Data Modules: Friend Or Foe?

by Guy Smith-Ferrier

Data Modules were introduced in Delphi 2 in response to developers attempting to turn forms into makeshift data modules in Delphi 1. Since Delphi 2 was launched, the mechanics of using data modules has changed very little and the only significant enhancement has been the addition of design-time diagrams in Delphi 5. Most developers will have seen, and probably used, data modules, so this article is not an explanation of what they are and how to use them. Instead, I will cover what the online help doesn't tell you, what problems you will encounter in their use in the real world, and (you will be relieved to hear) how you can overcome these problems.

Reusing Data Modules In SDI And MDI Applications

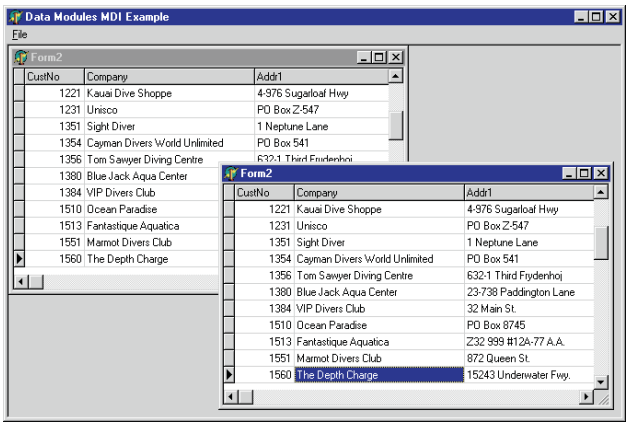
So let's take a look at how data modules behave in an MDI application (actually the same behaviour can be exhibited in an SDI application, it is just more readily demonstrable in an MDI application). Assume that I create a new application and set the main form's `FormStyle` to `fsMDIForm`. Now add a second form called `CustomerForm` and set its `FormStyle` to `fsMDIChild` (remember to take it out of the AutoCreate list). Add a `TMainMenu` to the main form with a menu item called `Browse Customers` and add this code to the menu item:

```
var
  Form2: TForm2;
begin
  Form2:=TForm2.Create(
    Application);
  Form2.Show;
end;
```

Add an `OnClose` event to `TForm2` to set `Action` to `caFree`. Create a new data module, ensure that it is autocreated, add a `TTable` to it for the `DBDEMOS Customer.db` table, and set `Active` to `True`. Ensure that `Form2` uses `DataModule1` and add a `TDataSource` to the form and set its `DataSet` to `DataModule1.Table1`.

Lastly, add a `DBGrid` to `Form2` and connect it to `DataSource1`. The MDI application is complete.

➤ *Figure 1*



Now for the demo. Run the program and click on `Browse Customers` twice. You will see that the windows are tiled. Click on any record in either of the two windows and you will see that the two windows are not independent of each other (see Figure 1).

Well, this isn't too surprising, because there is only one data module being used and it is being shared between two different forms. Clearly, there is only a single instance of the `TTable` object and it has only one record pointer, so when the record pointer is updated in one form the other form is updated with the change to the same record pointer.

At first the whole subject may seem a moot point because you could ask 'why would the user want to run the same menu item twice anyway?'. The idea isn't as daft as it seems. If the `Customers` form has a menu option which allows the user to set a scope so that they could, for example, set the city of the customer then the user could show customers from one city in one form and customers from a different city in another form, thereby allowing them to compare the two lists. Further, if you think that this problem won't happen to you because you don't write MDI applications, then you should think again. The MDI example is simply one way to illustrate the sharing of a data module across an application: data modules in an SDI application are shared just as readily.

So the solution is simple, right? Just create a second data module to go with the second form? Ok, let's try this. Remove the data module from the autocreate list and change the menu item's code to that shown in Listing 1.

Unfortunately, this does not solve the problem, as the two forms are still linked to each other via the same data module.

