Chasing Objectivity With TDBNavigator

by Robert Palomo

A commercial Delphi 1 software project I have been working on raised an interesting problem related to the TDBNavigator component. The application is Multiple Document Interface (MDI) and each of the MDI Child forms enables the user to display and update data from a different table. The MDI parent form features a single navigator control that operates on whatever child form is active on the desktop. Although there are parallel menu commands, the main control for inserting, deleting and posting to the underlying Paradox database is the main form's navigator control.

A twofold problem involving the use of the TDBNavigator component arose in this project. First, if a key violation or other BDE error occurred when the user updated the database, the plain vanilla BDE error message would display. We wanted a way to intercept the error and display a more informative message. Second, if the user attempted to delete a record in violation of the database's referential integrity rules, we wanted to trap that event and present information to the user for analyzing the problem and correcting the data (somehow, merely showing end users the BDE message *Master Has Detail Records. Cannot Delete or Modify* just didn't seem like enough!).

A little research quickly pointed up the fact that the TDBNavigator component has a mind of its own when it comes to firing off calls to the BDE. You cannot intercept a Post or Delete call using the component, there are no properties or events available that enable you to do so. An examination of the DBCTRLS.PAS unit, the VCL source code unit where the TDBNavigator class is located, showed that indeed, the component unconditionally sends off calls (Insert, Post, Delete etc.) to the BDE. I had thought of using the Before/After events of TTable, but this proved not to be an option. It turns out that TDBNavigator's BDE calls reference a TDataSource object, thus bypassing TTable events.

Delphi being the object-oriented animal that it is, the solution seemed easy enough on the surface. I thought that I could derive a new object from the TDBNavigator class and override the method that sends BDE calls, putting in my own code to handle BDE errors. If TDBNavigator were designed with as much object orientation as other Delphi components, that's what I should have been able to do. But, as I'll show you, it's not that simple!

While most components lend themselves to effortless subclassing, the author of TDBNavigator *[Anyone in Borland want to own up to this one...?! Editor]* seems to have taken pains to make it impossible to subclass the component and add any useful functionality to it.

My research led me to conclude that the only practical means of solving my problem without modifying and recompiling the original Borland source code was with a non object oriented approach which I will describe in this article. This was a very surprising conclusion after my previous experience of subclassing other components.

The drawback to duplicating my method is that you can do so only if you have VCL source code. For those of you without access to the code, a compiled version of my modified TDBNavigator accompanies this article on this month's disk. You can install and use it like any other custom VCL, but you won't be able to make any other changes. The remainder of the article will at least help you understand what's happening behind the scenes.

Problems With Inheritance

In the DBCTRLS.PAS source code, the key interaction with the BDE takes place in a method called BtnClick, which is declared as public, which at first glance might make one think that it can be overridden in a descendant object. However, since BtnClick is a static method, rather than a virtual or dynamic method, overriding it in a descendant is out. Further inspection reveals that BtnClick is always called by the Click method, which is declared private! The upshot of this is that in a descendant object, you can't override BtnClick and you can't access Click (except by calling it with the inherited keyword, which doesn't help, as I'll demonstrate).

In working out this solution, I did think about using inherited to try to stay on the true object oriented path. I tried subclassing the component, calling the inherited Click to keep the compiler happy, but placing the call to Click in a protected block to try and trap the BDE error:

```
try
  Inherited Click;
  Except on EDBEngineError do
  { ... error handling code }
end;
```

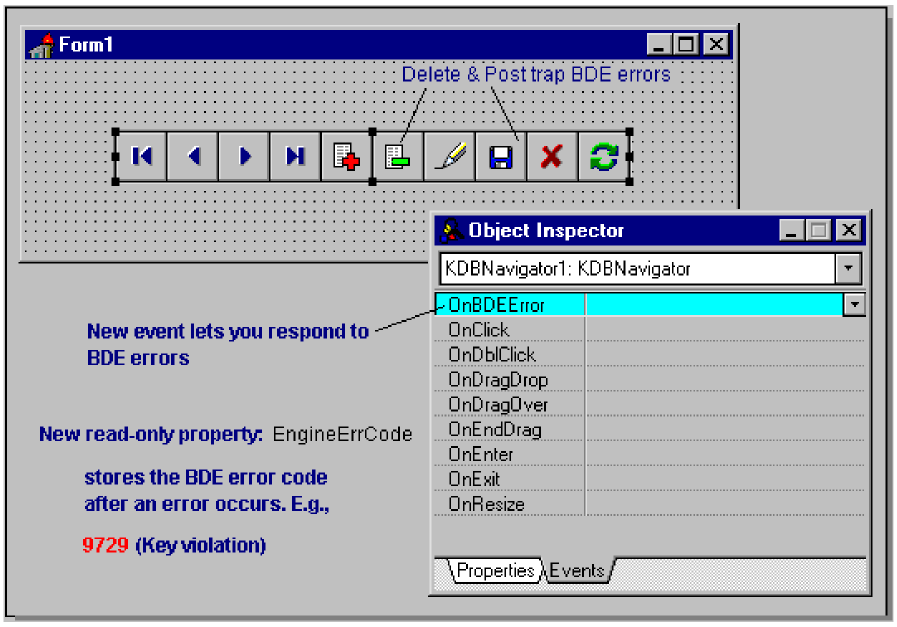
Unfortunately, this didn't help at all. The ancestor component still blithely fires off its calls (Post, Delete etc) to the engine. If a Key or Referential Integrity violation occurs, the system architecture is such that an exception is immediately raised during the execution of the ancestor's method and the plain vanilla engine message pops up right then and there. There is still no opportunity to intercept the error, so the protected block was a wasted effort and I was right back where I started.

