Surviving Client/Server: Replicating Server Constraints In The Client

by Steve Troxell

There is a constant struggle in client/server development about where and how to enforce data constraints. Check constraints and default values can be defined within the table definition on the server, so data integrity is enforced automatically by the server regardless of the application that is accessing the data. On the other hand, it is often not practical to wait until we attempt to write the record to the database to find out we've entered incorrect data. Further, if we're going to get a default value for a field, it would be nice to see that default value on the screen before we decide to save to the database.

Traditionally, client/server projects have tried to balance both methods by defining the constraints on the server side and recreating them in code in the client application. The client code provides a cleaner, more robust, handling of the validation, while the server side constraints provide the ultimate assurance of data integrity, particularly from standalone SQL utilities that modify the data directly.

Delphi tries to simplify the synchronization of constraint definitions between the server and the client by providing a conduit through which constraints defined on the server can flow into the dataset components in our applications. While this technique does the job it advertises, there are a number of awkward points that make it difficult to work with on larger projects. I decided to cover the system as Inprise intended it to work and try to point out the pitfalls wherever possible, then let you decide if it's worth the trouble.

Delphi's Data Dictionary

Since Delphi 2, the SQL Explorer utility has supported the concept of a data dictionary for Delphi projects. The Data Dictionary is a repository of predefined `TField` properties for selected database fields. With the Data Dictionary, a given database field can be consistently presented throughout a project or across several projects, without the need to set each `TField` instance individually.

For example, let's say you have an application that allows entry of a part number on several different screens. Perhaps the part number is always a particular format, such as three letters followed by up to four digits, so you would typically want an edit mask of `LLL0999` for the part number edit control. Normally, you would simply enter this value into the `EditMask` property of the persistent `TField` component in the dataset's Fields Editor. If you had several different tables that required a part number, you would have to make sure the `EditMask` property was set correctly for each `TField` component in each dataset.

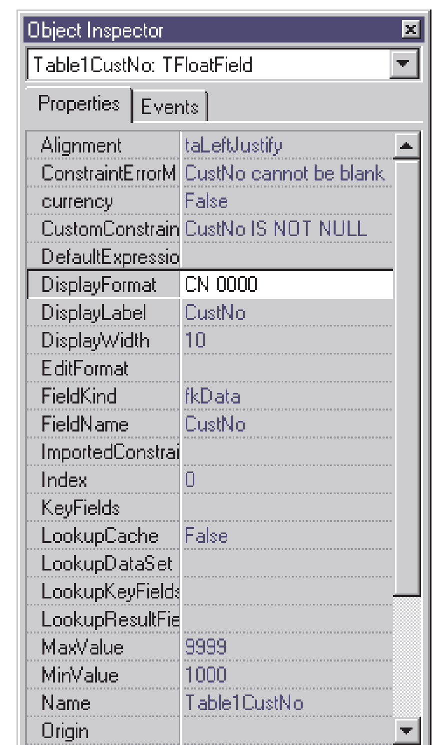
With the Data Dictionary, we can define the edit mask for the part number field externally, so that every time we produce a `TField` component for a part number, the `EditMask` would be set for us automatically. Not only would the `EditMask` be preset for us for any dataset we create in the project, but also for any dataset we create in *any* Delphi project using the same database.

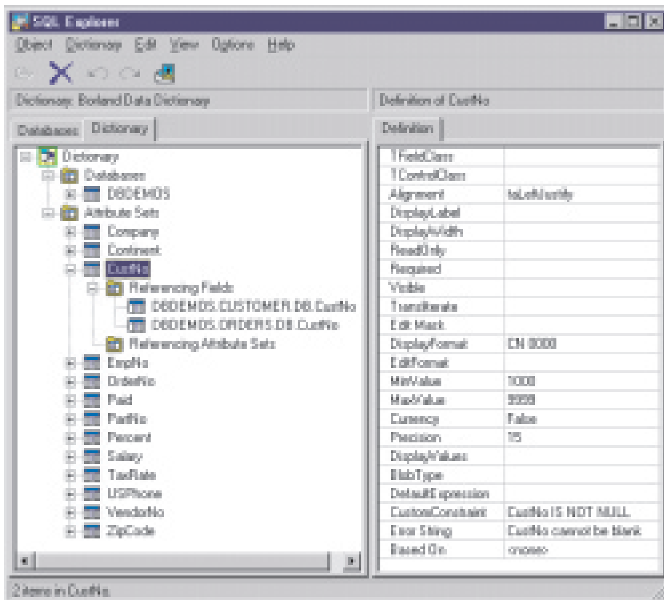
Let's see the Data Dictionary in action. Delphi ships with an example Data Dictionary for the `DBDEMOS` database. Create a new project in Delphi and drop a `TTable` component onto the form. Set the table's `DatabaseName` property to `DBDEMOS` and set the `TableName` property to `CUSTOMER.DB`. Double click the `TTable` component to bring up the Fields Editor. Then right click and select `Add All Fields` from the menu. Click on the `CustNo` field and look at the Object Inspector (Figure 1).

