Delphi 3 Add-In Packages: Digging The Dirt On Archaeopteryx

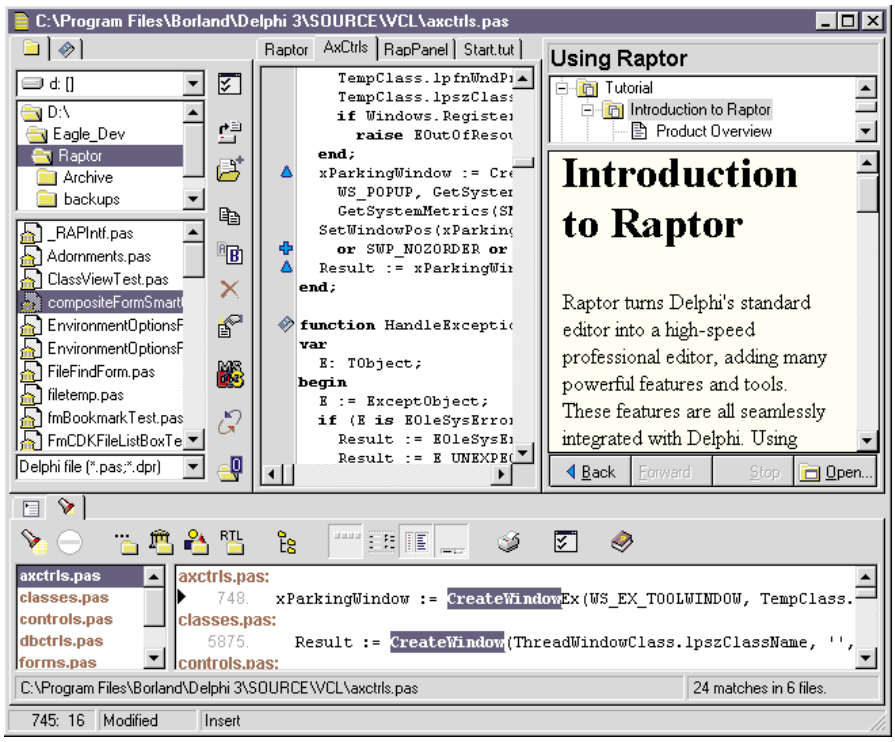
by Brian Long

Archaeopteryx is a Delphi 3 package I wrote some while back after seeing a beta stage add-in package designed to enhance Delphi's IDE called T-Rex. T-Rex was written by Mike Scott and John Howe and did such things as add entries onto the component palette's right-click menu that represent each page on the component palette. When a menu item is chosen, the corresponding page is selected. It also added the ability to turn on the component palette tab control's `MultiLine` property, so all pages could be viewed without scrolling.

It would appear that T-Rex was created after the authors saw another Delphi 3 IDE enhancer called (whilst in beta) Raptor, by Eagle Software (Figure 1). This product goes to town and significantly enhances the Delphi Editor, Object Inspector, various dialogs and much more besides. T-Rex, of course, is the name of a famous dinosaur. Raptor is short for Veloceraptor, a dinosaur made well known by the film *Jurassic Park*.

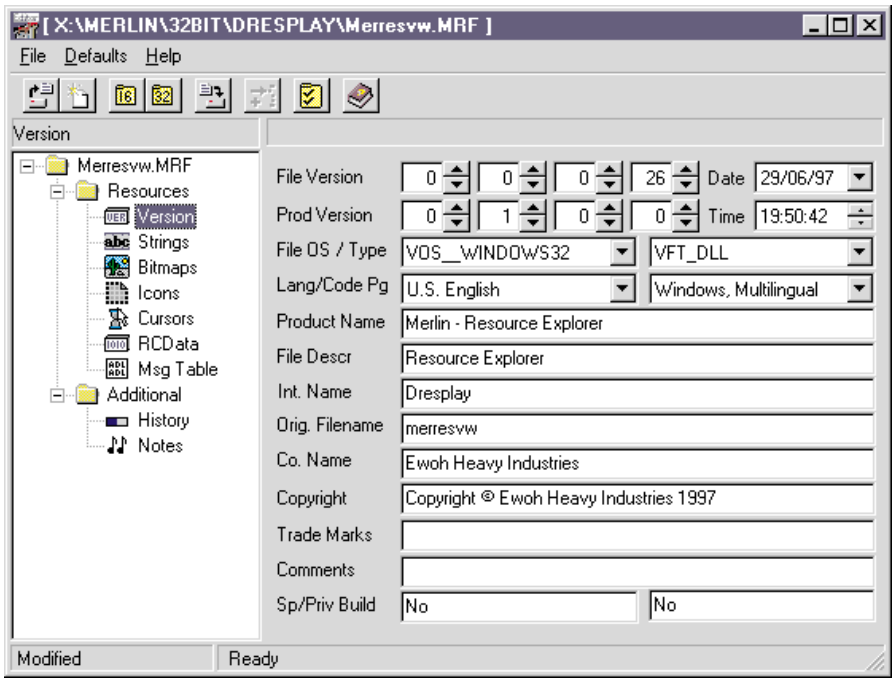
Having seen these packages, I wondered how they achieved such cunning internal IDE devilry and put some thought into the matter. When I worked it out, I felt obliged to see if I could do it too. The result was Archaeopteryx (another dinosaur: the first feathered bird type thing). I chose one with a complicated name to try and ensure people couldn't spell it and so would find these other add-in products on the Web first. After all, T-Rex (whose functionality I, errrm, based my package's on) was designed to show what would be in a planned product called Merlin (Figure 2), and Raptor did significantly more than my humble creature does.

