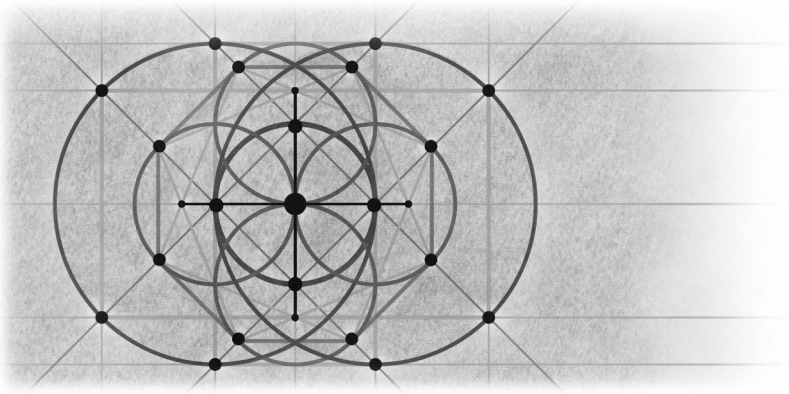

3

Upgrading Options

This chapter provides an overview of your upgrading options and serves as a guide to evaluating your application portfolio. It will help you assess the reasons for and against upgrading applications to Visual Basic .NET. It also explains the role of the Upgrade Wizard and the issues that you need to be aware of when considering upgrading any project. The last part of the chapter describes the decision process for selecting projects to upgrade and criteria for evaluating the suitability of your applications.

Upgrading Is Optional

When evaluating your arsenal of applications, it is important to keep in mind that upgrading to Visual Basic .NET is purely optional. It is highly likely that you will have at least some applications that you will never upgrade. Not all applications will benefit from the new features in Visual Basic .NET. Some applications work perfectly already and there is no reason to change them. When evaluating an application for upgrading, the following four upgrade options are available to you:

- Leave it in Visual Basic 6.
- Perform a partial upgrade.
- Perform a full upgrade.
- Perform a partial or full upgrade with interoperability.

Don't Upgrade

Leaving an application in Visual Basic 6 is by far the easiest option. Although you forgo the benefits of Visual Basic .NET and the .NET Framework, you still

have an application that runs on a supported platform. This option can be a long-term or short-term one, depending on your business requirements. You may want to put off upgrading for the immediate future, or you may decide that there will never be a need to do it. The one important drawback, worth repeating, is that .NET is the future of the Windows platform. Keeping an application in Visual Basic 6 effectively relegates it to an obsolete platform.

Partial Upgrade

When your application consists of at least two distinct projects (an executable file and one or more ActiveX DLL projects, for example), you might perform a partial upgrade. A partial upgrade occurs when one or more components of an application are upgraded, but not the entire application. This partial upgrade is often a good first step toward a full upgrade, and it gets you many of the benefits without the effort of a complete upgrade.

A partial upgrade involves using the COM interop layer to communicate between the Visual Basic 6 COM and Visual Basic .NET managed components. An example would be a two-tier application using a client-side forms-based project with an ActiveX DLL component containing business logic and data access. You could upgrade the user interface while leaving the ActiveX DLL in Visual Basic 6. The components in the DLL are then accessed by a COM reference in Visual Basic .NET that looks like any other Visual Basic .NET component. Don't let this fool you, though. COM interop does a lot under the covers to make this all possible.

What Is Managed Code?

Throughout this chapter we use the term *managed*. **Managed** is the common term used to describe any code that runs under the common language runtime. It refers to the fact that the runtime automatically "manages" memory, security, versioning, array bound checking, and other run-time services—a capability programming languages traditionally have not provided. For example, both Visual Basic .NET and C# create managed executables.

Unmanaged or *native* code is any code that runs outside the runtime. Visual Basic 6 creates "unmanaged" or "native" executables, that is, code that runs outside the common language runtime. C++ .NET is unique in that it can create managed executables, unmanaged executables, and executables that contain both managed and unmanaged code.