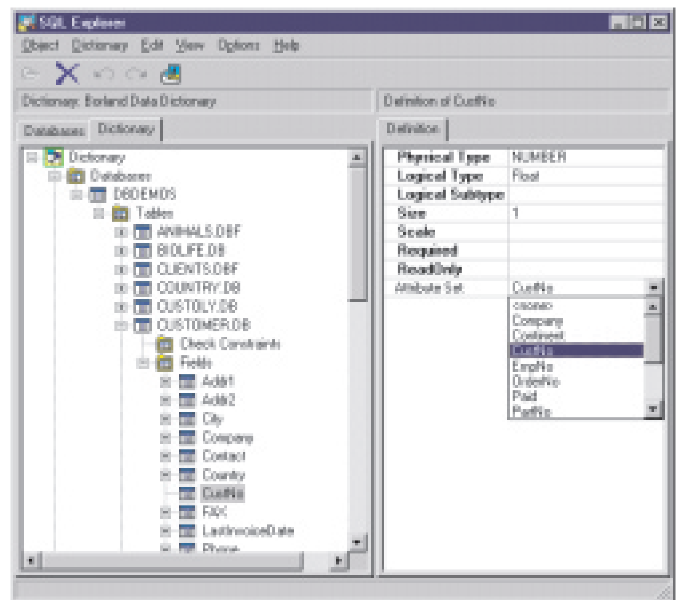
If you look closely, you will notice that the `CustomConstraint`, `DisplayFormat`, `MaxValue` and `MinValue` properties all have preset values that would normally be blank. This is because these properties have been assigned specific values in the Data Dictionary for this particular field in this particular database. You are free to change any or all of the values in the field component. The Data Dictionary simply provides new default values; it does not make them irrevocable.

To see the actual Data Dictionary that is governing the `DBDEMOS` database, launch the SQL Explorer application, click `Dictionary | Select` from the main menu, and select the `Borland Data Dictionary`. Figure 2 shows the Data

➤ *Figure 1*







➤ *Above: Figure 2*

➤ *Right: Figure 3*

Dictionary's settings for the `CustNo` field.

Field properties are defined in the Data Dictionary by creating an 'attribute set' with an arbitrary name and then setting one or more of the field properties shown in the right pane. Attribute sets in and of themselves are independent of any particular field in the database. This gives us the flexibility of assigning the same attribute set to more than one field in the dictionary. By looking under `Referencing Fields` in Figure 2 we find that the `CustNo` attribute set is bound to the `CustNo` fields in two tables: `Customer` and `Orders`. We associate an attribute set with a database field in the dictionary by drilling down to the field in the database and assigning an attribute set to the field (see Figure 3).

Once an attribute set is assigned to a database field, whenever a `TField` component is created in a Delphi dataset using the same database, the property values assigned in the dictionary will be used to default the corresponding property values in the `TField` component. The key factor that makes all this work is that the same BDE alias is used to refer to the database in both the Data Dictionary and the Delphi project.

