RESPONSIVE SOFTWARE

FRAMEWORK ARCHITECTURE

"Everything should be made as simple as possible, but not simpler." - Albert Einstein.

Purpose of Document

The purpose of this document is to describe the architecture of our Delphi Framework. But before we do this we would like to give a brief outline of the events that led to its development.

Brief History

We were commissioned by a small finance company (5 employees) in 1996 to rewrite the application they were using to record the details of their clients' accounts. This was a DOS-based system that had a number of serious deficiencies (including Y2K incompatibilities!)

Although most of the commercial work we had done at that time was in C++ we have always preferred Pascal because of its superior readability when it comes to understanding coded business logic. Because of our own pleasant experience with Turbo Pascal and because Delphi was new on the market it became the obvious choice for the new system particularly as the client had no strong feelings either way.

The new Delphi finance system went into production in early 1997 when we converted the data from the old system. After some teething troubles (corrupt index, index out of date etc.) our application settled down to give trouble-free running to such a degree that the only time we heard from our client was when they wanted a new feature. In fact a few years passed without us hearing from them at all!

In spite of reports from other developers that Delphi, the BDE and Paradox were inherently unstable our experience taught us that if we program in a consistent manner and understand and treat the BDE correctly we can achieve excellent results.

So in 2003 when we were asked to write a complex system for a fish processing company to track all the fish processed in their plant we again chose Delphi. However this time we were able to approach the project in a somewhat different manner having benefited from the lessons we'd learned from the finance company and also some significant maintenance experience with a number of other business systems.

What we noticed was this. When developing business applications there are a number of things that are common to all applications whether it be finance, fish or any other type of business activity. So once the fish program was completed and operating smoothly at the processing plant we had an idea.

This idea was to make a copy of the source code from the fish program and remove from it everything that had to do with fish. This left us with a compilable and fully functional (albeit limited) business application we called the Framework. (Please note that this was done with the full knowledge and agreement of the fish processing company to whom we owe a debt of gratitude!)

Since that time we have added to the Framework a general ledger, cashbook, point-of-sale screen, document and image repository, HTTP server and the ability to run as a three-tier client/server application with either the BDE or the Firebird/Interbase

database server. We have also used it to successfully develop customized systems for a food warehouse, a furniture manufacturer, a jewellery manufacturer and a firm of consulting actuaries. And it forms the basis of the free Ledger program on our website, our demo Retail system and the new GenFinII integrated finance system. (The GenFinII system has now been used to replace the original Delphi finance system we wrote in 1996.)

Design Philosophy

The basic design philosophy behind the Framework is unashamedly based on the Cathedral model (Fredrick P. Brooks Jr.) as opposed to the Bazaar model (Eric S. Raymond). The Framework is intended to provide a reference point around which a business application of arbitrary complexity can be built while at the same time achieving the conceptual integrity described by Brooks in his seminal book *The Mythical Man-Month*. (We consider this book a must-read for anyone involved in software development.)

The ultimate goal of the Framework is to allow a business to own its own high-quality customized software and source code that will serve as a platform on which to further develop its information systems if it so desires and at the same time avoid being boxed in by the limitations of proprietary software (or overwhelmed by its complexity.) This will allow a competent professional analyst-programmer to effectively meet the needs of a business by working at the code level in direct response to the business's changing requirements.

By using a single language in a single application the Framework is able to minimize the technical complexity associated with building a business application and thereby free the programmer's mind to focus on the complexities inherent in the business problem domain.

Design Details

Passive Database

For the sake of robustness and portability we use only the most basic of database and SQL features in the Framework. This should allow any SQL-compatible database to be used with only minor changes. We use Paradox tables and the BDE by default with an option to convert to Firebird/Interbase should this become necessary either for the sake of performance or because of the sheer quantity of data being stored.

Variable length textual data is stored by our application in a separate table in the database that automatically splits it into fixed-length strings. The data for the document and image repository is stored in binary files in a single folder outside the database. This means the use of variable length or BLOB data types can be avoided. Business logic is also handled by the application (client or server) so that no use is made of triggers or stored procedures.

We believe the use of the advanced features of a database should be a deliberate design decision taken after a careful consideration of both the costs and the benefits.

Please note however that there is nothing in the Framework that precludes the use of the advanced features of a database should this become necessary or desirable.