By now, I expect you can imagine I was feeling pretty frustrated in coming up with an object oriented solution to this seemingly simple problem. There was one last thing I thought I might try. According to the documentation, you can replace a static method of an ancestor class in a descendant by declaring and coding a method with the same name in the descendant object's unit *[See Jim Cooper's article on Type-Safe Lists last month for examples. Editor]*. So, I thought, why don't we just subclass the component, cut and paste the `Click` and `BtnClick` methods from the `TDBNavigator` section of the DBCTRLS unit into the new unit and modify `BtnClick` to suit our needs? Again, it seemed that someone had gone out of their way to trip me up...

After I'd done the cut and paste job and modified `BtnClick` to handle a BDE error, I found that the last line of that procedure references a field in the ancestor, `FOnNavClick`. Going back and looking at its declaration in DBCTRLS, I found that `FOnNavClick` is declared `private`, meaning there is no way for my descendant object to know about it, so the new component wouldn't compile. Once again, I was SOL *(sorta outta luck)* for a truly object oriented solution.

Non Object Oriented Solution

Despite a great deal of anguished gnashing of teeth and shouts of "there has to be an object oriented solution to this!" amongst my colleagues and myself, the project deadline nonetheless loomed large on the horizon and we needed something that worked and we needed it yesterday. Since we are proud licensees of the VCL source, I regretfully went for a non object oriented hack which at least works and which I'll share with you to the extent I can without actually reprinting the source code which Borland prefers that you buy. If you want to implement this solution yourself, perhaps making other modifications, you'll need your own copy of the VCL source. The following section will be easier to understand if you can open up the


Form1
Delete & Post trap BDE errors
Object Inspector
KDBNavigator1: KDBNavigator
OnBDEError
OnClick
OnDblClick
OnDragDrop
OnDragOver
OnEndDrag
OnEnter
OnExit
OnResize
Properties
Events
New event lets you respond to BDE errors
New read-only property: EngineErrCode
stores the BDE error code after an error occurs. E.g.,
9729 (Key violation)

➤ *Figure 1: KDBNavigator in action*

DBCTRLS unit and refer to it as we go along.

Overview Of The Process

Let me preface this by saying that you could, if you wanted to, go in and modify the `TDBNavigator` sections of the DBCTRLS unit, modifying the functionality of the `TDBNavigator` class itself. All you would really need to do is to declare the `BtnClick` method for `TDBNavigator` as `virtual` and recompile the unit. That should enable you to subclass the component and override the method. I chose to leave Borland's units alone and create my own component unit containing only those portions of DBCTRLS that pertain to the `TDBNavigator`. Here's what I did.

In a new project, after closing the default new form without saving it, create a new unit (`File | New Unit`). Then open the DBCTRLS.PAS file in the IDE (`File | Open File`). This is normally located in the directory \DELPHI\SOURCE\VCL in installations having the VCL source (tip: open this file in a second edit window `View | New Edit Window`).

Next, copy the DBCTRLS `Interface` section's `Uses` clause to the `Interface` section of your new unit. Copy the `Interface` section's type declarations for class `TDBNavigator` to the `Interface` of your new unit. You can locate this code quickly by starting at the top of the unit file and searching for the first `{ TDBNavigator }` comment string. Note that you need to begin copying with the `type` reserved word that appears nine lines or so *before* this first instance of the comment string.

Now copy the declarations for `TNavButton` and `TDataLink` that follow the `TDBNavigator` declarations. You are now ready for the `Implementation` section of the unit.

Copy the `Uses` clause of the DBCTRLS `Implementation` section to the `Implementation` section of your unit. Also copy the `$R` compiler directive (see the sidebar at the end for information on enhancing the appearance of your new component).

