Realtime publishers

The Definitive Guide™ To

Building Code Quality

sponsored by

Don Jones

Introduction to Realtime Publishers

by Don Jones, Series Editor

For several years now, Realtime has produced dozens and dozens of high-quality books that just happen to be delivered in electronic format—at no cost to you, the reader. We've made this unique publishing model work through the generous support and cooperation of our sponsors, who agree to bear each book's production expenses for the benefit of our readers.

Although we've always offered our publications to you for free, don't think for a moment that quality is anything less than our top priority. My job is to make sure that our books are as good as—and in most cases better than—any printed book that would cost you $40 or more. Our electronic publishing model offers several advantages over printed books: You receive chapters literally as fast as our authors produce them (hence the "realtime" aspect of our model), and we can update chapters to reflect the latest changes in technology.

I want to point out that our books are by no means paid advertisements or white papers. We're an independent publishing company, and an important aspect of my job is to make sure that our authors are free to voice their expertise and opinions without reservation or restriction. We maintain complete editorial control of our publications, and I'm proud that we've produced so many quality books over the past years.

I want to extend an invitation to visit us at http://nexus.realtimepublishers.com, especially if you've received this publication from a friend or colleague. We have a wide variety of additional books on a range of topics, and you're sure to find something that's of interest to you—and it won't cost you a thing. We hope you'll continue to come to Realtime for your educational needs far into the future.

Until then, enjoy.

Don Jones

Copyright Statement

Chapter 1: Quality Coding for Visual Studio and the .NET Framework

There are many things that contribute to the quality of an application. From the initial requirements through the functional specifications, data modeling, security considerations, and choice of interfaces and support services all contribute to how well the application performs, how easy it is to maintain, and how well people will adopt it. But for all these factors, nothing is as directly telling as the quality of the code itself. A set of crystal clear requirements, a brilliant architecture, the most friendly of user interfaces (UIs)s is nothing if the code does not perform reliably, errors are not well handled, and the code does not perform the expected tasks quickly.

In this guide, I intend to walk you through the process of building a quality coding practice. I'm assuming that you're either a professional developer yourself, or that you manage developers as a major part of your job. I'm assuming that you've worked with recent versions of Visual Studio quite a bit, too. From getting the most from the development environment to writing clear and efficient code to analyzing and eliminating errors before they become issues to making the applications run at peak efficiency, this guide will help you refine your code practice and churn out better code faster.

This guide isn't just about indenting your code and how to capitalize variable names; we'll also tackle more complicated topics such as the right way to conduct a peer code review, how to address security and performance problems, and more—everything that contributes to quality code. We'll begin by making sure we're on the same page with toolset and technologies.

.NET Framework and Visual Studio Overview

Microsoft introduced the .NET Framework in early 2002. It represented a significant change in Microsoft's software development strategy, moving away from *native code* that was compiled for a specific hardware and operating system (OS) toward a more Java-like model that offered the possibility for broader platform support.

> **Note**
>
> *Native code* is a binary that is loaded directly into the memory of the processor and executed. Microsoft Win32, COM, and DCOM applications are compiled in this manner. Native code is still used for the OS and device drivers. It can run fast but is difficult to write efficiently and without errors.

To simplify and secure the process of writing code, .NET uses a type of virtual machine, called the Common Language Runtime (CLR), as a buffer between the processor and the application. This layer can make things run more securely. Some of the coding problems that would cause a computer to crash are mitigated by this approach.

.NET is purely object-oriented, supporting full encapsulation, inheritance, and polymorphism. It forces the declaration of variables and a strict object life cycle (a problem that frequently plagued COM and created memory leaks). None of the Microsoft languages prior to this, with the exception of J#, was purely object-oriented. Object-oriented languages are not inherently of higher quality but they offer many features—and force a certain programming style—that make it easier to write high-quality code.

.NET introduced structured error handling for *all* languages—not just a bolt on as it was in earlier languages such as C++. Structured error handling is a key capability for writing higher-quality code; without it, your code cannot respond as consistently to error conditions.

In addition to the Framework itself, Microsoft created new versions of its software development tools, including Visual Studio integrated development environment (IDE). The Framework and Framework-based development aren't entirely without its critics, and Microsoft itself has run into significant hurdles in using the Framework to create OS components.

One problem with the Framework is semantics: The term ".NET Framework" is used generically to refer to several distinct elements, each of which is important to code quality in a different way. In order to enable our discussion on code quality, and to make it clearer, we need to first break down the Framework into its constituent elements.

The CLR and Related Languages

When developers speak of programming for the Framework, they're usually writing code in either the C# or Visual Basic (formerly "VB.NET") languages—although other languages are available (Microsoft provides C++ and J#; other manufacturers provide other languages). The beauty of this approach is that the programmer can write in the language that he or she is most comfortable with. If you need to port code from a COBOL, APL, Fortran, or other system, that code can be used within the .NET Framework, providing that you have support for that language.

The language in which the program is written is NOT the language that the Framework uses. When a developer *builds* (or *compiles,* although that term is technically inaccurate) a C# project, what's produced is not an executable capable of running independently on a computer's OS. Instead, Visual Studio's main job is to translate your Visual Basic, C#, J#, or whatever code into the Common Intermediate Language (CIL).

> **Note**
>
> You'll still see the term *MSIL* pretty often, although since the language specification was formalized and standardized as ECMA-335, the term *CIL* is more correct.

