Programming in Delphi

CHAPTER

1

IN THIS CHAPTER

This chapter is intended to provide you with a high-level overview of Delphi, including history, feature sets, how Delphi fits into the world of Windows development, and general tidbits of information you need to know to be a Delphi developer. And just to get your technical juices flowing, this chapter also discusses the need-to-know features of the Delphi IDE, pointing out some of those hard-to-find features that even seasoned Delphi developers might not know about.

This chapter isn't about providing an education on the very basics of how one develops software in Delphi. We figure you spent good money on this book to learn new and interesting things—not to read a rehash of content you can already find in Borland's documentation. True to that, our mission is to deliver the goods: to show you the power features of this product and ultimately how to employ those features to build commercial-quality software. Hopefully, our backgrounds and experience with the tool will enable us to provide you with some interesting and useful insights along the way. We feel that experienced and new Delphi developers alike will benefit from this chapter (and this book!), as long as new developers understand that this isn't ground zero for a Delphi developer. Start with the Borland documentation and simple examples. Once you've got the hang of how the IDE works and the general flow of application development, welcome aboard and enjoy the ride!

The Delphi Product Family

Delphi 6 comes in three flavors designed to fit a variety of needs: Delphi 6 Personal, Delphi 6 Professional, and Delphi 6 Enterprise. Each of these versions is targeted at a different type of developer.

Delphi 6 Personal is the entry-level version. It provides everything you need to start writing applications with Delphi, and it's ideal for hobbyists and students who want to break into Delphi programming on a budget. This version includes the following features:

- Optimizing 32-bit Object Pascal compiler, including a variety of new and enhanced language features.
- Visual Component Library (VCL), which includes over 85 components standard on the Component Palette.
- Package support, which enables you to create small executables and component libraries.
- An IDE that includes an editor, debugger, form designer, and a host of productivity features.
- IDE enhancements such as visual form inheritance and linking, object tree view, class completion, and Code Insight.

- Full support for Win32 API, including COM, GDI, DirectX, multithreading, and various Microsoft and third-party software development kits (SDKs).
- Licensing permits building applications for personal use only: No commercial distribution of applications built with Delphi 6 Personal is permitted.

Delphi 6 Professional is intended for use by professional developers who don't require enterprise development capabilities. If you're a professional developer building and deploying applications or Delphi components, this product is designed for you. The Professional edition includes everything in the Personal edition, plus the following:

- More than 225 VCL components on the Component Palette
- More than 160 CLX components for cross-platform development between Windows and Linux
- Database support, including DataCLX database architecture, data-aware VCL controls, dbExpress cross-platform components and drivers, ActiveX Data Objects (ADO), the Borland Database Engine (BDE) for legacy connectivity, a virtual dataset architecture that enables you to incorporate other database types into VCL, the Database Explorer tool, a data repository, and InterBase Express native InterBase components
- InterBase and MySQL drivers for dbExpress
- DataCLX database architecture (formerly known as MIDAS) with MyBase XML-based local data engine
- Wizards for creating COM/COM+ components, such as ActiveX controls, ActiveForms, Automation servers, property pages, and transactional components
- A variety of third-party tools and components, include the INDY internet tools, the QuickReports reporting tool, the TeeChart graphing and charting components, and NetMasters FastNet controls
- InterBase 6 database server and five-user license
- The Web Deployment feature for easy distribution of ActiveX content via the Web
- The InstallSHIELD MSI Light application-deployment tool
- The OpenTools API for developing components that integrate tightly within the Delphi environment as well as an interface for PVCS version control
- NetCLX WebBroker tools and components for developing cross-platform applications for the Internet
- Source code for the Visual Component Library (VCL), Component Library for Cross-platform (CLX), runtime library (RTL), and property editors
- License for commercial distribution of applications developed with Delphi 6 Professional

Delphi 6 Enterprise is targeted toward developers who create enterprise-scale applications. The Enterprise version includes everything included in the other two Delphi editions, plus the following:

- Over 300 VCL components on the Component Palette
- BizSnap technology for creating XML-based applications and Web services
- WebSnap Web application design platform for integrating XML and scripting technologies with Web-based applications
- CORBA support for client and sever applications, including version 4.0x of the VisiBroker ORB and Borland AppServer version 4.5
- TeamSource source control software, which enables team development and supports various versioning engines (ZIP and PVCS included)
- Tools for easily translating and localizing applications
- SQLLinks BDE drivers for Oracle, MS SQL Server, InterBase, Informix, Sybase, and DB2
- Oracle and DB2 drivers for dbExpress
- Advanced tools for building SQL-based applications, including SQL Explorer, SQL Monitor, SQL Builder, and ADT column support in grid

Delphi: What and Why

We're often asked questions such as "What makes Delphi so good?" and "Why should I choose Delphi over Tool X?" Over the years, we've developed two answers to these types of questions: a long answer and a short answer. The short answer is *productivity*. Using Delphi is simply the most productive way we've found to build applications for Windows. Of course, there are those (bosses and perspective clients) for whom the short answer will not suffice, so then we must break out the long answer. The long answer describes the combined qualities that make Delphi so productive. We boil down the productivity of software development tools into a pentagon of five important attributes:

- The quality of the visual development environment
- The speediness of the compiler versus the efficiency of the compiled code
- The power of the programming language versus its complexity
- The flexibility and scalability of the database architecture
- The design and usage patterns enforced by the framework

Although admittedly many other factors are involved, such as deployment issues, documentation, third-party support, and so on, we've found this simple model to be quite accurate in

explaining to folks why we choose Delphi. Some of these categories also involve some amount of subjectivity, but that's the point; how productive are *you* with a particular tool? By rating a tool on a scale of 1 to 5 for each attribute and plotting each on an axis of the graph shown in Figure 1.1, the end result will be a pentagon. The greater the surface area of this pentagon, the more productive the tool.

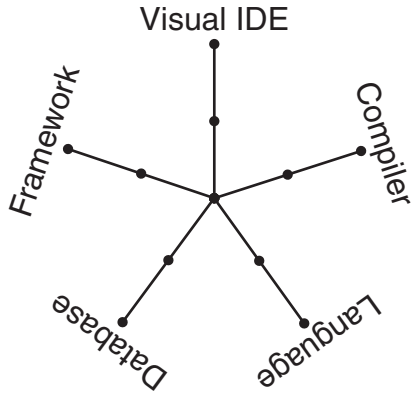

FIGURE 1.1
The development tool productivity graph.

