Effective Delphi Class Engineering Part 1: Crossing The Chasm

by David Baer

This is the first instalment in a series in which we'll look at what it takes to become an object oriented programmer using Delphi. The intended audience are those of you who have been using Delphi, who know the Object Pascal language reasonably well, and who want to be able to take advantage of the *Object* part of the language of Delphi. For more information on how this series came to be, please see the sidebar opposite.

In the series, we'll focus on what is required in effectively designing, creating and using classes and objects in Delphi. Stated another way, we're going to focus exclusively on Delphi classes. To that end, we *won't* look at a number of other related topics. For example, while all Delphi components are classes, the technology of components is beyond that of classes, and we're not going that far. We're also not going to examine the implications of Delphi classes in the context of COM (Microsoft's Component Object Model). The mindset for applying the COM approach to objects is somewhat different than that for native Delphi OO pursuits. Finally, we won't have time to explore Object Oriented Analysis, that is, the act of determining class composition, class relationships, and so forth. It's not that OOA is a waste of time, but I believe you'll find that a healthy dose of common sense, plus a good understanding of the problem domain of your intended application, will take you a very long way indeed, without a formal study of OOA.

My approach in presenting this material will be much like that taken by Scott Meyers in his *Effective C++* books (to which I am considerably indebted for inspiring this series). I'll provide a one-sentence rule, guideline or suggestion, and follow that with an explanation. Unlike Dr Meyer's books, which assume some C++ OO experience on the part of the reader, we're going to start completely at the beginning.

In this first instalment, we'll focus on what it takes to make the philosophical transition from developer to *object oriented* developer. Much of the material will not seem very Delphi-specific, but fear not. We'll begin to get down to the Delphi metal in Part 2. Likewise, in this first instalment, you won't find much in the way of code examples. There's too much groundwork that needs to be established first. More code will be forthcoming when it's appropriate. So, without further ado, let us begin.

Learning OO

First and foremost, believe that this pursuit is worthwhile.

Learning to use OO effectively is not a trivial undertaking. While the mechanics of using the language to implement classes are only moderately challenging, developing your skills to the point that effective application of these techniques is second nature will require discipline, study and practice. But, have no doubts, your efforts will be handsomely rewarded!

OO has been with us for well over fifteen years now, and it has become a mainstream technology over the last five or six years. Unlike a lot of faddish methodologies or technologies that appear with a huge fanfare, momentarily attract a great deal of attention, and virtually disappear a short time later, OO has unfailingly gained increasing acceptance with each passing year. While many in our profession still don't 'get it', of those who have undertaken a legitimate study of OO, few walk away disenchanted or cynical (and in my experience, 'few' means none).

This is because OO provides a framework within which you can organize your data and program logic in such a fashion that your software will be more reliable, robust and malleable.

It will be more reliable because, provided you are capable of writing coherent code in the first place, the organizational tools OO offers let you produce code with fewer bugs. The bugs that do creep in will often be easier to diagnose and correct.

Your software will be more robust, because you'll waste less time debugging the simple bits, and the OO framework will allow more complex solutions than would be possible with a conventional programming approach.

Finally, your software will be more malleable (that is, more flexible in accommodating changing functionality requirements), again largely because of the organizational qualities inherent in an OO approach.

Delphi is not the only game in town when it comes to OO, but it's a great place to start this study. Pascal already has many positive attributes (even without considering the object capabilities), and those can make this pursuit all the more efficient. To be sure, the Object Pascal object model lacks a few niceties found in other languages, but it also offers a few not found elsewhere. Irrespective of the language, however, developing an OO view of software development is beneficial in its own right. The language is just a vehicle. Once you comprehend the OO philosophy, that understanding

can transcend language-specific considerations, and you should benefit from it throughout your career.

So, are you convinced yet? Learning OO can advance your career, elevate your professional prestige, and even improve your... err... social life. That's a pretty brassy claim indeed, but not really all that far fetched. Let's see: more money (from career advancement), more self-esteem (professional prestige), and more energy (fewer long days chasing elusive bugs). Maybe it's not so far fetched after all!

Speaking for myself, I can tell you that few things impart more satisfaction than designing an elegant and powerful class, and I can only hope you will all feel that excitement yourselves one day. It's an immensely good feeling. You'll feel like clenching your fist, making a gesture like you've just won matchpoint at Wimbledon, going to your office door, and shouting 'The doctor... is *in*!'

Delphi Class Development

Understand that you've been coding Delphi classes all along.