However, thanks to Bob Swart for putting my bird on his website (www.drbob42.com), it seemed to prove quite popular. So, this article professes to show you how to do the same sort of customisation yourself.



➤ *Above: Figure 1, Raptor*

➤ *Below: Figure 2, Merlin*



About Archaeopteryx...

The latest Archaeopteryx does the following to Delphi 3. It adds menu items to Delphi's component palette popup menu for each palette

page, adds a Window menu (with customisable caption to avoid clashes), allows you to turn the flat speed buttons back to non-flat buttons, turns the component palette into a multiline tab control, turns the palette pages into buttons (like Windows task bar, to avoid multiline tab shuffling), turns on component palette hot tracking (which means the tab text changes colour when the mouse is over it) and un-hides several quite useful Delphi main menu items which have been left hidden by Borland.

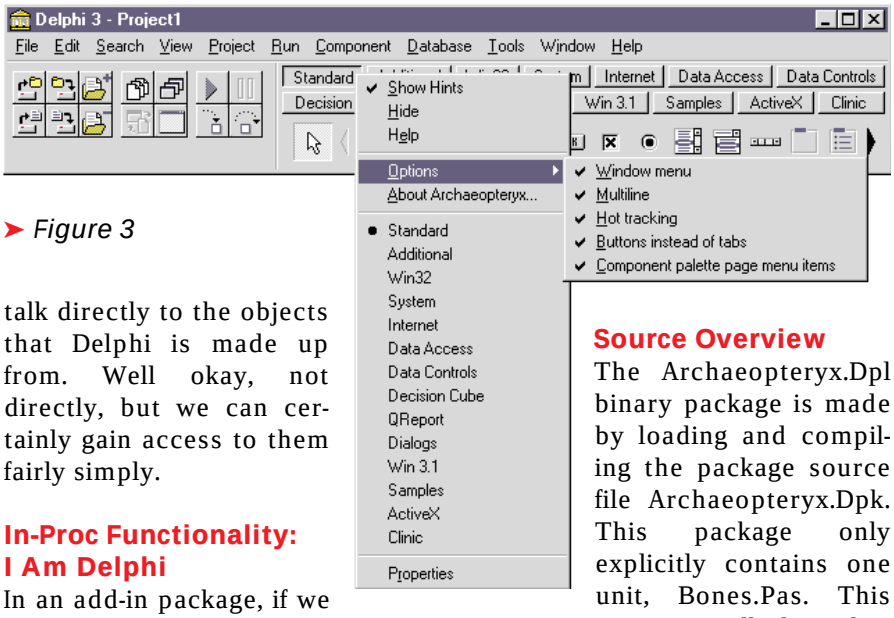
All but the last facility can be toggled on and off. Figure 3 shows Archaeopteryx in full flight.

Implications Of Packages

To understand how to implement this functionality requires really only one fundamental point of understanding about packages (most things added into the IDE go in as packages). What you need to remember is that in your applications, whether you compile your units into packages or not, you can access the symbols therein just the same. In other words, despite packages being implemented as DLLs, Delphi generates code to access the symbols from the relevant files transparently. Use of packages should be thought of as just a linker option and nothing more. So what we see is that the code in a package behaves just as if it is part of the main application's address space. Semantically a package's code is part of the main application.

