Application Architecture

by Max Rahder

This article discusses application architecture using a sample application based on the DBDEMOS database that ships with Delphi.

The companion disk contains two versions of the sample application: one for the initial implementation and one that reflects the enhancements added by the end of the article. Note that the application uses frames; as a result, the code will only compile under Delphi 5 and later versions.

Introduction

An application architecture is an infrastructure of classes that makes it easier to extend and modify a program. When creating an architecture the designer determines core functionality then creates classes that meet that functionality. Programmers then implement the application by extending classes and implementing key methods. The architecture uses the information provided by those methods to make things work. The result is dramatically reduced development and maintenance costs.

There are a number of benefits of having an architecture. Complexity and functionality is encapsulated in abstract classes: this simplifies the programming tasks of each developer, thus raising productivity and lowering the amount of code needed to be implemented. Changes to the core classes also affect the entire application, reducing development and maintenance costs. Application behavior is consistent, which makes it easier to use, reduces end-user training and raises end-user productivity.

The architecture I will describe in this article is primarily designed to facilitate discussion of concepts: it is not intended to be a turnkey solution. I have intentionally omitted a few important features in order to simplify the discussion.

Before reading this article you may want to review Guy Smith-Ferrier's article *Visual Form Inheritance and Frames: Friend or Foe* from the April 2000 issue. You may also want to review the discussion of class methods and overriding methods in the Delphi 5 help topic *Object oriented programming for component writers*.

Description

The sample application is a customer order system which uses the DBDEMOS Paradox database included with Delphi.

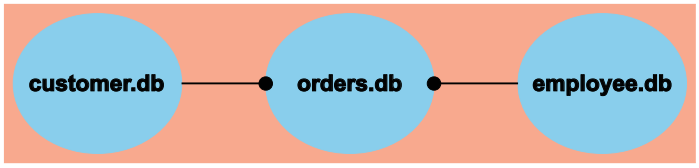
For the first part of the article the application references two tables from DBDEMOS: customer.db and orders.db. At the end of the article I'll add the employee.db table. The relationship between the tables is simple: There are customers, each of whom have from zero to many orders. Each order is taken by an employee.

If you compile and run the sample from the companion disk you'll see that the user initially sees a main form containing a toolbar and a main menu. From the main form the user selects either the customer or orders list. From the list the user can open a detail form used to view or edit customer or order data. The user interacts with four key types of forms and controls: the main form, list forms (which contain a frame-based DBGrid grid showing result-set data), detail forms (used to view and edit detailed information) and frame-based DBGrids (which show a list of data and allow the user to go to the detail form for a selected record).

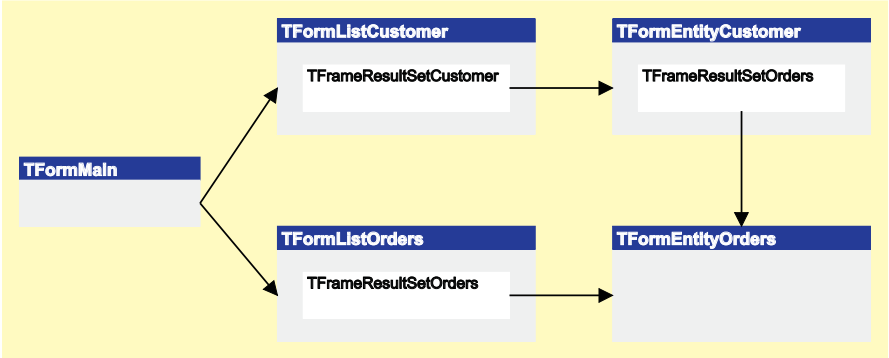
Initially, the application allows selecting and editing of customer and order data from the customer.db and orders.db tables. Figure 2 shows the user's flow of control and the some of the key user interface objects that exist at runtime. Later in the article I'll extend the application to allow selecting and editing of employee data.

Note that lists and the customer entity form contain frames. The application includes a frame class that displays result-set data and allows the user to open a detail form for the selected record. Use of frames facilitates reuse. For example, both the customer list form and the customer entity form contain an order result-set frame. This means the TFrameResultSet functionality to open an entity form is available from both places.

➤ *Figure 1: The sample application data model.*



➤ *Figure 2: Runtime objects and the user's flow of control.*



Our forms sub-class from the common ancestor `TFormBase`, objects sub-class from `TObjectBase`, and frames sub-class from `TFrameResultSetBase`.

