Realtime publishers

The Definitive Guide™ To

Building Code Quality

sponsored by

MICRO FOCUS®
Leading the Evolution™

Don Jones

Copyright Statement

Chapter 3: Coding Analysis and Peer Reviews

All the coding standards in the world, combined with all the testing in the world, cannot provide the same level of code quality that you can achieve when you add in automated code analysis and peer code reviews. Consider this: Developers are people. Quite often they're smart people, and they're often smart people under pressure from deadlines. If you remember the 1980s television show "MacGyver," you know what smart people do under pressure: they improvise. Merely making sure that code compiles and runs properly doesn't mean improvisation didn't occur—you can have some pretty ugly-looking code that will pass those two tests. Consider this pseudo-code:

```
Function RestartServers(X) {
  ForEach (S in X) {
    If (S.Name contains any of ("DC","SQL","EXCH")) {
      Return Null
    }
    S.Connect().Reboot()
  }
}
```

This code might violate a few standards. For one, it has multiple exit points—it can exit by falling to the end of the function and through the Return statement in the middle. Variable naming is shoddy—what do "S" and "X" represent? Multiple operations—a Connect and Reboot—are being conducted on a single line of code. All of these are generally held as bad practices by some professional developers, but the code would compile and execute just fine.

Code analysis and a peer review, however, might catch these problems. Fixing them would result in less-complex code, easier-to-maintain code, and so forth—all of which leads to higher-quality code.

Understanding Peer Review of Code

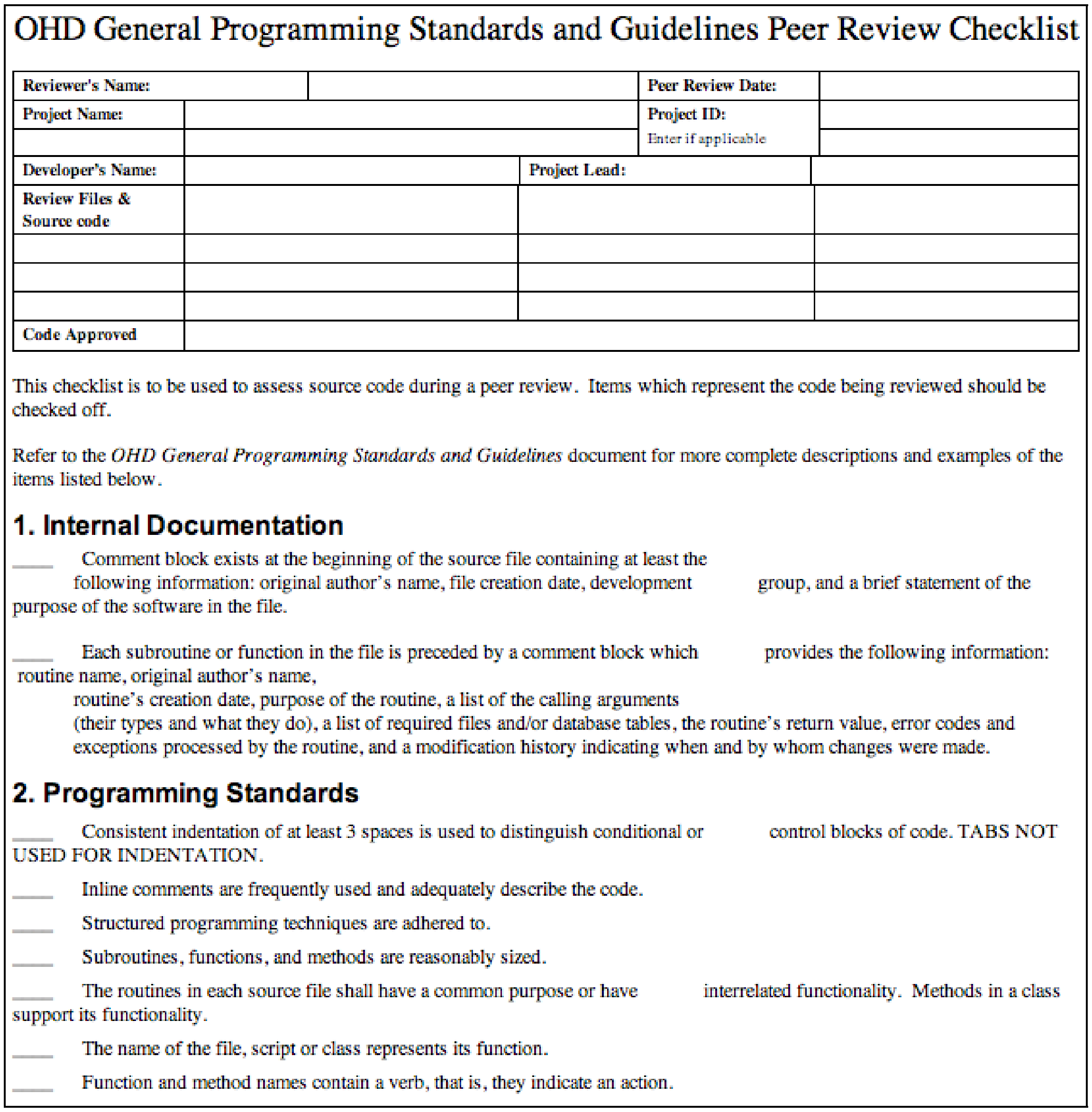
Peer review can be a sensitive subject. Not many developers like to have someone looking over their shoulder and critiquing their art (remember that coding is as much art as science, and developers can be as sensitive and temperamental as artists). Unlike a painting, however, code is *functional* art, and can benefit from a second look by another developer. Peer reviews offer a number of benefits, which should be communicated to developers:

- Junior developers can learn from their own mistakes, and mini-mentoring sessions can be incorporated into the review
- Everyone can gain more practice in observing coding standards; in fact, by having *each* developer act as a reviewer, you'll find that they start observing coding standards more consistently in their own code
- Peer reviews can be a team-building opportunity, rather than a competitive event, when managed properly

The key is to make sure everyone knows that the peer review is an opportunity to put the collective code into the best possible condition—not to "pick on" any one person. The code is "owned" by the entire team (managers can emphasize that by rotating developers through different portions of the code on a regular basis), not by one developer; a peer review is a chance to make that common property the best it can be.

Peer reviews are, unfortunately, easy to skip. They take time, and time is often something development teams are perpetually short on. Management may not understand the value of peer review, and may not want to allocate time to it. That's a point of education: Industry statistics are very clear that peer-reviewed code is almost universally of higher quality than code that isn't peer reviewed. If management wants code that can outlast the career of a single developer—in other words, more maintainable, standardized code—then peer reviews are a must.

Start by adopting a set of guidelines for peer reviews. The National Oceanic and Atmospheric Administration (NOAA), for example, uses the site at http://www.nws.noaa.gov/oh/hrl/developers.html to document their programming standards and conventions, and to document checklists for peer reviews. These checklists help ensure that peer reviews don't descend into personal attacks, and help keep the review focused on value-added activities. Figure 3.1 shows a portion of a general peer review checklist; you can see that it is very to-the-point and focused on coding standards.



OHD General Programming Standards and Guidelines Peer Review Checklist

Reviewer's Name:		**Peer Review Date:**	
Project Name:		**Project ID:** Enter if applicable	
Developer's Name:		**Project Lead:**	
Review Files & Source code			
Code Approved			

This checklist is to be used to assess source code during a peer review. Items which represent the code being reviewed should be checked off.

Refer to the *OHD General Programming Standards and Guidelines* document for more complete descriptions and examples of the items listed below.

1. Internal Documentation

____ Comment block exists at the beginning of the source file containing at least the following information: original author's name, file creation date, development group, and a brief statement of the purpose of the software in the file.

____ Each subroutine or function in the file is preceded by a comment block which provides the following information: routine name, original author's name, routine's creation date, purpose of the routine, a list of the calling arguments (their types and what they do), a list of required files and/or database tables, the routine's return value, error codes and exceptions processed by the routine, and a modification history indicating when and by whom changes were made.

2. Programming Standards

____ Consistent indentation of at least 3 spaces is used to distinguish conditional or control blocks of code. TABS NOT USED FOR INDENTATION.