So if we write a package and add it into Delphi, we are effectively extending the Delphi EXE file functionality. This rather differs from Delphi 1 and 2, where the additional files get added into a component library, a regular DLL, and so become explicitly part of a different module. Semantically a DLL's code is completely separate from the main application's.

So what does this imply for the Delphi add-in writer? Well it means that we can write the sort of thing that traditionally resides in experts, with much less effort. Because our package code is inherently part of Delphi itself we can



➤ *Figure 3*

talk directly to the objects that Delphi is made up from. Well okay, not directly, but we can certainly gain access to them fairly simply.

In-Proc Functionality: I Am Delphi

In an add-in package, if we use the `Forms` unit and refer to the `Application` object, we are talking to Delphi's Application object, the object that owns most of the visible forms and has a `MainForm` property that refers to the Delphi main window. That form has a `MainMenu` property which points to Delphi's menu bar. Are you starting to get the idea?

If we know the names of objects in the Delphi IDE we can use `FindComponent` to gain access to them (see Listing 1, an extract from CommonStuff.Pas). A modified version of the Resource Explorer (see this month's *Delphi Clinic*) proves invaluable here. Of course one of the other side-effects of writing these add-ins as packages is that the binary files are very small indeed. Archaeopteryx takes all of 30kb.

Source Overview

The Archaeopteryx.Dpl binary package is made by loading and compiling the package source file Archaeopteryx.Dpk. This package only explicitly contains one unit, Bones.Pas. This unit uses all the other relevant units that implement all the functionality. There is one extra unit for each piece of new functionality available (these units are implicitly contained in the package). For some reason, when I wrote Bones.Pas I thought I might want to make different versions of Archaeopteryx that contained different amounts of the available functionality and so it has several `$defines` that cause the units to be pulled in (see Listing 2). There is nothing else in this main unit.

Since I need event handlers here, there and everywhere and since event handlers are typically implemented as methods, each unit defines a class to house these event handling methods. The `initialization` section of the unit creates an instance of the class and

➤ *Listing 1*

```
function GetComponent(Owner: TComponent; const Name, Error: String): TComponent;
begin
  Result := Owner.FindComponent(Name);
  if not Assigned(Result) then
    raise Exception.Create(Error);
end;
```

➤ *Listing 2*

```
{$define WindowMenu}
{$define PalettePopup}
{$define PaletteTweaking}
{$define HiddenMenus}
{$define NonFlatButtons}
uses
  {$ifdef HiddenMenus}HiddenMenus,{$endif}
  {$ifdef NonFlatButtons}NonFlatButtons,{$endif}
  {$ifdef PalettePopup}PalettePopup,{$endif}
  {$ifdef PaletteTweaking}PaletteTweaking,{$endif}
  {$ifdef WindowMenu}WindowMenu,{$endif}
  Messages;
```

the finalization section frees the instance. In this way each unit is completely self-contained. The classes are defined in the implementation section of their respective unit, making them private and ensuring the final binary package doesn't get bloated with automatically exported references to the classes and all their methods.

Common Code

Most of these class units use one other unit called CommonStuff. This defines the GetComponent function as well as another class (instantiated and tidied away in the unit). The TIvoryHacker class is in the interface section, as is the Stuff object reference which gets set to an instance of the class. As Listing 3 shows, Stuff has a number of data fields that end up proving useful in the other classes. There is Ini, an object for talking to the registry (for storing Archaeopteryx state information) and a couple of fields that get mapped onto various IDE components (using GetComponent), a reference to Delphi's component palette and the palette's popup menu.

Also we find the About box procedure, DoAbout, which uses a fairly unused VCL routine called CreateMessageDialog as used internally by ShowMessage and MessageDlg (see the *Delphi Clinic* in Issue 23). This is so the code can alter the normal icon used to be a custom icon before the dialog is displayed (see Listing 4). Xavier Pacheco's article this month has information on extracting version information from your executable. Similar code is used to get the product version and version for display in the About box.

