# *Under Construction:* Testing And Debugging

*by Bob Swart*

After over a year of intensive Delphi component and expert building in this column, this month we're going to focus our attention on component testing and debugging. Most of the techniques I discuss will work with both Delphi 1 and Delphi 2.

Making sure your components *really* work is very important, especially if you (and possibly your colleagues) plan to re-use them over and over again. You can fix bugs during design, implementation, beta testing and finally when the system is delivered and the client discovers something is wrong. Remember that the cost of fixing a bug increases (almost exponentially) as time goes by. It's fairly cheap to fix a logical design bug while you're still in the design phase, but it'll cost you a whole heap more when the bug is discovered by your client, two weeks after the system has been installed...

## Debugging By Design

You can help prevent bugs by making a good solid design beforehand. For big projects you may even use an OO design methodology, like the Object Modeling Technique of James Rumbaugh et al, described in the book *Object-Oriented Modeling and Design*, Prentice-Hall, ISBN 0-13-630054-5. This book will teach you how to write a solid design using a proven technique.

But what about components? Often, a component builder is just a component hacker, putting some handy code into a re-usable class. The Delphi compiler is fast enough to offer you an iterative component building path and so you decide to worry about the design and documentation later. However, I believe that for all but the simplest of components it's still best to write a design, since it could save you a lot of time later on.

## Debugging Comments

Once you start coding, there's one thing you must do: use comments! In a few months time, you (or someone else) will take a look at the same sections of code again and wonder what you were thinking when you wrote them. Comments are especially important if you don't have the design documented the way you should have, for example when you're using Delphi as a RAD prototyping tool.

If you are writing code that may be used with both Delphi 1 and 2, then watch out for `//` comments, which are valid only in Delphi 2.

One final remark about comments: keep them up-to-date. There's nothing worse than a piece of code and accompanying comment which contradict each other. As Murphy used to say: if the code and comment disagree, then both are probably wrong.

## Debugging Names

Naming conventions come in all shapes and sizes. A good book that includes information on naming conventions (and lots of other useful coding guidelines) is *Code Complete* by Steve McConnell (MS Press, ISBN 1-55615-484-4). This is an excellent book with many gems on software construction. If you are a professional software engineer, then you should read it. If you're working in a team, make sure the entire team reads it!

You should use prefixes so that like components are grouped together in the Object Inspector. It makes it much easier to locate a particular component. It doesn't matter exactly what prefixes you use, as long as everyone can easily identify them, and you are consistent (at least within a single project).

Tables and queries always get their full name (eg `TableBIOLIFE` for the table connected to BIOLIFE.DB) and the same happens to the data-aware controls that connect to the fields: prefixed with the native (non-data-aware) field, but postfixed with the database or table name and the name of the field, eg `MemoTableBIOLIFENotes`.

Even within a component we should use naming conventions, such as an `F` prefix for a hidden field that holds the property value, and `Get` and `Set` prefixes for property methods.

## Early Component Pitfalls

Almost every component writer must have made the mistake shown in Listing 1 at least once (or is a good straight-faced liar).
Now why doesn't this code work correctly? In fact, it doesn't seem to work at all, especially if you derive from components higher in the hierarchy (like a `TEdit` or `TTable`) and want to add your own behaviour. We've forgotten to tell the compiler that we wanted to override the behaviour of the `Create` constructor. Since we didn't include the `override` keyword, the compiler is happily creating a new constructor `Create`, making the previous one invisible in our scope and hence not callable! We'd end up never initialising the component we are descending from: a serious mistake.

In my view, this should at least merit a warning from the compiler (we have warnings now, why not use them for this as well, please, Borland?). Speaking of warnings, if you try to catch bugs and problems early on, then you should always have warnings as well as hints enabled in your compiler options. I always make sure to have eliminated most of them, or at least write comments to explain why the hint or warning is still present at that point.

## Error 202 in COMPLIB

The Get and Set methods that are used to get and set values of properties can be used to implement pre- and post-conditions as well as regular value checking. They can also contain unintentional errors.

Consider class TBuggy in Listing 2, a component derived from TComponent, with one property Data, which reads the value from an internal field FData and writes it back using a SetData method. The SetData method can now be used to check for a valid value of Data.

If you add TBuggy to the component palette of Delphi, then at first sight nothing is wrong – until you set the value of the Data property to a value other than 0. In Delphi 1 it happens real fast, Delphi 2 lets you wait a little bit longer (actually 16 bits longer), but they both give a stack overflow error.

