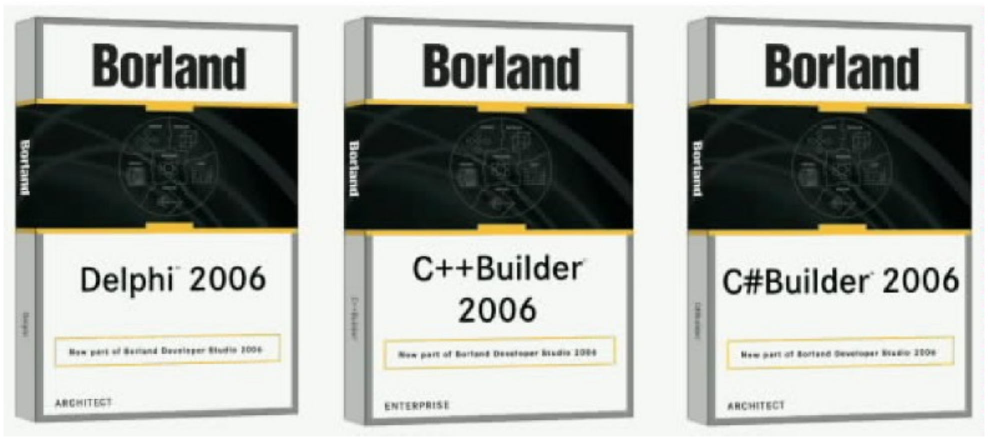
Borland® Developer Studio 2006
Reviewer's Guide

The Complete Windows® Development Solution



A Borland White Paper

Produced for Borland by Cary Jensen, Jensen Data Systems, Inc.

Contents

Overview

Welcome to the Borland® Developer Studio 2006 Win32 Reviewer's Guide. This guide is designed to familiarize you with the wealth of new and improved features that you will find in Borland Developer Studio 2006, the next generation in the evolution of Borland's venerable Delphi™ product line.

Borland Developer Studio 2006 is the complete solution for building applications for the Windows and .NET platforms. No other product offers the range of tools that Borland provides you in this one product. From collaborative requirements management with CaliberRM™ to UML-based design with Borland® Together® to team version control and collaboration with StarTeam®, to Model Driven Architecture™ (MDA) with ECO™ (enterprise core objects), Borland Developer Studio 2006 provides you with the peerless support tools necessary to ensure the success of your projects. Having these support tools married to the state-of-the-art suite of development tools for Windows and .NET in Delphi 2006, C#Builder 2006, and C++Builder® 2006 is a match made in heaven.

Borland Developer Studio 2006 is the result of a convergence of decades of Borland technologies, combined with the thoughtfully integrated support tools that Borland has acquired in recent years. Finally, everything that you need is seamlessly blended into the most complete Windows IDE for Borland yet.

One IDE, Multiple Personalities

Whether you are coding in Delphi or C#, writing Win32 applications or .NET managed code, building ASP.NET Web pages or traditional client applications, Delphi 2006's IDE provides you with a consistent and powerful set of development tools designed to increase your productivity. With Delphi 2006, the IDE keeps track of what kind of application you are working with, providing you with the designers, views, and features consistent with the task at hand. If you create a new Win32 client application, or open an existing one, the VCL (visual component library) designer kicks in, again providing you with unmatched support for designing your user interfaces. You can even create project groups that include two or more different kinds of projects. When you do this, the type of application that is currently active in the project group determines which designers are available, and which options you see in the supporting views. For instance, if your project group includes both an ASP.NET Web Service application and a Win32 VCL Form application, Delphi 2006 notes which of these projects is currently active, providing you with the designer and editor appropriate for each as you switch between your projects.

One IDE, Multiple Languages

Delphi 2006 is more than just context-sensitive designers — it is a full multiple-language development environment. The native languages and debuggers that are included in Delphi 2006 are Delphi for Win32 development, Delphi for the Microsoft .NET Framework, and C# for the Microsoft .NET Framework. While other IDEs support multiple languages, Delphi 2006 is unique in that it supports both multiple platforms and multiple languages transparently. For example, you can create a project group that includes a C# ASP.NET Web application, a Delphi for .NET Web Control class library, and a traditional Windows DLL (dynamic link library) written in Delphi Win32. Not only will the appropriate compiler and debugger be used for each project, based on its underlying language, but also the code editor features and Tool

Palette will expose the appropriate features as you navigate between the various projects. Delphi 2006 can also support additional compilers, if you wish. For example, so long as you have the VB for .NET compiler installed on your workstation, you can create, open, edit, compile, and debug VB for .NET applications without ever leaving the Delphi 2006 IDE.

This reviewer's guide is organized by the major areas of software development and support in Borland Developer Studio. Each section begins with a general overview, which is then followed by a description of the many related updates, enhancements, and additions introduced in this product. You begin with a discussion of the updates you will find in the Delphi Language. From there, you will learn about the improvements to the designer, editor, debugger and database support.

Disclaimer
Not all features described here apply to all editions and all versions of Borland Developer Studio 2006. Please review the Borland Developer Studio 2006 feature matrix to learn which features you will find in each edition. Your link to this feature matrix is located at http://www.Borland.com/Delphi.

Delphi Language Enhancements

The Delphi language is mature, providing you with nearly everything that you need and want in a sophisticated and modern programming language. Nonetheless, Borland has managed to add a couple of additional enhancements. These are described in the following sections.

Borland compilers are legendary for their speed and compatibility, and this legacy continues with Delphi 2006. Actually, Delphi 2006 ships with three compilers. One of these compilers, the C# compiler, is licensed from Microsoft. Consequently, C# applications you build in Delphi 2006 generate the same intermediate language (IL) code as those built with Visual Studio. The other two compilers are Delphi compilers, one for compiling traditional 32-bit Windows executables and the other for generating IL for the .NET Framework. Both of these compilers have received significant updates in the Delphi 2006 release. This section begins with a discussion of features added to both the Win32 and the .NET versions of the Delphi compilers. Later in this section, you will learn about the new features that are specific to one or the other of these compilers.

Updates for Both Win32 and .NET Delphi Compilers

Several new features have been added to both of Delphi 2006's Delphi compilers. The most significant of these include the new **`for...in loop`** and Unicode support. These new compiler features are described in the following sections.

The For...In Loop

The Delphi language has been updated to include a new looping control structure similar to the C# **`foreach`** keyword. In Delphi, this new loop is referred to as a **`for...in`** loop. Unlike traditional for loops in Delphi, the **`for...in`** loop does not require an ordinal control variable. Instead, the **`for...in`** loop systematically retrieves a reference to the next object in a collection of like objects. For example, the following code segment can be used to iterate through the DataRows of a DataTable's Rows property (this property is of the type DataRowCollection):

```
var
  Row: DataRow;
begin
  //...
  for Row in MyDataTable.Rows do
      ListBox1.Items .Add (Row[ 0] .ToString);
```

For the .NET Delphi compiler, for...in can be used with any object that satisfies at least one of the following conditions:

- it implements the IEnumerable interface
- has a public GetEnuermerator function
- is an array, set, or string

For the Win32 compiler, for...in can be used with any class that has

- a public GetEnumerator function, or
- is an array, a set, or a string.

Classes that implement a GetEnumerator function include TList, TCollection, TStrings, TMenuItem, TFields, to name a few.

Support for Unicode and UTF8 Formats

Both of Delphi's compilers can now compile UTF8 and Unicode source files. Previously, only ANSI source files were supported. For the Delphi for .NET compiler, this feature supports CLS (common language specification) standard Unicode identifiers in both metadata and in source code.

The Delphi for .NET Compiler

Borland's Delphi for .NET compiler made its first debut with the release of Delphi 8 for the Microsoft .NET Framework. In addition to the updates listed in the preceding section, this compiler has received a number of updates that apply specifically to .NET applications. These include a revision to how namespaces are created and managed, forward-declared record types, and support for weak packaging in VCL for .NET applications. The updates to the Delphi for .NET compiler are described in the following sections.

Delphi Code and Namespaces

The biggest change to the .NET compiler is in how it generates namespaces for the symbols defined in your units. Under the previous version of the compiler, the unit name was the namespace. For some developers, particularly those accustomed to using classes defined in C#, the namespaces created by Delphi appeared awkward. Specifically, these namespaces revealed the physical structure of the underlying code, which is irrelevant from the perspective of the person using your classes, and can be distracting.

The Delphi 2006 compiler takes a new approach to namespace generation, allowing multiple units, and even multiple applications, to contribute to a common namespace, if desired. At the same time, it is just as easy to make each unit contribute to a separate namespace. Here is how it works. If your unit names do not use dot notation, the unit name is the namespace. This is how it worked before.If a unit includes a multipart name, using dot notation, the namespace is defined by dropping the last part of the unit name. For example, if a unit has the name YourCompany.Data.Unit1, the classes within that unit will reside in the YourCompany.Data namespace. Classes that appear in the YourCompany.Data.Unit2 and YourCompany.Data.Unit3 units will be in the YourCompany.Data namespace as well. Global variables, constants, functions, and procedures declared in Delphi code represent something of a challenge, in that .NET requires all declarations to be associated with a class. Therefore, the global symbols of a Delphi unit named YourCompany.Data.Unit1 are implemented in .NET metadata as members of a class named Unit 1 within the namespace YourCompany.Data.Units.

How Delphi symbols appear in .NET metadata has no effect on your Delphi source code. You only need to consider how your Delphi code will appear in the .NET metadata for the portion of your code that you want developers using other .NET languages to use. In general, you should avoid using global variables, global constants, or global procedures and functions when writing Delphi code that you intend to be used by other .NET languages.

Support for Weak Packaging in VCL for .NET Applications

A runtime package in the VCL for .NET is a managed .NET assembly, it contains declarations that the application can load and use at runtime. Under normal circumstances, if you compile a VCL for .NET application to use a runtime package, you are required to deploy that package, just as you are required to ensure the deployment of all assemblies (DLLs) that are referenced in your application.
Weak packaging of a unit addresses a problem that arises when a runtime package contains one or more units that statically link to an external DLL, in particular, a DLL that is not commonly available. Under normal conditions, this situation requires that you deploy both the runtime package and the DLL.

Consider the Microsoft DLL PenWin.dll for pen device input, which is not distributed with Microsoft operating systems. The PenWin unit in Delphi statically links to the DLL PenWin.dll. If your unit uses PenWin, and includes calls to one or more functions in the statically linked PenWin.dll, adding your unit to a runtime package without weak packaging would require that the PenWin.dll be available from any application that loaded that runtime package. By making this unit weakly packaged, only applications that actually call PenWin.dll functions will require PenWin.dll.

Weak packing permits an application to link a non-packaged version of the unit into the executable instead of using the runtime package that contains this unit. As a result, applications that need the features of the weakly packaged unit will link the non-packaged version of the unit (that stored in the compiler-generated DCPIL file), and as a result, require the DLL. Applications that do not use the unit will not require the DLL, even if they are compiled to use the package that contains the weakly packaged unit. Weak packaging has been available for some time in the Delphi Win32 compiler. Weakly packaged unit semantics are now supported by the Delphi for .NET compiler.

Forward Declared Record Types

Record types can now be forward declared in Delphi VCL for .NET and FCL applications. A forward declared record instructs the compiler to recognize the record as a valid type, even though its formal declaration appears later in the same type block. Forward declared record types permit two type declarations, specifically records, classes, and interfaces, appearing in the same type block to reference one another in their member fields, properties, or methods. You create a forward declared record type by declaring the record type symbol but omitting the record's field lists.

The Delphi Win32 Compiler

The degree of compatibility between the Delphi Win32 and .NET compilers is one of the truly remarkable Delphi 2006 features. This compatibility permits single projects to be compiled as true Win32 applications and then effortlessly migrated to 100% .NET managed code applications. In many cases, a single set of source files can be compiled by both the Win32 and the .NET versions of the Delphi compiler. No other development environment lets you do this as easily. Equally compelling for developers is Borland's continued support for the Win32 platform with the most modern IDE on the market. While Borland is committed to the .NET platform as the future of Windows development, Borland also knows that the majority of desktop developers maintain applications on the Win32 platform, and Borland is just as committed to providing those developers with the advanced features that they need. Although the bulk of the enhancements to the Win32 compiler have already been described earlier in this guide (in the section "Updates for Both Win32 and .NET Compilers"), the following sections discuss some of the unique features added to the Delphi Win32 compiler in Delphi 2006.

Function inlining

Function inlining is an operation performed by the Win32 compiler at compile time. The feature was introduced in Borland Developer Studio 2005. When a function is inlined, the compiler replaces a call to the subroutine (a method, function, or procedure) with the compiled instructions defined within the subroutine. Function inlining can increase application performance by eliminating the overhead associated with function, procedure, and method calls. There are two ways to influence whether the compiler will inline a function or not. One way is to include the inline directive in the function, procedure, or method declaration. This directive is a request to the compiler to consider whether or not to inline the function. If inlining has not been disabled, and the compiler determines that the function can be safely inlined, the inlining will be performed. The second way is to use the {$INLINE} compiler directive. This directive can be passed with one of three parameters, ON, OFF, and AUTO. With the ON parameter, the default, the compiler will inline functions declared using the inline directive, whenever the compiler determines that inlining is safe. No inlining takes place when you specify the OFF parameter.

When you use the {$INLINE} compiler directive with the AUTO parameter, the compiler attempts to inline, if possible, any small function — one whose code size is roughly 32 bytes or less. While function inlining can produce performance improvements, Borland is quick to note that it should be applied judiciously, and does not recommend using the AUTO parameter with the {$INLINE} compiler directive.

Inlining can produce larger executables, even some that are dramatically larger. Also, inlined functions do not always produce performance benefits. In some cases, inlining can actually reduce performance. There are a number of conditions that prevent a subroutine from being inlined. For example, subroutines that include inlined assembly instructions cannot be inlined. Similarly, methods of a class that access one or more of that class's private members cannot be inlined into a method in another class. Borland has applied the inline directive to some of the smaller routines in the VCL and RTL, where deemed appropriate. As a result, code that uses these routines will execute faster than before, but with slightly larger executables.

More Inlining of RTL Functions

Even more of Delphi's runtime library (RTL) has been inlined in Borland Developer Studio 2006, including the Length function, which is one of the more frequently called functions. When a function is inlined, the compiler generates the code to perform the operation and inserts this code in place of each function invocation. Without inlining, the compiler generates the function code only once, and all calls to that function require a jump to the generated code. In some cases, such as in the Length function, the amount code that must be generated to perform the jump is greater that the amount of code in the function itself. With functions like Length, inlining produces more efficient code.

Support for Nested Types

A nested type is a type declaration inside another type declaration. The Delphi for .NET compiler already supports nested types. Delphi's Win32 compiler does now, too. The following is an example of a class that contains a nested type. This example is taken from the Delphi 2006 Help, and can be found under the heading Nested Type Declarations.

```
type
  TOuterClass = class

  strict private
  myField: Integer;

  public

  type
  TInnerClass = class
  public
    myInnerField: Integer;
    procedure innerProc;
  end;

  procedure outerProc;
end;
```

Nested Type Constants in Class Declarations

Nested type constants are constant class member declarations inside of a class type declaration. Nested type constants are somewhat similar to class functions, in that they can be referenced using a class reference without an instance of the class. Unlike class functions, however, nested type constants always return a constant value.
Nested type constants are already available for your .NET projects. Now you can use them in your Win32 applications as well. Nested type constants can be of any simple type, such as ordinal, real, and String. You cannot declare a nested constant to be a value type, such as TDateTime.

The following is an example of a class that includes a nested type constant declaration:

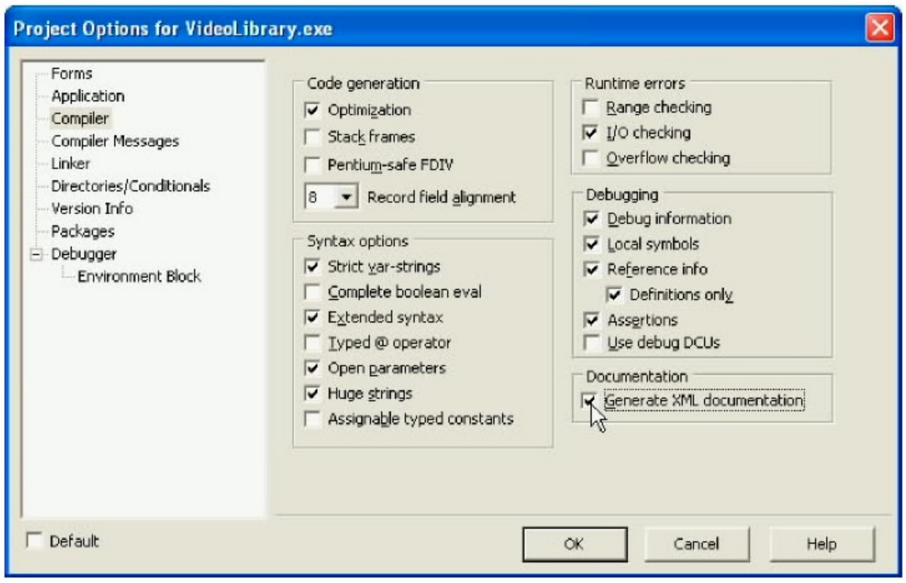
```
type
   TTemperatureConverter = class (TObject)
   public
   const AbsoluteZero = -273;

   procedure ConvertFtoC(Temp: Integer): Integer;
   //...
```

Support for Pentium 4 SSE3 and SSE2 Instruction Op Codes and Data Types

If you need to get close to the silicon, Delphi's Win32 compiler now permits you to include Pentium 4 SSE3 and SSE2 op codes and data types in your inline assembly routines.

XML Document Generation

XML document generation was introduced in the Delphi 8 for .NET and C#Builder compilers. You can now generate XML documentation files for your Win32 source code. To enable XML Doc generation, enable the Generate XML Documentation check box on the Compiler page of the Project Options dialog box. You display the Project Options dialog box by selecting Project | Options from the Delphi 2006 main menu.


Project Options for VideoLibrary.exe

- Forms
- Application
- Compiler
- Compiler Messages
- Linker
- Directories/Conditionals
- Version Info
- Packages
- Debugger
  - Environment Block

Code generation
- [✓] Optimization
- [ ] Stack frames
- [ ] Pentium-safe FDIV
- 8 Record field alignment

Syntax options
- [✓] Strict var-strings
- [ ] Complete boolean eval
- [✓] Extended syntax
- [ ] Typed @ operator
- [✓] Open parameters
- [✓] Huge strings
- [ ] Assignable typed constants

Runtime errors
- [ ] Range checking
- [✓] I/O checking
- [ ] Overflow checking

Debugging
- [✓] Debug information
- [✓] Local symbols
- [✓] Reference info
  - [✓] Definitions only
- [✓] Assertions
- [ ] Use debug DCUs

Documentation
- [✓] Generate XML documentation

[ ] Default

When Generate XML Documentation is enabled, the compiler produces one XML file for each of your source files. This file has the same name as the source file, but with the .xml extension. If you have included custom XML Documentation comments in your source files, these will be inserted into the generated XML file.

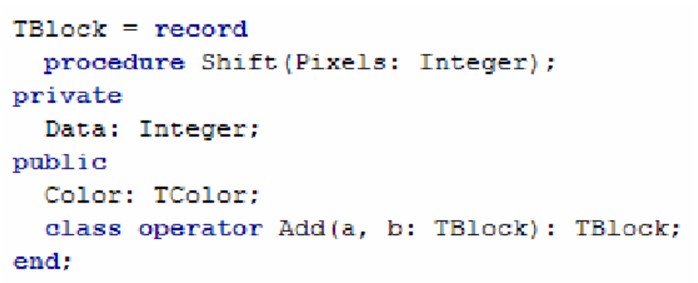
The XML files generated when you compile with Generate XML documentation enabled can be used with widely available documentation generating tools. Alternatively, you can write your own XML parser to use this information any way you see fit.

Enhanced Support for Records

Several enhancements have been introduced for record structures. The most important of these is that records now support properties and methods. While a record with properties and methods sounds more like an object, it is not. Specifically, records are value types and are allocated on the stack. By comparison, objects are reference types and are allocated on the heap. Additional differences are that records do not support inheritance whereas objects do, records use copy on assignment semantics while objects do not, and records include an automatic constructor while objects must be explicitly instantiated.

One additional capability added to records is operator overloading. You create an operator overload by introducing a class operator method into your record declaration using one of the following names: Implicit, Explicit, Negative, Positive, Inc, Dec, LogicalNot, BitwiseNot, Trunc, Round, Equal, NotEqual, GreaterThan, GreaterThanOrEqual, LessThan, LessThanOrEqual, Add, Subtract, Multiply, Divide, IntDivide, Modulus, ShiftLeft, ShiftRight, LogicalAnd, LogicalOr, LogicalXor, BitwiseAnd, BitwiseOr, and BitwiseXor. Your implementation of the declared class operator method defines how the overloaded operator behaves.

The following is an example of a record type named TBlock. The add (+) operator of this record is overloaded:



```
TBlock = record
  procedure Shift(Pixels: Integer);
private
  Data: Integer;
public
  Color: TColor;
  class operator Add(a, b: TBlock): TBlock;
end;
```

Record declarations in Delphi for .NET applications support interfaces, but those in Win32 Delphi do not.

Class Variables

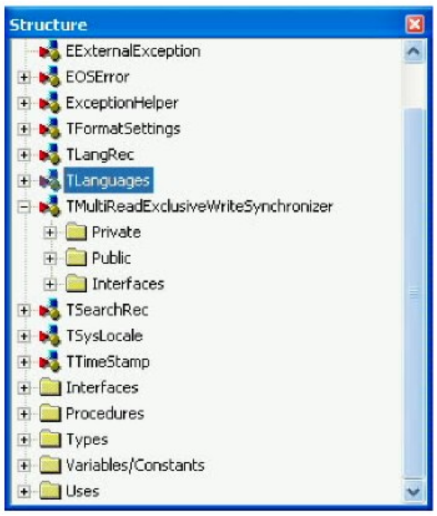
Classes in Delphi for Win32 now support class variables. A class variable is stored in a single memory location that is shared by all member of the class, as well as by descendants of that class. Class variables join the static class constants that were introduced into Delphi for Win32 in Delphi 2006.

The Integrated Development Environment

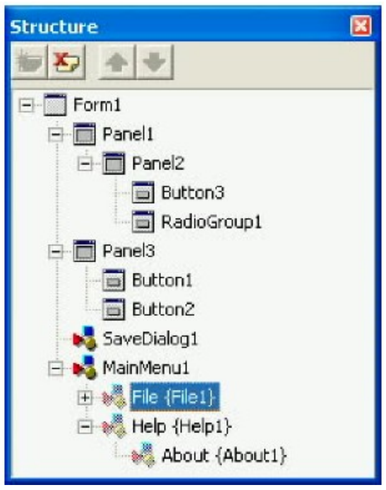
The Delphi IDE (integrated development environment) represents state-of-the-art in software development tools enabling you to develop applications faster and better. This section focuses on the features found in the various panels, designers, dialog boxes, and views of the IDE. Features that are specific to the code editor are detailed separately in a later section of this guide.

The Structure Pane

The Structure pane is a context-sensitive view that provides you with detailed information about what ever is displayed in your main view. When you are using the code editor, the Structure pane displays the classes, types, interfaces, and other symbols in the current file, as shown in the following figure. (In Delphi 7, this view was called the Code Explorer.)



By comparison, when you are designing a VCL Form, the Structure pane displays the components that appear on your form, with the various nodes representing the containership of your controls.
(In Delphi 7, this view was referred to as the Object Tree View.)



