Realtime publishers

The Definitive Guide™ To

Building Code Quality

sponsored by

Don Jones

Chapter 2: Coding Standards and Best Practices 20
The Purpose of Coding Standards 20
Modern Coding Standards 23
Approaches to Naming, Comments, and Formats 23
Naming Conventions 23
Commenting Conventions 25
Formatting Conventions 26
Functional Conventions 27
Established Standards for Visual Basic, C#, C++, and ASP.NET 28
Defining Coding Standards in Visual Studio 29
Resistance to Coding Standards 30
Best Coding Practices, Processes, and Procedures 32
IntelliSense and Code Refactoring Tools 32
Task List and Code Snippets 33
Background Compilation and Continuous Code Feedback 34
Code Complexity Assessments 35
Integrated Debugger 36
Visual Design Tools 36
Software Development Methodologies 37
Waterfall 37
Agile 38
Incremental/Iterative (Extreme Programming) 39
Others 40
Continuous Integrated Testing and Nightly Builds 40
Defining Your Coding Standards 41

Realtime publishers
MICRO FOCUS Leading the Evolution™

Copyright Statement

© 2009 Realtime Publishers, Inc. All rights reserved. This site contains materials that have been created, developed, or commissioned by, and published with the permission of, Realtime Publishers, Inc. (the "Materials") and this site and any such Materials are protected by international copyright and trademark laws.

THE MATERIALS ARE PROVIDED "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, TITLE AND NON-INFRINGEMENT. The Materials are subject to change without notice and do not represent a commitment on the part of Realtime Publishers, Inc or its web site sponsors. In no event shall Realtime Publishers, Inc. or its web site sponsors be held liable for technical or editorial errors or omissions contained in the Materials, including without limitation, for any direct, indirect, incidental, special, exemplary or consequential damages whatsoever resulting from the use of any information contained in the Materials.

The Materials (including but not limited to the text, images, audio, and/or video) may not be copied, reproduced, republished, uploaded, posted, transmitted, or distributed in any way, in whole or in part, except that one copy may be downloaded for your personal, non-commercial use on a single computer. In connection with such use, you may not modify or obscure any copyright or other proprietary notice.

The Materials may contain trademarks, services marks and logos that are the property of third parties. You are not permitted to use these trademarks, services marks or logos without prior written consent of such third parties.

Realtime Publishers and the Realtime Publishers logo are registered in the US Patent & Trademark Office. All other product or service names are the property of their respective owners.

If you have any questions about these terms, or if you would like information about licensing materials from Realtime Publishers, please contact us via e-mail at info@realtimepublishers.com.

[**Editor's Note:** This eBook was downloaded from Realtime Nexus—The Digital Library for IT Professionals. All leading technology eBooks and guides from Realtime Publishers can be found at http://nexus.realtimepublishers.com.]

Chapter 2: Coding Standards and Best Practices

Coding standards have been around for nearly as long as programming languages. In fact, as soon as programming languages became complex and flexible enough that developers could adopt their own individual styles, and as soon as one developer realized he didn't like another developer's personal style, coding standards were invented.

Software development is, in many ways, more analogous to art than science. Like writing prose, the author—or programmer—constructs algorithms and concepts in her head, then translates those to the screen using the grammar of whatever language she is working in. Because programming languages must offer flexibility in order to be useful, there is inherently more than one way to accomplish any given task. Coding standards seek to reduce that flexibility just a bit so that developers can more easily understand the code written by others (or, frankly, that they wrote themselves).

In a way, coding standards represent the difference between "good grammar" and "bad grammar." For example, the English language is flexible enough to allow a person to write "I ain't got no candy" or to write "I do not have any candy." Most fluent speakers will understand either phrase but only the second one is considered proper grammar—in other words, only the second one follows the accepted standards for the language. The standards don't restrict the language's flexibility to express different sentiments, but they do try to provide consistency and form to keep everyone more or less on the same page.

The Purpose of Coding Standards

A Web search for the phrase "coding standards" turns up thousands of results. Many are different coding standards that have their own name, their own fans, and their own sets of rules. Some of the oldest documented coding standards are for the COBOL language, which was originally used on mainframe computers before the advent of personal computers. The COBOL Style Forum (http://home.swbell.net/mck9/cobol/style/style.html) collects commonly accepted styles for that language, and neatly categorizes them into groups:

- Cosmetics—How to format comments, indentation, the use of white space, etc.
- Names—Naming conventions for variables and other language elements
- Subprograms—Ways in which modules are implemented, how parameters are defined, and so forth
- Error handling—Types of errors, how errors are reported, aborting errors, and more

This list is useful because it helps to define the purpose of coding standards (or "coding style," if you prefer): To help achieve consistency in not only cosmetic considerations such as comment formatting but also functional issues such as how modules are entered, executed, and exited.

One seminal work on coding style is *The Elements of Programming Style,* written by Brien Kernighan and P. J. Plauger in the 1970s and illustrated with examples from Fortran and PL/I—two commonly used languages of the time. Accepting the concept of coding style as analogous to the grammar of a human language (such as English or Spanish), the authors patterned the book after Strunk & White's *The Elements of Style,* a standard work used by writers and journalists. Although revised in a second edition in 1978, the book has fallen largely into disuse simply because its examples are no longer relevant to modern programming languages. However, the book's suggested standards survive in numerous derivative works on programming style.

Style and Complexity

It's easy to understand how cosmetic issues such as comment formatting could be considered "style." It's even fairly straightforward to understand how "style" can dictate how code is modularized, at least at a high level. It's less straightforward to see how "style" can dictate things like how clever a programmer should be. Yet Kernighan filled his book with cautions on writing overly complex or what he termed overly "clever" code. One quote sums up his feelings: "Debugging is twice as hard as writing the code in the first place. Therefore, if you write the code as cleverly as possible, you are, by definition, not smart enough to debug it." Although this sentiment is difficult to capture in a set of "style rules," it nicely sums up the whole point of coding standards—to help simplify code so that maintenance and debugging is easier and can be more easily accomplished by someone other than the original programmer (who naturally has a greater affinity for the code).

Consider this pseudo-code, which Kernighan might consider overly clever:

```
Var = FunctionA(FunctionB(FunctionC(Input1),Input2),Input3);
```

A problem with this coding style—nesting functions as inputs to other functions—is that it requires a human being to perform fairly complex mental gymnastics in order to sort out what that single line of code is doing. Some coding standards prefer that an individual line of code perform only one granular task:

```
Var1 = FunctionC(Input1);
Var2 = FunctionB(Var1, Input2);
Var3 = FunctionA(Var2,Input3);
```

The idea is to make the code less compact and easier for a human to follow. The compiler for most languages would treat these two examples nearly identically, but one is distinctly easier for a human to read, debug, and maintain in the future. This is perhaps the ultimate expression of the purpose of coding standards: To write code in a consistent fashion that supports future maintenance and helps reduce defects.