Here's an example of CIL code:

```
.method static void main()
{
    .entrypoint
    .maxstack 1
    ldstr "Hello world!"
    call void [mscorlib]System.Console::WriteLine(string)
    ret
}
```

It's actually a pretty unintuitive, hard-to-read language, which is why few developers work in it directly (for the old school audience, this is a lot like Assembly language). Once your code has been converted to CIL, it's further assembled into *bytecode* (which is essentially a binary compact form of shorthand for CIL), creating a *.NET assembly*—typically an EXE or DLL file, depending on how it will be executed. Figure 1.1 illustrates this process.

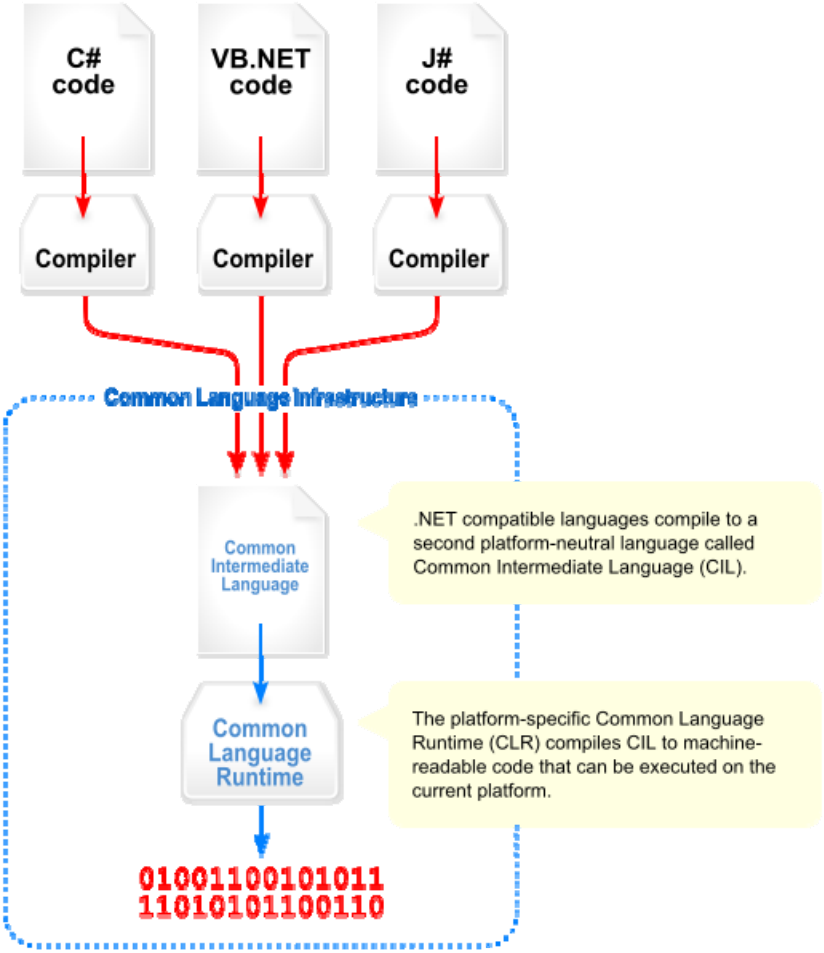

Figure 1.1: Compiling .NET Framework code.

When a user runs an assembly containing CIL, the Framework's CLR springs into action. The CLR receives the bytecode from the assembly and executes a Just-In-Time (JIT) compiler that generates code capable of executing on the platform within the CLR. This model allows code to be written once and to be executed on any platform for which a CLR is available. There is a performance hit while the JIT converts the bytecode to executable binary, but it caches the resulting native image for future use (it automatically recompiles when the source assembly is seen to have changed). A Framework utility called NGEN can be used to generate native images in advance; this process eliminates the initial performance penalty but ties the native image to a single platform. That is one reason, for example, Windows PowerShell (which is written in the Framework) is distributed in different versions for different Windows OSs.

The CLR acts as a type of *virtual machine.* The CLR is responsible for executing Framework applications, handling their memory usage, managing garbage collection, handling exceptions, managing security, and so forth. Unlike Java, which typically permits very limited access to the underlying host OS, the CLR is designed to provide access to the native Windows OS features—although it does so only through a strict set of security controls.

The .NET Framework Class Library

An integral part of .NET development is the .NET Framework class library—what we can properly refer to as "the Framework." This is an extensive collection of preprogrammed classes that give developers much of the functionality they need to write applications. Framework classes exist to connect to databases, create graphical user interfaces (GUIs), and so forth; developers just—to oversimplify a bit—connect the various classes to one another to create applications. Thus, developers are less concerned about low-level details such as how to open and read a file, and more concerned about high-level functionality, such as implementing business logic.

The Framework is extensible. Additional classes can be added to cover functionality that the Framework itself doesn't address and that a developer doesn't want to create manually. Classes for remote connectivity, creation of charts and graphs, alternative UI elements, and so forth are all popular.

Each version of the Framework class library is unique, and developers must *target* a specific version when creating their applications. Framework versions are not necessarily cumulative nor are they necessarily cross-compatible. For example, if you create an application that uses features from only v2 of the Framework but your project targets v3.5, your application will run only if v3.5 of the Framework is installed even though v2 technically contains all the needed functionality.

Each version of the Framework tends to add new classes to what came before; each version may also contain changes to functionality introduced in previous versions. Figure 1.2 shows the major portions of the Framework as of v3.5, along with initial plans for future versions. As you can see, the Framework is truly enormous—much larger and more all-inclusive than similar class libraries for languages such as C++.