Database Design

Every table in the database created by the Framework uses a 32-bit integer (Paradox) or a 64-bit integer (Firebird) primary key that contains a database-unique value. We believe this to be the most flexible and (dare we say it) correct way to design a database and cannot see any reason why this should not be mandatory when developing interactive business applications. The fact that it is database-unique allows the application to manage database records from multiple tables as polymorphic object collections in memory.

Within the Framework every table has a corresponding class derived from a common base class named TDatabaseObject. This base class contains all the virtual methods for generic handling of database tables including the loading of data from and the saving of data to a table, the streaming of data over a TCP/IP connection and the restructuring of a table when adding additional fields.

The Framework also includes a class named TDatabaseObjectCollection that is used to hold collections of objects derived from TDatabaseObject.

Database Access

Data is retrieved from and updated to the database using the generic methods provided by TDatabaseObject and TDatabaseObjectCollection in addition to a number of utility functions. The primary key field is used to uniquely identify an object in memory with a record in the database. Whenever an object is saved to the database this field is checked and if it has not been set (i.e. it is still zero) it is assigned a new unique value and a new record is created in the table. If it has already been set this indicates it is not a new object and the existing record in the table is updated.

Database Maintenance

The creation of database tables and the addition of new fields to tables is handled automatically by the Framework application each time it is started. If a table does not exist it is created. It will also check for the existence of new fields in existing tables and create these if they do not already exist.

These functions allow for the easy deployment of new versions of an application into production because the application itself takes care of any database restructuring required. When adding a new table to the database the programmer simply derives a new class from TDatabaseObject and provides implementations for the appropriate virtual methods LoadFromDatabase, SaveToDatabase, LoadFromStream, SaveToStream etc. that were declared in the base class.

Business Logic

Business logic in the Framework that relates directly to a database table/entity is coded as a method of the corresponding class derived from TDatabaseObject. Business logic that is not directly related to a class is coded either as a global utility procedure or placed within the user interface logic where it belongs.

Single Executable

The Framework compiles to a single executable that is guaranteed to run on any version of Windows (including 95.) It therefore makes an ideal platform on which to develop a business application that can be distributed as a package to unknown users who could be using any version of Windows. The same executable runs in different modes based on simple command-line parameters supplied at the time the application is started (via shortcut properties.)

Minimal Dependence on Third Party Components

Apart from the standard Delphi components the only third party component used by the Framework is QuickReports as distributed with Delphi 6.

Single SDI Window

The main form of the Framework is an SDI window that is visible on a screen resolution of 800x600. No scaling of controls or fonts is performed under different screen resolutions in order to simplify the development task. Our experience with scaling has shown us that it can absorb an inordinate amount of time and should only be attempted if a client specifically requires it. In most cases we have found that the standard Windows fonts and control sizes are perfectly adequate for the vast majority of business applications.

All the data entry screens are treated as separate frame components on the main form and displayed or hidden as required in response to the user clicking on the navigation control or pressing a shortcut key. The state of each screen is preserved so the user can navigate to other screens within the application at the same time as a data entry operation is in progress.

Navigation Control

A tree view component is used as a navigation control on the left of the main form. The frames for the data entry screen are made visible or invisible in response to movements within the navigation tree view. This allows additional entry screens to be added as required and for related entry screens to be grouped together in a hierarchical fashion in the navigation control.

Data Entry Screens

The data entry screens are frame components on the main form and are designed to appear as simple and uncluttered as possible. The normal layout includes a bold heading line at the top indicating the purpose of the screen and buttons at the bottom

for New, Edit, Cancel, Save and so on. However these could be laid out in any manner suitable for the application. In the situation where there is a header entity and associated details in a one-to-many relationship (e.g. an invoice with a list of items) the details will often be displayed in a grid using a string grid component.

The most significant thing to note about the data entry screens is that no use is made of the data-aware components. All database access is performed using the generic methods and utilities provided. This provides a clear separation between the user interface and the database and allows the latter to be located on a local machine on the LAN or on a remote computer via a TCP/IP connection with the user interface unaware of the actual location of the data.

Object Find Screen

A generic object find screen is used to allow a list of objects/records to be displayed for the user to make a selection. The TDatabaseObject class provides virtual methods that must be implemented in the derived class in order to specify how an object/record should be displayed in the object find screen.

Object Maintenance Screen

A generic object maintenance screen is used to allow a list of objects/records to be displayed for the user to make changes. The TDatabaseObject class provides virtual methods that must be implemented in the derived class in order to specify how an object/record should be displayed in the object maintenance screen and how each field should be edited/updated.