It's important to understand that coding standards are, in fact, anything but "standard." Different standards conflict widely, often over the simplest of items. One standard for C#, for example, requires that constructs' opening curly brace appear on the line that follows the construct statement:

```
Function myfunction() {
}
```

While another standard for the same language suggests that curly braces always appear at the start of a new line:

```
Function myfunction()
{
}
```

Fortunately, these conflicts don't matter. Different coding standards often have names, especially once they're widely accepted: GNU, Hungarian, NOAA, Macadamian, Horde4, and many, many more. When it comes to picking and following a coding style, there are really only three important factors:

- Everyone on the development team needs to be following the same standard. It will make the code more uniform and thus easier to maintain.
- If you can choose a standard that basically reflects the style already in use by the majority of your team members, it'll be easier for them to follow. Most coders have a style with which they are familiar. Once the style becomes internalized, it will help them code more consistently.
- If you are using automated tools to help enforce the coding style, you obviously need to pick a standard that is supported by those tools. For example, putting comments in the beginning of Visual C# functions using the tool-mandated format facilitates automatic documentation of the code itself.

Today, any programming language with a sufficiently large audience will be accompanied by one or more coding standards. Because many programming languages derive from similar sources (most of today's popular programming languages, for example, have a general structure derived from C), some elements of various coding styles tend to overlap, such as the standards on where to put braces. This chapter will focus primarily on coding standards that are either completely generic or relate specifically to Visual Basic, C++, and C#, the two primary languages included with Microsoft Visual Studio and two of the most popular languages used to develop for the Microsoft .NET Framework. We'll also look at standards specific to Microsoft's ASP.NET Web development framework.

Modern Coding Standards

If coding standards began by addressing issues such as formatting and naming, they've definitely evolved to include much more. In fact, the switch from the word "style" to "standard" implies the additional depth that modern coding standards address: How modules are linked, how error handling is performed, how classes are constructed, and even in some cases, how user interface elements are utilized.

Recognizing that different organizations will adopt different standards for varying reasons, this chapter will focus first on *approaches* to standards—general principles, if you will—before looking at some of the more popular standards currently used for Visual Basic, C#, C++, and ASP.NET.

Approaches to Naming, Comments, and Formats

These approaches fall generally under the "cosmetic" category and are intended to help produce code that is more readable.

Naming Conventions

Naming guidelines typically seek to establish consistency in the naming of code elements: functions and subroutines, classes, variables, constants, and so forth. Naming standards focus on the use of capitalization, embedded data within names, and so forth. For example, Microsoft's Visual Basic Scripting Edition (VBScript) language was commonly used with Hungarian naming conventions for variables in which each variable name included a three-character prefix that indicated the variable's intended data type: strName for a string variable, for example, or intCounter for an integer.

Hungarian Notation

Microsoft so popularized Hungarian Notation, before abruptly turning away from it, that you'll still see many developers using it. A brief explanation of why it has fallen out of favor is, I think, in order.

Invented by Charles Simonyi in the 1970s, the basic gist for Hungarian notation was to name things so that their purpose or logical type was evident in the name. Prefixes such as "int" and "str" were straightforward enough, but as languages advanced to include more complex data types, prefixes such as "rgsz" began to diminish the intended purpose of the notation, which was to make things easily understandable.

Hungarian Notation has a strong association with Visual Basic primarily because Visual Basic introduced so many new people, often without formal development training, to the world of development. Hungarian was a great, easy-to-understand way for them to produce variable names better than "x" and "y." Hungarian had some specific and unique advantages for Visual Basic. For example, prefixing a variable with "obj" not only let you know it contained an object but also reminded you that you needed to use the Set statement to assign something to it.

Realtime publishers
MICRO FOCUS Leading the Evolution™

But when .NET came along, the old Visual Basic went away. Today's Visual Basic has more in common with C++, in fact, than its own predecessor languages. *Everything* in .NET is an object, and prefixing a variable name with "obj" is redundant. Tools such as Visual Studio keep track of variable data types so that the tool will help you remember that something is a string object whether its name has an "str" prefix or not. .NET supports extremely long variable names, and IntelliSense auto-completion will have Visual Studio help you type those names with less actual typing on your part. In short, all the things that Hungarian Notation helped fix are, by and large, no longer a problem.

One place you might still see Hungarian Notation is in the naming of user interface elements, with prefixes such as "frm" for a form and "btn" for a button. Some developers do this so that their form controls are grouped in Visual Studio's IntelliSense code-hinting menus, which list elements alphabetically.

Naming conventions also strive for readability. Most coding standards discourage the use of abbreviations or acronyms, preferring longer names such as "DirectoryUserName" over less-obvious alternatives such as "ADUN." The idea is to keep the code readable for months and years into the future—and 2 years after writing "ADUN," you might be left wondering what it was supposed to mean in the first place. Exceptions are made that allow for widely recognized acronyms in extreme circumstances: "DN" standing in for "DistinguishedName" is acceptable because the shorter version is used industry-wide, and it does help create somewhat more compact code.

Where Abbreviations Were Born

Abbreviations in code can be traced primarily back to early developers' desire to have their code consume less memory. One of the most famous is Bill Gates' use of "OK," rather than the more traditional "READY," as a prompt in early versions of Microsoft BASIC because his alternative used three fewer bytes.

Modern programming languages don't have to deal with any of the issues that led to this space-saving approach. To begin with, neither disk space nor computer memory is as precious as it once was: Both are relatively inexpensive resources and most computers come equipped with more than they'll ever need. More importantly, though, and of particular relevance to .NET Framework programming, is that your source code is going to be tokenized into a more-compact intermediate language anyway. Using shorter variable names does not necessarily lead to a smaller final assembly than using longer variable names. Longer variable names can offer more readability and clarity, so they are generally preferred—to a point, of course. No developer wants to have to repeatedly type lengthy variable names (although, as we'll see, modern tools help make that easier, too, making an even stronger argument for longer variable names that are clear and purposeful).

Naming conventions also tend to favor clarity. A function name such as GetLength conveys the purpose of the function even to a non-programmer; an alternative such as GetInt might be technically correct if the length is indeed an integer, but it's less specific and clear about what the function is actually getting. In general, language-specific terms such as Int, String, Char, and so forth shouldn't be used. Instead, use variable names with more semantic meaning, such as Length, UserName, and so forth.

