

Essential Delphi

IDE 10.3

Sample content of a coming book introducing the Delphi IDE (and updated for Delphi 10.3). The material is based (with permission) on classic editions of Mastering Delphi.

This ebook is copyright Marco Cantu 1995-2020. All rights reserved.

For more information refer to www.marcocantu.com

01: A Form Is A Window

The Delphi IDE is a very complex application, with many feature accumulated over 25 years the product has been in existence. So the question where to start covering it is more than legitimate. In this first chapter of the book I want to provide a practical introduction, in case you've never build an application with Delphi. Starting from the next chapter I'll go deeper in coverage going over each of the IDE areas in more and more detail.

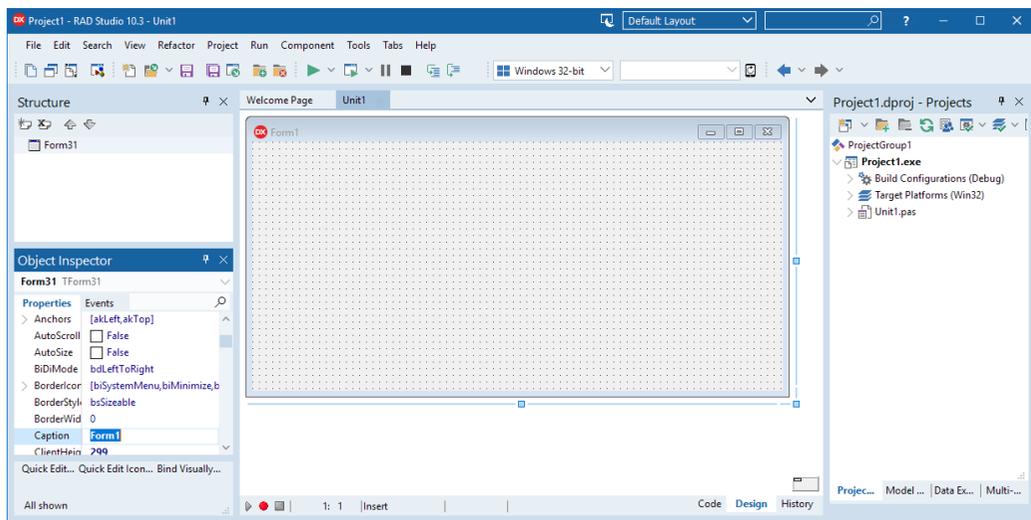
Let's start our exploration of Delphi by looking to the simplest scenario, building Windows applications using the VCL library. This is the easiest starting point, while most concept will also apply to mobile and multi-device development.

Windows applications are usually based on windows. So, how are we going to create our first window? We'll do it by using a form. As the chapter title suggests, a form really is a window in disguise. There is no real difference between the two concepts, at least from a general point of view.

note If you look closely, a form is always a window, but the reverse isn't always true. Some Delphi components are windows, too. A push button is a window. A list box is a window. To avoid confusion, I'll use the term form to indicate the main window of an application or a similar window and the term window in the broader sense. In the mobile world these relationships are a bit more intricate and platform-specific, but the broad concept still applies.

Creating Your First Form

Even though you have probably already created at least some simple applications in Delphi, I'm going to show you the process again, to highlight some interesting points. Creating a form for a Windows application is one of the easiest operations in the system: You only need to create a new “Windows VCL Application” from the File | New menu or the Welcome page and Delphi will add to it a new, empty form, as you can see in the figure below. That's all there is to it.

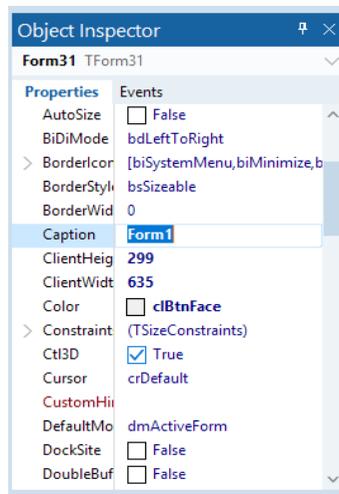


Believe it or not, you already have a working application. You can run it, using the Run button on the toolbar or the Run | Run menu command or just pressing F9, and it will result in a standard Windows program. Of course, this application won't be very useful, since it has a single empty window with no capabilities, but the default behavior of any Windows window.

4 - 01: A Form Is a Window

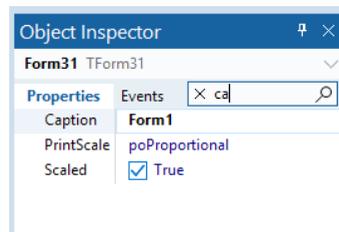
Adding a Title

Before we run the application, let's make a quick change. The title of the form is Form1. For a user, the title of the main window stands for the name of the application. Let's change Form1 to something more meaningful. When you first open Delphi, the Object Inspector window should appear on the left side of the form (if it doesn't, open it by choosing View | Tools Windows | Object Inspector or pressing the F11 key):



The Object Inspector shows the properties of the selected component. The window contains a tab control with two pages. The first page is labeled Properties. The other page is labeled Events and shows a list of events that can take place in the form or in the selected component.

The properties are listed in alphabetical order, so it's quite easy to find the ones you want to change, but you can also type the name of the property in the search box to get quickly to it:

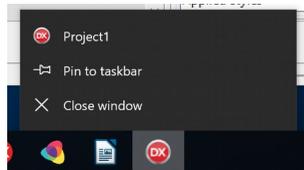


note It is also possible to group properties in the Object Inspector by category, but this feature is seldom used by Delphi developers.

We can change the title of the form simply by changing the `Caption` property. While you type a new caption, you can see the title of the form change. If you type Hello, the title of the form changes immediately. As an alternative, you can modify the internal name of the form by changing its `Name` property. If you have not entered a new caption, the new value of the `Name` property will be used for the `Caption` property, too.

tip Only a few of the properties of a component change while you type the new value. Most are applied when you finish the editing operation and press the Enter key (or move the input focus to a new property).

Although we haven't done much work, we have built a full-blown application, with a system menu and the default Minimize, Maximize, and Close buttons. You can resize the form by dragging its borders, move it by dragging its caption, maximize it to full-screen size, or minimize it. It works, but again, it's not very useful. If you look at the menu in the Windows Taskbar, you'll see that something isn't right:



Instead of showing the caption of the form as the icon caption, it shows the name of the project, something like `Project1`. We can fix this by giving a name to the project, which we'll do by saving it to disk with a new name.

Saving the Form

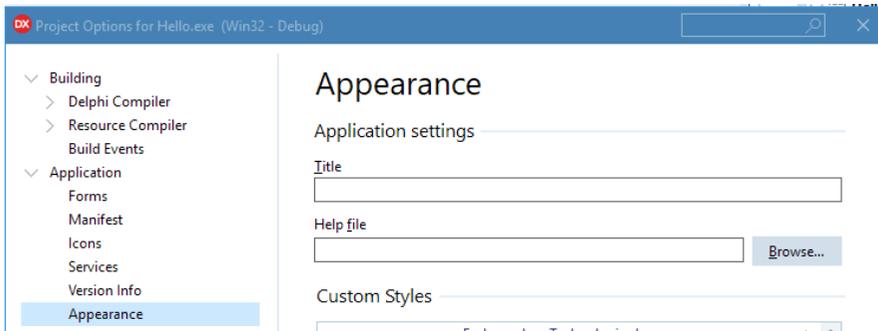
Select the Save Project or Save Project As command from the File menu (or the matching toolbar button), and Delphi will first ask you to give a name to the source code file, or unit, associated with the form, and then to name the project file. Since the name of the project should match the caption of the form (Hello), I've named the form source file `HelloForm.pas`. I've given the project file the name `Hello.dpr`.

Unfortunately, we cannot use the same name for the project and the unit that defines the form; for each application, these items must have unique names. You can add Form, just use Unit, or call every initial form unit `MainForm`, or choose any other naming convention you like. I tend to use a name similar to the project name,

6 - 01: A Form Is a Window

as simply calling it MainForm means you'll end up with a number of forms (in different projects) that all have the same name.

The name you give to the project file is used by default at run-time as the title of the application, displayed by Windows in the Taskbar while the program is running. For this reason, if the name of the project matches the caption of the main form, it will also correspond to the name on the Taskbar. You can also change the title of the application by using the Application | Appearance page of the Project Options dialog box (choose Project | Options):



As an alternative you can write a line of code to change the Title property of the Application global object, but that's for a later time.

Using Components

Now it's time to start placing something useful in our Hello form. Forms can be thought of as component containers. Each form can host a number of components or controls.

You can choose a component from the Palette toolbox (by default on the right of the form designer, under the Project manager. If you choose the Button component from the Standard page of the Palette, for example, you can do any of the following four simple ways to place a component on a form:

- Click on the component, move the mouse cursor to the form, press the left mouse button to set the upper-left corner of the button, and drag the mouse to set the button's size.
- Select the component as above, and then simply click on the form to place a button of the default height and width.

- Double-click on the icon in the Components Palette, and a component of that type will be added in the center of the form.
- Shift-click on the component icon, and place several copies of the component in the form using one of the above procedures. When you are done click on the Arrow button in the Palette toolbar to disable the selection.

Our form will have only one button, so we'll center it in the form. You can do this by hand, with a little help from Delphi. When you choose View | Toolbars | Position, a new section gets added to the Delphi IDE toolbar with positioning icons:



note The Positioning toolbar (which was originally mixed with Align toolbar) is a rarely used feature these days. But I've decided to keep these steps and the example from the original Mastering Delphi book anyway.

This toolbox includes buttons to bring control in front of push the behind other controls, and center them in the form. A separate Align toolbar helps aligning controls one to the other. Using the last two buttons, you can place a component in the center of the form.

Although we've placed the button in the center, as soon as you run the program, you can resize the form and the button won't be in the center anymore. So the button is only in the center of the form at start up. Later on, we'll see how to make the button remain in the center after the form is resized, by adding some code. For now, our first priority is to change the button's label.

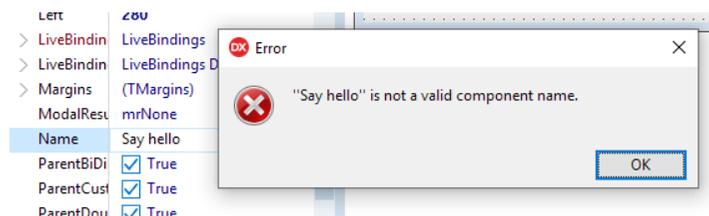
Changing Properties

Like the form, the button has a `Caption` property that we can use to change its label (the text displayed inside it). As a better alternative, we can change the name of the button. The name is a kind of internal property, used only in the code of the program. However, as I mentioned earlier, if you change the name of a button before changing its caption, the `Caption` property will have the same text as the `Name` property. Changing the `Name` property is usually a good choice, and you should generally do this early in the development cycle, before you write much code.

8 - 01: A Form Is a Window

note It is quite common to define a naming convention for each type of component (usually the full name or a shorter version, such as “btn” for Button). If you use a different prefix for each type of component (as in “ButtonHello” or “BtnHello”), the combo box above the Object Inspector will list the components of the same kind in a group, because they are alphabetically sorted. If you instead use a suffix, naming the components “HelloButton” or “HelloBtn,” components of the same kind will be in different positions on the list. In this second case, however, finding a particular component using the keyboard might be faster. In fact, when the Object Inspector is selected you can type a letter to jump to the first component whose name starts with that letter.

Besides setting a proper name for a component, you often need to change its `Caption` property. There are at least two reasons to have a caption different from the name. The first is that the name often follows a naming convention (as described in the note above) that you won’t want to use in a caption. The second reason is that captions should be descriptive, and therefore they often use two or more words, as in my Say hello button. If you try to use this text as the `Name` property, however, Delphi will show an error message:

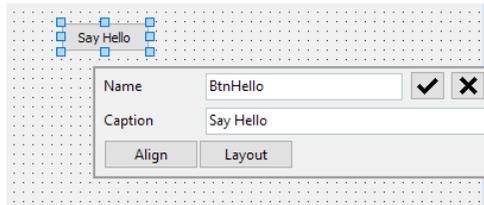


The name is an internal property, and it is used as the name of a variable referring to the component. Therefore, for the `Name` property, you must follow the rules for naming an identifier in the Pascal language:

- An identifier is a sequence of letters, digits, or underscore characters of any length.
- The first character of an identifier cannot be a number; it must be a letter or the underscore character.
- No spaces are allowed in an identifier.
- Identifiers are not case-sensitive, but usually each word in an identifier begins with a capital letter, as in `BtnHello`. But `btnhello`, `btnHello`, and `BTNHHello` refer to this same identifier.

tip You can use the `IsValidIdent` system function in your applications to check whether a given string is a valid identifier.

While in the sections above we've gone over manual steps, there is a faster way to set key properties of a component like its `Name` and `Caption`, and that is the use of the Quick Edit feature. Select the component in the form designer, right click on it, and pick the Quick Edit menu item. You'll see near the component a pane that allows you to quickly edit those two properties and buttons to expand panels for managing other common features:



Here is a summary of the changes we have made to the properties of the button and form. At times, I'll show you the structure of the form of the examples as it appears once it has been converted in a readable format (I'll describe how to convert a form into text later in this chapter). I won't show you the entire textual description of a form (which is often quite long), but rather only its key elements. I won't include the lines describing the position of the components, their sizes, or some less important default values. Here is the code:

```

object Form1: TForm1
  Caption = 'Hello'
  OnClick = FormClick
  object BtnHello: TButton
    Caption = 'Say hello'
    OnClick = BtnHelloClick
  end
end

```

This description shows some attributes of the components and the events they respond to. We will see the code for these events in the following sections. If you run this program now, you will see that the button works properly. In fact, if you click on it, it will be pushed, and when you release the mouse button, the on-screen button will be released. The only problem is that when you press the button, you might expect something to happen; but nothing does, because we haven't assigned any action to the mouse-click yet.

Responding to Events

When you press the mouse button on a form or a component, Windows informs your application of the event by sending it a message. Delphi responds by receiving an event notification and calling the appropriate event-handler method. As a programmer, you can provide several of these methods, both for the form itself and for the components you have placed in it. Delphi defines a number of events for each kind of component. The list of events for a form is different from the list for a button, as you can easily see by clicking on these two components while the Events page is selected in the Object Inspector. Some events are common to both components.

There are several techniques you can use to define a handler for the `OnClick` event of the button:

- Select the button, either in the form or by using the Object Inspector's combo box (called the Object Selector), select the Events page, and double-click in the white area on the right side of the `OnClick` event. A new method name will appear, `BtnHelloClick`.
- Select the button, select the Events page, and enter the name of a new method in the white area on the right side of the `OnClick` event. Then press the Enter key to accept it.
- Double-click on the button, and Delphi will perform the default action for this component, which is to add a handler for the `OnClick` event. Other components have completely different default actions.

With any of these approaches, Delphi creates a procedure named `BtnHelloClick` (or the name you've provided) in the code of the form and opens the source code file in that position:

```
procedure TForm1.BtnHelloClick(Sender: TObject);  
begin  
28 |  
end;  
30 |  
end.
```

Even if you are not sure of the effect of the default action of a component, you can still double-click on it. If you end up adding a new procedure you don't need, just leave it empty. Empty method bodies generated by Delphi will be removed as soon as you save the file. In other words, if you don't put any code in them, they simply go away.

note When you want to remove an event-response method you have written from the source code of a Delphi application, you could delete all of the references to it. However, a better way is to delete all of the code from the corresponding procedure, leaving only the declaration and the begin and end keywords. The text should be the same as what Delphi automatically generated when you first decided to handle the event. When you save or compile a project, Delphi removes any empty methods from the source code and from the form description (including the reference to them in the Events page of the Object Inspector). Conversely, to keep an event-handler that is still empty, consider adding a comment to it, like `//`, so that it will not be removed.

Now we can start typing some instructions between the `begin` and `end` keywords that delimit the code of the procedure. Writing code is usually so simple that you don't need to be an expert in the language to start working with Delphi. You can find many tutorials online if you need help or check the bibliography in the appendix of this book.

Of the code below, you should type only the line in the middle, but I've included the whole source code of the procedure to let you know where you need to add the new code in the editor:

```
procedure TForm1.BtnHelloClick(Sender: TObject);
begin
    MessageDlg ('Hello, guys', mtInformation, [mbOK], 0);
end;
```

The code is simple. There is only a call to a function, `MessageDlg`, to display a small message dialog box. The function has four parameters. Notice that as you type the open parenthesis, the Delphi editor will show you the list of parameters in a hint window, making it simpler to remember them.



If you need more information about the parameters of this function and their meanings, you can click on its name in the edit window and press F1. This brings up the Help information. Since this is the first code we are writing, here is a summary of that description (the rest of this book, however, generally does not duplicate the reference information available in Delphi's Help system, concentrating instead on examples that demonstrate the features of the language and environment):

- The first parameter of the `MessageDlg` function is the string you want to display: the message.
- The second parameter is the type of message box. You can choose `mtWarning`, `mtError`, `mtInformation`, or `mtConfirmation`. For each type of

12 - 01: A Form Is a Window

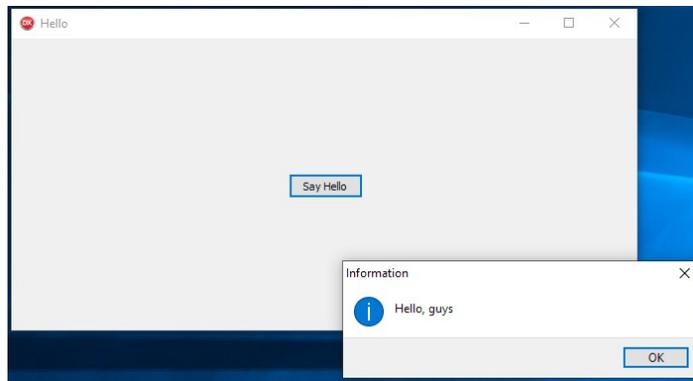
message, the corresponding caption is used and a proper icon is displayed at the side of the text.

- The third parameter is a set of values indicating the buttons you want to use. You can choose `mbYes`, `mbNo`, `mbOk`, `mbCancel`, or `mbHelp`. Since this is a set of values, you can have more than one of these values. Always use the proper set notation with square brackets (`[` and `]`) to denote the set, even if you have only one value, as in the line of the code above.
- The fourth parameter is the help context, a number indicating which page of the Help system should be invoked if the user presses F1. Simply write 0 if the application has no help file, as in this case.

The function also has a return value, which I've just ignored, using it as if it were a procedure. In any case, it's important to know that the function returns an identifier of the button that the user clicked to close the message box. This is useful only if the message box has more than one button.

note Programmers unfamiliar with the Pascal language might be confused by the distinction between a function and a procedure. In Pascal and Delphi, there are two different keywords to define procedures and functions. The only difference between the two is that functions have a return value, while procedures are like “void functions” in C/C++ terms.