Quick Reports

All reports are created using QuickReports components. A generic object listing report is used to display a list of objects/records. The TDatabaseObject class provides virtual methods that must be implemented in the derived class in order to specify how an object/record should be displayed on this report.

Modal Operation

The Framework application consists of a single compiled executable that runs in any one of various modes indicated by command line switches. These different modes are standard mode, client mode, server mode, POS mode, POS offline mode and conversion mode. Although this may sound complicated it actually has the effect of simplifying both the development and deployment of an application and creating a very robust process for supporting the application in a production environment.

Standard Mode

Standard mode is the default mode where the application runs as a stand-alone application and accesses a Paradox database directly via the BDE. This mode is the easiest to use when developing the application from within the IDE. It also makes for a very simple installation procedure when providing a demonstration or packaged application via the Internet as is the case with the free Ledger program on our website.

The user can just install the application and then run it directly from a single shortcut on the desktop.

Client Mode (/c)

From the user's perspective client mode appears almost indistinguishable from standard mode apart from a subtle but significant difference. On start-up the user will be prompted to enter the IP address of the computer on which the server application is running (actually another instance of the same application in server mode,) the port number the server is using to accept client connections, and the user name and password.

Once connected to the server the application will function in the same manner as if it was in standard mode except for a potential delay caused by the speed limitation of the data connection whenever data is being retrieved from or sent to the database.

Server Mode (/s)

In server mode the main form that is displayed in standard and client modes is hidden. Instead another form is displayed that provides status information to the user including the IP address of the computer on which the server is running, the port number that is used to accept client connections, the port number that is used to accept HTTP requests, the number of bytes sent to and received from connected clients, the number of pages/objects supplied in response to HTTP requests and a list of all the connected clients showing user name, IP address and the date and time the connection was established.

POS Mode (/p)

In POS mode the application functions as a connected client in the same manner as client mode. However the main form is not displayed. Instead another form is displayed that provides the user with a special purpose interface allowing them to perform specific functions related to a particular task, in this case the recording of a sale transaction at POS (point-of-sale.) This concept could easily be adapted to any other situation where the user needs only a limited range of functions in order to carry out a particular task without giving them access to all the functions provided in client mode.

If the data connection with the server is lost the application automatically switches to POS offline mode.

POS Offline Mode (/p /o)

In POS offline mode the application functions as if it were in POS mode but without a connection to the server. This allows the user to continue using the application to carry out their task even when there is no communication link to the computer hosting the server application.

This will only work in a situation where all the data required for the task can be conveniently retrieved from the server, maintained on the client and transmitted back

to the server when the communication link becomes available and the application is again operating in POS (connected) mode.

Conversion Mode (/v)

Conversion mode is a special mode used to convert the data stored in Paradox tables to the Firebird/Interbase database server. This allows for an easy upgrade of the database when it reaches a size that becomes difficult to manage using Paradox tables.

Peer-to-peer Communications

When developing an application there is often a need to notify other users that an object has changed in the database. The Framework includes a robust mechanism to allow the details of an object update or deletion to be notified to all the other active instances of the application operating in either Standard mode or Client mode.

Data Encryption

When using the Framework in a client/server configuration all data sent or received over a connection is encrypted using a simple encryption algorithm based on a sequence of random numbers. This level of encryption is normally all that is required to prevent recognition of the data by a casual observer.

If necessary it would be quite straightforward to incorporate a more secure encryption algorithm into the application at the clearly defined points where the data is being transferred between the client and the server.

User Authentication

When starting the Framework in client mode the user is prompted to login by entering their user name and password. The corresponding user name and encrypted password are then retrieved from a table in the database in order to authenticate the user. This table is maintained via the server application. Invalid login attempts are recorded in a log file by the server application on the host computer.

Workstation Configuration

The Framework application uses a TWorkstationConfiguration object to manage the settings that are specific to an individual user or workstation for example the customized control colour chosen by the user. Additional settings can be easily added to this object as required.

Global Configuration

The Framework application uses a TGlobalConfiguration object to manage the settings that have a global effect for example the port numbers used for client/server and HTTP communications. Additional settings can be easily added to this object as required.

