Moving to VB.NET: Strategies, Concepts and Code – Sample Chapters

by Daniel Appleman

Published by Apress. ISBN: 1893115976. http://www.apress.com or http://www.desaware.com/MovingtoVBNet.htm.

This book is divided into four parts. In Part 1, Strategies, VB6 programmers learn about Visual Basic.NET from a strategic, business and economic perspective. Through this they learn not only Microsoft's strategies behind VB.NET, but how to develop their own strategy of how and when to migrate to VB.NET.

In Part 2, Concepts, VB6 programmers learn the fundamental concepts that every programmer must understand in order to program in VB.NET. They learn to unlearn VB6 habits that do not apply to VB.NET and that can actually lead them to writing unreliable VB.NET code.

In Part 3, Code, VB6 programmers learn about the changes to the VB.NET language. Always building on their existing VB6 knowledge, these chapters cover every aspect of the VB.NET language itself.

In Part 4, "The Wonderful World of .NET", the focus continues to be on code, but this time the emphasis is not on changes to the language but rather on exposing programmers to a wide variety of .NET techniques and namespaces, while continuing to cover the concepts that will allow them to continue to learn on their own.

Rather than provide one sample chapter, the following excerpt contains sections from all three focus areas of the book.

This book is compatible with VB.NET beta 2.

***The excerpts provided here are based on initial manuscript submissions and have not gone through the entire editing process**.*

From Part I – Strategies

Excerpted from chapter 2:

Oh My God, They Broke VB!

The .NET framework is built on a new runtime called the "Common Language Runtime" (or CLR). The subject of runtimes is one that is a bit sensitive to many VB programmers. For a while, VB programmers felt that there was something "wrong" with Visual Basic because it had this huge distribution runtime that supposedly was not needed with C++ programs. Then they learned that most C++ programmers actually required huge runtimes as well (such as MFC[1]). Only C++ programmers writing pure API code or those

[1] Microsoft Foundation Classes

willing to study the intricacies of ATL could get away with creating components and applications that did not require runtimes. The Common Language Runtime is a considerably more ambitious runtime than the old VB or MFC runtimes.[2] According to Microsoft, the CLR was created with a number of goals in mind:

- To make it easy to create objects that can be shared among languages.
- To make it easy to create scalable and robust components and applications (for example, with good handling of threading issues and with no resource or memory leaks).
- To make it possible to create verifiably secure components and applications.
- To make it easy to create components and applications for Web-based applications as well as traditional Windows applications.

I'll be writing more about these goals and others[3] as we continue. For now, let's just consider the impact these goals would have on a language.

A language designed for .NET needs to support the these features, among others:

- Inheritance
- Free-threaded multithreading
- Support for all the CLR-defined variable types
- Attributes and metadata (don't worry, you'll find out what this is later)

These features are part of a new "Common Language Specification" (CLS) that all .NET languages should support.

Visual Basic 6 did not support any of these features.

This presented the developers at Microsoft with an interesting challenge: how to "fix" each of their languages to work with the new Common Language Runtime that is the foundation on which .NET is built.

- They could graft a set of extensions onto the language. This is the choice they made for Visual C++.[4]
- They could create a new language designed specifically for the CLR. This is the choice they made in creating the new C# language (pronounced C-Sharp), which I'll discuss a bit more later.

[2] It is difficult to compare runtime sizes because these language require not only the base runtime file such as the 1.3MB VB6 runtime or the 1MB MFC runtime, but also a variety of dependent DLL's relating to OLE or the C runtime. We don't know how large the .NET runtime will be, but it is expected to be considerably larger—tens of megabytes.

[3] Microsoft also claimed that applications written for the runtime will be able to run on "any platform" that supports the CLR—leaving open the suggestion that the CLR will exist on other systems such as Apple, Linux, or even Palm handhelds. Apple, I can believe. The others? This I gotta see…

[4] I've only briefly looked at the way managed extensions are implemented with VC++. I must confess, my first impression was not particularly favorable. It seems a bit awkward.

* They could revise the language to fit the needs of the CLR even at the risk of breaking backward compatibility with previous editions of the language. This is the choice they made with Visual Basic.NET.