A partial upgrade is the most likely scenario for your large-scale legacy applications. You may have components that you do not want to change or that cannot easily upgrade to the managed world. By performing a partial upgrade, you get the benefit of improved performance and scalability in the components that require it. A partial upgrade is also the fastest upgrade path because you can choose to leave the difficult-to-upgrade code as is. This approach preserves your investment in existing code and minimizes churn in your application. It does have a cost, however, in that the performance improvements will not be as noticeable as in a fully managed upgrade because of the added overhead of the COM interop marshaling layer. COM interop between .NET components and COM components is slightly slower than managed components working together or COM objects working together. This performance impact is not noticeable in most applications—highly scalable Web sites and applications that do thousands of calls per second to COM methods are the most affected. Chapter 9 gives you more information on COM interop, including how it works and factors that affect its performance.

Applications with a dependency on COM components are harder to deploy and maintain than applications fully upgraded to .NET since COM objects need to be registered and have versioning issues that .NET objects do not.

Complete Upgrade

A complete upgrade is a purebred. It also requires the most effort. It entails not only upgrading code to Visual Basic .NET but also upgrading all of the technologies to their .NET equivalents. An application that qualifies as a complete (or fully managed) upgrade does not use any COM objects; it uses only the .NET Framework. This option is probably the most attractive for developers, and yet it is also the most elusive for larger, more complex applications. More often than not, with a large code base, portions of your application will need to be rewritten or accessed through the COM interop layer.

Upgrade with Interoperability

Interoperability is a good middle-of-the-road option. It usually means that your application has been upgraded to Visual Basic .NET, while core technologies remain in COM components. Whether the core code resides in a custom DLL of your own making or in other components, like ActiveX Data Objects (ADO) or Microsoft XML Parser (MSXML), the COM interop layer still comes into play. In other words, even though your entire application is in Visual Basic .NET, it is not a pure managed application, and, again, there are certain implications for performance and scalability.

Role of the Upgrade Wizard

The Upgrade Wizard imports your Visual Basic 6 applications into Visual Basic .NET. The wizard is activated when you open a Visual Basic 6 project from Visual Basic .NET. Unlike moving from previous versions of Visual Basic to Visual Basic 6, upgrading to Visual Basic .NET is hardly a simple operation involving only minor changes. The Upgrade Wizard has to make significant structural and functional changes to your code. These changes are necessary because of a number of differences in the structure of Visual Basic .NET. You should not expect the Upgrade Wizard to produce an application that works perfectly. In the vast majority of applications, you will need to make modifications and fixes to get everything compiled and running smoothly.

Many of the technologies used in your applications will upgrade easily to their equivalents in the .NET Framework via the Upgrade Wizard. Certain Visual Basic 6 project types are not supported, however. Two project types that do not map to Visual Basic .NET and are not supported by the Upgrade Wizard are ActiveX DHTML page projects and ActiveX document projects. You will have to either rewrite or manually convert these project types to another project type before upgrading.

The Upgrade Report

The Upgrade Wizard produces a report that highlights problem areas in your code that it encountered during the upgrade process. This report includes items that the wizard left alone because they could not be upgraded. The wizard also inserts comments into your code to help guide you through the remainder of the upgrade and provides pointers to additional documentation. Figure 3-1 shows a sample upgrade report.

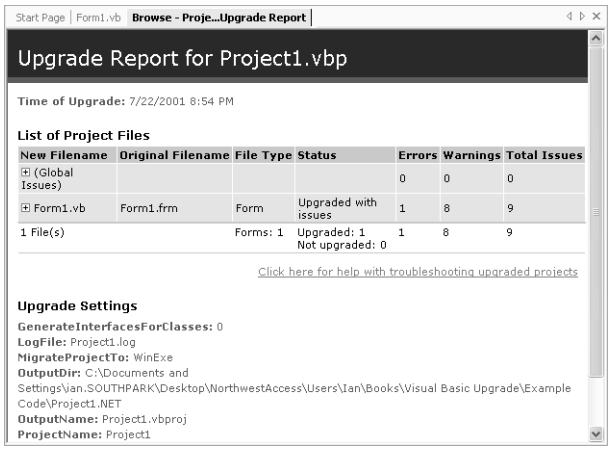

Figure 3-1 An upgrade report.

Not Just a One-Trick Pony

Without a doubt, the Upgrade Wizard is the best way to move applications from Visual Basic 6 to Visual Basic .NET. Although it is possible to do the work by hand, the Upgrade Wizard can do most or all of the grunt work for you. It enables you, the developer, to focus on a smaller number of details rather than trying to figure out how to cram Visual Basic 6 classes, modules, and forms into a Visual Basic .NET equivalent.