We won't tell you what we came up with when we used this formula—that's for you to decide! Let's take an in-depth look at each of these attributes and how they apply to Delphi as well as how they compare with other Windows development tools.

The Quality of the Visual Development Environment

The visual development environment can generally be divided into three constituent components: the editor, the debugger, and the form designer. Like most modern *rapid application development (RAD)* tools, these three components work in harmony as you design an application. While you're working in the form designer, Delphi is generating code behind the scenes for the components you drop and manipulate on forms. You can add additional code in the editor to define application behavior, and you can debug your application from the same editor by setting breakpoints, watches, and so on.

Delphi's editor is generally on par with those of other tools. The CodeInsight technologies, which save you a lot of typing, are probably the best around. They're based on compiler information, rather than type library info like Visual Basic, and are therefore able to help in a wider variety of situations. Although the Delphi editor sports some good configuration options, I would rate Visual Studio's editor as more configurable.

Recent versions of Delphi's debugger have finally caught up with the debugger support in Visual Studio, with advanced features such as remote debugging, process attachment, DLL and package debugging, automatic local watches, and a CPU window. Delphi also has some nice IDE support for debugging by allowing windows to be placed and docked where you like during debugging and enabling that state to be saved as a named desktop setting. One very nice debugger feature that's commonplace in interpreted environments such as Visual Basic and some Java tools is the ability to change code to modify application behavior while the application is being debugged. Unfortunately, this type of feature is much more difficult to accomplish when compiling to native code and is therefore unsupported by Delphi.

A form designer is usually a feature unique to RAD tools, such as Delphi, Visual Basic, C++Builder, and PowerBuilder. More classical development environments, such as Visual C++ and Borland C++, typically provide dialog editors, but those tend not to be as integrated into the development workflow as a form designer. Based on the productivity graph from Figure 1.1, you can see that the lack of a form designer really has a negative effect on the overall productivity of the tool for application development.

Over the years, Delphi and Visual Basic have engaged in a sort of tug-of-war of form designer features, with each new version surpassing the other in functionality. One trait of Delphi's form designer that sets it apart from others is the fact that Delphi is built on top of a true object-oriented framework. Given that, changes you make to base classes will propagate up to any ancestor classes. A key feature that leverages this trait is *visual form inheritance* (VFI). VFI enables you to dynamically descend from any of the other forms in your project or in the Gallery. What's more, changes made to the base form from which you descend will cascade and reflect in its descendants. You'll find more information on this feature in the electronic version of *Delphi 5 Developer's Guide* on the CD accompanying this book in Chapter 3, "Application Frameworks and Design Concepts."

The Speediness of the Compiler Versus the Efficiency of the Compiled Code

A speedy compile enables you to develop software incrementally, thus making frequent changes to your source code, recompiling, testing, changing, recompiling, testing again, and so forth: a very efficient development cycle. When compilation speed is slower, developers are forced to make source changes in batch, making multiple modifications prior to compiling and adapting to a less efficient development cycle. The advantage of runtime efficiency is self-evident; faster runtime execution and smaller binaries are always good.

Perhaps the best-known feature of the Pascal compiler upon which Delphi is based is that it's fast. In fact, it's probably the fastest high-level language native code compiler for Windows.

C++, which has traditionally been dog-slow in terms of compile speed, has made great strides in recent years with incremental linking and various caching strategies found in Visual C++ and C++Builder in particular. Still, even these C++ compilers are typically several times slower than Delphi's compiler.

Does all this compile-time speed mean a tradeoff in runtime efficiency? The answer is, of course, no. Delphi shares the compiler back end with the C++Builder compiler, so the efficiency of the generated code is on par with that of a very good C++ compiler. In the latest reliable benchmarks, Visual C++ actually rated tops in speed and size efficiency in many cases, thanks to some very nice optimizations. Although these small advantages are unnoticeable for general application development, they might make a difference if you're writing computation-intensive code.

Visual Basic is a little unique with regard to compiler technology. During development, VB operates in an interpreted mode and is quite responsive. When you want to deploy, you can invoke the VB compiler to generate the EXE. This compiler is fairly slow and its speed efficiency rates well behind Delphi and C++ tools. At the time of this writing, Microsoft's next iteration, Visual Basic.NET, is in beta and promises to make improvements in this area.

Java is another interesting case. Top Java-based tools such as JBuilder and Visual J++ boast compile times approaching that of Delphi. Runtime speed efficiency, however, often leaves something to be desired because Java is an interpreted language. Although Java continues to make steady improvements, runtime speed in most real-world scenarios lags behind that of Delphi and C++.

The Power of the Programming Language Versus Its Complexity

Power and complexity are very much in the eye of the beholder, and this particular category has served as the guidon for many an online flame war. What's easy to one person might be difficult to another, and what's limiting to one might be considered elegant by yet another. Therefore, the following is based on the authors' experience and personal preferences.

Assembly is the ultimate power language. There's very little you can't do. However, writing even the simplest Windows application in assembly is an arduous and error-prone venture. Not only that, but it's sometimes nearly impossible to maintain an assembly code base in a team environment for any length of time. As code passes from one owner to the next to the next, design ideas and intents become more and more cloudy, until the code starts to look more like Sanskrit than a computer language. Therefore, we would score assembly very low in this category because, although powerful, assembly language is too complex for nearly all application development chores.

C++ is another extremely powerful language. With the aid of really potent features such as preprocessor macros, templates, operator overloading, and more, you can very nearly design your own language within C++. If the vast array of features at your disposal are used judiciously, you can develop very clear and maintainable code. The problem, however, is that many developers can't resist overusing these features, and it's quite easy to create truly horrible code. In fact, it's easier to write bad C++ code than good because the language doesn't lend itself toward good design—it's up to the developer.