And then the Microsoft developers made a rather courageous and certainly controversial decision. They decided that as long as they had to break backward compatibility anyway, they might as well clean up the language completely—and by clean up, I mean clean up the syntax and add the kinds of features like strict type checking that professional developers have been asking for for years. Let's face it; as much as we like VB, can anyone really say that the syntax of the Line command makes any sense? Part Three of this book will focus entirely on specifics of changes to the language and how to migrate software. For now, the important things to know are:

* VB.NET is a Visual Basic, but it is not "Visual Basic" as it has evolved from VB1 to VB6. It is a different language.
* VB6 code will not load without conversion into VB.NET.
* Microsoft has a migration wizard that converts VB6 code to VB.NET when you load a VB6 project into Visual Studio.NET. How good it will be remains to be seen.[5]
* The Forms engine has changed from the current "Ruby" based forms engine to the new .NET windows forms engine. In addition to the expected programmatic changes, one should expect subtle behavior changes as well.
* The new features in VB.NET open the opportunity to new and improved software architectures.

Which brings us to our next subject.

Everything You Know Is Now Obsolete

[5] I hear that it is excellent—but that was from Microsoft folks who aren't likely to say otherwise. Hopefully, that is the case. However, even if it is, I can't imagine it being so good that a major application or one that uses advanced techniques won't need substantial additional work and testing.

From Part II – Concepts

Excerpted from chapter 4

The Common Language Runtime

It is essential to learn about technology in its proper context. Floods of new features and hyped-up promises are fine for marketing briefs but when evaluating a new technology, there is nothing like understanding the context in which it exists and the problems it was intended to solve. This is especially important when you evaluate whether a certain technology is appropriate to solve your particular problems.

So, now that you know what is wrong with COM, it's time to take a look at how .NET addresses these issues.

Let's start by asking this question: what would it take to do the following?[6]

- Allow a program that uses a component to continue to work even if someone installs a later version of the component that is not backward compatible onto a system.
- Allow a program to detect if someone installs an incompatible version of a component over a working version of that component.
- Eliminate the need to register components.
- Eliminate the problem of circular references and associated memory leaks even in the case where a programmer neglects to free an object.

First, you would need the ability to run multiple versions of a component on a system at one time. That way, the presence of a newer or older copy of a component somewhere on a system would not interfere with a good copy that is available in the application directory. This is side-by-side execution.

Next, you'd need to build into the operating system a mechanism by which it could compare the methods and properties expected by a program with those actually present in a component and not allow the application to run at all if there is a problem. This test should be performed before the program runs rather than waiting for a runtime error or Memory exception to occur.

You'd need to modify the operating system to automatically find and "register" components without using the registry—typically by looking for them in the application's directory or other defined locations where shared components can reside.

Finally, you'd need a way for the operating system to keep track of every object used by an application as it is running and to automatically free those that are no longer in-use.

[6] One can't help but wonder whether the .NET development team actually had these features on a list and in what order they may have placed them. All we can do is speculate based on the results.

These requirements cannot be met with Windows as it exists now. Nor are they compatible with COM in its current implementation. Meeting these requirements requires an entirely different architecture—a different virtual machine. That virtual machine is provided by the Common Language Runtime.

A Visual Basic DLL or EXE created with VB.NET is very different from those you've used in the past. Yes, it does use the PE (Portable Executable) format internally but if you try to run a VB.NET executable on a system without the CLR installed, you'll get a bunch of "DLL not found" errors.[7] That's because Windows needs the CLR to interpret new types of records that are stored in the executable file.

In the .NET world, rather than focusing on DLL's and EXE files, you'll hear the term Assembly used. I'll discuss assemblies more later. For now, just assume that there is a one-to-one correspondence—each DLL and EXE file you create will contain one assembly and every assembly will be made up of just one DLL or EXE.[8]

One of the new records stored in a .NET executable file is called a manifest. A manifest contains a huge amount of information about an assembly. It contains a list of all the components used by the assembly. It contains the version numbers of those components and hashed values that allow the runtime to determine if those components have changed. It contains a list of all of the objects exposed by the assembly as well as all of their methods, properties, associated parameters, and return types. It also includes a list of all of the objects required by the assembly and their methods, properties, associated parameters, and return types.

Another new type of record is an Intermediate Language record—or IL for short. I'll discuss this in more detail shortly.

Manifests

The manifest is the answer to the problems of versioning and deployment that exist under COM. Let's look at these issues one by one.

- The manifest allows a program that uses a component to continue to work even if someone installs a later version of the component that is not backward compatible onto a system.

The CLR supports side-by-side execution. This means that if a component exists in the application directory, the CLR will load that component even if the same component exists (in the same or differing version) somewhere else on the system—and even if the other component is in one of the shared assembly directories.

Does this mean that developers will begin to install components in their own private directories instead of system32 or other shared directories in order to minimize component distribution problems?

[7] In the prerelease version, that is. One would hope that they might come up with a more friendly way to detect and notify users that an application requires the .NET runtime before release.