> Numerous other naming conventions exist; Microsoft offers recommendations at http://msdn.microsoft.com/en-us/library/ms229002(VS.80).aspx to cover capitalization, general naming, namespaces, classes, types, parameters, and much more.

Commenting Conventions

Any developer with a modicum of experience has had to, at one time or another, read through a few hundred lines of code written by another developer—and without a comment in sight. In contrast, I have personally had to read through code where every one or two lines of code was accompanied by a dozen lines of witty commentary. Clearly, comments are like food—you need a certain amount to thrive, but too much is definitely a bad thing.

Some commenting conventions are designed to improve readability. It's generally agreed, for example, that comments following a line of code are a Bad Thing, because they're simply more difficult to read. Other commenting conventions can seem nitpicky, such as requiring that the comment character be followed by a space and then a comment that starts with a capital letter and ends with a period—conventions that contribute dubious value to actual readability and technical documentation.

Most organizations adopt commenting conventions designed to convey a sense of how comments should be used. Guidelines might suggest:

- Use a comment when a variable is first declared, to specify what that variable will be used for.
- Use a comment before logic constructs to describe the logical decision being made, and then again within each logical branch to indicate the decision that led to that branch.
- Use comments before or at the beginning of functions and other modules to describe their purpose, their expected input and output, along with any notes on expected input ranges or other information.
- Comments on changes made when modifying code during maintenance. This can help identify changes and provide a history of when, where, who, and why code changes were made.

Some coding standards discourage the use of multi-line block comments, while others encourage them. Note that some programming languages support a specialized comment syntax that can actually help automate code documentation. C#, for example, uses a "triple hack" syntax where three slashes indicate an XML-formatted comment:

```
/// <exception cref="BogusException">
/// This exception gets thrown as soon as a
/// Bogus flag gets set.
/// </exception>
```

These XML-formatted comments can be scanned by automated tools that produce attractive, hyperlinked documentation for the code.

> **Note**
>
> Microsoft's documentation for these XML comments can be found at http://msdn.microsoft.com/en-us/library/b2s063f7.aspx; a good article about their use is available at http://msdn.microsoft.com/en-us/magazine/cc302121.aspx.

Formatting Conventions

Formatting is designed to make code easier to read *and,* believe it or not, to help prevent bugs. Probably the most universally accepted formatting convention is that code within a construct be indented:

```
ForEach ($server in $servers) {
  $server.Restart();
}
```

This technique has numerous advantages:

- It's easier to make sure that each construct has been properly closed
- It's easier to see the conditional, looping, or other code contained within the construct
- The white space directly under the construct's statement ("ForEach" in the above example) helps draw the eye to the construct itself, making it easier to follow the code's logic

This is such a universally accepted practice that most development environments, including Visual Studio, go to great lengths to produce this formatting automatically. Automated code-reformatting tools (sometimes called "code beautifiers") automatically indent code in this fashion, as well. As with any standard, of course, interpretation leads to inconsistency: Most organizations (and tools) get very specific about this formatting, requiring that each indentation be a tab character with tab stops every four spaces, for example.

Other formatting conventions are also designed to improve readability, such as requiring whitespace around operators:

```
A = (b + c) * d
```

Rather than:

```
A=(b+c)*d
```

> **Note**
> A list of commonly-accepted formatting conventions can be found at http://wiki.scummvm.org/index.php/Code_Formatting_Conventions. Although these conventions were produced for a particular software project (and are, in fact, a good example of how your organization might choose to document your standards), these conventions are generally accepted and used by a wide range of developers.

Functional Conventions

Coding standards become more complex and language-specific when you move away from mere formatting and naming and start moving into more functional conventions. These conventions might include considerations such as

- All functions must have only a single exit point—functions must not exit midway through the code unless they are raising an error.
- All elements must be declared with the tightest scoping possible that still enables their intended purpose. For example, variables must be declared as private unless they are explicitly needed in different scopes.
- All functions and other modules must accept a Boolean parameter which defaults to True, but when set to False, will cause the function or module to skip any permanent action and to instead log a description of the action it would have taken.

> **Note**
> That last example was taken from a consulting project I worked on. The intent was that by setting all the parameters to False, you could test the entire application without actually committing information to a database or other storage. Instead, you'd get a log file describing what would have been done. Eventually, that organization created a kind of framework that encompassed that functionality, making it easier for their developers to implement. The point is that strictly internal coding standards are absolutely fine if they're needed to meet business or management goals regarding development projects.

As with all coding standards, the goal is twofold: One, to make the code easier to read and more consistently implemented; second, to help reduce code defects by more clearly defining how code will be implemented. For example, a code module with fixed entry and exit points will be easier to debug simply because the ways in which it can be called are limited. With fewer scenarios to test for, unit testing can catch more bugs, leaving fewer bugs for integration testing.

> **Note**
>
> I stated earlier that coding standards seek to partially reduce a developer's flexibility within their programming language, and that is certainly true. It might be more accurate, however, to say that standards really seek to reduce and control the complexity of code. Less-complex code is easier to debug earlier in the development life cycle, and is easier to debug throughout the development cycle.

Established Standards for Visual Basic, C#, C++, and ASP.NET

Microsoft has published a number of standards for code, which the company uses in most of the samples and documentation they produce. Although these are by no means comprehensive, these standards serve as an excellent start. An advantage to using an external set of conventions—at least as a starting point—is that you'll be developing code that is more easily interchanged. If, for example, one of your developers needs to use a snippet from a Microsoft example, she won't have to re-work the code to meet internal standards if your internal standards are based upon Microsoft's.

> **Note**
>
> As the world's largest software company, you can also argue that Microsoft has a vested interest in producing coding standards that help produce higher-quality code. There are certainly worse things you could do than to copy the basic standards of a company that employs tens of thousands of developers and produces millions of lines of code annually.

Resources for major coding conventions include:

- Visual Basic coding conventions are available at http://msdn.microsoft.com/en-us/library/h63fsef3.aspx; most of these are actually applicable to C# as well. This is one of Microsoft's most comprehensive yet concise documents on coding conventions.
- A C# Coding Style Guide by Salman Ahmed is at http://www.csharpfriends.com/articles/getarticle.aspx?articleid=336. This is a concise guide with commonly accepted conventions, focusing on both cosmetic and functional issues.
- An ASP.NET-specific guide is at http://www.visualize.uk.com/resources/asp-net-standards.asp, and applies to both C# and Visual Basic. Based on Microsoft guidelines, this guide provides a few ASP.NET-specific items, such as naming for user interface elements (using Hungarian Notation), and a focus on XML-based comments.