Two languages that we feel are very similar in that they strike a very good balance between complexity and power are Object Pascal and Java. Both take the approach of limiting available features in an effort to enforce logical design on the developer. For example, both avoid the very object-oriented but easy-to-abuse notion of multiple inheritance in favor of enabling a class to implement multiple interfaces. Both lack the nifty but dangerous feature of operator overloading. Also, both make source files first-class citizens in the language rather than a detail to be dealt with by the linker. What's more, both languages take advantage of power features that add the most bang for the buck, such as exception handling, Runtime Type Information (RTTI), and native memory-managed strings. Not coincidentally, both languages weren't written by committee but rather nurtured by an individual or small group within a single organization with a common understanding of what the language should be.

Visual Basic started life as a language designed to be easy enough for programming beginners to pick up quickly (hence the name). However, as language features were added to address shortcomings over the years, Visual Basic has become more and more complex. In an effort to hide the details from developers, Visual Basic still maintains some walls that must be navigated around in order to build complex projects. Again, Microsoft's next-generation Visual Basic.NET is making significant changes in this area, albeit at the expense of backward compatibility.

The Flexibility and Scalability of the Database Architecture

Because of Borland's lack of a database agenda, Delphi maintains what we feel to be one of the most flexible database architectures of any tool. Out of the box, dbExpress is very efficient (although at the expense of advanced functionality), but the selection of drivers is rather limited. BDE still works and performs relatively well for most applications against a wide range of data sources, although it is being phased out by Borland. Additionally, the native ADO components provide an efficient means for communicating through ADO or ODBC. If InterBase is your bag, the IBExpress native InterBase components provide the most effective means to communicate with that database server. If none of this provides the data access you're looking

for, you can write your own data-access class by leveraging the abstract dataset architecture or purchase a third-party dataset solution. Furthermore, DataCLX makes it easy to logically or physically divide, into multiple tiers, access to any of these data sources.

Microsoft tools logically tend to focus on Microsoft's own databases and data-access solutions, be they ODBC, OLE DB, or others.

The Design and Usage Patterns Enforced by the Framework

This is the magic bullet or the holy grail of software design that other tools seem to be missing. All other things being equal, VCL is the most important part of Delphi. The ability to manipulate components at design time, design components, and inherit behavior from other components using object-oriented (OO) techniques it a critical ingredient to Delphi's level of productivity. When writing VCL components, you can't help but employ solid OO design methodologies in many cases. By contrast, other component-based frameworks are often too rigid or too complicated.

ActiveX controls, for example, provide many of the same design-time benefits of VCL controls, but there's no way to inherit from an ActiveX control to create a new class with some different behaviors. Traditional class frameworks, such as OWL and MFC, typically require you to have a great deal of internal framework knowledge in order to be productive, and they're hampered by a lack of RAD tool-like design-time support. Microsoft's .NET common library finally puts Microsoft on the right track in terms of component-based development, and it even works with a variety of their tools, including C#, Visual C++, and Visual Basic.

A Little History

Delphi is, at heart, a Pascal compiler. Delphi 6 is the next step in the evolution of the same Pascal compiler that Borland has been developing since Anders Hejlsberg wrote the first Turbo Pascal compiler more than 17 years ago. Pascal programmers throughout the years have enjoyed the stability, grace, and, of course, the compile speed that Turbo Pascal offers. Delphi 6 is no exception—its compiler is the synthesis of more than a decade of compiler experience and a state-of-the-art 32-bit optimizing compiler. Although the capabilities of the compiler have grown considerably over the years, the speed of the compiler has remarkably diminished only slightly. What's more, the stability of the Delphi compiler continues to be a yardstick by which others are measured.

Now it's time for a little walk down memory lane, as we look at each of the versions of Delphi and a little of the historical context surrounding each product's release.

Delphi 1

In the early days of DOS, programmers had a choice between productive-but-slow BASIC and efficient-but-complex assembly language. Turbo Pascal, which offered the simplicity of a structured language and the performance of a real compiler, bridged that gap. Windows 3.1 programmers faced a similar choice—a choice between a powerful-yet-unwieldy language such as C++ and an easy-to-use-but-limiting language such as Visual Basic. Delphi 1 answered that call by offering a radically different approach to Windows development: visual development, compiled executables, DLLs, databases, you name it—a visual environment without limits. Delphi 1 was the first Windows development tool to combine a visual development environment, an optimizing native-code compiler, and a scalable database access engine. It defined the phrase *rapid application development (RAD)*.

The combination of compiler, RAD tool, and fast database access was too compelling for scads of VB developers, and Delphi won many converts. Also, many Turbo Pascal developers reinvented their careers by transitioning to this slick, new tool. Word got out that Object Pascal wasn't the same as that language we had to use in college that made us feel like we were programming with one hand behind our backs, and many more developers came to Delphi to take advantage of the robust design patterns encouraged by the language and the tool. The Visual Basic team at Microsoft, lacking serious competition before Delphi, was caught totally unprepared. Slow, fat, and dumb, Visual Basic 3 was arguably no match for Delphi 1.

The year was 1995. Borland was appealing a huge lawsuit loss to Lotus for infringing on the 1-2-3 "look and feel" with Quattro. Borland was also taking lumps from Microsoft for trying to play in the application space with Microsoft. Borland got out of the application business by selling the Quattro business to Novell and targeting dBASE and Paradox to database developers, as opposed to casual users. While Borland was playing in the applications market, Microsoft had quietly leveraged its platform business to take away from Borland a vast share of the Windows developer tools market. Newly refocused on its core competency of developer tools, Borland was looking to do some damage with Delphi and a new release of Borland C++.

Delphi 2

A year later, Delphi 2 provided all these same benefits under the modern 32-bit operating systems of Windows 95 and Windows NT. Additionally, Delphi 2 extended productivity with additional features and functionality not found in version 1, such as a 32-bit compiler that produces faster applications, an enhanced and extended object library, revamped database support, improved string handling, OLE support, Visual Form Inheritance, and compatibility with 16-bit Delphi projects. Delphi 2 became the yardstick by which all other RAD tools are measured.

The year was 1996, and the most important Windows platform release since 3.0—32-bit Windows 95—had just happened in the latter part of the previous year. Borland was eager to make Delphi the preeminent development tool for that platform. An interesting historical note is that Delphi 2 was originally going to be called *Delphi32*, to underscore the fact that it was designed for 32-bit Windows. However, the product name was changed before release to Delphi 2 to illustrate that Delphi was a mature product and avoid what is known in the software business as the "1.0 blues."