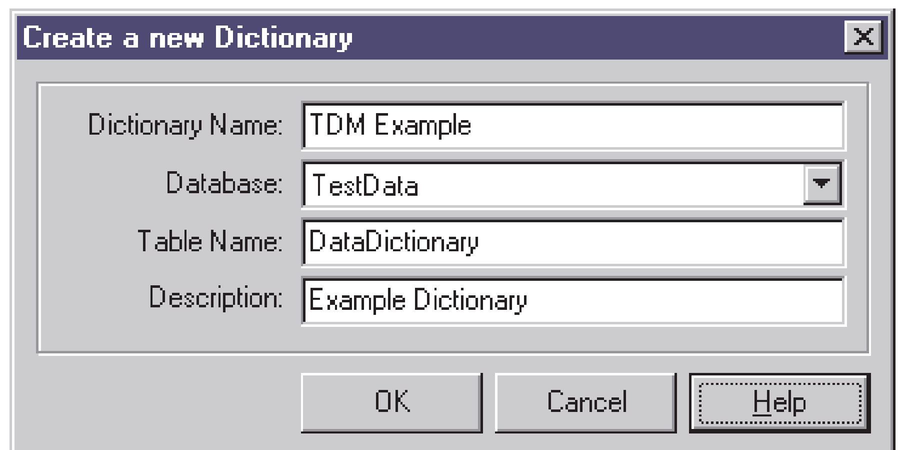
What you gain by using the Data Dictionary is that you can define selected field properties independently of any particular form or project so that the field properties can be applied consistently to all occurrences of that field throughout all your projects. Starting with Delphi 3, the Data Dictionary also has the ability to read field properties directly from the database's schema information. This means that defaults and constraints defined as part of the server-side table definition can be imported automatically into the Data Dictionary. These in turn could be replicated in the `TField` components used in your Delphi projects.

Creating A Data Dictionary

Now that we've seen that a Data Dictionary might be a useful thing, how do we set one up? Let's assume we have an existing database with a BDE alias called `TestData`. In SQL Explorer, select `Dictionary | New` from the main menu. This brings us to the dialog shown in Figure 4.

Dictionary Name simply identifies this dictionary from other dictionaries we might have set up. *Database* is the BDE alias for the database in which we will store the dictionary information. The data dictionary information will be stored in a special table in this database. Note that this is not necessarily the same as the database we are creating the dictionary for. The Borland Sample Dictionary applies to the `DBDEMOS` database, but Inprise chose to store the dictionary itself in a database called `DefaultDD`. This approach may be practical if the data dictionary holds definitions for multiple physical databases. For our purposes here, we will store the dictionary in the same database that we

➤ *Figure 4*



```
CREATE TABLE Employee(
   EmpID      char(9)        NOT NULL,
   FirstName  varchar(20)    NOT NULL,
   MiddleInit char(1)        NULL,
   LastName   varchar(30)    NOT NULL,
   JobID      smallint       NOT NULL DEFAULT 1,
     CONSTRAINT CK_JobID CHECK (JobID > 0),
   JobLevel   smallint       NOT NULL DEFAULT 10,
   PubID      char(4)        NOT NULL,
   DateOfHire datetime       NOT NULL DEFAULT getdate()
)
```

➤ *Listing 1*

are creating the dictionary for. Why scatter our project data needlessly? Besides, by storing the data dictionary in the same database, we ensure that it gets backed up as consistently as the database itself does.

Table Name is the name of the table which will be created in the dictionary database to hold the dictionary information. We can call this table anything we want as long as it is legal for the database we selected. *Description* is an arbitrary text field that further defines our dictionary. As far as I can tell, the description is not used anywhere except for display in SQL Explorer.

Now that we've created the dictionary, we need to associate it with the database. From SQL Explorer's main menu, select `Dictionary | Import From Database`. Then select the BDE alias for the database we want to associate with the dictionary. In our case this is the same `TestData` database which we used to store the dictionary table, but remember that we could have stored the dictionary table in a completely different database. If our application accessed several different databases, we could add any number of databases to the dictionary. This could be helpful if we had fields, like customer number, that spanned multiple databases in our system. We could define the field properties once in the dictionary and bind those properties to all instances of customer number across all the databases in the dictionary.

It's important to note here that we aren't really adding databases to the dictionary, we are adding BDE aliases, in other words pointers to the actual databases. The only way the Data Dictionary information can get integrated into the datasets in your applications is if they refer to the database by its BDE alias.

Importing Server Constraints

Once we import a database into the dictionary, field properties defined in the database's metadata, such as defaults and check constraints, are automatically imported into the dictionary. Our example database includes a table called `Employee` defined as shown in Listing 1. Let's see what the Data Dictionary gave us for this table simply by importing it.

If we look at the bottom of Listing 1 we see that `JobID` has a check constraint on it, and three fields have default values. The default value for `DateOfHire` is the SQL Server function `getdate()` which returns the current date and time.

Figure 5 shows the Data Dictionary after importing the database. Under the `Databases` branch we have entries for all the tables and all the fields in each table. Notice that our `DataDictionary` table, the one we set up to hold the dictionary information, is shown here too. Under the `Employee` table, we also see an entry for our check constraint, with the actual constraint expression stored in the `ImportedConstraint` property. The Data Dictionary records this as a record level constraint rather than a field level constraint.