Visual Studio Auto-Generated Code

Interestingly, the code that is auto-generated by Visual Studio doesn't always follow a particular set of coding standards. You'll notice this especially in the code generated by the Visual Studio visual forms designer. That's okay, though; that auto-generated code isn't intended to be especially human-friendly, and you're actively discouraged from editing that code manually. Changes to the code are accomplished by working in the visual forms designer, allowing it to re-generate the code. Because the code isn't intended for "human consumption," any real or perceived lack of standards compliance isn't important.

Given the many different coding standards available, it would be impossible for Visual Studio to generate "standardized" code that would comply with every possible coding standard an organization might use. Given that impossibility, Microsoft decided to just have Visual Studio produce "neat" code rather than trying to force any particular coding style or standard.

Understand that I have no particular attachment to any one set of coding standards, and most organizations will need to develop their own. In fact, developing your own—using some industry standards as a starting point—can be a great way to help overcome developer resistance to standards, which I'll discuss shortly.

Defining Coding Standards in Visual Studio

Visual Studio 2008 helps developers observe coding standards in a few nice ways (some of these features also exist in earlier versions of the product). First, the product's Options dialog box enables developers to configure various auto-formatting features (see Figure 2.1). The "pretty listing" feature in particular helps maintain proper indentation within constructs and offers other cosmetic features.

Realtime publishers
MICRO FOCUS Leading the Evolution™

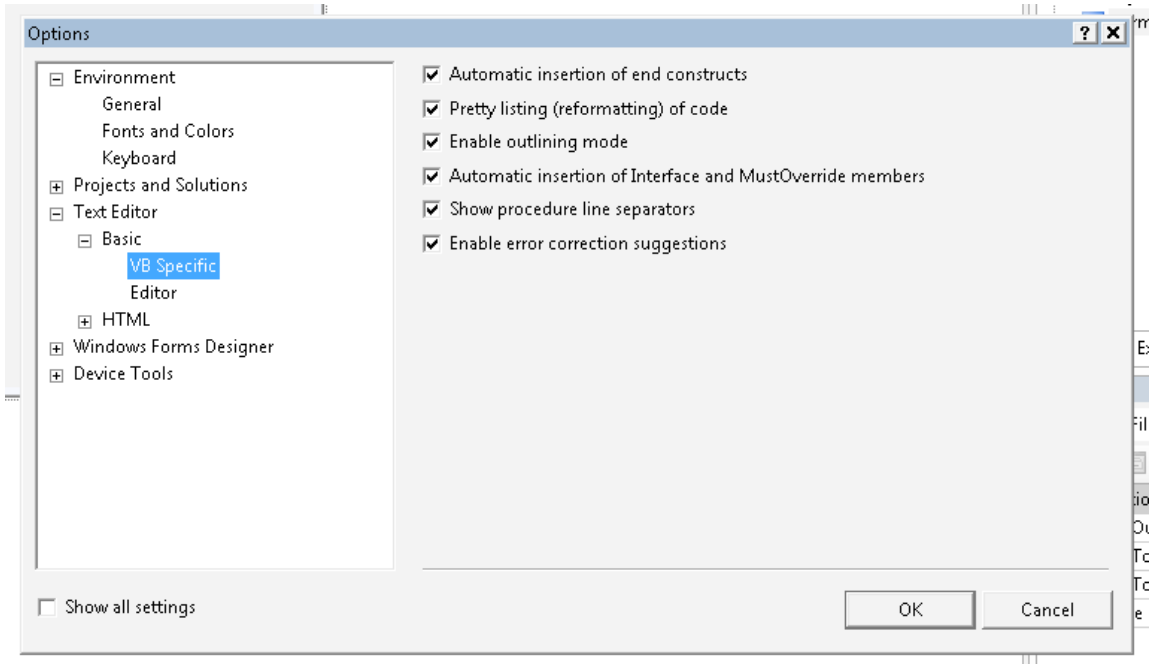


Figure 2.1: Configuring Visual Studio to help "prettify" your code.

Another major feature is that Visual Studio is extensible, meaning that both free and commercial add-ins from third parties can be included to provide additional "built in" capabilities for enforcing coding styles. These include code formatters, refactoring tools, and much more. One such freeware add-in is from Microsoft, and is called StyleCop (it's only for C# projects). You can find it at http://code.msdn.microsoft.com/sourceanalysis.

Resistance to Coding Standards

Sadly, not every developer welcomes coding standards. As I've stated, software development is often more art than science, and no artist likes to be told how to create. Even coders who consistently use standards may be enamored with their own personal standard. This can create friction when the developer is expected to employ the institutional standards. Developers who don't have a background in following coding standards have almost always adopted at least a loose personal style, and switching over to a formal set of standards can consume time and seem frustrating.

Organizations have resistance to standards, too, not the least of which is the additional time and effort needed to observe and maintain the standards over time. Fortunately, modern programming tools make this a bit easier, and in the end, the time and effort that goes into observing standards must be seen as an investment. The return on that investment is a decrease in defects coming out of unit testing, an increase in code maintainability (meaning lowered future costs), and ultimately higher code quality.

There are techniques, especially useful in organizations that don't have strong coding standards, to make standards easier to accept and observe:

- Collaborate with developers when creating standards. Developers will be much more likely to accept standards if they feel they have played an active role in their creation. Align standards with the corporate culture as much as possible—for example, smaller entrepreneurial organizations may adopt standards that allow developers more personal freedom, while larger organizations with numerous documented practices and processes will feel more comfortable adopting more rigid standards.
- When you can't agree, learn to pick your battles. Determine which of the standards are really important and which aren't. It is not worth losing sleep over missing standards that are not important. Allow your developers' creative sides to emerge—don't be too restrictive. Compromising on a few relatively unimportant standards may pave the way to enforce more important ones.
- If there is still disagreement and no compromise is possible, determine whether the dissenting opinions are really caused by displaced resentment for being told *how* to code or some unexplained fear of code reviews. If your organization is serious about having and enforcing coding standards, these developers will need to get over it.
- When all else fails, a management mandate may be necessary to resolve a particular issue. This is best used as a last resort but can help in extreme situations. There have of course been cases where developers have quit or have been fired due to their unwillingness to follow (or even agree to) standards; ultimately, the organization or the developer—correctly—decided that someone who couldn't work on a team didn't belong on the team.
- Maintain a regular review of the standards and modify the standards when it is obvious that there is a better way. In my experience, a *small* standards committee is usually a good idea, as it allows developers to continue to have an active role in the standards process. New team members should be given a voice in this committee, because it helps bring a fresh viewpoint and helps rotate the committee membership amongst the team members.

And did you see that I snuck in the phrase *code review?* It's coming in a later chapter, and it's crucial to not only enforcing standards but also creating much higher-quality code.