In addition to making the upgrade process easier to manage, the Upgrade Wizard can also be used to analyze existing applications so that you can improve them beforehand, smoothing the eventual upgrade. The upgrade report highlights all of the issues that your application will face. The issues you discover will also help you determine the effort that will be required to complete the upgrade before you start down the upgrade path.

Testing

Testing is extremely important for any application. It is even more important when an application has been heavily modified, such as when you use the Upgrade Wizard. It is dangerous to assume that upgrading a functional Visual Basic 6 application to Visual Basic .NET will result in an application that behaves in exactly the same way after the upgrade. After all, the application is now running in a different environment, and significant changes have been made to the source code. Even though Visual Basic .NET controls and objects are similar to their Visual Basic 6 equivalents, there are differences (in both interfaces and behaviors). These differences place the onus on the developer to thoroughly test the upgraded application to ensure that the desired result has been achieved.

The key point here is to test, test, and test again.

Upgrading from Earlier Versions of Visual Basic

We've talked about Visual Basic 6. What about backward compatibility with versions 5 and earlier of Visual Basic? The answer is simple. If you are upgrading any projects from Visual Basic 5 and earlier, you're on your own. The Upgrade Wizard only supports upgrading Visual Basic 6 projects. Interestingly, the format of Visual Basic 5 and Visual Basic 6 projects is virtually identical.

Although the Upgrade Wizard does understand and will attempt to upgrade Visual Basic 5 projects, this is not a supported scenario—some Visual Basic 5 ActiveX controls will not work in Visual Basic .NET. The Upgrade Wizard team has focused solely on testing for Visual Basic 6 projects. If the Upgrade Wizard works well with your Visual Basic 5 project, consider it a bonus. Not everyone will be so lucky.

While it is possible to open Visual Basic 5 projects with the Upgrade Wizard, converting them to Visual Basic 6 first can make the move much easier. To do this, open the Visual Basic 5 project in Visual Basic 6, choosing the Update Controls option. This changes all of your controls to Visual Basic 6 controls (if they are available). Making this step also means that the Upgrade Wizard will generate better and more predictable results.

For projects older than Visual Basic 5, the best option is to upgrade them to Visual Basic 6 first. When you have them working, use the Upgrade Wizard to upgrade the Visual Basic 6 version of the project to Visual Basic .NET.

Selecting Projects to Upgrade

Selecting projects for upgrading is generally a straightforward process. Some applications are no-brainers (they will be upgraded no matter what); others are trivial enough that the upgrade process can be handled in a matter of days. On the other hand, more-complex applications can require much more thought and consideration. Multitiered applications using a wide variety of technologies or applications that are just really big can require an upgrade process that is accomplished in stages over an extended period of time.

For most applications, the selection process goes like this:

1. Evaluate the benefits of upgrading the application to Visual Basic .NET. What features do you need or would you like to take advantage of?
2. Evaluate the effort required by the upgrade. What will have to be rewritten or modified?
3. Develop an upgrade plan. Will the upgrade be handled in stages, or will it be done all at once? Will some or all COM objects be upgraded to their Visual Basic .NET equivalents?

Figure 3-2 illustrates this process.

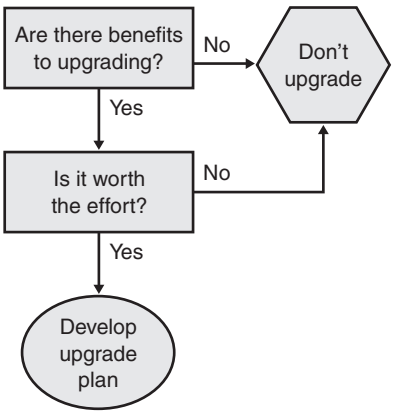

Figure 3-2 Upgrade decision process.

Evaluating the Benefits

To evaluate the benefits of upgrading, you must carefully examine your application and its architecture, components, and business requirements and determine an upgrade strategy. Ask yourself if it is worth just upgrading to Visual Basic .NET without moving from ADO to ADO.NET or moving from the SOAP toolkit to XML Web services, for example. Decide whether upgrading in stages makes sense.