Not only does the Structure pane provide you with valuable insight into your projects, it also serves as a convenient tool for navigating the symbols and objects that you are using. When you are editing your code, double-clicking a symbol in the Structure pane takes you to the associated line of code in the editor. When you are designing a VCL Form, clicking an object selects it in the designer, permitting you to quickly change its properties or assign event handlers.

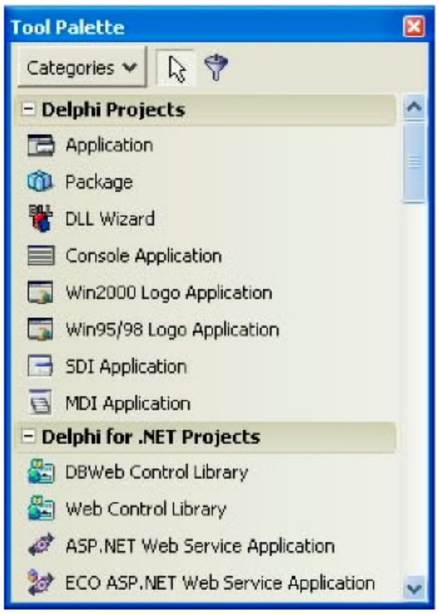
The Structure pane is also invaluable when there are errors in your code. When Delphi 2006's new Error Insight feature identifies problems in your source files, these appear automatically, as you type, in the Structure pane, permitting you to quickly navigate to the position in the code editor where problems exist. Error Insight is described in more detail in the "The Next Generation Code Editor" section of this guide.

The Tool Palette

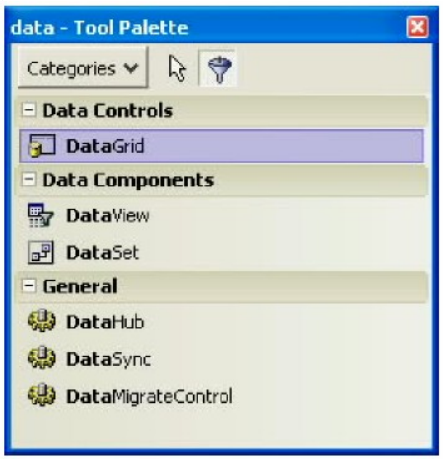
When you work in a component-based environment like Delphi 2006, you typically make extensive use of design-time components, which are placed into the designer and configured using the Object Inspector. These components are available from the Delphi 2006 Tool Palette. The Tool Palette is organized by component category. Which categories are displayed, and which components appear within them, is context sensitive, based on the type of project on which you are working. Furthermore, the Tool Palette permits you to controls its organization. You can change the position of a component within a Tool Palette category, as well as move a component to a different category, simply by dragging the component within the Tool Palette. You can even define your own custom categories into which you can drag your components.

Enhanced Tool Palette Behavior

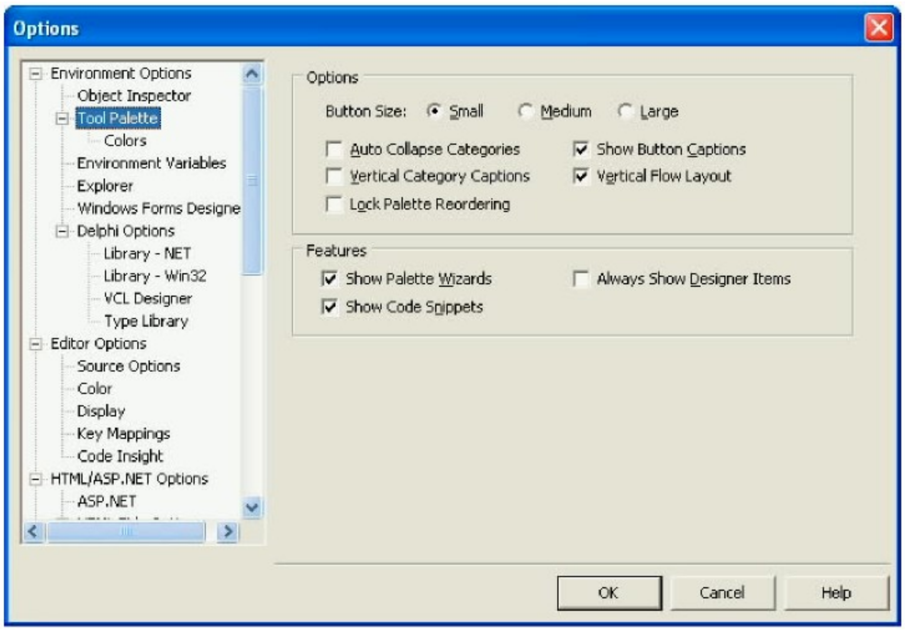
Delphi 2006's Tool Palette is better than ever. In addition to providing access to design-time components and code snippets, depending on whether you are working with a designer or code editor, the updated Tool Palette can also be used to create new projects, files, and objects. When you do not currently have a project open, the Tool Palette provides access to all of the wizards and templates of the Object Repository. Some of these are shown in the following figure.



When you are using the code editor, the Tool Palette now includes these same options in addition to code snippets, and reusable pieces of code that you can drag into the code editor. Selecting objects from the Tool Palette has also been enhanced, greatly improving the speed with which you can build forms and applications. Simply click the Filter Current Items button in the Tool Palette toolbar, or press Ctrl-Alt-P, and start typing the name of the object you want to select. As you type, the characters you've entered so far appear in the Tool Palette title



You also have additional options for controlling the Tool Palette display. To see these options, select Tools I Options from the main menu. Tool Palette configuration options are available under the Tool Palette node of the Options dialog box.



Options

Button Size: [✓ Small] Medium Large

- [ ] Auto Collapse Categories
- [ ] Vertical Category Captions
- [ ] Lock Palette Reordering
- [x] Show Button Captions
- [x] Vertical Flow Layout

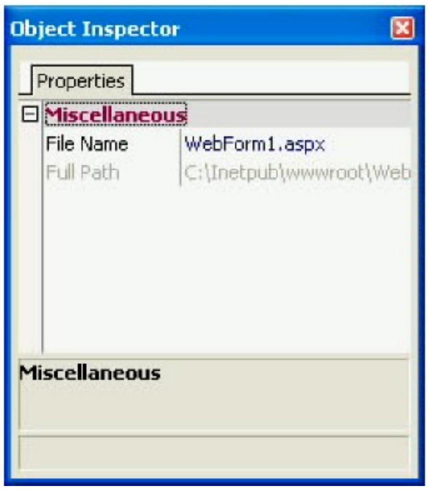
Features

- [x] Show Palette Wizards
- [x] Show Code Snippets
- [ ] Always Show Designer Items

Finally, the Tool Palette in Delphi 2006 now supports true drag-and-drop placement of components into the designer you are working with. Previously, component placement with VCL Forms could be better described as click-and-click, though that technique also works in Delphi 2006.

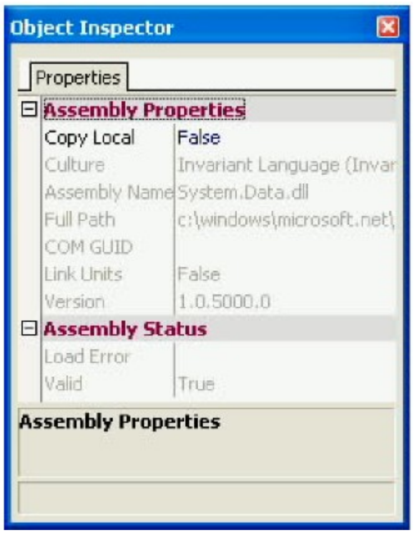
The Object Inspector

The Delphi 2006 Object Inspector, which you use to configure objects placed on your form at design time, has also been updated. Not only does the Object Inspector permit you to configure properties and events for the objects that you have placed into the designer, but you can also use it to control file names and get information about objects that you select in the Project Manager. For example, select a file in the Project Manager, such as an .aspx file in an ASP.NET Web application, and the file path and file name will appear in the Object Inspector, as shown in the following figure.



File Name	WebForm1.aspx
Full Path	C:\Inetpub\wwwroot\Web

The File Name property in the preceding figure is shown in an enabled font, indicating that you can edit the name of this file using the Object Inspector. Changing the file name here not only changes the name of the file displayed within the Project Manager, but since this file is a Delphi unit, the unit name changes as well. Of course, you can still rename a file the old fashioned way, by selecting File | Save As from the main menu. Other objects selectable within the Project Manager can also be viewed in the Object Inspector. For example, if you select one of the assemblies listed under the References node of a .NET project in the Project Manager, the Object Inspector displays details about that assembly, as shown in this next figure.



Assembly Properties

Copy Local	False
Culture	Invariant Language (Invar
Assembly Name	System.Data.dll
Full Path	c:\windows\microsoft.net\
COM GUID	
Link Units	False
Version	1.0.5000.0

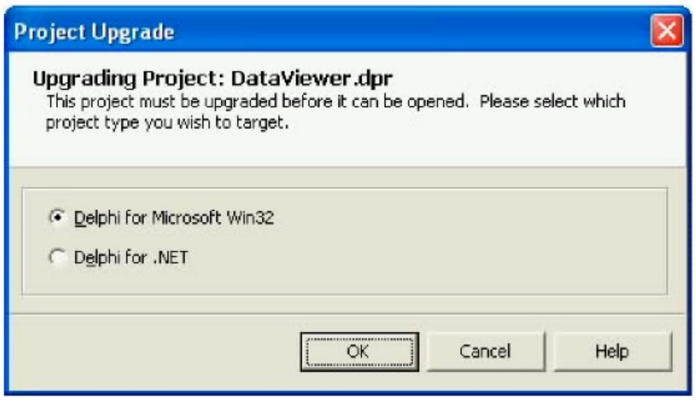
Assembly Status

Load Error	
Valid	True

Assembly Properties

The Upgrade Project Wizard

Because Delphi 2006 includes both Win32 and .NET compilers for the Delphi language, it can be used to create new Win32 applications as well as further the development of your existing Win32 projects that you created in Delphi 7 and earlier. You can also use Delphi 2006 to migrate your existing Win32 applications to VCL for .NET, the 100% .NET managed-code solution that maintains component and source code compatibility between Win32 and .NET development. The Upgrade Project Wizard is a special utility that runs the first time you open a Win32 application in Delphi 2006. Using this utility, you can choose to continue the current project as a Win32 application, or you can convert it to a .NET application.



Upgrading Project: DataViewer.dpr

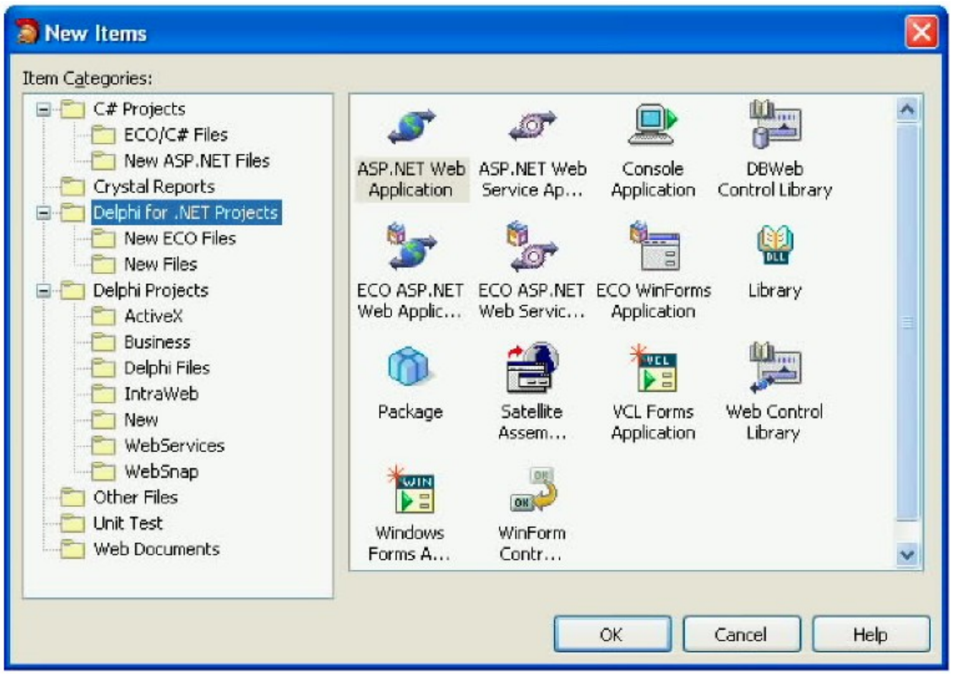
This project must be upgraded before it can be opened. Please select which project type you wish to target.

(● Delphi for Microsoft Win32) ○ Delphi for .NET

Once you made your choice using this wizard, Delphi 2006 will remember your selection. If you tell the Project Upgrade Wizard that you want to continue working with a Delphi project as a Win32 project, and at some later time decide to migrate it to VCL for .NET, simply delete your project's *.bdsproj file. After that, open the .dpr file in Delphi 2006. Once again, the Project Update Wizard will ask you to choose whether to continue working with the project as a Win32 project or to migrate it to VCL for .NET.

Delphi 2006 Wizards

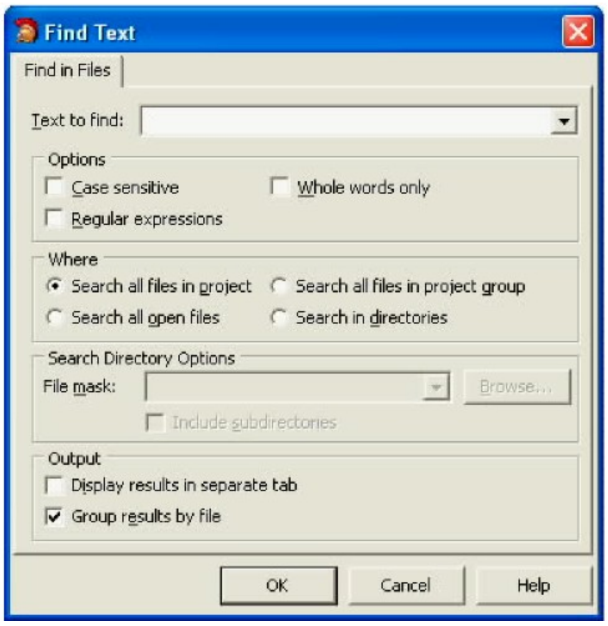
Wizards are small applets that help you to quickly create the projects, objects, and files that you use in Delphi 2006. For example, the ASP.NET Web Application Wizard creates for you the necessary web.config, global.asax, and initial .aspx file, and configures an IIS virtual directory into which these are placed, among other tasks. In short, wizards increase your productivity, getting you off to a fast start in the right direction. The following figure shows the Delphi 2006 object repository, displaying just a few of the many available wizards.



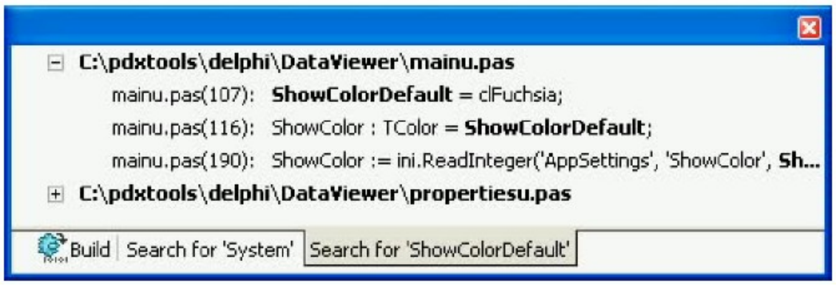
Delphi has always provided you with a rich collection of wizards that support almost every aspect of Windows development. For Win32 development, these include the Windows 2000 Logo Wizard, the DLL Wizard, the Automation Object Wizard, the Web Service Wizard, the IntraWeb Application Wizard, the Database Form Wizard, and the Thread Wizard. These are just some of the dozens of powerful wizards that are available. For Delphi for .NET and C#, you will find the ASP.NET Web Application Wizard, the Windows Form Application Wizard, ASP.NET Web Service Application Wizard, the Web Control Library Wizard, and many, many more. Delphi 2006 includes Wizards that were previously available in Delphi 7, Delphi 8 for the Microsoft .NET Framework, and C# for the Microsoft .NET Framework. In addition, Delphi 2006 includes a number of new and improved wizards accelerating your development efforts even more. These include the updated New Component Wizard, the new DB Web Control Library Wizard, the ECO ASP.NET Application Wizard, the ECO Web Service Application Wizard, and the Satellite Assembly Wizard, just to name a few.

Find in Files Enhancements

Delphi 2006 makes it even easier for you to search your project files by allowing you to group search results by file. Simply check the Group results by file check box in the Find Text dialog box.



The following figure shows what a grouped search result looks like. As you can see, each file in which the search string appears forms a base node in a tree view. Expanding the node for a given file lists the lines on which the located search string was found. You can then double-click a particular entry to go to that line of code in the code editor.



```
⊟ C:\pdxtools\delphi\DataViewer\mainu.pas
      mainu.pas(107):  ShowColorDefault = clFuchsia;
      mainu.pas(116):  ShowColor : TColor = ShowColorDefault;
      mainu.pas(190):  ShowColor := ini.ReadInteger('AppSettings', 'ShowColor', Sh...
⊞ C:\pdxtools\delphi\DataViewer\propertiesu.pas
```

Updated Support for International Characters

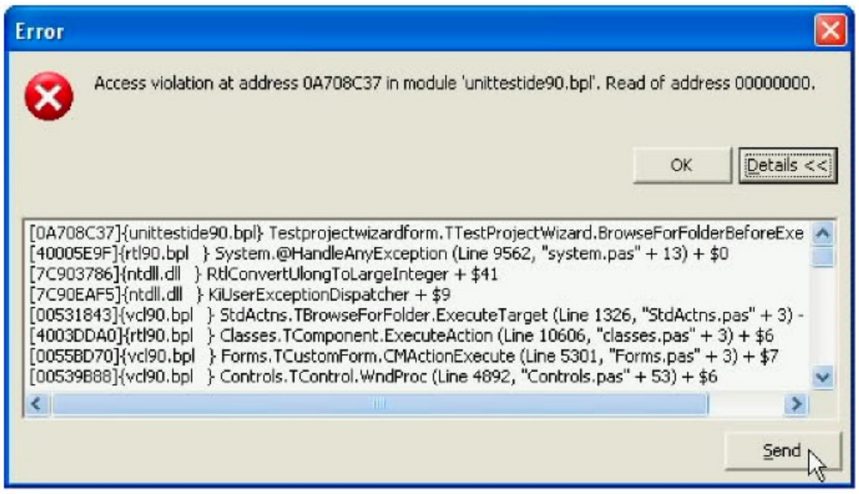
The Delphi 2006 IDE has been upgraded across the board to support UTF-8 characters in all of its wizards, windows, dialog boxes, and panes.

Message List Enhancements

Delphi 2006 uses the Message List pane to list compiler errors, warnings, and hints. You can now save the contents of the Message List pane by right clicking in the Message List pane and selecting either Copy, to copy selected messages to the Windows clipboard, or Save, to save the Message List contents to a file.

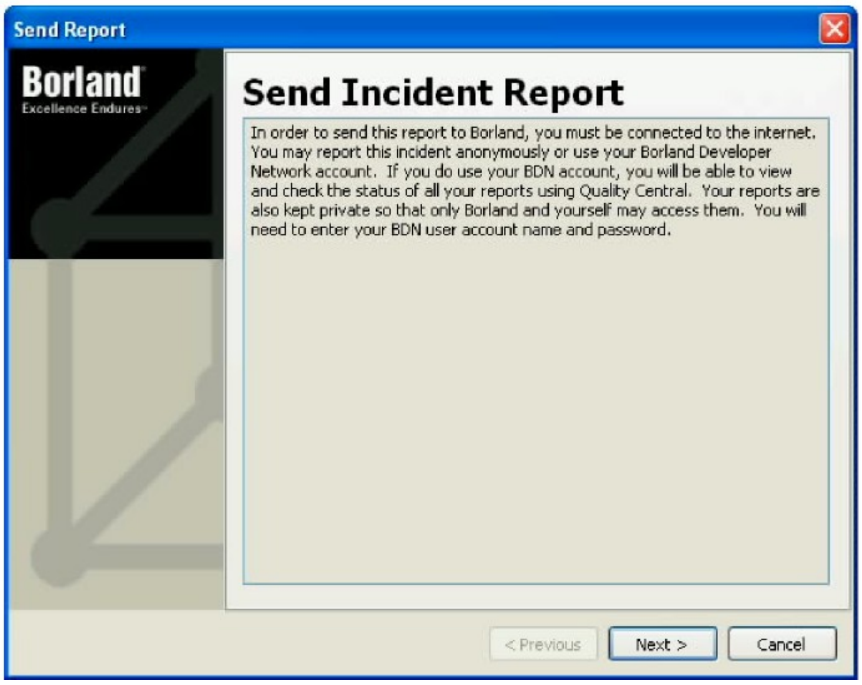
IDE Error Reporting

Borland's commitment to creating better software has lead to the development of a number of programs for reporting and fixing problems. One of the most recent of these is Quality Central, a Web-based application for submitting bug reports located at http://qc.borland.com. With Delphi 2006, Borland has embedded an error reporting system directly into the IDE. This feature is called IDE Error Reporting. If an exception is raised within the IDE, Delphi 2006 displays the Error dialog box. If you click the Details button, you see a detailed trace of the error.



```
Access violation at address 0A708C37 in module 'unittestide90.bpl'. Read of address 00000000.

[0A708C37]{unittestide90.bpl} Testprojectwizardform.TTestProjectWizard.BrowseForFolderBeforeExe
[40005E9F]{rtl90.bpl  } System.@HandleAnyException (Line 9562, "system.pas" + 13) + $0
[7C903786]{ntdll.dll  } RtlConvertUlongToLargeInteger + $41
[7C90EAF5]{ntdll.dll  } KiUserExceptionDispatcher + $9
[00531843]{vcl90.bpl  } StdActns.TBrowseForFolder.ExecuteTarget (Line 1326, "StdActns.pas" + 3) -
[4003DDA0]{rtl90.bpl  } Classes.TComponent.ExecuteAction (Line 10606, "classes.pas" + 3) + $6
[0055BD70]{vcl90.bpl  } Forms.TCustomForm.CMActionExecute (Line 5301, "Forms.pas" + 3) + $7
[00539B88]{vcl90.bpl  } Controls.TControl.WndProc (Line 4892, "Controls.pas" + 53) + $6
```

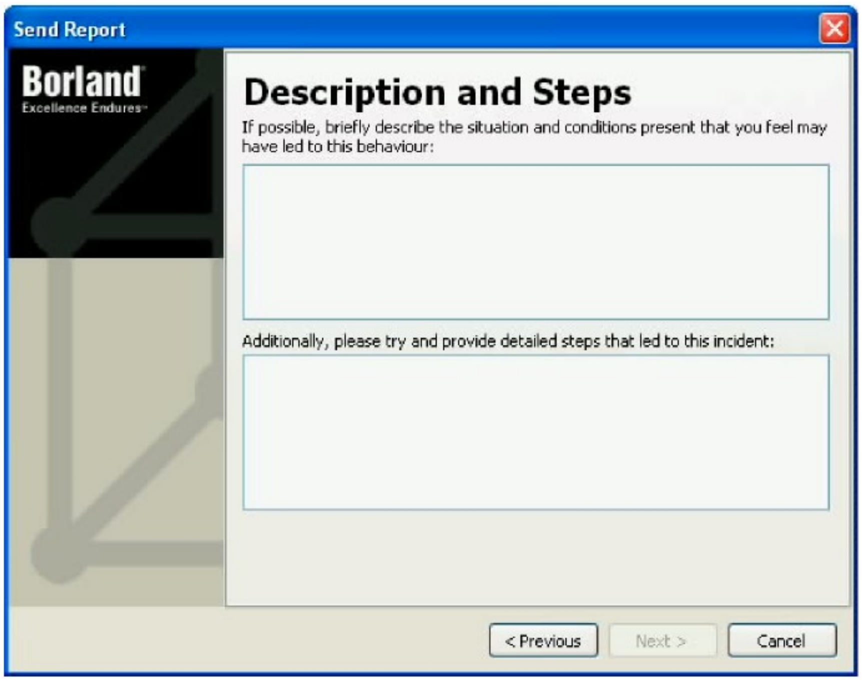
Clicking the Send button displays the Send Report dialog box.