Best Coding Practices, Processes, and Procedures

Most developers are well aware of Visual Studio features such as IntelliSense, snippets, the Windows Clipboard, and so forth; but they're typically aware of them as *convenience* features rather than *quality* features. What's the difference? Convenience is a pretty relative term. When I started using IntelliSense for the first time, I didn't like it—I'm already a fast typist, and those little menus popping up distracted me. So I tended to not use the feature. That's the problem with convenience: If an individual developer doesn't find it to be personally convenient, she won't use it. Quality, however, is different. Many of Visual Studio's "convenience" features can make a strong, positive contribution to code quality—and that's a reason to use them even if someone doesn't find them to be particularly convenient.

IntelliSense and Code Refactoring Tools

IntelliSense is an umbrella brand name for Visual Studio's code hinting and code completion features. There are two distinct IntelliSense features that most developers are familiar with: code completion (see Figure 2.2) looks at what you've typed, what you might be typing, and offers a list of suggestions. This list might include variable names, constants, object member names, and so forth.

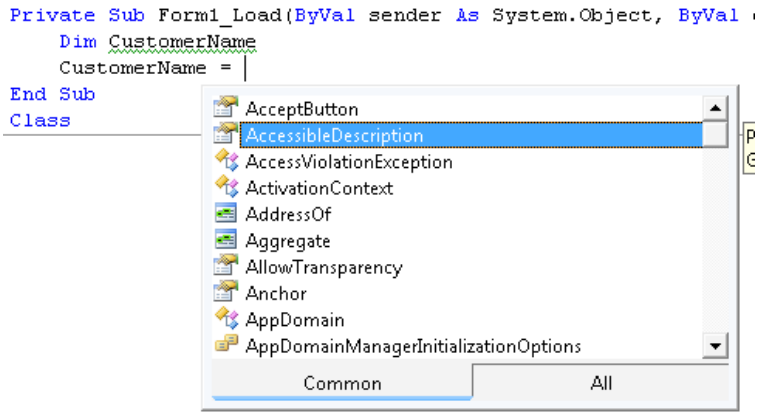


Figure 2.2: IntelliSense code-completion feature.

Why use this? Because if you let Visual Studio do the typing, you won't make typos—pure and simple. This IntelliSense feature also encourages the use of full class names because it makes them easier to type; using full class names (rather than abbreviations) can help make the code easier to read. Another IntelliSense feature, code hinting, takes the form of tooltips that remind developers of the correct syntax for statements and methods. Figure 2.3 shows it in action.

Realtime publishers
MICRO FOCUS Leading the Evolution™

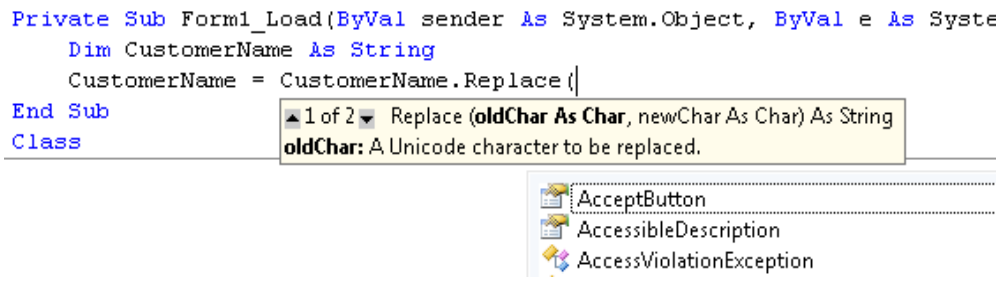


Figure 2.3: IntelliSense code-hinting feature.

In addition to being a timesaver, this little feature can help prevent bugs by making it easier for developers to "look up" the correct syntax without having to actually turn to the documentation. This is especially important with .NET Framework class methods that have multiple overloads. Without a quick reference for unfamiliar methods, a developer might mistakenly use the wrong overload, resulting in run-time logic errors.

Finally, *refactoring* is a sometimes-complex process that helps implement code standards in otherwise non-compliant code. Refactoring can search for variables that are declared but not used and remove them (Visual Studio visually highlights these variables, as well), rename variables to conform to code standards, correct formatting problems (such as improper indentation), and rename variables, functions, and other items, and so forth.

Task List and Code Snippets

Visual Studio provides a number of features that help reduce typing—again, a convenience feature on the surface, but one that ultimately provides a quality benefit in that it enables code re-use with less potential for typos. Code re-use also makes it easier to observe coding standards, provided the code you're reusing was standards-compliant in the first place. The primary feature for code reuse is the Snippets feature, which consists of short code snippets that your team can put together into a library. Pressing Ctrl+K and then Ctrl+X in Visual Studio's text editor summons a list of available snippets (numerous ones are included with the product), allowing a developer to simply start typing a snippet name. This can be useful for complex code blocks all the way down to simple skeletons for constructs. Figure 2.4 shows an example.

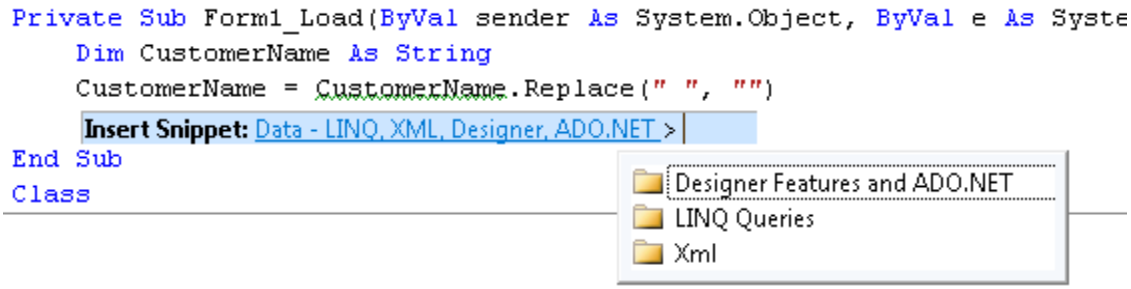


Figure 2.4: Using the Snippets feature in Visual Studio.

Realtime publishers
MICRO FOCUS Leading the Evolution™

Snippets can be especially useful for helping developers incorporate new or unfamiliar techniques or technologies. By having one senior developer produce a set of snippets, other developers can save time—and remain consistent and correct—by using those snippets as starting points.

Visual Studio's auto-generated task list also helps developers stay on track. As Figure 2.5 shows, developers can insert comments that begin with special predefined tokens (and you can define new tokens); the task list scans for these and assembles a task list automatically. This feature allows unfinished code, hacks, debug code, and other non-production code to be easily identified and documented for the entire team, and makes it easier to jump back to those items and clean them up later.

