterPost and OnAfterDelete event handlers of the cdsCategory, cdsCdDvd, and cdsFriend as follows:

```
void __fastcall TDataModuleDBX::cdsCategoryAfterDeleteorPost(TDataSet *DataSet)
{
  dynamic_cast<TClientDataSet*>(DataSet)->ApplyUpdates(0);
}
                    Listing 4. ApplyUpdates After Delete or Post
```

Note that this single event handler can be shared by all three TClientDataSet components, and in both their OnAfterPost and OnAfterDelete event handlers. The required implementation code is the same in all cases.

Update on Request
Although this may feel like a small and efficient solution, we're depriving ourselves of a very nice feature, available if we do not call ApplyUpdates right away: the ability to Undo changes still in the ChangeLog.
As a result, I often either place an explicit button on the form that says "Apply Updates", or call the ApplyUpdates method in the FormClose event handler. That way, we can perform undo (re-implemented in the next section), and still ensure that our changes are sent to the database as soon as we explicitly want, or as soon as the application is closed.
You can even show a dialog to verify to the end user that there is unsaved data, which should be saved (this is left as exercise for the reader).
The final calls to ApplyUpdates can also be placed in the DataModuleDestroy event of the dbExpress data module, as follows:

```
void __fastcall TDataModuleDBX::DataModuleDestroy(TObject *Sender)
{
  if (cdsCategory->ChangeCount)
    cdsCategory->ApplyUpdates(0);
  if (cdsCdDvd->ChangeCount)
    cdsCdDvd->ApplyUpdates(0);
  if (cdsFriend->ChangeCount)
    cdsFriend->ApplyUpdates(0);
}
                    Listing 5. ApplyUpdates on DataModule Destroy
```

Note that I'm checking to see if the ChangeCount is actually non-zero before calling ApplyUpdates (to see if there are any updates to apply in the first place).

Enabled Undo

Assuming we only call ApplyUpdates in the FormClose, we can rely on the availability of the Undo functionality calling UndoLastChange, RevertRecord or CancelUpdates.

The implementation of the Undo button, to undo changes made to the CdDvd TClientDataSet, for example, is implemented similar as before.

```
void __fastcall TForm1::btnUndoClick(TObject *Sender)
{
  DataModuleDBX->cdsCdDvd->RevertRecord();
}
```
<p align="center">Listing 6. Undo</p>

As enhancement to the earlier implementation, it would be nice to see if a record can be undone. For that, we can implement the OnDataChange event handler of the dsCdDvd TDataSource, which is fired as soon as the data changes (including after a scroll or navigation operation). In the OnDataChange event handler, we can call and examine the result of the UpdateStatus method from the cdsCdDvd. UpdateStatus returns the status of the current record in the TClientDataSet, and returns usUnmodified, usModified, usInserted or usDeleted.

The btnUndo button should be disabled if UpdateStatus return usUnmodified, and enabled in all other cases, which is implemented as follows:

```
void __fastcall TForm1::dsCdDvdDataChange(TObject *Sender, TField *Field)
{
  btnUndo->Enabled = (DataModuleDBX->cdsCdDvd->UpdateStatus() != usUnmodified);
}
```
<p align="center">Listing 7. Enable Undo Button</p>

Borrow DataSet

In order to implement the Borrow DVD button, we need to add another dataset to the dbExpress data module. Again, this will be a combination of three components: TSQLDataSet (called sqlBorrow), TDataSetProvider (called dspBorrow) and TClientDataSet (called cdsBorrow), all three connected to each other in the usual way. Enter "SELECT * FROM Borrow" in the CommandText property. Double-click on cdsBorrow and use the Fields Editor to add all (explicit) fields.
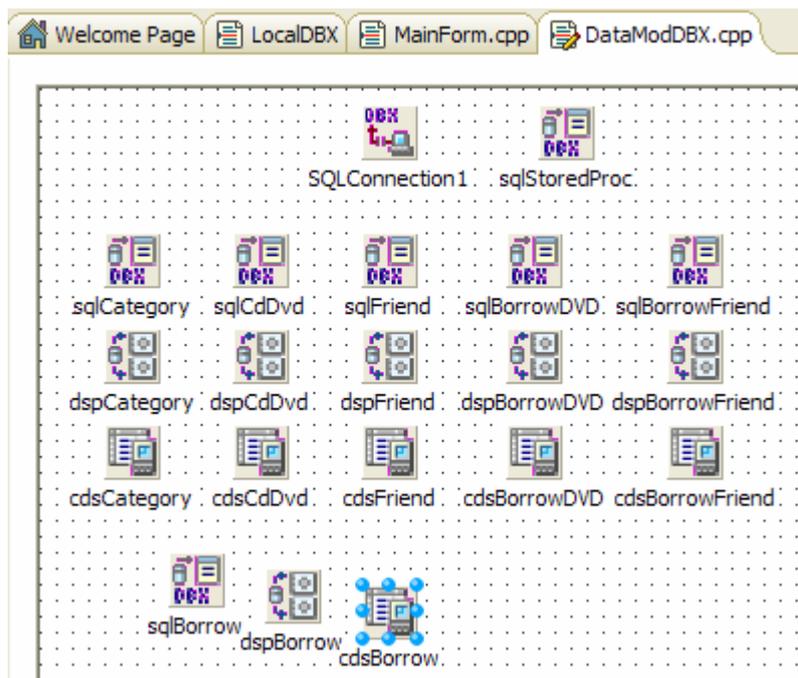

