

Tech Notes

Top Ten Reasons to Upgrade from Delphi 7 to Delphi 2009

Nick Hodges, Embarcadero Technologies

December 2008

Corporate Headquarters
100 California Street, 12th Floor
San Francisco, California 94111

EMEA Headquarters
York House
18 York Road
Maidenhead, Berkshire
SL6 1SF, United Kingdom

Asia-Pacific Headquarters
L7. 313 La Trobe Street
Melbourne VIC 3000
Australia

There are still a few die hard developers out there that are using Delphi 7. They do so in the face of an overwhelming onslaught of features and capabilities of the latest version of Delphi, Delphi 2009. Below, I discuss ten reasons why Delphi 7 developers should upgrade to Delphi 2009.

Now, mind you, I am limiting this article to ten items. It could easily be way more than that. But I figure that if these ten aren't enough, I can write "Ten More Reasons to Upgrade from Delphi 7".

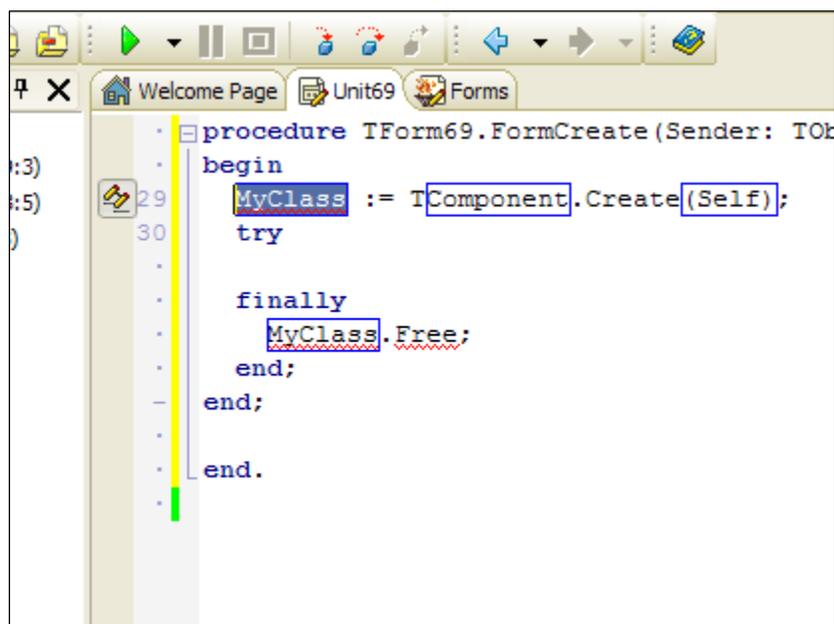
THE INTEGRATED DEVELOPMENT ENVIRONMENT

The Integrated Development Environment (IDE) has been drastically improved and enhanced since Delphi 7. Numerous features and a complete rearchitecting have added countless productivity enhancements that puts the Delphi 2009 IDE light-years ahead of the older IDEs found in Delphi 7 and earlier versions. This section will highlight just a few of those changes.

LIVE TEMPLATES

In my view, Live Templates are one of the coolest – and one of the most under-appreciated features of Delphi 2009. They might take a bit of getting used to, but they are extremely powerful, they can really improve your productivity, and they are completely customizable. If you don't like the way that they work, you can change any of the templates. If you want the system to do more than it does, you can easily create your own templates. In addition, you can even write your own scripting engines to make it do almost anything you want.

Live templates are an editor feature that enable you to quickly inject and fill in any type of code construct into your code. The default set of Live Templates includes templates for all the standard language constructs like if, while, case, for, and many others. In addition, many of



these templates include scripting capabilities that add further power to the Live Template feature. For instance, if you declare a known enumerated type with a case statement, the Live Template engine will fill in all the enumeration elements inside the case statement for you, saving you a whole lot of time, keystrokes and formatting.