Since Delphi 2 has a much larger (potential) stack than Delphi 1, it takes more time for Delphi 2 to overflow the stack. Delphi 1 has a stack limited to 64Kb minus the data segment size and the local heap size. The default Delphi 1 stack size is 8Kb, although I often increase this to 32Kb. You can get an overview of these figures after you've compiled your project from the Information dialog. A stack overflow in Delphi 1 can be caused by string processing, too many local variables on the stack (in a local routine) or infinite recursion.

Delphi 2 can have a much bigger stack: the minimum is set to 16Kb, the maximum to 1Mb. Hence, a stack overflow in Delphi 2 is most often caused by infinite recursion, since the stack is generally too big to have any problems with local variables or short strings on the stack and Delphi 2 long strings are allocated on the heap.

So, assuming that recursion is the problem in this case, where did we use recursion? Actually, this is the danger of using properties, hidden fields and property methods. You name the field with an F prefix, name the methods with Get and Set prefixes, but you still think of it as the regular property name (Data, not FData or SetData).

This is where the mistake happens, as SetData checks the Data property and updates it if the new value is not already equal to the existing one. But in that case, SetData sets the value of the Data property, which in turns causes a call to... SetData. This is a typical case of hidden property method recursion, which can be really dangerous!

➤ *Listing 1*

```
Type
  TBug = class(TComponent)
    constructor Create(
      AOwner: TComponent);
  end;
implementation
constructor TBug.Create(
  AOwner: TComponent);
begin
  inherited Create(AOwner);
  ...
end;
```

➤ *Listing 2*

```
unit Buggy;
interface
uses Classes;
Type
  TBuggy = class(TComponent)
  private
    FData: Integer;
  protected
    procedure SetData(NewData: Integer);
  published
    property Data: Integer read FData write SetData;
  end {TBuggy};
  procedure Register;
implementation
procedure TBuggy.SetData(NewData: Integer);
begin
  if NewData <> Data then { only if changed }
    Data := NewData
end {SetData};
procedure Register;
begin
  RegisterComponents('Dr.Bob',[TBuggy])
end;
end.
```

## Recursion In Delphi 2

Consider the following example of TBoom (Listing 3). I've fixed the SetData method (this is what it should look like: accessing the FData field that holds the value, not the Data property itself), but I've also introduced another method GetData to obtain the value of the property. For the purpose of showing you the dangers that can happen, I've made the mistake of looking at the Data property, which causes another call to GetData to obtain the value for the Data property, which calls GetData again and again, until we get a stack error exception again, right? Wrong!

This time, when we register the TBoom component within Delphi something even worse happens. If we drop the component on a form the Object Inspector will try to display the property values. So, it will

➤ *Listing 3*

```
unit Boom;
interface
uses Classes;
Type
  TBoom = class(TComponent)
  private
    FData: Integer;
  protected
    function  GetData: Integer;
    procedure SetData(NewData: Integer);
  published
    property Data: Integer read GetData write SetData;
  end {TBoom};
  procedure Register;
implementation
procedure TBoom.SetData(NewData: Integer);
begin
  if NewData <> FData then FData := NewData
end {SetData};
function TBoom.GetData: Integer;
begin
  GetData := Data { boom! }
end {GetData};
procedure Register;
begin
  RegisterComponents('Dr.Bob',[TBoom])
end;
end.
```

call the GetData method of our TBoom component, again and again, until we get the message shown in Figure 1. The bad thing is if we click Close then Delphi will actually close, taking all our unsaved changes with it! Which leads to one very important rule for component builders: inside the component class, never use the property name itself, but always refer to the internal field that holds the value.

## IDE Debugger

Let's assume we've made all efforts to prevent bugs during the design and implementation. Murphy will make sure there's always one more bug, which means it's time to switch over to the debuggers!

Most of us are using exceptions to split the algorithm (and error detection) from the error handling and recovery. However, when using exceptions together with the internal IDE debugger, you must be aware of the Break on Exceptions option in the Environment Options | Preferences page. It's on by default, but I find it just causes confusion.

## External Debugger

We can also use the standalone Turbo Debugger for Windows (TDW). Why would we need TDW if we can already debug from the IDE itself? Well, you need it to debug DLLs, or view mixed source listings (eg a Delphi DLL and a C++ program). TDW is included in the RAD Pack (for Delphi 1) and Turbo Assembler 5.0 (for Delphi 2).

