Cascading Exceptions

by Andrew McLellan

Anyone who has tried to build a large Delphi application using exception handling will have faced the problem of where to trap exceptions and how to report them to the user. Delphi's VCL is built around the premise that anything may raise an exception and your code must be able to handle reporting in a meaningful way as well as cleaning up any resource allocations made. This is particularly true when dealing with the BDE and especially so when upsizing to a Client/Server environment where the error messages returned by remote services can be less than useful to the user (Figure 1) but the bland message that the user needs to see is of little use to the developer or technical support who need to understand the error.

In an ideal environment, an exception needs to be handled so that:

- The user is told that there is a problem in terms that they will understand.
- Sufficient information is given for technical support to understand the problem without sending along a programmer with a notebook PC.
- The user is given some idea of what they must do to correct the problem.

One way of showing the user all the available error information is to append it to the error string, which can be arbitrarily large (Listing 1).

But this is unsatisfactory as, in practice, there is a limit as to how complex you will want the error message to be. Some of the BDE errors returned by Client/Server applications are daunting and it is difficult to add extra information that assists technical support without baffling the user.

The simplest way to show all this error information is to use a stringlist in place of an ever expanding string and to be able to let the user view the error messages in ever increasing detail. Users of Paradox (and Database Desktop) will recognize the results.

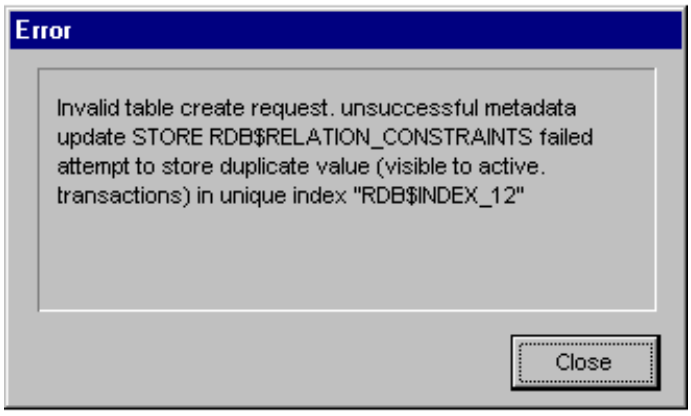
The Cascading Exception System

The intent of the cascading exception handler discussed in this article is to provide structured error information to the user without hiding the underlying error from the developer. A simple case, where a programming error has left the table name unset, produces the dialog shown in Figure 2 and pressing the `Previous` button gives the next error, as shown in Figure 3. The level of complexity of

➤ *Listing 1*

```
Procedure OpenCustomerTable;
Begin
  Try
    CustomerTable.Active := True;
  Except
    On E : Exception Do
      Raise Exception.Create('Error opening Customer table : ' + E.Message);
  End;
End;
```

➤ *Figure 1*



➤ *Below: Left: Figure 2*
Right: Figure 3

this error nesting is arbitrary and will depend on the application.

Three Layers Or More

For any non-trivial Delphi application, there are three or more layers of code in which exceptions can be raised and handled. At the top, there is the user interface layer in which error handling is simple, comprising a dialog with the user. At the bottom, there are the exceptions reported by the BDE and VCL. But the middle layer (your application code) is the one that needs most thought in design.

Consider the case where you have a table on a remote server and that table has primary and secondary indexes. For your application to successfully open the table, suppose that the table must exist, must have at least one record in it and that both indexes must be valid. This functionality needs to be packaged into a single method that does all four validity tests and presents the user with valid information should any test fail, while not discarding any error information that is useful to the support/development staff.

One Exception Handler

Additionally, we need to future-proof our code. If, when designing the code to run interactively, we trap the exception raised by the VCL failing to open the table and present a dialog box to the user, then we cannot convert the application to run as a background thread or as a batch application without substantial re-coding.

If all the exceptions go to a single, application-wide, exception handler, then converting the application to run as a batch operation is simply a matter of changing one routine to write the errors to a log file instead of showing them to the user. The solution is the cascading exception handler.