____ Inline comments are frequently used and adequately describe the code.

____ Structured programming techniques are adhered to.

____ Subroutines, functions, and methods are reasonably sized.

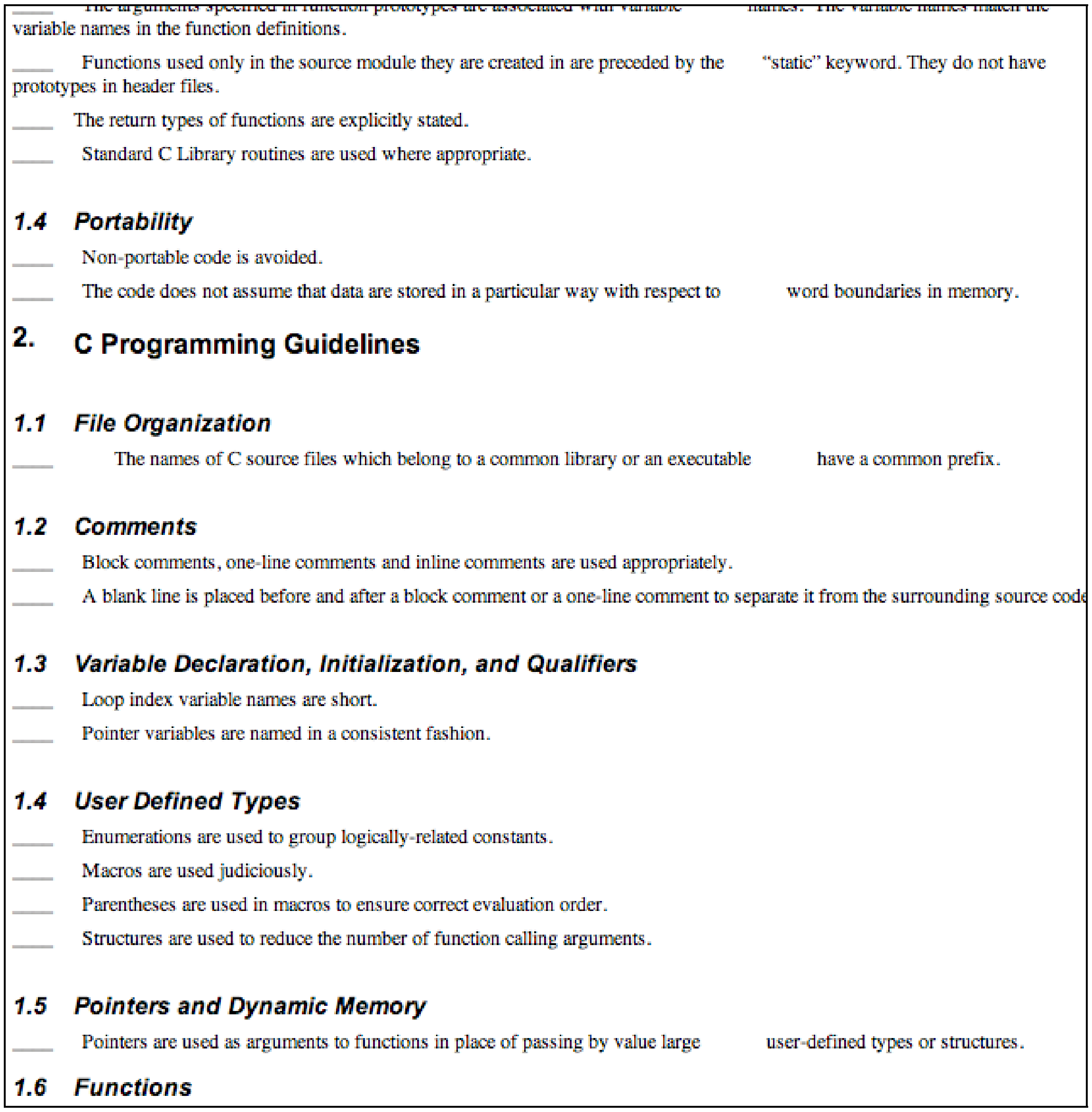
____ The routines in each source file shall have a common purpose or have interrelated functionality. Methods in a class support its functionality.

____ The name of the file, script or class represents its function.

____ Function and method names contain a verb, that is, they indicate an action.

Figure 3.1: General peer review checklist.

Figure 3.2 illustrates a checklist that is specific to C programming and which contains style and coding conventions only applicable to a C project. This combination of a "general" and "language-specific" checklist is useful for environments that maintain code in multiple languages.



____ The arguments specified in function prototypes are associated with variable names. The variable names match the variable names in the function definitions.

____ Functions used only in the source module they are created in are preceded by the "static" keyword. They do not have prototypes in header files.

____ The return types of functions are explicitly stated.

____ Standard C Library routines are used where appropriate.

1.4 Portability

____ Non-portable code is avoided.

____ The code does not assume that data are stored in a particular way with respect to word boundaries in memory.

2. C Programming Guidelines

1.1 File Organization

____ The names of C source files which belong to a common library or an executable have a common prefix.

1.2 Comments

____ Block comments, one-line comments and inline comments are used appropriately.

____ A blank line is placed before and after a block comment or a one-line comment to separate it from the surrounding source code

1.3 Variable Declaration, Initialization, and Qualifiers

____ Loop index variable names are short.

____ Pointer variables are named in a consistent fashion.

1.4 User Defined Types

____ Enumerations are used to group logically-related constants.

____ Macros are used judiciously.

____ Parentheses are used in macros to ensure correct evaluation order.

____ Structures are used to reduce the number of function calling arguments.

1.5 Pointers and Dynamic Memory

____ Pointers are used as arguments to functions in place of passing by value large user-defined types or structures.

1.6 Functions

Figure 3.2: Language-specific checklist.