After you have written the line of code above, you should be able to run the program. When you click on the button, you'll see the message box shown below:



Every time the user clicks on the push button in the form, a message is displayed. What if the mouse is pressed outside that area? Nothing happens. Of course, we can add some new code to handle this event. We only need to add an `onClick` event to the form itself. To do this, move to the Events page of the Object Inspector and select the form. Then double-click at the right side of the `onClick` event, and you'll

end up in the proper position in the edit window. Now add a new call to the `MessageDlg` function, as in the following code:

```
procedure TForm1.FormClick(Sender: TObject);
begin
    MessageDlg ('You have clicked outside of the button',
               mtWarning, [mbOK], 0);
end;
```

With this new version of the program, if the user clicks on the button, the hello message is displayed, but if the user misses the button, a warning message appears. Notice that I've written the code on two lines, instead of one. The Delphi compiler completely ignores new lines, white spaces, tab spaces, and similar formatting characters. Unlike other programming languages, program statements are separated by semicolons (;), not by new lines.

There is one case in which Delphi doesn't completely ignore line breaks: Strings cannot extend across multiple lines. In some cases, you can split a very long string into two different strings, written on two lines, and merge them by writing one after the other.

Compiling and Running a Program

Before we make any further changes to our Hello program, let's stop for a moment to consider what happens when you run the application. When you click on the toolbar Run button or select Run | Run, Delphi does the following:

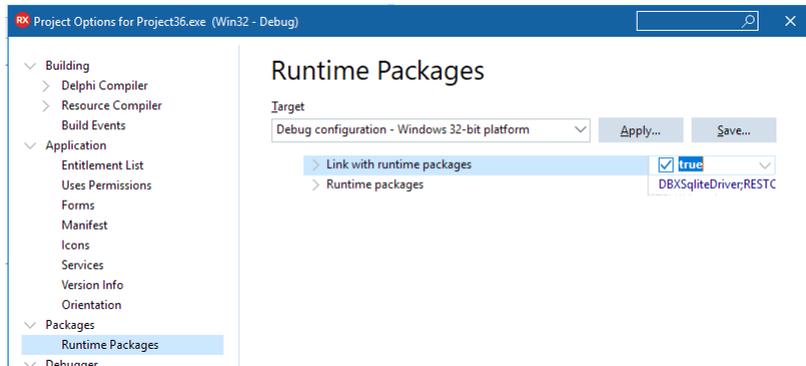
- 1: Compiles the Pascal source code file (.pas) describing the form
- 2: Compiles the project file (.dpr)
- 3: Builds the executable (EXE) file, linking the proper libraries
- 4: Runs the executable file, usually in debug mode

The executable file you obtained by default is a stand-alone executable program with no dependency on library file or a run-time library (as it happens for many other competing tools). Delphi allows you also to link the required libraries code into the executable file, but you can also specify the use of separate run-time packages, making the executable file much smaller but introducing a run-time dependency.

14 - 01: A Form Is a Window

note The fact that Delphi produces standalone executable files implies it does not require run-time compilation of a source code file (like it happens for JavaScript or Python), it doesn't need compilation of intermediate byte-code or IL representation (like C# or Java), and it doesn't even require an execution environment like the .NET run-time or the Java virtual machine. Delphi executable is just binary assembly of the given CPU and platform, desktop or mobile.

The key point is that when you ask Delphi to run your application, it compiles it into an executable file. You can easily run this file from the Windows Explorer or using the Run command on the Start button. Compiling this program as usual, linking all the required library code, produces an executable of about a few hundred Kb (much more with debug information). By using run-time packages, this can shrink the executable to about 20 Kb. Simply select the Project | Options menu command, move to the Packages page, and select the check box Build with run-time packages:



Packages are dynamic link libraries containing Delphi components (the Visual Components Library, for example). By using packages you can make an executable file much smaller. However, the program won't run unless the proper dynamic link libraries (such as vclxxx.bpl) are available on the computer where you want to run the program. The BPL extensions stands for Borland Package Libraries; it is the extension used by Delphi (and C++Builder) packages, which are technically DLL files. Using this extension makes it easier to recognize them (and find them on a hard disk).

note The xxx in vclxxx.bpl stands for the specific version number, such as csl260.bpl for Delphi 10.3.x. Each major version of Delphi is incompatible with libraries for previous versions, and has a new set of run-time packages, with a different number.

If you add the size of this dynamic library to that of the small executable file, the total amount of disk space required by the program built with run-time packages is much bigger than the space required by the bigger stand-alone executable file. For this reason the use of packages is not always recommended. The great advantage of

Delphi over competing development tools is that you can easily choose whether to use the stand-alone executable or the small executable with run-time packages.

In both cases, Delphi executable files are extremely fast to compile, and the speed of the resulting application is comparable with that of a C or C++ program. Delphi compiled code runs much faster than the equivalent code in interpreted or semi-compiled tools (although improvements in JIT – Just-In-Time compilation – technology has reduce the gap) and very fast to start as there is no initial compilation time to incur.

Some users cannot believe that Delphi generates real executable code, because when you run a small program, its main window appears almost immediately, as happens in some interpreted environments.

In the tradition of Borland's Turbo Pascal compilers, the Object Pascal compiler embedded in Delphi works very quickly. For a number of technical reasons, it is much faster than any C++ compiler. One reason for the higher speed of the Delphi compiler is that the language definition is simpler. Another is that the Pascal compilers and linkers have less work to do to include libraries or other compiled source files in a program, because of the structure of units (the compiled DCU file, more on this later in the chapter).

note To be honest the compilers based on the LLVM architecture (that is, most of the non-Windows compilers) are not as fast compiling and significantly slower when linking, as they use more standard techniques of the LLVM architecture and are less optimized.

Changing Properties at Run-Time

Let's return to the Hello application. We now want to change some properties at run-time. For example, we might change the text of HelloButton from *Say hello* to *Say hello again* after the first time a user clicks on it. You may also need to widen the button, as the caption becomes longer. This is really simple. You only need to change the code of the HelloButtonClick procedure as follows:

```
procedure TForm1.HelloButtonClick(Sender: TObject);
begin
    MessageDlg ('Hello, guys', mtInformation, [mbOK], 0);
    btnHello.Caption := 'Say Hello Again';
end;
```

16 - 01: A Form Is a Window

note If you are new to the language, notice that Pascal and Delphi use the `:=` operator to express an assignment and the `=` operator to test for equality. At the beginning, this can be confusing for programmers coming from other languages. For example in C and C++, the assignment operator is `=`, and the equality test is `==`. After a while, you'll get used to it. In the meantime, if you happen to use `=` instead of `:=`, you'll get an error message from the compiler

A property such as `Caption` can be changed at run-time very easily, by using an assignment statement. Most properties can be changed at run-time, and some can be changed only at run-time. You can easily spot this last group: They are not listed in the Object Inspector, but they appear in the Help file for the component (or in its source code). Some of these run-time properties are defined as read-only, which means that you can access their value but you cannot change it.

Adding Code to the Program

Our program is almost finished, but we still have a problem to solve, which will require some real coding. The button starts in the center of the form, but will not remain there when you resize the form. This problem can be solved in two radically different ways.

One solution is to change the border of the form to a thin frame, so that the form cannot be resized at run-time. Just move to the `BorderStyle` property of the form, and choose `bsSingle` instead of `bsSizeable` from the combo box.

The other approach is to write some code to move the button to the center of the form each time the form is resized, and that's what we'll do next. Although it might seem that most of your work in programming with Delphi is just a matter of selecting options and visual elements, there comes a time when you need to write code, of course. As you become more expert (and your applications become larger), the percentage of the time spent writing code will generally increase significantly.

When you want to add some code to a program, the first question you need to ask yourself is *where*? In an event-driven environment, the code is always executed in response to an event. When a form is resized, an event takes place: `OnResize`. Select the form in the Object Inspector and double-click next to `OnResize` in the Events page. A new procedure (a method, to be precise) is added to the source file of the form. Now you need to type some code in the editor, as follows:

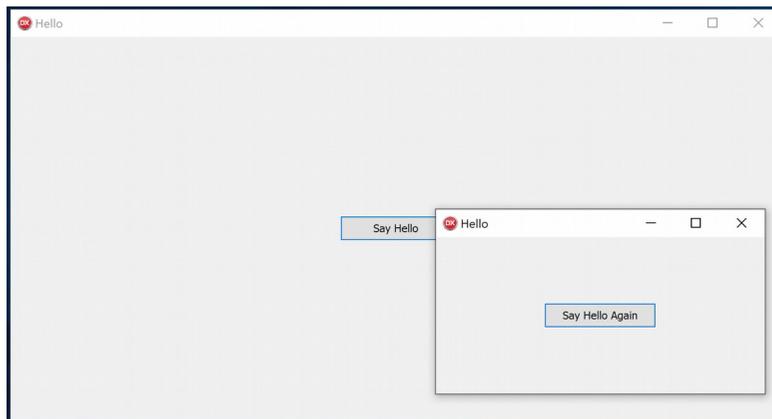
```
procedure TForm1.FormResize(Sender: TObject);  
begin  
    BtnHello.Top := Form1.ClientHeight div 2 -
```

```

    BtnHello.Height div 2;
    BtnHello.Left := Form1.ClientWidth div 2 -
    BtnHello.Width div 2;
end;

```

To set the `Top` and `Left` properties of the button — that is, the position of its upper-left corner — the program computes the center of the form, dividing the height and the width of the internal area or client area of the form by 2, and then subtracts half the height or width of the button. Note also that if you use the `Height` and `Width` properties of the form, instead of the `ClientWidth` and `ClientHeight` properties, you will refer to the center of the whole window, including the caption at the top border. This final version of the example works quite well as you can see below:



This figure includes two versions of the form, with different sizes. By the way, this figure is a real snapshot of the screen. Once you have created a Windows application, you can run several copies of it at the same time by using the Explorer or using Run Without Debugging from the Delphi IDE. By contrast, the Delphi environment can start only one copy of a program in debugging. When you use the Run button to start a program within Delphi, you execute it in the integrated debugger, and the IDE cannot debug two programs at the same time.

note You could possibly start two copies of the Delphi IDE and let each debug a different application, but this is a more advanced use case than I'm planning to cover in this book.

A Two-Way Tool

In the Hello example, we have written three small portions of code, to respond to three different events. Each portion of code was part of a different procedure (actually a method). But where does the code we write end up? The source code of a form is written in a single Pascal language source file, the one we've named `HelloForm.pas`. This file evolves and grows not only when you code the response of some events, but also as you add components to the form. The properties of these components are stored together with the properties of the form in a second file, named `HelloForm.dfm`.

Delphi can be defined as a two-way tool, since everything you do in the visual environment ends up in some code. Nothing is hidden away and inaccessible. You have the complete code, and although some of it might be fairly complex, you can edit everything. Of course, it is easier to use only the visual tools, at least until you are an expert Delphi programmer.

The term two-way tool also means that you are free to change the code that has been produced, and then go back to the visual tools. This is true as long as you follow some simple rules.

Looking at the Source Code

Let's take a look at what Delphi has generated from our operations so far. Every action has an effect — in the Pascal code, in the code of the form, or in both. When you start a new, blank project, the empty form has some code associated with it, as in the following listing.

```
unit Unit1;

interface

uses
  Winapi.Windows, Winapi.Messages, System.SysUtils, System.Variants,
  System.Classes, Vcl.Graphics, Vcl.Controls, Vcl.Forms, Vcl.Dialogs;

type
  TForm1 = class(TForm)
  private
    { Private declarations }
  public
    { Public declarations }
  end;
```

```

var
  Form1: TForm1;

implementation

  {$R *.dfm}

end.

```

The file, named `Unit1`, uses a number of units and defines a new data type (a class) and a new variable (an object of that class). The class is named `TForm1`, and it is derived from `TForm`. The object is `Form1`, of the new type `TForm1`.

Units are the modules into which a Pascal program is divided. When you start a new project, Delphi generates a program module and a unit that defines the main form. Each time you add a form to a Delphi program, you add a new unit. Units are then compiled separately and linked into the main program. By default, unit files have a `.pas` extension and program files have a `.dpr` extension.

If you rename the files as suggested in the example, the code changes slightly, since the name of the unit must reflect the name of the file. If you name the file `HelloForm.pas`, the code begins with

```

| unit HelloForm;

```

As soon as you start adding new components, the form class declaration in the source code changes. For example, when you add a button to the form, the portion of the source code defining the new data type becomes the following:

```

| type
  TForm1 = class(TForm)
    Button1: TButton;
    ...

```

Now if you change the button's `Name` property (using the Object Inspector) to `BtnHello`, the code changes slightly again:

```

| type
  TForm1 = class(TForm)
    BtnHello: TButton;
    ...

```

Setting properties other than the name has no effect in the source code. The properties of the form and its components are stored in a separate form description file (with a `.dfm` extension).

note FireMonkey multi-device applications use a very similar structure, only the textual definition of resources is saved in a file with the `.fmx` extension

20 - 01: A Form Is a Window

Adding new event handlers has the biggest impact on the code. Each time you define a new handler for an event, a line is added to the data type definition of the form, an empty method body is added in the implementation part, and some information is stored in the form description file, too.

It is worth noting that there is a single file for the whole code of a form, not just small fragments for each of the event handlers. This is the complete code of the unit (something I'd generally avoid to list in the book, as it repeats a lot of boilerplate code):

```
unit HelloForm;

interface

uses
  Winapi.Windows, Winapi.Messages, System.SysUtils, System.Variants,
  System.Classes, Vcl.Graphics, Vcl.Controls, Vcl.Forms, Vcl.Dialogs,
  Vcl.StdCtrls;

type
  TForm1 = class(TForm)
    BtnHello: TButton;
    procedure BtnHelloClick(Sender: TObject);
    procedure FormClick(Sender: TObject);
    procedure FormResize(Sender: TObject);
  private
    { Private declarations }
  public
    { Public declarations }
  end;

var
  Form1: TForm1;

implementation

{$R *.dfm}

uses
  System.UITypes;

procedure TForm1.BtnHelloClick(Sender: TObject);
begin
  MessageDlg ('Hello, guys', mtInformation, [mbOK], 0);
  BtnHello.Caption := 'Say Hello Again';
end;

procedure TForm1.FormClick(Sender: TObject);
begin
  MessageDlg ('You have clicked outside of the button',
    mtWarning, [mbOK], 0);
end;
```

```

procedure TForm1.FormResize(Sender: TObject);
begin
    BtnHello.Top := Form1.ClientHeight div 2 -
        BtnHello.Height div 2;
    BtnHello.Left := Form1.ClientWidth div 2 -
        BtnHello.Width div 2;
end;

end.

```

Of course, the code is only a partial description of the form. The source code determines how the form and its components react to events. The form description (the DFM file) stores the values of the properties of the form and of its components. In general, source code defines the actions of the system, and form files define the initial state of the system.

The Textual Description of the Form

As I've just mentioned, along with the PAS file containing the source code, there is another file describing the form, its properties, its components, and the properties of the components. This is the DFM file, a text file with the definition of the configuration you create at design time with the form designer and Object Inspector.

note In the early versions of Delphi the DFM file was a binary file. Now this is by default a text file converted to a binary resource during the compilation process. The binary version is what gets into the executable, because it is a more compact representation and a faster to process one. Whatever the format, if you load this file in the Delphi code editor, it will be converted into a textual description.

In any case, you can determine if the DFM is text or binary for a new module by opening the IDE Tools | Options menu and selecting User Interface | Form Designer going over the Module creation options and using the check box *New forms as text*.

You can open the textual description of a form simply by selecting the shortcut menu of the form designer (that is, right-clicking on the surface of the form at design-time) and selecting the View as Text command. This closes the form, saving it if necessary, and opens the DFM file in the editor. You can later go back to the form using the View as Form command of the local menu of the editor window. The alternative is to open the DFM file directly in the Delphi editor.

To understand what is stored in the DFM file, you can look at the next listing, which shows the textual description of the form of the first version of the Hello example. This is exactly the code you'll see if you give the View as Text command in the local

22 - 01: A Form Is a Window

menu of the form (again, in the book I'll generally include snippets of DFM files, but rarely a complete listing):

```
object Form1: TForm1
  Left = 0
  Top = 0
  Caption = 'Hello'
  ClientHeight = 299
  ClientWidth = 635
  Color = clBtnFace
  Font.Charset = DEFAULT_CHARSET
  Font.Color = clWindowText
  Font.Height = -11
  Font.Name = 'Tahoma'
  Font.Style = []
  OldCreateOrder = False
  OnClick = FormClick
  OnResize = FormResize
  PixelsPerInch = 96
  TextHeight = 13
  object BtnHello: TButton
    Left = 261
    Top = 137
    Width = 113
    Height = 25
    Caption = 'Say Hello'
    TabOrder = 0
    OnClick = BtnHelloClick
  end
end
```

You can compare this code with what I used before to indicate the key features and properties of the form and its components. As you can see in this listing, the textual description of a form contains a number of objects (in this case, two) at different levels. The `Form1` object contains the `BtnHello` object, as you can immediately see from the indentation of the text. Each object has a number of properties, and some methods connected to events (in this case, `OnClick`).

note Once you've opened this file in Delphi, you can edit the textual description of the form, although this should be done with extreme care. As soon as you save the file, it will be turned back into a binary file. If you've made incorrect changes, this compilation will stop with an error message, and you'll need to correct the contents of your DFM file before you can reopen the form in the editor. For this reason, you shouldn't try to change the textual description of a form manually until you have a good knowledge of Delphi programming.

An expert programmer might choose to work on the text of a form for a number of reasons. For big projects, the textual description of the form is a powerful documenting tool, an important form of backup (in case someone plays with the form, you can understand what has gone wrong by comparing the two textual versions),

and a good target for a version control tool. For these reasons, Delphi also provides a DOS command-line tool, `CONVERT.EXE`, which can translate forms from the compiled version to the textual description and vice versa. As we will see in the next chapter, the conversion is also applied when you cut or copy components from a form to the Clipboard.

The Project File

In addition to the two files describing the form (`PAS` and `DFM`), a third file is vital for rebuilding the application. This is the Delphi project file (`DPR`). This file is built automatically, and you seldom need to change it, particularly for small programs. If you do need to change the behavior of a project, there are basically two ways to do so:

- You can use the Delphi Project Manager and set some project options
- You can manually edit the project file directly

This project file is really a Pascal language source file, describing the overall structure of the program and its start-up code:

```
program Hello;

uses
  Vcl.Forms,
  HelloForm in 'HelloForm.pas' {Form1};

{$R *.res}

begin
  Application.Initialize;
  Application.MainFormOnTaskbar := True;
  Application.CreateForm(TForm1, Form1);
  Application.Run;
end.
```

You can see this file with the Project | View Source menu command (historically it was View | Project Source). As an alternative, you can select the project node in the Project manager and use the View Source option of the local menu.

What's Next

In this chapter, we created a simple program, added a button to it, and handled some basic events, such as a click with the mouse or a resize operation. We also saw

24 - 01: A Form Is a Window

how to name files, projects, forms, and components, and how this affects the source code of the program. We looked at the source code of the simple programs we've built, although some of you might not be fluent enough in Object Pascal to understand the details.

In the next chapter we'll start exploring the Delphi IDE in a more systematic way, going over the various features it has. After an overview chapter, the following ones will delve into very specific areas, like the form designer, the editor, and the debugger.

02: Highlights Of The Delphi IDE

In a visual programming tool such as Delphi, the role of the environment is certainly important, and the various tools help you get work done faster. After the introduction in the last chapter, this second part offers a deeper overview of the IDE and its features. Now in some cases the topics just introduce deeper coverage in following chapters, while in others there isn't much more to say.

This chapter won't discuss all of the features of Delphi, but it will give you the overall picture and help you to explore some of the environment traits that are not obvious, while suggesting some tips that may help you. You'll find more information about specific commands and operations throughout the book.