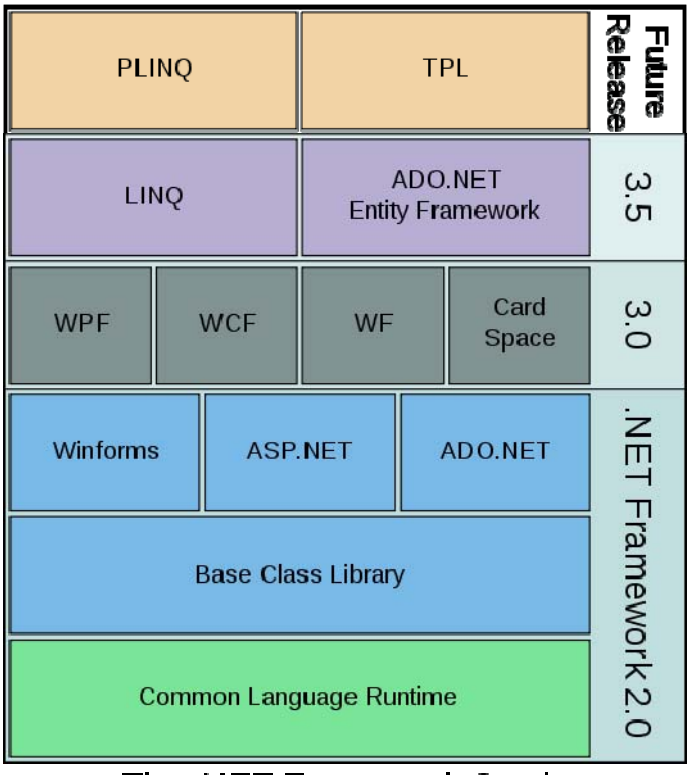

Figure 1.2: The .NET Framework class library.

Visual Studio

Although you could technically create .NET applications using nothing more than Windows Notepad and the freely available language compilers, you probably wouldn't enjoy it. An IDE provides numerous features to make development easier and faster, and Visual Studio is the de facto IDE for most .NET developers. Available in various editions that offer different features and functionality, Visual Studio is designed for .NET development (as well as for developing unmanaged C++ applications). It includes a core code editor and project management tools, and in some editions, offers features such as source control connectivity, testing tools, and so forth. Visual Studio is extensible; while Microsoft directly supports Visual Basic, J#, C++, and C#, third parties have extended support for languages such as Java, COBOL, Fortran, APL, Chrome, F#, Python, and Ruby. Free "express" editions support only a single language—either Visual Basic or C#, in most cases, although "express" editions for J# and C++ are also available.

Figure 1.3 shows Visual Studio, and illustrates one of its major IDE features: IntelliSense, Microsoft's brand name for its code-hinting and code-completion features.

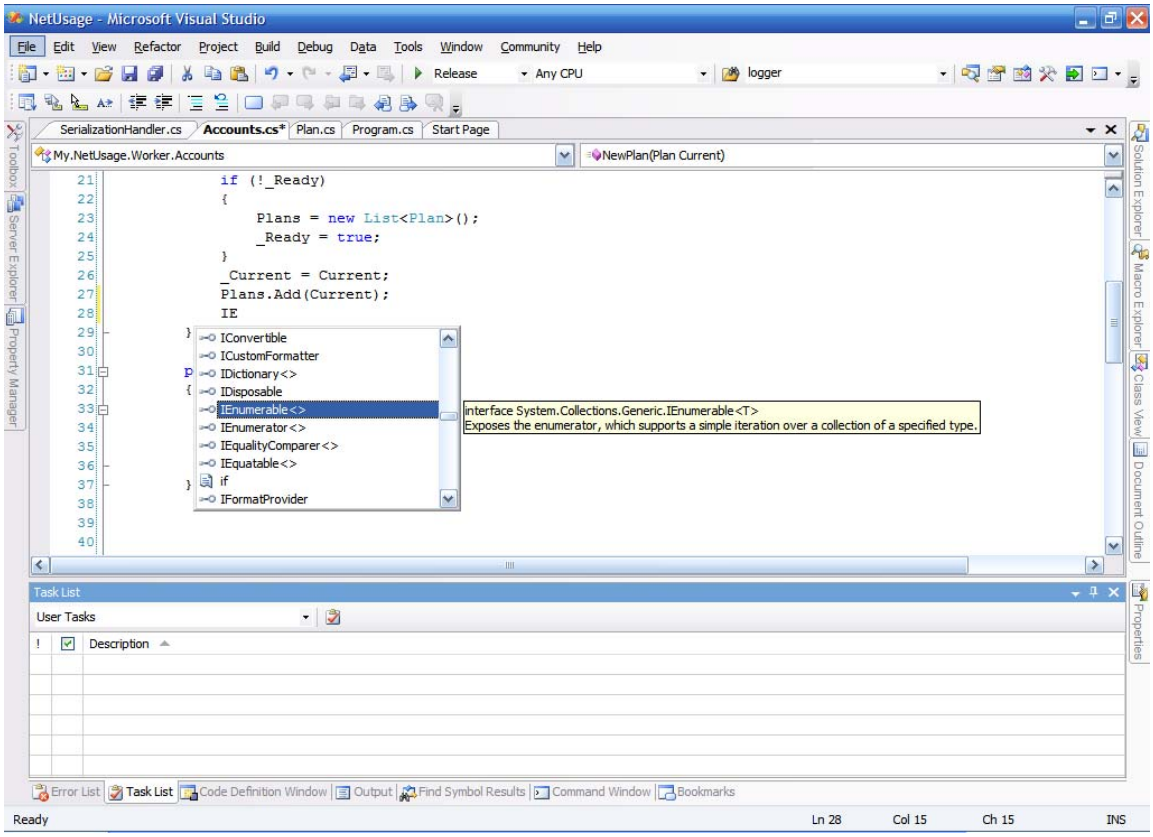

Figure 1.3: Visual Studio.