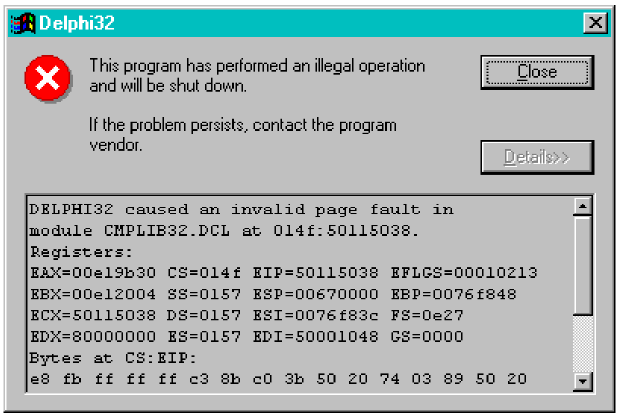
## Creative Debugging

No matter how good the debugging tool you use, the best debugger is always the one between your ears. If that debugger doesn't have a clue where to search for a bug, it'll be a long night... A good book that shows many useful techniques is *Debugging – Creative Techniques and Tools for Software Repair* by Martin Stitt (Wiley, ISBN 0-471-55831-1). It was first published in 1992 but the techniques are still valid and enjoyable to read about.

## MessageDlg Debugging

I often use message dialogs for debugging. For example, in many

➤ *Figure 1*



This program has performed an illegal operation and will be shut down.

If the problem persists, contact the program vendor.

```
DELPHI32 caused an invalid page fault in
module CMPLIB32.DCL at 014f:50115038.
Registers:
EAX=00e19b30 CS=014f EIP=50115038 EFLGS=00010213
EBX=00e12004 SS=0157 ESP=00670000 EBP=0076f848
ECX=50115038 DS=0157 ESI=0076f83c FS=0e27
EDX=80000000 ES=0157 EDI=50001048 GS=0000
Bytes at CS:EIP:
e8 fb ff ff ff c3 8b c0 3b 50 20 74 03 89 50 20
```

```
{$IFDEF DEBUG}
  if MessageDlg(Format('Post record %s %s %s ?',[f1,f2,f3]),
                mtConfirmation, [mbYes, mbNo], 0) = mbNo then
    Exception.Create('Don''t post this record!')
{$ENDIF}
```

➤ *Listing 4*

event handlers, I insert code with an {$IFDEF} that shows or doesn't show a MessageDlg with some status information, like the pseudo-code shown in Listing 4, from an OnBeforePost event.

The code shown uses Format to show me some of the key fields of the record to be posted and if I don't click the Yes button an exception will be generated which, in this case, will prevent the record from being posted. Of course, the exception part is optional, the important thing is to be able to check some critical values while the program is running. For production code, just un-define DEBUG.

## Log Files

MessageDlg debugging is nice, but may get you into trouble when debugging anything to do with mouse clicks. Since the mouse down will often initiate a click, the MessageDlg will come up and automatically eat your mouse up! If you then close the dialog, you won't have received the mouse up message: the button will remain down and you will need to do another mouse down to get yet another mouse up.

In order to prevent that, I often write mouse or keyboard input events to a log file instead and check the contents of the file afterwards. Of course, this means I can't check the log file until the program's completed.

## WinCrt And CONSOLE

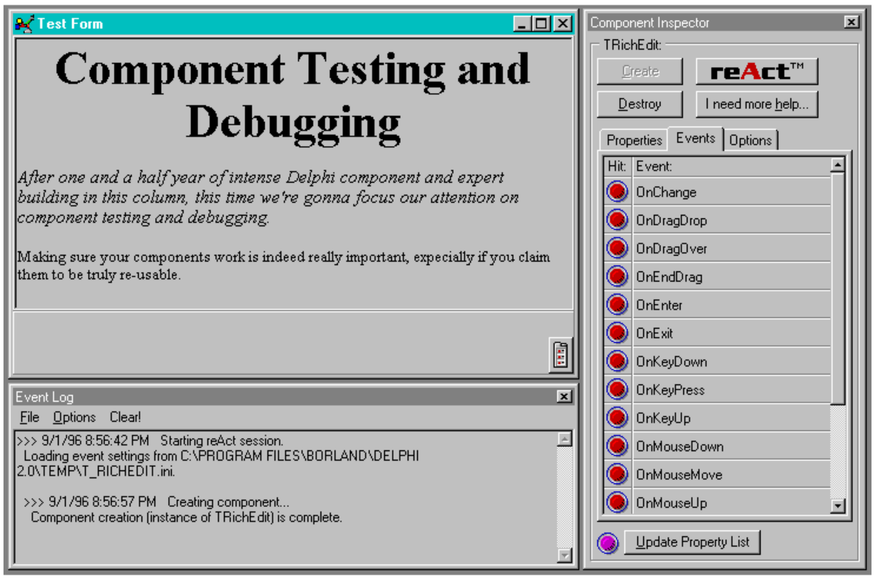
Did you know that even if you have a form-based application you can still use WinCrt (for Delphi 1) or make it a CONSOLE application (for Delphi 2)? I didn't, until I found out by accident (I used a unit that had a uses WinCrt inside and did some WriteLns). You can actually have a floating WinCrt window next to your application and use WriteLn inside your application to put stuff into this window. This overcomes the limitations of a log file.