Microsoft attempted to counter with Visual Basic 4, but it was plagued by poor performance, lack of 16-to-32-bit portability, and key design flaws. Still, there's an impressive number of developers who continued to use Visual Basic for whatever the reason. Borland also longed to see Delphi penetrate the high-end client/server market occupied by tools such as PowerBuilder, but this version didn't yet have the muscle necessary to unseat such products from their corporate perches.

The corporate strategy at this time was undeniably to focus on corporate customers. The decision to change direction in this way was no doubt fueled by the diminishing market relevance of dBASE and Paradox, and the dwindling revenues realized in the C++ market also aided this decision. In order to help jumpstart that effort to take on the enterprises, Borland made the mistake of acquiring Open Environment Corporation, a middleware company with basically two products: an outmoded DCE-based middleware that you might call an ancestor of CORBA and a proprietary technology for distributed OLE about to be ushered into obsolescence by DCOM.

Delphi 3

During the development of Delphi 1, the Delphi development team was preoccupied with simply creating and releasing a groundbreaking development tool. For Delphi 2, the development team had its hands full primarily with the tasks of moving to 32 bit (while maintaining almost complete backward compatibility) and adding new database and client/server features needed by corporate IT. While Delphi 3 was being created, the development team had the opportunity to expand the tool set to provide an extraordinary level of breadth and depth for solutions to some of the sticky problems faced by Windows developers. In particular, Delphi 3 made it easy to use the notoriously complicated technologies of COM and ActiveX, World Wide Web application development, "thin client" applications, and multitier databases architectures. Delphi 3's Code Insight helped to make the actual code-writing process a bit easier, although for the most part, the basic methodology for writing Delphi applications was the same as in Delphi 1.

This was 1997, and the competition was doing some interesting things. On the low end, Microsoft finally started to get something right with Visual Basic 5, which included a compiler to address long-standing performance problems, good COM/ActiveX support, and some key

new platform features. On the high-end, Delphi was now successfully unseating products such as PowerBuilder and Forte in corporations.

Delphi lost a key member of the team during the Delphi 3 development cycle when Anders Hejlsberg, the Chief Architect, decided to move on and took a position with Microsoft Corporation. The team didn't lose a beat, however, because Chuck Jazdzewski, long time co-architect was able to step into the head role.

Delphi 4

Delphi 4 focused on making Delphi development easier. The Module Explorer was introduced in Delphi, and it enabled you to browse and edit units from a convenient graphical interface. New code navigation and class completion features enabled you to focus on the meat of your applications with a minimum of busy work. The IDE was redesigned with dockable toolbars and windows to make your development more convenient, and the debugger was greatly improved. Delphi 4 extended the product's reach into the enterprise with outstanding multitier support using technologies such as MIDAS, DCOM, MTS, and CORBA.

This was 1998, and Delphi had effectively secured its position relative to the competition. The front lines had stabilized somewhat, although Delphi continued to slowly gain market share. CORBA was the industry buzz, and Delphi had it and the competition did not. There was a bit of a down-side to Delphi 4 as well: After enjoying several years of being the most stable development tool on the market, Delphi 4 had earned a reputation among long-time Delphi users for not living up to the very high standard for solid engineering and stability.

The release of Delphi 4 followed the acquisition of Visigenic, one of the CORBA industry leaders. Borland changed its name to *Inprise* in an effort to better penetrate the enterprise, and the company was in a position to lead the industry to new ground by integrating its tools with the CORBA technology. To really win, CORBA needed to be made as easy as COM or Internet development had been made in past versions of Borland tools. However, for various reasons, the integration wasn't as full as it should have been, and the CORBA-development tool integration was destined to play a bit part in the overall software-development picture.

Delphi 5

Delphi 5 moved ahead on a few fronts: First, Delphi 5 continued what Delphi 4 started by adding many more features to make easy those tasks that traditionally take time, hopefully enabling you to concentrate more on what you want to write and less on how to write it. These new productivity features include further IDE and debugger enhancements, TeamSource team development software, and translation tools. Second, Delphi 5 contained a host of new features aimed squarely at making Internet development easier. These new Internet features include the

Active Server Object Wizard for ASP creation, the InternetExpress components for XML support, and new MIDAS features, making it a very versatile data platform for the Internet. Finally, Borland built time into the schedule to deliver the most important feature of all for Delphi 5: stability. Like fine wine, you cannot rush great software, and Borland waited until Delphi 5 was ready before letting it out the door.

Delphi 5 was released in the latter half of 1999. Delphi continues to penetrate the enterprise, whereas Visual Basic continues to serve as competition on the low end. However, the battle lines still appear stable. Inprise brought back the Borland name but only as a brand. The executive offices went through some turbulent times, with the company divisionalized between tools and middleware, the abrupt departure of CEO Del Yocam, and the hiring of Internet-savvy CEO Dale Fuller, who refocused the company back on software developers.

Delphi 6

Clearly the primary theme of Delphi 6 is compatibility with Borland's Kylix development tool for Linux. To this end, Borland developed the new Component Library for Cross-Platform (CLX), which includes VisualCLX for visual development, DataCLX client data-access components, and NetCLX Internet components. Applications written using only the CLX library and portable RTL elements will easily port between the Windows and Linux operating systems.

The new dbExpress set of components and drivers is one of the biggest breakthroughs to come out of the effort for Linux compatibility because it finally provides a real alternative for the BDE, which has really begun to show its age in recent years.

A secondary theme of Delphi 6 is essentially to embrace all things XML. This includes XML for database applications, Web-based applications, and SOAP-based Web services. Delphi developers have the tools they need to fully embrace the industry-wide trend toward XML, which provides great benefits in terms of applications that function across the traditional boundaries of different development tools, platforms, databases, and across the Internet.

Of course, in addition to all these improvements and additions, Delphi 6 brings the normal host of improvement you've come to expect between product versions in core areas like VCL, the IDE, the debugger, the Object Pascal language, and the RTL.

The Delphi IDE

Just to make sure that we're all on the same page with regard to terminology, Figure 1.2 shows the Delphi IDE and calls attention to its major constituents: the main window, the Component Palette, the toolbars, the Form Designer, the Code Editor, the Object Inspector, Object TreeView, and the Code Explorer.