We'll be discussing Visual Studio in more detail in just a bit.

Issues with .NET Development

The Framework isn't entirely without issues and critics, and if we're going to use it to build quality applications, we need to consider some of the major criticisms and understand how they may impose limitations on our own ability to deliver quality code.

Applications that run in a managed environment—which is what the CLR is—tend to run a bit slower or require more system resources in order to run as quickly as applications that run natively on the OS. This performance issue, in fact, was a contributor to Microsoft's difficulties delivering on technology promises such as WinFS and Avalon (a file system and graphics subsystem based on managed code) for Windows Vista. This is an important consideration because you, as a developer, have little control over the performance hit introduced by the CLR; if a certain set of performance metrics are mandated in your software's requirements, you could run into situations in which you can't optimize the performance further because the CLR is acting as a bottleneck of sorts. You're unlikely to find situations where you can't produce *acceptable* performance, but depending on the aggressiveness (and realism) of your required performance metrics, meeting those metrics may be challenging.

Framework assemblies contain a bytecode version of CIL, meaning you are essentially distributing your source code. Numerous utilities exist that can "de-compile" a .NET assembly into C#. In fact, most of the actual Framework classes can be decompiled, as Microsoft relies on intellectual property (IP) law to protect its source code. Smaller developers with fewer lawyers often turn to *obfuscation* tools, which perform various semantic tricks to render decompiled code less readable. Modern versions of Visual Studio, in fact, come with a "community edition" of a popular commercial obfuscator, giving developers a baseline obfuscation capability and tacitly acknowledging the downside of bytecode assemblies.

Managed environments such as the CLR must periodically clean up after the applications they run. This process, called *garbage collection,* involves freeing up and helping to de-fragment memory that is no longer in use by the application—such as memory that had been allocated for variables that are no longer in scope. Garbage collection halts execution of the application for a bit, and although this time period is usually imperceptible, it can in some instances be perceived as poor application quality. Developers must take some care to deal with memory according to best practices so that garbage collection doesn't become an impediment to the application.

The exact version of the Framework you develop for is important. Older versions, such as v2, are fairly ubiquitous, having been around for some time and having been preinstalled on most modern versions of Windows. Newer versions may not be available on all client computers, meaning that running your application first requires users to install the proper version of the Framework. Visual Studio has traditionally been poor at allowing developers to target a specific version of the Framework; Visual Studio 2005, for example, was introduced with v2 of the Framework and doesn't really "like" to compile assemblies that are compatible with v1.1 of the Framework—even though it *could* create such assemblies.

The Framework itself adds a great deal of overhead to your application's footprint: up to 197MB for v3.5 of the Framework. Although this is mitigated by the fact that multiple applications can share the same Framework installation, users who need to have v1.1, v2, v3, and v3.5 installed to support various applications are taking a pretty big hit in terms of drive space.

The Microsoft Visual Studio Environment

It might seem like overkill to talk about an IDE in a book about code quality, but the IDE is where you're writing your code, so the IDE is where quality begins. And in fact your IDE can do a lot to help or harm your code quality, depending on how you use that IDE.

A Brief History of Visual Studio

Visual Studio was first released in 1997, bundling many of Microsoft's programming tools for the first time. Prior to that release, products such as Visual C++ and Visual Basic were independent tools with their own, unique IDEs. Visual Studio 97 included Visual Basic 5.0 and Visual C++ 5.0—this was before the .NET Framework was even a glimmer in anyone's eye. Visual Studio 97 also included Visual J++, Visual FoxPro 5.0, and Visual InterDev for Web development as well as Microsoft's MSDN Library documentation.

In 1998, Visual Studio 6.0 was introduced and was the last version created to run under Windows 95, Windows 98, or Windows Me. This was still pre-.NET Framework, with a focus on Component Object Model (COM) development, but it offered a more consistent and unified IDE—although the unified IDE was used only by Visual J++ and Visual InterDev; Visual Basic, C++, and FoxPro continued to use their own distinct IDEs. It originally included Visual J++, but that was discontinued after Microsoft's legal settlement with Sun Microsystems.

In 2002, Visual Studio .NET was released to correspond with the first release of the Framework itself. This introduced the complete transition to CLR-managed code, a common IDE for the included Visual Basic .NET, C#, and J# programming languages as well as Managed C++, a set of extensions to C++ that allowed for .NET development in that language. Visual FoxPro was removed from Visual Studio and placed on its own development track; it is not a .NET Framework language.

Visual Studio .NET 2003 included .NET Framework v1.1 as well as the .NET Compact Framework for mobile devices. This was a fairly minor upgrade in many respects.

A more major upgrade came in Visual Studio 2005, when the ".NET" moniker was dropped from the name (Microsoft had by this time dropped ".NET" from the many things it had been applied to, except the Framework itself). V2 of the Framework accompanied Visual Studio 2005, and project types for ASP.NET Web services were added to the IDE. 64-bit support was introduced, although the Visual Studio application itself was only 32-bit. Subsequent add-ons to Visual Studio 2005 enabled support for .NET Framework v3 features, including Windows Workflow Foundation, Windows Communication Foundation, and Windows Presentation Foundation. This version of Visual Studio also introduced numerous "team" editions with task-specific functionality for application architects, developers, testers, and so forth.