So what's going wrong? To understand how to fix this problem first we need to understand what is actually happening. Here goes. As each data module is created it makes a call to `Screen.AddDataModule(self)`. This adds the data module to the global Screen's private `FDataModules TList`. `TDataModule` calls `Screen.RemoveDataModule(self)` in its destructor. When the form which uses the data module is created

➤ *Listing 1*

```
var
  Form2: TForm2;
begin
  DataModule1:=TDataModule1.Create(Application);
  Form2:=TForm2.Create(Application);
  Form2.Show;
end;
```

```
var
  Form2: TForm2;
begin
  DataModule1:=TDataModule1.Create(Application);
  Form2:=TForm2.Create(Application);
  DataModule1.Name:='';
  Form2.Show;
end;
```

➤ *Listing 2*

each of the components which it owns is also created. As each component is created, `FindGlobalComponent` is called to find the component's associated data module. `FindGlobalComponent` searches through `Screen`'s list of data modules for one matching the name it is looking for (note that it doesn't make use of the data module's global variable at all and, like all forms which are not autocreated, you can delete this global variable). So the fixup between the components on the form and the data module is based on the name of the data module. The reason that the form gets attached to the wrong data module is that as each data module is created it is created using the same name. Table 1 shows the contents of `Screen.FDataModules` after two data modules have been created.

So, when `FindGlobalComponent` is called upon to find a data module matching the supplied name it always finds the first instance of the data module, no matter how many copies of the data module have been created.

With this in mind we can modify our menu item: see Listing 2.

This is a very interesting trick. As the data module is created it is given its correct name (ie `DataModule1`). As the form is created it binds correctly to the data module based on the data module's name. However, to prevent subsequent forms from binding to a previously created data module instead of the most recently created data module, we set the data module's name to empty. Table 2 shows the contents of `Screen.FDataModules` after two data modules have been created.

Of course, the solution isn't finished yet. It breaks one of the basic rules of object oriented design: that is, at present the user of the `TForm2` class is required to know more about the class than they should have to know (ie that in order to use the form they are also responsible for creating a data module). So we need a little encapsulation.

Add a protected field called `DataModule` of type `TDataModule1` (or even `TDataModule`) to `TForm2`. Add an `OnCreate` event to `TForm2` to create the data module and pass `self` as the owner of the data module (this allows the form to destroy its own data module when it is safe to do so). However, we still have to set the data module's name to empty and the best place to achieve this is in the form's `OnShow` event. So, with a certain amount of playing around, the problem can be solved.

There is an alternative solution, however. It follows the same steps as the previous solution except for setting the data module's name to empty in the form's `OnShow` event. Instead, you can set the data source's dataset directly in the `OnCreate` event after the data module has been created:

```
DataSource1.DataSet:=
  DataModule1.Table1;
```

However, this solution is more fragile than the first and therefore less preferable. The problem is that as `TDataSources` are added, deleted and renamed, the form's `OnCreate` event needs to be modified to keep up with the changes. By comparison, the first solution can be written once and forgotten about.

This also brings up the subject of where to put the data source: on the form or on the data module? There are pros and cons with each location. If the data source is placed on the form and you are using the solution of setting the data source's `DataSet`, as shown above, then all of the data aware controls on the form can be attached to the data source and there is only a single assignment of the data module's table to the data source's dataset. By contrast, if the data source is on the data module then there would be an assignment for each and every data aware control on the form. Alternatively, if the data source is placed on the data module then master/detail relationships can be defined in the data module. If you use the solution of setting the data module's name to empty then the best location for the data source is on the data module.

Chaining Data Modules In The IDE

The next problem with using data modules is using 'chained' data modules in the Delphi IDE. By 'chained' I mean one data module which uses another data module. In some ways the IDE is really quite clever. Assume you have an application which has a main form and one other form, we'll call it `Form2`. It also has a single data module containing a `TDatabase`, a `TTable` and a `TDataSource`. `Form2` uses the data module and it contains a simple `DBGrid` which uses the datasource in the data module. If neither `Form2` nor the data module is currently open in the IDE then `Form2` can be

Screen.FDataModules i	TDataModule(Screen.FDataModules[i]).Name
0	'DataModule1'
1	'DataModule1'

➤ *Above: Table 1*

➤ *Below: Table 2*

Screen.FDataModules i	TDataModule(Screen.FDataModules[i]).Name
0	''
1	'DataModule1'

opened without any undesired effects. This is where Delphi's IDE has performed some clever trickery. If the IDE blindly opened the form then it would result in an error, because the `TDataSource`, `TTable` and `TDataBase` upon which the form is based are not available. Instead, the `DBGrid` shows the data as if the table was open and this can only happen if the IDE surreptitiously opens the data module upon which `Form2` is based. The data module is opened but it is hidden. It's a rather useful trick really.