[8] In fact, it is possible for an assembly to be made up of multiple DLL and EXE files. However VB.NET does not support this capability in the current beta and there is no evidence that this will change for the final release of this version of VB.NET.

You'd better believe it. It will still be possible to create shared components but it will undoubtedly become less common.

Doesn't this mean that systems will start becoming cluttered with multiple versions of the same component? Isn't that a terrible waste of disk space? And won't that also waste memory when components are loaded simultaneously when it isn't necessary? Wasn't the ability to share memory and reduce disk space use the whole idea behind dynamic link libraries in the first place?

Yes, this approach is potentially wasteful in terms of disk space and memory use. And yes, DLLs were originally created in order to reduce memory and disk requirements. But those features were created back in the days where a normal system had 640K of memory and a high-end system had maybe a few megabytes—and disk space cost ten dollars per megabyte or more.[9] These days, the minimum memory on a low-end system is 64MB and you have to work hard to spend more than more than a penny per megabyte of disk space. With those kinds of numbers, the amount of waste due to duplication of DLL files on a system or even in memory is negligible. Developers today try to avoid memory leaks that occur while an application is running, which can ultimately use up even these large amounts of memory (or disk space) in applications designed to run continuously for days or weeks on end. Developers also try to minimize changes to components in one application from interfering in any way with another application.

- The manifest allows a program to detect if someone installs an incompatible version of a component over a working version of that component.

But what if you actually overwrite an existing component with a newer one that is incompatible with the one your application needs?

In this case, what happens depends to some degree on how you've configured the application. You can, for example, require that your application always use a specific version of an assembly. If the correct version is not found, the application will fail to load. The manifest even contains hashed signatures for the dependent assemblies so it can detect if they were changed even if the developer forgot to update the version number!

You can also allow your application to attempt to use newer versions of a component. In this case, the CLR is able to use the manifest to check the new version of the component and verify that it exposes all of the correct objects; and also that the methods and properties and their associated parameters exactly match those expected by your application. If they don't, the application won't be allowed to run.

- The manifest eliminates the need to register components.

The CLR obtains manifest information from your application and its dependency components at load time. None of this information is stored in the registry. It always searches for components first in the directory of your application and its subdirectories. Only afterwards does it check the global assembly cache.

[9] I remember the thrill of having a whole 500MB on my machine and what a bargain it was to get it for only $1,000!

Curiously enough, this capability and others provided by use of a manifest make possible a radically new feature with regard to deploying applications. It will actually be possible to successfully deploy an application simply by copying files (or XCopy a directory structure) onto a system![10]

By the way, it should go without saying (but I'll mention it just in case), that as wonderful as these capabilities are, they aren't magic. In other words, if you use traditional COM components from your .NET application, all of the old rules still apply. You'll need to register them and watch out for the usual compatibility issues—but only for those traditional COM components.

I must also note that the long-term success of this approach will also depend a great deal on Microsoft's ability to keep the CLR itself backward compatible as they enhance it.[11]

[10] I'm sure I'm not the only one thoroughly entertained by the fact that after all these years of advanced Windows technology, we are finally able to do something that was an everyday occurrence under DOS.

[11] However, I've heard rumors that it will be possible to deploy different versions of the CLR itself on a system, in which case it should be possible to avoid runtime-based incompatibilities.

From Part IV– The Wonderful World of .NET

Excerpted from chapter 12

Printing

The name of the printing namespace, System.Drawing.Printing gives an early hint of what is coming. You have just read how GDI+ requires VB6 programmers to switch from the old “VB” style of graphics to a variation of the kind of graphic programming familiar to Win32 API programmers. It probably won’t be too much of a surprise to learn that printing requires the same kind of transition.

The transition can be described in five words:

The Printer object is gone.

And no, it hasn’t been renamed. The whole concept of an object that you can simply print to, is gone.

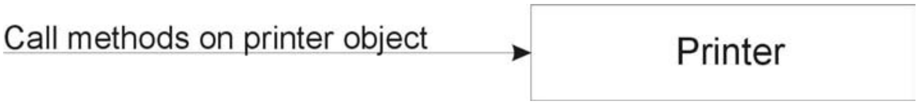
But don’t panic.

The good news is that the .NET approach to printing is incredibly easy to use once you understand a few simple concepts. And it is very, very powerful.

My goal in this section is to guide you through the mental shift needed to understand printing under .NET. Once you have that down, you’ll be able to learn the rest from the namespace documentation.