Send Incident Report

In order to send this report to Borland, you must be connected to the internet. You may report this incident anonymously or use your Borland Developer Network account. If you do use your BDN account, you will be able to view and check the status of all your reports using Quality Central. Your reports are also kept private so that only Borland and yourself may access them. You will need to enter your BDN user account name and password.

Click the Next button to see the stack trace that will be submitted to Borland along with your error report. Click Next again to enter a description of what you were doing when the error occurred.



Description and Steps

If possible, briefly describe the situation and conditions present that you feel may have led to this behaviour:

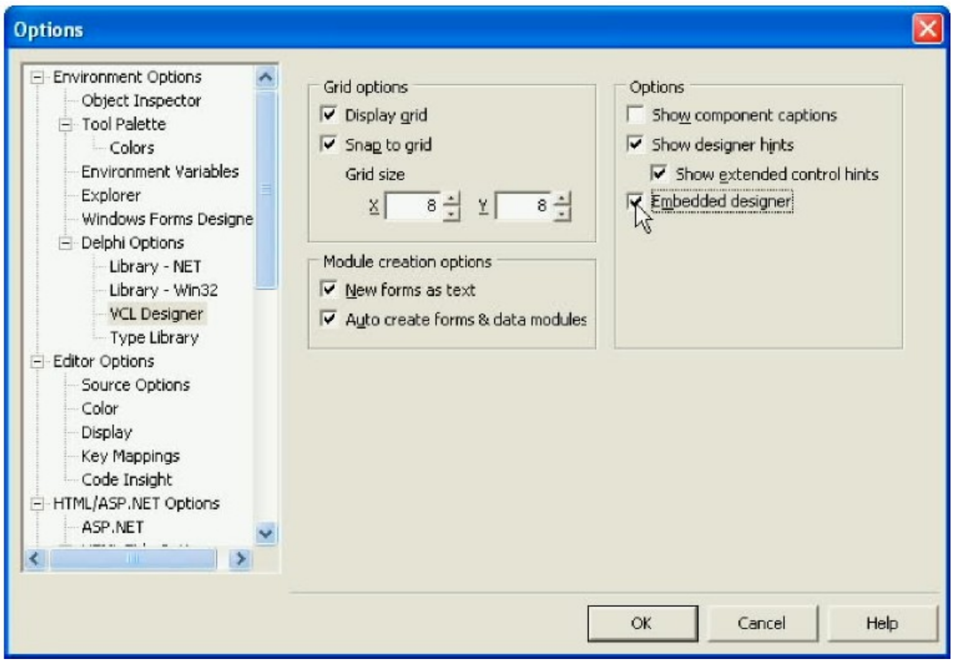
Additionally, please try and provide detailed steps that led to this incident:

Click Next once more to optionally provide your Borland Developer Network (BDN) logon email address and password. Submitting your report using your BDN account allows you to easily follow up on your report using Borland's Quality Central. If you want to submit the report anonymously, check the Anonymous Report check box. Click Next one final time to submit the error report.

The Improved VCL Form Designer

As a developer, you will greatly appreciate the enhancements found in Borland Developer Studio's integrated development environment. Indeed, more productivity improvements have been introduced in Borland Developer Studio 2006's IDE than in any previously updated Borland IDE. Regardless of which earlier version of Delphi, C#Builder, or C++Builder that you are currently using, the IDE updates you find in Borland Developer Studio 2006 alone justify the upgrade. If you are not currently using a Borland IDE, you will find that the collective features of Borland Developer Studio's development environment are enough to make you consider switching.

For Delphi development of VCL and VCL for .NET applications, Delphi 2006 provides you with a choice between using the .NET-style embedded designer or the classic floating designer. To enable the floating designer in Delphi 2006, select Tools | Options. Navigate to the VCL Designer node under Delphi Options, and uncheck the Embedded designer check box.



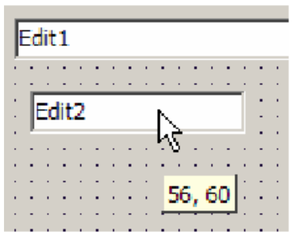
Grid options
[✓] Display grid
[✓] Snap to grid
Grid size X 8 Y 8

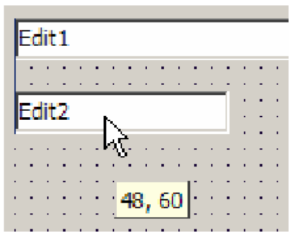
Module creation options
[✓] New forms as text
[✓] Auto create forms & data modules

Options
[ ] Show component captions
[✓] Show designer hints
[✓] Show extended control hints
[✓] Embedded designer

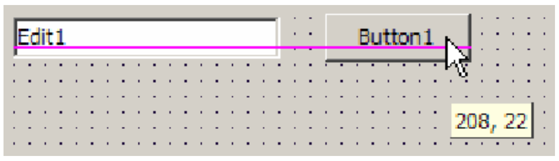
Component-based development involves a significant amount of time configuring components at design time, and when it comes to graphical user interfaces, this requires a first-class form designer. For VCL (visual component library) development, Borland Developer Studio 2006 provides you with rich new features for building your using interfaces. These include dynamic alignment guides, design guidelines, and the Form Positioner. Each of these features is described in the following sections.

Dynamic Alignment Guides





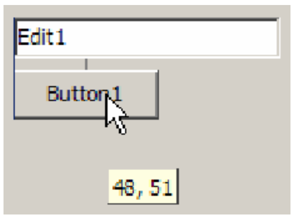
Dynamic alignment guides are visual cues displayed in the form editor as you position your visual controls. For most controls, these guides permit you to effortlessly align your controls with respect to their top, left, bottom, and right borders. Specifically, as you drag a control on a form, visual cues appear each time one of the control's edges comes into perfect alignment with an edge of another control on your form. Consider the following two figures. In the figure on the left, a TEdit (Edit2) is being dragged. The figure on the right shows how the alignment guide appears once the dragged TEdit comes into left alignment with the TEdit above it. The alignment guide disappears as soon as the dragged control is no longer aligned with another control. For those controls that contain textual elements, alignment guides permit you to align those controls with respect to their text. For example, the alignment guide in the following figure appears when the text elements of the TEdit and TButton come into alignment.



As a component developer, you can specify which elements of your visual controls participate in alignment guides. In addition, you can create new alignment points in your custom components, permitting those developers using your components to effortlessly align your components with respect to specific features they contain.

Design Guidelines

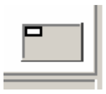
Like alignment guides, design guidelines are visual cues displayed in the VCL designer that permit you to better position your controls. However, instead of cueing vertical and horizontal location, design guidelines provide you with hints about the spacing of your controls. Design guidelines are associated with the Margins and Padding properties of visual controls and their containers. Using the Margins property, you can define the ideal minimum spacing between your control and other visual controls. Whenever the placement of your control matches the spacing suggested by the guidelines, a line appears. This line is visible in the following figure, in which both an alignment guide and a design guideline are shown. The design guideline is the short grey line that connects the middle-top of the TButton to the bottom of the TEdit. This line indicates that the current spacing of these two controls satisfies the design guidelines defined by the bottom margin of the TEdit and the top margin of the TButton.

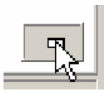


While the Margins property of a control specifies its ideal distance from a margin of another control, the Padding property defines the ideal spacing of controls that appear inside a container from their container's borders. Consequently, only container controls have a Padding property. A design guideline appears when a control's Margin and its container's Padding combine to suggest an ideal distance for the control from its container's corresponding border. For those components that support an Align property, Borland has introduced a new property called AlignWithMargins. When this property is set to True, the value of Align respects the Margin properties defined for the aligned objects, as well as the Padding properties of the container in which they are aligned. For example, if the Top Padding property of a TPanel is set to 2, and a TMemo with a Top Margin property of 5 and Align property of alTop appears within that panel, setting the TMemo's AlignWithMargin property to True will cause the top of the TMemo to remain exactly 7 (2 + 5) pixels from the top of the TPanel. While most developers will use alignment guides and design guidelines for design time placement of controls, there is another, powerful purpose for these properties. When used with the new TFlowPanel and TGridPanel containers, Margin and Padding properties are used at runtime to manage the dynamic placement of the controls contained within them. TFlowPanel and TGridPanel are described in detail in the section Visual Control Library Updates, found later in this reviewer's guide.

Form Positioner

There is a new visual cue on the VCL form designer that permits you to specify where on the screen a form will be displayed (so long as the form's Position property allows for default positioning). This cue is called the Form Positioner, and it appears in the lower-right corner of the VCL form designer. How you use of the Form Positioner is represented in the following two figures. The figure on the left shows the Form Positioner, which in this case indicates that the form, if displayed in the default position, will appear in the upper-left corner of the screen when made visible. If you want your form to appear somewhere else on the screen when displayed, drag the form position indicator within the Form Positioner to the desired location, as depicted in the figure on the right.





Visual Component Library Updates

The Visual Component Library (VCL) is the principle library for development in Win32 Delphi and C++Builder applications. It is also the compatibility library for Delphi for .NET applications. As in every other release of Delphi, Borland Developer Studio 2006 now includes several new and valuable components, as well as some enhancements that affect a large number of existing components. These are described in the following sections.

New VCL Components

For Delphi VCL-based development, the Tool Palette now includes a number of new controls for creating better using interfaces. These include a TButtonGroup, TCategoryButtons, and TDockTabSet. These components, which you can use in your Win32 and VCL for .NET applications, permit you to easily create interfaces similar to those used in Delphi 2006's Tool Palette and the Structure pane. As you have probably already guessed, these new components are the same ones that Borland engineers developed to build the Delphi 2006 IDE.

Other new components in the VCL are TGridPanel and TFlowPanel, specialized containers that enable dynamic runtime layout of the controls they contain. The TGridPanel component provides component layout similar to that available in an HTML table or in a Java GridBagLayout layout manager. Specifically, you configure a TGridPanel to have a specific number of rows and columns. When the size of the TGridPanel changes, the position, and potentially the size, of controls contained within its cells are adjusted. Importantly, the new positions (and sizes, if applicable) respect the Padding properties of the TGridPanel, as well as the Margin properties of the controls displayed in the cells.

The TFlowPanel is a specialized panel that positions the contained controls similar to how a word processor wraps the text in a document. (It is also similar to the Java FlowLayout and VerticalFlowLayout layout managers.) By default, this flow is right-to-left and top-to-bottom, though the TFlowPanel can be configured to use every other combination of flow.

Also new on the ToolPalette is the TTrayIcon component. This component permits you to easily create applications that appear as icons in the system tray of the task bar. Among the properties of the TTrayIcon component are those that permit you to select the glyph to display, a popup menu to associate with the icon, whether that icon is animated or not, balloon hints, and event handlers that respond to a user's interaction with the icon, to name a few.

In addition, VCL for .NET has been expanded to include even more Delphi VCL-compatible classes. These additional classes make it even easier than before to migrate your existing Win32 projects to the .NET framework. For a complete list of the new components in Delphi 2006, see "What's New in Delphi 2006" in the Delphi 2006 help.

Intellimouse® Support for VCL

The VCL now supports Intellimouse scrolling in any container class that supports mouse scrolling. In order to enable Intellimouse scrolling, simply add the IMouse unit to your project source uses clause, or any other uses clause for that matter. (Although for ease of maintenance, you are encouraged to use the project uses clause or the interface section uses clause of your main form.)
IMouse only needs to appear once in your project to affect all scrollable containers. When you enable Intellimouse support, scrollable VCL containers can be scrolled using the mouse-wheel, or equivalent interface, on your pointing device, given that your pointing device supports Intellimouse scrolling. If you want to enable Intellimouse scrolling in your custom components, ensure that the csPannable flag appears in your

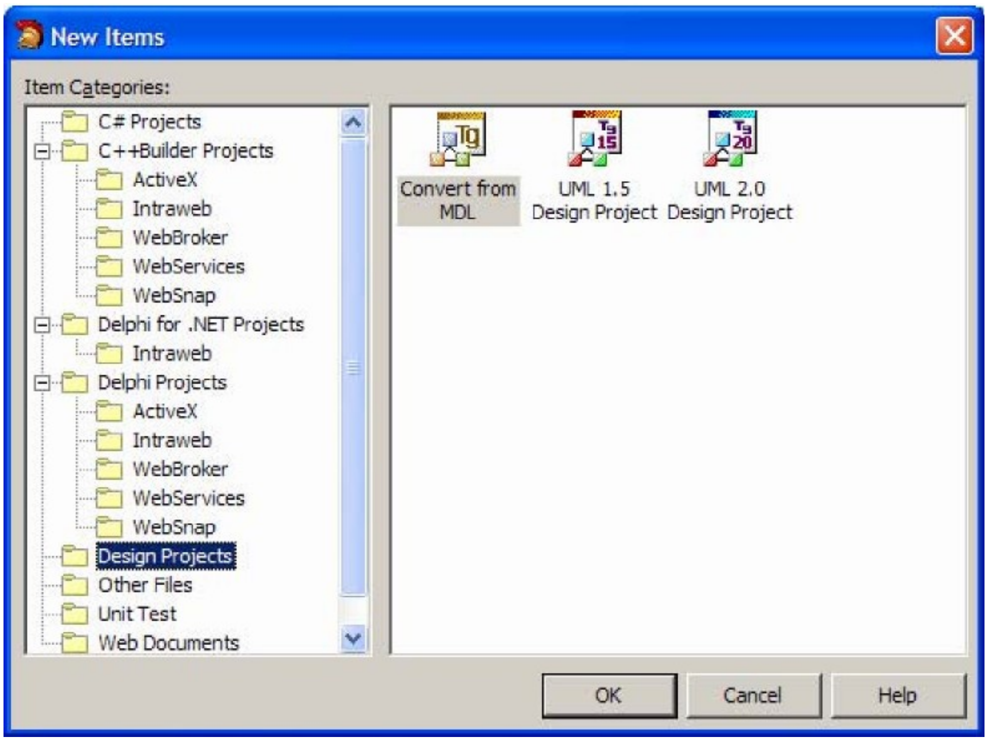
control's ControlStyle set property.

Other VCL Enhancements

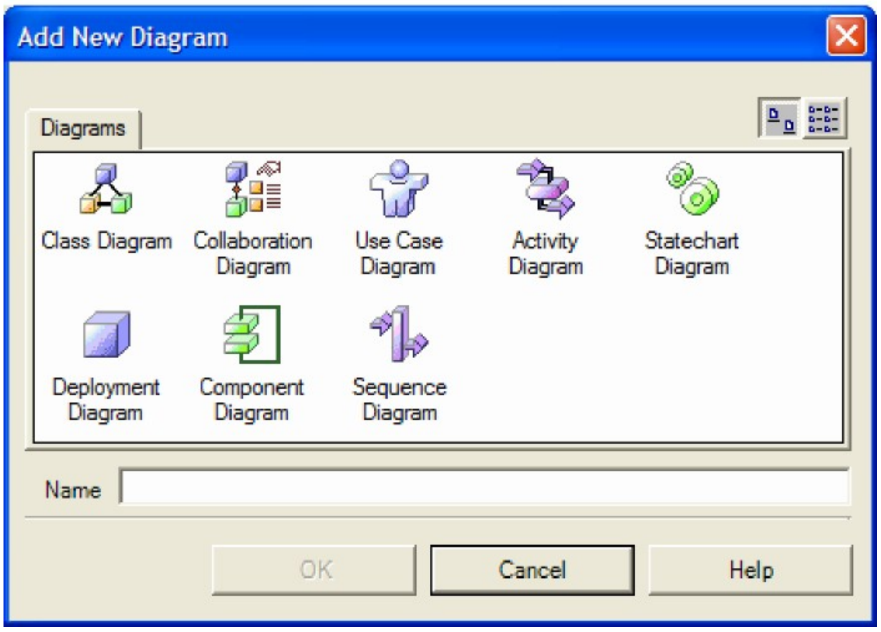
In addition to the preceding updates to the visual control library, Borland Developer Studio 2006 includes a new class, named TCustomTransparentControl, which you can descend from if you want to create transparent visual controls. In addition, some of the existing classes in the VCL have been enhanced. For example, Borland Developer Studio 2006 introduces accessibility enhancements to the TActionBar component, as well as color gradients to the TControlBar and TToolbar components. For a complete list of updated and enhanced components, "What's New in Borland Developer Studio 2006" from the Welcome page in the IDE.

Integrated modelling using Borland Together® technollgy

Borland Together is the visual tool that you use to create UML models for your applications. When used with ECO III, these models can be used to generate the objects that you can use directly in your applications. In response to customer feedback, Borland has re-engineered the Together engine to provide better performance and more efficient use of memory. This updated Together engine is seamlessly integrated into Borland Developer Studio 2006. You begin a design project in Borland Developer Studio 2006 by selecting the appropriate wizard from the Design Projects page of the Object Repository. As you can see in the following figure, both UML 1.5 and UML 2.0 projects are supported. This figure also shows the Convert from MDL wizard, which you use to import models from Rational® Rose.

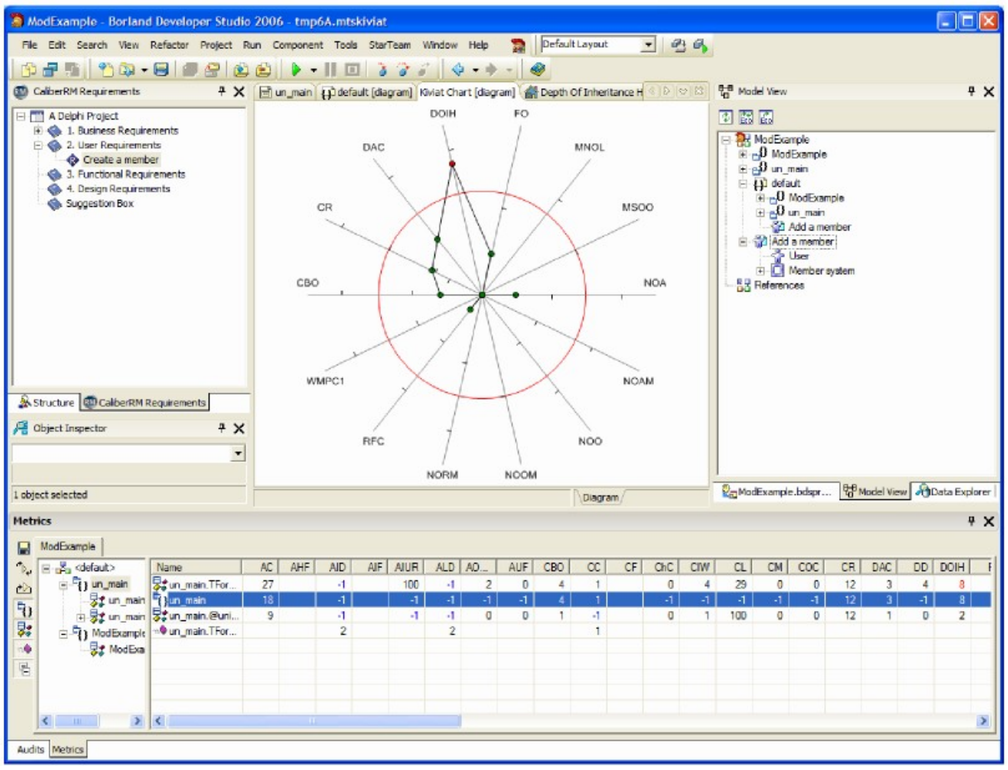


In addition to supporting UML design patterns, you can use Together to create many other industry standard diagrams, including use case diagrams, class diagrams, and sequence diagrams. The following figure shows the Add New Diagram dialog box, which shows the various new diagram wizards available in Together.



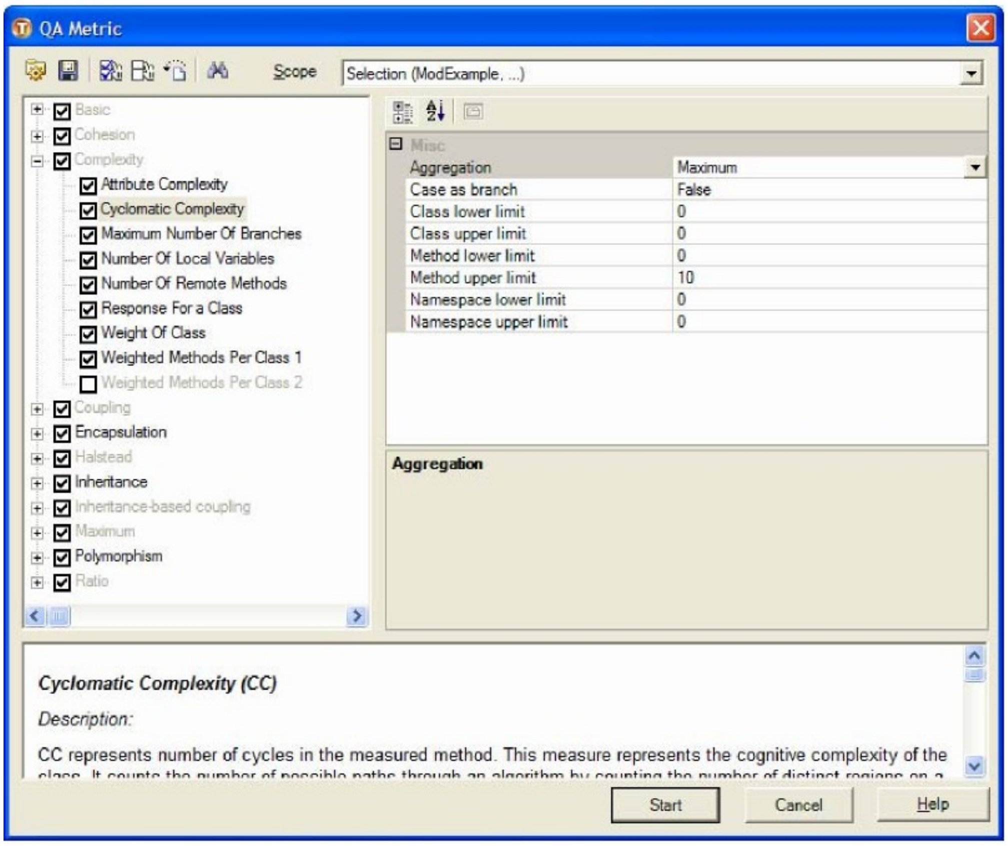
Audits and Metrics

More than simply a modeling tool, you can use Together to analyze your existing projects, generating metrics and audits that you can use to uncover potential problems with your applications. For example, the following figure shows a Together-generated Kiviat chart. A Kiviat chart depicts a best practices circle. Points that are within the inner circle are considered acceptable, and those outside denote areas that may deserve attention. For example, in the graph shown here, the depth of inheritance hierarchy (DOIH) appears outside the user-defined inner circle. This metric indicates that one or more of your classes are abnormally deep, with respect to the root class (Object or TObject).