The constructor is quite simple, it creates the TRegIniFile object and then sets up the IDE component fields as in the following statement:

```
const
  SGenericError =
    'Cannot find requested component: ';
FTabControl := GetComponent(
  Application.MainForm,
  'TabControl', SGenericError +
  'TabControl') as TTabControl;
```

The other code section worth looking at in this unit is the AddOptionsItem method which adds the Options menu item on the component palette's popup menu (remember that most options hang off this menu and they are added by different units, each of which was designed to work independently). To manufacture the menu item, rather than calling the appropriate class constructor and setting properties individually, we use an undocumented routine from the Menus unit called NewItem. NewItem makes a TMenuItem component and sets the Caption, ShortCut, Checked, Enabled, HelpContext, Name and OnClick properties with values passed as parameters. This can be a good time-saver. Other undocumented routines from the Menus unit include NewMenu, NewPopupMenu, NewSubMenu and NewLine.

The General Approach

The key units implement classes that do much the same job to a greater or lesser extent. We'll take a brief look through a simple one and then move on to one that has additional requirements. So firstly, the NonFlatButtons unit which adds an option onto the speedbar to allow you to turn the new sleek flat speedbuttons into the old Delphi 2 normal, non-flat buttons. The TNonFlatButtons class is in Listing 6. It has object references that the constructor connects to the speed bar (a TPanel) and the speed bar's popup menu. Another object reference (FSpeedBarFlat) is associated with a new menu item that gets inserted into the popup menu. The DoFlatButtons method becomes it's OnClick event handler, which simply toggles the checkmark property on the menu item and then makes the buttons flat or non-flat as required. That last job is implemented in the SetFlatButtons method.

The implementations of the methods are shown in Listing 7. You can see in the constructor that

➤ *Listing 3*

```
TIvoryHacker = class(TObject)
public
  FTabControl: TTabControl; //Component palette
  FPalettePopup: TPopupMenu; //Palette popup menu
  FOptions: TMenuItem; //Archaeopteryx options menu item
  Ini: TRegIniFile; //Used to save and restore options in registry
  procedure DoAbout(Sender: TObject); //Shows About box
  procedure AddOptionsItem; //Ensures Options item exists
  constructor Create;
  destructor Destroy; override;
end;
var Stuff: TIvoryHacker;
```

➤ *Listing 4*

```
procedure TIvoryHacker.DoAbout(Sender: TObject);
begin
  //Would normally use MessageDlg, but I want to customise
  //the icon, so use the more primitive CreateMessageDialog
  with CreateMessageDialog(SAboutMsg + VersionNumber, mtInformation, [mbOk]) do
    try
      (FindComponent('Image') as TImage).Picture.Icon.Handle :=
        LoadIcon(HInstance, 'Archaeopteryx');
      Caption := 'About Archaeopteryx';
      ShowModal;
    finally
      Free
    end;
end;
```

➤ *Listing 5*

```
procedure TIvoryHacker.AddOptionsItem;
begin
  { If another unit needs to add options items, they call this to add the main
    Options sub-menu just above the last menu item (Properties) }
  if not Assigned(FOptions) then begin
    FOptions := NewItem('&Options', 0, False, True, nil, 0, '');
    FPalettePopup.Items.Add(FOptions);
    FOptions.MenuIndex := FPalettePopup.Items.Count - 1;
  end;
end;
```

as well as locating the relevant IDE components (whose names were found using Resource Explorer) the new menu item is manufactured. Its initial checked state is decided by reading the relevant entry from the registry. If this is the first time Archaeopteryx has run, the registry will not have a value and so the current state of the speed bar's flat-ness is used instead. Once the menu item has been added into the popup menu, the appropriate state of the speed buttons' `Flat` properties is set with a call to `SetFlatButtons`.

So that shows a simple IDE modification. But not all modifications are that simple.

Changes Not Involving Properties

When it comes to customising the component palette tab control, most of it can be done by writing code that sets properties of the Delphi 3 `TTabControl` component. Properties such as `MultiLine` and `HotTrack` deal with some of the functionality, however to turn the tabs into buttons there is no such property (see last month's *Things You Can Do With Standard Controls: TPageControl* article for details).

So we have to set the appropriate window style using the Windows API. But care must be taken, when various other things happen (such as the Archaeopteryx user toggling the multi-line option) the underlying control will be recreated and our new window style will be lost. So the code that deals with setting the hot-tracking and multi-line options must also ensure the tabs-as-buttons option is still respected. Also, when the Multi-line option is toggled the IDE needs to be resized bigger or smaller to accommodate the new number of tab rows. Listing 8 shows the three key methods for these points from the `TPaletteTweaking` class in PaletteTweaking.Pas. Notice `SetMultiLine` makes use of another utility routine `UpdateIDESize`, passing it the number of tab rows that there used to be. It then calculates how much the IDE should be resized by and does it (by sending a `wm_Size` message).