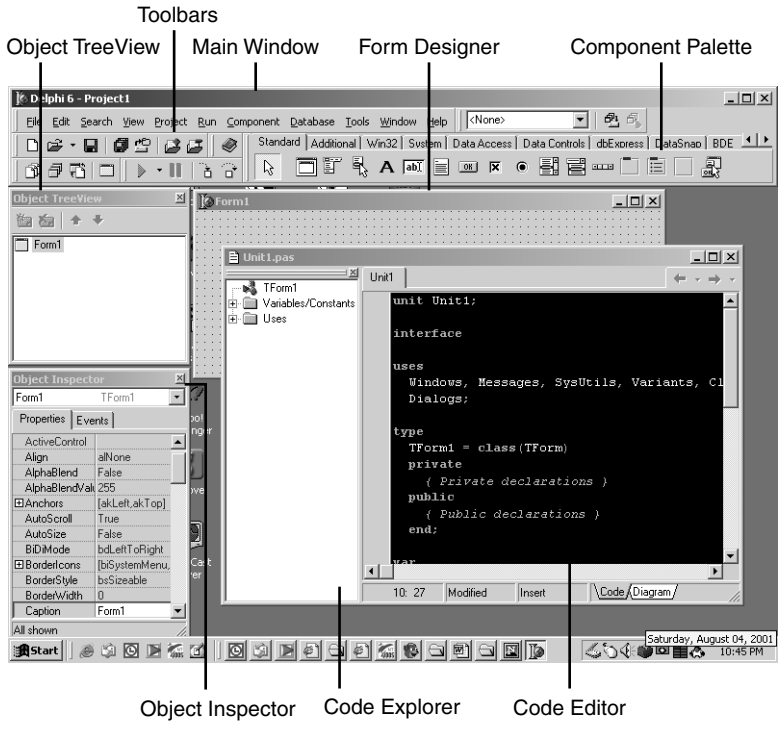

FIGURE 1.2
The Delphi 6 IDE.

The Main Window

Think of the *main window* as the control center for the Delphi IDE. The main window has all the standard functionality of the main window of any other Windows program. It consists of three parts: the main menu, the toolbars, and the Component Palette.

The Main Menu

As in any Windows program, you go to the main menu when you need to open and save files, invoke wizards, view other windows, modify options, and so on. Most items on the main menu can also be invoked via a button on a toolbar.

The Delphi Toolbars

The toolbars enable single-click access to some operation found on the main menu of the IDE, such as opening a file or building a project. Notice that each of the buttons on the toolbars offer a *tooltip* that contain a description of the function of a particular button. Not including the Component Palette, there are five separate toolbars in the IDE: Debug, Desktops, Standard,

View, and Custom. Figure 1.2 shows the default button configuration for these toolbars, but you can add or remove buttons by selecting Customize from the local menu on a toolbar. Figure 1.3 shows the Customize toolbar dialog box. You add buttons by dragging them from this dialog box and drop them on any toolbar. To remove a button, drag it off the toolbar.

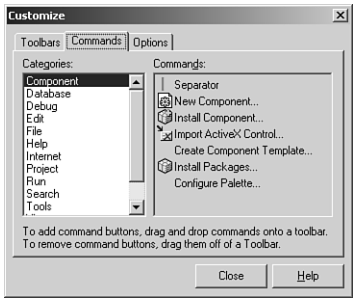

FIGURE 1.3
The Customize toolbar dialog box.

IDE toolbar customization doesn't stop at configuring which buttons are shown. You can also relocate each of the toolbars, the Component Palette, or the menu within the main window. To do this, click the raised gray bars on the left side of the toolbars and drag them around the main window. If you drag the mouse outside the confines of the main window while doing this, you'll see yet another level of customization: The toolbars can be undocked from the main window and reside in their own floating tool windows. Undocked views of the toolbars are shown in Figure 1.4.

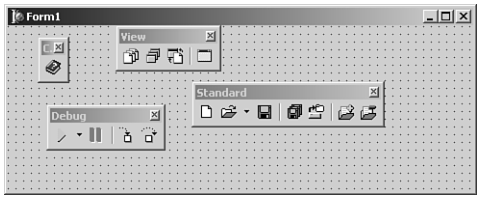

FIGURE 1.4
Undocked toolbars.

The Component Palette

The Component Palette is a double-height toolbar that contains a page control filled with all the VCL components and ActiveX controls installed in the IDE. The order and appearance of pages and components on the Component Palette can be configured via a right-click or by selecting Component, Configure Palette from the main menu.

The Form Designer

The Form Designer begins as an empty window, ready for you to turn it into a Windows application. Consider the Form Designer your artist's canvas for creating Windows applications; here is where you determine how your applications will be represented visually to your users. You interact with the Form Designer by selecting components from the Component Palette and dropping them onto your form. After you have a particular component on the form, you can use the mouse to adjust the position or size of the component. You can control the appearance and behavior of these components by using the Object Inspector and Code Editor.

The Object Inspector

With the Object Inspector, you can modify a form's or component's properties or enable your form or component to respond to different events. *Properties* are data such as height, color, and font that determine how an object appears onscreen. *Events* are portions of code executed in response to occurrences within your application. A mouse-click message and a message for a window to redraw itself are two examples of events. The Object Inspector window uses the standard Windows *notebook tab* metaphor in switching between component properties or events; just select the desired page from the tabs at the top of the window. The properties and events displayed in the Object Inspector reflect whichever form or component currently has focus in the Form Designer.

Delphi also has the capability to arrange the contents of the Object Inspector by category or alphabetically by name. You can do this by right-clicking anywhere in the Object Inspector and selecting Arrange from the local menu. Figure 1.5 shows two Object Inspectors side by side. The one on the left is arranged by category, and the one on the right is arranged by name. You can also specify which categories you would like to view by selecting View from the local menu.

One of the most useful tidbits of knowledge that you as a Delphi programmer should know is that the help system is tightly integrated with the Object Inspector. If you ever get stuck on a particular property or event, just press the F1 key, and WinHelp comes to the rescue.

The Code Editor

The Code Editor is where you type the code that dictates how your program behaves and where Delphi inserts the code that it generates based on the components in your application. The top of the Code Editor window contains notebook tabs, where each tab corresponds to a different source code module or file. Each time you add a new form to your application, a new unit is created and added to the set of tabs at the top of the Code Editor. The local menu in the Code Editor gives you a wide range of options while you're editing, such as closing files, setting bookmarks, and navigating to symbols.