Name	AC	AHF	AID	AIF	AIUR	ALD	AD...	AUF	CBO	CC	CF	ChC	CIW	CL	CM	COC	CR	DAC	DD	DOIH
un_main.TFor...	27		-1		100	-1	2	0	4	1		0	4	29	0	0	12	3	4	8
un_main	18		-1		-1	-1	-1	-1	4	1		-1	-1	-1	-1	-1	12	3	-1	8
un_main.@uni...	9		-1		-1	-1	0	0	1	-1		0	1	100	0	0	12	1	0	2
un_main.TFor...			2			2				1										

The inner-circle is user-defined because you set the limits for the metrics used by Together. The following figure shows the QA Metric dialog box where these metrics are configured.



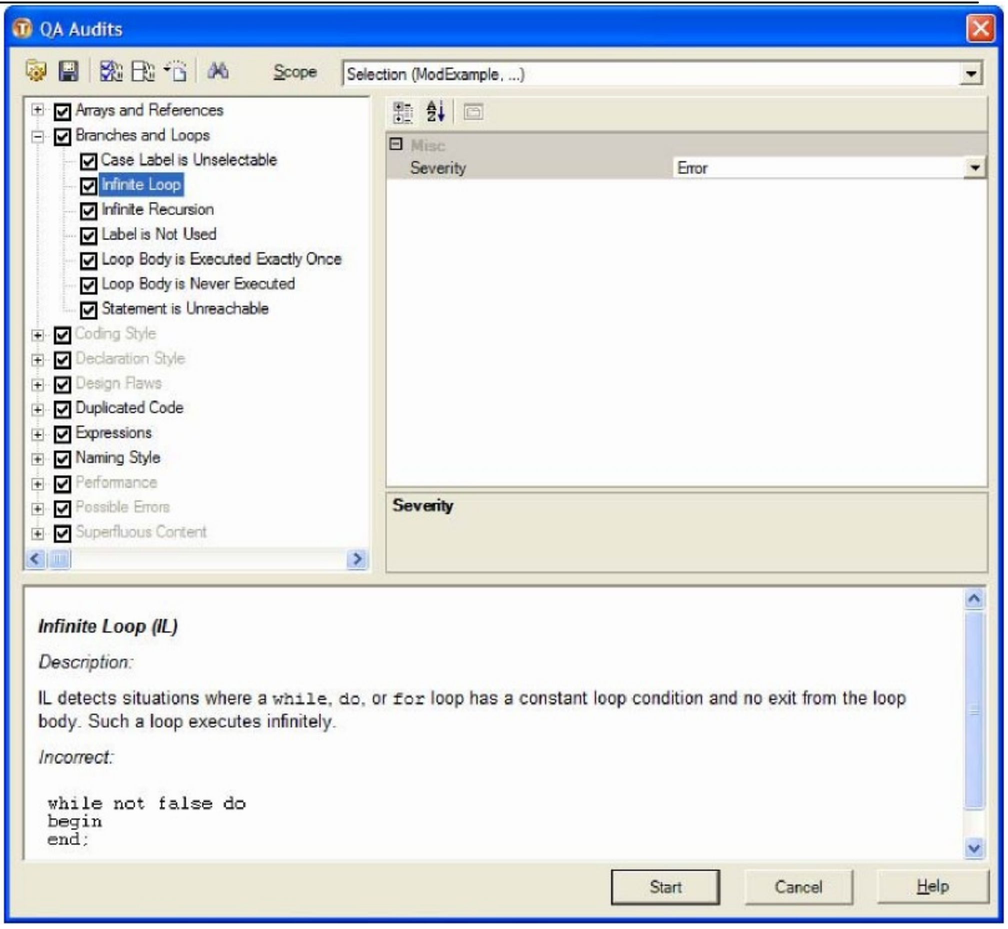
Aggregation	Maximum
Case as branch	False
Class lower limit	0
Class upper limit	0
Method lower limit	0
Method upper limit	10
Namespace lower limit	0
Namespace upper limit	0

Cyclomatic Complexity (CC)

Description:

CC represents number of cycles in the measured method. This measure represents the cognitive complexity of the

The use and configuration of audits is similar to metrics. The following figure shows the QA Audits dialog box, which contains the audits equivalent to the QA Metrics dialog box.



Severity: Error

Severity

Infinite Loop (IL)

Description:

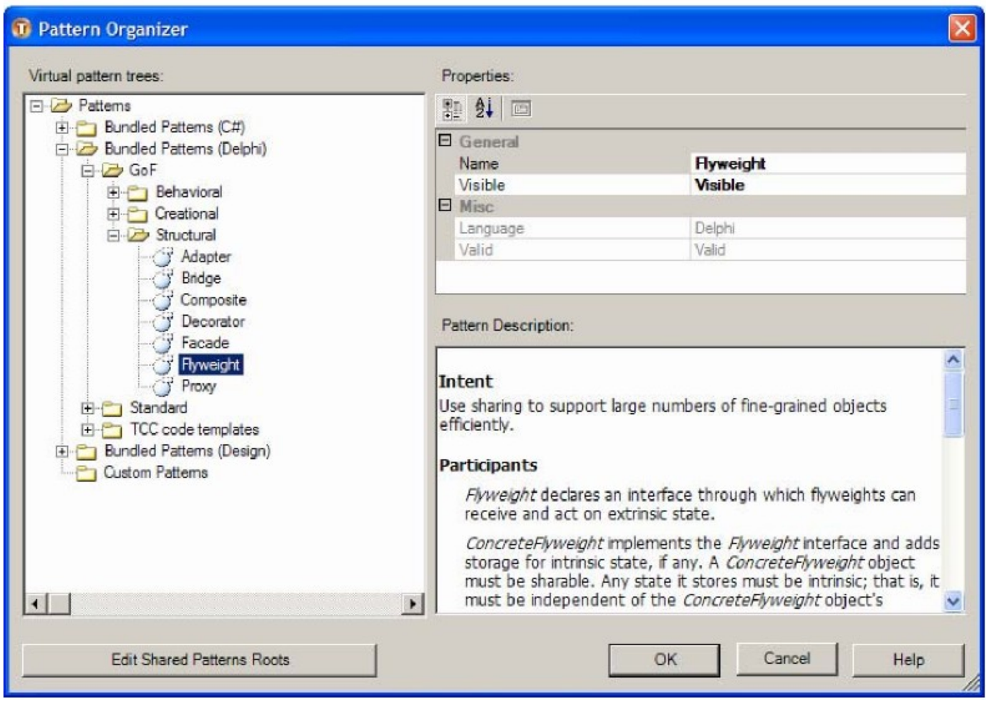
IL detects situations where a `while`, `do`, or `for` loop has a constant loop condition and no exit from the loop body. Such a loop executes infinitely.

Incorrect:

```
while not false do
begin
end;
```

Design Patterns

As mentioned earlier, Borland Together now provides you with high-level support for design patterns. Borland Developer Studio 2006 ships with a large number of industry-standard patterns, including those defined in the now classic book “Design Patterns: Elements of Reusable Object-Oriented Software “(1995, Addison-Wesley Professional Computing Series) by Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides (affectionately know as the Gang of Four, or GoF). In addition to the bundled patterns, you can also create and save custom design patterns. Both the bundled patterns and your custom patterns can be accessed using the Pattern Organizer, which is shown in the following figure.



Name	Flyweight
Visible	Visible
Language	Delphi
Valid	Valid

Intent
Use sharing to support large numbers of fine-grained objects efficiently.

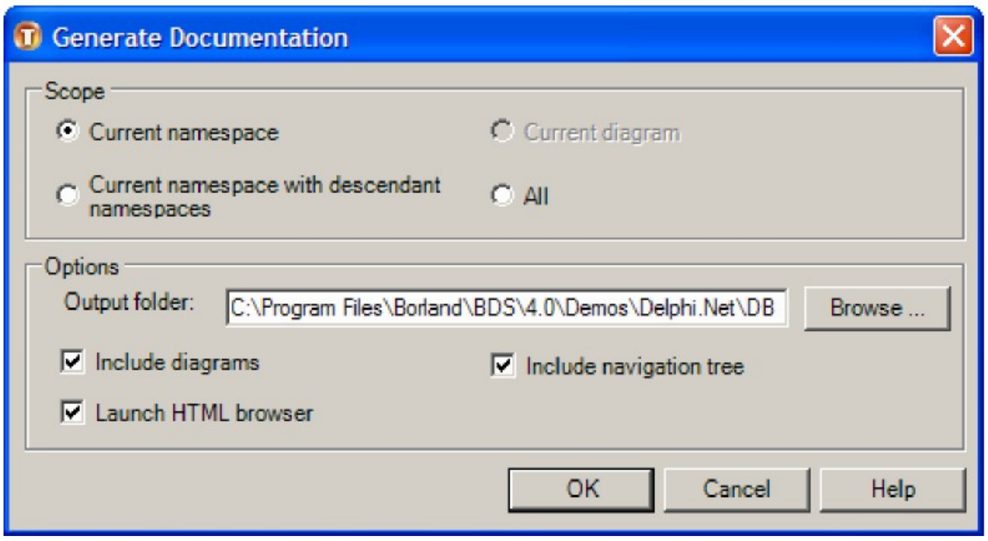
Participants

Flyweight declares an interface through which flyweights can receive and act on extrinsic state.

ConcreteFlyweight implements the *Flyweight* interface and adds storage for intrinsic state, if any. A *ConcreteFlyweight* object must be sharable. Any state it stores must be intrinsic; that is, it must be independent of the *ConcreteFlyweight* object's

Document Generation

One of the major benefits to you from the Together integration is the automatic generation of documentation. This documentation is generated as HTML, permitting you to view it in a browser, as well as permitting you to post it to your team's internal project Web site. To generate HTML documentation using Together, right-click the Model View pane from a Together-enabled project and select Generate Documentation. Use the Generate Documentation dialog box, shown in the following figure, to select the information that you want to include in the generated documentation.



Generate Documentation

Scope

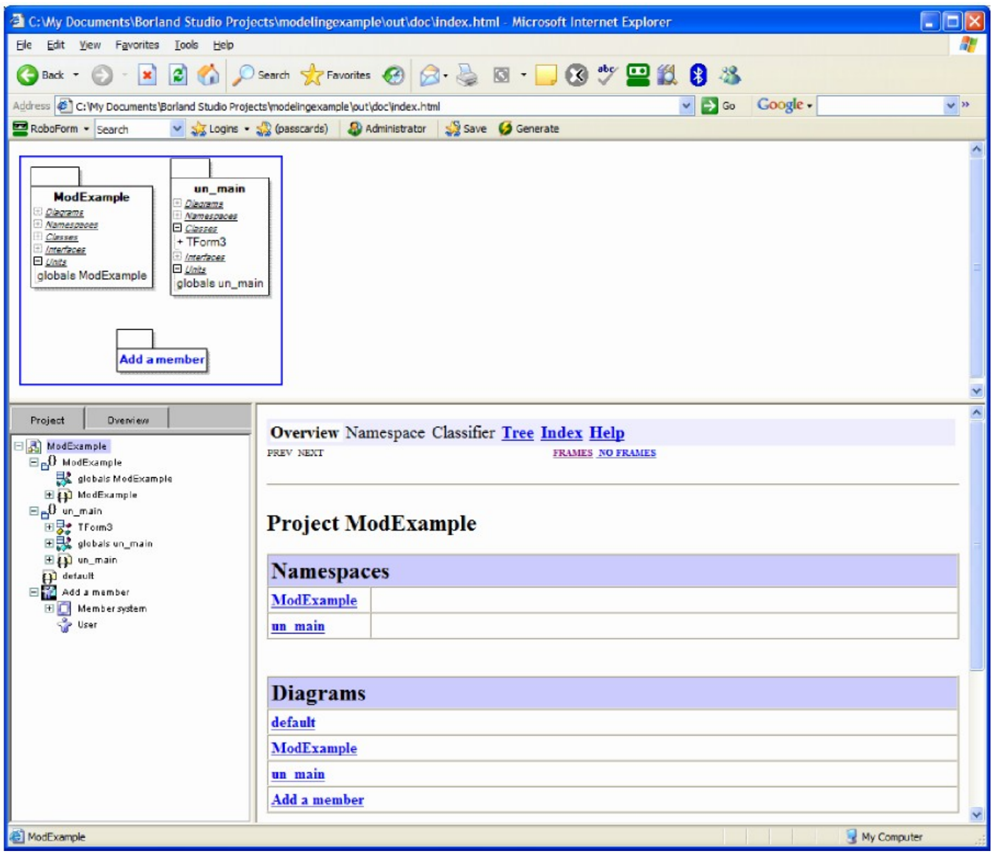
- (●) Current namespace
- ( ) Current diagram
- ( ) Current namespace with descendant namespaces
- ( ) All

Options

Output folder: C:\Program Files\Borland\BDS\4.0\Demos\Delphi.Net\DB

- [✓] Include diagrams
- [✓] Include navigation tree
- [✓] Launch HTML browser

The following figure shows a sample of Together-generated documentation being displayed in a browser.



C:\My Documents\Borland Studio Projects\modelingexample\out\doc\index.html - Microsoft Internet Explorer

Address C:\My Documents\Borland Studio Projects\modelingexample\out\doc\index.html

Overview Namespace Classifier Tree Index Help

PREV NEXT FRAMES NO FRAMES

Project ModExample

Namespaces
ModExample
un_main

Diagrams
default
ModExample
un_main
Add a member

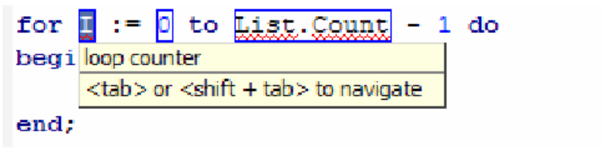
Editor Improvements

As a typical developer, there is only one feature of the IDE that you are likely to use more than the form designer, and that feature is the code editor. Fortunately, this is the area where you will enjoy the greatest number of updates and improvements in Borland Developer Studio 2006. The newly added features include live templates, block surround, and block completion. The enhancements included a greatly improved editor gutter, powerful new refactorings, enhanced method navigation, and updated open unit management features. Each of these new and improved editor features is described in the following sections.

Delphi 2006 continues Borland's heritage of providing developers with a world-class programming environment. To most developers, that also means a world-class code editor. And that's exactly what you get in Delphi 2006. In fact, for most developers, the updates that Borland has introduced to the code editor in Delphi 2006 will provide ample justification to upgrade from a previous version of Delphi or C#Builder. These features include refactoring support, SyncEdit, Error Insight, Help Insight, the History Manager, and much, much more. These new features are described in the following sections.

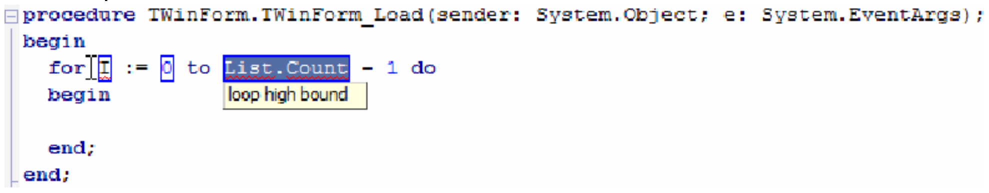
Live Templates

While code templates have been in Delphi since Delphi 4, they seem simple compared to the live templates available in Borland Developer Studio 2006. Live templates offer self-describing, intelligent code insertion, and interactive navigation to the variable parts of the template. For example, consider the inserted template shown in the following figure. This template was created by executing the forb template, which inserts a for loop with a begin. . end block.



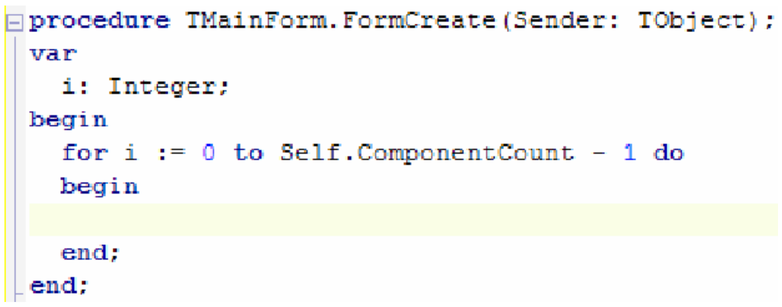
```
for I := 0 to List.Count - 1 do
begi
end;
```

Notice that initially the I (the loop iteration variable) is highlighted. This permits you to immediately edit the variable name, if you like. For example, many Delphi developers use a lowercase variable named i as their loop iterator (even though Delphi itself is case insensitive). You will also notice that the I, as well as the 0 and List.Count appear as bordered text. These bordered text areas represent the variable parts of the live template. After changing I to i, press Tab or Shift-Tab to navigate between the variable parts of the template. As you navigate, you will notice that the hints change to describe the role of the currently selected variable part. The following figure shows the same live template shown in the preceding figure after you have navigated to the third variable part.



```
procedure TWinForm.TWinForm_Load(sender: System.Object; e: System.EventArgs);
begin
  for I := 0 to List.Count - 1 do
  begin

  end;
end;
```

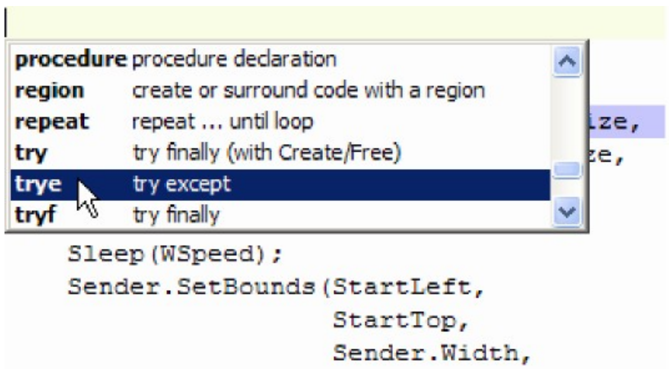
Many live templates, such as the forb template, also support intelligent code insertion, or scripting. With scripting, code executes when you complete the template, conditionally performing one or more tasks for you. In the case of the forb template, if the function in which you insert this template does not already contain a declaration for the loop iterator variable, one will be inserted for you. The following code segment depicts how a previously empty method might look after the execution of the forb live template script.



```
procedure TMainForm.FormCreate(Sender: TObject);
var
  i: Integer;
begin
  for i := 0 to Self.ComponentCount - 1 do
  begin

  end;
end;
```

Invoking Live Templates

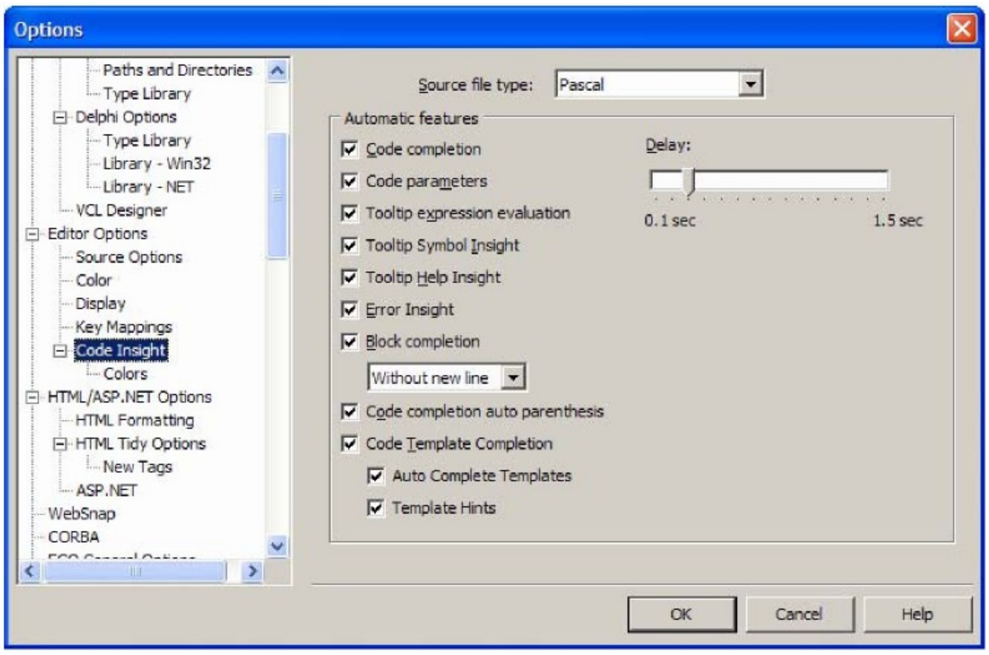
Borland Developer Studio 2006 provides you with a large number of standard live templates for each of its language personalities. These templates can be invoked either on demand or automatically. To invoke a live template on demand, position your cursor within the editor where you want to place the live template and press Ctrl-J. Borland Developer Studio responds by displaying the Insert Template menu, shown in the following figure. Select the template you want to insert from this menu and press Enter.



procedure	procedure declaration
region	create or surround code with a region
repeat	repeat ... until loop
try	try finally (with Create/Free)
trye	try except
tryf	try finally

```
                                                  ize,
                                                  ze,
    Sleep(WSpeed);
    Sender.SetBounds(StartLeft,
                     StartTop,
                     Sender.Width,
```

Similarly, you can begin typing the first few characters of the template shortcut name before pressing Ctrl-J. If only one template's shortcut name matches the characters you've typed, that template is selected and inserted, without the Insert Template menu being displayed. If two or more template shortcut names match what you've typed, the Insert Template menu appears, though only the matching template names are shown, in which case you then select which of the displayed templates you want and press Enter. Your live templates are invoked automatically when the Code Templates Completion checkbox is checked, as shown in the following figure. With automatic live template invocation, live templates are inserted when you type the name of a template shortcut and press Tab. For example, if you type forb and press Tab with code templates completion enabled, the forb live template is instantly inserted into your code at the position of your cursor.