More problems arise when Archaeopteryx is being loaded as Delphi 3 starts. The unit `initialization` sections get executed and create all the customisation objects. In the case of the `TPaletteTweaking` object, the constructor is supposed to restore the saved values for the component palette options. However it fails, because when Archaeopteryx is being loaded, the tab control is in no state to be told to change its appearance.

All the settings are ignored. To avoid this I have had to resort to a rather unpleasant workaround of using a timer that keeps ticking every half second until it is safe to talk to the tab control. If, when Delphi 3 is loading, some error occurs causing a message box to be displayed (like a package not being found) the timer spots this and keeps on ticking. When it sets the options, it disables itself. The timer is created and set up in the main object's constructor and freed in the destructor. Listing 9 shows the `OnTimer` handler.

Event Chaining: A Necessary Sin?

Most of the rest of the problems come from needing react to some of Delphi's events. This poses a bit of a problem. The business of event chaining is *reasonably* commonplace amongst Delphi hackers. For examples of it you could refer to the *Tips & Tricks* section in Issue 7 (Event Chains) or the *Delphi Clinic* in Issue 6 (Combo Woes). Event chaining involves saving the current event handler for an event, assigning a new handler to the event, and in the new handler calling the old one. This allows you to add functionality to an existing event handler.

➤ *Listing 6*

```
TNonFlatButtons = class(TObject)
private
  FSpeedBarPopup: TPopupMenu;
  FSpeedBar: TPanel;
  FSpeedBarFlat: TMenuItem;
protected
  procedure SetFlatButtons(Value: Boolean);
  procedure DoFlatButtons(Sender: TObject);
public
  constructor Create;
  destructor Destroy; override;
end;
```

➤ *Listing 7*

```
constructor TNonFlatButtons.Create;
begin
  inherited Create;
  //Find speed bar
  FSpeedBar := GetComponent(Application.MainForm,
    'SpeedPanel', SGenericError + 'SpeedPanel') as TPanel;
  //Find speed bar's popup menu
  FSpeedBarPopup := GetComponent(Application.MainForm,
   'SpeedbarMenu', SGenericError + 'SpeedbarMenu')
   as TPopupMenu;
  //Set up option menu item
  FSpeedBarFlat := NewItem(SFlat, 0,
    Stuff.Ini.ReadBool('PaletteLocalPopup', 'Flat Speedbar',
    (FSpeedBar.Controls[0] as TSpeedButton).Flat), True,
    DoFlatButtons, 0, '');
  FSpeedBarPopup.Items.Insert(1, FSpeedBarFlat);
  //Set the speedbar properties as dictated by registry
  SetFlatButtons(FSpeedBarFlat.Checked);
end;
destructor TNonFlatButtons.Destroy;
begin
  //Get rid of option menu item
  FSpeedBarFlat.Free;
  //Restore IDE original state
  SetFlatButtons(True);
  //Save option information
  Stuff.Ini.WriteBool('PaletteLocalPopup', 'Flat Speedbar',
    FSpeedBarFlat.Checked);
  inherited Destroy
end;
procedure TNonFlatButtons.SetFlatButtons(Value: Boolean);
var Loop: Integer;
begin
  //Set Flat option as appropriate
  with FSpeedBar do
    for Loop := 0 to ControlCount - 1 do
      if Controls[Loop] is TSpeedButton then
        TSpeedButton(Controls[Loop]).Flat := Value
end;
procedure TNonFlatButtons.DoFlatButtons(Sender: TObject);
begin
  //Toggle Flat option
  with (Sender as TMenuItem) do begin
    Checked := not Checked;
    SetFlatButtons(Checked)
  end
end;
```

This is fair enough in a self-contained program, but when writing add-ins for another application can be fraught with problems. Consider this scenario. You install T-Rex into Delphi 3 and T-Rex chains onto a certain event it needs. The value of the event property in Delphi now refers to T-Rex code, and T-Rex calls the original routine. Now you load Archaeopteryx which happens to chain the same event (as a lot of the functionality of the two packages is the same).