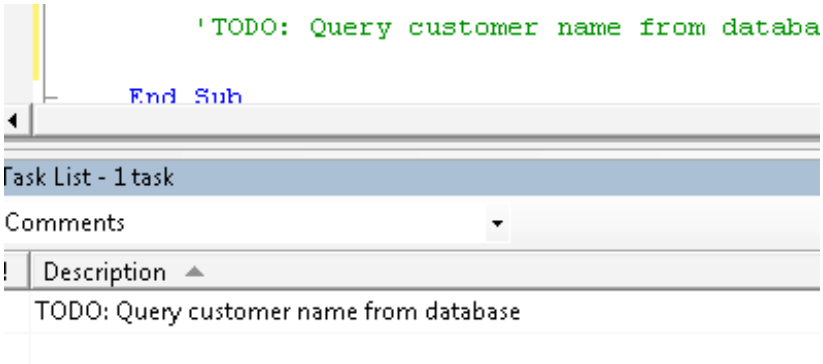


Figure 2.5: Visual Studio's task list feature.

Note

These may seem like kind of minor features, but the fact is that they help developers maintain code quality by allowing them to continue working the way they want to. Developers *need* the ability to add debug code, to come back to code later and finish it, and so forth; tools like the Task List enable them to do these things while helping prevent any negative impact to the code quality by *documenting* these "hacks" and making them easier to clean up later.

Background Compilation and Continuous Code Feedback

One of Visual Studio's best features for helping to maintain code quality is its ability to continuously compile code in the background and provide visual feedback to the developer. This feature doesn't attempt to *execute* the code, but it does warn of numerous conditions that could cause run-time or compile-time errors. These can be anything from simple coding problems such as not assigning an initial value to a variable (see Figure 2.6)…

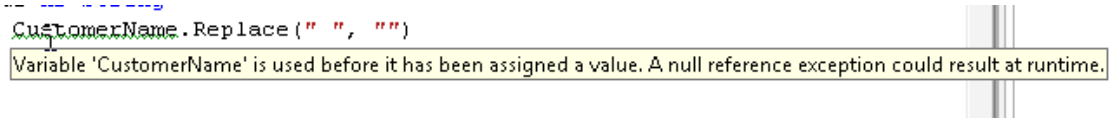


Figure 2.6: IntelliSense code feedback for non-critical issues.

Realtime publishers
MICRO FOCUS Leading the Evolution™

…to more serious problems (see Figure 2.7). This visual feedback—similar to the automatic spell checking in Microsoft Word—helps developers spot and correct these problems before they even begin unit testing, helping save time and improve code quality.

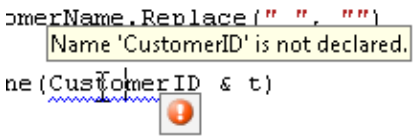
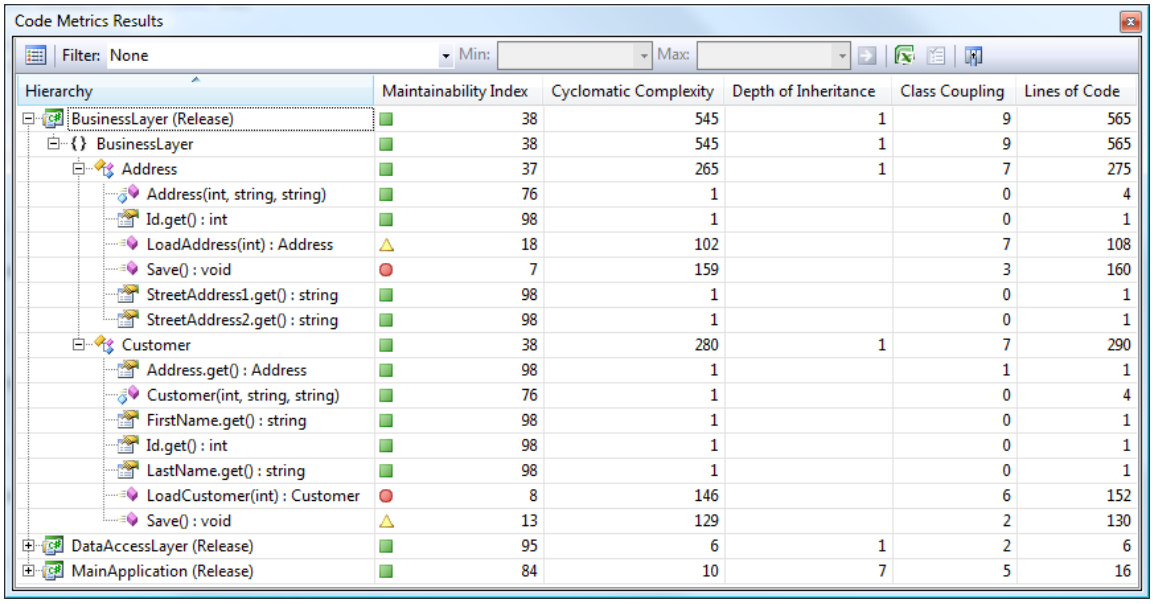


Figure 2.7: IntelliSense code feedback for more critical issues.

Code Complexity Assessments

Some premium editions of Visual Studio 2008 include code metrics, a static analysis tool that helps analyze the complexity of code using some of the algorithms I discussed in the previous chapter. Figure 2.8 shows an example of the Code Metrics analysis. As you can see, the tool identifies major code units and their interfaces, calculates their complexity, and displays a "maintainability index." Anything with a yellow or red icon in that column is something that deserves special attention—high maintainability means a higher likelihood of bugs, too.

You can read more about this tool at http://blogs.msdn.com/fxcop/archive/2007/10/03/new-for-visual-studio-2008-code-metrics.aspx. A more mature and feature-rich product, DevPartner, provides a greatly expanded set of features that are described at http://www.microfocus.com/products/DevPartner/index.asp.



Hierarchy	Maintainability Index	Cyclomatic Complexity	Depth of Inheritance	Class Coupling	Lines of Code
BusinessLayer (Release)	38	545	1	9	565
BusinessLayer	38	545	1	9	565
Address	37	265	1	7	275
Address(int, string, string)	76	1		0	4
Id.get() : int	98	1		0	1
LoadAddress(int) : Address	18	102		7	108
Save() : void	7	159		3	160
StreetAddress1.get() : string	98	1		0	1
StreetAddress2.get() : string	98	1		0	1
Customer	38	280	1	7	290
Address.get() : Address	98	1		1	1
Customer(int, string, string)	76	1		0	4
FirstName.get() : string	98	1		0	1
Id.get() : int	98	1		0	1
LastName.get() : string	98	1		0	1
LoadCustomer(int) : Customer	8	146		6	152
Save() : void	13	129		2	130
DataAccessLayer (Release)	95	6	1	2	6
MainApplication (Release)	84	10	7	5	16

Figure 2.8: Visual Studio 2008's Code Metrics tool.