Options

- Paths and Directories
- Type Library
- Delphi Options
  - Type Library
  - Library - Win32
  - Library - NET
- VCL Designer
- Editor Options
  - Source Options
  - Color
  - Display
  - Key Mappings
  - Code Insight
    - Colors
- HTML/ASP.NET Options
  - HTML Formatting
  - HTML Tidy Options
    - New Tags
  - ASP.NET
- WebSnap
- CORBA

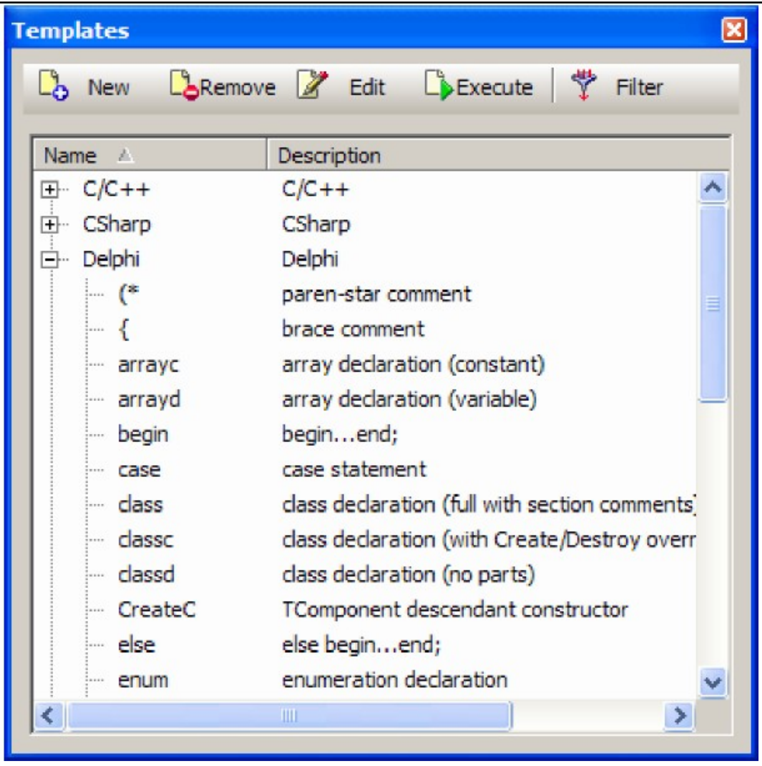
Source file type: Pascal

Automatic features

- [✓] Code completion
- [✓] Code parameters
- [✓] Tooltip expression evaluation
- [✓] Tooltip Symbol Insight
- [✓] Tooltip Help Insight
- [✓] Error Insight
- [✓] Block completion
  - Without new line
- [✓] Code completion auto parenthesis
- [✓] Code Template Completion
  - [✓] Auto Complete Templates
  - [✓] Template Hints

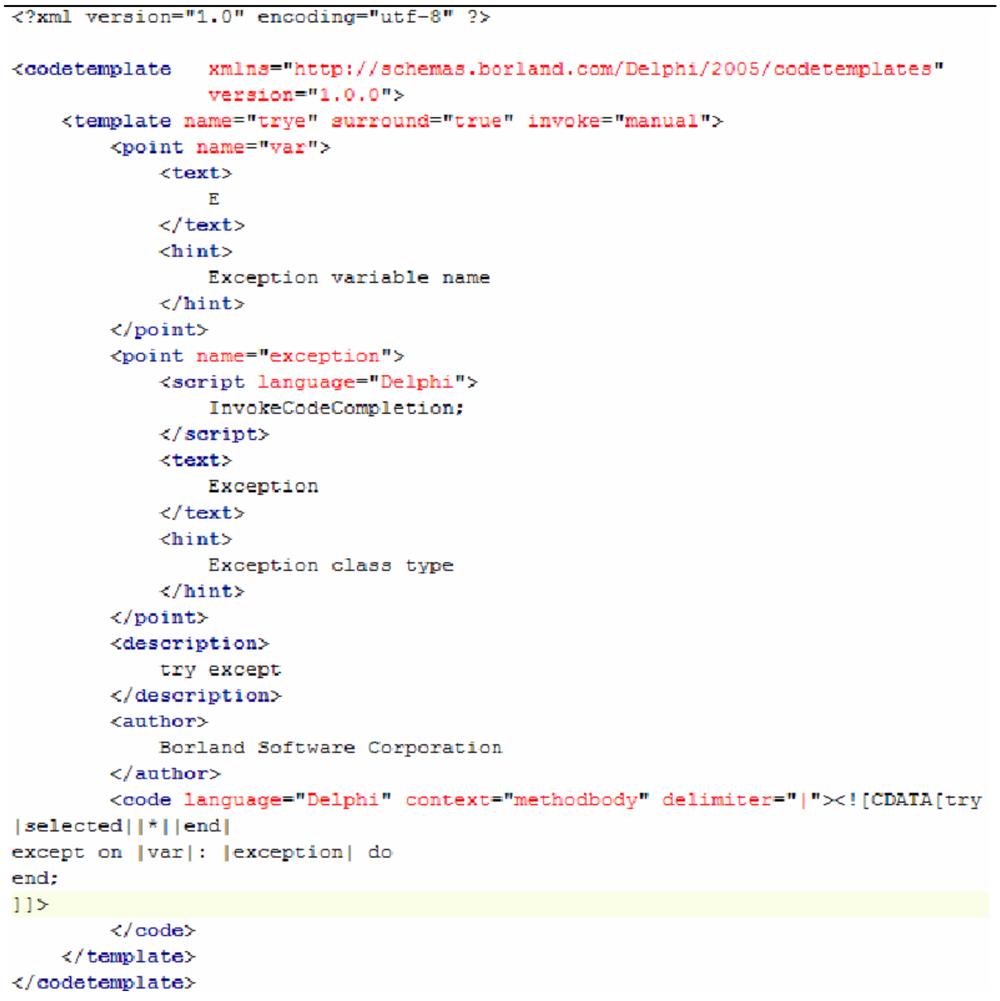
Delay: 0.1 sec – 1.5 sec

You can also invoke live templates from the Templates pane, shown in the following figure. To display the Templates pane, select View | Templates from the Borland Developer Studio main menu.



Name	Description
C/C++	C/C++
CSharp	CSharp
Delphi	Delphi
(*	paren-star comment
{	brace comment
arrayc	array declaration (constant)
arrayd	array declaration (variable)
begin	begin...end;
case	case statement
class	class declaration (full with section comments
classc	class declaration (with Create/Destroy overr
classd	class declaration (no parts)
CreateC	TComponent descendant constructor
else	else begin...end;
enum	enumeration declaration

Each of the standard templates that ship with Borland Developer Studio are defined by an XML file, which you can edit if you want to change the template's content or behavior. To edit one of the standard templates, select the template in the Template pane and then click Edit from the Template pane's toolbar. The following figure shows the trye (try ... except) live template in the code editor.

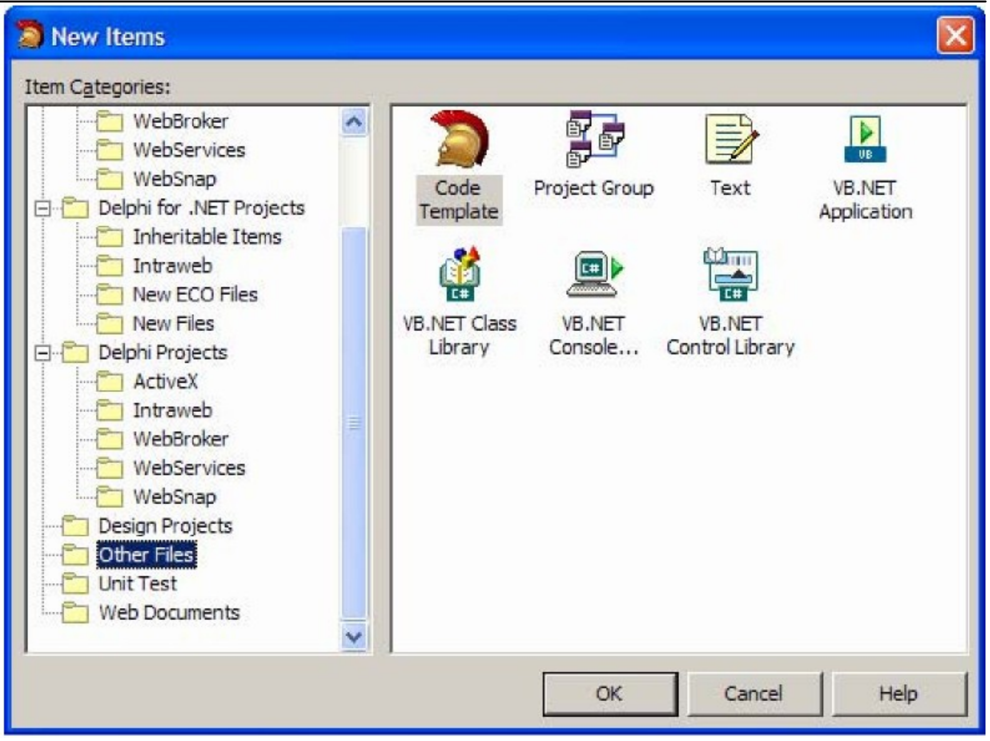


```
<?xml version="1.0" encoding="utf-8" ?>

<codetemplate   xmlns="http://schemas.borland.com/Delphi/2005/codetemplates"
                version="1.0.0">
    <template name="trye" surround="true" invoke="manual">
        <point name="var">
            <text>
                E
            </text>
            <hint>
                Exception variable name
            </hint>
        </point>
        <point name="exception">
            <script language="Delphi">
                InvokeCodeCompletion;
            </script>
            <text>
                Exception
            </text>
            <hint>
                Exception class type
            </hint>
        </point>
        <description>
            try except
        </description>
        <author>
            Borland Software Corporation
        </author>
        <code language="Delphi" context="methodbody" delimiter="|"><![CDATA[try
|selected||*||end|
except on |var|: |exception| do
end;
]]>
        </code>
    </template>
</codetemplate>
```

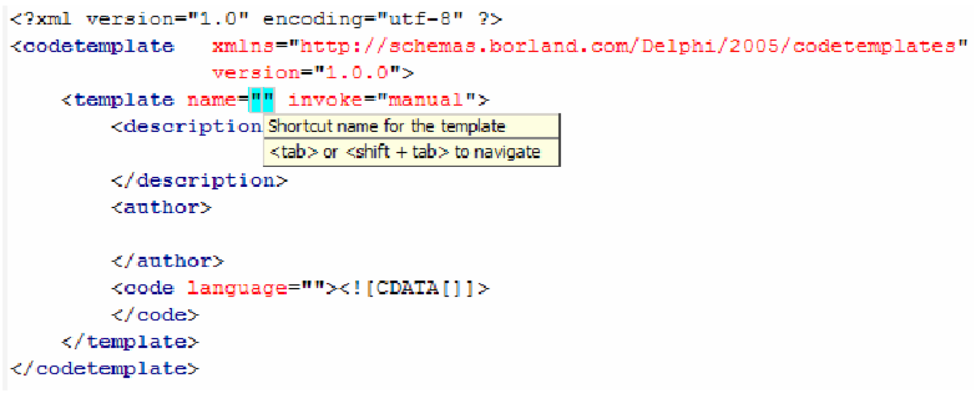
The trye live template is a good example to learn from, since it includes hint, text, and script elements. Furthermore, this is a surround template, which means that a code segment that you select before executing this template will result in the selected block of code being enclosed in the try block.

Custom Live Templates

While the standard live templates that ship with Borland Developer Studio 2006 are extremely useful, you will be delighted to discover that you can create your own, custom live templates. To create a custom live template, click New from the Templates pane's toolbar. Alternatively, you can select the Template Wizard from the Object Repository. To do this, select File | New | Other from the Borland Developer Studio main menu. Then, select the Template Wizard from the Other Files node of the Object Repository and select Ok.



The new template appears as an XML document in the code editor, as shown in the following figure. As you inspect this figure, you will notice that the new template is displaying hints. This is because the new custom live template is created by a live template. Not only does this reveal the incredible power of live templates, but the hints that are provide greatly simplify the process of creating your own new, live template.



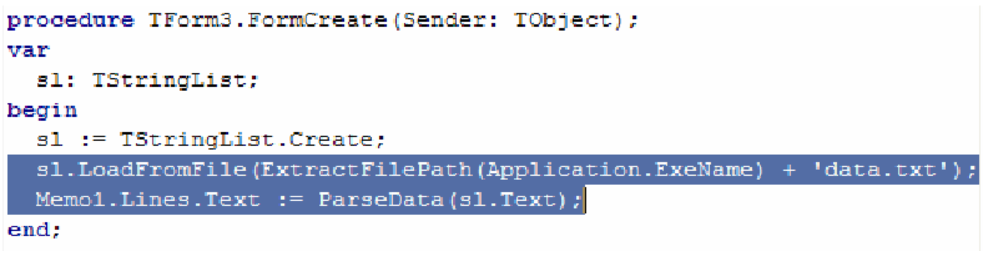
```
<?xml version="1.0" encoding="utf-8" ?>
<codetemplate   xmlns="http://schemas.borland.com/Delphi/2005/codetemplates"
                version="1.0.0">
    <template name="" invoke="manual">
        <description
        </description>
        <author>

        </author>
        <code language=""><![CDATA[]]>
        </code>
    </template>
</codetemplate>
```

Shortcut name for the template
<tab> or <shift + tab> to navigate

To complete your custom live template, fill in the various variable parts of the template, and add your own elements, as needed, to define the features you want in your template. In addition to the hints that the Live Template set up provides you with, you can read the Borland Developer Studio 2006 help for information on creating custom live templates. You should also examine the standard live templates that ship with Borland Developer Studio 2006 for insight and tips into writing your own, first-rate custom live templates.

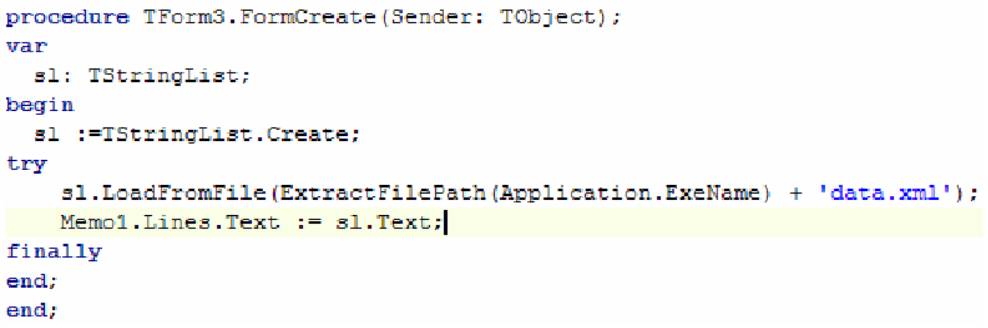
Surround Templates

Surround templates are a special subset of live templates that permit you to quickly enclose, or surround, a selected block of code. Examples of surround templates include forb, try, procedure, and comment live templates.
For example, imagine that you want to enclose the following code selection in a try . .. except block in a Delphi project.



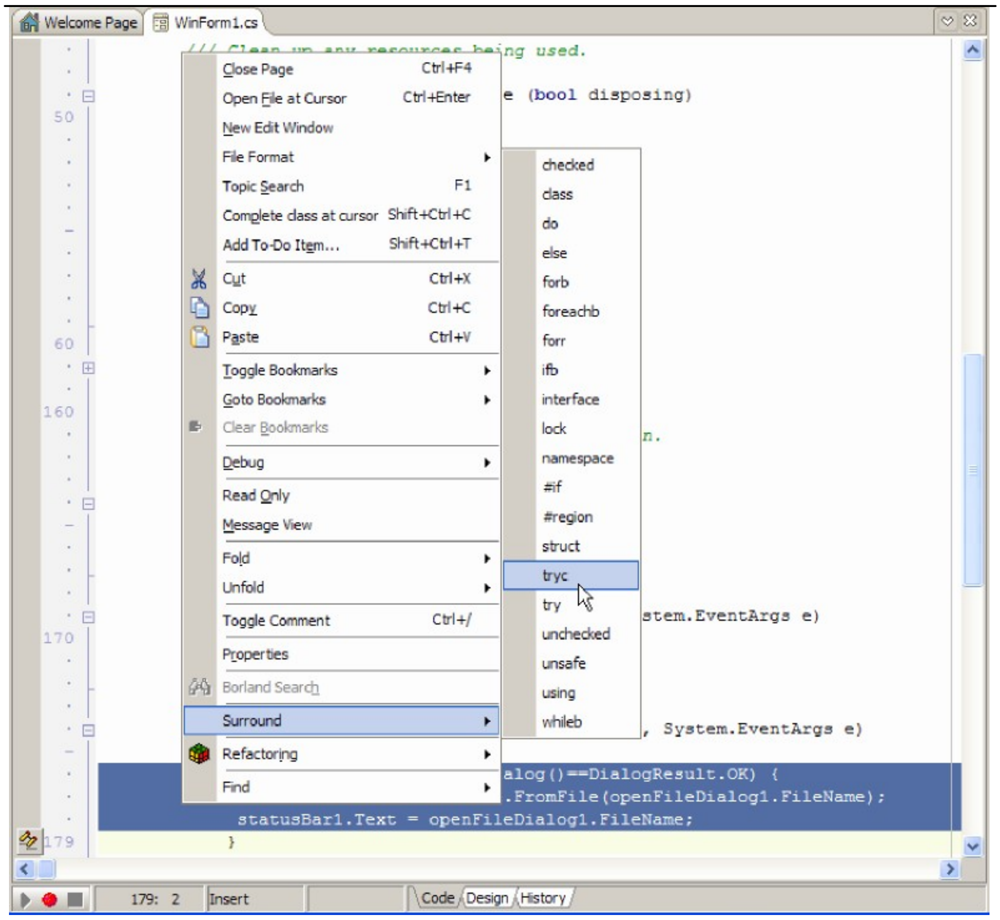
```
procedure TForm3.FormCreate(Sender: TObject);
var
  sl: TStringList;
begin
  sl := TStringList.Create;
  sl.LoadFromFile(ExtractFilePath(Application.ExeName) + 'data.txt');
  Memo1.Lines.Text := ParseData(sl.Text);
end;
```

With the desired code selected, select the tryf live template from the Templates pane, and then click Execute. As shown in the following figure, the selected code will be enclosed in the try block, and a finally block will be inserted after it.



```
procedure TForm3.FormCreate(Sender: TObject);
var
  sl: TStringList;
begin
  sl :=TStringList.Create;
try
    sl.LoadFromFile(ExtractFilePath(Application.ExeName) + 'data.xml');
    Memo1.Lines.Text := sl.Text;
finally
end;
end;
```

Notice also that the code block that you selected has been indented as well, making the resulting code more readable. You are now ready to insert a call to the Free method of the TStringList in the finally block, ensuring its destruction once you are through with it. You can also invoke a surround template from the Surround menu. Begin by selecting the block of code in the code editor that you want to surround. Next, right-click the selected block and select Surround. Borland Developer Studio responds by displaying a submenu of the surround live templates for your current code personality. For example, the following figure displays the Surround submenu for a selected block of C# code.

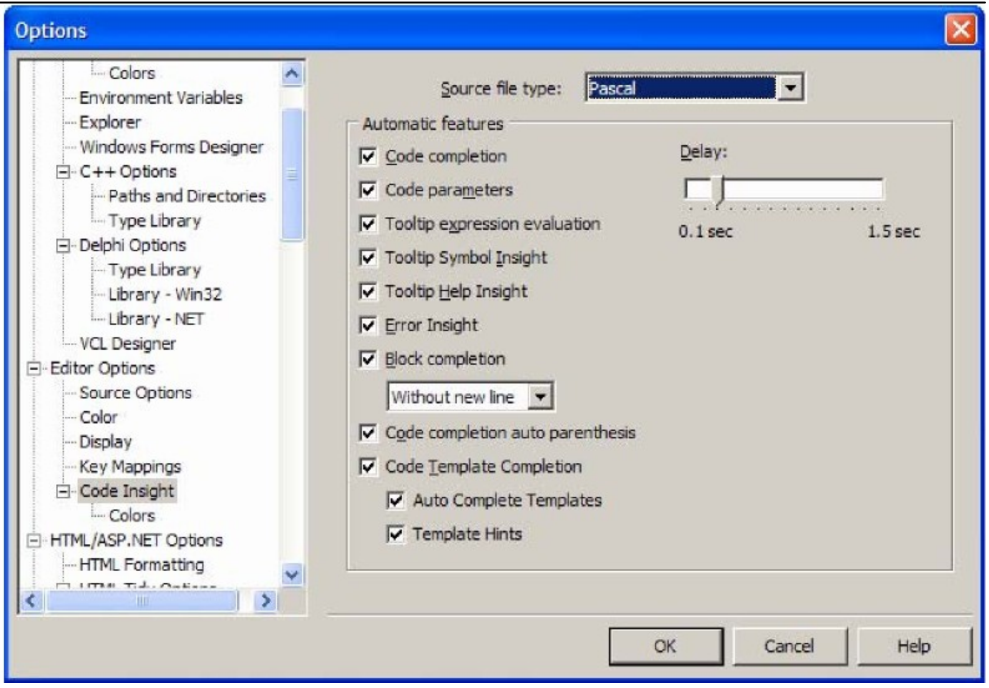


```
/// Clean up any resources being used.
e (bool disposing)
n.
stem.EventArgs e)
, System.EventArgs e)
alog()==DialogResult.OK) {
.FromFile(openFileDialog1.FileName);
  statusBar1.Text = openFileDialog1.FileName;
}
```

Select the desired surround template and press Enter to enclose the selected code. If you create your own custom surround templates, those will automatically appear in the Surround submenu. Specifically, the Surround submenu includes any custom live templates that you've created for the current language personality where the surround attribute is set to true in the <template> element.

Block Completion

Block completion is the latest addition to Borland's CodeInsightTM feature. With block completion, Borland Developer Studio 2006 will automatically complete a code block that you initiate. For example, if you type begin using the Delphi personality, and then press the Enter key, Borland Developer Studio 2006 will complete the block by adding the end keyword. Similarly, type try and press Enter, and the finally and end keywords are inserted. As mentioned, this feature is sensitive to the language of your current IDE personality. For example, when writing code in C#, entering the open curly brace ({) and pressing Enter causes the close curly brace to be inserted. Like other aspects of CodeInsight, you configure block completion using the CodeInsight page of the Options dialog box shown in the following figure.



Source file type: Pascal

Automatic features

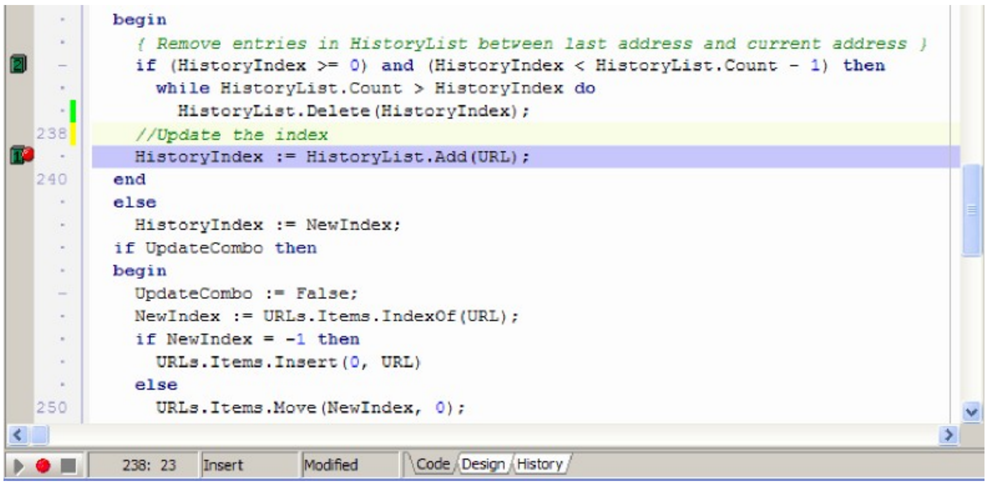
- [✓] Code completion
- [✓] Code parameters
- [✓] Tooltip expression evaluation
- [✓] Tooltip Symbol Insight
- [✓] Tooltip Help Insight
- [✓] Error Insight
- [✓] Block completion
  - Without new line
- [✓] Code completion auto parenthesis
- [✓] Code Template Completion
  - [✓] Auto Complete Templates
  - [✓] Template Hints

Delay: 0.1 sec — 1.5 sec

Notice that you can selectively turn off or on each of the CodeInsight features, as well as configure the default delay before automatic features execute.

The Enhanced Code Editor Gutter

The code editor gutter, that area to the left of the code editor where break points, line numbers, and bookmarks appear, has received a significant facelift in Borland Developer Studio 2006. In short, the code editor gutter is now less cluttered, while providing significant improvements in information content. Many of these improvements can be seen in the following figure.



```
        begin
          { Remove entries in HistoryList between last address and current address }
          if (HistoryIndex >= 0) and (HistoryIndex < HistoryList.Count - 1) then
            while HistoryList.Count > HistoryIndex do
              HistoryList.Delete(HistoryIndex);
238       //Update the index
          HistoryIndex := HistoryList.Add(URL);
240     end
        else
          HistoryIndex := NewIndex;
        if UpdateCombo then
        begin
          UpdateCombo := False;
          NewIndex := URLs.Items.IndexOf(URL);
          if NewIndex = -1 then
            URLs.Items.Insert(0, URL)
          else
250         URLs.Items.Move(NewIndex, 0);
```

One of the first things that you will notice is that line numbers are not displayed on every line. Instead, line numbers are displayed every 10 lines, by default. In addition, a line number always appears on the current line, which is line 238 in the preceding figure. This feature is configurable, in case you really do want to see line number on every line of code. In addition, bookmarks and breakpoints now occupy less space than in previous editors, once again resulting in a less cluttered look. This can be seen in the preceding figure, where a line number and a breakpoint both appear on line number 239. But the feature that you are likely to find most exciting is the change bar, which is a visual cue that appears on the right side of the left gutter. By default, lines that have been changed since you opened the current source file are marked with visual indicators. Those lines that have been modified, but not yet saved, are marked in yellow (by default). Lines that have been changed, and already saved, are marked in green.