Now search for the next occurrence of the `{ TDBNavigator }` comment string. You should see a `const` reserved word right after this comment. Copy the `const` declarations and all of the `TDBNavigator` type procedures, followed by the `TNavButton` procedures and the `TNavDataLink` procedures to your new unit. This brings you to the end of the DBCTRLS unit, which you can now close.

Save your new unit giving it an appropriate name. I called mine KDBNAV.PAS. We now have a unit that, if compiled with the necessary resources, is identical to

TDBNavigator. So identical in fact that we must make a few changes to avoid having the class names bump into each other. For the purposes of illustration, I'll call the unit we just created KDBNAV. Return now to the KDBNAV unit and make the following changes.

In the Interface section, you need to rename all the class declarations to avoid conflicts with TDBNavigator. I simply changed the leading T to the letter K (to give my employers, Kallista, a plug!). The only exception is that I renamed the TDBNavigator class declaration to TKDBNavigator (mainly because I'm used to working with components named T-something). For each type that you rename, you'll want to do a search and replace to change every occurrence of the old name in the unit.

Add the following procedure at the end of the unit:

```
procedure Register;
begin
  RegisterComponents(
    'DataControls',
    [TKDBNavigator]);
end;
```

You can substitute the name of any of the existing palette pages for 'DataControls', or the name of a new palette page that you want to create. Add the declaration for this procedure right before the Implementation reserved word of the unit.

Once you've completed these steps, you have a component TKDBNavigator that should compile and install to the palette (assuming you have the necessary resource files available in the component's directory). However, at the moment, it's identical in functionality to the TDBNavigator component. We now want to modify the component so that it can help us handle errors from the BDE a bit more gracefully.

Modifying The TKDBNavigator

We will add a new event OnBDEError that will occur whenever the EDBEngineError exception is raised in conjunction with the use of the TKDBNavigator. We'll also add a new read-only property EngineErrCode which will store the error code returned by the BDE. The following procedures assume that you renamed all your types substituting K for the leading T. Here are the steps for these changes (I'll summarize them after the last step).

In the first type declaration of the Interface section, add the following line before the class declaration for TKDBNavigator:

```
KEngineErrEvent =
  procedure(Sender: TObject)
  of object;
```

Add two new fields in the private section of TKDBNavigator's record:

```
FEngineErrCode: Word;
FOnBDEError: KEngineErrEvent;
```

The first will store a BDE error code and the second holds event handling code for the new OnBDEError event we are implementing.

Also in the private section, add declarations for a new function and procedure to read and write the new field:

```
function GetErrCode: Word;
procedure SetErrCode(
  const InVal: Word);
```

Declare the new property in the public section of TKDBNavigator's record:

```
property EngineErrCode: Word
  read FEngineErrCode
  write FEngineErrCode;
```

Finally, in the published section, declare the new event as a published property:

```
property OnBDEError:
  KEngineErrEvent
  read FOnBDEError
  write FOnBDEError;
```

For more information on the ins

```
unit Kdbnav;
uses
  SysUtils, WinTypes, WinProcs, Messages, Classes, Controls, Forms,
  Graphics, Menus, StdCtrls, ExtCtrls, DB, DBTables, Mask, Buttons;
const
  { constants from DBCTRLS unit }
type
  { declarations from DBCTRLS unit, plus... }
  KEngineErrEvent = procedure(Sender: TObject) of object;
  TKDBNavigator = { as declared in DBCTRLS unit, note new name for class }
  private
    { field declarations from DBCTRLS unit, plus... }
    FEngineErrCode: Word;  { added field- store BDE error code # }
    FOnBDEError: KEngineErrEvent; { added field- event for BDEErrors }
    { procedure and function declarations form DBCTRLS, plus... }
    function  GetErrCode: Word; { Gets BDE Error code no. }
    procedure SetErrCode(const InVal: Word); { Set BDE Error code no. }
  protected
    { declarations from DBCTRLS unit, nothing added }
  public
    { declarations from DBCTRLS unit, plus... }
    property  EngineErrCode: Word read FEngineErrCode write FEngineErrCode;
  published
    { declarations from DBCTRLS unit, plus... }
    property OnBDEError: KEngineErrEvent read FOnBDEError write FOnBDEError;
end;
```

➤ *Listing 1*

```
{ Added procedures }
function TKDBNavigator.GetErrCode:
 word;
begin
  Result := FEngineErrCode;
end;
procedure TKDBNavigator.SetErrCode(const InVal: Word);
begin
  FEngineErrCode := InVal;
end;
```

➤ *Listing 2*

and outs of private, protected, published etc see the *Component Writers Guide* and Bob Swart's article *Private Investigations* in the February 1996 issue.

So, leaving out all the other original Borland code, which they prefer that you pay for, the interface section for your new component unit looks something like Listing 1.