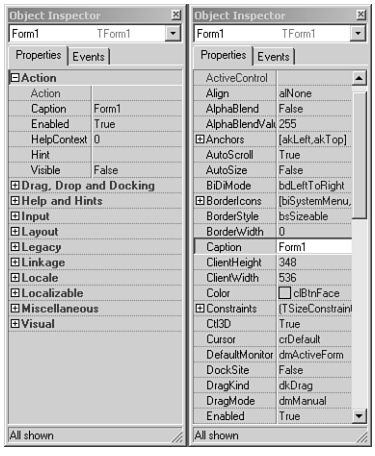

FIGURE 1.5
Viewing the Object Inspector by category and by name.

TIP

You can view multiple Code Editor windows simultaneous by selecting View, New Edit Window from the main menu.

The Code Explorer

The Code Explorer provides a tree-style view of the unit shown in the Code Editor. The Code Explorer allows easy navigation of units in addition to the ability to easily add new elements or rename existing elements in a unit. It's important to remember that there's a one-to-one relationship between Code Explorer windows and Code Editor windows. Right-click a node in the Code Explorer to view the options available for that node. You can also control behaviors such as sorting and filtering in the Code Explorer by modifying the options found on the Explorer tab of the Environment Options dialog box.

The Object TreeView

The Object TreeView provides a visual, hierarchical representation of the components placed on a form, data module, or frame. The tree displays the relationship between individual components, such as parent-child, property-to-component, or property-to-property relationships. In addition to being a means to view relationships, the Object TreeView also serves as a convenient means to establish relationships between components. This can be done most easily by

dropping one component from the palette or the tree on another in the tree. This will establish the relationship between two components that have a possibility of forming a relationship.

A Tour of Your Project's Source

The Delphi IDE generates Object Pascal source code for you as you work with the visual components of the Form Designer. The simplest example of this capability is starting a new project. Select File, New Application in the main window to see a new form in the Form Designer and that form's source code skeleton in the Code Editor. The source code for the new form's unit is shown in Listing 1.1.

LISTING 1.1 Source Code for an Empty Form

```
unit Unit1;

interface

uses
  Windows, Messages, SysUtils, Variants, Classes, Graphics, Controls, Forms,
  Dialogs;

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

var
  Form1: TForm1;

implementation;

{$R *.dfm}

end.
```

It's important to note that the source code module associated with any form is stored in a unit. Although every form has a unit, not every unit has a form. If you're not familiar with how the Pascal language works and what exactly a *unit* is, see Chapter 2, "The Object Pascal Language," which discusses the Object Pascal language for those who are new to Pascal from C++, Visual Basic, Java, or another language.

Let's take a unit skeleton one piece at a time. Here's the top portion:

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

It indicates that the form object, itself, is an object derived from `TForm`, and the space in which you can insert your own public and private variables is labeled clearly. Don't worry about what *class*, *public*, or *private* means right now. Chapter 2 discusses Object Pascal in more detail.

The following line is very important:

```
{$R *.dfm};
```

The `$R` directive in Pascal is used to load an external resource file. This line links the `.DFM` (which stands for *Delphi form*) file into the executable. The `.DFM` file contains a binary representation of the form you created in the Form Designer. The `*` symbol in this case isn't intended to represent a wildcard; it represents the file having the same name as the current unit. So, for example, if the preceding line was in a file called `Unit1.pas`, the `*.DFM` would represent a file by the name of `Unit1.dfm`.

NOTE

A nice feature of the IDE is the ability for you to save new DFM files as text rather than as binary. This option in enabled by default, but you can modify it using the New Forms As Text check box on the Preferences page of the Environment Options dialog box. Although saving forms as text format is just slightly less efficient in terms of size, it's a good practice for a few of reasons: First, it is very easy to make minor changes to text DFMs in any text editor. Second, if the file should become corrupted, it is far easier to repair a corrupted text file than a corrupted binary file. Finally, it becomes much easier for version control systems to manage the form files. Keep in mind also that previous versions of Delphi expect binary DFM files, so you will need to disable this option if you want to create projects that will be used by other versions of Delphi.

The application's project file; is worth a glance, too. A project filename ends in `.DPR` (which stands for *Delphi project*) and is really nothing more than a Pascal source file with a different file extension. The project file is where the main portion of your program (in the Pascal sense) lives. Unlike other versions of Pascal with which you might be familiar, most of the "work" of

your program is done in units rather than in the main module. You can load your project's source file into the Code Editor by selecting Project, View Source from the main menu. Here's the project file from the sample application:

```
program Project1;

uses
  Forms,
  Unit1 in 'Unit1.pas' {Form1};

{$R *.RES}

begin
  Application.Initialize;
  Application.CreateForm(TForm1, Form1);
  Application.Run;
end.
```

As you add more forms and units to the application, they appear in the `uses` clause of the project file. Notice, too, that after the name of a unit in the `uses` clause, the name of the related form appears in comments. If you ever get confused about which units go with which forms, you can regain your bearings by selecting View, Project Manager to bring up the Project Manager window.

NOTE

Each form has exactly one unit associated with it, and you can also have other "code-only" units that aren't associated with any form. In Delphi, you work mostly within your program's; units, and you'll rarely edit your project's `.DPR` file.

Tour of a Small Application

The simple act of plopping a component such as a button onto a form causes code for that element to be generated and added to the form object:

```
type
  TForm1 = class(TForm)
    Button1: TButton;
  private
    { Private declarations }
  public
    { Public declarations }
  end;
```

Now, as you can see, the button is an instance variable of the `TForm1` class. When you refer to the button in contexts outside `TForm1` later in your source code, you must remember to address it as part of the scope of `TForm1` by saying `Form1.Button1`. Scoping is explained in more detail in Chapter 2.