Unfortunately the IDE is smart only up to a point. Assume that you move one or more of the components in the data module to a second data module and have the first data module use the second. For the sake of our example, we'll move the `TDatabase` to the new data module. Now we have 'chained' several data modules. Now ensure that all data modules and forms except for the main form are closed and reopen `Form2`. The IDE reports an *'Unknown database'* error. Why? Well, the IDE is smart enough to open the data module upon which the form is based, but it isn't smart enough to open the data module upon which the data module is based. There is a potentially bigger problem waiting to happen here. Because the data module containing the `TDatabase` wasn't opened, the `TTable` on the other data module cannot be opened, so the IDE sets `Table1.Active` to `False`. If the project is subsequently saved then the table is saved in its new state (ie closed) and the application clearly will not function the same way as before. Perhaps the biggest problem here is that this effect can easily go unnoticed.

So what can we do about it? Well, one solution is never to leave tables open in the design-time environment. Instead, tables have to be opened in the relevant data module's `OnCreate` event. This is a rather poor solution. The problem with this solution is that you lose all of the benefits of the design-time environment. A general principle of development in any development environment is that you want to have as much of your application verified at design-time or compile-time as possible. If ever you have a choice between a solution which can be verified at design-time or compile-time, as opposed to a solution which can only be tested at runtime, then you should always choose the former, everything else being equal. By closing the tables at design-time you do not get the verification that the table exists and all of the required fields are still in the database. If there is one maxim which sustained application development has taught me, it is that change is the only constant. Everything changes. Changes to the database structure are a continuous process and leaving tables open at design-time helps protect against these changes.

Another solution to the problem is always to ensure that the data modules required by other data modules (ie the data modules higher up in the data module chain) are always open before opening up a form which is dependent on them. There are several ways to implement this solution.

The most laborious, cumbersome and error-prone way is to remember to manually open relevant data modules before opening dependent forms. The next solution is to use the `Project Desktop` option (on the `Preferences` tab in `Tools | Environment Options`). This option saves the current configuration of the desktop in the project's DSK file when the project is closed. This solution needs to be treated with care. The idea here is to use Delphi to ensure that the data modules are opened in the IDE *before* the forms which need them are opened. This requires an understanding of how Delphi saves and restores this information. When a project is closed Delphi saves the open modules in the project's DSK file. However, it is sensitive to which form currently has focus in the IDE. The open forms and data modules are saved in the `Modules` section of the DSK file (which is a regular INI file). Listing 3 shows the `Modules` section of a DSK file.

```
[Modules]
Module0=C:\DesignTime\DesignTimeDM2.pas
Module1=C:\DesignTime\DesignTimeU2.pas
Module2=C:\DesignTime\DesignTimeU1.pas
Count=3
EditWindowCount=1
```

➤ *Listing 3*

The form or data module which has focus in the IDE when the application is closed is the first module in the list. When the project is reopened the files are opened in the order dictated in the DSK file. With this knowledge it is easy to effect a solution and also to see how easy it is to let Delphi interfere with the solution.

The solution to the problem is to ensure that the data modules are listed in the DSK before any of the forms on which they are dependent. One way to do this is to ensure that, before you close an application, you always ensure that either all forms except the main form are closed and 'chained' data modules are opened, or that the 'chained' data module has focus. Unfortunately it is all too easy to break this solution because, as Delphi always saves the DSK when the project is closed (if Project Desktop is checked), then it is too easy to forget to move focus *away* from a form which is dependent on a data module chain. If the project is closed with a form having focus then that form is saved at the top of the modules list and it will be the first form opened when the project is reopened and the data module won't have been created and we are back to square one.