Delphi IDE Foundations

There is a lot to write about the Delphi IDE and I want to start offering a little history and covering the different versions, the different personalities and the start-up command line parameters you can use.

Different Versions of Delphi

Before delving into the details of the Delphi programming environment, let's take a side step to underline two key ideas. First, there isn't a single version of Delphi; there are two of them, with some variations:

- The Professional edition is aimed at professional developers building stand-alone applications or simple database ones (the FireDAC data access library is included, with limited client/server support). The Professional package has limitations in multi-tier development, but offers the full set of controls for UI development on desktop and mobile.
- The Community Edition (CE) has the same features of the Professional edition, from a technical point of view, but comes with a limited license:
 - You can use it only if you or your company makes less than 5,000 USD/year in revenues (covering students, hobbyist, retired people, start-ups, and more)
 - You can only install a maximum of 5 copies on your local network (notice that educational institution looking to install many copies on a lab can use the Academic versions, which are free or have a nominal fee)
- The Enterprise edition is aimed at developers building client/server and multi-tier applications. It includes all FireDAC drivers for most Enterprise level relational databases, DataSnap and RAD Server multi-tier architectures, and support for the Linux target.

Besides the different editions available, there are a number of ways to customize the Delphi environment. You can change the buttons of the toolbar, attach new commands to the Tools menu, hide some of the windows or elements, and resize and move all of them. You can also install a large number of different IDE add-ins, a few of which are made available directly by Embarcadero and I'll mention in the book.

A Short History of the Delphi IDE

The original Delphi IDE, that worked under Windows 3.1, provided the groundwork for all following versions right up to Delphi 7. Around that time, Borland decided to redesign the IDE to open it up to multiple programming languages (or personalities, as they were later called), with the goal to support Microsoft .NET development (a feature later abandoned), and to integrate several Application Lifetime Management (ALM) tools they had bought and were focusing on at the time.

The so-called “Galileo IDE” project is the foundation of all recent Delphi IDEs, from Delphi 8 up to the current Delphi 10 series. This IDE also went over different naming changes, partially reflected in some of the folders along with company names changes.

In terms of company, Borland formed a division called CodeGear and put in on sale. A company called Embarcadero acquired it, only to be alter bought by Idera, Inc. Idera has a large number of products, and Delphi is part of the Developers Tools division which includes Embarcadero along with other development tools. Each tool retains its originally brand and web site, like for Delphi www.embarcadero.com, while overall company information can be found at www.ideracorp.com.

In terms of the product name, Delphi has always been called Delphi, with C++Builder being its sibling product since the early days. Since debut of the “Galileo IDE”, it has been possible to combine the two – and additional products – into a single application, which is also sold as a combined product. For Delphi 2005 and Delphi 2006 the IDE was called Borland Developer Studio (hence the *BDS* name, which remains the name of the IDE application, *bds.exe*), while from the 2007 version and until today is has been called RAD Studio.

A Matter of “Personality” (-p flag)

As mentioned since the introduction of the “Galileo” IDE, RAD Studio encompasses multiple personalities. If you have a Delphi or C++Builder license, you can install only the matching personality, while if you own a RAD Studio license you can decide to install both. The installer lets you decide which personality and also which target platforms to install, and you can combine than with some flexibility – but not completely at will.

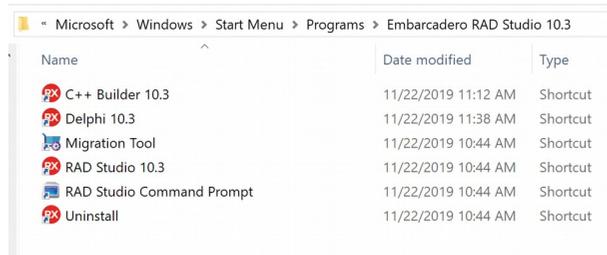
note The RAD Studio product over time included additional personalities beside Delphi and C++, including C#. In other cases additional languages (like PHP and HTML) were supported but not bundles in the IDE – they required installing a separate development environment.

28 - 02: Highlights of the Delphi IDE

If you own a RAD Studio license and have installed multiple personalities (that is, both Delphi and C++ compilers for at least some platforms) you can still decide to run the IDE by activating a single personality by using the `-p` flag followed by the name of the personality you want to use, like for example for my installation:

```
"C:\Program Files (x86)\Embarcadero\Studio\20.0\bin\bds.exe" "-pDelphi"
```

If you create a shortcut with this parameter, you can easily start the specific personality using it. Actually the RAD Studio installer already provides similar links in the Embarcadero RAD Studio folder of the Windows Start Menu:



Installation Folders

Original Delphi versions were installed under the *Program Files\Borland* folder. With changes in the product ownership, product name, and the need to support Windows folder permissions, the overall structure has changed considerably.

The main installation folder for 10.3, using the defaults on an English language version of Windows, is:

```
"C:\Program Files (x86)\Embarcadero\Studio\20.0"
```

Under this main folder there are many others. The most notable are :

- `bin` for all Win32 binaries, including the IDE, compilers, run-time packages and many utilities
- other `binxyz` folders include binaries in other formats, like Win64, Linux and macOS
- `lib` includes compiled library files (mostly in `dcu` format, but not only)
- `source` has extensive run-time libraries source code

Other files are installed outside of the *Program Files* section of the disk, because they are meant to be created or modified by the user. These additional folders are under the *Users* section of the disk and some of them can be under the individual user or the *Public* user depending on installation options. On an English language version of Windows, and using the defaults, the new projects folder, the examples folder, the catalog repository, the FireDAC database configuration are respectively:

```
C:\Users\marco\Documents\Embarcadero\Studio\Projects
C:\Users\Public\Documents\Embarcadero\Studio\20.0\Samples
C:\Users\marco\Documents\Embarcadero\Studio\20.0\CatalogRepository
C:\Users\Public\Documents\Embarcadero\Studio\FireDAC
```

Registry Settings and Using Alternative Configurations (-r flag)

Beside the Additional configuration information RAD Studio save a considerable amount of data in the registry. The default data for the registry is saved under *HKEY_LOCAL_MACHINE* (notice the *WOW6432Node*):

```
|| HKEY_LOCAL_MACHINE\SOFTWARE\WOW6432Node\Embarcadero\BDS\20.0
```

The first time the IDE is started, the configuration is copied under:

```
HKEY_CURRENT_USER\Software\Embarcadero\BDS\20.0
```

This is where your configuration is kept with you change IDE options, install packages, and the like. If you know what you are doing, you can tweak some of these settings in the registry directly.

You can also create a new snapshot of the configuration settings, by starting the IDE with the *-r* flag. This can be handy as you can keep different configurations active at the same time, for example with different IDE setting and also different third party packages.

You can also create a stripped down version of Delphi (for faster start-up and reduced memory consumption) whilst keeping the full version ready to use as needed.

Technically, the Delphi *-r* command line parameter specifies the base registry key to use. For example, create a shortcut like this:

```
|| "C:\Program Files (x86)\Embarcadero\Studio\20.0\bin\bds.exe" -rSmall
```

The first time you run it, Delphi creates a brand new set of registry keys, copying the default settings (not the current ones) into:

```
|| HKEY_CURRENT_USER\Software\Embarcadero\Small\20.0
```

In other words, the name you provide replaces the BDS name in the registry tree.

tip If you want to export the registry keys for the current configuration, or an additional one, and merge it with another configuration or move it to a different computer, you can leverage the Settings Migration Tool included with Delphi (*migrationtool.exe* in RAD Studio bin folder).

30 - 02: Highlights of the Delphi IDE

In addition to the `-r` parameter and the `-p` parameter, there are other command line parameters you can use when starting Delphi IDE, including:

- ns (no splash): disables the display of the splash screen
- np (no page): disables the Welcome page
- b (build): opens and builds the project passed as parameter

Delphi IDE Overview

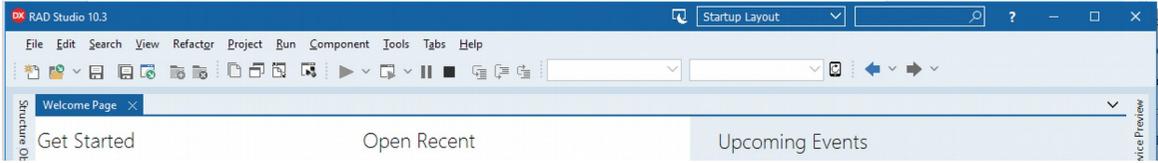
Now that we have the basics of the configuration, folders, and start-up options, let's have a first look of the Delphi IDE overall, highlighting some global features. The first time you start the IDE, it asks you if you want use the light or dark mode:



This is a configuration you can easily change later, so don't worry about your initial selection. I'll start right away covering the style configurations and the desktop configurations, and how you can change them.

As the Delphi IDE starts it has many area, but the most notable section to start with are those at the top:

- The special toolbar commands hosted in the title bar
- The menu bar
- The customizable toolbar

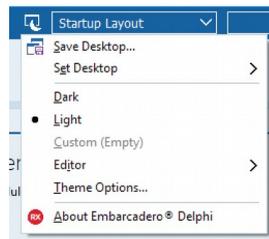


A Light or Dark IDE

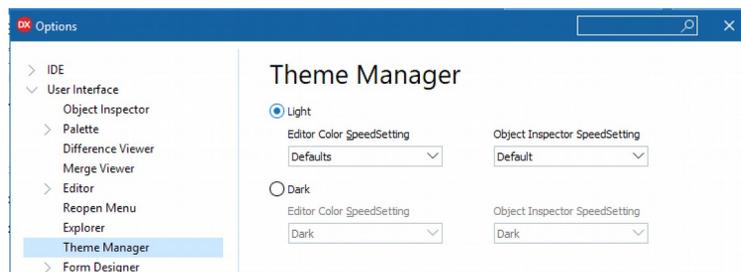
The Delphi IDE offers two distinct configurations, Light Theme and Dark Theme. You can pick one in the initial configuration, as in the image in the previous section, and change the active theme any time.

note Both because this is my preference and because very dark images are more complex to print with good quality, I'm going to stick with using the Light Theme for most images in this book.

The easier way to change the theme is to use the corresponding icon with the moon in the menu bar:



As you can see in this menu you can immediately pick the light or dark configuration, select a matching set of editor colors with the *Editor* submenu or use *Theme Options* to open the Theme Manager page of IDE Options dialog, where you can configure the editor and Object Inspector color configuration associated with the Light and Dark themes.



32 - 02: Highlights of the Delphi IDE

The other two items in this menu (Save Desktop and Set Desktop) can be used along with the desktop selection combo box also in the title bar to activate and configure desktop settings, covered in the next section.

Desktop Settings

Programmers can customize the Delphi IDE in various ways—typically, opening many windows, arranging them, and docking them to each other. However, you'll often need to open one set of windows at design time and a different set at debug time. Similarly, you might need one layout when working with forms and a completely different layout when writing components or low-level code using only the editor. Rearranging the IDE for each of these needs is a tedious task.

For this reason, Delphi lets you save a given arrangement of IDE windows (called a desktop setting) with a name and restore it easily. For each IDE status you save multiple desktop settings and pick a default one:

- The *Startup Desktop* is selected when the Delphi IDE starts and when no project is active (for example, after you select the File | Close All menu). The default option for this desktop configuration is called *Startup Layout*.
- The *Default Desktop* is selected when a project is active in the IDE and you are editing or using the form designers. The default option for this desktop configuration is called *Default Layout*.
- The *Debug Desktop* is selected when you start an application and debug it. The default option for this desktop configuration is called *Debug Layout*.

You can modify any of these configuration and override these *layouts* or add new ones with new names and define the default one for each IDE status.

Desktop settings are saved in files with the DST extension, which are INI files in disguise. These files are saved in a folder with the version number under C:\Users\\AppData\Roaming\Embarcadero\BDS\.

The saved settings include the information about the main window, the Project Manager, the Object Inspector, the editor windows (with the status of the Code Explorer and the Message View), and many others, plus the docking status of the various windows.

Here are some excerpts from a DST file, which should be easily readable:

```
[Main window]
PercentageSizes=1
Create=1
visible=1
```

```
Docked=0
State=0
...

[ProjectManager]
PercentageSizes=1
Create=1
Visible=1
Docked=1
StayOnTop=0

[MessageView]
PercentageSizes=1
Create=1
Visible=0
Docked=1
StayOnTop=0

[ToolForm]
PercentageSizes=1
Create=1
Visible=1
Docked=1
StayOnTop=0

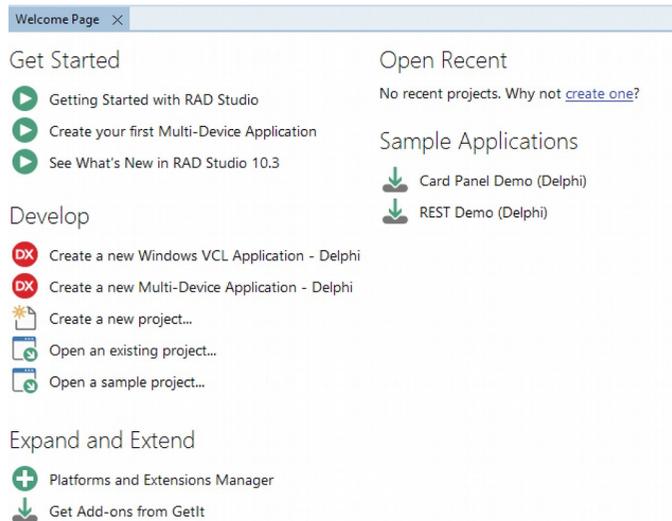
[PropertyInspector]
PercentageSizes=1
Create=1
Visible=1
Docked=1
StayOnTop=0
SplitPos=111
```

The Welcome Page

As you run the IDE, your starting point will be the Welcome page. The Welcome page is a pane hosting Internet Explorer and allowing you to view some pertinent information. The Welcome page has common operations and a list of recent projects you have worked with (empty below), alongside with some direct links to some Sample Applications.

On the side there are panels with lists of coming events and YouTube videos from the *EmbarcaderoTechNet* channel.

34 - 02: Highlights of the Delphi IDE



tip If you want to customize the Welcome page, you can modify some of its configuration files available in the `welcomePage` folder under the main RAD Studio installation folder. There you can find the core HTML and CSS files, along with the images displayed, plus JavaScript code and more.

The IDE Overall Structure

Going back to the structure of the Delphi IDE, notice that on the left and right side of the main pane (with the Welcome page, the editor and the form designer) there are several other windows hosting various panes.

By default (and this is something you can fully customize) there are the following main panes available:

- The Structure View (on the left, above) hosts a tree view with the components and controls on the current designer or the structure of the code in the open unit, depending on which of the two is active
- The Object Inspector (on the left, below) shows details of the currently selected component in the designer and can be used to modify the components. It also shows some information about elements in the project manager.
- A tabbed windows (on the right, above) hosts multiple views:
 - The Project Manager with a tree view structure with the current project (or multiple projects if a project group is active)

- The Model View shows UML modeling information when the feature is active for a project – something that was more popular time ago and now fairly neglected and seldom used
- The Data Explorer shows information about the predefined database connections and let's you explore, query and modify databases
- The Multi-Device Preview helps configuring the mobile and desktop previews for FireMonkey applications
- The Tool Palette (on the right, below) hosts the palette of available components when a designer is active, and lists the entries in the New Items dialog in other cases.

Another window at the bottom of the IDE generally hosts compiler results, search results, and a refactoring previews, but also breakpoints lists, event view and other debug panes. This window is displayed when necessary. There are also dozens of other windows used by the IDE, including a number of debugger views. I'll cover these various windows and panes when discussing related topics in this and the coming chapters.

IDE Insight

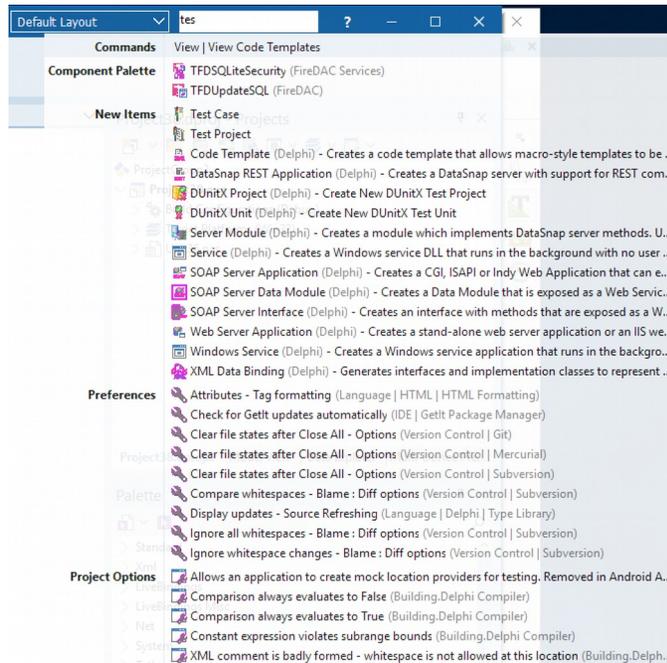
Both newcomers and expert users can easily get lost in the large number of menu items, settings, components, and features you can activate in the IDE. At times even experts get lost because features were moved from a version of Delphi they spent a lot of time with. That's why it is great to have searching capabilities in several dialog boxes and an overall search mechanism for the entire IDE, called “IDE Insight”. The IDE Insight search box is visible in the title bar, between layouts management and the Help button:



Beside clicking it with the mouse, you activate typing in this box by pressing the F6 key (or by using Ctrl + <period>). As you start typing, a pull down will show a filtered list of just about anything you might want to look for in the IDE:

As you can see above, the results are filtered by category and they can encompass many different areas of the development environment – some of which depend on the current selection (editor, form designer, start-up layout):

36 - 02: Highlights of the Delphi IDE



- **Commands** of the main menu of the IDE, including those added dynamically in the Tools menu or by Wizards or extensions of any kind (but the menu items of local popup menus)
- **Component Palette** elements, where the current view is a visual designer, like a form or a data module.
- **Components** used by the current designer, again where the current view is a visual designer. Components depend on the installed packages, and obviously include third-party ones.
- **Code Templates**, where the current view is an Object Pascal source code editor, a C++Builder editor, or any other editor supporting code templates.
- **Desktop Setting**, usually managed with the corresponding toolbar of the main form, the one with the small combo box.
- **Files** include the list of files of the current project (and other projects in the group) and is available only if a project is active in the Project Manager.
- **Forms** filters the forms and designers of the current project, again only if a project is active.
- **New Items** has elements of the New Items dialog box.
- **Open Files** provides fast access to any file currently open in the editor.

- **Preferences** filters on the individual elements of the IDE preferences (the Tools | Options dialog box) and will open the corresponding page of the dialog box when selected.
- **Project Options** does the same with the options of the current project (again, you need to have a project open). Finding project options by typing their names is a superb feature I'm using a lot.
- **Projects** let's you jump to a project of the current project group.
- **Recent Files** and **Recent Projects** filter the recently closed source code files and projects (which in Delphi 2010 can be customized much more than in the past, as we'll see in the section "Many More Recent Files").

What is less intuitive to figure out is that you can use wild cards when typing in this search box (and most other search boxes available in the IDE):

- **?** will match any single character
- ***** will match zero, one, or more characters

Notice that an implicit ***** is automatically added both at the beginning and at the end of the search text to match sub strings. The same wild cards work in most of the other filtered search dialog boxes added to the IDE.