Typically, it's not practical for every line of code in an application to be peer reviewed. Instead, select a sampling of subroutines to review, and ask that the developer being reviewed take any suggestions and apply them to all of the code—not just the reviewed code.

Note

This is where code metrics can come in handy. By selecting more complex portions of code, as identified by a code metrics analysis, you'll be reviewing the code with the highest potential risk—that is, the code that can stand to benefit most from the extra attention of a peer review.

Peer review suggestions should be identified as belonging to one of three categories:

- Cosmetic—Items primarily related to coding style
- Minor—Items that might relate to the efficiency of the software but do not impact its functionality
- Major—Items that relate to the functionality of the software

Although all these items should, ideally, be addressed, categorizing them in this fashion helps prioritize them in case management needs to make a tough call on quality versus other factors, like time and cost. Perhaps most importantly, each comment should be accompanied by a suggested solution. It's *never* enough to just point out a problem! Point out a solution as well. Doing so helps turn the peer review from a potential personal attack into a team-building exercise that emphasizes the shared ownership of the code.

Code Analysis Overview

In the previous chapter, I briefly mentioned the code metrics analysis tool that is provided with some editions of Visual Studio 2008 (specifically, the higher-priced Team System editions). Although there are third-party Visual Studio add-ins that provide deeper and broader metrics, Team System's tool provides a good starting point for understanding the value of such a tool. Also, code analysis generally encompasses more than just the basic metrics provided by this tool.

Note

Most of the Visual Studio-specific features discussed in this chapter are available only in the Team editions of the product, and some of them require interaction with a Visual Studio Team Foundation Server. Third-party products provide similar feature capabilities that might be compatible with other editions of Visual Studio, whether you're using Team System or not.

Code analysis consists of statistical analysis as well as other rules; those other rules may address naming, localization, portability, security, usage, and other coding conventions (which we discussed in the previous chapter). The idea is that code analysis is a fairly automated process, whereas peer review is a more manual process that involves another human being looking at a developer's code.

Note

Some of the statistics I'll discuss in the next few sections were introduced in the first chapter of this book. Many of them come from tools provided by Microsoft as part of Visual Studio Team Edition; because some of these statistics are only exposed by that tool and by a few other third-party tools, they may not be "standardized" to the point where every environment will use them. However, they're still valuable to review and understand because in many cases they parallel more commonly-used statistics.

Class Coupling

One of the items included in Visual Studio Team System's Code Metrics analysis is *class coupling.* This indicates the total number of dependencies that one item has on other types. It excludes primitives and built-in types (such as String or Object). The higher the class coupling number, the greater the likelihood that changes in other types will cause ripple effects through the analyzed item. A lower value at the type level indicates candidates for potential reuse. At http://blogs.msdn.com/fxcop/archive/2007/10/03/new-for-visual-studio-2008-code-metrics.aspx, Microsoft provides a helpful illustration of how class coupling works; this is reproduced in Figure 3.3.