A better variation of this solution is to open the project, open the main form, open the chained data modules, close any other forms, check the `Project Desktop` option and close the project. Then open the project and uncheck `Project Desktop`. In this way the project is always opened in a suitable state and order of the modules in

```
procedure TForm3.AfterPost(DataSet: TDataSet);
begin
  StatusBar1.SimpleText:='Saved';
end;
procedure TForm3.BeforeEdit(DataSet: TDataSet);
begin
  StatusBar1.SimpleText:='Editing';
end;
procedure TForm3.BeforeInsert(DataSet: TDataSet);
begin
  StatusBar1.SimpleText:='New record';
end;
```

➤ *Above: Listing 4*

➤ *Below: Listing 5*

```
procedure TForm3.FormCreate(Sender: TObject);
begin
  DataModule:=TDataModule1.Create(self);
  DataModule.Table1.BeforeInsert:=BeforeInsert;
  DataModule.Table1.BeforeEdit  :=BeforeEdit;
  DataModule.Table1.AfterPost   :=AfterPost;
end;
```

the DSK file is never rewritten because `Project Desktop` has been unchecked.

Still one more variation on this solution is to write a utility to automatically reorder the modules in the DSK file. This utility would look at each file in the list and determine if it is a form or a data module and then rewrite the list so that the data modules are placed at the top of the list. This isn't the simplest of undertakings, for two reasons. First, you'd need to know when a module is a form and when it is a data module (a simple solution could be based on a naming convention for your files). Secondly, it would require you to either run the utility manually every time before a project is opened or alternatively make use of the Open Tools API to ensure the DSK is rewritten immediately before a project is opened.

Reusing DataSets With Business Logic

Another problem with data modules is also centred around their reuse. This problem, however, doesn't concern the reuse of their state information (ie the record pointer, etc.). Instead it concerns the reuse of their business information. One of the frequently touted benefits of data modules is that they are useful for reusing business logic or, more accurately, centralising the definition of business logic. Certainly data modules do achieve this goal. However, the mechanism of data modules does not allow for a distinction between those properties and events which are used for 'reusable' business logic and those properties and events which are used for context specific handling of a dataset. For example, a dataset might include a `BeforePost` event to specify some record-level validation (eg that an employee's gender cannot be male if their title is *Mrs*). This is a business rule and, certainly, data modules succeed in allowing this validation code to be reused across an application. In one specific use of a data module this might be all that is required. However, in another context the programmer might want to add context specific behaviour. For example, the dataset might include an `AfterPost` event to update a status flag on a status bar to change the record state from `Editing` to `Saved`. Clearly this `AfterPost` event must only be used in this one specific context.

One solution to this problem is to hand code the exceptions to the business rules, ie the context specific part. Assuming that the data module is called `DataModule1` and it has a table called `Table1` and we want to add three context specific event handlers, Listing 4 shows the events which could be added to a form called `Form3`.

The form's constructor would be responsible for manually assigning these events (Listing 5).

Although this solution works in the example given, it is fraught with danger: if someone declares a `BeforeInsert` event for `Table1` in `DataModule1` then the subsequent assignment of `Form3`'s `BeforeInsert` event would cause the data module's event to be completely bypassed. You could, of course, set up a system of event chaining or event notification or broadcasting but as Delphi's event model is single casting instead of multicasting, any such solution is necessarily a proprietary solution.

Another solution to this problem is to use visual form inheritance. To inherit from an existing data module use `File | New`, select the tab with the same name as the project, select the data module and click `OK`. This creates a new data module which inherits from the original data module. All of the context specific changes can be added to this new data module without interfering with the original data module and whilst keeping the inheritance chain. This solution has some advantages over the previous solution. Firstly, it allows events to be set using the object inspector instead of in code

```
procedure TDataModule2.Table1BeforeEdit(DataSet: TDataSet);
begin
  inherited;
  Form.StatusBar1.SimpleText:='Editing';
end;
procedure TDataModule2.Table1BeforeInsert(DataSet: TDataSet);
begin
  inherited;
  Form.StatusBar1.SimpleText:='New record';
end;
procedure TDataModule2.Table1AfterPost(DataSet: TDataSet);
begin
  inherited;
  Form.StatusBar1.SimpleText:='Saved';
end;
```

➤ *Above: Listing 6*

➤ *Below: Listing 7*

```
procedure TForm4.FormCreate(Sender: TObject);
begin
  DataModule:=TDataModule2.Create(self);
  TDataModule2(DataModule).Form:=self;
end;
```

and this is an easier approach to development. Secondly, it doesn't suffer from the problem of overwriting the original data module's own events as each event created in the inherited data module starts with the word `inherited` in order to call the inherited event first. However, all is not completely well with this solution either. Firstly, encapsulation purists will be very disappointed with the amount of coupling involved with the form unit and the new data module unit. Consider Listing 6 which shows the code added to the new data module to update the form's status bar.