Application Registration

The Framework provides a mechanism to generate unique registration codes based on various parameters including the company name, the enabled features of the application, the number of workstations and the software expiry date. This can be used to prevent a user from using an application and/or restrict the use of certain functions until the appropriate licence fees have been paid.

Development Mode

When developing an application it is sometimes convenient to have certain features perform differently than they would in normal production. For example it is often helpful to gain direct access to the underlying tables of the database in order to view or edit the data but giving this access to an end user on a production system could potentially compromise the integrity of the data. We also like to display the amount of allocated memory on the title bar of the main form so that we can check constantly for memory leaks during development.

To accomodate this we have created a special development mode that is enabled by creating a setting in the Windows registry on the computer being used by the programmer. This can be switched on and off as required during development and testing.

Summary

Our main goal in developing the Framework has been to write an entire system structured in a way that can be easily understood, debugged and verified by a single programmer allowing them to have total quality control of any customized system in a production environment.

During the last several years the Framework has proven itself as the basis of a number of mission critical business systems that continue to provide their owners with ongoing trouble-free operation. Our experience has led us to believe it could be used as a platform on which to develop a reliable insurance system, banking system or any other custom-built enterprise-level business system.

Source Files

Ledger.bmp
POS.bmp
Reports.bmp
Splash.bmp

ARW03RT.ICO
FOLDER01.ICO
GRAPH04.ICO
GRAPH06.ICO
POINT04.ICO
SUN.ICO

Framework.cfg
Framework.dof
Framework.dpr(.res)

Accounts.pas(.dfm)
AccountsCacheUnit.pas
AccountStatementReportFormat.pas
AccountStatementReportUnit.pas(.dfm)
AttachmentCacheManagerUnit.pas
BalanceSheet.pas(.dfm)
BalanceSheetReportFormat.pas
BalanceSheetReportUnit.pas(.dfm)
Base.pas(.dfm)
BaseFrameUnit.pas(.dfm)
BusinessObjects.pas
Cashbooks.pas(.dfm)
CashbooksCacheUnit.pas
CashbookStatementReportFormat.pas
CashbookStatementReportUnit.pas(.dfm)
ChooseString.pas(.dfm)
ClientCommunicatorUnit.pas
CommunicationsManager.pas
CommunicatorUnit.pas
Compress.pas
Config.pas(.dfm)
Controls.pas (Bug fix to Borland's code)
DatabaseManager.pas
DatabaseObjects.pas
Documents.pas(.dfm)
Entries.pas(.dfm)
FTP.pas(.dfm)
GeneralUtilities.pas
Globals.pas
Graph.pas(.dfm)
GraphReportUnit.pas(.dfm)
HTTPResponder.pas
HTTPServerCommunicatorUnit.pas
HTTPUtilities.pas
IBSQL.pas (Bug fix to Borland's code)
IncomeStatement.pas(.dfm)
IncomeStatementReportFormat.pas
IncomeStatementReportUnit.pas(.dfm)
Items.pas(.dfm)
Ledger.pas(.dfm)
Main.pas(.dfm)
POS.pas(.dfm)
POSConfig.pas(.dfm)
POSMain.pas(.dfm)
Progress.pas(.dfm)
PromptAccountType.pas(.dfm)

PromptDate.pas(.dfm)
PromptHostNameUserIdPassword.pas(.dfm)
PromptPaymentType.pas(.dfm)
PromptSearchString.pas(.dfm)
PromptString.pas(.dfm)
PromptUserIdPassword.pas(.dfm)
PromptUserNamePassword.pas(.dfm)
ProxyDatabaseCollectionObjectFind.pas(.dfm)
ProxyDatabaseCollectionObjectMaintain.pas(.dfm)
ProxyDatabaseObjectCollectionUnit.pas
ProxyObjectListingReportUnit.pas(.dfm)
ReceiptReportFormat.pas
ReceiptReportUnit.pas(.dfm)
Register.pas(.dfm)
RegistrationInfo.pas(.dfm)
Reports.pas(.dfm)
Sales.pas(.dfm)
SalesManagerUnit.pas
SalesReportFormat.pas
SalesReportFrameUnit.pas(.dfm)
SalesReportUnit.pas(.dfm)
ServerCommunicatorUnit.pas
ServerMain.pas(.dfm)
ServerTest.pas
Splash.pas(.dfm)
Utilities.pas