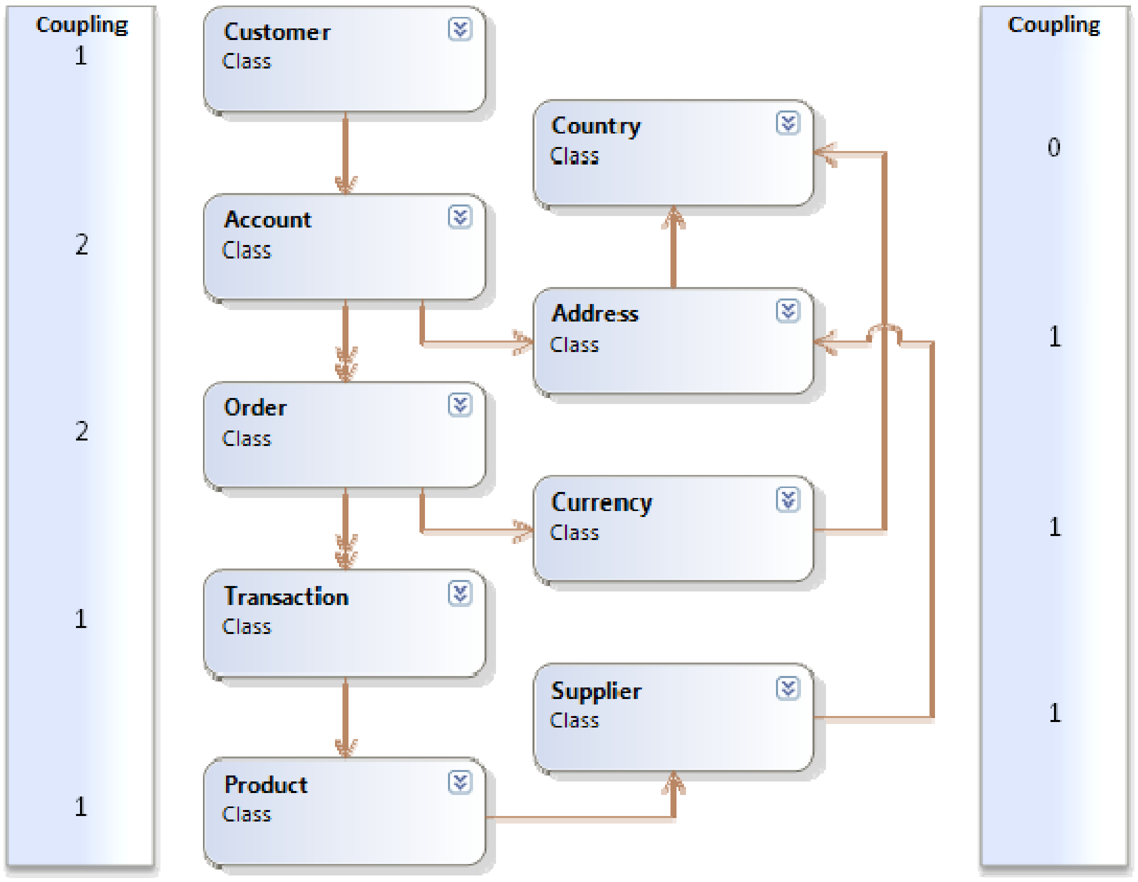


Figure 3.3: Class coupling diagram.

In this diagram, the Order class depends upon the Currency class, which depends upon the Country class. Country depends on nothing, so it has a coupling level of 0. Currency depends on Country, so it has a dependency of 1. Order depends on Currency, which in turn depends on Country, so it has a level of 2. The higher the level, the more at risk the class is of breaking.

This should not imply that class dependencies are bad. Combining common code into a single point of maintenance reduces copies of the same code throughout the project. It does mean that the more complex the dependencies become, the more care developers must take when working with them. Fortunately, modern tools simplify the process of tracking these dependencies, identifying risks, and mitigating issues when changing one class will affect others. Choosing tools that help prevent developers from making these mistakes can keep the code safe and still allow the developers to implement the best coding practices.

Depth of Inheritance

Depth of inheritance is a measurement of how many underlying types or classes a given class inherits from. In the .NET Framework, Object is an intrinsic type with a depth of 0; anything inheriting from it has a depth of 1. Anything inheriting from that level would have a depth of 2, and so forth. Much like class coupling, a higher number indicates a riskier element because changes to underlying elements can cause cascading effects. Figure 3.4 illustrates.