Asking for Help

The next element of the environment I want to mention is the Help system. There are basically two ways to invoke the Help system: select the proper command in the Help pull-down menu, or choose an element of the Delphi interface or a token in the source code and press F1.

When you press F1, Delphi doesn't search for an exact match in the Help Search list. Instead, it tries to understand what you are asking. For example, if you press F1 when the text cursor is on the name of the `TButton` component in the source code, the Delphi Help system automatically opens the description of the `TButton` class, since this is what you are probably looking for. This technique also works when you give the component a new name. Try naming the button `Foo`, then move the cursor to this word, press F1, and you'll still get the help for the `TButton` class. This means Delphi looks at the contextual meaning of the word for which you are asking help.

You can find almost everything in the Help system, but you need to know what to search for. Usually this is obvious, but at times it is not. Spending some time just

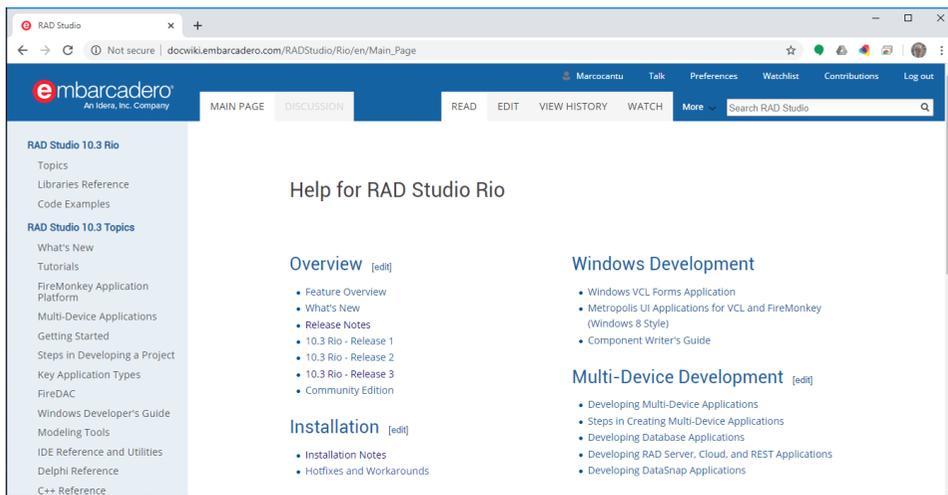
38 - 02: Highlights of the Delphi IDE

playing with the Help system will probably help you understand the structure of these files and learn how to find the information you need.

The Help files provide a lot of information, both for beginner and expert programmers, and they are especially valuable as a reference tool. They list all of the methods and properties for each component, the parameters of each method or function, and similar details, which are particularly important while you are writing code.

As an alternative you can refer to the online product documentation, powered by a Wiki engine, at

http://docwiki.embarcadero.com/RADStudio/en/Main_Page



tip The online version of the documentation on DocWiki gets updated more frequently than the version installed in the product.

Delphi Menus and Commands

There are basically three ways to issue a command in the Delphi environment:

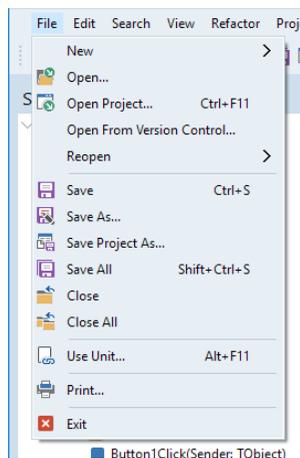
- Use the main menu

- Use the (customizable) toolbar
- Use one of the local menus activated by pressing the right mouse button in the various panes
- Use IDE Insight as explained in the section “IDE Insight” earlier in this chapter

The Delphi menus offer many commands and the IDE is very rich of features. While it might look to be boring, in this section I'll go over each sub-menu of the main menu and the various items as this gives me a very good way to offer an overview of the features of the IDE. In the following sections, I'll present some suggestions on the use of some of the menu commands. In some cases I'll just mention a feature which is detailed in a later chapter and refer to the deeper coverage elsewhere.

The File Menu

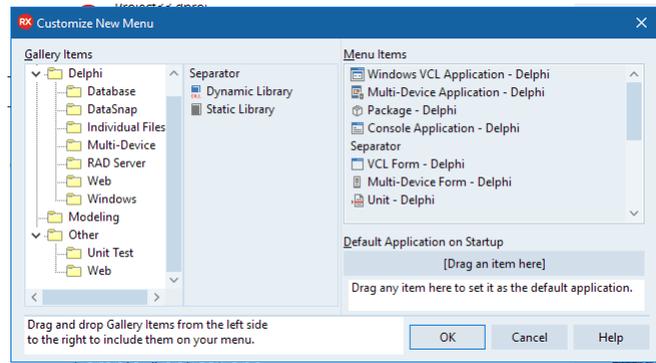
Our starting point is the File pull-down menu. The structure of this menu has kept changing from version to version of Delphi, with menu items for handling specific types of projects added and removed over time. Still, this menu contains commands that operate on projects and commands that operate on source code files and offers options for opening files, creating new ones, saving and closing.



The File | New Sub-Menu

The File | New sub-menu offers different options for common operations and the ability to open the File | New | Other dialog, also known as Object Repository. The basic options are those that tend to change over time. Currently (in Delphi 10.3) the menu offers by default the ability to create the following new items:

- *Windows VCL Application* creates a standard, empty project for the Windows platform only, based on the classic VCL library
- *Multi-Device Application* creates a FireMonkey application for desktop and mobile platforms. You can select one of a few predefined structures (with headers, footers, tabs, and more) or start with an empty form, or *Blank Application*.
- *Package* creates a components package or a package hosting IDE extensions or other features. Packages are a slightly more advanced topics than we can really cover in this book.
- *Console Application* creates a text-based console app you can use for different operating systems (including Linux, if you have the Enterprise version). A console app can just use direct interaction with the use via standard text input and output, and is often used for writing test or small utilities. A console application starts with basic, skeleton code.
- *VCL Form*, *Multi-Device Form*, and *Unit* create a new standard alone Pascal source code file or add a new one to the current project (if one is active). If a project is open, however, only the compatible elements are visible (that is, if you are working on a VCL project you won't see *Multi-Device Form* menu item). In each case the new Pascal source code file (unit) will have a standard basic structure for forms or will be empty if you pick a plan unit.
- *Other* opens the File | New | Other dialog, or Object Repository.
- *Customize* allows you to change the entries of the File | New menu adding new entries you use often or removing the current ones. The dialog to customize the File | New menu looks like the following (but actual content depends on your edition of RAD Studio or Delphi:

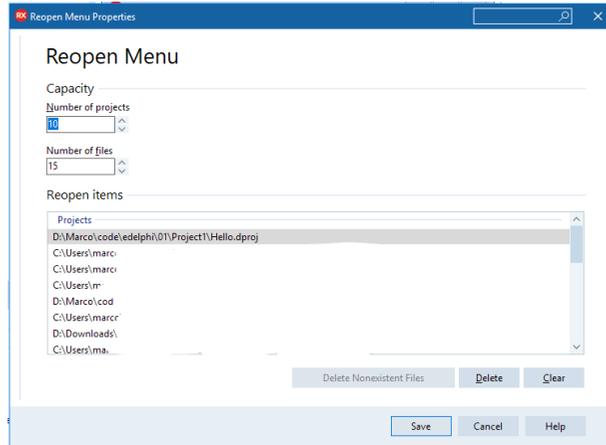


The File | Open Commands

Let's now get back to the main File menu. There are 4 open commands:

- *Open* can be used to open any file, including units, projects, project groups, but also plan text files, INI files, configuration files, HTML files or any other file a text editor can handle. Opening a new file generally doesn't affect the current open project or open files. Notice that there are different commands for adding an existing unit to the current project.
- Open From Version Control allows you to open a project from a remote repository in a Version Control System like Subversion, Git or Mercurial.
- *Open Project* can be used to open an exiting project, replacing the currently open project if any. Again, there is an alternative option which is add a new project to the existing project group, keeping both (or many) projects open ant the same time.
- Reopen allows you to open a recently closed project or file. When you select File | Reopen you see a list of recently closed files and projects. You can also customize how many of these files are kept in each group and do some cleanup in the list by using the File | Reopen | Properties menu, which leads to the following dialog box:

42 - 02: Highlights of the Delphi IDE



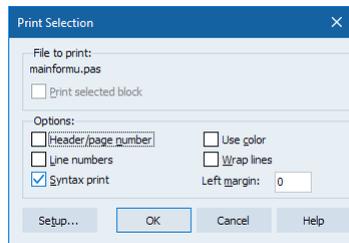
The Other File Menu Commands

Saving files and projects is quite straightforward, with the menu commands *Save*, *Save As*, *Save Project As*, *Save All*. Notice that if you save a project, Delphi will prompt you to save the existing units first, and give a name to any newly created one.

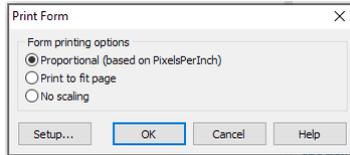
The Close and Close All commands are self-explanatory. Notice, however, the local menu of editor tabs offers additional closing operations, like closing all files save for the current one or all those to the left and right (considering that position generally depends on the opening sequence).

The following File menu command, *Use Unit*, offers the ability to add a reference from the current unit to another unit in the project (with a `uses` statement). The interesting point here is that this operation will let you add references to components in the other unit at design time in the IDE, for example connecting a visual component to a data source in another unit, like a Data Module.

Another peculiar command is Print. If you are editing source code and select this command, the printer will output the text with syntax highlighting as an option:



If you are working on a form and select Print from the File menu, however, the printer will produce the graphical representation of the form, offering these options:



Finally, there is the *Exit* menu item, which prompts you for saving any open file or project and shuts down the IDE.

tip It is worth noting that it is not fully recommended to keep a very complex software like Delphi running for many days in a row. Shutting it down from time to time cleaning up all memory and resources is a good recommendation.

The Edit Menu

The Edit menu has some typical operations, such as Undo and Redo, and the Cut, Copy, and Paste commands. The menu adapts to the context somehow (startup, editor, designer, etc). For example, when you work with the editor, the first command of this pull-down menu is Undo; when you work with the form, it becomes Undelete instead.

note Undo in the form designer is not enabled because a large number of component properties cause side effects that cannot really be reverted. Consider the active property of a database query, which opens a connection, loads meta data, populate field definitions, creates fields and fetches some records... how do you undo it? Compared to other IDEs where properties only set individual fields, the fact Delphi frameworks are more rich and offer live data at design time is the reason a general Undo operation is extremely complex to define. As a partial solution, there is an editor history that keeps multiple versions of the source code and designer files.

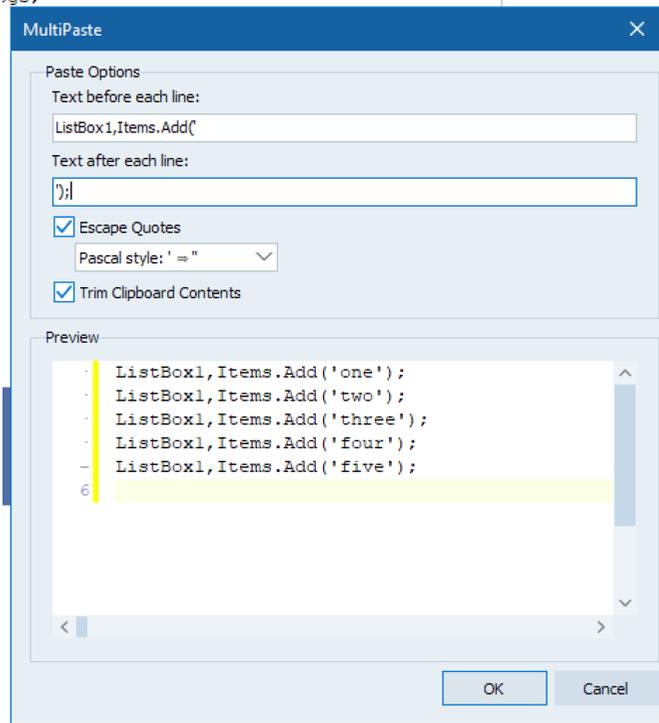
The copy and Paste operations (and the standard Ctrl+X, Ctrl+C, and Ctrl+V keyboard shortcuts) work both with text and with form components, as covered in the next section.

Besides using cut and paste commands, the Delphi editor allows you to move source code by selecting and dragging words, expressions, or lines. If you drag text while pressing the Ctrl key, it will be copied instead of moved.

There are also additional special operations for the code editor, like MultiPaste, which allows you to add the same modify multiple lines adding the same text before each of them and after each of them. This is handy for embedding the text of a SQL

44 - 02: Highlights of the Delphi IDE

statement in a string or adding multiple lines to a list box (like in the case in the image below, in which the selected text has just numbers), just to mention two examples.



Copying and Pasting Components

What you might not have noticed is that you can also copy components from the form to the editor and back. Delphi places components in the Clipboard along with their textual description. You can even edit the text version of a component, copy the text to the Clipboard, and then paste it back into the form as a new component.

For example, if you place a button on a form, copy it, and then paste it into an editor (which can be Delphi's own source code editor or any word processor), you'll get the following description:

```
object Button1: TButton
  Left = 112
  Top = 80
  Width = 75
  Height = 25
```

```

Caption = 'Button1'
TabOrder = 0
end

```

note Actually Delphi adds to the clipboard both the textual description of a component and an image of the form. If the target editor support graphics, as you paste the content you'll get the option to pick one or the other.

Now, if you change the name of the object, caption, or position, or add a new property, these changes can be copied and pasted back to a form. Here are some sample changes:

```

object MyButton: TButton
  Left = 200
  Top = 200
  Width = 180
  Height = 60
  TabOrder = 0
  Caption = 'My Button'
  Font.Name = 'Arial'
end

```

Copying the above description and pasting it into the form will create a button in the specified position with the caption *'My Button'* in an *Arial* font. To make use of this technique, you need to know how to edit the textual representation of a component, what properties are valid for that particular component, and how to write the values for string properties, set properties, and other special properties. When Delphi interprets the textual description of a component or form, it might also change the values of other properties related to those you've changed, and change the position of the component so that it doesn't overlap a previous copy. You can see how Delphi modifies the properties of the component by copying it back to the editor. For example, this is what you get if you paste the text above in the form, and then copy it again into the editor:

```

object MyButton: TButton
  Left = 200
  Top = 200
  Width = 180
  Height = 60
  Caption = 'My Button'
  Font.Charset = DEFAULT_CHARSET
  Font.Color = clWindowText
  Font.Height = -11
  Font.Name = 'Arial'
  Font.Style = []
  ParentFont = False
  TabOrder = 1
end

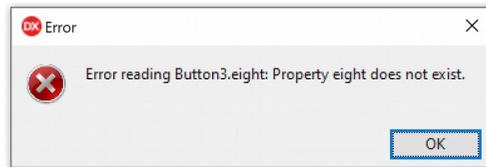
```

46 - 02: Highlights of the Delphi IDE

As you can see, some lines have been added automatically, to specify other properties of the font. Of course, if you write something completely wrong, such as this code:

```
object Button3: TButton
  Left = 100
  eight = 60
end
```

which has a spelling error (a missing ‘H’), and try to paste it into a form, Delphi will show an error indicating what has gone wrong:



You can also select several components and copy them all at once, either to another form or to a text editor. This might be useful when you need to work on a series of similar components. You can copy one to the editor, replicate it a number of times, make the proper changes, and then paste the whole group into the form again.

More Edit Commands

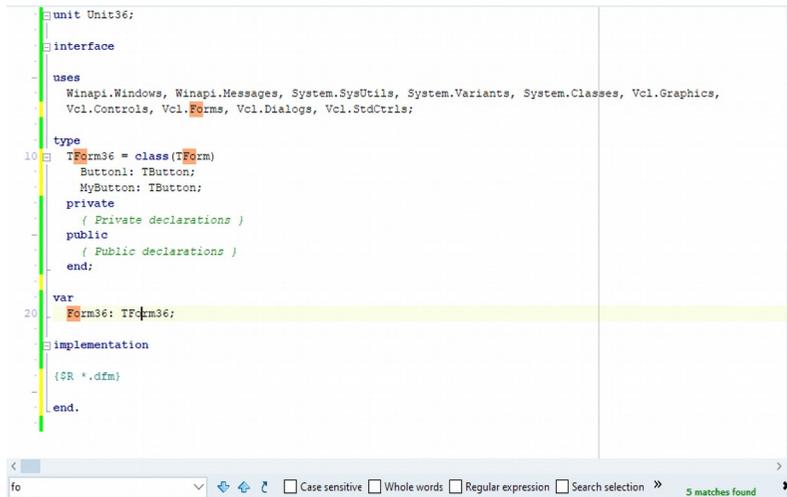
Along with the typical commands found on most Edit menus in Windows applications, Delphi includes a number of commands that are mostly related to forms. The specific operations for forms can also be accessed through the form shortcut menu (the local menu you can invoke with the right mouse button) and will be covered in the next chapter.

One command not replicated in a form’s local menu is Lock Controls, which is very useful for avoiding an accidental change to the position of a component in a form. For example, you might try to double-click on a component and actually end up moving it. Since there is no Undo operation on forms, protecting from similar errors by locking the controls after the form has been designed can be really useful.

The Search Menu

The Search menu offers some alternative methods for finding text or logical element of your program and eventually replace them. The basic Search | Find operation (Ctrl+F) opens a search pane at the bottom of the editor window. Here you can type your search, press enter and the editor will highlight all the hits, indicating the

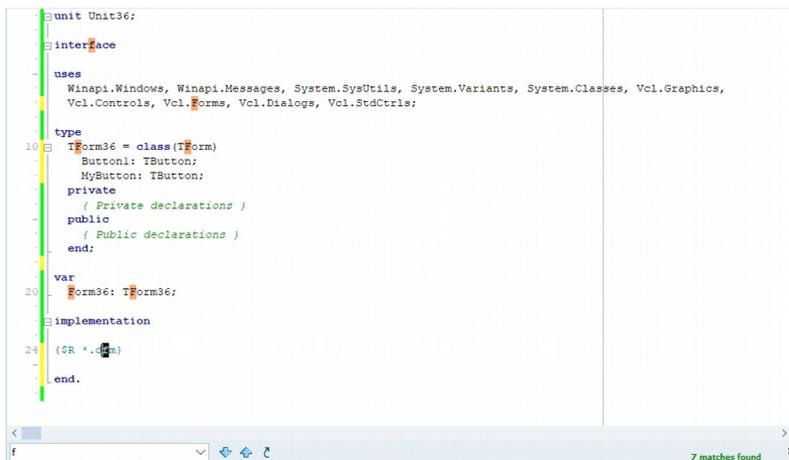
number of matches. You can navigate the results with the arrows and modify some of the search options with the matching check boxes.



```
unit Unit36;
interface
uses
Winapi.Windows, Winapi.Messages, System.SysUtils, System.Variants, System.Classes, Vcl.Graphics,
Vcl.Controls, Vcl.Forms, Vcl.Dialogs, Vcl.StdCtrls;
type
10 TForm36 = class(TForm)
    Button1: TButton;
    MyButton: TButton;
    private
        { Private declarations }
    public
        { Public declarations }
    end;
20 var
    Form36: TForm36;
implementation
{$R *.dfm}
end.
```

fo Case sensitive Whole words Regular expression Search selection 5 matches found

A quicker way to search is to use Search | Incremental Search (Ctrl+E) which moves to the closest match as you type, with no need to press enter to activate the search:



```
unit Unit36;
interface
uses
Winapi.Windows, Winapi.Messages, System.SysUtils, System.Variants, System.Classes, Vcl.Graphics,
Vcl.Controls, Vcl.Forms, Vcl.Dialogs, Vcl.StdCtrls;
type
10 TForm36 = class(TForm)
    Button1: TButton;
    MyButton: TButton;
    private
        { Private declarations }
    public
        { Public declarations }
    end;
20 var
    Form36: TForm36;
implementation
24 {$R *.dfm}
end.
```

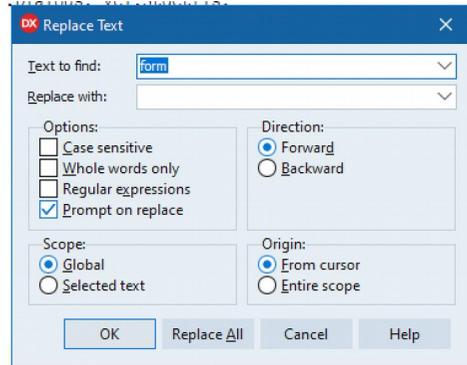
f 7 matches found

Incremental search has fewer options, and you can always switch back to the regular search mode if you need more control.