Specific field properties such as defaults are not stored directly with the table's field definition under `Dictionary | Databases`. Instead, field properties are stored separately in an attribute set. Figure 5 shows the attribute sets that were automatically generated when we imported the `TestData` database. Each attribute set is then bound to the actual database field it applies to (see the *Referencing Fields* section). When importing server-side constraints, the Data Dictionary automatically generates an attribute set name based on a combination of the

➤ *Figure 5*

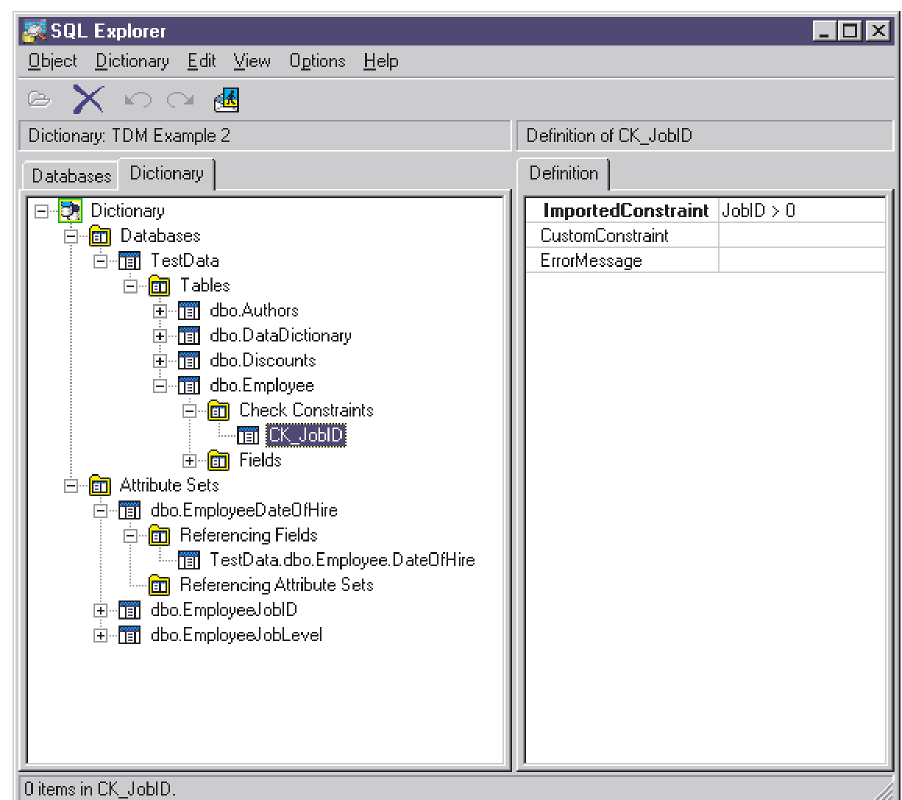


table name and field name. We can rename these if we wanted to. The default values themselves are reflected in each attribute set's `DefaultExpression` property.

Creating a new attribute set is as simple as right clicking within the `Attribute Sets` branch in the treeview and selecting `New`. From there you name your attribute set, fill out the properties page (Figure 2), then bind the attribute set to a field (Figure 3).

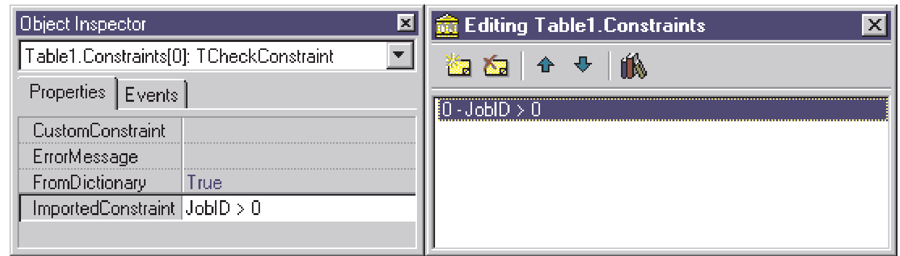
The Delphi Side Of Things

As we said at the beginning, the properties defined in the dictionary's attribute set are used to pre-populate the properties in the `TField` component when we create persistent field components for our Delphi datasets. Whether the value in the attribute set property was set by hand or imported from the server makes no difference to how it gets linked into our Delphi projects.