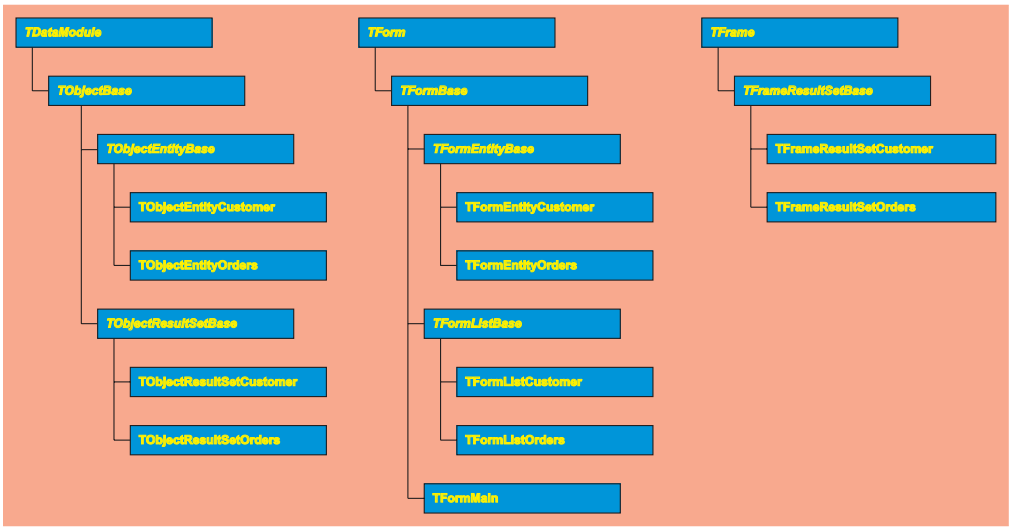
Entity forms provide the user interface for editing a specific business entity, which in this case means editing a customer or an order. Each entity form has a corresponding entity object; it's the entity object that interacts with the database via `TQuery` components. For example, a `TFormEntityCustomer` form has a reference to a `TObjectEntityCustomer`, the form is the user interface for viewing and manipulating the data provided by the object.

The application also features 'result set frames', which are frames containing database grids used to show a result set. These frames also have a corresponding object: `TFrameResultSetCustomer` is the view for data encapsulated in `TObjecdtResultSetCustomer`, and `TFrameResultSetOrders` is the view for `TObjectResultSetOrders`.

Key Features

The key features of the architecture are separation of business objects from the user interface, use of class methods to manage form and object instances, form inheritance, object inheritance and frames. These features are discussed below.

➤ *Figure 3: The TObjectBase, TFormBase, and TFrameResultSetBase class hierarchy.*



Separation of business objects from the user interface is the 'model-view' approach to application design. The benefits of the model-view approach are well documented in the object oriented literature but, in a nutshell, it reflects the fact that a user interface is more arbitrary than the functionality needed by a business object. Separating them makes the code more de-coupled, which makes modifications and maintenance easier.

The architecture uses two class methods to manage form and object instances. These methods use object class type and a key value to identify an object, and ensure that only a single instance of an object exists.

Form inheritance is used to reuse user-interface functionality. Many developers use form inheritance to introduce methods, and not so much to inherit controls. This is because it can be difficult to modify or extend controls, especially containers, placed on an ancestor form. For example, assume you have a panel on an ancestor form. If sub-classing forms have components on the panel, and you decide to change the ancestor panel to a group box, you've got a problem.

The architecture works by having the programmer implement key methods that provide information used by ancestor classes. For example, ancestor methods contain the code that creates a form's associated entity object by running a programmer-supplied method specifying the object's class.

`TDataModule` is the base class for business objects. Like our use of form inheritance, we'll put functionality in the ancestor and have the programmer implement key methods.

The benefit of sub-classing `TDataModule` over `TObject` is that the programmer can use Delphi non-visual components, such as `TQuery` components, when implementing the sub-class.

Frames are a very good way of getting user-interface reuse. Frames are like custom components in that you are creating a new class with methods and properties, and the programmer using the frame is actually creating an instance. Being able to create methods for a frame means they can have complex behavior, which makes them much more useful than component groups. Furthermore, since the programmer is using an instance of your frame class, changes to the frame are 'seen' wherever the frame is used throughout the application.

Frames have advantages over custom components. Components are pre-compiled. This means that if a component references classes in your application, those references are linked when you compile your component. If you then change the other classes, the Delphi compiler complains that the time stamp of the class when it was linked into the component no longer matches the current time stamp. This makes components pretty useless for reusing application-specific functionality. Frames, on the other hand, are treated like any other unit in your program: they are recompiled as needed. This, along with avoiding issues relating to design-time packages, gives frames a huge practical advantage over custom components.

Key Omissions And Simplifications

The architecture is designed to illustrate

concepts, not to be a turnkey solution; however, the architecture is a good basis for an application. For simplicity's sake I've intentionally omitted some key functionality and made some simplifying assumptions as described below.