The original event property now refers to Archaeopteryx code, and Archaeopteryx calls whatever event hander was there first (ie T-Rex code). If you decide at some point to uninstall T-Rex, then that package will (hopefully) obediently restore the original event handler. This means that when the event fires, the Archaeopteryx code will no longer execute and Archaeopteryx will therefore not function correctly.

Even worse would be if the original package were poorly written and did not restore the original event handler, then when Archaeopteryx chains back to the previous handler it will be trying to jump to an address that no longer has code there. This will generate the inevitable Access Violation that we all love to loathe.

I know a little about Raptor. This offers a programming API to allow implementing plug-in extensions to it. These plug-ins, baby Raptors if you like, are able to talk to any IDE components and trap whatever events they choose without conflict. A particular internal Raptor mechanism ensures this smooth path is followed, the details of which are not available. However in the case of Archaeopteryx, we have to build our own safety net.

In order to try and avoid problems Archaeopteryx checks any event handlers that it chains to see if they have already been chained. If it spots one that has already been interfered with it will alert the user installing Archaeopteryx of the potential problems. But only the first time. Any further problems

```
procedure TPaletteTweaking.SetMultiLine(Value: Boolean);
var OldRows: Integer;
begin
  OldRows := Stuff.FTabControl.Perform(tcm_GetRowCount, 0, 0);
  Stuff.FTabControl.MultiLine := Value;
  { If MultiLine property changes, window gets recreated so need to set button
    status back as appropriate since we hacked that option: it ain't a property }
  SetButtons(FButtonsOption.Checked);
  UpdateIDESize(OldRows);
end;
procedure TPaletteTweaking.SetHotTrack(Value: Boolean);
begin
  Stuff.FTabControl.HotTrack := Value;
  { If HotTrack property changes, window gets recreated so need to set buttons
    back as appropriate since we hacked that option: it ain't a property
  SetButtons(FButtonsOption.Checked)
end;
procedure TPaletteTweaking.SetButtons(Value: Boolean);
var Style: Longint;
begin
  { TTabControl/TPageControl doesn't make buttons facility available as property}
  Style := GetWindowLong(Stuff.FTabControl.Handle, gwl_Style);
  if Value then
    Style := Style or tcs_Buttons
  else
    Style := Style and not tcs_Buttons;
  SetWindowLong(Stuff.FTabControl.Handle, gwl_Style, Style);  //Set window style
end;
```

➤ *Listing 8*

```
procedure TPaletteTweaking.DoTimer(Sender: TObject);
begin
  { Triggers shortly after Delphi starts (or whenever package is initialised)
    Only perform settings if no error message (such as a package load failure).
    Errors are shown with MessageDlgs which are of type TMessageForm. Let timer
    keep running until it's gone so the settings do actually take effect }
  if not (Screen.ActiveForm.ClassName = 'TMessageForm') then begin
    FTimer.Enabled := False;
    SetMultiLine(FMultilineOption.Checked);
    SetHotTrack(FHotTrackOption.Checked);
    //Don't need to call this as both the previous routines do it anyway
    //SetButtons(FButtonsOption.Checked);
  end
end;
```

➤ *Listing 9*

are ignored (a registry setting ensures this).

Before investigating this safety net further let's see where this event chaining requirement crops up in Archaeopteryx. Firstly, when you right-click on the component palette, a menu pops up. One of the Archaeopteryx functions (implemented in PalettePopup.Pas) is to add individual menu items to this menu for each page of the component palette. In order to ensure that the items accurately reflect the current state of the component palette, whenever the menu pops up, we need to delete the old menu items and add in new items to match the current set of tabs. This means we need to chain the popup menu's `OnPopup` event.

When the Delphi main window is resized it might cause the multi-line component palette to generate extra or fewer rows and so we need to chain the form's `OnResize` event to catch this and grow/shrink the form's height as required.

Finally, if the component palette is in a multi-line state it seems that `Tools | Environment Options...` causes a cosmetic problem. When you close the options dialog there is an unpleasant degree of flickering as the palette gets redrawn repeatedly for some reason. To alleviate this we can chain that menu item's `OnClick` handler, set the multi-line option off before chaining to the old handler, and then when the dialog is closed, set the multi-line option back again.

As it happens there are a couple of extra requirements that could be satisfied with event handlers.