Visual Studio 2008 is the latest version as of this writing and contains numerous upgrades. A new Windows Presentation Framework designer is included, along with a new HTML/CSS editor for Web applications. J# has been dropped from the bundle, and the default Framework target version is v3.5. This is the first version to truly support targeting earlier Framework versions, including 2.0, 3.0, 3.5, the Silverlight CoreCLR, or the Compare Framework. New code analysis tools, including the new Code Metrics tool, are included in some premium editions, and the integrated debugger introduced support for easier debugging of multi-threaded applications.

Note

As of this writing, Visual Studio 2010 is under development.

Visual Studio Development Methods and Techniques

For an experienced developer, Visual Studio is a straightforward IDE. An entire software application can be developed in Visual Studio, whether it's a Windows GUI application, a Web application, a Web service, or whatever; the IDE contains all the necessary visual designer surfaces, code editor support, integrated help, and other features needed to produce a complete application.

The IDE also offers setup and deployment wizards to help build Windows application installers, deploy Web sites to a Web server, and so forth. An embedded Web server allows developers to perform immediate testing of Web applications without needing to deploy a separate Web server, which makes unit testing that much easier and that much more accessible.

Perhaps it should go without saying, but be aware that Visual Studio supports *only* .NET Framework development. It does not explicitly support older Microsoft development languages such as pre-Framework Visual Basic nor does it explicitly support languages such as PHP. Visual Studio is extensible, so it's not unthinkable for it to be extended to support these languages, but generally speaking, Visual Studio users are interested in developing Framework-based applications.

Note

Actually, Visual Studio *does* support development in one unmanaged language: Visual C++. That language continues to enjoy widespread support and use.

Visual Studio includes built-in support for Visual SourceSafe and Microsoft's Team Foundation Server source control capabilities but does not natively support other version repositories such as SubVersion, ClearCase, or CVS.

Note

Many of these versioning systems have their own clients that act as plug-ins to the Visual Studio IDE.

The IDE provides basic refactoring and "code beautifier" capabilities and provides basic support for comparing different versions of a file. These features all help developers create code that conforms to internal standards as well as industry best practices—both of which are important, as we shall see later, to building quality code.

Visual Studio Solutions, Projects and Procedures

Visual Studio's basic unit of work is a *project*, which more or less represents a single application. However, Visual Studio recognizes that many large-scale software solutions actually involve multiple discrete projects. Therefore, a developer may have an entire Visual Studio *solution* open in the IDE at once; this solution can consist of projects for back-end code, Web services, Windows GUI apps, middle-tier components, and so forth, allowing an entire application to be tested within the confines of the IDE.

This capability can offer a great deal of power and flexibility for developers but also requires coordination in order for it to work well. For example, a developer may be working on a particular middle-tier component and may check out the associated files from version control in order to work on them. Another developer may be working on a client application and might have a read-only copy of the first developer's middle-tier component code. This setup allows both developers to work independently, but the second developer will always be working on an out-of-date copy of the middle-tier code. That may be fine for basic unit testing, but at some point, all the latest code needs to come together in a single place, such as a full test environment where daily builds or other agreed-upon units of work are deployed for more formal integration testing. Visual Studio itself doesn't provide tools specifically for managing the inherent disconnect between developers working as a team, although frankly, no IDE really does or could. This challenge is a management issue and provides a good example of why effective management skills and processes are needed in complex development projects.

Issues with Visual Studio Development and Native Toolsets

If Visual Studio has a weakness, or weaknesses, it relates to code testing. Although the Visual Studio Team Test edition offers test-specific functionality, no edition of Visual Studio can be considered a full-fledged tool for application testing. It lacks strong support for test asset management, lacks the security and auditing controls necessary for strong test asset management, and lacks much of the higher-end automation and test management capabilities found in third-party tools.

> **Note**
>
> *Test assets* refers to a variety of resources, including sample data, which is used to test an application's behavior not only to proper input but also its handling of improper input. A major root cause of poor application quality is poor test data: test data that is not real-world does not lead to real-world quality testing. That said, real-world test data may come with significant baggage in terms of sensitivity, security, and privacy, so a test asset management tool must be able to properly secure and audit access to test data. If test data will, for example, include real-world customer data, that data must be treated with sensitivity for its privacy—and in some jurisdictions, all use of that test data must be audited. Visual Studio does not provide this level of management capability for test assets.

Another concern with Visual Studio development, with regards to code quality, involves management reporting. Even when all developers are using the "Team" editions of Visual Studio, there are relatively immature built-in means for a manager to report on developer productivity, review developer unit testing results, review formal results from QA testers, and so forth. Given the role management must play in properly coordinating development and testing and in ensuring the delivery of a quality application, Visual Studio's lack of management tools can be considered significant.

That said, we should recognize that Microsoft wants Visual Studio to be an *IDE,* not a be-all, end-all software development system. Microsoft never set out to make Visual Studio a tool for managers nor did it set out to build enterprise-class test asset management. Microsoft relies on a rich ecosystem of third-party Independent Software Vendors (ISVs) to provide these more business-specific capabilities, and many ISVs do so by integrating their toolsets directly into Visual Studio itself. Any lacking in Visual Studio shouldn't be construed as a failure on Microsoft's part but rather as recognition that software development is *complicated* and often requires tools that are more specific to a given line of business or development model.