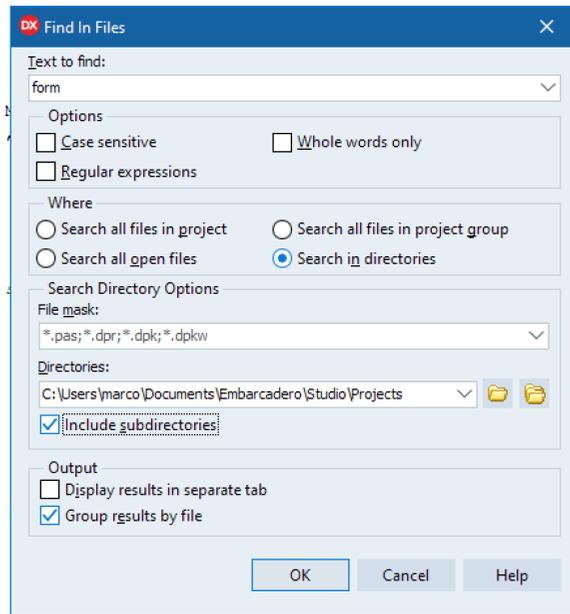
note Originally, the two search modes had a more clear difference, with Find opening a dialog and Incremental search using the editor toolbar. Since they were both modified to use a pane at the bottom of the editor, they have become similar in their user interface, but the behavior remains distinct. In general, Incremental search remains faster to use.

48 - 02: Highlights of the Delphi IDE

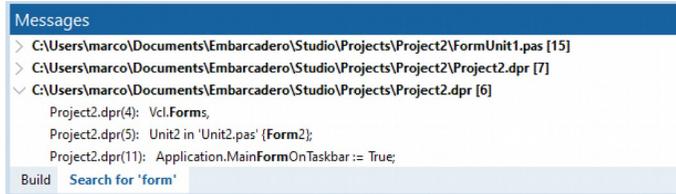
A third and more classic approach is used by the Search | Replace command, which opens a dialog for entering the text to find, the replacement text and some options:



All of the find commands above work only on the active editor file. For a broad search you can use the Search | Find in Files command. This allows you to search for a string in all of the source code files of a project, all the open files, or all the files in a directory (optionally including its sub-directories), depending on the radio button you check in the *Where* group:



The result of the search will be displayed in the message area at the bottom of the editor window. You can select an entry to open the corresponding file and show the specific matches, and then jump to the line containing the text by clicking on it:



A common case is using the Find in Files command to search for components, classes, and type definitions in the VCL source code.

There are a few configuration options for the Find in Files search. You can request that Delphi shows the results in a different page as you do a new search, so the results of previous search operations remain available. You can also use a check box in the Find Text dialog box to group the search results by source code file, as in the scenario above. Once you have many pages, you can press the Alt+Page Down and Alt+Page Up key combinations to cycle through the tabs of this window. (The same commands work for other tabbed views, but not all of them.)

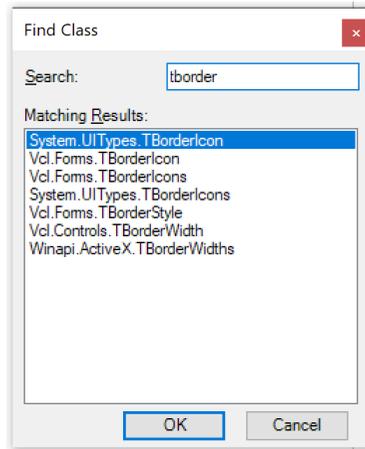
Syntactic Searches

In the Search menu there are also a few options that allow you to search specific elements of the code, rather than pure text matches as the operations covered above.

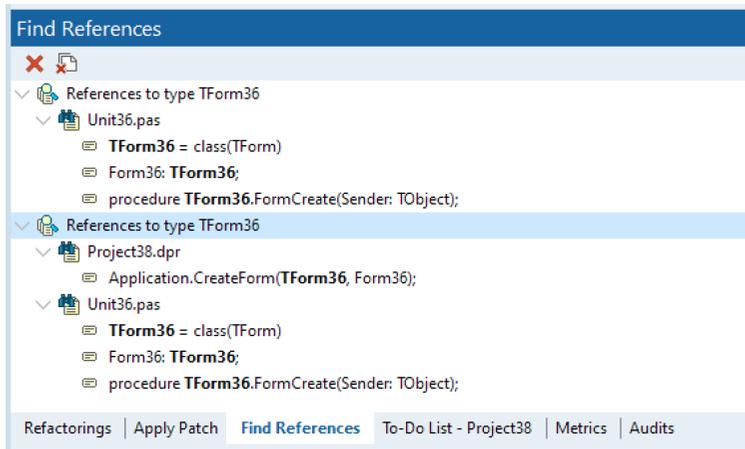
Find Class is a bit of a misnomer, as it allows you to search for data types in general (indicating the unit a class is into). Given this search is based on the active project, this encompasses only units included in the project (directly or indirectly via a uses statement). This is an example of the user interface:

Notice that this search work only with full match, that is if you omit the initial *T* in the search above you won't get any of those results. This search will open the unit with the definition of the symbol (something you can also achieve by using *Ctrl+Mouse click* on the same symbol in the editor.

50 - 02: Highlights of the Delphi IDE



Find References and Find Local References provide a list of the locations where the symbol selected in the editor appears within the current project or only within the current unit, and displays them at the bottom of the editor screen. Here you can use the local references (above) and all of them (below) for the `TForm36` symbol:



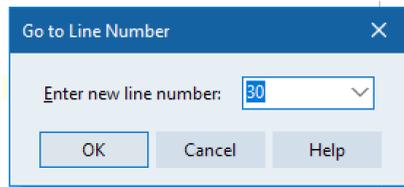
The difference compared to a text search is that it distinguishes comments, text in strings, from the use of a specific syntax element of the code. The drawback is that the code needs to be correct (from the compiler perspective) for the search to produce results.

Finally, Find Original Symbol moves you to the location in code where the symbol is declared (in the local unit only, though).

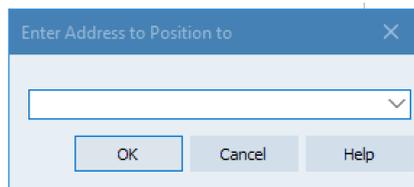
The Goto Commands

Continuing in the Search menu there is IDE Insight (something I've already covered in details earlier in this chapter) and two *Goto* Command:

- The Goto Line Number dialog lets you enter the line number (for the current unit) and jumps to it. This is handy if you know the number you are interested in and the file is fairly long (so that scrolling to it is going to take extra time):



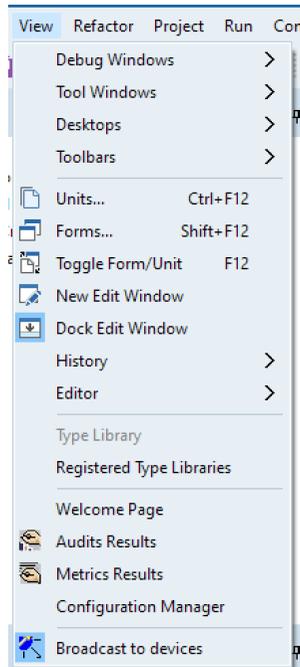
- The Goto Address command might seem strange at first. It can be used only while debugging the application and after you have compiled it, to find the source code line corresponding to memory address of the compiled code. This is an information that some error messages and exceptions show in their display information. (In fact, this Delphi menu item was originally called Find Error). Often, however, the error is not in one of the lines of your code, but in a line of library or system code; in this case the Goto Address command cannot might not be able to locate the offending line. This is the user interface:



The View Menu

Most of the View menu commands can be used to display one of the windows and panes of the Delphi environment, such as Project Manager, the Breakpoints list, or the Object Inspector. Some of these panes are active by default, others make sense only in specific scenarios. As this menu was getting too tall to fit on the screen, it has been rearranged a few versions back to group the view commands by area:

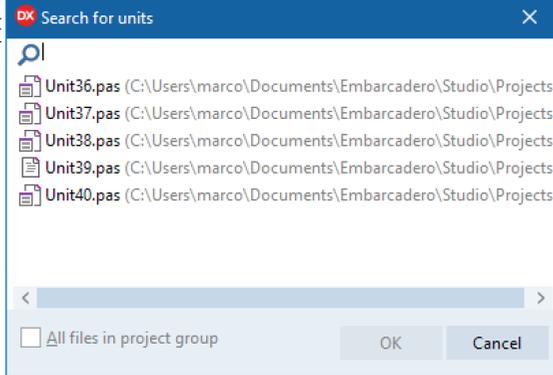
52 - 02: Highlights of the Delphi IDE



- Debug Windows activates debugger panes (most of them active by default in the debug desktop configuration), which are covered in Chapter 8.
- Tool Windows activates the various panes of the IDE that are used while designing forms or editing code.
- Desktops offers the same options for managing desktop settings available in the corresponding title bar menu and covered in the “Desktop Settings” section earlier in this chapter.
- Toolbars allows you to customize the toolbar by enabling various sections, with options matching the toolbar local menu and covered in the section “The Delphi Toolbar” later in this chapters

The commands on the second section of the View menu are commonly used, which is why they are also available on the default toolbar. The Toggle Form/Unit (or F12) command is used to move between the form you are working on and its source code in the editor. The Units and Forms commands let you select one of the units or forms in the current project – something you can also do using the Project Manager. The dialog boxes activated by these menu items have a local search option:

tip You can search for



The New Edit Window command opens a second edit window (along with the corresponding designer). It is the only way to view two source code files side by side in Delphi, since the editor uses tabs to show the multiple files you can load. Each edit window has its own set of tabs and open files. Notice that you can drag a tab from one edit window to the other to re-arrange their position (while you cannot open the same file in both tabs).

The following section has two menu items for working with type libraries, which is a fairly advanced area of Delphi.

The following four commands on the View menu can be used to activate the Welcome page (in case you closed it), the audit and metrics views (a topic I won't be covering in the book) and the Configuration manager for project groups, a feature related with project management and covered in Chapter 6.

Finally the last menu item, Broadcast to device, activates the ability to send the content of a design time form to your mobile device or desktop, using the matching LivePreview application. If you don't plan using this feature, you should disable it, given it opens a communication port for the purpose you probably don't need and don't want to keep active.

The Refactor Menu

The next menu is fully focused on the refactoring support of the Delphi IDE. While not updated in a while, this is a collection of nice features that help you write and modify your source code. Refactoring is covered in details in Chapter 5, so I won't go over the various menu command here.

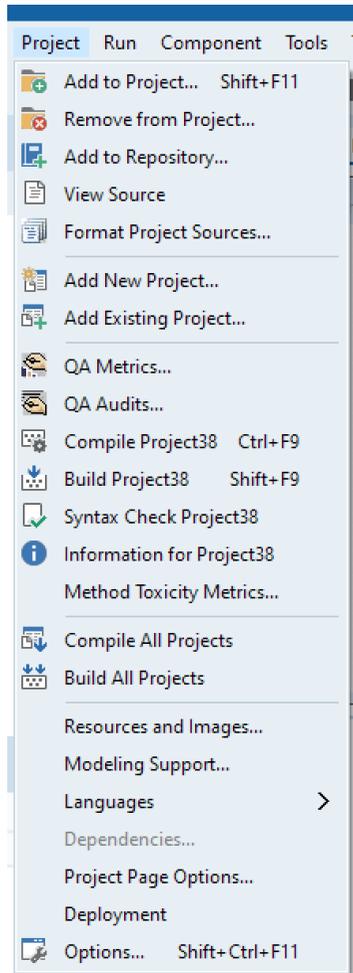
The Project Menu

The next pull-down menu, Project, has commands to manage a Delphi project and compile it. A number of these menu items (particularly those for adding and removing elements of the project and those for building) are also available in the local menu of the Project Manager window. Project management is explained in more detail in Chapter 6.

The first section of the menu has these commands:

- Add to Project opens an Open File Dialog box to select an existing unit to be added to the active project
- Remove from Project offers a list of the project units and allows you to pick one to remove.
- Add to Repository offers a mechanism for adding a project to the Object Repository, to be used as starting point for a future project. This is a seldom used feature, these days.
- View Source opens the main project file source code (we saw an example in the section “The Project File” towards the end of Chapter 1).
- Format Project Sources offers the ability to format the entire source code of all of the units of the project. Automatic formatting is covered in Chapter 4.

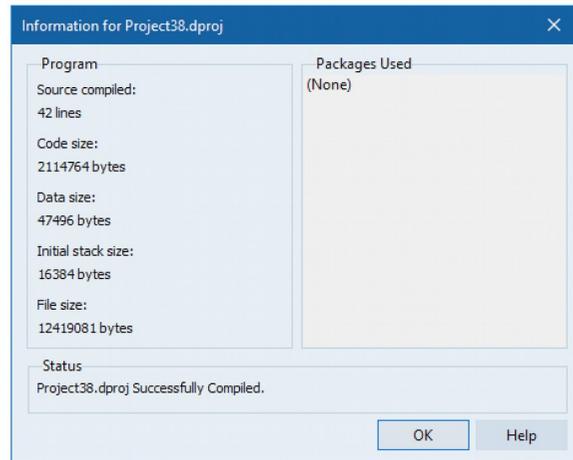
The second section has two menu items, Add New Project and Add Existing Project that work at the project group level (Delphi can keep multiple projects open at the same time in a project group). This is also covered in Chapter 6.



Next there is a section with the main commands for compiling and building the current project (the different is that Compile – called Build in other system – works only on the modified units, while Build – also called Build All – repeats the entire process from scratch). Syntax Check does a quick scan of the code, with the compiler parsing it but skipping code generation. Information for gives summary data of a the last compiled project:

Finally, Audits, Metrics, and Toxicity Metrics offer different types of information about the code after analyzing it.

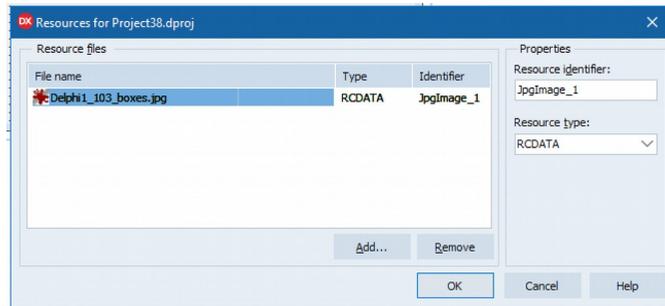
56 - 02: Highlights of the Delphi IDE



The next section has two commands for the entire project group, Compile All Projects and Build All Projects. I'll provide some more information on how to customize their behavior in Chapter 6.

The final section of the Project menu has a number of unrelated commands:

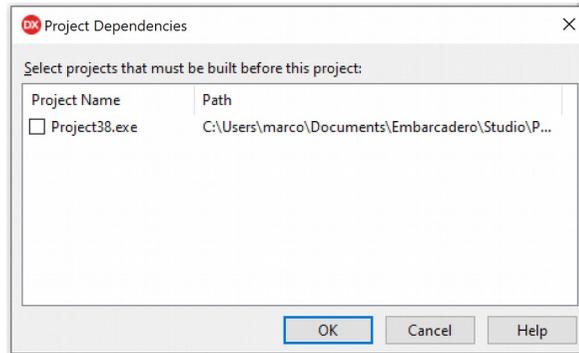
- Resources and Images can be used to add additional external images and other resource files to a project, so that these resources are bundled to the executable file:



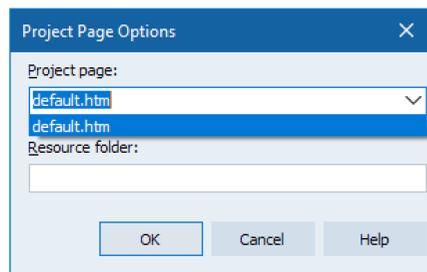
- Modeling Support activates the UML-related modeling features that are (still) part of Delphi – with features available only in the Enterprise edition.
- Languages sub-menu allows you to add support for localizing a VCL application in different languages (this doesn't work for FireMonkey). The localization technology in Delphi is fairly old and currently “not supported” so I won't cover it in this book.

02: Highlights of the Delphi IDE - 57

- Dependencies can be used to define dependencies for projects within a project group, defining which projects need to be built before a given project (notice you can also define the build sequence in a project group):



- Project Page Options allows you to pick an HTML file already included in the project as the page to be displayed when the project opens (a very rarely used feature, to my knowledge):



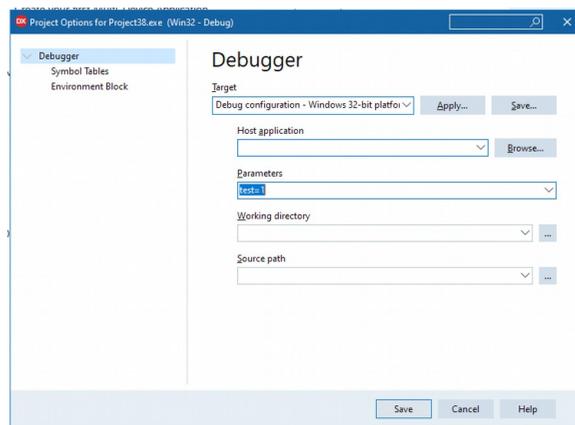
- Deployment opens the Deployment dialog box. In a VCL application, this will list run-time packages your application depend on, and you can add additional files. As you later do the Deploy operation, these will all be copied in a destination folder. For FireMonkey and on mobile the same feature allows you to assemble all of the files needed to deploy on a device or on an application store.
- Options is used to set compiler and linker options, application object options, and so on. We will discuss project options in detail in Chapter 9 but also in other sections of the book in the context of related topics.