In one sense, the worst enemy to learning Delphi OO programming is Delphi itself. Delphi's RAD capabilities make it easy to produce sophisticated applications without forcing you to examine the magic that makes it all happen. And the source of that magic? It's none other than the Delphi class and object machinery (along with a lot of brilliant engineering on the part of Borland, of course).

The problem is that the Delphi IDE and the VCL collaborate to produce the wonderfully powerful development instrument we've all come to know and love. But it makes it easy to overlook the fact that, when you develop a form, you are actually creating a Delphi class. Unfortunately, there's a compromise that's been made. In order for you to be able to effortlessly create your applications, the practice of certain proper OO techniques is abandoned in favor of ease of use. Marco Cantù explained this conflict nicely in a short paper called *When RAD Is Bad*, and I recommend that you read it when you feel ready. You can find it at the Borland Community website:

```
http://community.borland.com/
  devnews/article/
  1,1714,10463,00.html
```

But my point in bringing this up is not to criticise Borland's solution. Rather, I just want to make you realize that, as we start to explore some particular topic, experiencing some sense of déjà vu may not be all that inexplicable. As you learn more about Delphi class development, you should examine your familiar Delphi coding practices to see they map to the more formal OO you'll presently be learning.

Two Little Words

Understand what a class is and what an object is.

It seems reasonable that we should start with a formal definition of terms *class* and *object*, given that we'll be using them a *lot*. But I think it may be better to start with informal definitions. We'll add to those definitions, at least implicitly, as we proceed. We can get off to a good start by considering an analogy to something with which you should already be quite familiar, Delphi components.

Unfortunately, this analogy has a built-in hindrance. What are

So, Who Died And Named You King?

The above question is a delightfully sarcastic rejoinder I recall from younger days. It was reserved for those occasions where someone was acting bossy or pretentious. It's certainly reasonable to question whether a person who would write 'Effective *Anything*' is guilty of arrogance, especially when, like myself, that person does no routine teaching or mentoring. I can only offer my qualifications, and let you be the judge.

This series came about as a result of an email correspondence with *Our Esteemed Editor*. In one of my messages, I bemoaned the fact that coming up with worthy topics was perhaps the hardest part of writing for a publication in which my fellow contributors have such extraordinary credentials. I was somewhat surprised by his reply: he informed me that a good number of readers wanted more fundamental (ie, less esoteric) information about how to leverage our favorite tool. I had always thought that a series of tutorials on basic object oriented programming using Delphi would be worth writing, and was pleasantly surprised to learn that some people out there might actually want to read it. Perhaps now, with Kylix on the not-too-distant horizon, the value will be even greater. Who knows what size hordes will stampede into the Delphi arena when the Linux doors open?

As for my credentials, I can state with utter confidence that no one has a greater belief in the value of object oriented approaches to software development than myself. I first came to see the power of OO in the early nineties. I was researching a way to migrate some heavy-duty PL/1 programs to an open systems platform that had no PL/1 compiler. When I discovered that C++ string objects could be made to behave just like string variables in PL/1, I was hooked. I immediately became an acolyte and became an OO evangelist not long thereafter. Throughout my career, I seem to have continually gravitated toward projects whose purpose was to empower other developers: compilers, coding tools of various sorts, and larger scale enabling frameworks and architectures. For me, OO was the discovery of a lifetime. Then, Delphi came along and made a good thing even better.

There you have it. There are many who are qualified to write a series like this and some of them, no doubt, could do it better than I. But, as no one else has stepped forward to do so, I'm going to try. I very much hope some find it of value.

those things on the component palette that you click on when you want to drop a control of a form called? They're *components*, of course. Now, what are those controls that appear after they're dropped on your form called? Um... again, *components*?

You see, the little image on the component palette is a representation of an abstraction, that being the declaration and all the implementation code underlying the component. On the other hand, that control on your form is a living, breathing instance of the abstraction (well, metaphorically speaking, anyway). Our common use of the term *component* does not differentiate between the abstraction and the instance.

Take away the IDE interactions and the streaming capabilities of components and we have classes and objects. All component abstractions are *classes*, and all component instances are *objects*. Classes/objects do not have to be components, however. As you will see later, classes have a much broader application than those things on your component palette you've grown to be so dependent upon.

It's quite easy to inadvertently use the term *object* when you mean *class*, even when you've got the concepts down cold. Incorrectly calling a class an object will rarely cause confusion. What's important at this point is the concept. If it helps, simply think of a class as a cookie cutter and the cookies it produces as objects.