The check constraints, on the other hand, do not pop up in our datasets automatically and require a bit more manual intervention. After adding a dataset component to your project and setting it appropriately (`TQuery.SQL` or `TTable.TableName`), select the `Constraints` property and click. To bring up the property editor you must right click the component. Then select `Read From Dictionary` to load any check constraints that are defined in the dictionary for the table. Imported constraints will appear in the `ImportedContraints` property (Figure 6).

Once the constraint and default values are defined, Delphi takes care of enforcing them. When a new record is inserted in the dataset, any field with a `DefaultExpression` will be pre-populated.

Interestingly enough, the default expressions are handled by the BDE and allow certain server-specific function calls to be made. For example, Listing 1 shows the default for the `DateOfHire` field to be the SQL Server function `getdate()`. If we examine the Data Dictionary and the `TField` component for this field we find that the



➤ *Figure 6*

`getdate()` function call is propagated all the way down the line, even though this is a function that Delphi knows nothing about. However, when we insert a new record into the dataset, the `DateOfHire` field is correctly prepopulated with the current date and time.

The Delphi VCL is handling this by passing `DefaultExpression` through the BDE to handle server-specific function calls. This is handled completely in the BDE layer (most likely in the SQL Links driver). There also seems to be support for only a limited subset of the server-side functions. While Delphi can figure out what to do with the `getdate()` function call, it cannot resolve calls to the functions `user_name()` or `rtrim()`, for example. When importing defaults into the Data Dictionary which make use of server-side functions that the BDE does not support, the default will be brackets with SQL comment delimiters in the Data Dictionary (that is, `/* user_name() */`). This default still appears in the `DefaultExpression` property of any `TField` components you set up for that field, but the default value itself will be ignored.

Some further notes on using `getdate()` as a default value are useful here. When a record is inserted into a dataset and the default value is applied, the current date and time used in response to `getdate()` is the date and time as it is set on the client workstation, not the server as you would get with a server-side default. Also bear in mind that the current time value will reflect the moment the record was inserted into the dataset component, not the moment that the record actually was stored in the database. Again, a subtle difference from a true server-side default.

When the record is posted in the dataset, if any of the check constraints are violated, the post will be aborted and an error message displayed. The generic error message for a constraint violation is rather bland: *Record or field constraint failed. JobID > 0.* Fortunately, you can supply your own text for constraint violations by setting the `ErrorMessage` property of the constraint (which can also be predefined in the Data Dictionary).

Quirks, Aberrations And Pitfalls

What I've given you up to now is how the Data Dictionary is *supposed* to work. It largely does work as advertised, but if I were to leave you now, you would quickly run into problems making the Data Dictionary work. There are a number of oddities in the system that you will only find out about through trial-and-error, or, like me, through some very helpful participants in the Borland newsgroups.

BDE AliasName Versus TDatabase.DatabaseName

As I've been saying, the entire Data Dictionary system revolves around using the same BDE alias to refer to the database in your Delphi applications as in the Data Dictionary. It turns out that Delphi is *very* picky about this matter. Most database applications would use a `TDatabase` component to handle the database connection and then link all the dataset components to the `TDatabase` component (by setting each `TBDEDataset.DatabaseName` equal to the `TDatabase.DatabaseName`). However, if you do this, Delphi will not automatically import any information from the Data Dictionary. Further, when you

try to import check constraints from the `Constraints` property editor, the `Read From Dictionary` menu item will be disabled. You must set the `TBDEDataset.DatabaseName` property to the actual BDE alias name for the database. You can always hook the dataset back to the `TDatabase` after you've loaded the property values from the Data Dictionary.

If you right click on a field in the Fields Editor you will see a number of menu items at the bottom of the menu related to attributes sets from the Data Dictionary. You can manually associate a `TField` with an attribute set by selecting `Associate Attributes` from the menu. You then pick the desired attribute set from a list of choices for the current Data Dictionary. This is the technique you would use if you did not want to link your dataset components to the BDE alias name directly in order to load the property defaults automatically.

In fact, you will have to manually associate your fields in either case. When you let Delphi automatically load the default property values, it links to the correct attribute set to get the correct property information but then forgets all about that association. If you right click on the field and look at the menu items related to the Data Dictionary, many of them are disabled as though there is no association with an attribute set. You have to define this association by hand even though Delphi loaded the property values correctly.