Whatever governs the project selection process, it is important to keep "value" at the forefront of your mind. Ask yourself whether the upgrade will enable your application to support new features and technologies that provide benefits. Ultimately, if there is no value in either the upgrade experience or the result, you are probably dealing with a project that will live on perfectly happy as a Visual Basic 6 application until it is replaced or circumstances change. The five most common reasons to upgrade to Visual Basic .NET are

- To standardize all your applications on a single platform
- To upgrade a pet project
- To take advantage of new language features and improved application architecture
- To increase performance and scalability
- To take advantage of the .NET Framework

The Value of Standardization

While using standardization as a justification for upgrading a project may seem somewhat facetious, it is often important to larger organizations. It's not unrealistic to assume that a good portion of new Visual Basic development within your organization will be done using Visual Basic .NET. If this is the case, it makes sense to move applications to a single consistent platform. Savings on training is one of the most obvious reasons, but using one platform also enables developers to move between applications without having to switch between their Visual Basic 6 and Visual Basic .NET hats.

The Infamous Pet Project

Let's face it: We all have pet projects. In fact, the majority of critical business applications started as somebody's pet project at one point or another. Most of us want our "pets" to survive, grow, and thrive.

New Features, New Language

While it may have some detractors, Visual Basic .NET is pretty cool. It includes a whole host of features that finally make Visual Basic a player in the object-oriented world. Sometimes the benefits of upgrading to Visual Basic .NET are immediate and obvious. A case in point is inheritance. We are all familiar with the practice of implementing common interfaces and code within Visual Basic:

1. Open a class file.
2. Select all and copy.
3. Create a new class file and paste.
4. Make your changes to adapt this copied class to your needs.

The problem is that this approach generates a whole lot of duplicate code and is a maintenance headache. Inheritance in Visual Basic .NET offers a much better solution.

We found that implementing inheritance in one project containing a host of similar classes that represented various related functional tasks resulted in a 75 percent reduction in overall code size. The benefits were immediate and tangible. Bug fixes were simplified and required less work to implement. (They needed to be fixed in only one place instead of in many.) Upgrading also made it possible to implement a consistent error-handling scheme across all classes, resulting in a much cleaner project.

Performance and Scalability

Do you have a performance-critical application? Does your application need to be scalable, and is performance key to the success of the application? If so, you will definitely want to upgrade your application to Visual Basic .NET. Visual Basic .NET and the .NET Framework have features that can improve scalability

and performance in most applications when used correctly. The ADO.NET data access object model, for example, provides in-memory local data caches and a data reader object that is optimized for speed. Also consider that ASP.NET pages are compiled and have improved caching mechanisms and dramatically enhanced session state management capabilities.

Two examples of major contributors to improved application performance are the *SqlDataReader* and *StringBuilder* utility classes. Let's take a quick look at what these features can do for you.

Fast SQL Server Access

The *SqlDataReader* class is analogous to the Forward-Only cursor *Recordset* from ADO. It is a network-level data provider that offers a low-overhead object model and fast queries to Microsoft SQL Server databases. The result is that you can often double your data performance just by substituting *SqlDataReader* for a classic ADO *Recordset*. For more information on upgrading your existing ADO code to ADO.NET, see Chapter 20.

String Concatenation

Not everyone realizes that a major problem area for performance in Visual Basic 6 is string concatenation. This is a very wasteful operation with no real alternatives, other than rolling your own solution (usually as a C++-based COM component). In Visual Basic .NET, it is still possible to do the same old string concatenation, but there is a new utility object that provides fast and efficient string manipulation: *StringBuilder*. *StringBuilder* is a class found in the System.Text namespace.

How Fast Is Fast?

Here is an easy example that demonstrates the performance difference between string concatenation and *StringBuilder*:

```
Sub StringTest()
   Dim before, after As DateTime
   Dim diff As Double

   before = Now()
   SlowString()
   after = Now()

   diff = (after.Ticks - before.Ticks) / Math.Pow(10, 7)
   Debug.WriteLine("Standard concat: " & diff)
```

(continued)