To update the status bar the data module must have intimate knowledge of the form. So the new data module has a public field, `Form: TForm4` (where `TForm4` is the name of the new form). The form's `OnCreate` event is shown in Listing 7.

So the form knows about the new data module and the new data module knows about the form. Although the problem of the circular reference between the two forms is easily solved (by placing one `uses` clause in the `implementation` section) the heavy coupling of two units could be frowned upon from a design viewpoint.

Friend Or Foe?

So are data modules your friend or your foe? Well, let's look at what they're good at. Data modules allow you to keep your forms tidy by placing non-visual components in a separate location. In addition they allow you to reuse business logic across your application. One could also argue that the new Data Diagram support added in Delphi 5 allows you to represent the relationship between your tables graphically. However, in its current form, I think this is a bit of a non-feature, as most applications with a reasonably well developed database are assisted by some entity relationship or UML tool. What would have been more useful than the diagramming support would have been an extension to the Open Tools API to allow third parties to write drivers for their case tools to integrate with the IDE. This would have also been exceptionally useful for bringing back to life the unusable data dictionary also introduced in Delphi 2.

So the main benefit of data modules is that they allow business logic to be centralised in an application. However, this feature comes with a price, as I have shown. To reuse data modules at runtime you must ensure that each form has its own data module. Certainly, using the trick of setting the data module's name property to empty solves this problem, but the difficulty here is that this solution is not commonly known. I would have thought that since their release in early 1996 Borland would have provided better facilities for automating this (perhaps `TForm` could have a new published array property, `DataModules`, where simply adding a data module to this array would take care of all of the details).

Furthermore the IDE isn't smart enough yet to understand how to chain data modules. The most common solution to this is to close all datasets at design-time and then reopen them in code. This has always struck me as a poor solution. The problem is that the solution is, in itself, another problem which requires another solution (ie opening tables manually in the `OnCreate` method).

Lastly, as the very reason for the existence of inheritance shows, there are always exceptions to the rule. Data modules are reused in their entirety without making a distinction between those properties and events which should be reused and those properties which should not. Visual Form Inheritance helps solve this problem but, as with all of the other solutions to common data module problems, it brings with it new problems.

An Alternative

One alternative to data modules is to create business objects. The business objects can be as 'closed' or as 'open' as the designer's own personal preferences. In an 'open' business object (my own personal preference) each business object would descend ultimately from `TDataSet` (to allow it to be used in regular Delphi application development). Thus you would have a package of business objects (`TCustomers`, `TOrders`, `TContacts`, etc.) which would be installed on the palette and business objects would be dropped directly onto forms. The problems of reuse at runtime would disappear because each form would have its own instance of each business objects. The problems of chaining in the IDE would disappear because there would be no chaining. The problems of context specific differences would disappear because developers could inherit from business objects and add context specific changes to the subclass.

OOP purists would, rightly, complain that the approach of descending directly from `TDataSet` allows the business rules to be undermined by providing an opportunity for programmers to bypass them. This is too long an argument to cover adequately here, but for brevity I will say that although this is a very valid argument I opt for pragmatism instead of purity. Many others would disagree.

Conclusion

Data modules do allow business rules to be reused across an application, but not without a fair share of problems. The solutions to these problems are not without their own difficulties. I hope that Delphi 6 will address these issues. The alternative is to use business objects. If you believe, as many people do, that the relational model is on its way out and that the object model will be its replacement, then this might be a better approach for you, because business objects map more closely to an object database than the approach of using data modules does.

Guy Smith-Ferrier is Technical Director of Enterprise Logistics Ltd (www.EnterpriseL.com), a training company specialising in Delphi. He can be contacted at gsmithferrier@EnterpriseL.com