As is the case with all color coding available in Borland Developer Studio 2006, you can customize the actual colors used to identify these changes.

Code Navigation Enhancement

Code navigation is a feature of Delphi 2006 that permits you to easily move between sections of your code. For example, by pressing Ctrl-Shift-UpArrow (or Control-Shift-DownArrow), you can move effortlessly between a method name in a Delphi class declaration to the associated implementation of that method.
Delphi 2006 introduces a small but valuable enhancement to code navigation in Delphi code, allowing you to move between your interface and implementation section uses clauses, as well as between your unit's initialization and finalization sections, using Ctrl-Shift-UpArrow. Code navigation is not necessary in C# projects, as the associated modules in C# do not have a two-part structure, as is the case with Delphi units.

Toggling Code to/from Comments

Delphi 2006 introduces a new feature that permits you to quickly comment and uncomment a selected code block. To comment one or more consecutive lines of code, select the code in the code editor, right-click, and then select Toggle Comment from the displayed context menu (or press Ctrl-/). When you do this, Delphi 2006 places the single line comment characters (//) at the start of each of the lines in the selected block. To uncomment one or more consecutive lines, select those lines and press Ctrl-/, or right-click and select Toggle Comment. Delphi 2006 will respond by removing the single-line comment characters from each selected line in the block. The single-line comment characters do not have to be in the first column of the code editor for Delphi 2006 to remove them.

Persistent Bookmarks

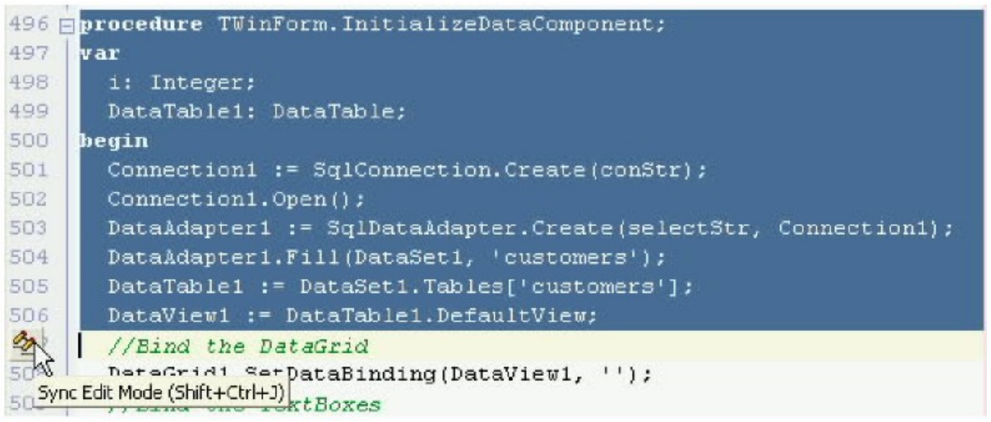
Bookmarks are special tags that you place within a source file to enhance your navigation within that file. You place a bookmark by pressing Ctrl-Shift, followed by a single digit, from 0 to 9. Once placed, the bookmark appears in the left gutter of the code editor using a glyph that represents the digit. Once a bookmark has been placed, you can quickly navigate to that bookmark within the code editor by pressing the Ctrl key followed by the digit used to place the bookmark. For example, if you have previously placed a bookmark using Ctrl-Shift-1, and subsequently navigate to a different area of your code file, you can instantly return to the bookmarked line in your source code by pressing Ctrl-1. Delphi 2006 now supports persistent bookmarks. If persistent bookmarks are enabled, a placed bookmark will remain in the source code until you specifically remove it. This means that you can place a bookmark in one editing session, and that bookmark will still be there the next time you open that source code file in Delphi 2006. In order to enable persistent bookmarks, check the Project desktop check box under the Autosave options group on the Environment Options page of the Options dialog box. You display the Options dialog box by selecting Tools | Options from the main menu.

Improved Method Navigation

Delphi 6 introduced module navigation, which permits you to quickly move between the methods in your class type declarations and their corresponding implementations by pressing Ctrl-Shift-Up arrow and Ctrl-Shift-Down arrow. In Borland Developer Studio 2006, method navigation has received another significant update with the introduction of method hopping. Method hopping permits you to quickly move between the methods of a unit by pressing CtrlAlt-Down arrow (to navigate to the next method in the implementation section) and Ctrl-AltUp arrow (to move to the previous method). Similarly, pressing Ctrl-Alt-Home moves to you the first method implementation in the source file, and Ctrl-Alt-End moves you to the last. In its normal mode, method hopping moves between method implementations without respect to class. For example, pressing Ctrl-Alt-Down arrow will move you to the next method (if one exists), even if it is a method for a class different than the one your cursor is currently pointing to. You can control this behavior by using class lock, which you invoke by pressing Ctrl-Q"L (Ctrl-Q, followed by L). When enabled, method hopping with class lock limits your navigation to the methods of the current class. For example, with method locking enabled, pressing Ctrl-Alt-Home moves your cursor to the first method of the current class in your source file, while Ctrl-Alt-Down arrow moves you to the next method in the current class. If you are on the last method implementation in the current class and you press Ctrl-Alt-Down arrow, your cursor does not move. While in class lock mode, pressing Ctrl-Q"L again disables class lock. While module navigation doesn't apply to C# and C++ code, method hopping does. Class lock, however, is currently only available in the Delphi and C# editors.

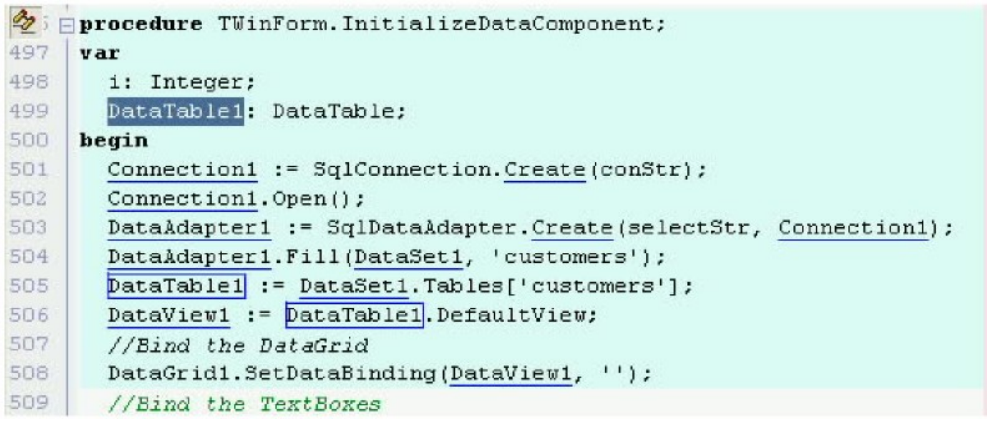
SyncEdit

SyncEdit, introduced in Delphi 2005, provides support similar to symbol renaming refactoring. Unlike symbol renaming, however, SyncEdit performs localized renaming of symbols for a selected code block only. This is a powerful capability and one of the most popular new features with developers. SyncEdit becomes available anytime you select a code block that includes at least two instances of the same symbol name. For example, consider the following figure, which depicts a selected code block that includes more than one reference to a local variable named DataTable1 (as well as DataSet1, DataAdapter1, Connection 1, and the Create method).



```
procedure TWinForm.InitializeDataComponent;
var
  i: Integer;
  DataTable1: DataTable;
begin
  Connection1 := SqlConnection.Create(conStr);
  Connection1.Open();
  DataAdapter1 := SqlDataAdapter.Create(selectStr, Connection1);
  DataAdapter1.Fill(DataSet1, 'customers');
  DataTable1 := DataSet1.Tables['customers'];
  DataView1 := DataTable1.DefaultView;
  //Bind the DataGrid
  DataGrid1.SetDataBinding(DataView1, '');
  //Bind the TextBoxes
```

The SyncEdit icon appears in the left gutter of the editor window, indicating that synchronized changes to the selected code block are available. To enter the SyncEdit mode, you either click this icon or press Shift-Ctrl-J. Once you enter the SyncEdit mode, the duplicate symbols are identified, and the symbol selected for synchronized editing appears highlighted (with the duplicates being displayed enclosed in boxes). If you want to edit a symbol other than the one selected by default, press the Tab key until the symbol you want to SyncEdit is selected.



```
procedure TWinForm.InitializeDataComponent;
var
  i: Integer;
  DataTable1: DataTable;
begin
  Connection1 := SqlConnection.Create(conStr);
  Connection1.Open();
  DataAdapter1 := SqlDataAdapter.Create(selectStr, Connection1);
  DataAdapter1.Fill(DataSet1, 'customers');
  DataTable1 := DataSet1.Tables['customers'];
  DataView1 := DataTable1.DefaultView;
  //Bind the DataGrid
  DataGrid1.SetDataBinding(DataView1, '');
  //Bind the TextBoxes
```

After selecting the symbol to edit, begin typing. The name of the selected symbol, and its duplicates, are updated as you type. The following figure shows the name of the DataTable being changed to CustTable. (The edit is being performed on the first instance of DataTable1, which appeared in the var declaration of this method.)

SyncEdit is a great productivity tool when you are writing functions, procedures, and methods, in that this feature is so easy to use. There are, however, important differences between SyncEdit and symbol renaming refactorings. SyncEdit is lexical, so it works with comment lines as well as compilable code, unlike symbol renaming refactorings, which work only on actual symbol references. Likewise, symbol renaming refactoring extends its reach into descendant classes, as well as to resource files (such as VCL and VCL for .NET form files). SyncEdit only applies to the currently selected code block.

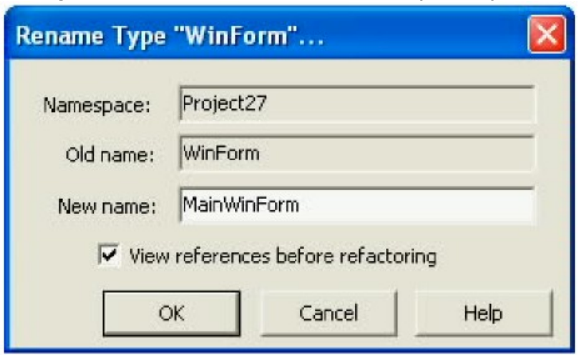
Refactoring

Refactorings were introduced in Delphi 2005, and Borland Developer Studio 2006 introduces a number of important new refactorings. Refactoring is the process of updating existing code to improve its readability, maintainability, and efficiency, without changing the essential behavior of the software. Common refactorings include providing more expressive names for variables, replacing duplicate code segments with a call to a common function that performs the same task, and replacing literal values with constants or resource references. Delphi 2006 includes a number of impressive refactorings. These include symbol renaming, method extraction, variable and field declarations, and resource refactorings.

Symbol Renaming

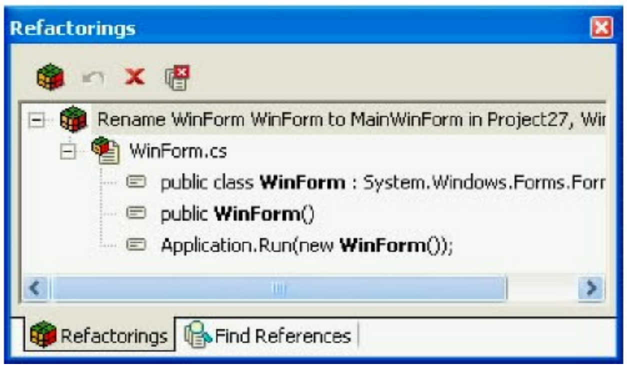
Symbol renaming allows you to change all instances of a symbol's name throughout your project. Unlike a search-and-replace feature, symbol renaming respects the context in which the symbol name appears. Symbols that can be renamed using this refactoring include class and interface names, properties, methods, functions and procedures, as well as variables and constants.

To perform a symbol renaming refactoring, select the symbol whose name you want to change in the code editor, and select Refactoring | Rename. Use the Rename dialog box to define a new name for your symbol.



Namespace: Project27
Old name: WinForm
New name: MainWinForm
[✓] View references before refactoring

If you leave the View references before refactoring option checked, Delphi 2006 displays the Refactorings pane, which lists all of the instances within your code where the change will be applied.



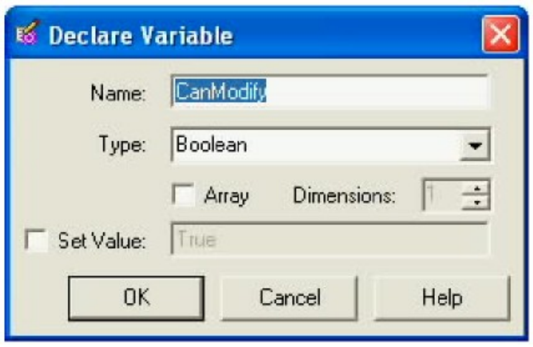
Rename WinForm WinForm to MainWinForm in Project27, Wir
WinForm.cs
public class **WinForm** : System.Windows.Forms.Forr
public **WinForm**()
Application.Run(new **WinForm**());

Click the Refactor button on the Refactoring pane toolbar to apply the changes.

Alternatively, you can choose to remove one or more of the refactorings before applying them, or even cancel the refactoring altogether.

Variable and Field Declarations

The Declare Variable and Declare Field options on the Refactor menu permit you to quickly create a local variable or member field declaration. This option is available with Delphi code, but not with C# projects. (This feature is not needed in C# since fields can appear almost anywhere within a C# class. By comparison, in Delphi, variables must appear in a var block, and member fields must appear in a type block.)
To insert a local variable or member field, select the symbol name that you created in the code editor, and select Refactor | Declare Variable or Refactor | Declare Field (or press Ctrl-Shift-V or Ctrl-Shift-D, respectively). If you select Declare Variable, the Declare Variable dialog box is shown.


Declare Variable

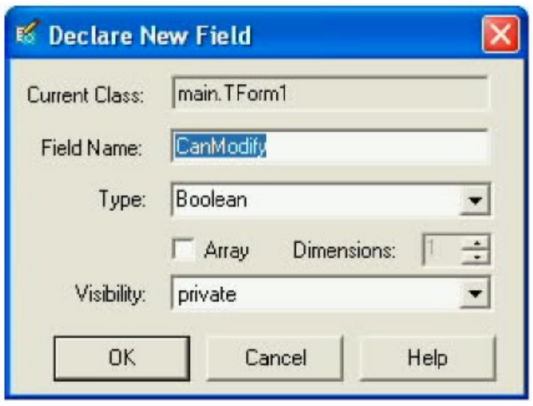
Name: CanModify

Type: Boolean

Array: unchecked — Dimensions: 1

Set Value: unchecked — True

You use the Declare Variable dialog box to change the variable name, set its data type, make the variable an array type with a specific dimension, or to initialize the newly created variable to a specific value. Click OK to create the local variable, and initialize its value (if you chose that option).If you select Declare Field, the Declare New Field dialog box is displayed. You use this dialog box to set the name and data type of the new field, to declare it as an array of a given dimension, and to define its visibility within the associated class. When you click OK, the newly named field is created in the selected section of the class within whose method the symbol is located.


Declare New Field

Current Class: main.TForm1

Field Name: CanModify

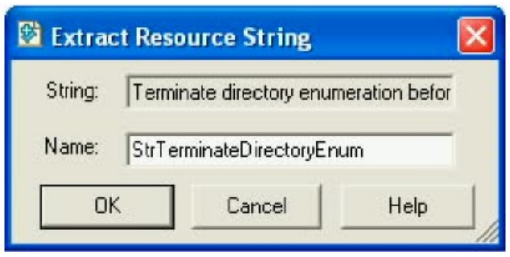
Type: Boolean

Array: unchecked — Dimensions: 1

Visibility: private

Resource Refactoring

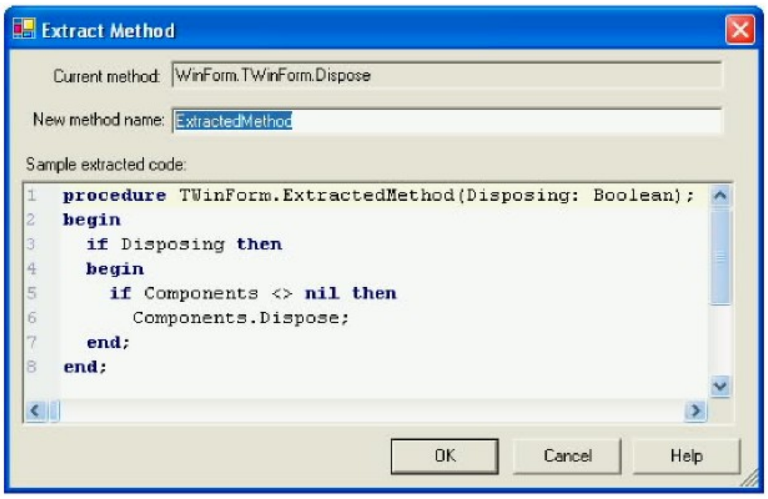
Resource refactorings are used in Delphi code to convert string literals into resourcestring block entries, replacing the original literal with the resource string symbol. (There is no resourcestring block in the C# language.) Using resource strings instead of string literals is particularly valuable when a specific string literal is used repeatedly, as well as when you need to create localized (language and/or culture specific) versions of your application. After placing your cursor within a string literal in the Delphi code editor, select Refactor I Extract Resource String. Use the Extract Resource String dialog box to modify the string and to change the default name for the resource symbol. When you click OK, the string literal is replaced with the resource symbol, and the named symbol is inserted into a resourcestring block in the associated unit's interface section.



String: Terminate directory enumeration befor
Name: StrTerminateDirectoryEnum

Extract Method Refactoring

Most developers think of method extraction when they think of refactoring. Method extraction involves converting one or more lines of code into an independent method call, replacing those lines with an invocation of the extracted method. In Delphi 2006, method extraction refactoring is only available for the Delphi language. Method extraction is particularly useful when the same or similar lines of code appear repeatedly in your project. By extracting those lines to a separate method, replacing each of the repeated instances with an invocation of the method, you greatly enhance your code's maintainability by creating a single location where changes to those lines of code, if desired, need to be implemented. To perform a method extraction refactoring, select the lines of code that you want to extract to a method, and then select Refactor | Extract Method. Use the Extract Method dialog box to define a name for the new method, as well as to examine the code that will be placed inside of this new method.



Current method: WinForm.TWinForm.Dispose

New method name: ExtractedMethod

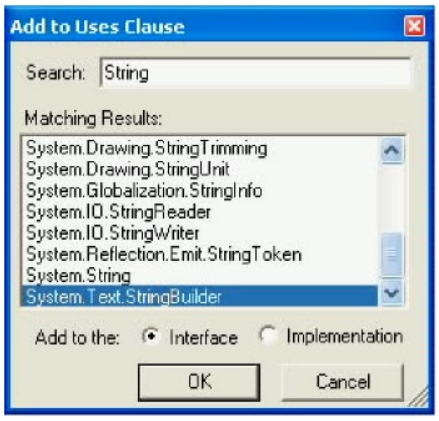
Sample extracted code:

```
procedure TWinForm.ExtractedMethod(Disposing: Boolean);
begin
  if Disposing then
  begin
    if Components <> nil then
      Components.Dispose;
  end;
end;
```

Delphi 2006's extract method refactoring is intelligent, with respect to variables, properties, and objects referenced within the code being extracted. For example, since the code in the preceding figure includes a reference to the Disposing property of the method's class, the value of this property is passed by value to the refactored code. By comparison, if the code actually made a change to the value of a variable that needs to be passed into the refactored method, the associated parameter would be passed by reference (using the var keyword).

Import Namespace (C#) and Find Unit (Delphi)

Although not exactly a refactoring, the Import Namespace and Find Unit options under the Refactor menu permit you to quickly locate and import the namespace associated with a particular symbol. If you are coding in C#, you select Refactor | Import Namespace. Delphi developers select Refactor | Find Unit. After selecting this option from the Refactor menu, the displayed dialog box lists all of the classes in all of the namespaces available to the environment you are working in. For example, if you are creating a Delphi .NET Windows Forms application, the namespaces of the FCL and RTL for .NET (the .NET version of the Delphi runtime library) are available. Delphi VCL for .NET developers will find the VCL for .NET namespaces as well.



Search: String

Matching Results:

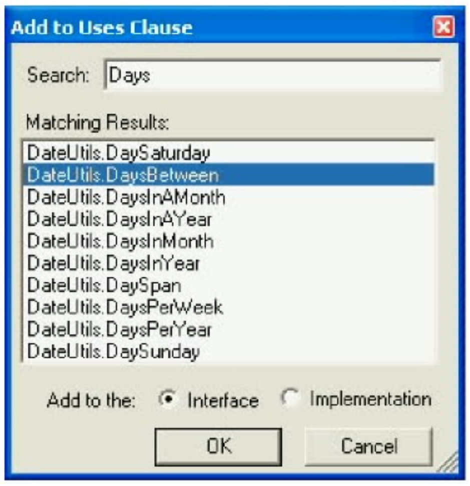
```
System.Drawing.StringTrimming
System.Drawing.StringUnit
System.Globalization.StringInfo
System.IO.StringReader
System.IO.StringWriter
System.Reflection.Emit.StringToken
System.String
System.Text.StringBuilder
```

Add to the: [✓ Interface] Implementation

By comparison, if you are creating a Delphi Win32 application, the various units of the

VCL and RTL are listed. Type the name of the class that you want to be able to access in the Search field. As you type, the Matching Results list is filtered to include only those classes, and their associated namespaces, whose names match what you've typed so far.

Select the name of the class whose namespace you want and click OK. If you are working in Delphi, you can also specify whether the namespace will be added to your interface or implementation section uses clause.



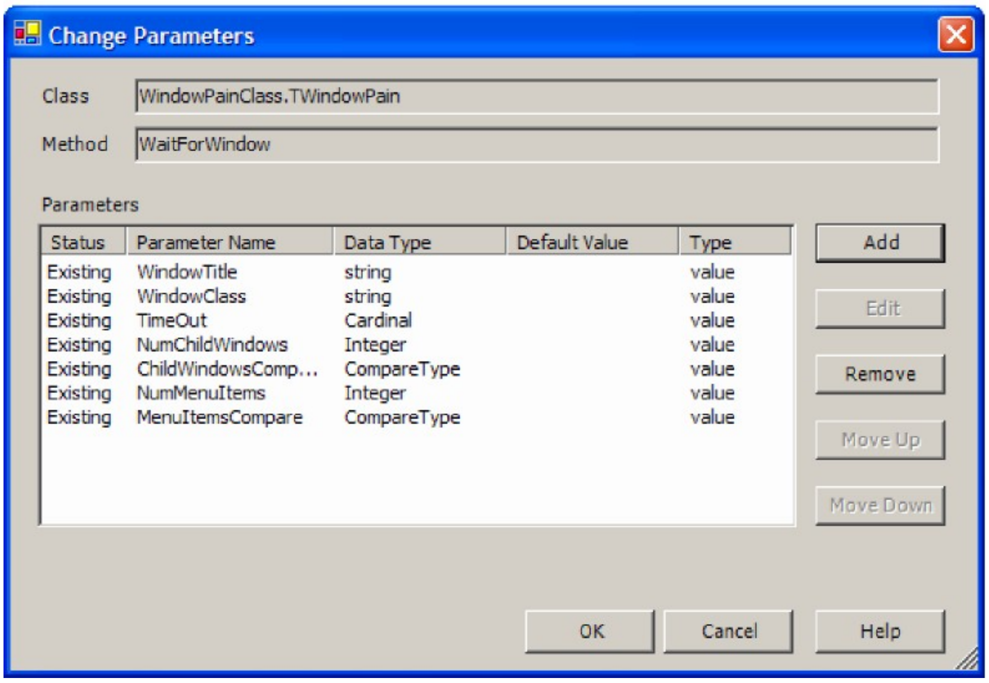
Search: Days

DateUtils.DaySaturday
DateUtils.DaysBetween
DateUtils.DaysInAMonth
DateUtils.DaysInAYear
DateUtils.DaysInMonth
DateUtils.DaysInYear
DateUtils.DaySpan
DateUtils.DaysPerWeek
DateUtils.DaysPerYear
DateUtils.DaySunday

Add to the: [✓ Interface] Implementation

Change parameter refactoring

For the Delphi personality, the change parameter refactoring permits you to quickly modify a method declaration and its implementation block. This feature allows you to quickly add and remove parameters, as well as change the default values of existing parameters. To use this feature, select a method, function, or procedure in the editor (either its declaration in your class or its implementation) and select Refactor I Change Params. Use the Change Parameters dialog box, shown in the following figure, to make the changes you want.