Figure 9. Borrow DataSet

The OnClick event handler of the Borrow button can now be implemented as follows:

```
void __fastcall TForm1::btnBorrowClick(TObject *Sender)
{
  DataModuleDBX->cdsBorrow->Active = true;
  DataModuleDBX->cdsBorrow->Append();
  DataModuleDBX->cdsBorrowCdDvdID->Value = DataModuleDBX->cdsCdDvdCdDvdID->Value;
  DataModuleDBX->cdsBorrowFriendID->Value = DataModuleDBX->cdsFriendFriendID->Value;
  DataModuleDBX->cdsBorrow->Post();
  DataModuleDBX->cdsBorrow->ApplyUpdates(0);
  DataModuleDBX->cdsCdDvd->Refresh();
}
```
<center>Listing 8. Borrow CD or DVD</center>

Note that I have to call the ApplyUpdates method of the cdsBorrow TClientDataSet to make sure the fact that the DVD is borrowed is known to the database right away. We then also have to call Refresh() of the cdsCdDvd->TClientDataSet, to ensure that the Availability status is updated correctly (implemented using a calculated field, looking at the actual Borrow table).

Apart from being able to Borrow a DVD, we should also place a "Returned" button, in case friends ever return my CDs or DVDs (it actually does happen from time to time). The implementation of the OnClick event handler of this button is similar to the Borrow one, only this time we have to find a record and remove it, before calling ApplyUpdates and Refresh again.

```
void __fastcall TForm1::btnReturnClick(TObject *Sender)
{
  DataModuleDBX->cdsBorrow->Active = true;
  DataModuleDBX->cdsBorrow->Filtered = false;
  DataModuleDBX->cdsBorrow->Filter = "CdDvdID = " +
    DataModuleDBX->cdsCdDvdCdDvdID->AsString;
  DataModuleDBX->cdsBorrow->Filtered = true;
  DataModuleDBX->cdsBorrow->Delete();
  DataModuleDBX->cdsBorrow->Filtered = false;
  DataModuleDBX->cdsBorrow->ApplyUpdates(0);
  DataModuleDBX->cdsCdDvd->Refresh();
}
```
<center>Listing 9. Return CD or DVD</center>

Note that I'm using the Filter property of the TClientDataSet to locate the Borrow record of the current DVD (assuming that a DVD can only be borrowed by one person at a time – I'm not a DVD store).

Who's Got It?
We have to remove the Borrowed button which calls the Stored Procedure (as implemented in the ADO example), since the SQL Server / MSDE Stored Procedure cannot be called using dbExpress at this time.
However, we can add a different implementation, returning the name of the Friend who has borrowed the current DVD. That would be helpful at times.
For the implementatin of this, I'm using two quick filters: one to locate the current CdDvdID in the cdsBorrow table, which will deliver the FriendID, and another filter to locate the Friend record in the cdsFriend, so I can display the name of the Friend.

```
void __fastcall TForm1::btnBorrowedClick(TObject *Sender)
{
  DataModuleDBX->cdsBorrow->Active = true;
  DataModuleDBX->cdsBorrow->Filtered = false;
  DataModuleDBX->cdsBorrow->Filter = "CdDvdID = " +
    DataModuleDBX->cdsCdDvdCdDvdID->AsString;
  DataModuleDBX->cdsBorrow->Filtered = true;
```

```
    DataModuleDBX->cdsFriend->Filtered = false;
    DataModuleDBX->cdsFriend->Filter = "FriendID = " +
      DataModuleDBX->cdsBorrowFriendID->AsString;
    DataModuleDBX->cdsFriend->Filtered = true;

    ShowMessage(DataModuleDBX->
      cdsFriendName->AsString + " got it");

    DataModuleDBX->cdsFriend->Filtered = false;
    DataModuleDBX->cdsBorrow->Filtered = false;
  }
```
<center>Listing 10. Who's Got It?</center>