To make the application robust, `try..except` and `try..finally` blocks should be added throughout the code, especially in class methods used to create objects and when referencing programmer implementations of abstract methods. I've omitted exception handling in the sample application.

The application allows users to view and edit records from the `DBDEMOS` database. However, in practice users would need to have the ability to add records. I'm omitting this functionality to avoid the general topic of transaction handling, and because of the way the architecture identifies an object by its class and key. The key corresponds to the primary key used to fetch data from the database. Therefore, to create a new object you have to somehow determine the key to be used if the record were saved. This is a standard (but thorny) issue, and involves deciding on how and when sequence numbers should be issued by the database.

The class methods used to create and manage object instances use class type and a key to identify an object. The key is an integer, which corresponds to the primary key of the data on the database.

There are several advantages of using a single integer primary key for database tables. First, many DBAs consider it good practice. From a Delphi programmer's perspective, having a single integer primary key means that you have lots of opportunities for saving the key in order to re-fetch a record. For example, if you have a `TTreeView` component that reflects the data from several master-detail tables, you can save each record's key in the `TTreeNode Data` property. (You have to type cast the integer key to be of type `Pointer` to store and fetch the data.) You can then use the key as needed to fetch record data. Many components have a `Data` property, and all have `Tag` properties, either of which can be used for this purpose.

The architecture can be modified to handle concatenated keys by changing the parameters being passed to the object management methods. For example, you could use `array of const` or variant arrays to pass an arbitrary number of key values.

The sample application uses data-aware components. Actually, the architecture can be used with non-data-aware controls, but doing so means you have to write code to keep the user interface in sync with the data in the business object (for an example of doing this see my article *A Delphi Multicaster Class* from Issue 53).

In the sample application there is a one-to-one correspondence between the business objects and `DBDEMOS` tables. That allows me to use 'live result set' queries. However, for most applications, that won't be the case. In a real world application one would typically use `TQuery` and `TUpdateSQL` components (or `TDataSetProvider` and `TClientDataSet` components). See the Delphi documentation for information on these features.

➤ *Listing 1. Class function TFormBase.FetchForm.*

```
class function TFormBase.FetchForm(const aKey: integer): TFormBase;
begin
  // "Self" in a class method referes to the class type
  // See if the form already exists
  Result := FindForm(Self, aKey);
  if (Result = NIL) then begin
    // The form doesn't exist - create it
    Result := Self.PrivateCreate(Application);
    // Add the newly created form to the list of all existing forms
    FListForms.Add(Result);
    Result.Key := aKey;
    // Do special ancestor-level initializations.
    Result.AncestorInitializeForm;
    // Now that everything is set up in the architecture run the
    // sub-classing form's implementation of InitializeForm
    Result.InitializeForm;
  end; // then begin
end;
```

Form And Object Management

In the architecture, programmers are not allowed to directly run a constructor or destructor for `TFormBase` or `TObjectBase` classes. Instead, class methods are used. This is needed because the user may ask to view a form twice. The routine that fetches the form has to determine whether the form already exists. If it does, the user goes to the existing form, if it does not a new form is created. A similar need exists for `TObjectBase` objects.

Forms are created by calling a class method, passing the key of the business entity to be viewed. For example, to view the customer entity form for a given key the programmer codes:

```
TFormEntityCustomer.FetchForm(
  aKey).ShowForm;
```

`ShowForm` is an ancestor method used to restore a minimized form before showing it to the user.

`FetchForm` is a class method that determines whether the form already exists by calling the private ancestor method `FindForm`, and either creates the form or returns a reference to the existing form.

`FindForm` works by looping through a `TList` containing a reference to all forms previously added by `FetchForm`. If the form exists it returns its reference, if it doesn't exist it returns NIL, see Listing 1.

Note that after creating a new form, `FetchForm` runs `InitializeForm` on the new object. `InitializeForm` is a virtual method that forms are free to override. It is run after the form is fully created and ancestor properties initialized. In our architecture, `InitializeForm` is where the form gets an instance of its entity object and associates the object's query with the data source on the form.

Business objects are created by passing the key of the business entity to be viewed to a class

method. For example, to create the customer object for a given key the programmer codes:

```
TObjectEntityCustomer.
  FetchReference(Self, aKey);
```

FetchReference is similar to TBaseForm.FetchForm. However, in the architecture objects are reference counted. This is needed because any part of the user interface is allowed to reference an object. For example, the orders form has a reference to its associated orders object. An order also relates to a customer, so the orders form also has a reference to a customer object. There is a data-aware label on the orders form that is tied to a field on the customer object's query. In other words, after opening an order form, there are two business objects in memory: one for the order and one for the customer: see Figure 4. If the associated customer form were open then there would be two references to the same customer object: one reference for the order form and one for the customer form.