Some minor limitations of Archaeopteryx include when a component is installed (real or template), or the palette reconfigured (using either of the two available menu items) or the component palette resized using the splitter between it and the speed bar, the multi-line palette may also change the number of lines of tabs displayed. All those

IDE options are triggered through events which could also be chained in much the same way as the previous one. However my README.TXT file informs you that I took the easy way out by suggesting the user turn the multi-line option off before taking any of these actions and turning it back on again afterwards.

In all these cases a data field is required to store the old handler and an appropriate method is defined to act as the new handler.

Before finally changing the original event property value the aforementioned interference check is made through a call to `TestChainedEventHandler`. This takes two parameters, the address of the code of the current event handler, and the address of the original event handler as set up by Borland. An example of all this chaining is shown in Listing 10 (all the extraneous bits of code have been removed from the listing to make it easier to follow) along with the implementation of `TestChainedEventHandler`.

The event handling method is typecast into a `TMethod` record and the code address is passed along. The Resource Explorer was used to get the name of the original event handling method from the form resource. This is then turned into an address using the main form's `MethodAddress` method. If the two don't match, a message is generated (the first time).

Final Comments

One more thing to discuss before letting you loose on the code on this month's disk: the Window menu. Firstly, because other add-in packages offer Window menus, this menu can be toggled on and off using a menu item on the Archaeopteryx `Options` menu. Also, to avoid clashes with other menu shortcuts the menu caption (which defaults to `W&indow`, making `Alt-I` the shortcut) can be customised with a registry entry.

When you add new menus onto the main menu bar there is something to be made aware of. In Delphi 3's `OnIdle` event, all the item's that show on the main menu bar have their `OnClick` events triggered. This is to allow the speedbar shortcut buttons to be enabled and disabled as required.

For example, when the `Edit` menu's `OnClick` handler is triggered, it ensures that the `Cut`, `Copy` and `Paste` speedbuttons (if present) look suitable as you work in the editor, selecting and unselecting text. Because of this, you should not put too much code in this `OnClick` handler, or perhaps even better would be to only execute your code if the menu is dropped down (that is the `OnClick` is a real one, not a fake one from Delphi). The `OnClick` will be real if the `Sender` parameter in the `OnClick` handler represents the menu item concerned. If it represents something else (`Application.MainForm`) then it is a fake `OnClick` from the `OnIdle` handler.

The job of the Window menu's `OnClick` handler is to ensure that the menu items that the user sees accurately reflect which windows Delphi has open, and to indicate which one is the active window. In implementing this I encountered two primary problems. The first was in my approach to find which windows Delphi had open. I thought it would be sensible to iterate either through `Application.Components` or `Screen.Forms`. This got most windows but missed the form designer windows. Delphi clearly keeps track of these in some other manner internally. This I remedied using Windows API calls to track down which visible windows were part of the Delphi process (see Listing 11).

The other problem came when I tried to make the Window menu a bit more friendly. Trying to add a shortcut of `F11` for the Object Inspector item in the menu caused a bit of a problem under NT. I was rather naively using `TextToShortCut` to manufacture the shortcut and was initially not differentiating between real menu clicks and fake Delphi ones. This effectively meant I was calling `TextToShortCut` over and over again as the `OnIdle` event handler triggered my `OnClick` event. Unfortunately it seems that `TextToShortCut`'s implementation falls foul of a little known NT memory leak. So the memory allocated to the Delphi process (according to NT Task Manager) kept increasing by about 8Kb per second. Eventually, after a