Realtime publishers
MICRO FOCUS Leading the Evolution™

Microsoft also offers FxCop, a free Visual Studio add-in (http://msdn.microsoft.com/en-us/library/bb429476(VS.80).aspx) that scans native code assemblies and reports possible improvements to the code design, performance, and security. The tool helps enforce Microsoft's own Design Guidelines for Class Library Developers, a set of best practices designed to help improve code quality and maintainability.

Integrated Debugger

If someone told you that an integrated debugger was a "quality tool," you might laugh and say, "of course it is!" *Any* debugger can, of course, help improve code quality by making it easier for developers to catch and correct bugs. But an *integrated* debugger goes a bit further, by helping developers seamlessly move between the debugger and the coding environment and by encouraging more granular unit testing and more debugging activity.

Visual Design Tools

Visual Studio's various visual designers are, again, more than just a convenience item; they help improve code quality. Designing a user interface strictly in code, for example, is a tedious task, prone to minor mistakes, misalignment of user interface elements, and so forth. By using a visual designer, developers can produce a better-looking product, and by having Visual Studio auto-generate the necessary underlying code, developers get better-quality code with less effort.

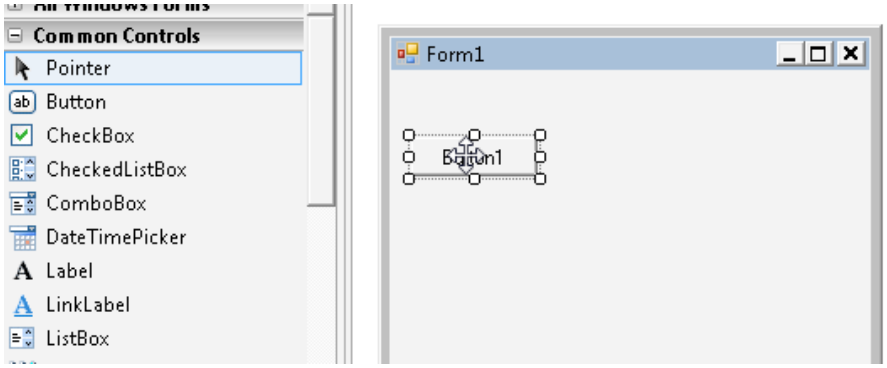


Figure 2.9: Visual Studio's visual forms designer.

Visual Studio includes several visual designers that produce underlying code, including designers for data mapping, Web classes, both Windows Forms and Windows Presentation Framework user interfaces, workflow, and more.

Realtime publishers
MICRO FOCUS Leading the Evolution™

Software Development Methodologies

The purpose of a "development methodology" is to lay out a standardized process for developing software. Why bother? A methodology provides consistency, offers more repeatable results, and best of all, incorporates the lessons learned by thousands of software developers over dozens of years.

Any industry engaged in complex, multi-discipline operations relies on methodologies; few industries document so many different methodologies as the software development industry. In this section, we'll look at a few of the more common methodologies—some of which are actually kind of sub-methodologies that can be combined. The goal here isn't to convince you to use one or another but rather to convince you to use one at all. A good methodology, properly followed, produces higher-quality code simply because it helps you avoid skipping steps and practices that lead to higher-quality code.

Waterfall

The Waterfall model is a sequential development process with distinct phases that follow each other: requirements, design, implementation, testing, and maintenance. The first formal description of the model dates back to a 1970 article by Winston Royce, although he did not use the term "waterfall." Ironically, Royce was presenting this model as an example of a flawed, non-working model.

The distinct phases of a common Waterfall model—requirements, design, and so forth—are not themselves seen as flaws; in fact, most subsequent methodologies use similar phases. What's often seen as Waterfall's biggest flaw is the strictly sequential nature: Once you are finished with the design, you never revisit it. This creates a rigid structure that does not adapt well to the ever-changing realities of business software development. In fact, the inflexibility of the model is what led to the naming of the Agile methodologies, which were intended to offer exactly the opposite experience.

Waterfall is widely used by larger development teams in organizations that typically have a rigid culture, such as the US Department of Defense, NASA, and others. The US Department of Defense moved away from a strict Waterfall model in 1994 with MIL-STD-498, and then moved further still with IEEE 12207, two alternate development methodologies.

Waterfall's supporters feel that the model—often characterized as "Big Design Up Front," or "BDUF"—emphasizes time spent in planning, thus saving time later. Ample evidence supports this claim, although many businesses have emotional difficulty committing a great deal of time creating what seems to amount to nothing more than paperwork. A trick with Waterfall is that it only works if you commit to it—you can't pretend to do a big, comprehensive design and then start developing, because under Waterfall, you'll never revisit that design; if it isn't truly complete, your project will suffer. Waterfall is also simpler, especially for inexperienced development teams and managers, than many methodologies that are more iterative and flexible. Waterfall also places an emphasis on documentation and source code, which supporters feel help improve long-term maintainability. Primarily, supporters feel that Waterfall is well-suited to stable software projects, such as shrink-wrap software, with a fairly known and fixed set of requirements. Because of its simplicity, Waterfall is often used in generic illustrations of software development where the precise methodology isn't particularly important, as in my book, *Definitive Guide to Quality Application Delivery.*

However, critics of Waterfall say that it is impossible, for any non-trivial project, to get one phase—such as the requirements or design—of a project completely worked out before moving to the next. They argue that today's rapid software development requirements force you to continually revisit earlier phases, and that attempting to stick with a strict Waterfall model impedes flexibility and encourages the poor practices that lead to poor quality. Modified versions of Waterfall, such as Peter DeGrace's Sashimi, offer improvements such as overlapping phases or "phases with feedback" to help address the otherwise-sequential nature of Waterfall.

Agile

Agile is a group of software development methodologies that are based on similar principles and was develop largely in backlash to the Waterfall method that was common at the time (and which is in fact still quite common). Agile methodologies generally promote project management processes that encourage inspection and adaptation—both excellent points that help produce a higher-quality product. Agile recognizes that most organizations are interested in rapid software development (generally, only large commercial software manufacturers aren't interested in developing as rapidly as possible, and even they keep a close eye on the clock), and is specifically designed to help align development with business needs.