Sample Code

```
{*******************************************************}
{                                                       }
{  Responsive Software     http://www.responsive.co.nz  }
{                                                       }
{  Copyright (c) 2003-2006 Responsive Software Limited  }
{                                                       }
{*******************************************************}

unit ProxyDatabaseObjectCollectionUnit;

interface

uses
  Classes, DBTables, DB, IBDatabase,
  DatabaseObjects;

type
  TProxyDatabaseObject = class;

  // this is used to provide generic access to any collection of database objects
  // without requiring the objects to be loaded until they are required
  TProxyDatabaseObjectCollection = class
  private
    FDatabaseObjectClass : TDatabaseObjectClass;
    FDatabaseObjectCollection : TDatabaseObjectCollection;
    FProxyDatabaseObjects : TList;
    FDataset : TDataset;
    FCount : integer;
    function GetCount : integer;
    function ObjectLoaded
      (i : integer) : boolean;
```

```
    function GetObject
      (i : integer) : TDatabaseObject;
    function OpenLocalDataset
      (SelectionString : string) : integer;
    procedure CloseLocalDataset;
    function GetDatasetRecNoOffset
      (RecNo : integer) : integer;
  public
    constructor Create
      (DatabaseObjectClass : TDatabaseObjectClass;
       DatabaseObjectCollection : TDatabaseObjectCollection;
       SelectionString : string); overload;
    destructor Destroy; override;

    procedure InsertObject
      (Index : integer;
       DatabaseObject : TDatabaseObject);
    procedure DeleteObject
      (Index : integer);

    property Count : integer read GetCount;
    property Objects[i : integer] : TDatabaseObject read GetObject; default;
  end;

  TProxyDatabaseObject = class
  private
    FDatabaseObject : TDatabaseObject;
    FDatasetRecNo : integer;
    FRecNo : integer;
  public
    constructor Create
      (DatabaseObject : TDatabaseObject;
       DatasetRecNo : integer;
       RecNo : integer);
    destructor Destroy; override;
    function DatabaseObject : TDatabaseObject;
    function DatasetRecNo : integer;
    function RecNo : integer;
    function DatasetRecNoOffset : integer;
    procedure IncrementRecNo;
    procedure DecrementRecNo;
  end;

implementation

uses
  GeneralUtilities, Utilities, Globals, IBQuery,
  SysUtils, Dialogs, Controls;

{***** TProxyDatabaseObjectCollection methods *********************************}

constructor TProxyDatabaseObjectCollection.Create
  (DatabaseObjectClass : TDatabaseObjectClass;
   DatabaseObjectCollection : TDatabaseObjectCollection;
   SelectionString : string);
begin
  inherited Create;
  FDatabaseObjectClass := DatabaseObjectClass;
  // if collection supplied then use this and ignore the selection string
  if DatabaseObjectCollection <> nil then
    FDatabaseObjectCollection := DatabaseObjectCollection
  // otherwise create a proxy object collection
  else begin
    FProxyDatabaseObjects := TList.Create;
    if ClientMode then
      FCount := ClientCommunicator.OpenRemoteDataset
        (DatabaseObjectClass,SelectionString)
    else
      FCount := OpenLocalDataset(SelectionString);
  end;
end;

destructor TProxyDatabaseObjectCollection.Destroy;
begin
  // destroy proxy object collection
  DestroyList(FProxyDatabaseObjects);
  if FDatabaseObjectCollection = nil then
```

```
  begin
    if ClientMode then
      ClientCommunicator.CloseRemoteDataset
    else
      CloseLocalDataset;
  end;
end;

function TProxyDatabaseObjectCollection.GetCount : integer;
begin
  if FDatabaseObjectCollection <> nil then
    Result := FDatabaseObjectCollection.Count
  else
    Result := FCount;
end;

function TProxyDatabaseObjectCollection.ObjectLoaded
  (i : integer) : boolean;
var
  j : integer;
begin
  for j := 0 to FProxyDatabaseObjects.Count - 1 do