As a side effect of using the filter, you may notice that the DataGrid showing the Friends will show the "wanted" Friend only, for as long as the Message dialog is displayed. As soon as we click OK, the filters are disabled, and all friends are visible again.
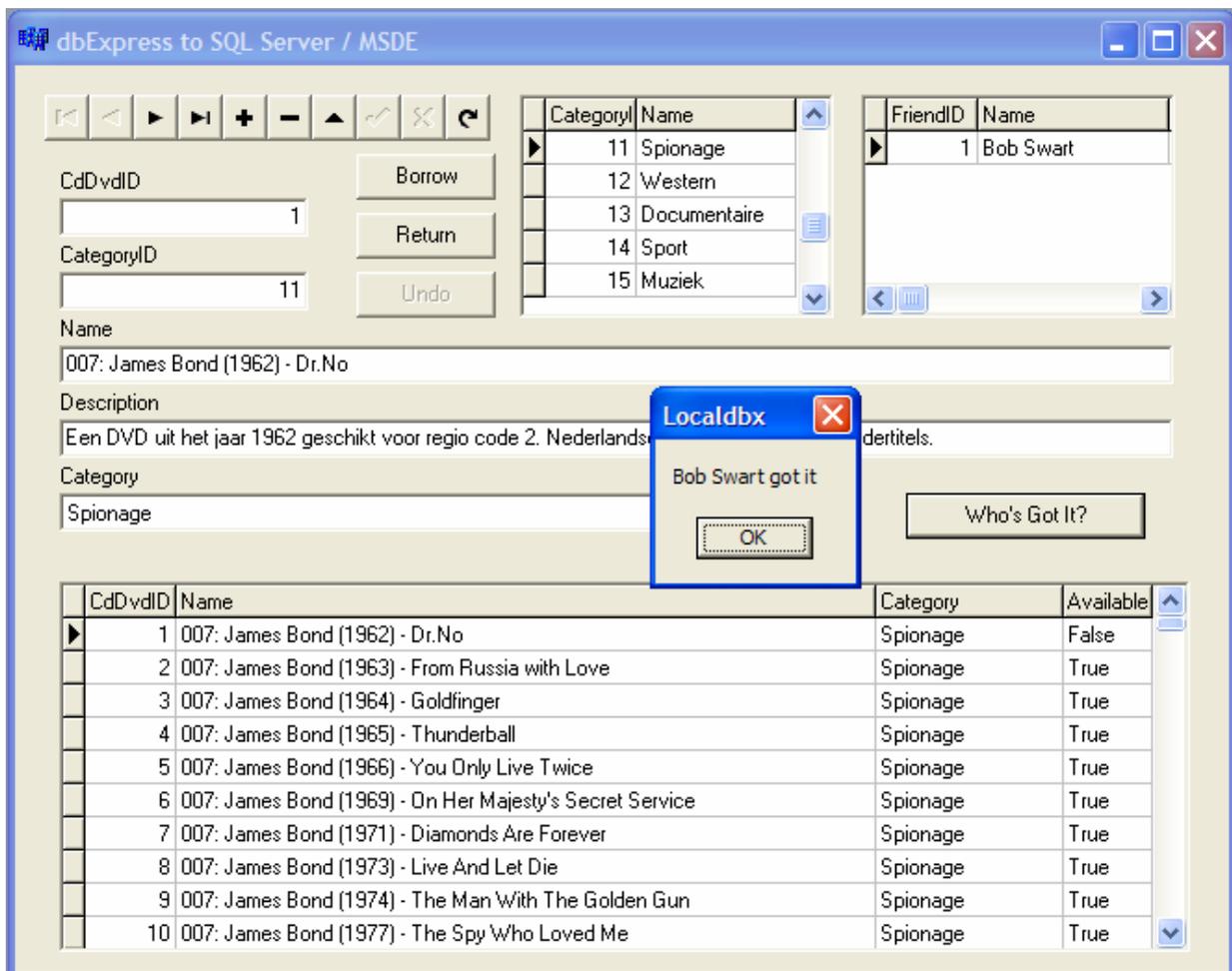

Figure 1. Who's Got It?

Summary
In this section, I've migrated the DVDs from the local MyBase format back into the SQL Server / MSDE database. We then migrated the data modules to a disconnected model using dbExpress, and learned that a dbExpress dataset is unidirectional and read-only, meaning we have to use a TDataSetProvider and TClientDataSet in combination with a dbExpress dataset in order to be able to navigate and work with (modify) the data.

This ends the Turbo C++ / C++Builder Database Development series for now. If this manual proves successful, I'll be back with multi-tier database coverage including DataSnap.