Understanding and Assessing Code Quality

Is it possible to look at code and assess its quality without even running the application? Certainly we can assess an *aspect* of the code's quality statically, just as we can assess other aspects by running the application. It's important to understand that, ultimately, code quality is the *combination* of these things.

Yardsticks: Errors, Completeness, Security, and Performance

How do you, and your organization, measure code quality? Businesspeople tend to focus on whether an application does what they need in the way they need it done. Call that *completeness*. Businesspeople also focus—often above all else—on their perception of the application's performance. Are they telling customers, "Sorry, the computers are slow today?" If so, they likely perceive the application as one of poor quality. Too often, in fact, developers aren't armed with a simple tool that would allow them to meet those quality expectations: a list of actual expectations! In other words, if you don't *know* what features and performance are expected, you probably won't deliver them. That's where well-written requirements can help drive better quality, simply by letting you know what's expected of the application.

> **Note**
>
> Remember: Many things can contribute to a poor-quality *application.* Most people will, however, always attribute poor *application* quality to poor *code* quality. Ensuring that the code is of high quality allows you, as a developer, to defend your work more easily.

Most anyone will agree that errors are a sign of poor quality. Obviously, some errors—such as a missing file or an unavailable network connection—are outside your immediate control, but handling those error situations gracefully can at least provide an improved perception of quality. Graceful failure can also help troubleshooters such as systems administrators or help desk staff solve problems faster. If your application can be clear on what the external problem is, troubleshooters can focus on that problem and not on your application. Of course, you need to make sure that the code paths leading to graceful failure are deterministic: displaying a "network error" when in fact the network is fine is a sure way to have your application thought of as "low quality." And no one likes seeing an "Unexpected error" message pop up on their screen.

Security is something many developers don't think of as they're writing code, and unfortunately, it's something that often may not be a problem for some time after an application has been released and is in use. But eventually, if security isn't coded in from the start, it *will* be a problem, and there are few symptoms of poor quality that cost more to deal with. Security isn't fun to code, and in most cases it winds up making applications more complicated and complex than they would be otherwise, but in today's world, a securely coded application is simply a *requirement.*

All of these yardsticks are valuable indicators of quality because they focus on end user perceptions, which are critically important in measuring an application's overall success or failure. However, all these yardsticks *are* very subjective and can be very difficult to measure. For that reason, it is—*in addition to these yardsticks*—beneficial to have more measureable metrics against which to judge the quality of your code.

Commonly Used Code Metrics

To say that there are numerous formal metrics for code quality is something of an understatement. Massive quantities of literature have been published on the topic. One common theme, however, is that measuring bugs *is not* a measure of quality—it's a measure of "non-quality," to coin a phrase. Measuring error density per thousand lines of code over the course of a year doesn't tell you whether your code is good; it only tells you if your code is bad. Not that tracking errors isn't essential; they're just not the be-all, end-all of measuring code quality.

In 1993, the Institute of Electrical and Electronics Engineers (IEEE) published a software quality metrics methodology that was intended as a systematic approach for establishing quality requirements, validating quality metrics, and so forth. There's that word *requirements* again. Figure 1.4 shows the IEEE's basic methodology.

Software Quality Activity	Development Cycle Phasing
Establish software quality requirements	_____
Identify software quality metrics	_____
Implement software quality metrics	_____
Analyze results of these metrics	_____
Validate the metrics	_____

Figure 1.4: IEEE software quality metrics methodology.

Without establishing your *requirements,* you have nothing against which to measure quality! Although various software development methodologies provide means for measuring quality, all begin with well-written requirements, which must define what "quality" means for that application. *Software Measurement: A Visualization Toolkit for Project Control and Process Measurement*, by Simmons, Ellis, Fujihara, and Kuo, was published in 1997 and in many ways is the best treatment of current software metrics. It's a dense tome, but it's worth a read, as it also explores historically verified formulas that do things like predict the number of defects per thousand lines of code based on project size (middle-sized projects tend to have the least bugs, for example). Simmons, et. Al, propose a basic metric set that encompasses reliability, verification, and usability, with each acting as the leg of a triangle:

- Usability is a product's fitness for its intended purpose.
- Verification is the ability to determine whether a product is usable and reliable.
- Reliability is usually defined as bug potential, bug-fix efficiency, and bugs that still exist in "finished" code.

More recently, the *function point* has been proposed as a quality metric because it doesn't rely on simple lines-of-code counts, which rarely provide satisfactory quality metrics. Instead, a function point represents interfaces within the application because interfaces are most often where errors occur. A typical function point developed by IBM looks like this:

- Multiply 4 times the number of external inputs
- Multiply 5 times the number of external outputs
- Multiply 10 times the number of logical internal files
- Multiply 7 times the number of external interface files
- Multiple 4 times the number of online inquiries supported

These are called average weighting factors, or W_{ij}. X_{ij} is the number of each component type in the application. Feed all that info a function like the one shown in Figure 1.5.

$$FC = \sum_{i}^{5} \sum_{i}^{3} w_{ij} * x_{ij}$$

Figure 1.5: Function point metric formula.

Then you use a scale of zero to 5 to rate the potential impact of 14 general system characteristics, with 5 representing a very likely impact on the application overall, and 0 representing a lesser impact.

The system characteristics are:

- Data communications
- Distributed functions
- Performance
- Heavily used configuration
- Transaction rate
- Online data entry
- End-user efficiency
- Online update
- Complex processing
- Reusability
- Installation ease
- Operational ease
- Multiple sites
- Facilitation of change

Add your scores and call the sum Ci, and use the formula in Figure 1.6 to find a value adjustment factor (VAF).