    if TProxyDatabaseObject(FProxyDatabaseObjects[j]).RecNo = i + 1 then
    begin
      Result := true;
      Exit;
    end;
  Result := false;
end;

function TProxyDatabaseObjectCollection.GetObject
  (i : integer) : TDatabaseObject;
var
  j : integer;
  ProxyDatabaseObject : TProxyDatabaseObject;
  DatabaseObject : TDatabaseObject;
  RecNo : integer;
  DatabaseObjectCollection : TDatabaseObjectCollection;
  DatasetRecNoOffset : integer;
begin
  Result := nil;
  // if we are using a supplied collection then
  // just return the corresponding object in this
  if FDatabaseObjectCollection <> nil then
  begin
    Result := FDatabaseObjectCollection[i];
    Exit;
  end else begin
    if (i<0) or (i>=FCount) then
      Exit;
    // look to see if this object is already loaded
    for j := 0 to FProxyDatabaseObjects.Count - 1 do
    begin
      ProxyDatabaseObject := TProxyDatabaseObject(FProxyDatabaseObjects[j]);
      if ProxyDatabaseObject.RecNo = i + 1 then
      begin
        Result := ProxyDatabaseObject.DatabaseObject;
        Exit;
      end;
    end;
    // determine the offset for this record number between
    // the proxy collection and the dataset record number
    DatasetRecNoOffset := GetDatasetRecNoOffset(i+1);
    // load and add the next twenty records to the collection
    // in anticipation that they may be needed
    if ClientMode then
    begin
      DatabaseObjectCollection := TDatabaseObjectCollection.Create;
      ClientCommunicator.LoadDatabaseObjectsFromRemoteDataset
        (DatabaseObjectCollection,i+1+DatasetRecNoOffset,i+20+DatasetRecNoOffset);
      // transfer ownership of objects to proxy collection
      DatabaseObjectCollection.Owned := false;
      for j := 0 to DatabaseObjectCollection.Count - 1 do
      begin
        RecNo := i + 1 + j;
        DatabaseObject := DatabaseObjectCollection[j];
        if not ObjectLoaded(i+j) then
```

```
        begin
          ProxyDatabaseObject := TProxyDatabaseObject.Create
            (DatabaseObject,RecNo+DatasetRecNoOffset,RecNo);
          FProxyDatabaseObjects.Add(ProxyDatabaseObject);
        end else
          DatabaseObject.Free;
        if RecNo = i + 1 then
          Result := DatabaseObject;
      end;
      DatabaseObjectCollection.Free;
    end else begin
      for j := 0 to 19 do
      begin
        if not ObjectLoaded(i+j) then
        begin
          RecNo := i + 1 + j;
          if RecNo+DatasetRecNoOffset <= FDataset.RecordCount then
          begin
            FDataset.RecNo := RecNo+DatasetRecNoOffset;
            DatabaseObject := FDatabaseObjectClass.Create;
            DatabaseObject.LoadFromTable(FDataset);
            ProxyDatabaseObject := TProxyDatabaseObject.Create
              (DatabaseObject,RecNo+DatasetRecNoOffset,RecNo);
            FProxyDatabaseObjects.Add(ProxyDatabaseObject);
            if RecNo = i + 1 then
              Result := DatabaseObject;
          end;
        end;
      end;
    end;
  end;
end;

function TProxyDatabaseObjectCollection.OpenLocalDataset
  (SelectionString : string) : integer;
begin
  if Firebird then
    FDataset := FDatabaseObjectClass.OpenIBQuery(SelectionString)
  else
    FDataset := FDatabaseObjectClass.OpenQuery(SelectionString);
  Result := FDataset.RecordCount;
end;

procedure TProxyDatabaseObjectCollection.CloseLocalDataset;
begin
  if FDataset <> nil then
  begin
    FDataset.Active := false;
    if Firebird then
    begin
      if TIBQuery(FDataset).Transaction.InTransaction then
        TIBQuery(FDataset).Transaction.Commit;
      TIBQuery(FDataset).Transaction.Free;
    end;
  end;
  FDataset.Free;
end;

function TProxyDatabaseObjectCollection.GetDatasetRecNoOffset
  (RecNo : integer) : integer;
var
  i : integer;
  ProxyDatabaseObject : TProxyDatabaseObject;
  SelectedProxyDatabaseObject : TProxyDatabaseObject;
  Offset : integer;
begin
  SelectedProxyDatabaseObject := nil;
  // find the proxy object with the record number immediately