Class: WindowPainClass.TWindowPain

Method: WaitForWindow

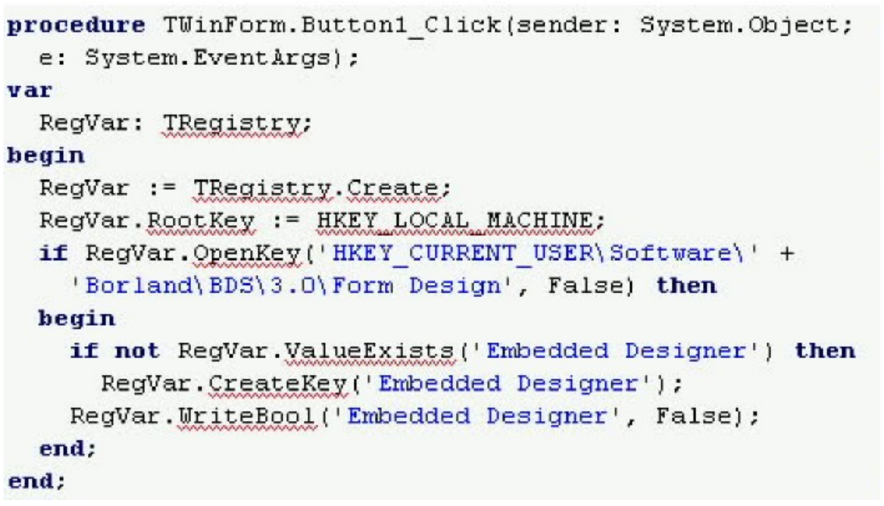
Parameters

Status	Parameter Name	Data Type	Default Value	Type
Existing	WindowTitle	string		value
Existing	WindowClass	string		value
Existing	TimeOut	Cardinal		value
Existing	NumChildWindows	Integer		value
Existing	ChildWindowsComp...	CompareType		value
Existing	NumMenuItems	Integer		value
Existing	MenuItemsCompare	CompareType		value

In addition, a large number of pattern-related refactorings have been introduced in both the C# and Delphi personalities. These refactorings include move, extract interface, extract superclass, pull members up, push members down, safe delete, inline variable, introduce field, and introduce variable. In order to use these refactorings, you must enable modeling support in your corresponding C# or Delphi project. Finally, the powerful rename refactoring has been introduced in the C++ personality.

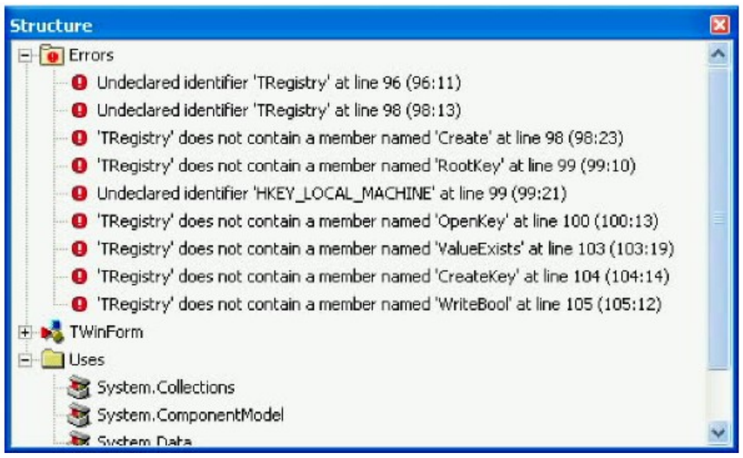
Error Insight

Error Insight, which makes its debut in Delphi 2006, provides you with a service that can be roughly described as spell checking and grammar checking for programmers. As you write your Delphi or C# code, the IDE actively evaluates your work, identifying the symbols, keywords, and directives that you use, looking for syntax and semantic errors that the compiler cannot resolve. When Error Insight locates an error, it identifies the problem by underscoring the offending text with red squiggly lines, similar to how Microsoft Word identifies words not in its dictionary.



```
procedure TWinForm.Button1_Click(sender: System.Object;
  e: System.EventArgs);
var
  RegVar: TRegistry;
begin
  RegVar := TRegistry.Create;
  RegVar.RootKey := HKEY_LOCAL_MACHINE;
  if RegVar.OpenKey('HKEY_CURRENT_USER\Software\' +
    'Borland\BDS\3.0\Form Design', False) then
  begin
    if not RegVar.ValueExists('Embedded Designer') then
      RegVar.CreateKey('Embedded Designer');
    RegVar.WriteBool('Embedded Designer', False);
  end;
end;
```

When you pause your mouse pointer briefly over a symbol that Error Insight does not recognize, Error Insight displays a hint window with information about the identified error. In addition to the Error Insight features available in the code editor, the problems located by Error Insight also dynamically appear in the Structure pane, under the Errors node, and disappear as they are corrected. The following figure shows the Structure pane with a number of identified errors.



Structure

Errors
- Undeclared identifier 'TRegistry' at line 96 (96:11)
- Undeclared identifier 'TRegistry' at line 98 (98:13)
- 'TRegistry' does not contain a member named 'Create' at line 98 (98:23)
- 'TRegistry' does not contain a member named 'RootKey' at line 99 (99:10)
- Undeclared identifier 'HKEY_LOCAL_MACHINE' at line 99 (99:21)
- 'TRegistry' does not contain a member named 'OpenKey' at line 100 (100:13)
- 'TRegistry' does not contain a member named 'ValueExists' at line 103 (103:19)
- 'TRegistry' does not contain a member named 'CreateKey' at line 104 (104:14)
- 'TRegistry' does not contain a member named 'WriteBool' at line 105 (105:12)

TWinForm

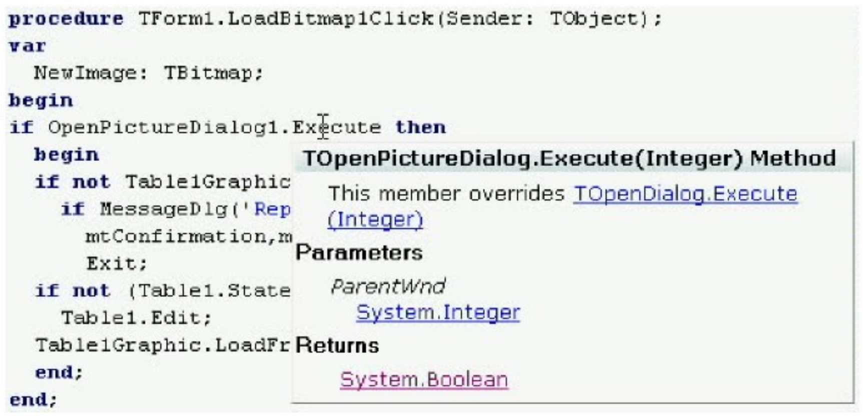
Uses
- System.Collections
- System.ComponentModel
- System.Data

In the example shown in the preceding figures, adding the Borland.Vcl.Registry unit (for the TRegistry class) and Borland.Vcl.Windows (for the HKEY_LOCAL_MACHINE constant) to this unit's uses clause allows Error Insight to see the various symbols that it identified as problems. Once these two units are added to the uses clause, both the Structure pane and the code editor are updated, indicating that no problems are detected in this code. You can configure Error Insight from the Code Insight node of the Options dialog box. Display this dialog box by selecting Tools | Options from the main menu.

Help Insight

Another new Code Insight feature appearing in Delphi 2006 is Help Insight. Help

Insight provides you with information about the classes, interfaces, methods, properties, and fields that appear in your code, without you ever having to leave the code editor. To access Help Insight, briefly pause your mouse pointer over a symbol in the code editor. After a moment, a hint window appears, displaying information about the symbol.



```
procedure TForm1.LoadBitmap1Click(Sender: TObject);
var
  NewImage: TBitmap;
begin
if OpenPictureDialog1.Execute then
  begin
  if not Table1Graphic
    if MessageDlg('Rep
      mtConfirmation,m
      Exit;
  if not (Table1.State
    Table1.Edit;
  Table1Graphic.LoadFr
  end;
end;
```

TOpenPictureDialog.Execute(Integer) Method

This member overrides TOpenDialog.Execute (Integer)

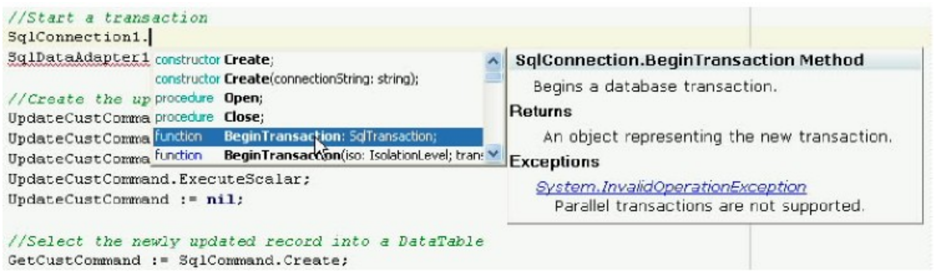
Parameters

ParentWnd
System.Integer

Returns

System.Boolean

In many instances, Help Insight includes one or more links within the hint window. Clicking one of these links may drill down into the help, displaying an additional hint window with information about the link you clicked. Alternatively, clicking a link may take you to the line of code where the clicked symbol is defined. Help Insight is also available from the windows displayed by Code Insight, including the Class Completion and Argument Value List windows. When a Code Insight window is active, select an item in the Code Insight window to show the Help Insight for that item. For example, in the following figure Help Insight is displaying information about the BeginTransaction method of a SqlConnection object. This help became available after BeginTransaction was selected in the Code Completion window.



```
//Start a transaction
SqlConnection1.
SqlDataAdapter1

//Create the up
UpdateCustComma
UpdateCustComma
UpdateCustComma
UpdateCustCommand.ExecuteScalar;
UpdateCustCommand := nil;

//Select the newly updated record into a DataTable
GetCustCommand := SqlCommand.Create;
```

```
constructor Create;
constructor Create(connectionString: string);
procedure   Open;
procedure   Close;
function    BeginTransaction: SqlTransaction;
function    BeginTransaction(iso: IsolationLevel; trans
```

SqlConnection.BeginTransaction Method

Begins a database transaction.

Returns

An object representing the new transaction.

Exceptions

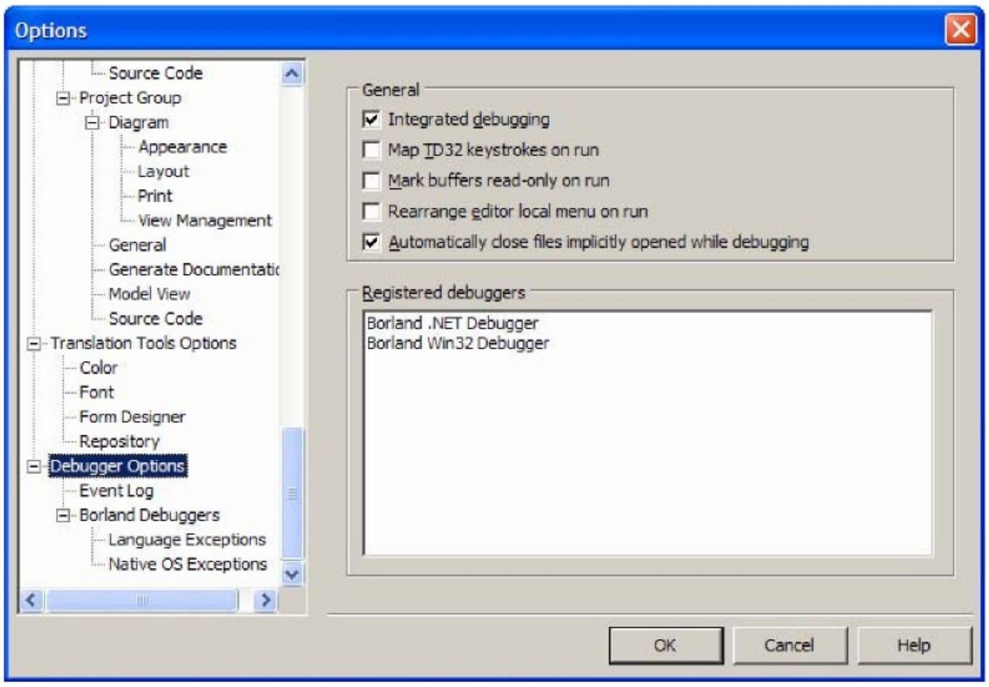
System.InvalidOperationException
Parallel transactions are not supported.

You can configure Help Insight from the Code Insight node of the Options dialog box.

Improved Page Management

Two improvements to editor page management have been introduced in Borland Developer Studio 2006. The first is that you can now configure whether units opened during debugging remain open, or not, once you exit the debugger. Specifically, when you step into a unit that is not already opened in the code editor, that unit is opened automatically. Previously, that unit would have remained opened after you exited the debugger. You can now control this behavior by enabling or disabling the Automatically close files implicitly opened while debugging, check box on the

Debugger Options page of the Options dialog box, shown in the following figure.



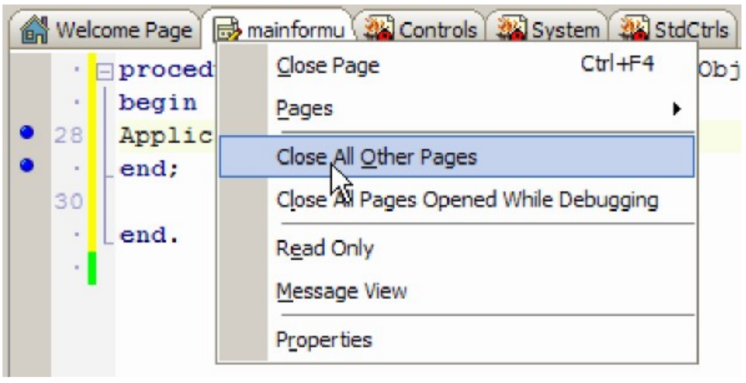
General

- [✓] Integrated debugging
- [ ] Map TD32 keystrokes on run
- [ ] Mark buffers read-only on run
- [ ] Rearrange editor local menu on run
- [✓] Automatically close files implicitly opened while debugging

Registered debuggers

Borland .NET Debugger
Borland Win32 Debugger

The second code page management upgrade can be found on the code page right-click menu. This menu now includes an option to close all opened pages, with the exception of the code page you right-click, as shown in the following figure.



Together, these improvements to code page management save you time and reduce clutter in your editor.

The History Manager

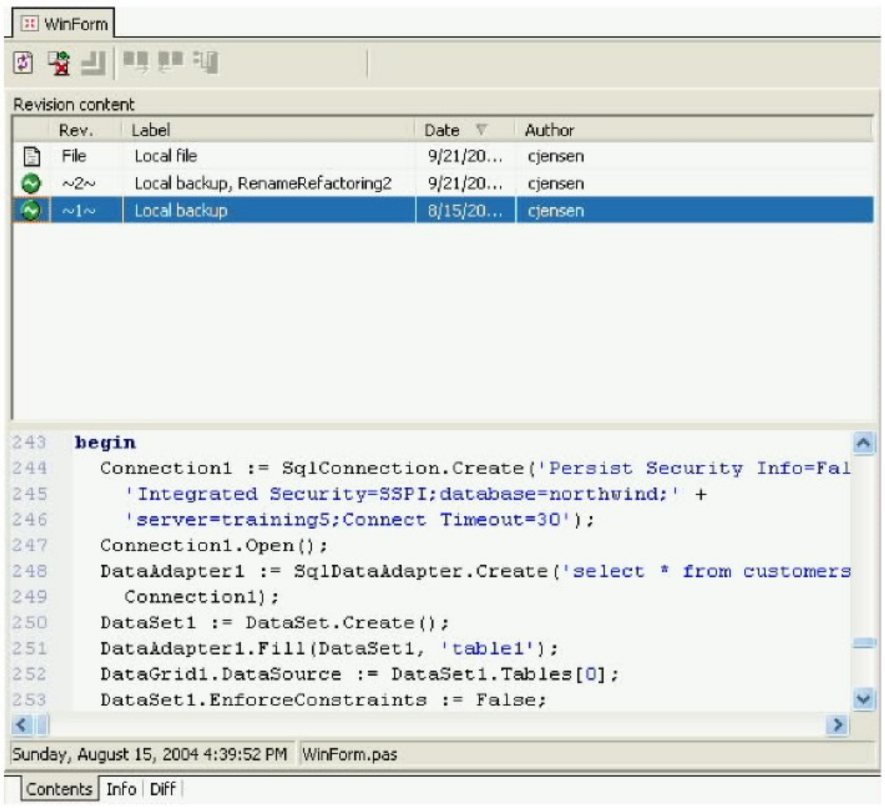
One of the more exciting additions to the Delphi 2006 code editor is the History Manager. The History Manager, which you display by clicking the History tab when a source file is active in the code editor, allows you to view changes to your source files over time, view comments about specific versions of your source code, view the differences between the various saved versions of your files, and easily revert to any backup state or checkin.

By default, the History Manager transparently maintains local copies of your source files in a folder named __history under your project directory each time you save your changes. This feature is called local file backup, and you use the Options dialog box to configure how many versions of your local backup to keep. Delphi 2006 maintains the last 10 saved versions of each source file, by default. Depending on your available hard disk space, you may want to increase the number of backups. If you are using Borland's StarTeam version control server, the History Manager maintains StarTeam checkins as well. Using this feature, you can not only view changes that you have made to the source files, but also compare your changes with those implemented by other developers working on the StarTeam-managed project. StarTeam also permits you to track changes even after you have changed a file's name. In short, the History Manager provides you with a convenient and powerful interface to the robust StarTeam project asset management system.

It's worth noting that the History Manager also works with the DFM files of VCL and VCL for .NET applications. DFM files are used in those applications to persist information about the properties of the objects that appear on your forms, data modules, and frames. As a result, the History Manager permits you to view, manage, and restore changes made to your form designs using the same tools as those used on Delphi code files. There are three panes available within the History Manager. These panes are named Content, Info, and Diff. Each of these panes is described in the following sections.

The Content Pane

You use the Content pane to review the contents of your saved source files, and optionally revert to a previously saved version. When you select a specific backup or the current saved version of the file, the contents of that file are displayed in the code area. In addition, the file name and the date last saved appear in the History Manager's status bar.



Rev.	Label	Date	Author
File	Local file	9/21/20...	cjensen
~2~	Local backup, RenameRefactoring2	9/21/20...	cjensen
~1~	Local backup	8/15/20...	cjensen

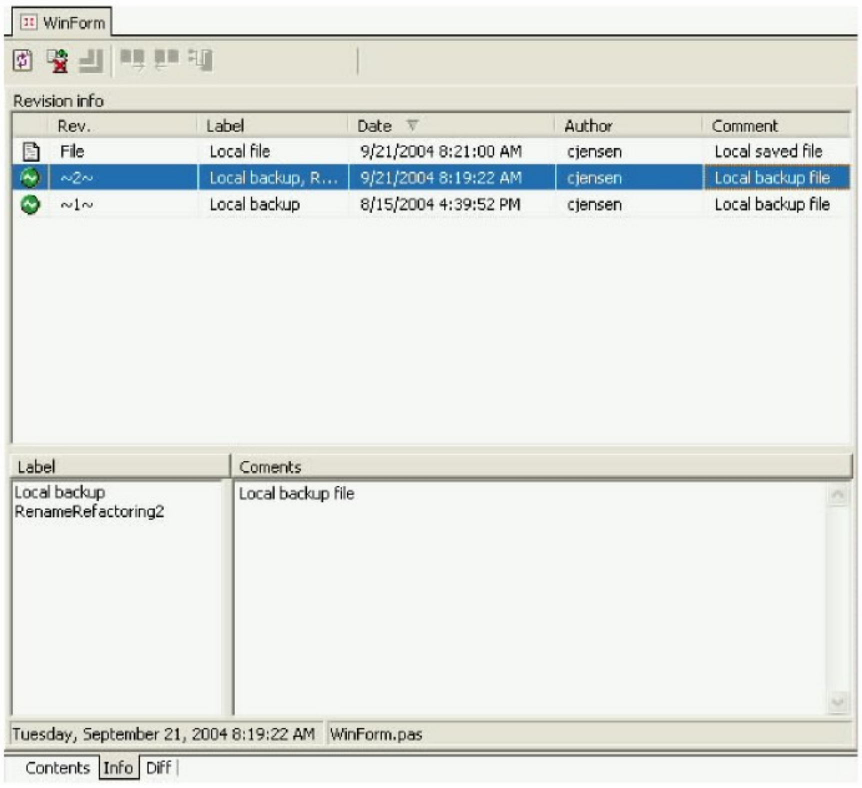
```
begin
  Connection1 := SqlConnection.Create('Persist Security Info=Fal
    'Integrated Security=SSPI;database=northwind;' +
    'server=training5;Connect Timeout=30');
  Connection1.Open();
  DataAdapter1 := SqlDataAdapter.Create('select * from customers
    Connection1);
  DataSet1 := DataSet.Create();
  DataAdapter1.Fill(DataSet1, 'table1');
  DataGrid1.DataSource := DataSet1.Tables[0];
  DataSet1.EnforceConstraints := False;
```

Sunday, August 15, 2004 4:39:52 PM WinForm.pas

Use the code area to view the contents of the selected file. If you want, you can use the code area to select and copy (Ctrl-C) lines of code that you want to paste elsewhere within your project (or even into other projects).
If you want to revert your code to one of the previous saved versions, select the saved backup that you want to revert to and click the Revert to previous version button in the History Manager toolbar.

The Info Pane

You use the Info pane to view comments and notes associated with a particular version of your source file. If you are using StarTeam to manage your History Manager contents, these comments are linked to your StarTeam backups. If you are using local backups, these comments are generated by Delphi 2006 and cannot be modified. Some operations, such as refactorings, write information into the Info pane of the History Manager.