Let's take another approach to arriving at a definition of *class*. I'll assume that you are already familiar with Pascal records. Suppose we say that a class combines the data declarations of a record with an associated set of procedures and functions that operate on that data. Now, let's add one more element: suppose we can dictate that some or all of the data members in the record are for the private use of the associated procedures, and that others of those procedures are likewise available only for the private use of the non-private ones.

We're starting to get close to a viable (if incomplete) working definition of what a class is. In doing so, we've also encountered the first great concept of the object oriented philosophy, which is called *encapsulation*. It's often stated that the three main principles of OO are *encapsulation*, *inheritance* and *polymorphism*. Now, inheritance and polymorphism are somewhat elusive topics, and we've got a lot of territory to cover before we start examining them.

Encapsulation, however, is really simple and basic. And compelling too! Even without the other two principles, encapsulation would make OO a worthy pursuit in its own right. It's the main reason that OO programs can be made more reliable and flexible than their non-OO counterparts. We'll see how encapsulation is achieved in the next item.

➤ *Listing 1*

```
TSimpleClass = class
private
  Amount1: Integer;
  Amount1: Integer;
public
  procedure SetAmount1(Value: Integer);
  function GetAmount1: Integer;
  function GetAmount2: Integer;
end;
...

procedure TSimpleClass.SetAmount1(Value: Integer);
begin
  Amount1 := Value;
  Amount2 := Value * 2;
end;
function TSimpleClass.GetAmount1: Integer;
begin
  Result := Amount1;
end;
function TSimpleClass.GetAmount2: Integer;
begin
  Result := Amount2;
end;
```

Class Member Visibility

Make your privates inaccessible to the public.

In defining an OP class, you have four options when choosing the visibility of class data and class procedures and functions: *private*, *protected*, *public* and *published*. First, *private* says 'for internal class use only'. *Public* says 'available to interested outside parties'. *Protected* is a form of *private* that has implications when dealing with inheritance. We'll ignore it for now, but return to it in a later instalment. *Published* is a form of *public*, and is, for the most part, a designation meaningful only for components. As such, we'll not involve ourselves with it at all.

Consider the declaration of a simple class in Listing 1. It has two `integer` data members, `Amount1` and `Amount2`. These are declared as `private`, they are not accessible by the outside world. So, what has access to them?

The answer is that the code in the class's two functions and one procedure do. Class functions and procedures are called *methods*. That's not particularly important at this point, but it will save me a lot of typing from now on. Class methods are like normal functions and procedures, but they also are 'aware' of the class instance (ie, the object) on which they're operating. This is something we'll look at much more closely in the next instalment. But, back to the business at hand.

Our primitive class allows the outside world to supply a value for `Amount1`, but `Amount2` is available only for reference: only methods of the class are allowed to supply a value. In effect, `Amount2` is a read-only value, available via the `GetAmount2` method. `Amount1` may be read by invoking `GetAmount1`, and it may be assigned a new value by invoking `SetAmount1`. We'll see in the third instalment of the series that Delphi supplies a much more elegant capability, called properties, for accessing and assigning class data member values.

And there you have it! With one simple stroke, we've acquired the capability of allowing our objects

to go about their business with only as much interference from the outside world as is necessary to deliver the functionality they are meant to deliver. Internal mechanisms for managing the affairs of a class instance (ie an object) and for keeping track of its current state (which should not be any concern of outside parties) can be kept private.

Class Design

Design your classes to have a clear and dedicated purpose.

Now that we have the magic elixir of encapsulation, let's try to use it properly. There are several important considerations here. First, your classes will be more effective if they're unequivocal in their purpose. In other words, try to avoid giving a class a split personality by attempting to make it do too many things. If a class is a container, say of a list of names and addresses, avoid making it take on unrelated responsibilities (like, for example, supplying a visual control for displaying those names and addresses).

Instead, use multiple classes, which can collaborate on delivering the desired result, or define a higher level class that incorporates both. When each participating class is clearly focused, messy entanglements are avoided and you're in a much better position to respond rapidly to changing functionality requirements. And I hope it goes without saying that in avoiding those entanglements, you're also giving yourself a considerable leg up when it comes to finding problem behavior.