The reference counting is done with a private TList object declared in TObjectBase. As FetchReference is run, and objects are created, the referencing object, passed as Self, is added to the list of referencing objects.

The reference count is decremented when the programmer runs FreeReference(Self). When this is run, the routine checks the list of references and removes the referencing object. If the reference count goes to zero (ie, the TList becomes empty) then the object is destroyed, see Listing 2.

The Business Object Class Hierarchy

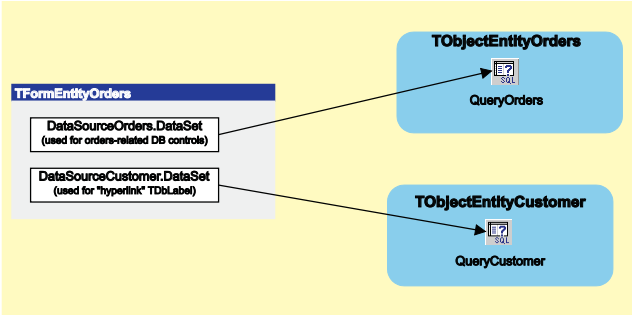
I'm using the term 'business object' to refer to objects which provide entity information, such as information on customers and orders, as well as objects that represent a result set. The entity forms need a customer or orders business object to provide the data being shown on the form. The frames need result set objects to provide data being shown on the frame's TDBGrid.

TObjectBase contains the routines used to create and free objects: FetchReference and FreeReference. TObjectBase also includes database table-related methods such as Save, Cancel, and SaveAll. These could have been placed in the TObjectEntityBase class, since that's where we actually need to save data. However, leaving it in the ancestor gives us the option of adding those operations in other sub-classes if the need arises.

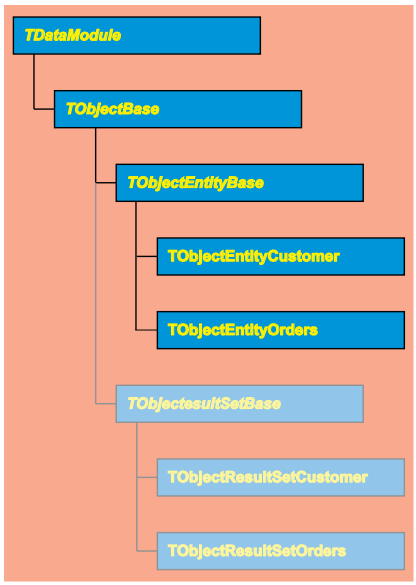
TEntityBaseObject sub-classes from TBaseObject. All entities, such as the customer class and order class, sub-class from TEntityBaseObject. In the sample application there is no functionality added in TEntityBaseObject; in practice you might choose to put methods relating to fetching and saving data here. For example, database transaction control statements could be put at this level.

```
procedure TObjectBase.FreeReference(aReferencingObject: TComponent);
var
  i: integer;
  s: string;
begin
  // Exit if the programmer runs the routine after the reference has
  // already been cleared.
  if (Self = NIL) then exit;
  // Remove the referencing object from the list of referencing objects.
  // If the reference count goes to zero then destroy the object.
  i := FReferencingObjectList.IndexOf(aReferencingObject);
  if (i = -1) then begin
    // If the referencing object isn't found at all then something is wrong.
    // It's probably most appropriate to raise an excaption here, but for
    // debugging purposes it may be best to just display an error message.
    s := 'TObjectBase.FreeReference referencing object not found';
    MessageDlg(s, mtError, [mbOK], 0);
    exit;
  end; // then begin
  FReferencingObjectList.Delete(i);
  if (FReferencingObjectList.Count = 0) then begin
    // There are no references to this object. Free it up.
    FinalizeObject;
    PrivateClassObjectList.Remove(Self);
    AbstractFinalizeObject;
    // Run Destroy in the ancestor because we aren't allowing the programmer
    // to run TObjectBase.Destroy
    inherited Destroy;
  end; // then begin
end;
```

➤ *Listing 2: TObjectBase.FreeReference.*



➤ *Figure 4: The runtime relationship between the orders form and the orders and customer objects it references.*



➤ *Figure 5: The TObjectBase class hierarchy.*

The concrete business entity classes, TObjectEntityCustomer and TObjectEntityOrders, sub-class from TObjectEntityBase. The architecture requires that all concrete entity classes override four

methods: `InitializeObject`, used to do initializations, such as fetching data for the entity; `Save`, which posts changes to the database; `Cancel`, which cancels pending changes; and `UpdatesPending`, which returns true or false depending on whether the user has changed the record.