When this button is selected in the Form Designer, you can change its behavior through the Object Inspector. Suppose that, at design time, you want to change the width of the button to 100 pixels, and at runtime, you want to make the button respond to a press by doubling its own height. To change the button width, move over to the Object Browser window, find the `Width` property, and change the value associated with `Width` to `100`. Note that the change doesn't take effect in the Form Designer until you press Enter or move off the `Width` property. To make the button respond to a mouse click, select the Events page on the Object Inspector window to reveal the list of events to which the button can respond. Double-click in the column next to the `OnClick` event, and Delphi generates a procedure skeleton for a mouse-click response and whisks you away to that spot in the source code—in this case, a procedure called `TForm1.Button1Click()`. All that's left to do is to insert the code to double the button's width between the `begin..end` of the event's response method:

```
Button1.Height := Button1.Height * 2;
```

To verify that the "application" compiles and runs, press the F9 key on your keyboard and watch it go!

NOTE

Delphi maintains a reference between generated procedures and the controls to which they correspond. When you compile or save a source code module, Delphi scans your source code and removes all procedure skeletons for which you haven't entered any code between the `begin` and `end`. This means that if you didn't write any code between the `begin` and `end` of the `TForm1.Button1Click()` procedure, for example, Delphi would have removed the procedure from your source code. The bottom line here is this: Don't delete event handler procedures that Delphi has created; just delete your code and let Delphi remove the procedures for you.

After you have fun making the button really big on the form, terminate your program and go back to the Delphi IDE. Now is a good time to mention that you could have generated a response to a mouse click for your button just by double-clicking a control after dropping it onto the form. Double-clicking a component automatically invokes its associated component editor. For most components, this response generates a handler for the first of that component's events listed in the Object Inspector.

What's So Great About Events, Anyway?

If you've ever developed Windows applications the traditional way, without a doubt you'll find the ease of use of Delphi events a welcome alternative to manually catching Windows messages, cracking those messages, and testing for window handles, control IDs, `WParam` parameters, `LParam` parameters, and so on. If you don't know what all that means, that's okay; Chapter 3, "Adventures in Messaging," covers messaging internals.

A Delphi event is often triggered by a Windows message. The `OnMouseDown` event of a `TButton`, for example, is really just an encapsulation of the Windows `WM_xBUTTONDOWN` messages. Notice that the `OnMouseDown` event gives you information such as which button was pressed and the location of the mouse when it happened. A form's `OnKeyDown` event provides similar useful information for key presses. For example, here's the code that Delphi generates for an `OnKeyDown` handler:

```
procedure TForm1.FormKeyDown(Sender: TObject; var Key: Word;
Shift: TShiftState);
begin
end;
```

All the information you need about the key is right at your fingertips. If you're an experienced Windows programmer, you'll appreciate that there aren't any `LParam` or `WParam` parameters, inherited handlers, translates, or dispatches to worry about. This goes way beyond "message cracking" as you might know it because one Delphi event can represent several different Windows messages, as it does with `OnMouseDown` (which handles a variety of mouse messages). What's more, each of the message parameters is passed in as easy-to-understand parameters. Chapter 3 gets into the gory details of how Delphi's internal messaging system works.

Contract-Free Programming

Arguably the biggest benefit that Delphi's event system has over the standard Windows messaging system is that all events are contract free. What *contract free* means to the programmer is that you never are *required* to do anything inside your event handlers. Unlike standard Windows message handling, you don't have to call an inherited handler or pass information back to Windows after handling an event.

Of course, the downside to the contract-free programming model that Delphi's event system provides is that it doesn't always give you the power or flexibility that directly handling Windows messages gives you. You're at the mercy of those who designed the event as far as what level of control you'll have over your application's response to the event. For example, you can modify and kill keystrokes in an `OnKeyPress` handler, but an `OnResize` handler provides you only with a notification that the event occurred—you have no power to prevent or modify the resizing.

Never fear, though. Delphi doesn't prevent you from working directly with Windows messages. It's not as straightforward as the event system because message handling assumes that the programmer has a greater level of knowledge of what Windows expects of every handled message. You have complete power to handle all Windows messages directly by using the `message` keyword. You'll find out much more about writing Windows message handlers in Chapter 3.

The great thing about developing applications with Delphi is that you can use the high-level easy stuff (such as events) when it suits you and still have access to the low-level stuff whenever you need it.

Turbo Prototyping

After hacking Delphi for a little while, you'll probably notice that the learning curve is especially mild. In fact, even if you're new to Delphi, you'll find that writing your first project in Delphi pays immediate dividends in the form of a short development cycle and a robust application. Delphi excels in the one facet of application development that has been the bane of many a Windows programmer: user interface (UI) design.

Sometimes the design of the UI and the general layout of a program is referred to as *prototyping*. In a nonvisual environment, prototyping an application often takes longer than writing the application's implementation, or what is called the *back end*. Of course, the back end of an application is the whole objective of the program in the first place, right? Sure, an intuitive and visually pleasing UI is a big part of the application, but what good would it be, for example, to have a communications program with pretty windows and dialog boxes but no capacity to send data through a modem? As it is with people, so it is with applications; a pretty face is nice to look at, but it has to have substance to be a regular part of our lives. Please, no comments about back ends.

Delphi enables you to use its custom controls to whip out nice-looking UIs in no time flat. In fact, you'll find that after you become comfortable with Delphi's forms, controls, and event-response methods, you'll cut huge chunks off the time you usually take to develop application prototypes. You'll also find that the UIs you develop in Delphi look just as nice as—if not better than—those designed with traditional tools. Often, what you "mock up" in Delphi turns out to be the final product.

Extensible Components and Environment

Because of the object-oriented nature of Delphi, in addition to creating your own components from scratch, you can also create your own customized components based on stock Delphi components. For more details on this and other types of components, you should take a look at Part IV, "Component-Based Development."

In addition to allowing you to integrate custom components into the IDE, Delphi provides the capability to integrate entire subprograms, called *experts*, into the environment. Delphi's Expert Interface enables you to add special menu items and dialog boxes to the IDE to integrate some feature that you feel is worthwhile. An example of an expert is the Database Form Expert located on the Delphi Database menu. Chapter 17, "Using The Open Tools API," outlines the process for creating experts and integrating them into the Delphi IDE.

The Top 10 IDE Features You Must Know and Love

Before we can let you any further into the book, we've got to make sure that you're equipped with the tools you need to survive and the knowledge to use them. In that spirit, what follows is a list of what we feel are the top 10 IDE features you must learn to know and love.