In addition, the templates make it easy to create common coding constructs such as class declarations, class instantiations with a block (See Figure 1), functions and methods, and more.

Figure 1 -- The Live Scripting template for creating a class instance instance, including the try...finally block.

Live templates themselves are actually simply XML files that describe a text caption to be replaced and the text to replace it. You can create a live template to insert any code or text you want into the code editor. If you have, say, a standard header you put at the beginning of every file, you can create a live template to insert it, even creating “code points” to insert specific information at specific places. I’ve used them in demos to insert large chunks of code that illustrate the feature being demonstrated. I also have some live templates that correct some of my common typographical errors (e.g, replacing `stirng` with `string`). The possibilities are endless, and every live template can drastically increase your productivity by saving you time and keystrokes.

In addition, the XML files can specify which language the template should be used with, and which scripts should be run on the template as well. You can create your own live templates from scratch. Or even better, you can use the live template that creates new live templates! Live templates are also completely scriptable. You can build your own scripting engine using Delphi to define almost any behavior during the use of a live template. I’ve created a set of classes that make it easy to create a custom template, including templates that make it easy to inject the date and time into your source code. [You can get it here at CodeCentral](#). Give it a look and see how easy it is to do almost anything you want with Live Templates.

MORE INFORMATION

- [Live Templates on the Delphi Wiki](#) - a collection of useful live templates.
- [Technical Information about Live Templates on the Delphi Wiki](#)
- [A video demonstrating Live Templates](#)

HISTORY TAB

Sometimes you are coding, and you realize that you’ve just headed in a direction that you really don’t want to go. Sometimes you delete a whole bunch of code and a few minutes (and a few CTRL-S events) later, you think, “Uh oh”. And of course, this always happens without having a good backup or a good save point in your source control management tool. (You do backup and you do use a source control management tool, don’t you?) Enter the IDE’s history tab. The History Tab is sort of a “poor man’s source control system”. It will save physical files of your code each time you save a file, ensuring that there is a history of your changes. Set the number of files saved to a configurable value, save your code frequently, and you can be sure that any changes you make can be easily undone.

The history tab keeps track of your files by placing the backup copies in a special hidden directory. It tracks changes to any text file that is part of your project – including *.DFM files – and provides you with a nice interface that allows you to peek into any of the files from your history. It even has a simple file compare tool that enables you to do a diff on two separate files.

One side benefit I’ve found with the History Tab is that it makes me a bit “braver” when I’m coding – I feel much more inclined to try something a little “scary” because I know that I can really easily go back to a known state without having to do a check in to the source control system.