How Fast Is Fast? *continued*

```
   before = Now()
   FastString()
   after = Now()

   diff = (after.Ticks - before.Ticks) / Math.Pow(10, 7)
   Debug.WriteLine("String Builder: " & diff)
End Sub

Const loopMax As Integer = 20000
Function SlowString() As String
   Dim str As String = "MyString "

   Dim i As Integer

   Dim result As String = ""
   For i = 0 To loopMax
      result &= str
   Next

   Return result
End Function

Function FastString() As String
   Dim str As String = "MyString "

   Dim i As Integer

   Dim result As New System.Text.StringBuilder()
   For i = 0 To loopMax
      result.Append(str)
   Next

   Return result.ToString()
End Function
```

When we ran this code on a laptop, the results showed that using *StringBuilder* was more than 200 times faster than simple string concatenation. In other words the *FastString()* method can be called approximately 200 times in the same time period it would take *SlowString()* to complete just once. An added benefit is that the *FastString()* method required less memory than the *SlowString()* method.

When you're concerned about scalability and performance, Visual Basic .NET definitely has a lot to offer.

Plenty More Where That Came From

Of course, the performance improvements are not limited to the *SqlDataReader* and *StringBuilder* classes. They are merely two examples from a vast application framework that offers much, much more. Isolate the performance-critical areas of your application, list the technologies that are used, and then research the .NET equivalents. Most of the time, options exist that can dramatically improve the application's performance.

> **Note** Because of the architectural differences in the common language runtime, ASP.NET, and ADO.NET, it is possible, with a fully upgraded application, to see up to a fivefold increase in performance over an equivalent ASP/Visual Basic 6/ADO application based on real-world systems. A twofold gain is far more typical, however.

The .NET Framework

We've talked about the performance advantages of Visual Basic .NET, but the .NET Framework also has many compelling features that are not exclusively performance related. XML Web services are an excellent example of such a feature. Others are remoting, reflection, and a GDI+ graphics model, for starters.

Moving On

That sums up the benefits of upgrading to Visual Basic .NET. If you're still planning to forge ahead, it's time to move on to evaluate the effort that the upgrade will require.

Evaluating the Effort Required

Four main physical factors determine an application's suitability for an upgrade:

- Architecture
- Code quality
- Technologies
- Size

The sections that follow discuss each of these factors in turn.

Application Architecture

The overall application architecture will play a large part in the determination of whether and how to upgrade an application. An application that is Web based, has a simple architecture, and demands high availability and responsiveness

would be a good candidate for a complete upgrade. On the other hand, if the application is form based and has any or all of the following attributes, you might prefer either to take a tiered approach to the upgrade or to forgo the upgrade altogether:

- Has a small user base
- Uses the Visual Basic 6 drawing model
- Contains many unmanaged Win32 API calls
- Contains old code that uses obsolete language elements

It may be that some or all of your Visual Basic 6 applications can take advantage of the new features in Visual Basic .NET. You may even want to convert your legacy client/server applications to a thin Web-based client model with the improved Web controls.

Although many variations are possible, the following are the four general types of Windows application architectures.

ASP Web applications This type of application is usually performance driven and needs to be scalable to support hundreds or thousands of concurrent users across the Internet or intranets. It is one of the simplest kinds of applications to upgrade to ASP.NET because of the similarities between the two technologies. ASP components on the client end can, with some effort, be converted to ASP.NET using any of the managed languages. Middle-tier components written in Visual Basic 6 can be upgraded using the Visual Basic .NET Upgrade Wizard, the data access components can be converted to ADO.NET, and any third-party components can be accessed through the COM interop layer. By their nature, these applications depend on performance and scalability—two things the .NET Framework delivers. All other factors being constant, this type of application will probably see the biggest benefit from an upgrade. The one downside to upgrading ASP applications is that there is no upgrade tool to do the grunt work for you. The middle-tier components can take advantage of the Upgrade Wizard, but the ASP pages all need to be upgraded by hand.

Forms-based client/server applications A typical application that uses this architecture consists of a fat client, possible middle-tier COM components with business logic, a data access layer, and a connection to a data store. These applications are different from thin client/server applications in that some of the processing and business logic resides in the client. Fat-client systems come in all shapes and sizes, some with ActiveX controls and active documents embedded in a Web browser and others with client-installed executables, COM components, and third-party controls. Unless their design is fairly straightforward, these applications require more work to upgrade to the .NET Framework. Since

the middle-tier components are much the same as in a thin-client system, the upgrade of the middle tier will not pose any more of a problem than upgrading a thin-client system. However, upgrading the client may be more difficult than the transition from ASP to ASP.NET for thin-client systems. The difficulty in upgrading the client lies in the fact that Visual Basic has been completely redesigned for Visual Basic .NET, and the functions and controls do not map exactly to their counterparts in Visual Basic 6.