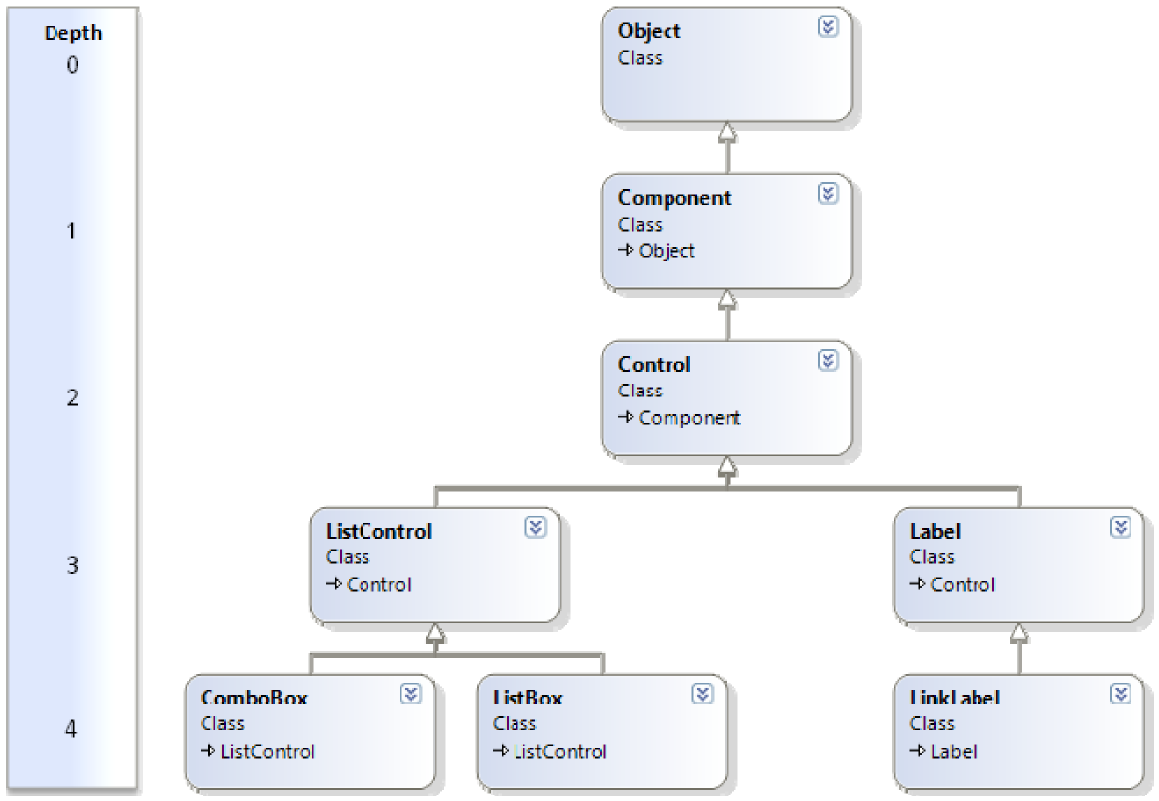
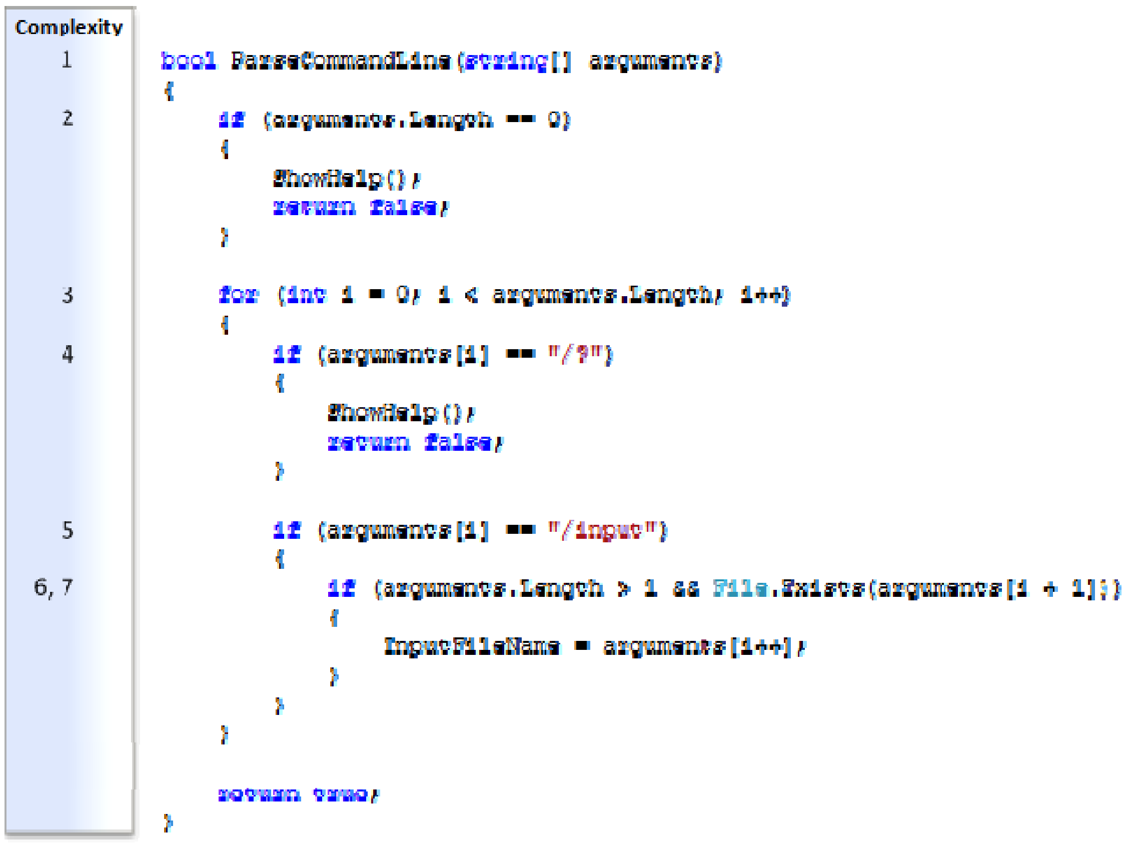


Figure 3.4: Depth of inheritance.