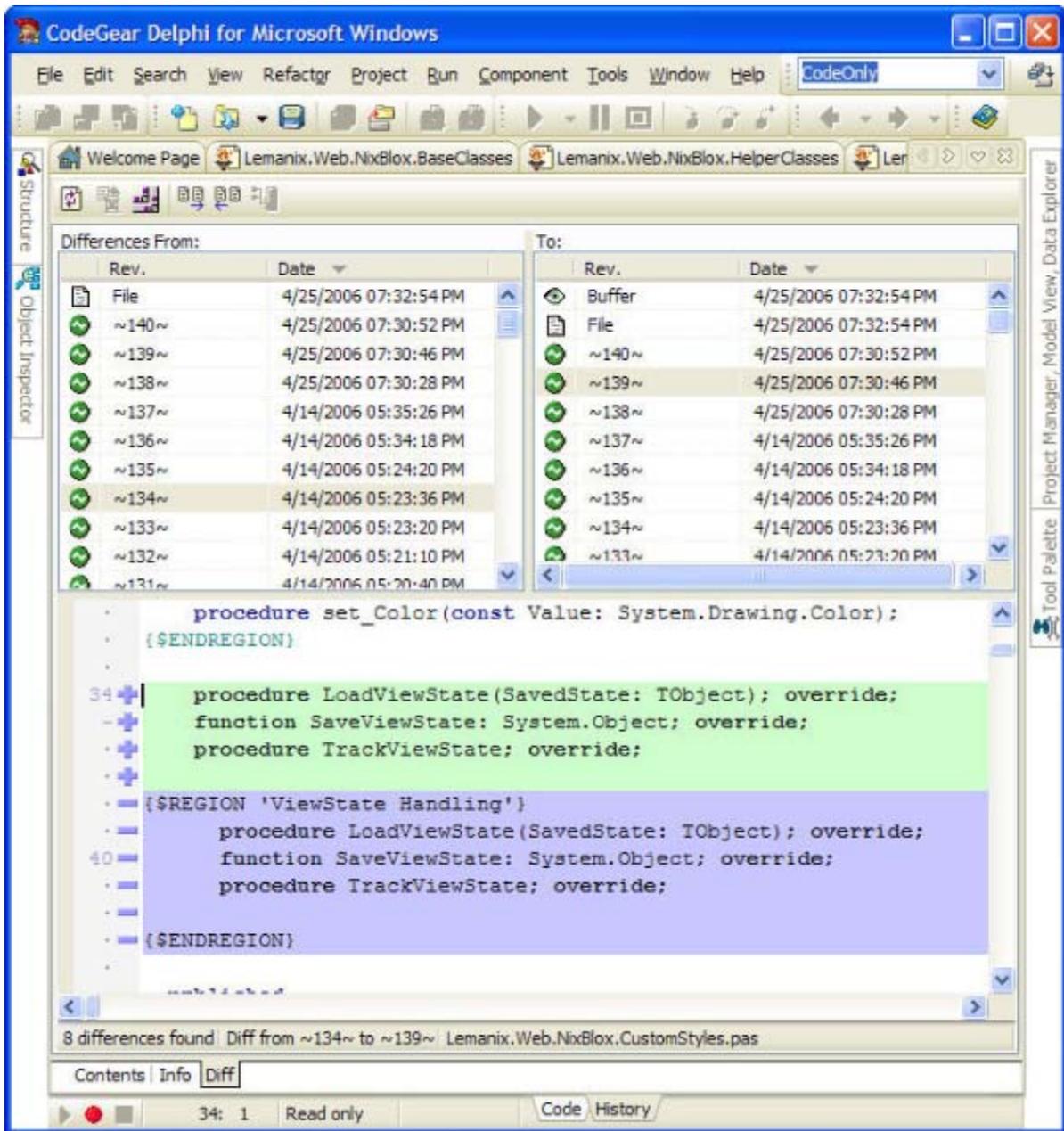


Figure 2 -- The History Tab showing the file compare feature.

DOCKABLE/CUSTOMIZABLE IDE

VCL DESIGNER GUIDELINES

Setting up forms is no fun, especially forms with a lot of components on it. Getting things aligned and spaced just right can be challenging. And those pesky users can notice when one control is out of place by a single pixel. Previous versions of Delphi provided alignment tooling for ensuring that controls line up properly, but they often took time, and didn't always make things as easy as they could be. Delphi 2009 solves this problem very nicely by providing VCL control guidelines.

The VCL Control guidelines provide a visual reference for seeing when controls are properly spaced and aligned. As you drag controls on a form, guidelines appear that show when a control is properly aligned with another control. The designer will even "snap" the controls into place as they are moved, making it very easy to align controls properly.