Frames use the `TObjectBase` sub-class `TObjectResultSetBase` as the source of data being shown on the frames' `TDBGrid`. All frames-related classes are discussed in the section titled *The Frames Class Hierarchy* later in the article.

The Form Class Hierarchy

In this architecture, forms provide the user interface for a business object. The business object contains business rules. This means that the forms themselves need very little code.

Forms are created by calling the class method `FetchForm`, passing the key for the business object being viewed by the form. The form ancestor class creates the business object using information provided by the sub-classing form. The ancestor form also frees up the object as the form is being destroyed.

The architecture requires that the entity forms, `TFormEntityCustomer` and `TFormEntityOrders`, override two methods: `InitializeForm`, used to do initializations, such as associating the form's datasource with the business object's query, and `BusinessObjectClass`, which returns the class type of the associated business object.

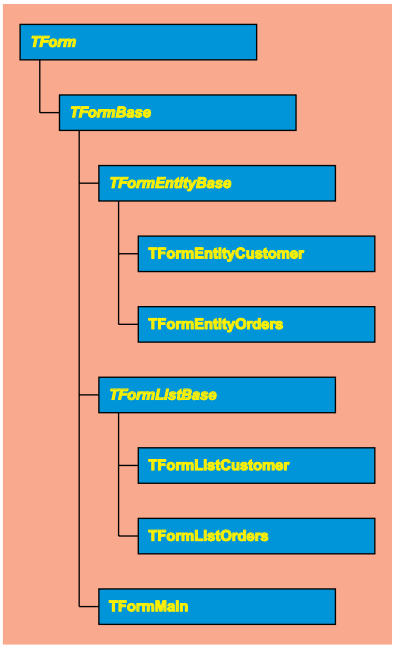
Besides associating the form's data source with the business object's query, `InitializeForm` can be used for other initializations. For example, `TFormEntityCustomer` contains a frame to show the customer's orders. Therefore, `TFormEntityCustomer` uses `InitializeForm` to initialize the frame's `where` clause before activating it, Listing 3. (See the section titled *The Frames Class Hierarchy* for information on how the frames work.)

The orders form also needs to do special processing in `InitializeForm`. In this case the form contains a 'hyperlink' to the associated customer. This was implemented using a `TDBText` component, with the text set to blue and underline, and the cursor set to `crHandPoint`. The label references a field from the query associated with the customer entity. This means the orders form has two references to a business entity: one for the associated order and one for the associated customer (see Figure 4). The form ancestor manages the reference to the order's business object, but since the reference to the customer object is unique to the order form, it is responsible for freeing up the customer object reference.

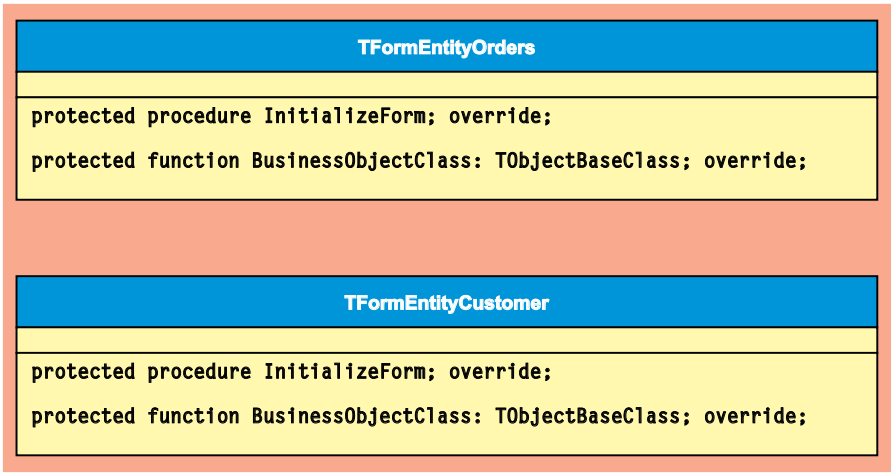
For the orders entity form, we free up the object reference in an optional method called `FinalizeForm`. `FinalizeForm` is run by the architecture immediately before destroying a form. (See Listings 4 and 5.)