Here, the ComboBox is a riskier class because it has an inheritance depth of 4. Changes to Object, Component, Control, or ListControl may cause cascading effects that introduce bugs or other problems into ComboBox.

Cyclomatic Complexity

At each level, *cyclomatic complexity* measures the total number of individual paths through code. Essentially, you count the number of decision points, such as If constructs, and add 1. Because tests should strive to test every possible code path, the cyclomatic complexity is a rough measurement of the minimum number of different unit tests that will need to be performed to thoroughly test the code. Lower numbers are obviously more desirable! Figure 3.5 shows a block of C# code with the corresponding calculations for cyclomatic complexity.



```
Complexity
   1       bool ParseCommandLine(string[] arguments)
           {
   2           if (arguments.Length == 0)
               {
                   ShowHelp();
                   return false;
               }

   3           for (int i = 0; i < arguments.Length; i++)
               {
   4               if (arguments[i] == "/?")
                   {
                       ShowHelp();
                       return false;
                   }

   5               if (arguments[i] == "/input")
                   {
 6, 7                  if (arguments.Length > i && File.Exists(arguments[i + 1]))
                       {
                           InputFileName = arguments[i++];
                       }
                   }
               }

               return true;
           }
```

Figure 3.5: Cyclomatic complexity.

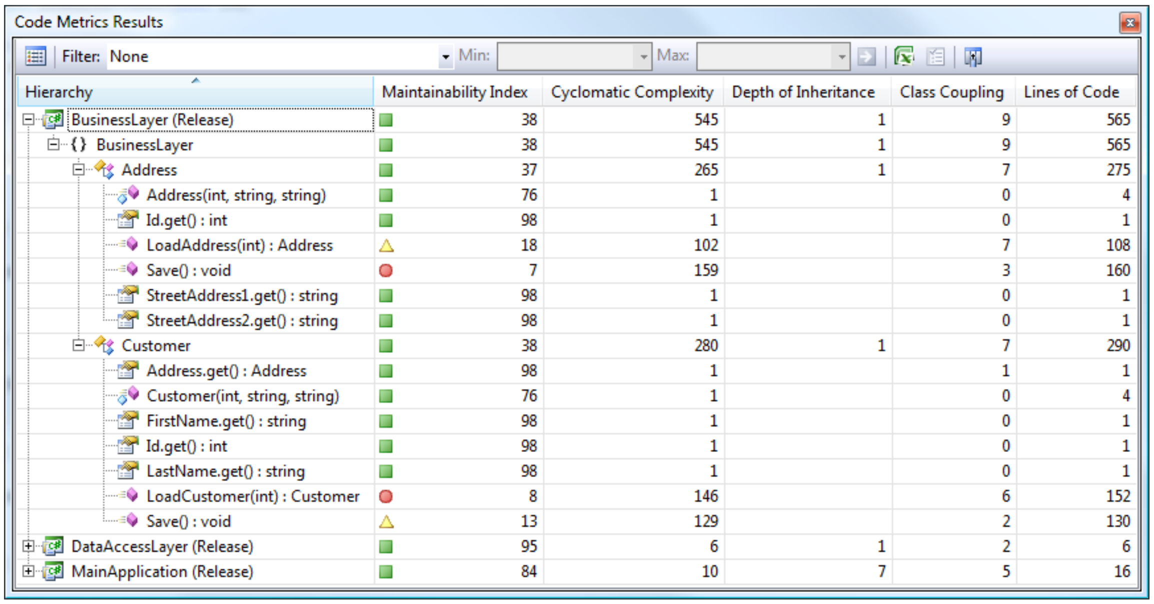
Cyclomatic complexity is a much more standardized metric, and is something that most non-Microsoft tools will calculate for you.

Lines of Code

An oft-overused metric that nonetheless has some value, *lines of code (LoC)* should not be used to calculate developer productivity or programming progress because those activities involve more than just hitting “Enter” in a code editor. Bill Gates famously suggested that using LoC to calculate progress or productivity was like using weight to calculate the progress of building an airplane. Sure, the airplane gets heavier as you go but that’s not necessarily an indication of desirable progress. Commonly, LoC excludes white space, comments, braces on empty lines, member declarations, and other non-functional lines of code.

Maintainability Index

The *maintainability index* is an index number, commonly expressed from 0 to 100 (although different tools may use different ranges) indicating the overall maintainability of that member of type. In Visual Studio Team System's Code Metrics analysis, shown in Figure 3.6, a 100 indicates a "100% maintainable" element, while a 0 indicates an element that is going to be difficult, if not impossible, to maintain over the long term.