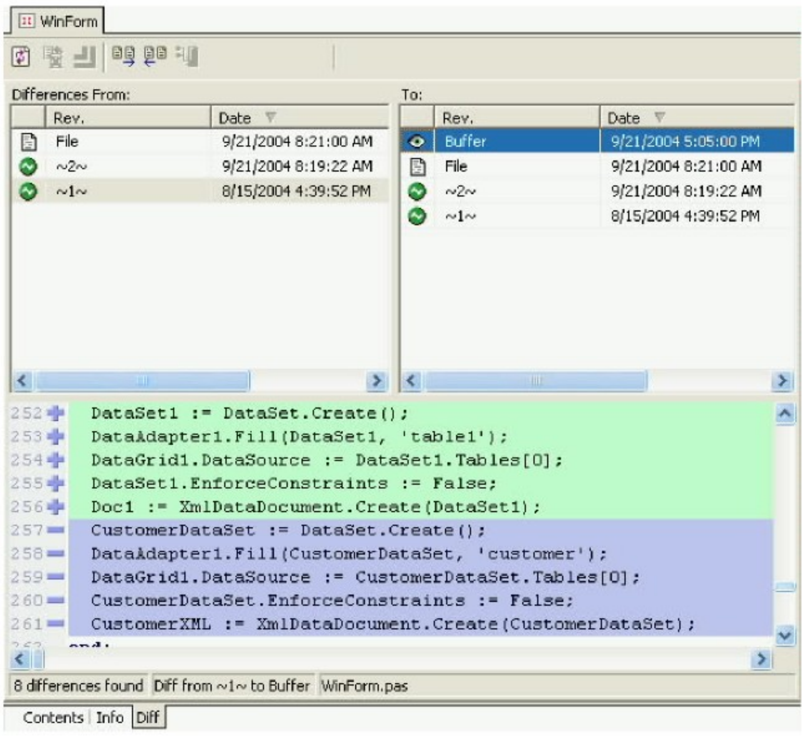
Rev.	Label	Date	Author	Comment
File	Local file	9/21/2004 8:21:00 AM	cjensen	Local saved file
~2~	Local backup, R...	9/21/2004 8:19:22 AM	cjensen	Local backup file
~1~	Local backup	8/15/2004 4:39:52 PM	cjensen	Local backup file

Label	Coments
Local backup RenameRefactoring2	Local backup file

Tuesday, September 21, 2004 8:19:22 AM WinForm.pas

The Diff Pane

For most developers, the Diff pane offers the most valuable feature of the History Manger. The Diff pane provides insight into the differences between the multiple versions of your source code, including comparisons between the current edit buffer and saved source files. Select one of the saved versions of your source file from the Differences From: list on the left side of the Diff pane, and either the contents of the current edit buffer or one of the other saved versions from the To: list on the right side. The difference view is displayed in the code area, with the newer code versions identified with a plus sign (+) in the left gutter, and the older versions identified with a minus (-) sign. The following figure depicts changes between the current version of the file in the edit buffer and one of the saved local backups.



Differences From:

Rev.	Date
File	9/21/2004 8:21:00 AM
~2~	9/21/2004 8:19:22 AM
~1~	8/15/2004 4:39:52 PM

To:

Rev.	Date
Buffer	9/21/2004 5:05:00 PM
File	9/21/2004 8:21:00 AM
~2~	9/21/2004 8:19:22 AM
~1~	8/15/2004 4:39:52 PM

```
+ DataSet1 := DataSet.Create();
+ DataAdapter1.Fill(DataSet1, 'table1');
+ DataGrid1.DataSource := DataSet1.Tables[0];
+ DataSet1.EnforceConstraints := False;
+ Doc1 := XmlDataDocument.Create(DataSet1);
- CustomerDataSet := DataSet.Create();
- DataAdapter1.Fill(CustomerDataSet, 'customer');
- DataGrid1.DataSource := CustomerDataSet.Tables[0];
- CustomerDataSet.EnforceConstraints := False;
- CustomerXML := XmlDataDocument.Create(CustomerDataSet);
  end;
```

8 differences found | Diff from ~1~ to Buffer | WinForm.pas

Updated Memory Manager

There is one major enhancement in particular that affects every aspect of the IDE, as well as any Win32 applications you compile using Borland Developer Studio 2006, but this enhancement is not visual in nature. Borland Developer Studio received a significant upgrade in the form of a new memory manager. Based on FastMM, the new memory manager optimizes memory utilization while providing significantly better performance. Importantly, this same memory manager is included in any Win32® applications that you compile with Borland Developer Studio 2006. In other words, without making any other changes to your code, simply re-compiling your existing Win32 Delphi or C++ applications in Borland Developer Studio 2006 should improve the performance of those applications. While the new memory manager has no affect on compiled .NET code (that is the responsibility of the .NET just-in-time compiler), it does provide significant performance improvements in the Borland Developer Studio 2006 IDE, even when you are working with .NET applications.

Debugger Enhancements

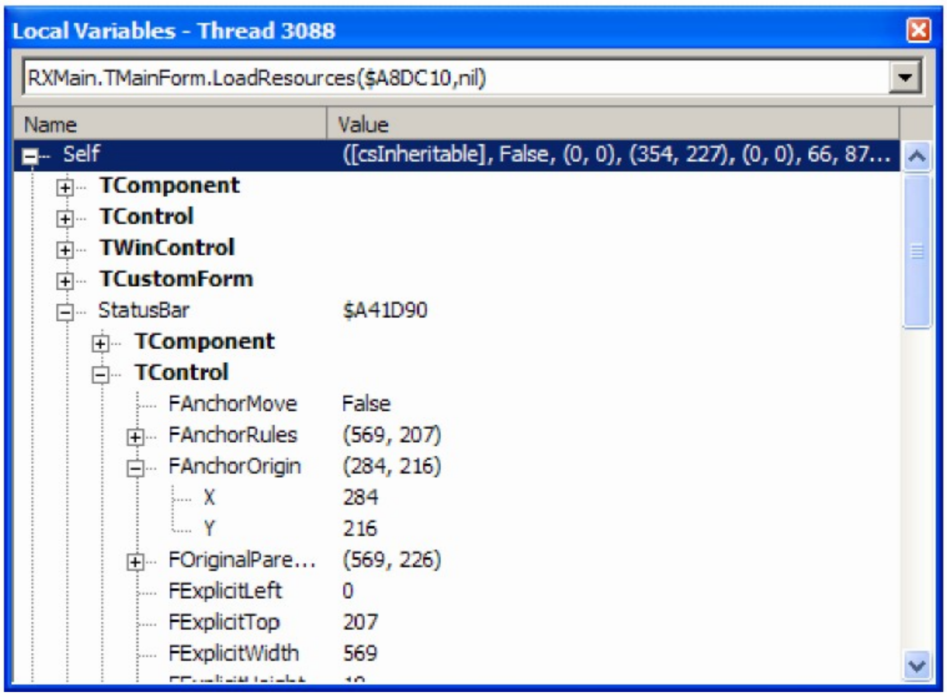
No comprehensive development environment would be complete without a world-class debugger, and Borland Developer Studio 2006 has two of them. One debugger is for Win32 debugging and the other is for .NET. As in all other areas of Borland Developer Studio 2006, Borland has introduced a number of performance and feature enhancements to these debuggers. Some of these updates are relatively minor, providing you with added convenience. For example, individual breakpoints can now be easily enabled or disabled by simply Ctrl-clicking the breakpoint in question in the editor gutter. Likewise, as mentioned earlier in this guide, code files that are implicitly opened during debugging can be configured to automatically close when execution terminates. Another enhancement is that you can now copy content from the CPU window to the Windows clipboard. In addition, there are also some major enhancements and updates to Borland Developer Studio 2006's debuggers. These are described in the following sections.

Remote Debugging

More so than ever, the applications you build today are likely to execute in a distributed environment, an environment that introduces significant challenges if you need to debug a remote process. Fortunately, Borland Developer Studio 2006 marks the return of one of Delphi's more powerful debugging features – remote debugging. With the remote debug server, you can use your development machine to step into code that is running on another computer on your network or the Internet. With remote debugging, you can see first hand how your program is actually running on the server to which it is deployed.

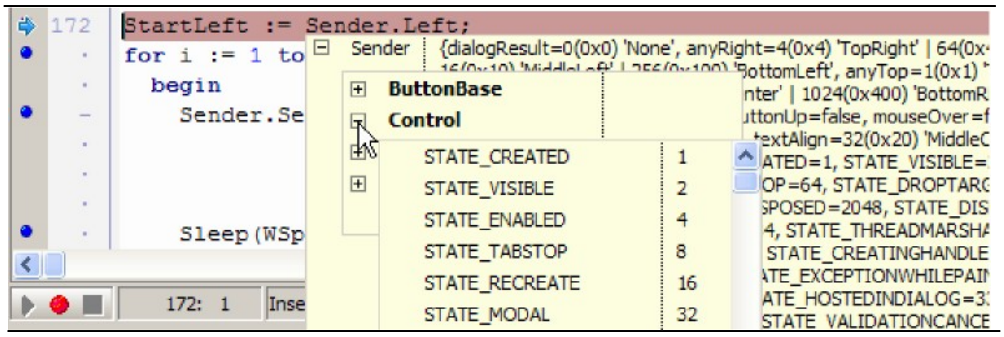
Expandable Watches and Local Variables

When debugging, the properties and fields of symbols that appear in the Watch List and Local Variable panes are now expandable. This feature greatly reduces desktop clutter while you are debugging. Examples of expandable symbols are shown in the following figure. Here the Local Variables pane is used to inspect Self, which in this case is the main form of this project. Notice that each of the form's properties and fields can be expanded, which in turn can be further expanded. You had to open multiple inspector windows in previous versions of Delphi to obtain this same information.


Local Variables - Thread 3088
RXMain.TMainForm.LoadResources($A8DC10,nil)

```
Name                 Value
⊟ Self               ([csInheritable], False, (0, 0), (354, 227), (0, 0), 66, 87...
  ⊞ TComponent
  ⊞ TControl
  ⊞ TWinControl
  ⊞ TCustomForm
  ⊟ StatusBar        $A41D90
    ⊞ TComponent
    ⊟ TControl
        FAnchorMove      False
      ⊞ FAnchorRules     (569, 207)
      ⊟ FAnchorOrigin    (284, 216)
          X              284
          Y              216
      ⊞ FOriginalPare... (569, 226)
        FExplicitLeft    0
        FExplicitTop     207
        FExplicitWidth   569
```

Tooltip windows are also now expandable. Specifically, when you pause your cursor over a symbol while the debugger is engaged, and the symbol can be drilled into, the displayed tooltip includes a plus sign ("+") that you can click to expand the help. The following figure shows a portion of an expanded tooltip displayed by the debugger.



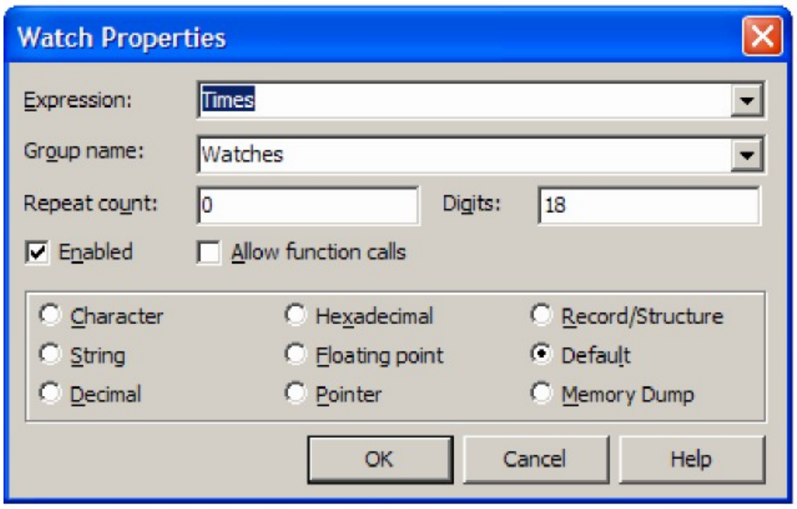
```
StartLeft := Sender.Left;
for i := 1 to
  begin
    Sender.Se

    Sleep(WSp
```

```
Sender  {dialogResult=0(0x0) 'None', anyRight=4(0x4) 'TopRight' | 64(0x
        'BottomLeft', anyTop=1(0x1)
        nter' | 1024(0x400) 'BottomR
        ittonUp=false, mouseOver=f
        textAlign=32(0x20) 'MiddleC
        ATED=1, STATE_VISIBLE=
        OP=64, STATE_DROPTARG
        SPOSED=2048, STATE_DIS
        4, STATE_THREADMARSHA
        STATE_CREATINGHANDLE
        ATE_EXCEPTIONWHILEPAIN
        ATE_HOSTEDINDIALOG=3
        STATE_VALIDATIONCANCE
```

ButtonBase	
Control	
STATE_CREATED	1
STATE_VISIBLE	2
STATE_ENABLED	4
STATE_TABSTOP	8
STATE_RECREATE	16
STATE_MODAL	32

Grouped Watches

Watches can now be grouped, permitting you to better organize and control the symbols whose values interest you. You assign a particular watch to a group when you define the watch. As you can see from the Watch Properties dialog box shown in the following figure, you can select the name of the group to which the watch belongs when you define it.



Watch Properties

Expression: Times

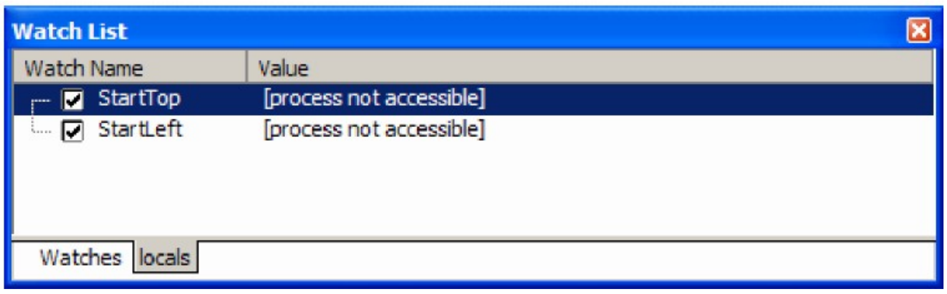
Group name: Watches

Repeat count: 0   Digits: 18

[✓ Enabled]   Allow function calls

Character   Hexadecimal   Record/Structure
String   Floating point   [✓ Default]
Decimal   Pointer   Memory Dump

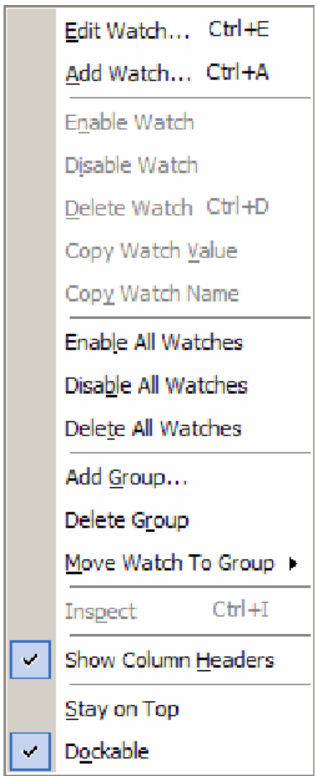
Each group is represented by a separate tab in the Watch List pane. To view the symbols for a particular group, select the tab for that group. The Watch List pane in the following figure contains two groups, Watches and locals. In this figure, the locals tab is currently selected.



Watch List

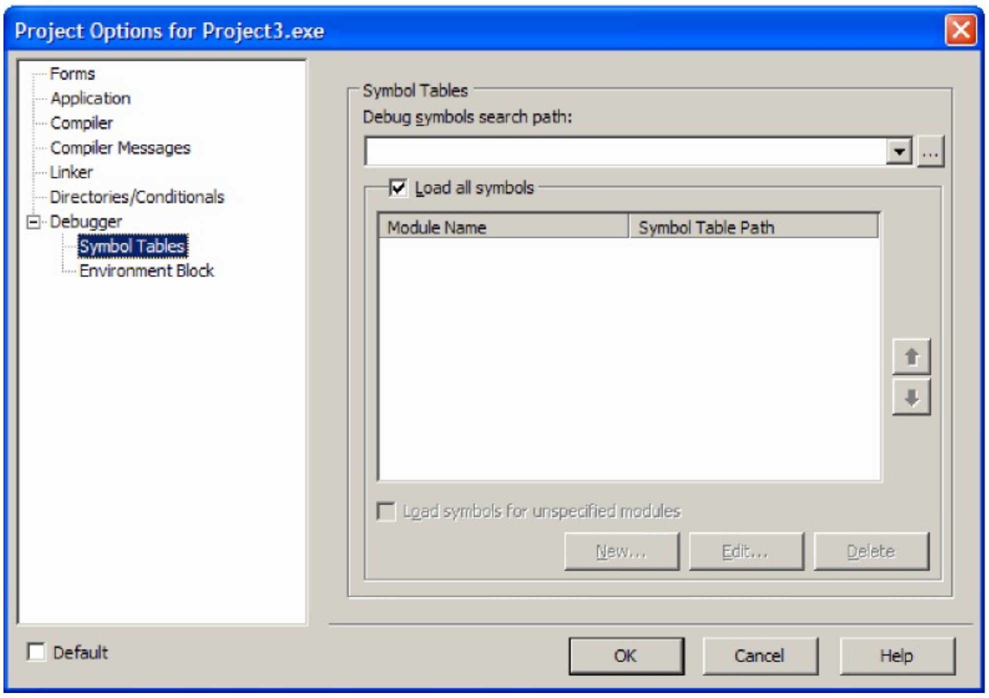
Watch Name	Value
✓ StartTop	[process not accessible]
✓ StartLeft	[process not accessible]

To further support groups, the Watch List context menu has been updated to permit you to add, delete, and manage your Watch List groups. This menu is shown in the following figure.



Symbol Table Management

Borland Developer Studio 2006 now permits you to control the order in which symbol tables are loaded for the module you are debugging. You can also choose to restrict the debugger's search to particular symbol tables, a technique that can improve your debugging performance. To configure symbol table usage, use the Symbol Tables node of the Project Options dialog box, as shown in the following figure. You can also access this feature from the Run Parameters dialog box, which you display by selecting Run | Parameters from the Borland Developer Studio main menu.

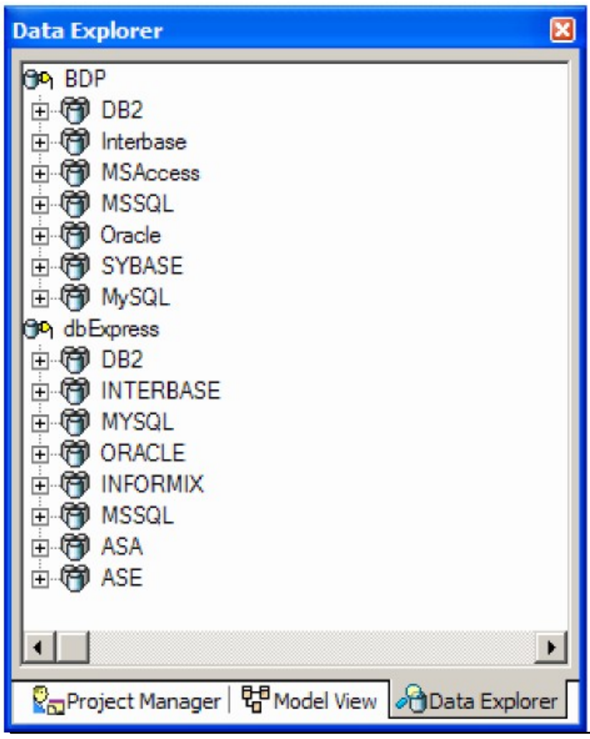


Database

Delphi's exceptional database support has always set it apart from other development environments this legacy continues with Borland Developer Studio 2006. And although Borland Developer Studio 2006 has far more database support than any other single product, Borland has managed to squeeze even more database features into this release. These are described in the following sections.

Support for dbExpress™ in the Data Explorer

The Data Explorer, shown in the following figure, now includes support for dbExpressTM databases. Features accessible from the Data Explorer for dbExpress-supported databases include SQL testing and execution, stored procedure execution with parameter definition, data viewing, and drag-and-drop placement of components in the designer. These features apply to both for VCL and VCL for .NET applications.



Borland Database Drivers

The following table contains a complete list of the databases that are directly supported by custom Borland drivers through BDP for ADO.NET and dbExpress. Note, however, that Borland Developer Studio also provides access to virtually every available database through its support for the Borland Database Engine, Ole Db Providers, ODBC drivers, ADO.NET data providers, as well as third-party database components.

BDP for ADO.NET	dbExpress
Borland® InterBase® V 7.5 and 7.5.1	Borland InterBase V 7.5 and 7.5.1
Oracle® 10g	Oracle®10g
IBM® DB2® 8.x	IBM DB2 8.x
MS SQL Server TM 2000 and 2006 and MSDE ® 2000	MS SQL Server 2000 and 2005

Sybase® 12.6	Sybase 12.6
MySQL® 4.0.24	MySQL 4.0.24
	Informix® 9
	SQL Anywhere® 9

Other Updates to dbExpress

In addition to Data Explorer support for dbExpress, dbExpress has received several additional upgrades in Borland Developer Studio 2006. For instance, the TSqlQuery and TSqlDataSet components now support both IN and OUT parameters. In addition, dbExpress for .NET now includes full Unicode® support.

Summary

At the beginning of this reviewer's guide you learned that Borland Developer Studio 2006 is the ultimate development tool, providing you with a peerless combination of development and support tools in a tightly integrated product.

Without question, Delphi 2006, C++Builder 2006, and C#Builder 2006 are the best and most flexible tools available for building applications for the Win32 and .NET platforms. But the benefits of having these developer tools tightly integrated with world-class application lifecycle management tools simply cannot be stressed enough. The seamless integration you find in Borland Developer Studio 2006 helps foster better communication between you and the other major stakeholders of your project and development teams. By enabling better communication, you can design and develop faster and better, and significantly reduce missteps. The Together integration allows you to provide advanced models of your systems before and during the development process, and the code audits and metrics ensure that what you are developing is of the highest standards and is easier to maintain.

The StarTeam integration gives you much more than the standard check-in and check-out process that you normally associate with a source code version control system. It permits you to submit change requests and generate automatic notifications of underlying changes directly to your team. In addition, the type of license you get with Borland Developer Studio 2006 means that you and your team are ready to work together. It also allows you to upgrade at a significant savings to more advanced versions that support bug tracking, newsgroups, and customized workflow capabilities. Having up-to-date requirements is key to your producing quality results on time. A recent report found that 71% of all projects that failed did so in part due to poorly defined requirements. The integration found in Borland Developer Studio 2006 increases your team's communication throughout the entire lifecycle of your project. Not only will this ensure that your projects run smoother, but it also allows you to implement projects that are in concert with your organization's needs.

About Borland Software Corporation

Founded in 1983, Borland Software Corporation (NASDAQ: BORL) is the global leader in platform independent solutions for software delivery optimization. The company provides the software and services that align the teams, technology and processes required to maximize the business value of software. To learn more about delivering quality software, on time and within budget, visit: http://www.borland.com.

About the Author

Cary Jensen is President of Jensen Data Systems, Inc., a software development, training, and consulting company (http://www.jensendatasystems.com). He is an award-winning, bestselling author of nineteen books, a frequent columnist on the Borland Developer Network (http://bdn.borland.com), and a popular speaker at conferences, workshops, and training seminars around the world. Cary has a Ph.D. in Human Factors Psychology, specializing in human-computer interaction, from Rice University in Houston, Texas. You can contact Cary at cjensen@jensendatasystems.com.

Acknowledgements

Many people at Borland contributed their valuable time to comment on and contribute to this reviewer's guide. In particular, the author wishes to thank Rob Cheng, Michael Swindell, Michael Rozlog, Malcolm Groves, Henrik Jondell, Bruneau Babet, Allen Bauer, Adam Markowitz, Darren Kosinski, Chris Benson, Ramesh Theivendran, David Intersimone, John Kaster, and last, but not least, Jason Vokes, for their comments and input.