Exceptions Are Objects

Delphi is, of course, an object oriented language and exceptions are objects. We can build on the existing exception objects to add functionality of our own, as with any other object.

```
Type
  EError = Class(Exception)
    Messages : TStringList;
    Destructor  Destroy; Override;
    Procedure   Add(Const S : String);
  End;
Destructor EError.Destroy;
Begin
  Messages.Free;
  Inherited;
End;
Procedure EError.Add(Const S : String);
Begin
  If Not Assigned(Messages) Then
    Messages := TStringList.Create;
  Messages.Add(S);
End;
```

➤ *Listing 2*

```
procedure NewError(E : Exception; Const NewMessage : String);
Var NewE : EError;
Begin
  NewE := EError.Create(NewMessage);
  If Assigned(E) Then Begin
    NewE.Messages := TStringList.Create;
    If (E Is EError) And (EError(E).Messages <> Nil) Then
      NewE.Messages.Assign(EError(E).Messages);
    NewE.Messages.Add(E.Message);
  End;
  Raise NewE;
End;
```

➤ *Listing 3*

The `Exception` class is defined in `SysUtils` and has eight constructors for raising an exception in different ways, with or without a call to `Format`, with the option to look for the error message in a resource file and with optional help. In this example, we'll ignore all the constructors other than the simple `Create(Msg : String)`.

An exception is raised by Delphi creating an exception instance, before trawling up through the `Try..Finally` blocks until an exception handler is reached. This can either be an `Except..End` block or the application's handler, which can be either the application's default handler or one supplied by you.

In an `Except..End` block we have three choices of what to do with the exception: we can handle it, re-raise it or raise a new exception. It is this last method that we will use: exceptions raised in low-level routines will be trapped and re-raised as our exception class, `EError`, which behaves exactly as a 'normal' exception (which is reasonable, as it *is* a normal exception) but which carries a larger payload.

EError

There are four steps to adding the `EError` class to your application:

- Writing the new `EError` class.
- Writing a procedure to call it.
- Adding an exception handler to your main form.
- Displaying 'normal' and `EError` exceptions in that exception handler.

This only needs to be done once: Delphi 2.0's Object Repository allows us to save the code and re-use it in all subsequent applications.

Creating the new exception requires little work (see Listing 2). The `EError` class reports the highest level error message in `EError.Message` and subsequent, lower level, error messages in the `Messages StringList`. However, we will rarely want to create an instance of `EError` directly. For this we use the procedure (not a method of an object) shown in Listing 3.

This code instantiates an `EError` exception, `NewE`. If the exception

```
Unit FErrorForm;
Interface
Uses
  SysUtils, Windows, Messages, Classes, Graphics, Controls,
  Forms, Dialogs, ExtCtrls, StdCtrls, Buttons, Errors;
Type
  TErrorForm = class(TForm)
    ErrorLabel: TLabel;
    Bevel1: TBevel;
    PreviousButton: TButton;
    NextButton: TButton;
    CloseButton: TButton;
    procedure FormActivate(Sender: TObject);
    procedure PreviousButtonClick(Sender: TObject);
    procedure NextButtonClick(Sender: TObject);
  Private
    ErrPos :    Integer;
    FError :    EError;
    Procedure   Buttons;
    Procedure   Messages;
    Procedure   Position;
  Public
    Property    Error : EError Write FError;
  End;
Var
  ErrorForm: TErrorForm;
Procedure ShowError(E : Exception);

Implementation
{$R *.DFM}

procedure TErrorForm.FormActivate(Sender: TObject);
begin
  If Assigned(FError.Messages) Then
    ErrPos := FError.Messages.Count
  Else
    ErrPos := 0;
  Position;
  Buttons;
  Messages;
  Screen.Cursor := crDefault;
end;

Procedure TErrorForm.Buttons;
Begin
  PreviousButton.Enabled := ErrPos <> 0;
  NextButton.Enabled := Assigned(FError.Messages) And
    (ErrPos < FError.Messages.Count);
  ActiveControl := CloseButton;
End;

Procedure TErrorForm.Messages;
Begin
  If Not Assigned(FError.Messages) Or (ErrPos = FError.Messages.Count) Then
    ErrorLabel.Caption := FError.Message
  Else
    ErrorLabel.Caption := FError.Messages[ErrPos];
End;

Procedure TErrorForm.Position;
Begin
  Left := (Screen.Width - Width) Div 2;
  Top := (Screen.Height - Height) Div 2;
End;

procedure TErrorForm.PreviousButtonClick(Sender: TObject);
begin
  Dec(ErrPos);
  Buttons;
  Messages;
end;

procedure TErrorForm.NextButtonClick(Sender: TObject);
begin
  Inc(ErrPos);
  Buttons;
  Messages;
end;

Procedure ShowError(E : Exception);
Begin
  If Not (E Is EAbort) Then Begin
    If E Is EError Then Begin
      ErrorForm.Error := EError(E);
      ErrorForm.ShowModal;
    End Else
      MessageDlg(E.Message, mtError, [mbOk], 0);
  End;
End;

end.
```