The Run Menu

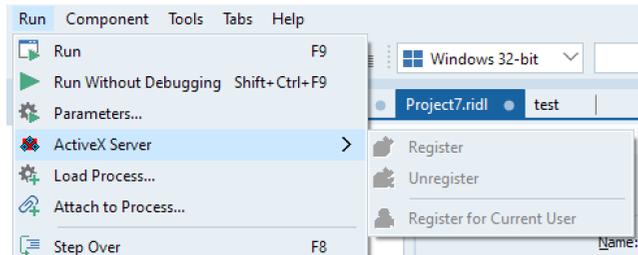
The Run menu could have been named Debug as well. Most of its commands are related to debugging, including the Run command itself. When you run a program within the Delphi environment, you execute it under the integrated debugger (unless you disable the corresponding Environment option). The Run command and the corresponding toolbar icon are among the most commonly used commands, since Delphi automatically re-compile a program before running it — at least if the source code has changed. Simply hit the F9 key to compile and run a program.

As an alternative, the Run without Debugging menu allows you run the program outside of the debugger. This makes the program faster to start and makes it behave more like an end user would execute it. Operations the debugger intercepts, like exceptions for example, won't trigger special processing if you run the program outside of the debugger.

The next command, Parameters, can be used to specify parameters to be passed on the command line to the program you are going to run, and to provide the name of an executable file that is loading your compiled code, when you want to debug a DLL or a package:



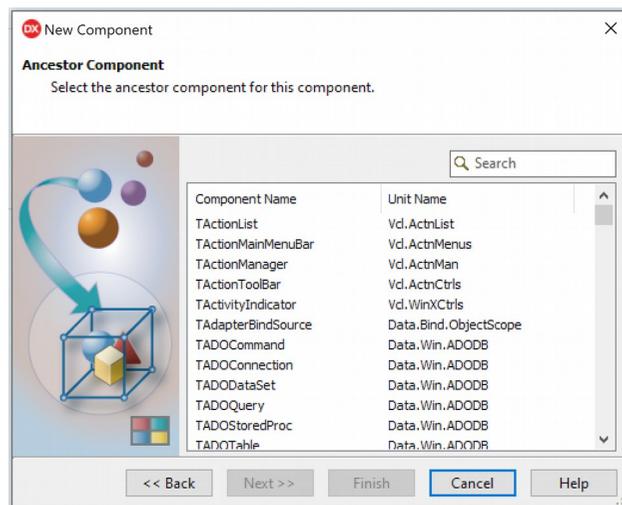
In the same initial section of the menu there are also commands to attach the debugger to a running application and detach it. What you can do is start a program without debugging (or from Explorer) and later *attach* the debugger to it. Finally, the first section has a sub-menu for ActiveX Server operations, which are in fact COM Server operations, including registration and the like. The Register ActiveX Server and Unregister ActiveX Server commands basically add or remove the Windows Registry information about the ActiveX control defined by the current project. Notice that these operations require to run the IDE with elevated permissions.



The following section has commands used during debugging, to execute the program step by step, trace into, stop debugging, set breakpoints, inspect the values of variables and objects, and so on. Some of these debugging commands are also available directly in the editor local menu and debugging in general is covered in Chapter 8.

The Component Menu

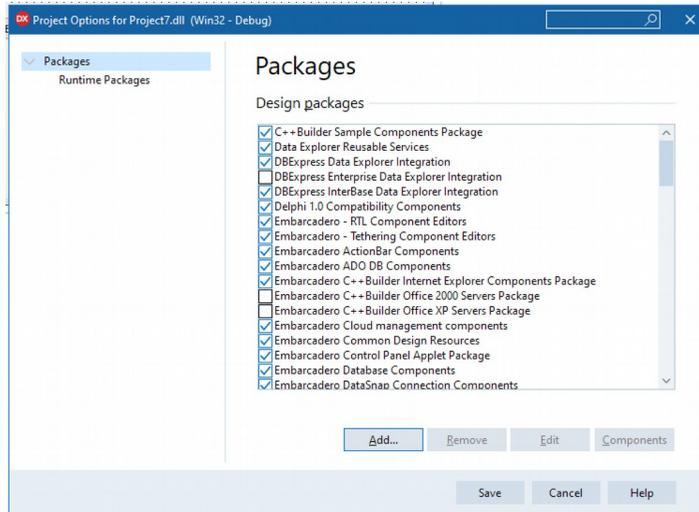
The commands of the Component menu can be used to write components, add them to a package, or to install packages in Delphi. The New Component command invokes a Component Wizard:



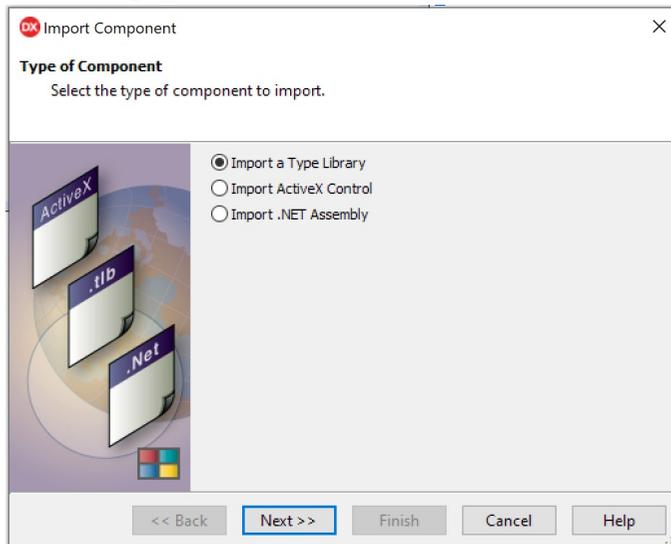
This wizard shares some logic with the Install Component wizard, which is used to add a unit with an existing component to a package.

60 - 02: Highlights of the Delphi IDE

The commands of the last section perform quite distinct operations. Install Packages opens the package configuration options for the current project (a similar dialog exists for the entire IDE as whole) and allows you to enable or disable registered packages or add a new existing, compiled package (.bpl) to the project or IDE:



Import Component (which was originally and more appropriately called Install ActiveX Library) can be used to create a Delphi component encapsulating a type library, An ActiveX server or COM server, or a .NET assembly respectively, using the options in the first page of the wizard:



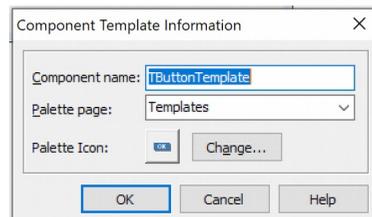
Finally, Import WSDL opens a wizard to create an interface for a SOAP server by importing the server definition from a local file or online URL.

There is also a stand-alone command in this menu, Create Component Template, which is used to create a very simple pseudo-component by copying the configuration of an existing component. This is explained in the following sub-section.

Using Component Templates

Suppose you want to create a brand new application, with a similar button and a similar event handler to a program you have written, like the Hello program of Chapter 1. It is possible to copy a component to the Clipboard, and then paste it into another form to create a perfect clone. However, doing so you copy only the properties of the component, and not the events associated with it.

Delphi allows you to copy one or more components, and install them as a new component template. This way, you also copy the code of the methods connected with the events of the component. Simply open the Hello example, or any other one, select the component you want to move to the template (or a group of components), and then select the Component | Create Component Template menu command. This opens the Component Template Information dialog box, shown here:



Here you enter the name of the template, the page of the Component Palette where it should appear, and a custom icon. By default, the template name is the name of the first component you've selected followed by the word template. The default template icon is the icon of the first component you've selected, but you can replace it with an icon file. The name you give to the component template will be used to describe it in the Tools palette.

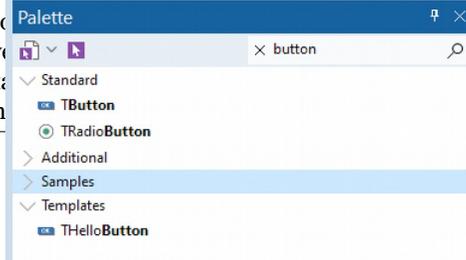
In this case you might want to call it `THelloButton` and after installing it you'll find the following entry among your components:

If you select it, the IDE will place the component template in the current form and you'll get a component with the given properties and also with the event handler's code attached to it as in the original version.

62 - 02: Highlights of the Delphi IDE

tip

All the information about a component is stored in a hidden file and there is apparently no way to retrieve it. If you modify it, you can instantiate the component in the previous definition.

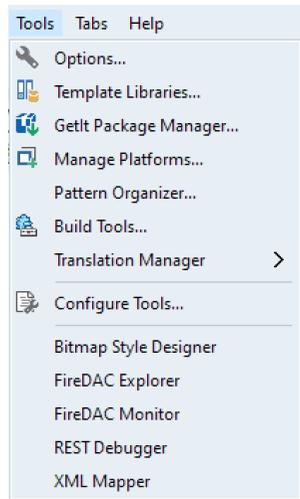


hidden file and there is apparently no way to retrieve it. If you add it to a project and give it the same name and overriding

In general terms, using Component Templates is a nice trick that can save you some time and effort, but they have been mostly superseded by frames, which extend and expand on the concept with much more power and flexibility.

The Tools Menu

The Tools menu offers some global Delphi IDE configuration options (in the first section) and links to external programs and tools (in the third section) that can be configured with the Configure Tools menu item:

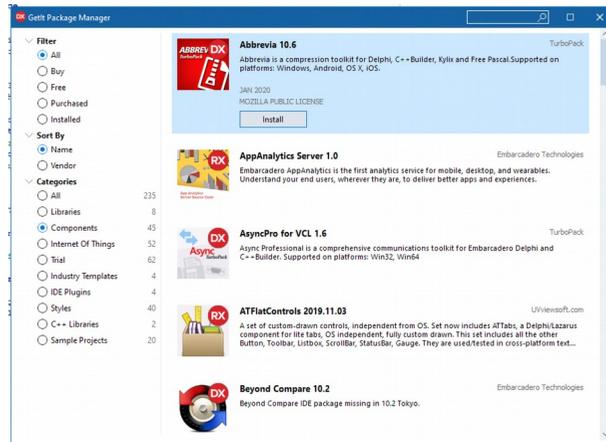


The first command, Options, opens the extremely detailed and complex Options dialog (originally called Environment Options dialog) of the Delphi IDE. The dialog box has many pages related to generic environment settings, packages and library settings, many editor options, a page to configure the Tools Palette, one for the Object Inspector, and one of the new Code Insight technology. Discussing those options goes beyond the scope of this chapter, and many of them are covered in different sections of this book.

02: Highlights of the Delphi IDE - 63

The Template Libraries command allows you to customize the Object Repository (part of the content of the File | New | Other dialog box) and covered in Chapter 6.

The GetIt Package Manager command opens the GetIt Package manager dialog box, where you can search for additional components, IDE plug-ins, demos, libraries, styles, and additional Delphi features provided by Embarcadero and third parties. From there you can install them in a seamless way:



Manage Platforms is available for Delphi installations made via the (default) Online Installer, and allows you to add platforms (meaning operating systems targets) you didn't install at start up and additional features and options (like the Help file, the product demos, unit testing, charting controls, InterBase Developer Edition and more). The content of this dialog varies from release to release.

Patterns Organizer opens a repository of coding patterns tied with UML modeling support integrated with Delphi and not very commonly used these days.

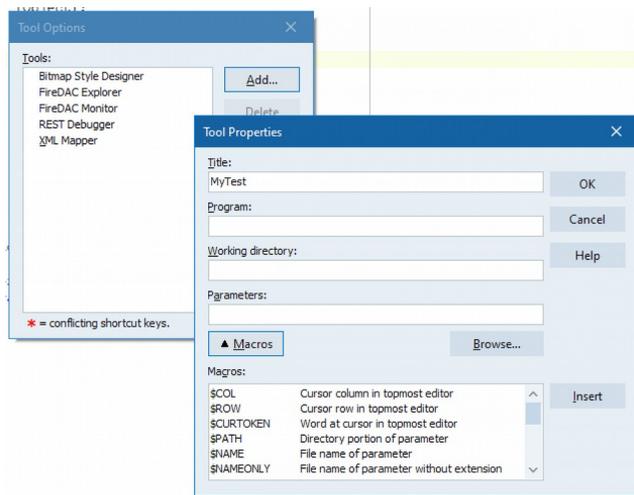
Build Tools offers the ability to configure external tools that can be invoked as part of the build process to perform any possible external action before the project is compiled, during compilation phases, or after the linker has completed generating the binary files. This integration allows you to configure build operations without the need to use an external build system – but external build systems are in general more powerful.

Translation Manager allows to configure the now deprecated VCL translation support (I already mentioned this as part of the Project | Languages menu item).

Moving to the second part of the menu, the Configure Tools command can be used to add new entries in the final part of the menu, with the ability to invoke any external executable and passing it some parameters that might depend on the current

64 - 02: Highlights of the Delphi IDE

active project. Complex parameter can be built by clicking the Macros button in the lower part of the Tool Properties dialog box:



The Tabs Menu

Like the Windows menu of a good old MDI application, the Tabs menu lists all “*windows*” open in the IDE, that is all tabs open in the main editor window. It also suggests the shortcuts you can use for circling over tabs left to right or right to left (Ctrl+Tab and Ctrl+Shift+Tab).

The menu also lists available *floating* windows, if you are not using a docked IDE configuration.

note Using the *undocked IDE* is not recommended and might be formally deprecated in the future.

The Help Menu

The Help menu can be used to get information about Delphi from the local help installed along with the product (if you selected that option) or from online resources:

- Delphi Help and its sub-menus open local help files, if installed

- RAD Studio DocWiki open the online wiki (already displayed in the section *Asking for Help* earlier in this chapter)
- Third-Party Help opens additional help files for Third Party tools, if available
- Platform SDK Help sub-menus have links to online documentation for the Microsoft Windows API and the Apple macOS platform-specific

The following section has a few links to online pages including Delphi product page and Embarcadero website. This is followed by a menu item opening the external License Manager application, which helps you install additional product licenses and check if they are properly configured and updated with your latest Update Subscription renewals. The Welcome Configuration command opens the dialog displayed at the first execution of the product.

Finally, the Help | About menu items displays the Delphi About box. In this window, you can see the product release version, information about your license, installed third party tools and more. You can also type some *not-so-hidden* key combinations to see can see a list of the people involved in building Delphi.

note The Help menu was often populated by third-party Delphi Wizards (before their default location became the Tools menu).

The Local Menu

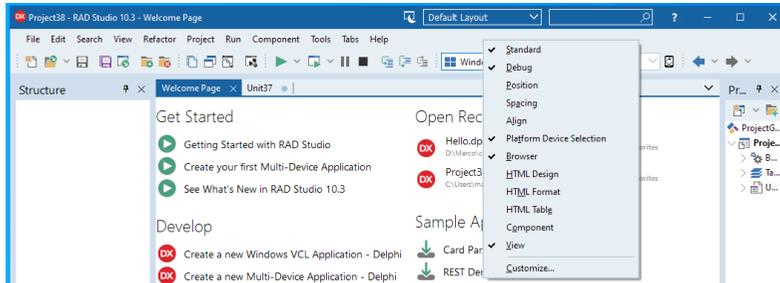
Although Delphi has a good number of menu items as we have seen in the sections above, not all of the commands are available through the pull-down menus. In other words, a large number of the main menu commands are available also in local menus of specific windows and panes, but often local menus offer additional commands not in the main menu. We'll explore some of the local menu while exploring various windows of the IDE in coming chapters.

The Delphi Toolbar

The most commonly used menu items of the main menu are also available in the toolbar and in the title menu bar. What is important to notice is that the Delphi IDE toolbar is fully customizable.

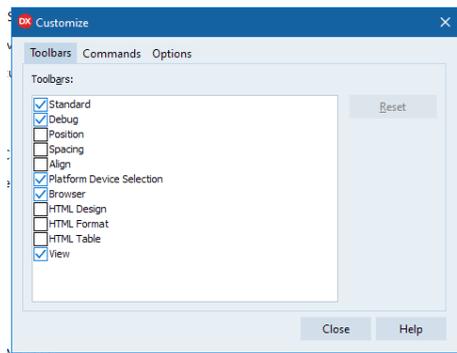
66 - 02: Highlights of the Delphi IDE

First, it is made of multiple areas (that is, multiple toolbar sections) and you can select which ones are active, by using the toolbar local menu and selecting the various elements:



Second you can drag the various sections of the toolbar to new positions, by dragging the separator with 5 vertically align dots on the left of each toolbar section.

Third you can fully customize the toolbar by re-arranging the items and adding others not initially available. The toolbar customization is done via a specific fairly complex dialog box. As you open View | Toolbars | Customize of the Customize menu items of the toolbar local menu, you see a first page with the various sections and check boxes to activate them:



This is similar to what we can achieve with the local menu of the toolbar shown earlier. If you move to the commands page however, you can now select one on the many categories of commands (or actions) and drag them to one of the toolbar sections to add a new item. While that dialog is visible, you also move items from one



area of the toolbar to a different one, or move them off, deleting them. Here I've added to new buttons (Options and Build Tools) at the right end of the toolbar:

What's Next

In this chapter, I've offered you an overview of the Delphi IDE and a fairly detailed analysis of its menus and the features they activate. Starting with the next chapter I'm going to focus on specific activities you use the IDE for, like designing the user interface of an application with the form designer (Chapter 3) or writing code in the editor (Chapter 4).

From there I'll continue getting to more advanced features the Delphi IDE includes, like code templates and projects management.

03. Using The Form Designer

Given Delphi is a component-based RAD tool, it expected that developers write code in its editor but also interact a lot with its visual designer. This can be used to work on different type of “designer surfaces”:

- Forms are the most common type of designer surfaces, and the tool is in fact generally known as Form Designer. Notice that there are significant differences when working on working on a VCL form in the designer or a FireMonkey form.
- Data Modules are containers of non-visual components, like database connection objects or other types of configuration and data access components
- Frames are some sort of panels that can be hosted by forms, offering a mechanism to replicate a similar design

I have already guided you step by step on how to add a component to the form designer and set its properties in the Object Inspector in Chapter 1. Rather than covering the basics, here I'm going over more detailed and non-obvious information about the various activities related with designing forms.

The VCL Form Designer

When you start a new, blank project, Delphi creates an empty form, and you can start working with it. You can also start with an existing form (using various templates available). A project (an application) can have any number of forms.

It is important to remember two key elements of Delphi's architecture:

- Every component you place on a form (or data module or frame) and every property you set is stored in a file describing the form, a `dfm` or `fmx` file, that is bundled as part of the executable as a resource
- Every operation you do at design time (from adding a component to setting properties to defining event handlers) can also be done at run-time in the application code, even if this might be less convenient

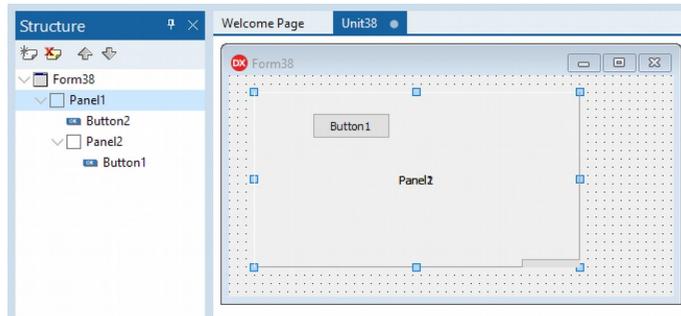
Common operations you do with the form designer (or other design surfaces) are adding components to it, selecting components and changing their properties, associating event handlers or creating new ones. These areas are all covered in the following sections. To add a component you can drag and drop it from the Tool Palette to the Form Designer (or double click in the designer, or select it and then click on the form in the position you want the component).

note In this section I'll generally refer to the VCL form designer. I'm going to highlight which of the features are specific to it. Later on I'll cover what's unique in the FireMonkey form designer instead.