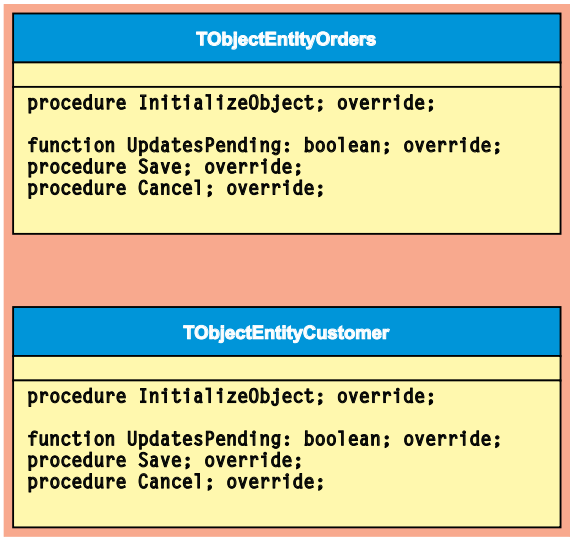
The 'hyperlink' `TDBText` component works by coding an `OnClick` event handler which uses the architecture's form creation method to create the customer form for the associated key, see Listing 6.

➤ *Figure 7: The TFormBase class hierarchy.*



➤ *Figure 8: Classes TFormEntityOrders and TFormEntityCustomer.*





➤ *Figure 6: TObjectEntityCustomer and TObjectEntityOrders.*

List forms do not have an associated business entity. The lists display data via a frame (discussed in the next section). In the case of the sample application on the companion disk, the lists are always shown non-modally. However, I've coded list ancestor methods `ShowModalPickListOKCancel` and `ShowModalPickListApplyClose` to allow you to show the lists modally, if needed. The list ancestor also has an `OnListSelect` method to allow the calling routine to react to the user pressing `OK` or `Apply`. The `ShowModalPickListOKCancel` method shows the list with `OK` and `Cancel` buttons, pressing the `OK` button runs the event handler and closes the form. The method `ShowModalPickListApplyClose` shows the list with `Apply` and `Close` buttons. Pressing `Apply` runs the event handler, but does not close the form. The lists do not provide result set filtering. In other words, the lists show all records from the `customer.db` or `orders.db` tables. At the end of the article I'll enhance the customer list to allow filtering on company name.

➤ *Listing 3: TFormEntityCustomer.InitializeForm.*

```
procedure TFormEntityCustomer.InitializeForm;
begin
  inherited;
  Self.DataSourceCustomer.DataSet := CustomerObject.QueryCustomer;
  FrameResultSetOrders.ResultSetObject.SqlWhereClause :=
    '(CustNo = ' + IntToStr(Key) + ')';
  FrameResultSetOrders.ResultSetObject.RefreshResultSet;
  Caption := 'Customer ' + IntToStr(Key);
end;
```

➤ *Listing 4: TFormEntityOrders.InitializeForm.*

```
procedure TFormEntityOrders.InitializeForm;
var
  aCustomerKey: integer;
begin
  inherited;
  DataSourceOrders.DataSet := OrdersObject.QueryOrders;
  aCustomerKey := OrdersObject.QueryOrders.FieldByName('CustNo').AsInteger;
  FCustomerObject :=
    (TObjectEntityCustomer.FetchReference(Self, aCustomerKey)
    as TObjectEntityCustomer);
  DataSourceCustomer.DataSet := FCustomerObject.QueryCustomer;
  Self.Caption := 'Order ' + IntToStr(Key);
end;
```

➤ *Listing 5: TFormEntityOrders.FinalizeForm.*

```
procedure TFormEntityOrders.FinalizeForm;
begin
  FCustomerObject.FreeReference(Self);
  inherited;
end;
```

➤ *Listing 6: The 'hyperlink' label's OnClick event handler.*

```
procedure TFormEntityOrders.DBText1Click(Sender: TObject);
var
  aCustomerKey: integer;
begin
  inherited;
  aCustomerKey := OrdersObject.QueryOrders.FieldByName('CustNo').AsInteger;
  TFormEntityCustomer.FetchForm(aCustomerKey).ShowForm;
end;
```

➤ *Listing 7: The implementation part of customer result set object.*

```
implementation
{$R *.DFM}
{ TObjectResultSetCustomer }
function TObjectResultSetCustomer.GetSelectedRecordKey: integer;
begin
  Result := Query.FieldByName('CustNo').AsInteger;
end;
function TObjectResultSetCustomer.GetSqlFromClause: string;
begin
  Result := 'customer.db';
end;
function TObjectResultSetCustomer.GetSqlSelectClause: string;
begin
  Result := '*';
end;
end.
```

The Frames Class Hierarchy

The architecture uses frames to show a `TDBGrid` of entity records. These frames also have a separate 'model' and 'view'. In this case, the model is provided by `TObjectResultSetBase`, and its entity-specific versions `TObjectResultSetCustomer` and `TObjectResultSetOrders`, see Figure 5.