1. Class Completion

Nothing wastes a developer's time more than have to type in all that blasted code! How often is it that you know exactly what you want to write but are limited by how fast your fingers can fly over the keys? Until the spec for the PCI-to-medulla oblongata bus is completed to rid you of all that typing, Delphi has a feature called *class completion* that goes a long way toward alleviating the busy work.

Arguably, the most important feature of class completion is that it is designed to work without being in your face. Simply type in part of a class declaration, press the magic Ctrl+Shift+C keystroke combination, and class completion will attempt to figure our what you're trying to do and generate the right code. For example, if you put the declaration for a procedure called `Foo` in your class and invoke class completion, it will automatically create the definition for this method in the implementation part of the unit. Declare a new property that reads from a field and writes to a method and invoke class completion, and it will automatically generate the code for the field and declare and implement the method.

If you haven't already gotten hooked on class completion, give it a whirl. Soon you'll be lost without it.

2. AppBrowser Navigation

Do you ever look at a line of code in your Code Editor and think, "Gee, I wish I knew where that method is declared"? Well, finding out is as easy as holding down the Ctrl key and clicking the name of the token you want to find. The IDE will use debug information assembled in the background by the compiler to jump to the declaration of the token. Very handy. And like a

Web browser, there's a history stack that you can navigate forward and back through using the little arrows to the right of the tabs in the Code Editor.

3. Interface/Implementation Navigation

Want to navigate between the interface and implementation of a method? Just put the cursor on the method and use Ctrl+Shift+up arrow or down arrow to toggle between the two positions.

4. Dock It!

The IDE allows you to organize the windows on your screen by docking together multiple windows as panes in a single window. If you have full window drag set in your windows desktop, you can easily tell which windows are dockable because they draw a dithered box when they're dragged around the screen. The Code Editor offers three docking bays on its left, bottom, and right sides to which you can affix windows. Windows can be docked side-by-side by dragging one window to an edge of another or tab-docked by dragging one window to the middle of another. Once you come up with an arrangement you like, be sure to save it using the Desktops toolbar. Want to prevent a window from docking? Hold down the Ctrl key while dragging it or right-click in the window and uncheck Dockable in the local menu.

TIP

Here's a cute hidden feature: Right-click the tabs of tab-docked windows, and you'll be able to move the tabs to the top, bottom, left, or right of the window.

5. The Object Browser

Delphi 1 through 4 shipped with essentially the same icky object browser. If you didn't know it was there, don't feel alone; many folks never used it because it didn't have a lot to offer. Delphi now comes equipped with an object browser that enables visual browsing of object hierarchies. Shown in Figure 1.6, the browser is accessible by selecting View, Browser in the main menu. This tool presents a tree view that lets you navigate globals, classes, and units and drill down into scope, inheritance, and references of the symbols.

6. GUID, Anyone?

In the small-but-useful category, you'll find the Ctrl+Shift+G keystroke combination. Pressing this keystroke combination will place a fresh new GUID in the Code Editor, which is a real timesaver when you're declaring new interfaces.

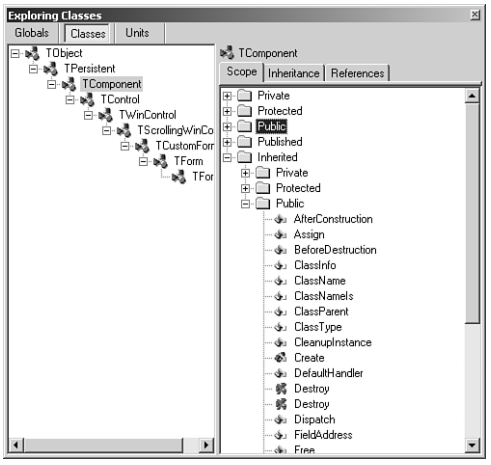

FIGURE 1.6
The new browser.

7. C++ Syntax Highlighting

If you're like us, you often like to view C++ files, such as SDK headers, while you work in Delphi. Because Delphi and C++Builder share the same editor source code, one of the advantages to users is syntax highlighting of C++ files. Just load up a C++ file such as a `.CPP` or `.H` module in the Code Editor, and it handles the rest automatically.

8. To Do. . .

Use the To Do List to manage work in progress in your source files. You can view the To Do List by selecting View, To Do List from the main menu. This list is automatically populated from any comments in your source code that begin with the token *`TODO`*. You can use the To Do Items window to set the owner, priority, and category for any To Do item. This window is shown in Figure 1.7, docked to the bottom of the Code Editor.

9. Use the Project Manager

The Project Manager can be a big timesaver when navigating around large projects—especially those projects that are composed of multiple EXE or DLL modules, but it's amazing how many people forget that it's there. You can access the Project Manager by selecting View, Project Manager from the main menu. There are a number of time saving features in the Project Manager, such as drag-and-drop copying and copy and paste between projects.

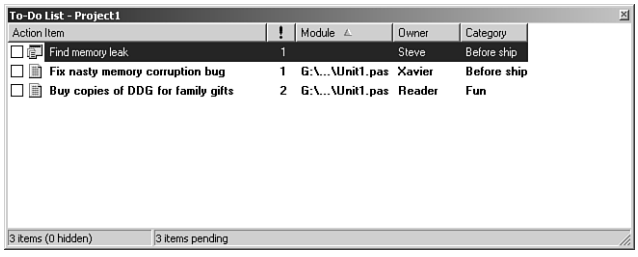

FIGURE 1.7
To Do Items window.

10. Use Code Insight to Complete Declarations and Parameters

When you type **`Identifier.`**, a window will automatically pop up after the dot to provide you with a list of properties, methods, events, and fields available for that identifier. You can right-click this window to sort the list by name or by scope. If the window goes away before you're ready, just press Ctrl+space to bring it back up.

Remembering all the parameters to a function can be a pain, so it's nice that Code Insight automatically helps by providing a tooltip with the parameter list when you type **`FunctionName(`** in the Code Editor. Remember to press Ctrl+Shift+space to bring the tooltip back up if it goes away before you're ready.

Summary

By now you should have an understanding of the Delphi 6 product line and the Delphi IDE as well as how Delphi fits into the Windows development picture in general. This chapter was intended to acclimate you to Delphi and to the concepts used throughout the book. Now the stage has been set for the really technical stuff to come. Before you move much deeper into the book, make sure that you're comfortable using and navigating around the IDE and know how to work with small projects.