So, how's that for brevity? We've managed to reduce the entire study of OO Design to two short paragraphs! Of course, there's much, much more to it than that. But I'm quite sincere in my claim that a healthy amount of common sense, along with keeping in mind this and the next guideline, will serve you very well indeed. And without that common sense, all the OOD study you can find time for may not get you very far.

Classes can be used for many things. It's probably easiest to recognize how to represent tangible things as classes. A list of names and addresses is tangible, and it won't be difficult to define such a class once you've learned the fundamentals. But, classes can also effectively represent more ephemeral things: processes, relationships, even algorithms.

As you progress in your OO pursuits, you will hopefully begin to see these more subtle uses for classes. As you do, you'll also become more skilled in designing effective class collaborations.

Playing Politics

Design your classes to be apolitical.

This is a continuation of the previous guideline. Assuming you've come up with a nicely focused class purpose, it's very important that you allow your class to not get involved with things that are not in its list of responsibilities.

It may be helpful to think of your class as having a contract for delivering its services to subscribing parties. That contract is the public interface and the specification of what happens when methods of that interface are invoked. You, as the class designer/engineer, need to act as a 'legal advisor'. You must ensure that your class delivers on what is promised by the contract, and that your class does so in a lawful fashion (eg, does not consume excessive system resources, avoids access violations, etc). Once you've got that under control, a further responsibility is to ensure that your class isn't burdened with supplying services outside of those promised by the contract.

There's an unfortunate tendency with many developers to remedy a problem by applying a code modification where it's easiest. But this kind of thinking can be contrary to good class design. It's not the responsibility of a class to correct a deficiency in one of its clients. As soon as you start catering to a specific requirement of one client, you may compromise the ability to serve all the other clients. If necessary, you may need to 're-negotiate' the contract, requiring a class client needing specialized treatment to supply a directive that such specialized behaviour should be enabled.

If you produce classes for use by associates, you will also likely encounter an expectation in some of the other developers using them that your classes be 'psychic', ie, that they be able to determine from some external context that a particular mode of operation is appropriate. When you hear arguments like 'Well, you can figure out from our form naming conventions that we should only be doing a partial clear of the internal list when...', then resist with vigour. Acquiescing to this type of request will almost inevitably compromise the integrity of the whole, and it will become a losing proposition for all concerned.

Planning

Start small, but think enterprise.

There are two aspects to this simple guideline, both of which suggest that, in all you do, consider that your efforts merit the attention and dedication appropriate to an enterprise-scale solution.

First of all, don't attempt overly ambitious class solutions for your first efforts. For example, you may be familiar with the notion that a well-designed application provides a clear separation between user interface issues and business rules and data. There's no question about it, this is an extremely wise philosophy to embrace. However, the design of effective business objects requires a lot of experience. Not only are there a lot of difficult issues with which to contend, business objects and their likely RDBMS data store are awkward partners. Business objects are an excellent goal to aspire to, but you will be likely to find them an elusive target early on.

Instead, consider solving more modest problems. Container classes can be an excellent learning vehicle. By container classes, I'm referring to a broad set of class types that offer storage and management (eg, assignment and retrieval) services appropriate to the type of data being contained

and to problem requirements. I once presented a series of lectures on Delphi OO programming to the developers at my company. The first exercise, that of creating a class for managing bits (simple one bit values accessible as a kind of variable length array), proved to be just about perfect in its scope and challenge.

Having settled upon an initial course of action, the follow-up is equally important. Don't be tempted to take shortcuts just because you feel your efforts aren't very significant. At this stage of the game, attention to detail (that is, attention to doing it *right*) is every bit as important as it would be if you were contributing a key piece of a mission-critical application. The best time to learn good habits and practices is right at the beginning.

The other way to apply this guideline is to observe good OO practices in your day-to-day coding. I expect that many of you reading this have needed to provide some kind of options dialog form to augment an application's main form services. Delphi makes this trivial to do, using either of two approaches. Let's consider a hypothetical case.

We have a requirement that the user be allowed to specify a starting count (say, initial widget count). Furthermore, some external criterion (say, a command-line parameter) dictates an upper limit on that initial value. OK, so we create a simple options form that has an edit control into which the user can type the value.

One approach is for the options form to read the limit value from some global variable on the main form. When the user types in a value, the options form validates it against the upper limit, and sets another global variable on the main form. Alternatively, the main form can access the option form's edit control, and be responsible for the validations and value retrieval.