`TObjectResultSetBase` has a `RefreshResultSet` method that re-creates an SQL string using methods provided by its sub-classes.

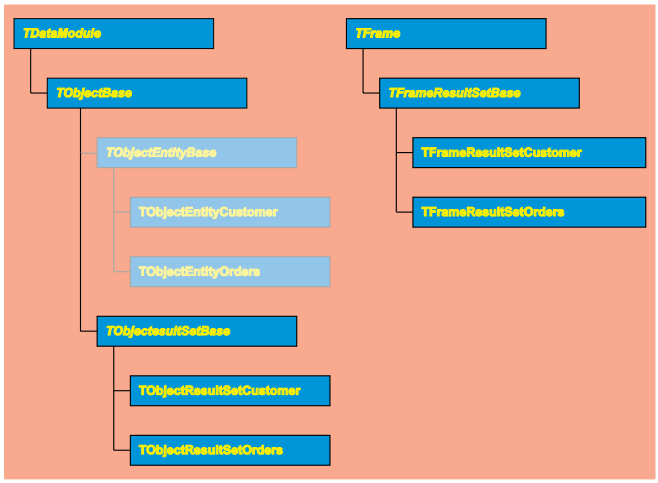
The architecture requires that the sub-classes, `TObjectResultSetCustomer` and `TObjectResultSetOrders`, implement three methods: `GetSqlSelectClause`, used to provide the `select` part of the SQL statement; `GetSqlFromClause`, used to provide the `from` part of the SQL statement; `GetSelectedRecordKey`, used to return the key of the selected record.

The entire implementation part of the customer list is shown in Listing 7.

All of our frames sub-class from `TFrameResultSetBase`.

This class requires that its sub-classes implement two key methods. The first, `GetClassOfFormToBeOpened`, tells the ancestor what form is to be opened when the user chooses the popup menu option `Open`. The second, `GetResultSetClass`, tells the ancestor the class of the associated result set object.

Like other parts of the architecture, only needing to implement these key methods makes the sub-classing programmer's task



➤ *Figure 9: The TObjectResultSetBase and TFrame class hierarchy.*

```
TObjectResultSetBase
property SqlSelectClause: string; (read-only)
property SqlFromClause: string; (read-only)
property SqlWhereClause: string;
property SqlOrderByClause: string;

property SelectedKey: integer; (read-only)

protected function GetSqlSelectClause: string; virtual; abstract;
protected function GetSqlFromClause: string; virtual; abstract;
protected function GetProtectedSqlWhereClause: string; virtual;

protected function GetSelectedRecordKey: integer; virtual; abstract;

public procedure RefreshResultSet;

public function Empty: boolean;
```

➤ *Figure 10: Class TObjectResultSetBase.*

very easy. The entire implementation part of the customer frame is shown in Listing 8.

Extending The Application

Class `TFormEntityBase` has a `TAction` (and associated popup menu item) used to save data by running Save on the associated business entity. One simple enhancement is to add a 'Show hints' option to the action list. This option will be reflected in all sub-classing forms.

The entity form ancestor has a top-aligned tool bar and a bottom-aligned status bar. To implement the 'Show hints' option I set the status bar's `AutoHint` property to `True` and added a `TAction` whose `OnExecute` and `OnUpdate` event handlers are as shown in Listing 9.

If you run the extended version of the sample application you'll see that this functionality is now seen on both entity forms. If we had put that functionality at the `TFormBase` level then the functionality would have been seen on all forms.

Let's say our users want to be able to sort a result set by clicking on a `TDBGrid` column title. To do this we go to the frame ancestor, `TFrameResultSetBase`, and code the following `OnTitleClick` and `OnColumnMoved` event handlers (both events are run when moving a column, which means we need the `OnColumnMoved` event to set a

flag specifying that column sorting not be done in that case), see Listing 10.

This powerful functionality is easily implemented by having the title click event handler set the `SqlOrderByClause` property for the frame's result set object, then refreshing the query. If you run the enhanced version of the sample application you'll see that *all* grids, on the customer and orders list, and the grid of orders on the customer entity form, now have column sorting.

The lists would be more useful if they allowed the user to filter the result set. To add this to the customer list I've added a dropdown combobox with the options `contains`, `matches exactly`, and `starts with` as well as an edit field for company name. I also added an `OnChange` event for the edit field, see Listing 11.

Run the extended sample, enter a company name on the company list, and choose `Find` to see how it behaves.

Adding support for a new entity type takes more coding. However, since almost all the functionality is built into the ancestor classes this involves creating several sub-classes, and implementing fifteen or so program statements.

To implement support for employee we need to sub-class these objects: `TObjectEntityBase`, `TFormEntityBase`, `TObjectResultSetBase`, `TFrameResultSetBase` and `TFormListBase`.