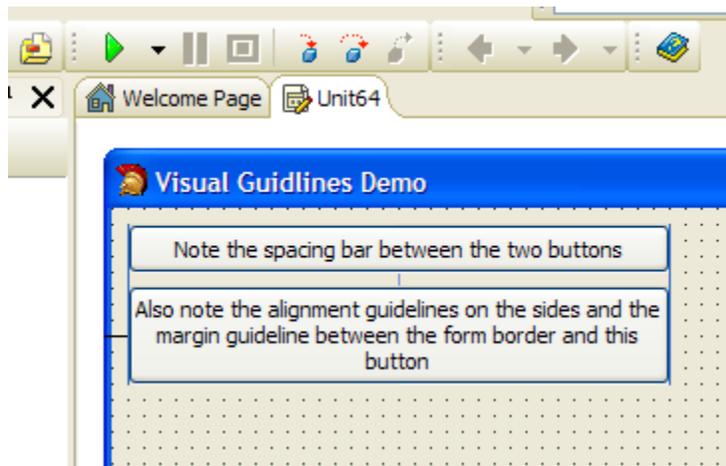


Figure 3 -- The VCL Designer guidelines showing alignment and spacing

For spacing controls, the guidelines show a small line between controls when they are spaced properly according to your settings.

The VCL also has been enhanced to support this, so that controls can define their own padding and margins, ensuring that the designer knows how a control wants to be spaced away from other controls. Padding and Margin can also affect a control when its Align property is set.

Component designers can also make their components "guideline aware" if their controls have any special requirements for alignment.

The VCL Designer Guidelines make it almost pathetically easy to get a form set up properly. I find that I almost never use the Align and Spacing dialogs anymore. It's just way to easy to align the controls right as you put them on the form – a great productivity enhancer.

NEW TOOL PALETTE

Personally, I find the new Tool Palette to be my favorite features of the new IDE. It is incredibly easy to find and add a component, even when you have lots and lots of components on it. The main reason is the filtering feature.

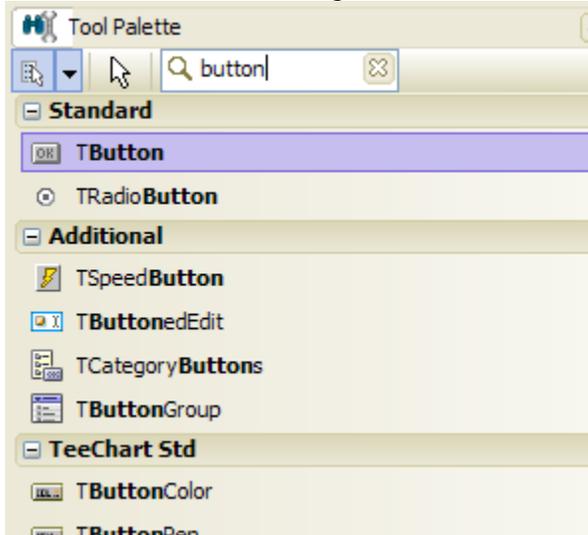


Figure 4 -- The Tool Palette showing the filtering feature.

FURTHER INFORMATION

I made [a video showing off the new tool palette](#).

REFACTORING

THE LANGUAGE

The Delphi language has had numerous feature improvements since Delphi 7 – features that greatly enhance the power available to the Delphi developer. This section outlines the latest language features that no Delphi developer can do without.

GENERICS

A new language feature in Delphi 2009, Generics are a powerful addition to Delphi. Generics allow you to define classes that don't specifically define the type of certain data members. Generic classes allow you to write a single class that can act the same way on multiple types without knowing what those types will be ahead of time.

For instance, the new unit in Delphi 2009 named `Generics.Collections.pas` contains a generic `TList<T>` class that enables you to maintain a type safe list of any type at all. For example, you can declare:

```
type
  TPerson = record
    FirstName: string;
    LastName: string;
    Birthday: TDate;
  end;

var
  PeopleList: TList<integer>;
  Person: TPerson;
begin
  Person.FirstName := 'Donald';
  Person.LastName := 'Duck';
  Person.Birthday := EncodeDate(1809, 2, 12);

  PeopleList := TList<TPerson>.Create;
  try
    PeopleList.Add(Person);
    // Add more items, and do something with the list
    ...
  finally
    PeopleList.Free;
  end;
end;
```

The code above uses the generic `TList<T>` to maintain a list of an arbitrary type, in this case a simple record. As a result, you have a single class that hold a list of anything, rather than having to create – and thus maintain – a whole bunch of specialized descendent classes to maintain lists of arbitrary types. One class means less code to worry about and fewer bugs.