Enterprise architecture with legacy or distributed systems Applications that contain back-end legacy systems, such as mainframes, can greatly benefit from Visual Basic .NET and its new features. Rewriting portions of the system to take advantage of the new XML Web services application architecture can improve code modularity and maintainability and can also increase accessibility to those legacy systems. With XML Web services, it is possible to expose and consume the functionality and data of the legacy system essentially by writing a wrapper component that exposes its interfaces through an XML Web service. This allows local and remote clients or components to easily interact with and program against the legacy system. The problem here is that any performance or scalability benefits would be hard to achieve, since the legacy systems are more often than not the performance bottlenecks. Although there might not be a huge performance benefit, your application can benefit architecturally by becoming more distributed and more flexible. XML Web services enable this by using SOAP as the underlying implementation, which is more flexible than DCOM in many ways.

Stand-alone applications Stand-alone applications are typically form-based applications such as games, graphics programs, controls, or general-purpose utilities. Simple applications are typically easy to upgrade.

Code Quality

Code quality will have a great deal of influence on an upgrade strategy. Some of the factors that will make your Visual Basic 6 application easier to upgrade include a modular, object-oriented design and good programming etiquette. This is not to say that applications that do not follow these principles are not upgradable, but the task will involve more effort. As a test, you can run the Upgrade Wizard and see what issues it discovers. See Chapter 4 for a more in-depth treatment of good coding practices.

Technologies

The technologies or Visual Basic 6 features that an application uses also affect the amount of effort required to upgrade an application. A typical area of difficulty involves the use of either many Win32 declared functions or the Visual Basic 6 graphics model. The Upgrade Wizard can't really do anything with the

declare statements, so it will leave them alone. But they should still work, with a few exceptions (which will be highlighted by the Upgrade Wizard). The Visual Basic .NET graphics model, on the other hand, is so different from what was available in Visual Basic 6 that there is no direct or indirect equivalent. Any code that uses the Visual Basic 6 drawing model will have to be rewritten. There is just no way around this.

Application Size and Scope

This point may be self-evident: a larger code base will take more time to upgrade. When you're becoming familiar with the .NET Framework, it is best to start with smaller applications. That will give you a chance to become acquainted with the new constructs, and debugging problems will be easier. Take the code size into consideration when screening applications to upgrade.

Developing the Upgrade Plan

Once you have determined which projects to upgrade, the rest of the process becomes a whole lot easier. Keep one thing in mind when working out your upgrade plan: If your application consists of a number of projects, some of which depend on others, upgrade the client project first. Next upgrade the project the client depends on. If this project in turn depends on another project, upgrade the more fundamental one next, and so on. Doing the upgrade in this order means you are moving up the dependency hierarchy. Why choose this order? For the simple reason that it's the easiest upgrade path. Using COM components from Visual Basic .NET is simpler than using .NET components from Visual Basic 6. The latter option requires changing both the .NET component and the COM component at the same time. It also means dealing with GUID and versioning issues that can make COM development a complicated process. Starting with the client and moving up the dependency hierarchy keeps all of the COM GUIDs the same and avoids COM versioning issues.

Keeping that rule in mind, you'll find that the rest of the process is simple. Break your application down into its core components, and work out the dependencies. Starting from the client side, decide what components you will upgrade and in what order, and the extent to which each of them will be upgraded (none, partial, full, interop). Large applications will no doubt be handled in stages, since attempting to pull off a full upgrade of all components would be extremely disruptive and would lead to all sorts of testing problems and new bugs. Try to go step by step, testing as you go, to ensure that you maintain a high level of quality.

Conclusion

You have a lot of choices facing you. Not all applications are well suited to being moved to Visual Basic .NET. Others practically compel the move. What you should take away from this chapter is that you should consider applications for upgrading on a case-by-case basis. Of course, there are many good reasons for upgrading applications; just don't feel that you are being forced to do so (unless, of course, you are). Upgrade only if and when it makes sense. When it does make sense, develop your upgrade plan and determine where you need to focus your efforts.