Reflecting Dictionary Changes In The Datasets

Why do we care about preserving the attribute set association if Delphi will load the property values anyway? Because once you've imported all the constraints and defaults from the database, the occasion is bound to arise when you'll need to change one or more of these definitions on the server. For example, let's say the `Employee.FirstName` field went from having no default to a default of an empty string. Once this change is made on the server, we need to re-import the `Employee` table into the Data Dictionary. We do this by selecting the `Employee` table in the Data Dictionary, right clicking, and selecting `Import To Dictionary` from the `context` menu. Now the new default is reflected in the Data Dictionary. At this point, all our datasets in our Delphi projects still have no `DefaultExpression` for the `FirstName` field.

As you can imagine, we couldn't very well expect the Data Dictionary to crawl through the DFMs of all our Delphi projects and change `TField` property values for us. But Inprise certainly didn't break a sweat making it easy for us to replicate that change in our projects ourselves. What you have to do is hunt down all your dataset components containing the `FirstName` field, select the field in the Fields Editor, right click, and select `Retrieve Attributes` from the `context` menu. This refreshes the property values from the Data Dictionary. `Retrieve Attributes` is not enabled unless an association is made between the `TField` and the

attribute set in the Data Dictionary. Unfortunately, the association must always be made manually by selecting `Associate Attributes` from the `TField`'s `context` menu. At least this is a step you only have to do once.

Reflecting Server Changes In The Dictionary

If you have made server definition changes and need to re-import those definitions into the Data Dictionary, there are a couple of things to look out for.

If your BDE alias for the database is using schema caching, you'll have to shut down all applications that use the BDE (including the SQL Explorer), in order to force the new schema information into the cache for that database. Once you've cleared the schema cache, then you select the table in the Data Dictionary, right click, and select `Import to Dictionary` from the `context` menu.

One problem I found occurs when you remove a default value from a field. If the Data Dictionary previously had a default value for that field and there is no longer a default value assigned for that field on the server, then after re-importing the table from the server the Dictionary retains the old default value. It isn't smart enough to clear the default value from the attribute set. On the other hand, it is smart enough to delete check constraints that no longer exist on the server.

Case Sensitivity Of Check Constraints

Keep in mind that there can be some subtle differences in the logic used by the BDE to evaluate check constraints versus the logic used by your database server. One point in particular is that if your database server evaluates string expressions without case sensitivity, BDE will be case sensitive and will cause some constraints to fail that would have passed the database server checking.

For example, if you have a constraint that a given field must contain a `Y` or `N` value, this constraint would probably be imported as `(IsActive = 'Y') or (IsActive = 'N')`. The BDE will test the values strictly against the case given in the conditional expression. A lowercase `y` or `n` would fail the test, whereas if the database server is case-insensitive, a lowercase `y` or `n` would pass the test.

You could try to get clever and change the constraint on the server use SQL pattern-matching logic to account for case sensitivity: `IsActive LIKE '[yYnN]'`. The pattern-matching syntax used here means 'match any one of the characters listed between the brackets'. This constraint will import into the Data Dictionary just fine, and even be imported into your dataset's `Constraints` property just fine. But when the BDE tries to enforce this constraint when a record is posted in the dataset, it will fail as though the constraint has been violated, *even if the data is valid*. The BDE cannot interpret the pattern-matching tokens in this expression, and unfortunately it makes no distinction between that failure and an actual constraint violation. However, the BDE can correctly interpret the SQL wildcard characters `%` and _ when used with the `LIKE` operator.

Other Odds And Ends

The Data Dictionary will not import primary key, foreign key, unique or nullability constraints. The nullability one surprised me because there is a `Required` property in the Data Dictionary. Also, the Data Dictionary will not import any server information whatsoever if the BDE alias used ODBC for connectivity. Finally, the enforcement of imported defaults and check constraints is tightly tied to the BDE and therefore is only available for the `TBDEDataset` descendants, not for custom dataset components derived from the abstract `TDataset` class.

Conclusion

I think there is something useful here, but it appears half-baked. The support in Delphi's dataset components to enforce constraints and default values is very good. But the mechanism for getting those constraints and defaults from the server and into the dataset components, in the form of the Data Dictionary in SQL Explorer, is weak. While it does the job to a limited extent, there are gaps in its support and accommodating changes to the server's metadata is cumbersome to say the least.

In a future issue, we'll look at some ways we might build upon this framework to make up for some of its shortfalls.

Steve Troxell is a software engineer with Ultimate Software Group in the USA. He can be contacted by email at Steve_Troxell@USGroup.com

For Delphi News check the Developers Review website at www.itecuk.com