Printing under VB6 uses a simple model shown in figure 12.3

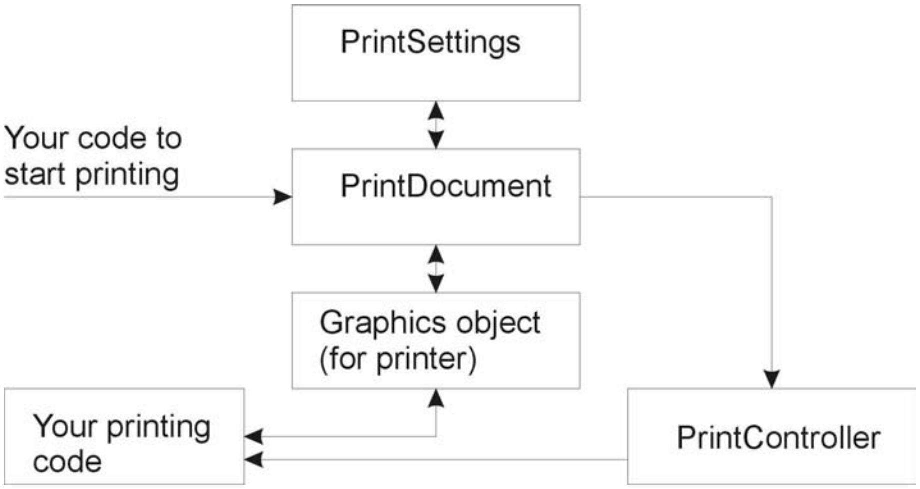
Figure 12.3 – Printing under VB6



You simply send commands to the printer – commands to start printing, commands to draw graphics and text, commands to switch pages, and commands to actually print.

Printing under .NET uses a more complex model shown in figure 12.4

Figure 12.4 – Printing under VB.NET



Whoa!!! This is simple?

Actually it is. Look closely.

The idea is this. Instead of printing each page in a single function, you use one function to tell .NET that you want to start printing. Then the .NET framework raises anevent whenever it is ready to print a page. You then draw into that page during the event.

To look at it another way.

Printing 3 pages in VB6 can be described using pseudocode as follows:

```
Sub Print
    Print page 1
    Print page 2
    Print page 3
End Sub
```

Printing 3 pages in VB.NET can be described using pseudocode as follows:

```
Sub Print
    Print The Document
End Sub

Sub PrintDocument_PrintPage Handles Document’s PrintPage event
    Print a page
    Return value that tells system whether you want to print another page or not
End Sub
```

Let's look how this translates into real code in the PrintingDemo sample project.

The sample project contains a text box that will contain a line of text to print. In this project we cheat by using the font specified by the text box to print to the printer. You can, of course, use any Font object (including one you configure using the System.Windows.Form.FontDialog common dialog box). The project also includes a picturebox that contains an image to print.

You begin the process of printing by creating a new System.Drawing.Printing.PrintDocument object as shown in listing 12.1 . This object has two other objects associated with it – a PrinterSettings object and a PrintController object. The PrinterSettings object determines the printer and print settings to use. The PrintController object actually handles the printing operation. In most cases you'll use the standard PrintController object, but you can derive a new one if you wish to perform specialized operations such as displaying a status window during printing.

In this example, the PrinterSettings object is set using the common dialog for printer settings that every Windows user is familiar with. This dialog is accessed using the PrintDialog objects. When this code returns from the prDialog.ShowDialog method, the PrinterSettings object for the document has been set to the selected printer.

That's right – in VB.NET it is trivial to print to any accessible printer – you don't have to worry about which is the default printer. Naturally, you can select printers and change their settings for a specific print job without using a dialog box – just by using the appropriate objects in the System.Drawing.Printing namespace.

The command to start the print job is the prDoc.Print method. But before calling it, you need to specify the event that is to be raised for each page. Fortunately, you've already read chapter 10, so you know that an event is actually a delegate, and it's trivial to use the AddHandler command to connect the event to any method – in this case, the PagePrintFunction method.

Listing 12.1 – Starting the print operation.

```
Private Sub cmdPrint_Click(ByVal sender As System.Object, _
ByVal e As System.EventArgs) Handles cmdPrint.Click
   Dim prDialog As New PrintDialog()
   Dim prDoc As New Drawing.Printing.PrintDocument()
   prDoc.DocumentName = "My new printed document"
   prDialog.Document = prDoc
   prDialog.ShowDialog()

   ' Wire up the event to be called for each page
   AddHandler prDoc.PrintPage, AddressOf Me.PagePrintFunction
   prDoc.Print()
   prDoc.Dispose()
   prDialog.Dispose()
End Sub
```