  // preceding the one we are about to load
  for i := 0 to FProxyDatabaseObjects.Count - 1 do
  begin
    ProxyDatabaseObject := TProxyDatabaseObject(FProxyDatabaseObjects[i]);
    // ignore any inserted records which do not have corresponding entries
    // in the dataset
    if ProxyDatabaseObject.DatasetRecNo = 0 then
      continue;
    // find the one with the maximum record number not exceeding
```

```
    // the one specified
    if (ProxyDatabaseObject.RecNo <= RecNo) and
       ( (SelectedProxyDatabaseObject = nil) or
         (ProxyDatabaseObject.RecNo > SelectedProxyDatabaseObject.RecNo) ) then
      SelectedProxyDatabaseObject := ProxyDatabaseObject;
  end;
  // return the offset after adjusting for inserted records
  if SelectedProxyDatabaseObject <> nil then
  begin
    Offset := SelectedProxyDatabaseObject.DatasetRecNoOffset;
    // adjust for any inserted records
    for i := 0 to FProxyDatabaseObjects.Count - 1 do
    begin
      ProxyDatabaseObject := TProxyDatabaseObject(FProxyDatabaseObjects[i]);
      if (ProxyDatabaseObject.DatasetRecNo = 0) and
        (ProxyDatabaseObject.RecNo > SelectedProxyDatabaseObject.RecNo) and
        (ProxyDatabaseObject.RecNo < RecNo) then
        Inc(Offset);
    end;
    Result := Offset;
  end else
    Result := 0;
end;

procedure TProxyDatabaseObjectCollection.InsertObject
  (Index : integer;
   DatabaseObject : TDatabaseObject);
var
  RecNo : integer;
  i : integer;
  ProxyDatabaseObject : TProxyDatabaseObject;
begin
  if FDatabaseObjectCollection <> nil then
    FDatabaseObjectCollection.Insert(Index,DatabaseObject)
  else begin
    RecNo := Index + 1;
    // increment all larger record numbers to make room
    for i := 0 to FProxyDatabaseObjects.Count - 1 do
    begin
      ProxyDatabaseObject := TProxyDatabaseObject(FProxyDatabaseObjects[i]);
      if ProxyDatabaseObject.RecNo >= RecNo then
        ProxyDatabaseObject.IncrementRecNo;
    end;
    // add the new object via a proxy object
    ProxyDatabaseObject := TProxyDatabaseObject.Create
      (DatabaseObject,0,RecNo);
    FProxyDatabaseObjects.Add(ProxyDatabaseObject);
    Inc(FCount);
  end;
end;

procedure TProxyDatabaseObjectCollection.DeleteObject
  (Index : integer);
var
  RecNo : integer;
  i : integer;
  ProxyDatabaseObject : TProxyDatabaseObject;
begin
  if FDatabaseObjectCollection <> nil then
    FDatabaseObjectCollection.Delete(Index)
  else begin
    RecNo := Index + 1;
    // remove matching record number and decrement all larger record numbers
    for i := FProxyDatabaseObjects.Count - 1 downto 0 do
    begin
      ProxyDatabaseObject := TProxyDatabaseObject(FProxyDatabaseObjects[i]);
      if ProxyDatabaseObject.RecNo = RecNo then
      begin
        FProxyDatabaseObjects.Delete(i);
        ProxyDatabaseObject.Free;
      end else if ProxyDatabaseObject.RecNo > RecNo then
        ProxyDatabaseObject.DecrementRecNo;
    end;
    Dec(FCount);
  end;
end;
```

```
{***** TProxyDatabaseObject methods ****************************************}

constructor TProxyDatabaseObject.Create
  (DatabaseObject : TDatabaseObject;
   DatasetRecNo : integer;
   RecNo : integer);
begin
  FDatabaseObject := DatabaseObject;
  FDatasetRecNo := DatasetRecNo;
  FRecNo := RecNo;
end;

destructor TProxyDatabaseObject.Destroy;
begin
  FDatabaseObject.Free;
end;

function TProxyDatabaseObject.DatabaseObject : TDatabaseObject;
begin
  Result := FDatabaseObject;
end;

function TProxyDatabaseObject.DatasetRecNo : integer;
begin
  Result := FDatasetRecNo;
end;

function TProxyDatabaseObject.RecNo : integer;
begin
  Result := FRecNo;
end;

function TProxyDatabaseObject.DatasetRecNoOffset : integer;
begin
  if FDatasetRecNo <> 0 then
    Result := FDatasetRecNo - FRecNo
  else
    Result := 0;
end;

procedure TProxyDatabaseObject.IncrementRecNo;
begin
  Inc(FRecNo);
end;

procedure TProxyDatabaseObject.DecrementRecNo;
begin
  Dec(FRecNo);
end;

end.
```