From an OO design perspective, both of these solutions are pretty abysmal. Suppose we chose the first approach and we later needed to modify the manner in which the main form calculates and stores the upper limit value? We've also got to modify our options form to keep things working. Conversely, suppose we took the second path, and later decided we wanted to replace the edit control with a slider bar. Now, the main form won't compile without changes.

So, what's the right way? There are are a number of viable alternatives. One of them would be to create an application options object, which retains the upper limit and current user-selected value. That object could also be responsible for the validation. The main form could create this object and pass a reference to it via a method call to the options form. The options form would interact with the options object, and the main form would retrieve the resultant values from the object after the options dialog had closed. Neither form would need to interact directly with the other. That's encapsulation at work for you!

OK, that's also admittedly a bit more work than doing it the 'easy' way. It's difficult to see the benefits of encapsulation in situations of low complexity. But the point is that by doing it the right way, you will have hopefully developed some good habits. In situations of high complexity, you'll benefit when these habits are second nature, not practices with which you have to struggle.

Design Methodologies

In the early stages of your OO education, don't feel compelled to learn advanced design methodologies.

This one is based upon personal experience (and is a potentially controversial opinion, admittedly). I recall that in my own early studies of OO, using C++, I was continually looking aside because I thought I needed to understand some peripheral philosophy, methodology or notation. In reading other material on OO practice and technique, you may well feel like you won't make any progress unless you stop to first learn, for example, the Unified Modelling Language. Now, I am not disparaging UML here, and it's probably something you will eventually benefit from learning.

Instead, I'm merely suggesting that you're embarking upon the study of a subject which has a seemingly unlimited 'knowledge space'. You'll always be able to find something new to learn, or ways to improve your existing expertise. Relax! No one can absorb all of this rapidly or easily. But, you do need to get off to a good start. Start by designing and building simple classes, but doing so with a goal that you're going to do it as well as can be done. Stay focused on this one goal, and the fundamentals can be mastered. At that point, feel free to follow your bliss wherever it takes you.

Plunder The Wise

Recognize that you're not going to be able to achieve proficiency in OO practice on your own: seek out the wisdom of the masters.

When faced with a programming problem they don't know how to solve, a great many developers feel content to find some example code that does somewhat the same thing, copy the techniques, get things working, and move on to the next problem. This approach won't take you very far with OO. Certainly, one can learn the rudiments of class design and implementation by looking at how others do it. You'll be able to get to the point where you can make things work, if only somewhat crudely. But there's an ocean of subtlety and nuance you'll never understand if this is your only means of enlightenment.

It seems to me that the study of OO has attracted some of the finest minds of our time. These individuals are *brilliant*, my friends! Furthermore, they frequently meet, they discuss and they argue. What has resulted is a considerable body of extraordinary wisdom. So much, in fact, that it's more than a little intimidating to consider an exploration of it. So, where to begin?

One excellent place to start is the book *Refactoring* by Martin Fowler (Addison-Wesley, ISBN 0-201-48567-2) reviewed in the April 2000 issue of *Developers Review*. The purpose of the book is to show how badly designed OO-based implementations can be methodically improved. The reason I'm recommending it as an early study aid is that Mr Fowler does an admirable job of enumerating bad OO techniques and practices which should be avoided in the first place! All the code examples are in Java, but the examples are mostly brief, and the text is quite adequate in conveying his points in any case, even if you can't follow the Java code.

When you're a bit further along in your adeptness with OO, a must-read is the highly regarded book *Design Patterns* by the so-called Gang of Four (Gamma, Helm, Johnson and Vlissides, Addison-Wesley, ISBN 0-201-63361-2). This remarkable book shows how a number of adaptable strategies may be enacted to solve a wide range of commonly encountered design challenges. In reading it, you may find yourself experiencing one epiphany after another. It's somewhat advanced, so don't be discouraged to not fully understand it upon first encounter (and the code, unfortunately, is all either C++ or Smalltalk).

Finally, there's the VCL itself. Not all classes in the VCL are exemplary, but there's plenty that are. This final suggestion may seem contrary to 'don't just copy' advice earlier. But an examination of a nicely formed solution in relation to a particular topic can be illuminating. I'll be making recommendations for specific classes to study as we go along.

Next Time

We'll make a dizzying descent from the 50,000 foot view to about the one inch view, where we'll take a close-up look at the object machinery as implemented by Delphi's compiler.

David Baer is Chief Software Architect at Spear Technologies in San Francisco. He'd much rather hear the phrase 'Oh Oh' used during the software design phase than the deployment phase. Contact David at dbaer@speartechnologies.com