By *Zack Urlocker*

The Triumph of Objects

Delphi's RAD Approach to Object-Oriented Programming

During the development of Delphi, one of our goals at Borland was to create a Rapid Application Development (RAD) environment that wouldn't hold you back. We wanted to ensure there weren't limitations that stopped developers from doing something complex or out of the ordinary. After all, if you're in the business of creating custom applications, the fundamental assumption is that unique tasks require custom code to complete.

Several developers said the first generation of RAD tools often ran out of gas — leaving them with only about 80% of the application completed. In many cases, applications had to be recoded either completely or partially, in C or C++, in order to be completed and run with acceptable performance.

With Delphi, we wanted to provide the best of both worlds — give developers a program that featured the rapid development of a 4GL, with the performance and flexibility of an optimizing 3GL compiler.

Luckily, we had a secret weapon: *Objects*.

Surprised? If you thought object-oriented programming (OOP) was something for rocket scientists, you're in for a pleasant surprise. In the early days, OOP was definitely a lot less visual and required greater discipline — but the payoff was still there. Now it's more visual and the payoff is even sweeter.

Do You Remember When?

My first object-oriented program for Windows was written on a 640KB 80286 with a 10MB hard disk and CGA graphics. I was running Actor 1.0 on Windows 1.0 — a beta of a beta, if ever there was one. At that time, the only other way to develop Windows applications was to use the outrageously difficult and expensive combination of Microsoft C and Windows SDK. I'd done a bit of graphical user interface (GUI) programming and wasn't eager to write hundreds of lines of code to deal with every Windows message (display context, handle, and all the low level details required). Actor was a very attractive alternative because it enabled me to build Windows applications from pre-existing components. You still had to write code to work in Actor, but the library of pre-built objects gave it a significant head start. Also, the ability to create new objects made it possible to easily extend the environment.

Of course, the industry has progressed a lot since then. Windows no longer runs on floppy disks, display adapters support graphics modes that require a magnifying glass, and Microsoft has eradicated all UAEs in Windows in favor of general protection (GP) faults. We also now accept that hand-coding Windows applications using C and Windows API is generally to be avoided. Heck, if God wanted everyone to program in C, he would have given us pointers instead of fingers. But I digress.

Your First Object

For a lot of programmers, Delphi's object-oriented environment is a new experience. We purposefully created Delphi so objects can be used easily with the user-interface builder without really having to know the details. First-timers can write a lot of code and complete many tasks without worrying about how the objects work. (You don't have to know pointers, inheritance, or constructors and destructors to get started.) Then, when you're ready to take your programming skills to the next level, the full power of OOP is already at your fingertips.

If we built Delphi right, hopefully many developers will be inspired to learn the concepts of OOP — namely inheritance, encapsulation, and polymorphism — and begin creating custom objects. After all, since Delphi is written in Delphi, there is no distinction between the components supplied and those built. So if you want to add some new control component that offers high performance WinG graphics, cool MIDI support, a rich text editor, or a better database grid, go ahead, you can do it.

But don't forget, you're not limited to creating visual objects, either. If you want to create new abstract objects that model customers, accounts, widgets, or meteors — just do it. The bottom line is simple: You can model whatever is needed in your application development and then reuse it.

The Key Is Reusability

When you begin creating new objects in Delphi, you'll want to keep a few *Object Lessons* in mind. Most importantly, if you want to create reusable objects plan ahead and design with *reuse* in mind. Don't build a complex application and then try to "pull the objects out". Instead, consider the objects needed and analyze the current problem. If you can identify "clusters" of related data and functionality, these are good candidates for objects.

In the early stages of your design, select the data and functionality required in an object and don't worry too much about inheritance. First determine what the object requires, then decide where to get that functionality. In many ways, inheritance is a balancing act between getting the right functionality and interface, while attempting to limit the amount of superfluous baggage. Don't make the mistake of inheriting 20 unneeded capabilities just to get one piece of useful code.

As you create new objects, try to make them consistent with existing *protocols* or ways of interacting. For example, if you want to create a new high-capacity *TRichText* control in Delphi, it would be wise to implement the same key properties and methods found in *TMemo* (such as *Text, Alignment*, *SelectAll*, *GetSelTextBuf*, *CopyToClipboard,* and so on). You should also make sure the methods take the same parameters as an existing *TMemo* control. That way the new control is "plug compatible" with the existing one. This doesn't mean you should base your implementation of a Rich Text editor from the existing *TMemo* class. Actually, you'd probably want to inherit from much higher in the hierarchy.

It often takes two or three iterations before an object is fully reusable. After building a new object with all the properties, methods, and events you think it needs — you may discover it needs just a bit more code. So before you finalize your object, try using it in a complex application. Better yet, get someone else to test and reuse it. If you find code that is frequently implemented by users of your object, try to determine a way to generalize the code and apply it as a method that's built in. As we built Delphi and the Visual Component Library (VCL), there were many cases where new methods or properties were added to objects only after beta testers put them into real world circumstances. ∆

Now go build some objects!

Zack Urlocker is Group Product Manager for Delphi at Borland International. He is a frequent speaker at industry conferences on object-oriented programming. The views expressed here are his own.