Since these classes are based on `TDataModule`, `TForm`, and `TFrame`, Delphi lists them on the project tab (labelled `CustomerOrders` in the sample) on the `New Items` dialog box. The dialog is accessed via `File | New...` from Delphi's main menu. As we create each sub-class we have to implement the key methods and add a few components, as follows.

- `TObjectEntityEmployee`: Add a `TQuery` and implement four methods (with a total of five program statements).
- `TFormEntityEmployee`: Add the data-aware controls and implement three methods (with a total of four program statements).
- `TobjectResultSetEmployee`: Implement three methods (with a total of three program statements).
- `TFrameResultSetEmployee`: Implement two methods (with a total of two program statements).
- `TFormListEmployee`: Implement one method (with a total of one program statement).

➤ *Listing 8: The implementation part of the customer frame.*

```
implementation
uses UnitObjectResultSetCustomer, UnitFormEntityCustomer;
{$R *.DFM}
{ TFrameResultSetCustomer }
function TFrameResultSetCustomer.GetClassOfFormToBeOpened: TFormBaseClass;
begin
  Result := TFormEntityCustomer;
end;
function TFrameResultSetCustomer.GetResultSetClass: TObjectResultSetClass;
begin
  Result := TObjectResultSetCustomer;
end;
end.
```

➤ *Listing 9: The OnExecute and OnUpdate event handlers.*

```
procedure TFormEntityBase.ActionShowHintsExecute(Sender: TObject);
begin
  inherited;
  Self.ShowHint := not(Self.ShowHint);
end;
procedure TFormEntityBase.ActionShowHintsUpdate(Sender: TObject);
begin
  inherited;
  ActionShowHints.Checked := Self.ShowHint;
end;
```

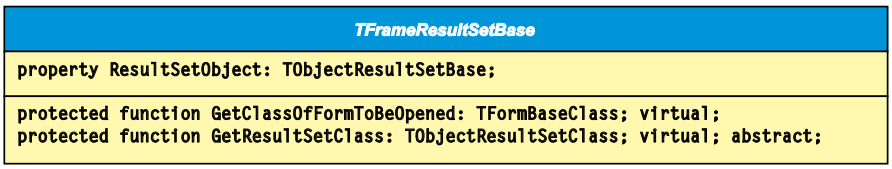
➤ *Listing 10: The OnColumnMoved and OnTitleClick event handlers.*

```
procedure TFrameResultSetBase.DBGridFrameResultSetBaseColumnMoved(Sender:
  TObject; FromIndex, ToIndex: Integer);
begin
  FMovingColumn := TRUE;
end;
procedure TFrameResultSetBase.DBGridFrameResultSetBaseTitleClick(Column:
  TColumn);
begin
  inherited;
  if not(FMovingColumn) then begin
    ResultSetObject.SqlOrderByClause := Column.Field.FieldName;
    ResultSetObject.RefreshResultSet;
  end;
  FMovingColumn := FALSE;
end;
```

➤ *Listing 11: The OnChange event handler.*

```
procedure TFormListCustomer.EditCompanyNameChange(Sender: TObject);
var s: string;
begin
  inherited;
  case ComboBoxCompanyName.ItemIndex of
    0: s := '(company like ''%'+EditCompanyName.Text+'%'')';
    1: s := '(company like '+EditCompanyName.Text+')';
    2: s := '(company like '+EditCompanyName.Text+'%'')';
    else // Unexpected value -- raise an exception or show an error msg.
  end; // case
  FrameResultSetCustomer.ResultSetObject.SqlWhereClause := s;
end;
```

➤ *Figure 11: TFrameResultSetBase.*



We need a way of getting to the list, so we need a new `TAction` (and associated menu item and tool button) on the main form. We also have to remove the new forms, data module, and frame from the auto-create list.

If you run the extended sample application you'll see that the new list works like the other lists. Its frame allows column sorting, `Open` opens the entity form. On the employee form, changes are reflected by the `Save` and `Save All` buttons. In other words, by sub-classing and implementing a handful of statements we get full-featured employee entity support. I did have to set a few properties, like the query's `SQL` and `RequestLive` properties, but adding this functionality was a matter of following a simple routine.

Conclusion

This article has illustrated how an application architecture encapsulates complexity and functionality in ancestor classes. By doing this, programmers extending the application need only sub-class certain classes and implement a few simple methods. The result is much higher programmer productivity, lower maintenance, and better consistency.

Max Rahder (www.rahder.org/max) is an independent consultant living in Madison, Wisconsin. He is a certified Delphi and JBuilder instructor. You can email Max at max@rahder.org