➤ *Left: Listing 4*

we're handling is already an `EError`, then the existing error messages are copied over. Finally, the new exception is raised.

Adding the exception handler to the main form is standard. In the `MainForm`'s `FormCreate` we add the line:

```
Application.OnException :=
  ExceptionHandler;
```

and then write the exception handler itself:

```
Procedure TForm1.ExceptionHandler(
  Sender : TObject; E: Exception);
Begin
  ShowError(E);
End;
```

Now, any exception in our application (which isn't handled deeper in our code) will come to this procedure. `ShowError` is just a wrapper around an error form (Listing 4).

The `ShowError` procedure looks to see what type of exception it is handling. If it is an `Abort`, it does nothing. If it is an `EError`, it displays the `TErrorForm` which has `Previous` and `Next` buttons to allow the user to drill down on the error, and if it is a simple exception, it displays the error in a `MessageDlg`.

We can change the error handler shown in Listing 1 to that shown below:

```
Procedure OpenCustomerTable;
Begin
  Try
    CustomerTable.Active :=
      True;
  Except
    On E : Exception Do
      NewError(E,
        'Error opening '+
        'customer table');
  End;
End;
```

Note that we never free an exception ourselves. The exception passed in to `NewError` is disposed of by Delphi when we no longer need it, and the `MainForm`'s exception handler will dispose of the `EError`.

```
Function FindCustomerName(Const Key : String) : String;
Begin
  Result := '';
  Try
    Assert(Key <> '', 'Customer key is blank');
    With CustomerTable Do Begin
      First;
      While Not EOF Do Begin
        If FieldByName('CustomerKey').AsString = Key Then Begin
          Result := FieldByName('CustomerName').AsString;
          Exit;
        End;
      Next;
    End;
    AssertFail('Failed to find customer from key ' + S);
  Except
    On E : Exception Do
      NewError(E, 'FindCustomerName error');
  End;
End;
```

➤ *Listing 5*

```
Procedure OpenAllTables;
Begin
  Try
    OpenCustomerTable;
    OpenProductTable;
    OpenMasterTable;
  Except
    {$IFNDEF Production}
    On E : Exception Do
      NewError(E,
        'OpenAllTables');
    {$ELSE}
    Raise;
    {$ENDIF}
  End;
End;
```

➤ *Listing 6*

Benefits

The cascading error handler is a boon to developers. In our debug code, every call to the BDE and a great many other procedures is encapsulated in a `Try..Except` block. Most routines have `Asserts` both on input, to check that parameters passed in are valid, and on output, to show that the routine succeeded – see Listing 5.

The result is that errors are reported where they happen (and the `Asserts` act as documentation of valid input and output in a way that comments usually fail to do). This code is left in the debug version permanently (we ship the debug and production versions).

Presenting too much information is nearly as bad as too little. The debug version should report more layers of error information than the production code. Listing 6 shows how to separate exception handling for both versions.

Enhancing Error Reporting

The error handler shown just adds the capability of reporting more than one error, but with object oriented exceptions, there is no limit on what can be reported. You could, for example, provide a bitmap at each level, so that database errors are reported with a picture of a table. One database validation routine we use encapsulates a hierarchy of errors into the extended `EError` class and presents them to the user in a `TreeView`.

Andrew McLellan is the principal developer for teraformation ltd, providing multi-dimensional analysis tools for OLAP databases. All development is in Delphi 2.