Generics also allow the parameterized type to be constrained, either to a specific class, to a class that implements a specific interface, or to require that the class have a constructor or be record. For instance, you can declare the following class:

```
TControlReporter<T: TControl, IEnumerable> = class
private
  FInternalType: T;
public
  constructor Create;
  procedure ProcessComponent;
  procedure SetType(aT: T);
end;
```

Given the above, you must pass a TControl or TControl descendent to any instance of TControlReporter<T>, and in addition, it must implement the IEnumerable interface. Any other types will cause a compiler error. In this case, generic constraints allow you to use the type internally as if it were a TControl or as if it had the methods of IEnumerable. In this way, you can create more powerful and specific generic classes. Without the ability to constrain generic types, generic members would be able to provide very little functionality.

As mentioned above, Delphi 2009 now includes the Generics.Collections.pas unit, which defines generic lists, queues, stacks and dictionaries for use in your code.

MORE INFORMATION

More information about Generics in Delphi can be found in the White Paper entitled "[Using New Delphi Coding Styles and Architectures.](#)"

ANONYMOUS METHODS (CLOSURES)

Another major language feature in Delphi 2009 is Anonymous Methods (sometimes called Closures). An anonymous method in Delphi is a mechanism to create a method value in an expression context. Anonymous methods allow you to pass code blocks as parameters, or to assign code blocks to variables. Anonymous methods are similar to method pointers, but they have one key difference noted above: variable capture. Anonymous methods can capture the state of the variables used in their code based upon the values set within the same scope as the anonymous method.

Since Delphi is a strongly-typed language, you always need to declare a type before you can actually use an anonymous method. For example:

```
type
  TIntProc = reference to procedure (n: Integer);
```

The above declares a simple anonymous method type that takes a single integer parameter. Given the above type declaration, you can now declare a code block of this type:

```
var
  anIntProc: TIntProc;
begin
  anIntProc :=
    procedure (n: Integer)
```

```
begin
  Memo1.Lines.Add (IntToStr (n));
end;
.....
anIntProc(42);
end;
```

Once the type is declared, you can, as shown above, declare a procedure that is assigned to the variable of the anonymous type. (Note the 'anonymous' part – the procedure doesn't have a name.) Once the variable is defined, you can then call the anonymous method, passing the correct parameter along.

Variables of the type of an Anonymous method can be passed as parameters to any method, can be declared and assigned to as variables and properties, and can be used in any way that standard variables are used.

The real power of anonymous methods come when they do local variable capture. For an example of that, I'd recommend reading our White Paper "[Using New Delphi Coding Styles and Architectures.](#)"

UNICODE SUPPORT

THE VISUAL COMPONENT LIBRARY

RIBBON CONTROLS

The VCL has always been about creating great looking user interfaces, and Delphi 2009 keeps that tradition going in a big way. Delphi 2009 includes a complete implementation of the Office 2007 user interface, commonly called Ribbon Controls.



Embarcadero Technologies, Inc. is a leading provider of award-winning tools for application developers and database professionals so they can design systems right, build them faster and run them better, regardless of their platform or programming language. Ninety of the Fortune 100 and an active community of more than three million users worldwide rely on Embarcadero products to increase productivity, reduce costs, simplify change management and compliance and accelerate innovation. The company's flagship tools include: Embarcadero® Change Manager™, CodeGear™ RAD Studio, DBArtisan®, Delphi®, ER/Studio®, JBuilder® and Rapid SQL®. Founded in 1993, Embarcadero is headquartered in San Francisco, with offices located around the world. Embarcadero is online at www.embarcadero.com.