For Delphi 2 just define {$APPTYPE CONSOLE} at the top of your project file and you're ready to go. For Delphi 1 make sure every unit that contains WriteLn statements has WinCrt in the uses clause.

## Design Versus Run Time

Remember that a component can have two behaviours: one at design time (the code in the Component Library) and one at run time (the .DCU file on disk). They can actually differ because one version may be older than the other (you may have updated the component source and re-compiled it to a newer .DCU file, while the loaded Component Library still contains the older version).

You can distinguish between design time and run time in your components by checking the flag cdDesigning in the ComponentState: see Listing 5.



Component Testing and Debugging

*After one and a half year of intense Delphi component and expert building in this column, this time we're gonna focus our attention on component testing and debugging.*

Making sure your components work is indeed really important, expecially if you claim them to be truly re-usable.

```
>>> 9/1/96 8:56:42 PM   Starting reAct session.
  Loading event settings from C:\PROGRAM FILES\BORLAND\DELPHI
2.0\TEMP\T_RICHEDIT.ini.

  >>> 9/1/96 8:56:57 PM   Creating component...
   Component creation (instance of TRichEdit) is complete.
```

➤ *Figure 2*

```
If csDesigning in ComponentState then begin
  writeln('CompLib');
  MessageDlg('You are in Design Mode', mtInformation, [mbOk], 0)
end else begin
  writeln('EXE');
  MessageDlg('You are in .EXE Mode', mtInformation, [mbOk], 0)
end;
```

➤ *Listing 5*

## reAct

There's a new third party tool available for Delphi that will help you test and debug your components. It's called reAct and is from Eagle Software, who brought us the CDK. reAct enables us to test a single component, but it can interact with other components. For each case, a new special test project is set up by reAct. Testing the component involves merely compiling and running the generated test project, which not only shows the test form, but also a Component Inspector and Event Logfile.

One of the strong points of reAct is that it lets us follow the chain of events, by showing them as lights that flash on and off when the events are triggered (Figure 2). Apart from that, you can set any property within the reAct Component Inspector (the reAct version of the Object Inspector), follow any event and still work with your component(s) as if you're in a real application. The log file can be viewed while testing and of course saved and printed afterwards. The generated source code for the test project is available to expand (and you'll see the conditional `INT $3` statements which cause debugger breakpoints that you can set automatically from within your reAct test program).

Every now and then (when you're doing new or complex things) you'll get a message from Mr CDK who offers helpful advice.

While this is not intended to be a product review (I've barely scratched the surface here), I didn't want to leave reAct out of the list of books, tools and techniques that are helpful for testing and debugging our components.

Currently, reAct is only for Delphi 2, but a 16-bit version is in the works as I write this in early September. For more information or some neat demos, visit http://eagle-software.com.

## Delphi 2 And NT 4.0 Beta 2

A final problem to end with: it has become clear that Windows NT 4.0 beta 2 has a problem with unloading (cached) DLLs that do not have the .DLL extension (like the component library, CMPLIB32.DCL). This unfortunately means that after a rebuild of the Delphi 2 component library (like after installing a new component), the library is not unloaded correctly and cannot be re-loaded. This causes an error 998 in Delphi.

The workaround is to bring NT down, start it up again and reload Delphi. At that time, you will see your new component library with the newly installed components.

You actually need to do this every time when your CMPLIB32.DCL has been unloaded (so maybe also if you stop Delphi and start it again). It's not nice, but it's a bug in NT, and Microsoft is said to have fixed it for the final release, which should hopefully be out by now, so this is one more reason to go out and get it instead of using the betas!

Another solution, proposed by Mike Scott, is to rename CMPLIB32.DCL to CMPLIB32.DLL (since it is in fact a DLL), which will tell NT that it should be unloaded properly. You have to open the .DLL version with the `Open Library` dialog, but after that all new components will be added to this 'DLL' version of CMPLIB32).

## Conclusions

This month, we've been through some ways to help debugging and to prevent debugging of your components. The techniques, books and tools mentioned can support you in your hunt for bugs, but in the end it all comes down to the biggest debugger of all: the human mind. And speaking of the human mind and intelligence, next time we'll focus on how to build some Artificial Intelligence components, an area my company Bolesian has been in for almost 15 years now.

Bob Swart (aka Dr.Bob, email 100434.2072@compuserve.com) is a professional software developer using Borland C++ and Delphi for Bolesian and a freelance technical writer. In his spare time he likes to watch video tapes of Star Trek Voyager and Deep Space Nine with his 2.5 year old son Erik Mark Pascal.