Selecting and Moving a Components

You can select a component directly with the mouse in the Form Designer, use the instance list of the Object Inspector, or use the Structure View, which is particularly handy when a control is behind another one or is very small. You can see the Structure View for a form with a few panels and buttons here:

70 - 03. Using the Form Designer



If one control covers another completely, you can use the Esc key to select the parent control of the current one. Press Esc one or more times to select the form, or press and hold Shift while you click the selected component. Doing so will deselect the current component and select the form by default.

For moving component, just drag them in the designer. While doing so, a hint will display the `Left` and `Top` positions. There are actually different hints for components in the form designer:

- As you move the pointer over a component, the hint shows you the name and type of the component. This is an alternative to the Show Component Captions environment setting, which I tend to keep always active.
- As you resize a control, the hint shows the current size (the `Width` and `Height` properties). Of course, this feature is available only for controls, not for non-visual components (which are indicated in the Form Designer by icons).
- As already mentioned, when you move a component, the hint indicates the current position (the `Left` and `Top` properties).

There are two alternatives to using the mouse to set the position of a component. You can either set values for the `Left` and `Top` properties, or you can use the arrow keys while holding down `Ctrl`. Using arrow keys is particularly useful for fine-tuning an element's position (when the Snap To Grid option is active), as is holding down `Alt` while using the mouse to move the control. If you press `Ctrl+Shift` along with an arrow key, the component will move only at grid intervals. By pressing the arrow keys while you hold down `Shift`, you can fine-tune the size of a component. Again, you can also do this with the mouse and the `Alt` key.

note What if you need to move a control at design time by dragging it, but its area is covered by a child control? Just drag the child control and then press the Esc key (while holding down the mouse button) to switch the dragging operation to the parent control.

Selecting Multiple Components

To select several components, click them with the mouse while holding down the Shift key; or, if all the components fall into a rectangular area, drag the mouse to “draw” a rectangle surrounding them. To select controls inside a container (say, the buttons inside a panel), drag the mouse within the panel while holding down the Ctrl key—otherwise, you move the panel.

To align multiple components or make them the same size, you can select them and set the Top, Left, Width, or Height property for all of them at the same time. You can also use some of the toolbar panels to align controls or change their relative position (bring to front and send to back), as shortly covered in Chapter 1 – but this is a less common feature compared to the early days of Delphi, where there were fewer alternatives.

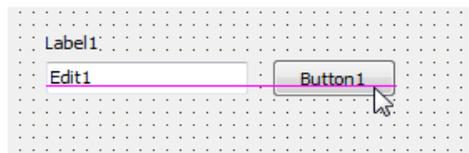
When you’ve finished designing a form, you can use the Lock Controls command of the Edit menu to avoid accidentally changing the position of a component in a form. This is particularly helpful, as there is no real Undo operation on forms (only an Undelete one).

Design Guidelines

The design time guidelines available in Delphi offer you a lot of power for aligning components to the sides, center, or the text baseline. This is a visual aid to properly aligning controls on a form. You can align controls to one of their sides, here the top:



Not only you can align the sides of a control with those of another one, but you can even align the text baseline:



72 - 03. Using the Form Designer

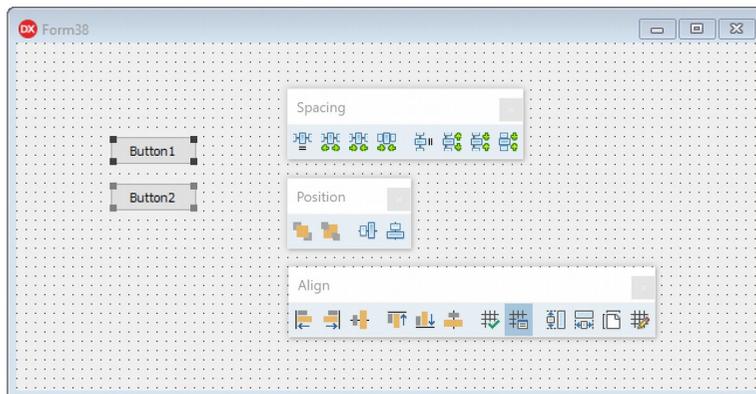
Controls automatically snap to the guidelines when they are close to them (if the corresponding option is set). They also snap when they are at a given margin to the border of the container control.

note Working with the Design Guidelines for the borders of the controls is quite obvious. But how does the designer know about the text baseline for a control? Of course, it doesn't: You need to provide a `TComponentGuidelines` class and register it with the `RegisterComponentGuidelines` function. You can find more details in the `DesignEditors` and `VCLEditors` units of the `ToolsAPI` VCL source code folder.

There is a `gtBaseline` value in the `TDesignerGUIDeType` enumeration, but it is not really referenced in the code, while the class method `TControlGuidelines.GetTextBaseline` seems to provide a default implementation. It looks like you can fully customize the behavior, but having a ready-to-use example would make this easier

Form Designer Toolbars

The Delphi toolbar has a few buttons you can use when working with the form designer, although as I mentioned earlier these are less commonly used compared to the early days of Delphi when fewer alternative options were available. Still, some of the commands of these toolbar panes, like those for equally spacing a number of controls horizontally or vertically are quite handy. Here are the three designer-related toolbar panes (dragged in front of the form to make this picture more focused, but I generally keep them inside the toolbar at the top of the IDE):



Spacing operations are enabled when multiple components are selected and allow to increase or reduce the spacing horizontally or vertically and to space the selected controls equally.

Position operations include ability to change the z-order (bring to front or move to back) and to center the control or controls in the middle of the form, vertically or horizontally.

This is different from the buttons used to align horizontal and vertical centers of the selected controls that are part of the Align operations, along with aligning on each of the four sides, make controls of the same size, and also enable the grid and snap to grid options.

Some of the same operations are also available via direct commands or dialog boxes activated via the local menu of the form designer, as described later.

The Form Positioner

Another interesting feature is the “Form Positioner” in the bottom right corner of the designer surface:



There you can see, in small, the position of the form on the screen, which is useful if the form uses absolute positioning (in many cases you let the operating system pick the position or center the form). You can also use the Form Positioner to modify the Top and Left properties of the form visually, by dragging that small rectangle.

A little known feature is that you can click the Form Positioner to get (temporarily) a bigger view and use it to position the form more precisely. Finally, if you have a background active in Windows, this is going to show up in the Form Positioner background.

More Form Designer Tip

Among its other features, the Form Designer offers some additional information and hints:

- The designer will show the name of a nonvisual component if you turn on the Show Component Captions check box in the Environment

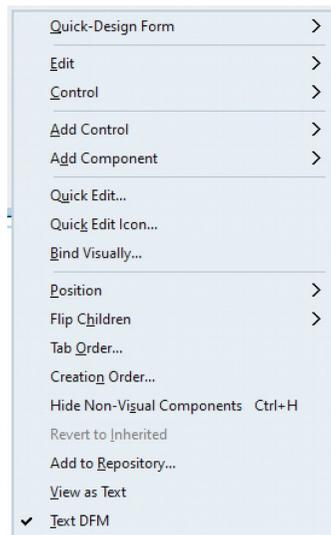
74 - 03. Using the Form Designer

Options/Delphi Options/VCL Designer page of the Options dialog box. This setting is disabled by default.

- As you move the pointer over a component, the hint shows you the name and type of the component. If extended hints are enabled (in the same settings page) you'll also see details about the control's position, size, tab order, and more.
- As you resize a control, the hint shows the current size (the Width and Height properties). As you move a component, the hint indicates the current position (the Left and Top properties).

Form Designer Local Menu

While you are working on a form, the local menu has a number of useful features (some of which are also available in the Edit pull-down of the main menu).



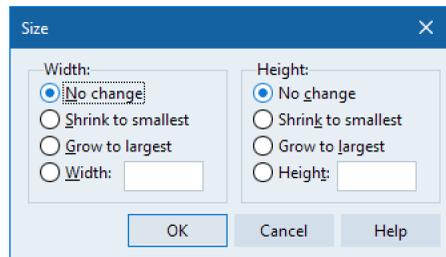
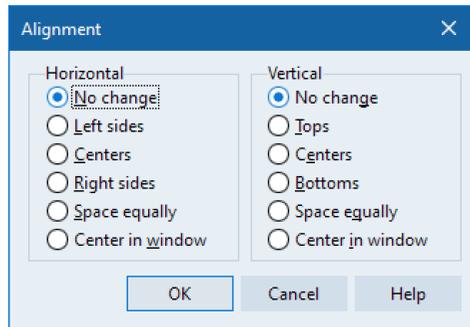
Quick Design is part of the Quick Edit operations (along with the Add Control, Add Component, and Quick edit commands) covered in a later section in this chapter.

The Edit menu has the standard Cut, Copy and Paste operations, while the Control menu offers the Bring To Front and Send To Back options to change the relative position of components of the same kind (you can never bring a graphical component in front of a component based on a window).

The Position sub-menu has commands to align controls to the grid, align two or more selected controls, size them, and scale the entire form. The Align and Size

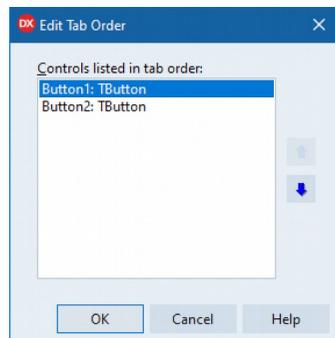
03. Using the Form Designer - 75

operations offers the following dialog boxes, which has features someone similar to the corresponding toolbar buttons I covered earlier:



The Flip Children menu options create a horizontally specular view of the selected controls or all of the controls of the form, this reverse (right to left) position of the controls in meant for languages written from the right.

The Tab Order command opens the tab position dialog box, which simplifies the definition of the `Taborder` property for a series of components, given you can list them all and define the flow in the dialog and the editor will apply the changes to the respective components property:



76 - 03. Using the Form Designer

The Creation Order dialog box is somehow similar to the tab order editor, but it affects the creation (and initialization) order of the non-visual controls. In most cases, you can ignore it as even components with cross references have proper mechanism allowing their creation in any order.

Hide Non-Visual Components is a relatively new option of the designer, used to remove the icons for components from the form designer. They remain listed in the Structure View and in the Object Inspector for selection.

note Other IDEs have a “gutter” area with non-visual components, but Delphi designers decided against it and that decision was maintained over time.

In an inherited form (or a FireMonkey derived view), you can use the command Revert to Inherited to restore the properties of the selected component to the values of the parent form.

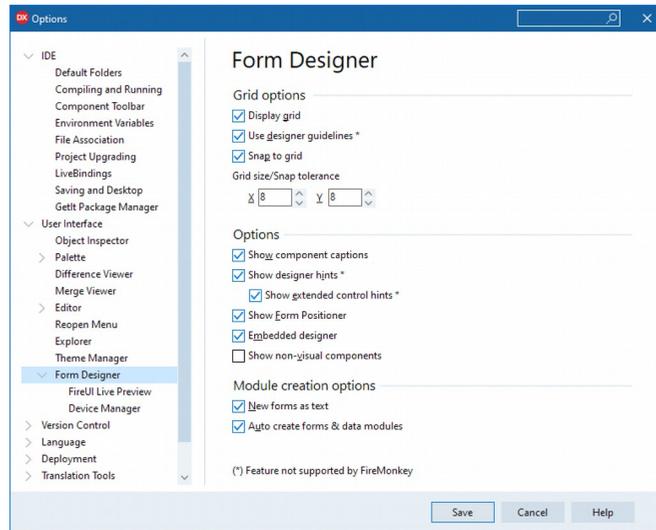
You can use the Add to Repository command to add a copy of the form you are working on to a list of forms available for use in other projects (a rarely used feature).

Finally, you can use the View as Text command to close the form and open its textual description in the editor (as already explained in Chapter 1). A corresponding command in the editor local menu (View as Form) will reverse the situation and get back to the form designer view. As already discussed, in Delphi all the visual operations you do are saved in a DFM or FMX file (for VCL and FireMonkey), which by default uses a text format (it used to be a binary format in the early versions of Delphi). The last command of the form designer local menu, Text DFM, toggles the format used for the form file.

note Having designer files stored as text lets you operate more effectively with version-control systems. When graphical elements are embedded, though, they are saved as binary data within the text file.

Form Designer Options

Along with specific local menu commands, you can set some form options by using the Tools | Options command and choosing the User Interface | Form Designer section. This page is shown here:



The Grid options enable the display of the grid, automatic snapping, and designer guidelines (for VCL only). The grid makes it easier to place components exactly where you want them on the form by “snapping” them to fixed positions and sizes. Without a grid, it is difficult to align two components manually (using the mouse).

The other options include component captions, hints, the form positioner, and how to handle non-visual components. There is an option to disable the embedded designer, which requires restarting the IDE and is not really recommended.

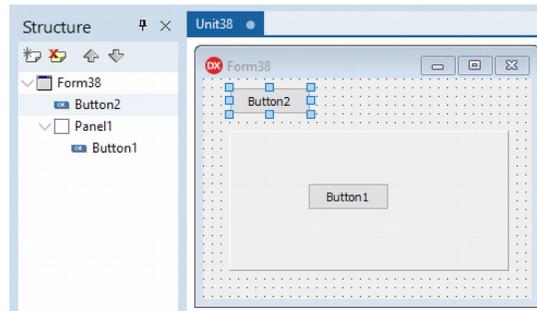
Finally the form creation options determine what happens when you add a new secondary form (or data module) to an existing application and affect the code added to the main project file.

The Structure View for Designers

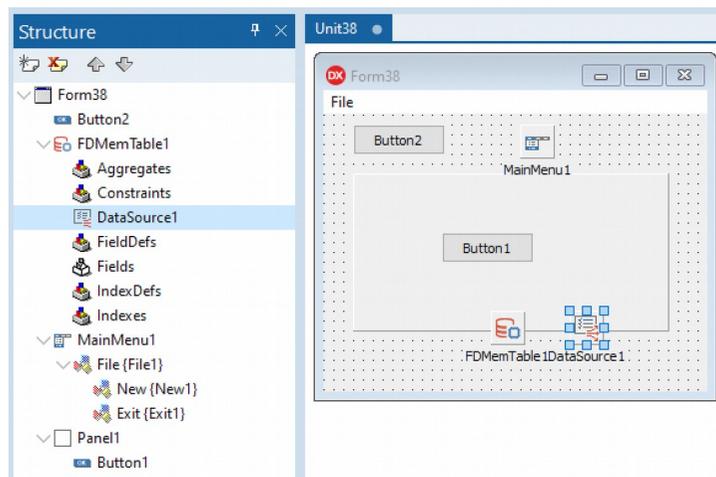
The Structure View is a window showing the tree-based structure of what's active in the IDE, either a designer surface or a source code file. I'll cover the first case here and the second in the next chapter.

The Structure View for a designer shows all the components and objects on the form in a tree representing their relations. The most obvious is the parent/child relation: if you place a panel on a form, a button inside the panel, and a button outside the panel, the tree will show one button under the form and the other under the panel, as shown here:

78 - 03. Using the Form Designer



Besides parent/child, the Structure View shows other relations, such as owner/owned, component/sub-object, and collection/item, plus various specific relations, including dataset/connection and data source/dataset relations. Here, you can see an example of the structure of a menu and the data source component under the dataset – although they are formally sibling components owned by the form:



The Structure View is particularly useful when working with collection properties and the database tables fields at design time, and you can use it to create new items rather than opening the specific designer.

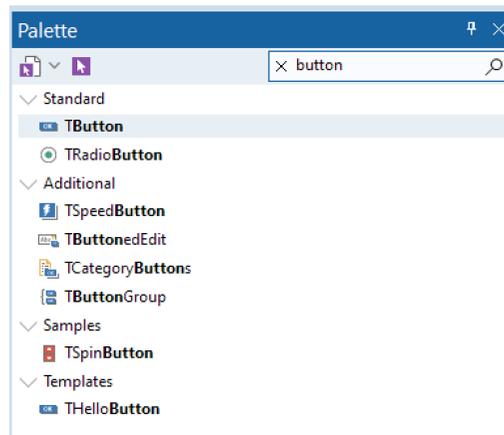
You can drag components within the Structure View—for example, moving a component from one container to another (in the case above, you can drag the `Button2` control over the `Panel1` control to make it a child of the panel). Moving instead of using cut and paste provides the advantage that any connections among components are not lost.

Finally, right-clicking any element of the Structure View displays a shortcut menu similar to the local menu for a component in the form designer – but this is handy as you can more easily select the specific component even if its surface is covered by other child controls. You can also delete items from the tree.

The Palette

Since the introduction of the “Galileo IDE” the Tool Palette (or just Palette, as the IDE refers to it) replaced the Component Palette as a way to select a new component to be added to a form or another designer surface. Notice that the Palette is context sensitive and shows either the list of components (when a designer is active) or the list of the New Items options (when the editor is active). The content of the Palette can be filtered by typing it its search box – and we have already seen how you can directly search for component also in the global IDE Insight search box positioned in the IDE title bar.

As you start typing, the search is smart enough to allow for partial matches (the results of course depends on the components you have installed):



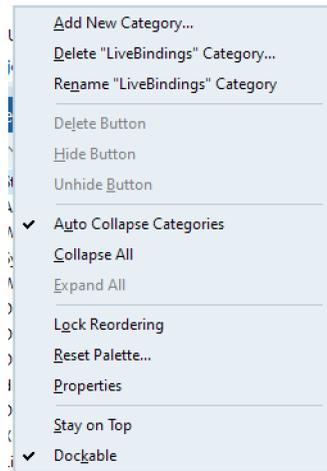
note The search in the Palette does not support the use of wild-cards (?, *). This works in the global search, which means the global search is more handy when you know portions of the words in the component name. Try for example “*but*gr*” for Button Group, it works in IDE insight, but not in the Palette search.

80 - 03. Using the Form Designer

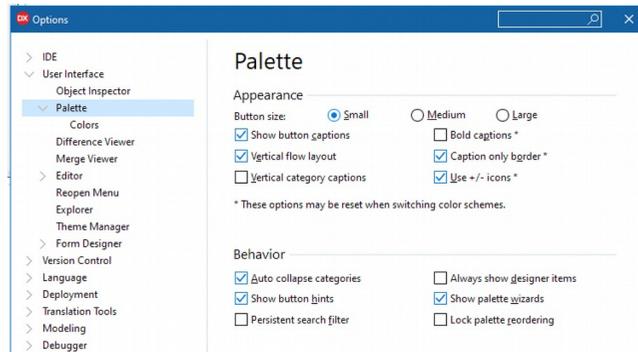
Each page of the palette has a number of components; each component has an icon and a name, which appears as a “fly-by” hint (just move the mouse on the icon and wait for a second). The hints show the official names of components, which I’ll use in this book. They are drawn from the names of the classes defining the component, without the initial T (for example, if the class is `TButton`, the name is *Button*).

If you need to place a number of components of the same kind into a form, shift-click on that component in the palette. Then, every time you click on the form, Delphi adds a new component of that kind. To stop this operation, simply click on the standard selector (the arrow icon) on the left side of the Component palette.