➤ *Listing 10*

```
TPalettePopup = class(TObject)
private
  ...
  FOldPaletteOnPopup: TNotifyEvent;
protected
  ...
  procedure DoPalettePopup(Sender: TObject);
public
  constructor Create;
  destructor Destroy; override;
end;
constructor TPalettePopup.Create;
begin
  inherited Create;
  ...
  { Chain component palette OnPopup event: may cause
    problems if someone else chains on to it afterwards, and
    then we are deleted. The later chainer will be referring
    to dead code -> AV time. Save old OnPopup handler }
  FOldPaletteOnPopup := Stuff.FPalettePopup.OnPopup;
  //Warn user if event was already chained
  TestChainedEventHandler(TMethod(FOldPaletteOnPopup).Code,
    Application.MainForm.MethodAddress(
      'PaletteLocalPopup'));
  //Replace Delphi's event handler with our own
  Stuff.FPalettePopup.OnPopup := DoPalettePopup;
end;
destructor TPalettePopup.Destroy;
begin
  ...
  //Unchain the chained event handler
  if Assigned(Stuff.FPalettePopup) then
    Stuff.FPalettePopup.OnPopup := FOldPaletteOnPopup;
  inherited Destroy;
end;
procedure TPalettePopup.DoPalettePopup(Sender: TObject);
begin
  ...
  //Chain onto old OnPopup event handler
  if Assigned(FOldPaletteOnPopup) then
    FOldPaletteOnPopup(Stuff.FPalettePopup)
end;
procedure TestChainedEventHandler(
  OldHandler, NewHandler: Pointer);
begin
  { If original (as designed) handler and current handler of
    an event are not same, report error to user first time }
  if (OldHandler <> NewHandler) and Stuff.Ini.ReadBool(
   'Archaeopteryx', 'Warning', True) then begin
    MessageDlg(SChainingWarning, mtWarning, [mbOk], 0);
    //Set registry flag so the error is not reported again
    Stuff.Ini.WriteBool('Archaeopteryx', 'Warning', False)
  end
end;
```

```
procedure TWindowMenu.DoWindowMenuClick(Sender: TObject);
var
  Loop, Count, OldCount, PID: Integer;
  Item: TMenuItem;
  Wnd: HWnd;
  WndClass, WndCaption: array[0..255] of Char;
begin
  { Delphi repetitively calls OnClick events of main menu
    items to allow IDE code to en/disable speedbuttons as
    necessary. We will only execute this code if it is a
    real menu click (Sender = the menu item), not a fake one
    from Delphi (where Sender = the main window) }
  if Sender = FWindowMenu then begin
    Count := 0;
    { It would normally be sensible to delete old items then
      add new items. But for some reason that goes screwy
      UI-wise, so instead we add new ones then delete old.
      So, how many Window menu items were there? }
    OldCount := FWindowMenu.Count;
    //Add new menu items for current windows
    Wnd := GetWindow(Application.Handle, gw_HWndFirst);
    while Wnd <> 0 do begin
      GetClassName(Wnd, WndClass, 255);
      GetWindowThreadProcessID(Wnd, @PID);
      //We only want windows in the Window menu
      //that are... visible, enabled, have a caption,
      //are not icon captions, are part of the Delphi
      //process and are not the Application window
      if IsWindowVisible(Wnd) and IsWindowEnabled(Wnd) and
        (GetWindowText(Wnd, WndCaption, 255) > 0) and
        (PID = GetCurrentProcessID) and
        (StrPas(WndClass) <> '#32772') and
        (StrPas(WndClass)<>Application.ClassName) then begin
        Inc(Count);
        //Make a new menu item, remembering to put the
        //checkmark on the currently selected page
        Item := NewItem(Format(SFormat,
          [Count, StrPas(WndCaption)]), 0,
          Wnd = Screen.ActiveCustomForm.Handle,
          True, DoWindowItemClick, 0, '');
        Item.RadioItem := True;
        //Set up some unique group index to make
        //menu items work like radio buttons
        Item.GroupIndex := 57;
        //Put F11 next to Object Inspector item as reminder
        if StrPas(WndCaption) = 'Object Inspector' then
          Item.ShortCut := vk_F11; //Don't use TextToShortCut!!
        FWindowMenu.Add(Item);
        //Ensure menu item has a reference to relevant form
        Item.Tag := Wnd;
      end;
      Wnd := GetWindow(Wnd, gw_HWndNext)
    end;
    //Add About menu item
    FWindowMenu.Add(NewLine);
    FWindowMenu.Add(NewItem('&About Archaeopteryx...', 0,
      False, True, Stuff.DoAbout, 0, ''));
    //Now delete old (potentially wrong) window menu items
    for Loop := 1 to OldCount do FWindowMenu.Items[0].Free;
  end
end;
```

➤ *Listing 11*

few hours, NT would grind to a halt because of Archaeopteryx (or really because of `TextToShortCut`).

There is plenty more code in the package but you can peruse it for yourself. I've said all I think I need to tell you about my findings writing this thing, so I'll leave it down to you now. However it's worth considering that with this new package model and the desire to write add-ons being reasonably large, it might not be long before we see an official API available for extending the IDE. I'll leave you with that thought. You can find more information about Raptor at:

`http://www.eagle-software.com`

and about Merlin at:

`http://www.boots.com/merlin`.

Brian Long is a UK-based freelance Delphi and C++ Builder consultant and trainer. He is available for bookings and can be contacted by email at brian@blong.com.