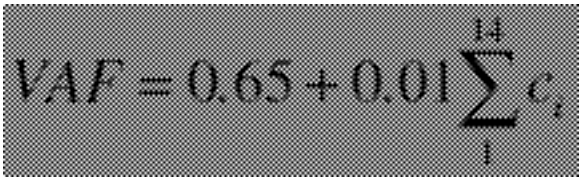

Figure 1.6: Function point metric formula.

Then come up with your final function point value by using the formula FP = FC × VAF. Your "function point value" is a score indicating the potential quality of the software (amongst other things). It's not measuring bugs but rather (in part) the potential for really bad bugs.

> **Note**
> If you can't get enough of the math, check out http://www.informit.com/articles/article.aspx?p=30306, which contains an excellent discussion of current metrics theory.

No kidding. This is why kids in school are told that software development requires an affinity for math. It's also why automated tools exist to help assess software quality.

Automating Code Quality Assessment and Using Code Quality Assessments to Drive Development

With code quality assessments involving so much math, counting, and other fairly tedious tasks, it's not surprising that tools exist to help automate the process. Most commercially available tools offer metrics based on many popular industry standards and methodologies. Assessment tools often produce everything from simple numeric scores to complex reports, including visualizations such as matrix plots, table lenses, and so forth.

Automated assessment tools can be quite complex. In addition to gathering basic information such as the number of lines of code and interfaces between code modules, they may analyze UI code, database code, and so forth, which must actually parse your programming language and make assessment decisions. A benefit of these assessments is that you can do them frequently, as your code grows, and you can use them to identify high-impact areas with a high defect potential. Identifying these areas of your code quickly helps you put more focus on them to help prevent bugs during development and during all phases of testing.

Ultimate Quality: Does It Meet the Requirements?

There's no better measurement of an application's quality than the answer to one simple question: Does the application meet its requirements? And the follow-up questions: Does it perform as required? Offer the functionality required? Meet the required security standards? Respond to errors as required? Without clearly articulated requirements, you *cannot* judge an application's quality.

I live in a nice enough house. It's one-story, has insulated windows, has great insulation in the roof, and isn't too drafty in the wintertime. Is it a quality house? Based on my requirements for energy efficiency, it definitely is. My parents, however, prefer two-story houses, like windows that slide up rather than sideways to open (they're easier to clean, Mom says), and want a garage that's a bit wider than mine. So by their requirements for form and function, my house isn't of very high quality. Without any of our requirements in mind, it's impossible to make a quality statement about the house. Even a home inspector has requirements: Is it up to code? Is there any rot or decay? Is anything broken or out of kilter? Everyone has different measurements for quality.

That's why written requirements are so important for programmers. Without them, you'll never know whether what you're producing is quality. Maybe you've fine-tuned every code to the minimum number of statements, optimized database access, and spent hours agonizing over the GUI color scheme. After all that effort, it's a poor quality application if it doesn't produce the printed output users really need.

The great part about well-documented requirements is that they are *not* subjective. Instead, they serve as a measurable checklist. Either your application *does* everything on the checklist or it *does not.* The more checks, the better the application quality. The *best* part about a well-written set of requirements is that you, as a developer, need to worry a lot less about the things *not* in the requirements! The requirements don't address printed output? Well, don't spend too much time on printing, then, because it's obviously not important to anyone. Instead, you can focus on what *is* in the requirements, do a good job implementing them, and have a final application that everyone can (or should) agree is of high quality.

There are, of course, "under the hood" aspects of coding that don't make it into requirements documents. Good coding practices lead to easier long-term code maintenance, better application stability, better application security, and other factors that often aren't written in a set of requirements—but that are still perceived as signs of poor quality, and so are things you still need to focus on. That's what much of this guide will focus on, too.

Top Code Quality Snafus

The SANS Institute and a collection of computer experts recently published a list of their top-25 code problems—all of which, ultimately, relate to code quality. Although many of these problems relate to security, others also relate to general stability and other issues. It's useful to review this list in this first chapter, as many of these issues are ones we'll focus on specifically as examples throughout later chapters.

- **Improper input validation.** Simply put, you're asking for trouble if you assume any input—either from users, data stores, or other systems—conform to your expectations (for example, you ask for their age, expecting a number, and the user types "old enough to know better").
- **Improper encoding or escaping of output.** This is at the root of most injection-based software attacks and is of particular concern because—as OSs become more hardened—applications are fast becoming a favorite target.
- **SQL Injection.** Hackers will use an input statement to execute their own code against your database. It is your job to prevent them from doing so.
- **Cross-site scripting.** In the world of Web development, this is one of the most popular ways to attack code. Quality code can stop it cold.
- **OS command injection.** Unsurprisingly, injecting commands into the OS is a popular attack. Even Google's first release of its Android mobile phone OS had this problem.
- **Clear-text transmission of sensitive information.** When you ask a user for credentials and the user trusts you, you need to protect that information. Passing it through the network (or worse yet, the Internet) in plain sight where anyone with a sniffer can read it is just irresponsible. Quality code builds security into every aspect of an application and is always careful with how it handles data.