Agile typically does things in small increments rather than trying to formulate long-reaching grand plans. Iterations are short time frames known as *timeboxes,* typically lasting less than a month. Each timebox includes a team working through an entire development process, including planning, requirements analysis, design, coding, unit testing, and acceptance testing. Short timeboxes produce less risk, because they are inherently simpler projects—you can only aim for so much in a week or two. Each iteration in and of itself might not represent a useful end product, but subsequent iterations repeat the process until a workable release is ready. You can think of each timebox as ending in a distinct functional milestone. The Agile Manifesto (http://agilemanifesto.org/) succinctly outlines the guiding principles behind these methodologies.

Some of the original Agile methodologies included Scrum, Crystal Clear, Extreme Programming, Adaptive Software Development, Feature Driven Development, and Dynamic Systems Development Method.

Each Agile timebox is essentially a mini-Waterfall, with the theory that it's easier for mere mortals to sit down and plan out a couple weeks' worth of work, see how it goes, and then sit down and tackle the next iteration. Agile methodologies do require more experienced and involved managers, since the focus on continual review and feedback, plus the need to coordinate successive timeboxes, places a great deal of responsibility on management. The cost of Agile is re-work. As you go along, you learn that the way you solved the problem yesterday, with yesterday's requirements, may not work today. The re-work means that you will also need to test to determine the side effects of the work. The code is being revised on an ongoing and fluid basis, so the need for consistent coding practices becomes more critical. Developers will need to understand what existing modules do in order to revise them.

Incremental/Iterative (Extreme Programming)

While Agile can be seen as a set of mini-Waterfall timeboxes, Extreme Programming is an Agile development methodology that almost entirely eschews the sequential Waterfall approach. It encompasses and embraces almost continual changes to the requirements as a natural part of development, and believes that adapting to change is more realistic and beneficial than attempting to define all requirements at the outset of a project.

Extreme Programming defines several "activities," including Coding, Testing, Listening, and Designing, which take place more or less simultaneously during a project's life cycle. It emphasizes coding standards, collective code ownership, simplified design, and continual testing—all of which can lead to higher-quality code. It is especially well-suited to new or prototype projects, where requirements are not understood at the outset and which evolve rapidly as the project continues. Small projects also work well with Extreme Programming. Other types of projects may also work well but require high levels of discipline and a very functional, communicative team that have few motives beyond the project itself—in other words, office politics can kill an Extreme project extremely quickly.

One of Extreme Programming's drawbacks is the equally extreme amount of discipline that must be maintained by the entire project team in order for the methodology to work properly. Extreme projects may have unstable requirements, which can lead to scope creep and—especially on outsourced projects—skyrocketing costs. Extreme also requires that programmers adopt a user-centric viewpoint and assumes that programmers want to do what's best for the user and that programmers *understand* what's best for the user; this may not always be the case. More rigid methodologies use a change control board to resolve the conflicts between what users need and what programmers want to do. Extreme is seen by many as a slightly organized form of "cowboy coding" (that is, coding without any methodology at all), and poorly managed Extreme projects can quickly devolve into ad-hoc programming that wastes resources.

Perhaps its biggest drawback is that little evidence exists to support the viability of large Extreme programming teams; claims have been made for teams as large as 60, but the pervasive feeling amongst experts is that a dozen or so team members is about the limit, unless the project can be successfully partitioned into multiple standalone teams.

> **Note**
>
> I'm simplifying many of the arguments for and against Extreme, primarily because the details have been the subject of countless books, articles, and so forth. The matter is made more complex by the fact that organizations are constantly developing hybrid methodologies; while a core principle in Extreme is that you either have to do everything it says or nothing, JPMorgan Chase has successfully combined principles of Extreme with methodologies from Capability Maturity Model Integration (CMMI) and Six Sigma. The world of development methodologies is rich and complex.

Others

Software development methodologies are rich in diversity and similarity. Some standouts include:

- Capability Maturity Model Integration (CMMI)—Designed by the Software Engineering Institute at Carnegie-Mellon University, CMMI defines several key process areas, including Requirements Management, Validation, Product Integration, and more (many more). You can read more about it at http://www.sei.cmu.edu/cmmi/.
- Six Sigma—Technically a business management strategy, originally developed by Motorola, Six Sigma has been adapted to numerous processes including software development. It focuses on quality management methods and fosters the creation of individuals ("black belts") within the organization who are experts at this method. Inspired by the Deming Total Quality Management (TQM) principles, Six Sigma defines various roles and processes that are used to drive quality in a final product. A good introduction to the use of Six Sigma in software development can be found at http://www.sei.cmu.edu/news-at-sei/features/2004/1/feature-3.htm.

Continuous Integrated Testing and Nightly Builds

An aspect of more agile methodologies (including Extreme Programming) is an emphasis on continuous testing; not just the *unit testing* performed by individual developers on their own code, but of continuous *integration testing* done by taking all the developers' work for the day, building a completed application, and running a series of tests against it (this can actually be helpful in modified Waterfall-style projects, too). This technique can really only be used when extending an existing application, or in the later life of a brand-new application; obviously, you need to get to a point in the code where you have enough to compile a usable application in the first place.

Essentially, a *nightly build* takes all the code that was checked in that day and attempts to compile it. The compiled result is then run through a series of standardized—and often automated—tests. The results are then provided to developers the next day, and helps drive what those developers will be working on that day.

Large development teams—think Microsoft's Windows Server team—rely heavily on nightly builds and continuous integrated testing to test the literally thousands of distinct components and capabilities of complex software applications, and to catch bugs as early as possible. Continuous testing doesn't fit within a strict Waterfall model, which waits until a specific phase to start integrated testing. Further, nightly builds and integration testing are difficult without tools to automate the extremely tedious and repetitive tasks involved. With the right tools, however, and as part of the right development methodology, this technique can help deliver a higher-quality product, even in extremely large and complex projects.

Defining Your Coding Standards

So where should you begin?

- Start by selecting a development methodology. This will provide a standardized set of processes, phases, and tasks to help guide your software development project.
- Select a set of coding standards—ideally, ones provided by a major manufacturer such as Microsoft or some other major authority. These can serve as a starting point for developing your own internal "style guide."
- Set up a small committee of developers and managers to further establish your own internal programming "style guide."
- Begin training developers to use Visual Studio's style and coding features. Encourage developers to force themselves to use these new features until they're habitual, and provide them with the time needed to do so.

Of course, the only way to ensure that your coding standards are being observed consistently is through code analysis and peer review—which shall be the subject of the next chapter.

Download Additional eBooks from Realtime Nexus!

Realtime Nexus—The Digital Library provides world-class expert resources that IT professionals depend on to learn about the newest technologies. If you found this eBook to be informative, we encourage you to download more of our industry-leading technology eBooks and video guides at Realtime Nexus. Please visit http://nexus.realtimepublishers.com.