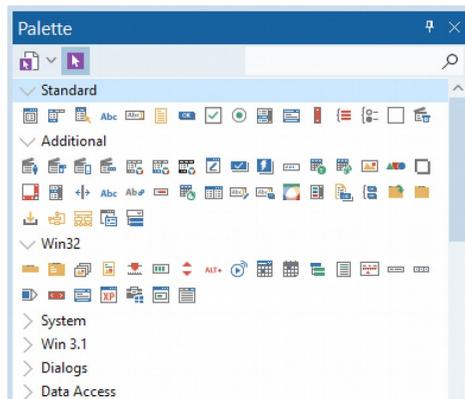
The Palette can be configured in many different ways. Its local menu has several configurations settings:



As you can see you can create new custom categories and rename the existing ones (after which you can drag component icons in this categories, re-arranging the entire organization), you can collapse or expand categories (and enable/disable the auto collapsing mechanism), lock the content (disabling dragging), and reset to default). The Properties menu item opens the configuration page part of the Tools | Options dialog box:



From horizontal to vertical layout, with horizontal or vertical category captions, component names (button captions) always displayed or in bold, and many different behaviors, you can indeed customize the palette to the extreme – something only a handful of developers seems to do. For example, this is the Palette without component captions and auto-collapsing (multiple pages are kept open):



The Object Inspector

When you are designing a form, you use the Object Inspector to set values of component or form properties. Its window lists the properties (or events) of the selected element and their values in two re-sizable columns. An Object Selector at the top of the Object Inspector indicates the current component and its data type; and you can use it to change the current selection.

The Object Inspector doesn't list all of the properties of a component. It includes only the properties that can be set at design-time. As mentioned in Chapter 1, other properties are accessible only at run-time. To know about all the different properties of a component, refer to the documentation.

82 - 03. Using the Form Designer

The right column of the Object Inspector allows only the editing operations appropriate for the data type of the property. Depending on the property, you will be able to do the following actions:

- insert a string or a number by typing them
- choose from a list of options from a drop down list
- invoke a specific editor (indicated by an ellipsis button)

At times more than one of these options are available. For some properties, such as Color, all three will work: you can type a value, select an element from the list, or invoke a specific Color editor.

tip Double-clicking on the value of the property can toggle the value, open the next in the list, or open an editor – depending on the property type and its configuration.

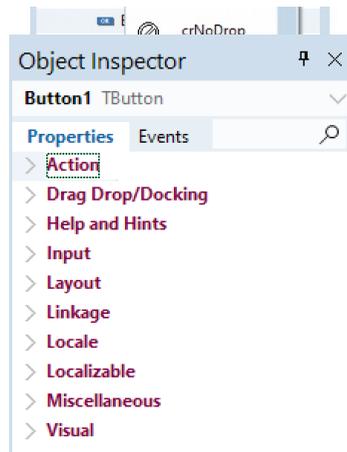
Other properties, such as Font, can be customized either by expanding their sub-properties (indicated by a plus or minus sign next to the name) or by invoking an editor. Also when a property refers to another object you can expand it in place, like a sub-property.

note Another feature of the Object Inspector is the ability to select the component referenced by a property. To do this, double-click with the left mouse button on the property value while keeping the Ctrl key pressed.

In other cases, such as with string lists, the special editors are the only way to change a property. The sub-property mechanism is available with sets and with classes. When you expand sub-properties, each of them has its own behavior in the Object Inspector, again depending on its data type.

Over the years, Delphi added many features to the Object Inspector (some of them have actually been removed or disabled, as they had issues like the rendering of font names with the font itself). The drop-down list for a property in the Object Inspector can include graphical elements. Many of the relevant properties use this feature by default: `Color`, `Cursor` and its variations, generally the `ImageIndex` property of components connected with an `ImageList` (such as an action, a menu item, or a toolbar button), the Pen and Brush styles, and a few others. For example, here you can see a portion of the list of cursors (for the `Cursor` properties):

Another feature that was added and ended up being ignored quite soon (although it is still in the product) is the ability to group properties by category. To understand this feature, you first need to make it visible. To display properties by category instead of by name, right-click in the Object Inspector and choose the proper Arrange option from the shortcut menu. You can see the effect of this choice here:



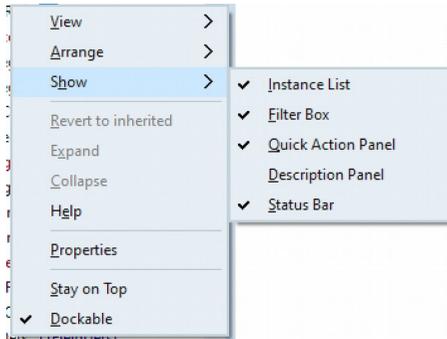
Notice that a property can show up in multiple categories, as categories are not mutually exclusive.

You can use the View sub-menu from the shortcut menu to hide properties of given categories, regardless of the way they are displayed (that is, even if you prefer the traditional arrangement by name, you can still hide the properties of some categories). If any property is hidden, the Object Inspector status bar will indicate how many.

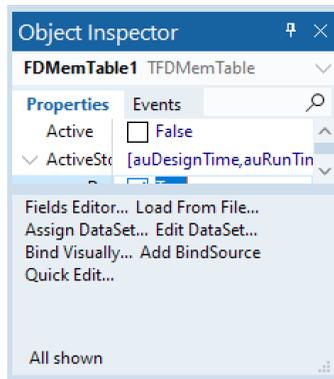
Here are some other tips related to the Object Inspector:

- The instance list at the top of the Object Inspector shows the type of the object and allows you to choose a component. You might remove this list to save some space, considering that you can select components in the Structure View.
- You can optionally view read-only properties in the Object Inspector. Of course, they are grayed out.
- Since Delphi 2010, there is a property editor for Boolean values, which displays a check box you can use to toggle the value (although the drop down list with True and False is still available).
- You can enable various sections of the Object Inspector with the Show local menu:

84 - 03. Using the Form Designer



- There is an optional Description panel at the bottom of the Object Inspector. This is supposed to show information about the current property, but all it does is repeat the property name (and as such it is quite useless – it was introduced to support .Net frameworks!).
- There is also a Quick Action panel (originally called Component Editor panel) which follows Visual Studio style of displaying actions you can do on components via Component Editors – alternatively to using the local menu after selecting a component:



Using Quick Edits

Quick Edits is a collection of features added to recent versions of Delphi and that collectively help you design your forms and edit the components faster. All of the

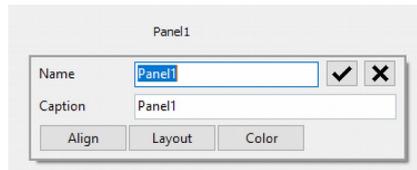
Quick Edit features show up in the local menu of the form designer (and also in the Quick Action panel of the Object Inspector, as you can see in the last image of the previous section). Many of the Quick Edits options depend on the selected component.

The Quick Edit Dialog

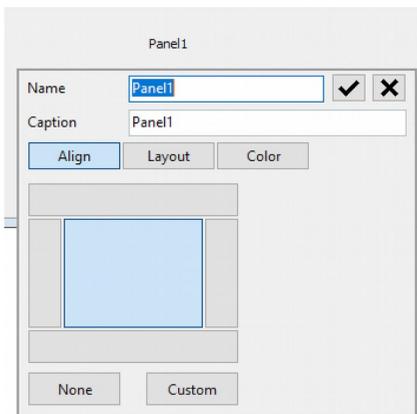
The most visible feature, available for all components, is the QuickEdit dialog, activated from the local menu of the form on the selected component. While the dialog is available for all components and controls, its content varies. In the most basic case of a non-visual component, the only option of the dialog is to edit the component's name:



Visual controls have a QuickEdit dialog including also the option to edit the caption or text, with three additional buttons depending on the control features:

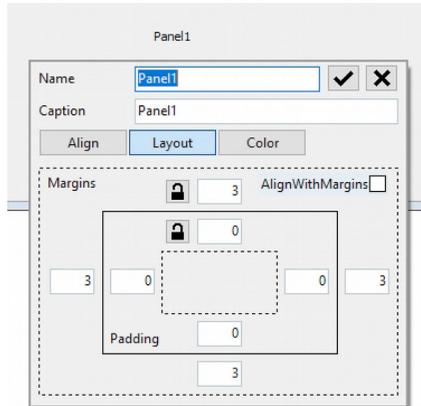


The Align button opens an alignment surface (here is the VCL one, the FireMonkey one is a bit overcomplicated):



86 - 03. Using the Form Designer

You can *visually* pick the alignment that you need, or use the two buttons on the bottom for special cases. Similarly, the layout options offers the ability to easily customize the margins and paddings of a visual control:

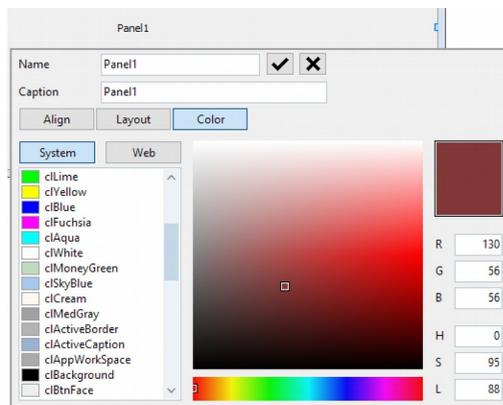


Having all of these related options in a single screen allows for faster configuration, and having them graphically depicted makes it also much easier to understand what you are doing.

tip

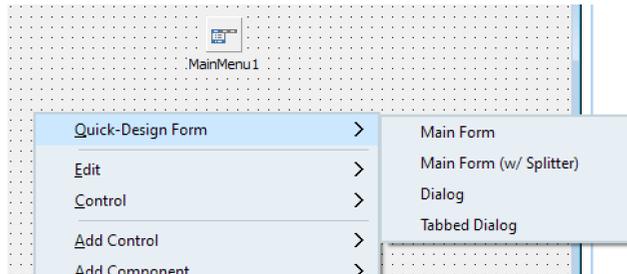
The locker symbol in the layout configuration of the QuickEdit dialog is used to enter the same element in all 4 related fields, without having to type the same value multiple times. It is more of a synchronized edit, than a *don't change* locker.

The third button opens a color selection pane allowing you to pick either a predefined system or web color, or a specific color value (via RGB, HSL, or mouse click over the color selector):

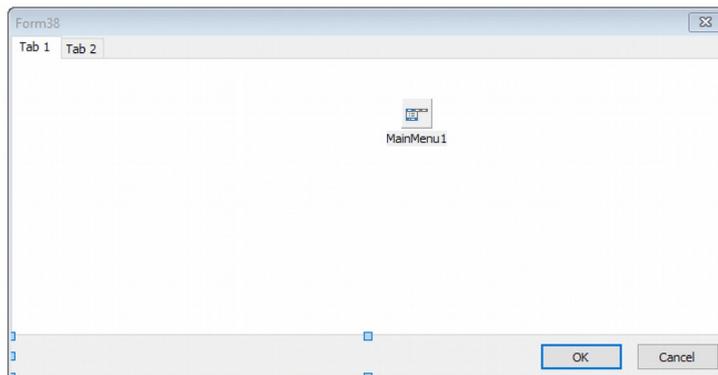


Quick Design Form

Another feature of QuickEdits is the ability to get started with a form predefined layout. When the form is selected, the form designer local menu has an item that offers four options, matching different default configurations:



Just select one to get going with a ready-to-use layout, like a tabbed dialog:

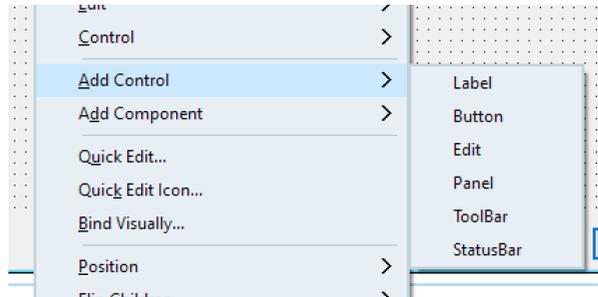


This is a very nice and quick way to get started with a new form.

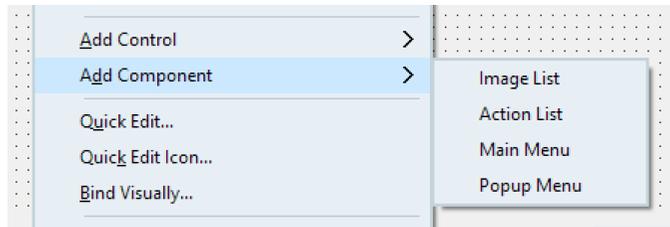
Quickly Adding Controls and Components

Another feature of QuickEdits is the ability to add commonly used controls and component rapidly via the local menu of the designer. Here for example you can see the list of controls available on a form:

88 - 03. Using the Form Designer



All other “container controls” like panels and tabs have similar options for adding new controls as children (rather than in the form). All of these controls also have the Add Component menu, but in this case this is for non visual components so it acts at the form (or designer) level:

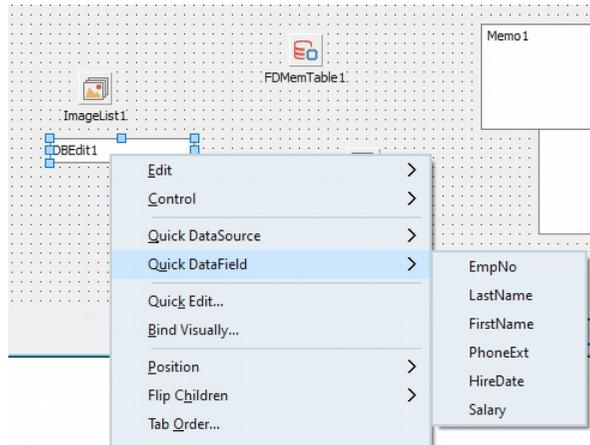


Special QuickEdits Options

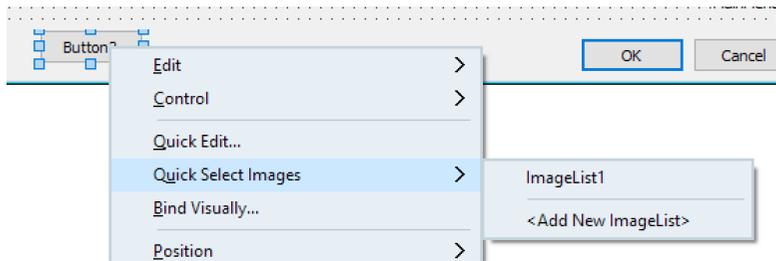
The remaining features of QuickEdits depend on the component selected and its properties. These features show up as additional items of the designer local menu. For example, a form has a special “Quick Edit Icon” option (see the last image above) opening the editor for its Icon property.

There are many other cases and scenarios, probably too many to offer an exhaustive list, but here are a few worth noticing.

Data-aware controls have a Quick DataSource option (visible only if a data source is indeed available). It lists available data source component for quick linking. Once this is set, you can use the Quick Data Field option for connecting a dataset field compatible with the component you are working on:



Controls with a reference to an ImageList and an Image Index, can set both via a specific Quick Select Images menu item:



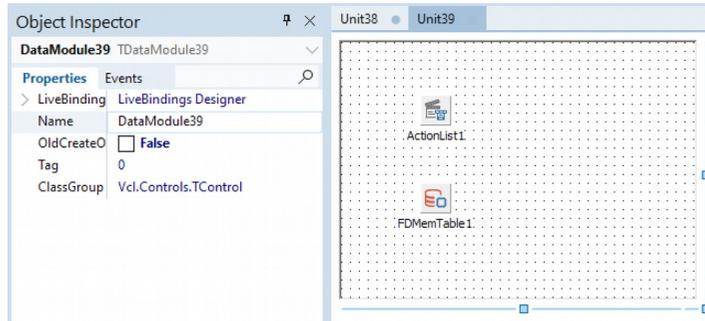
Again, there are other similar menu items that can help doing common operations for other controls, I recommend you keeping an eye for those as they can speed up your work considerably.

Using Data Modules

Forms are surfaces where you define an application user interface and they can host both visual and non visual controls. There are cases in which you want to define application logic and you don't need a user interface, but still want to take advantage of Delphi's design capabilities. In these cases you can use a Data Module, a container of non visual components like data access ones. A data module lacks all of the user interface related capabilities and it is lighter both in memory use and time to create compared to a form.

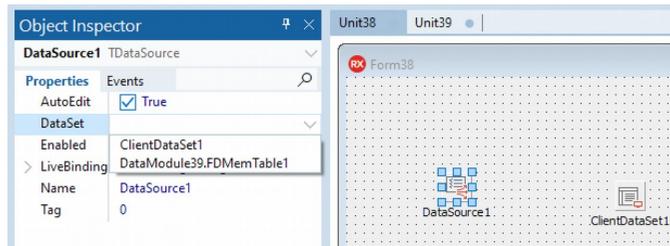
90 - 03. Using the Form Designer

As you create a data module you'll immediately see the differences, both in the design surface UI and in the properties in the Object Inspector:



Of course, if you try to add a visual control to a data module, you'll get an error message. The Tools palette, in any case, filters out the visual controls when a data module is the active designer, preventing you from adding them in the first place.

Once you have created a data module, you can refer to it and its components even at design time from a form (or another designer). Simply select the File | Use Unit command and at this point the data module components will be available like the local ones. Here is an example of selecting a data set of a data source on a form, offering the link to a local data set and one on a data module:



Using Frames

Another designer surface is that of a frame. A frame is a collection of controls and components that you can “replicate” in one of more forms. Whenever you need to repeat the same layout in multiple locations, using a frame you can avoid duplicating the code and configuration, but reuse it. The frame defines a template and each use is not actually a copy, but a reference: changes to the frame itself gets reflected in each instance. At the same time, you retain the ability to customize and modify the instances: each property you change gets *disconnected* from the original version (and won't change any more if the original changes) and you can restore it to the original value.

Frames are a fairly advanced topic to cover here, so I've decided to provide only a short introduction. Given I introduced Component Templates in the last chapter, let me just highlight the difference between these two approaches.

From Component Templates to Frames

When you copy one or more components from one form to another, you simply copy all of their properties. A more powerful approach, as we have seen, is to create a *component template*, which makes a copy of both the properties and the source code of the event handlers. As you paste the template into a new form, by selecting the pseudo-component from the palette, Delphi will replicate the source code of the event handlers in the new form.

Component templates are handy when different forms need the same group of components and associated event handlers. The problem is that once you place an instance of the template in a form, Delphi makes a copy of the components and their code, which is no longer related to the template. There is no way to modify the template definition itself, and it is certainly not possible to make the same change effective in all the forms that use the template. Am I asking too much? Not at all. This is what the *frames* technology in Delphi does.

A frame is a sort of panel you can work with at design time in a way similar to a form. You simply create a new frame, place some controls in it, and add code to the event handlers. After the frame is ready you can open a form, select the Frame pseudo-component from the Standard page of the Component Palette, and choose one of the available frames (of the current project). After placing the frame in a form, you'll see it as if the components were copied to it. If you modify the original frame (in its own designer), the changes will be reflected in each of the instances of the frame.

Like forms, frames define classes, so they fit within the VCL object-oriented model much more easily than Component Templates. As you might imagine from this short introduction, frames are a powerful technique.