- **Cross-site request forgery.** This combination of social engineering and scripting attack can leave unsuspecting Web applications vulnerable.
- **Error message information leak.** Clear error messages are obviously desirable but *too much* information isn't good—especially if sensitive information is displayed (for example, error messages that provide the name or credentials of the service account).
- **Memory buffer overflows.** One of the most common errors with native code programs is when a memory variable is not controlled. A hacker enters an input that overwrites the executable portion of the program. Thus, the hacker's code executes. Simply put, sloppy programming; and even today's advanced programming frameworks don't automatically provide 100% protection.
- **External control of state data.** Storing state data in a database or elsewhere is fine but don't assume that store is tamperproof unless it truly is, and make sure your application is validating that data when it's read back in.
- **External control of file names and paths.** As with many injection attacks, using user input to construct file names and paths leaves opportunities to attack and crash an application.
- **External control of search path.** If your application depends on underlying OS search paths, an attacker can gain control of that and misdirect application resource requests. Never assume an application is running in a safe sandbox.
- **Code injection.** Dynamic code offers cool capabilities but also provides an opportunity for serious vulnerabilities.
- **Downloaded code.** There is a lot of interesting code posted on the Internet. Don't depend on *anything* generated outside your application's own code. Unless proper coding security measures are employed, downloaded code can be easily hijacked.
- **Improper resource shutdown or release.** Don't leave it to the OS to release resources automatically—explicitly clean up after yourself.
- **Improper initialization.** Don't assume anything about your application's starting state; properly initialize everything you plan to use.
- **Incorrect calculation.** Using user input in calculations is an opportunity for unexpected buffer overflows and other problems.
- **Improper access control.** You need to continually check to make sure users are allowed to do what they're trying to do; don't put all your security into one 'front door' and give attackers an opportunity to bypass that through unexpected code paths.
- **Using bad cryptography.** 40-bit encryption is *so* 1980s. Using outdated or broken algorithms is as bad (or worse) than using none at all. Don't develop your own encryption schemes, either; use strong, industry-standard libraries.

- **Hard-coded passwords.** Bad idea. *All* software can be decompiled and your password can be revealed. Yet this has been a common poor-coding practice for more than four decades. And maybe it is not a good idea to keep password in an unencrypted XML configuration file on you server?
- **Insecure permission assignment for critical resources.** Sensitive data should be secured at many layers: within your application, in middle-tier components, at the database, and so forth.
- **Use of non-random random values.** Security features often rely on randomness, but computers aren't always good at generating random numbers. Are you making your software's security isn't too predictable?
- **Execution with unnecessary privileges.** In all probability, your application *doesn't* need to be run by a systems administrator, so make sure it will run without that unnecessary privilege. If you think you *must* have that kind of permission, you might be doing something wrong.
- **Client-side enforcement of server-side security.** If a server enforces security measures, don't try to duplicate that in your client application; instead, respond to security errors as appropriate. Your client application can be decompiled and used to thwart the security on the server itself.

Although security-heavy, the problems highlighted in this list also manifest as stability problems, create long-term maintenance problems, and display other aspects of poor quality. We'll return to some of these examples again as we look at ways to improve the quality of the code we release.

What to Expect in this Definitive Guide

So what's coming next? In Chapter 2, we'll dive into coding standards and best practices. You'd be shocked, and possibly horrified, at the number of developers out there who are still following coding standards originally developed for Visual Basic v1. Thus, we'll be going back to basics and really getting into modern coding practices. We'll not only cover what you should do but also what you shouldn't—and most importantly, *why.* We'll look at how some of Visual Studio's timesaving tools can, if you're not careful, actually create a negative impact on code quality, so we'll also look at ways to use those features safely. We'll wrap up by looking at different development methodologies, comparing and contrasting them so that you can select one, or a hybrid of them, that works best for you.

No developer works alone, so Chapter 3 will focus on code analysis and peer reviews, a critical tool in improving application quality. We'll examine the differences between manual and automated code reviews, and look at a set of code review rules. Code analysis is really a pretty complicated topic, so we'll spend lots of time looking at considerations such as maintainability indices, depth of inheritance concerns, and so forth.

In Chapter 4, we'll begin looking at ways to address coding errors. Hey, they're inevitable, so we might as well learn to deal with them, right? We'll create a sort of taxonomy for different types of errors, and look at specific ways to prevent, mitigate, and address each type. Obviously, getting rid of errors is a big step in improving code quality, so we'll also look at tools you can use to eliminate bugs more easily. I'll also present my Unified Theory of Bugs, which will help any developer who struggles with debugging really understand how to proceed. Even if you're an experienced developer, I think you'll find some things here that are useful—and most especially things that you can share with less-experienced members of your team.

Chapter 5 is where we'll address performance and security problems. They're not bugs per se, but they're definitely things that lead to a lowered perception of quality in a finished application. We'll look at different ways of detecting coding problems, explore where security problems come from, and really look at the things you, as a developer, can do to insist on better performance and security from you applications. You might be surprised at where I eventually lay the blame for bad performance and security, but I think you'll be happy with the result.

We'll dive into testing in Chapter 6, and start looking at both manual testing and automated testing. We'll explore the importance of solid use cases and testing assets, and look at how bad cases and assets can lead you to *think* you're producing a quality app—and lead to unpleasant surprises when you find out differently later on down the line. We'll look at different types of testing, such as unit and system testing, and make sure we agree on the value of each type.

Finally, Chapter 7 is where we'll look at automated debugging, code analysis, and testing. There's really no point in using automated tools until you've mastered doing things properly *without* automation; I'm a big believer that although automation can make things less boring and more consistent, it can't actually add much in the way of quality on its own. Your quality will only be as good as *what* you automate, which is why we're getting to automation only after we've explored all those other ways to improve code quality.

So that's our battle plan. I hope you'll stick with me through each chapter, and I hope you'll find plenty of information to help you produce higher-quality code.