Now we need to add a bit of code to the `implementation` section of the unit to read and write the new `FEngineErrCode` field we added to the component class record. Add the code in Listing 2 near the end of the file, right before the `Register` procedure.

The Final Frontier

If you've managed to hang in there with me through all of the foregoing steps, I congratulate you. I know it's a bit tedious. But we're now ready to get to the heart of the matter.

If you'll take a moment to study the `BtnClick` method in the `TDBNavigator` section of DBCTRLS, you will see the `Case` construct that issues those calls to the engine. Notice that the statements are unconditional. They send off that call no matter what, no error checking takes place. And that's what we want to change.

Only two of these `Case` statements really concern us, those that call `Post` and `Delete`, because it is on those actions that an error is most likely to occur and it is over those actions that we want to exert our own control when the `KDBNavigator` is being used to update a data set. We'll modify the `Post` and `Delete` cases to place the engine call in a protected `try..except` block (you could add protected blocks to any or all of the other `Case` statements if you see a need) The protected block will intercept the `EDBEngineError` exception, at the same time creating an instance of the exception object. From the properties of that object we can get the BDE error code number (we could also get the message string if we wanted it). We can then trigger our `OnBDEError` event, writing the engine error code to the field `FEngineErrorCode` which is subsequently accessible through our new public property `EngineErrCode`. If nothing is assigned to our new field `FOnBDEError` (ie there is no event handler code for the new `OnBDEError` event), then and only then do we allow the normal exception to be raised. Otherwise, we handle the error with code in `TKDBNavigator's OnBDEErrorEvent` handler. Listing 3 (next page) shows how to modify these two case statements.

All that remains to create the new `KDBNavigator` component is to compile it with the appropriate resource files (.RES and .DCR) present and install it to the component palette.

I'll be the first to admit that the solution I've presented here is little more than a quick-n-dirty hack to get something that works into a real project. In going to all the trouble of re-creating this new unit, the "right" thing to do would probably have been to re-write the component in such a way that it would be easier to derive descendants from it (declaring the `BtnClick` method as `virtual`, for example). Oh well, maybe next time around... when I have more time!

Now that you understand what goes on behind the scenes, I hope you'll latch onto the compiled KDBNAV.DCU and use it if the need arises.

Robert Palomo works as a Delphi developer for Kallista, Inc and can be contacted by email as 76201,3177 on CompuServe

```
case Index of
  ... {as in DBCTRLS TDBNavigator.BtnClick}
  ...
  nbPost:
    begin
      Try
        Post;
      Except on PErrObj: EDBEngineError do begin
        { Set BDE Error code no. into property value }
        SetErrCode(PErrObj.Errors[0].ErrorCode);
        If Assigned(FOnBDEError) then
          { If this error occurs & there is a user event handler, then
            execute it, otherwise, raise the exception }
          FOnBDEError(Self)
        Else
          Raise;
      end else
        Raise;
      end;
    end;
  nbDelete:
    begin
      if not FConfirmDelete or (MessageDlg(LoadStr(SDeleteRecordQuestion),
        mtConfirmation, mbOKCancel, 0) <> idCancel) then begin
        Try
          Delete;
        Except on DErrObj: EDBEngineError do begin
          { Set BDE Error code no. into property value }
          SetErrCode(DErrObj.Errors[0].ErrorCode);
          If Assigned(FOnBDEError) then
            FOnBDEError(Self)
            { If this error occurs & there is a user event handler then
              execute it, otherwise, raise the exception }
          else
            Raise;
        end else
          Raise;
        end;
      end;
    end;
  ...  { as in DBCTRLS TDBNavigator.BtnClick }
  ...
end; { case }
```

➤ *Listing 3*

Spice Up Your Navigator!

If you're developing in-house applications for your company, you can probably live with the "stock" graphics of the `TDBNavigator`, boring though they may perhaps be. If, however, you are developing apps of a more commercial grade, you'd probably like to put a little more life in the `TDBNavigator`.

Changing the graphics that display on this component is a snap. All you have to do is replace the bitmaps in the DBCTRLS.RES file (normally found in \DELPHI\LIB). If you're handy with the Image Editor, you can open up this file (make a backup first!) and draw some spiffy new images to replace the originals. For the less artistically inclined, you'll find several replacements for this resource file in the Delphi forum library on CompuServe (mine is included with the `KDBNavigator` .DCU).

To use one of these third party replacements, follow these steps.

➢ Preserve the original resource file by renaming it DBCTRLRC.OLD or some such name.

➢ Copy the third-party resource file to the \DELPHI\LIB directory, naming it DBCTRLS.RES.

➢ Recompile your COMPLIB.DCL (`Options | Rebuild Library`). You do have a recent back up of your COMPLIB.DCL, don't you?

Now whenever you drop a navigator into a form, you'll get the zippy new look.