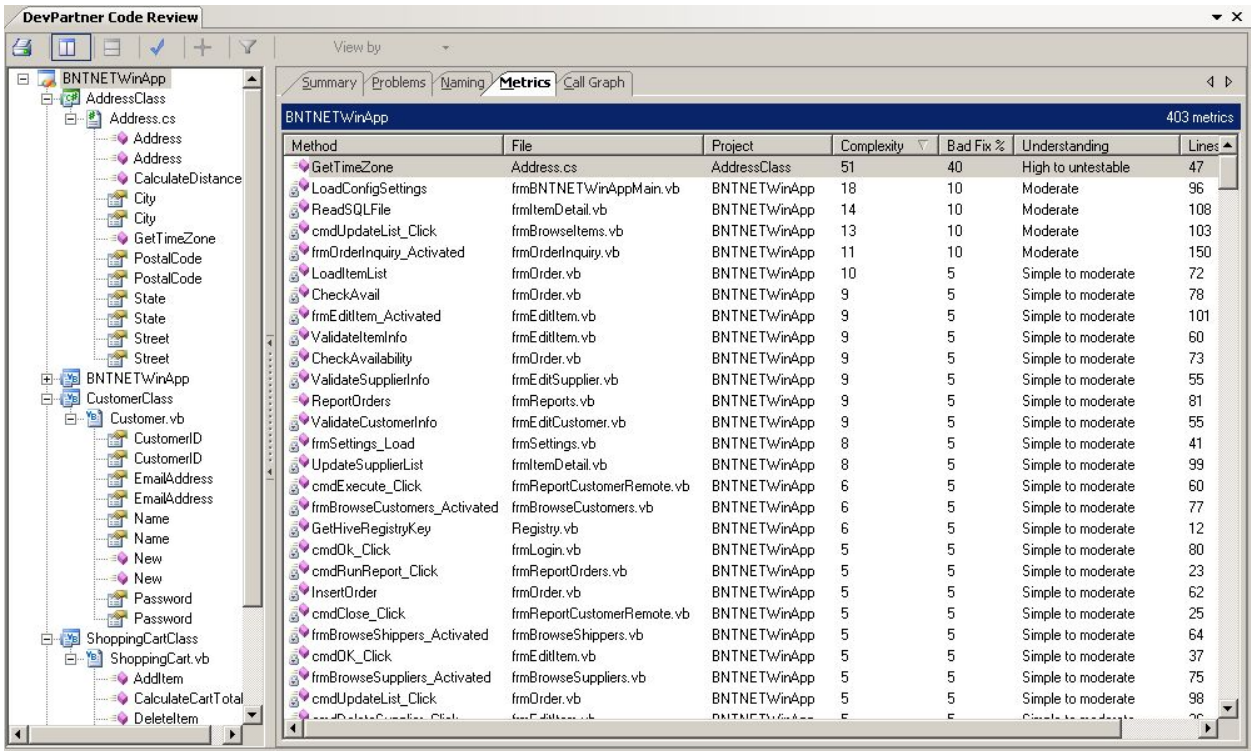


Hierarchy	Maintainability Index		Cyclomatic Complexity	Depth of Inheritance	Class Coupling	Lines of Code
BusinessLayer (Release)	🟩	38	545	1	9	565
{ } BusinessLayer	🟩	38	545	1	9	565
Address	🟩	37	265	1	7	275
Address(int, string, string)	🟩	76	1		0	4
Id.get() : int	🟩	98	1		0	1
LoadAddress(int) : Address	△	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 3.6: Code metrics with maintainability index.

The index itself is a rollup of other metrics, including the Halstead volume (which factors in the number and use of operands and operators), cyclomatic complexity, and lines of code. I have mixed feelings about metrics like this; while I think it's useful to have a dashboard-style "rollup," this type of metric can also hide a lot of underlying statistics and issues that really should be reviewed. Many third-party code analysis tools offer similar "rollup" views, and sometimes offer the ability to easily drill down into problem areas to get more detail—and that detail can help you prioritize and plan what you're going to do next. The point is to remember that the dashboard-style view is intended as a high-level overview; you shouldn't make project management decisions based solely on this information.

Figure 3.7 shows a similar dashboard-style view from Micro Focus DevPartner, which also works with Visual Studio. This view aims to provide information that's a bit more actionable by management; while it also lists a complexity statistic, it includes "understanding" and "bad fix" measurements as well. Code modules with a "high to untestable" understanding, for example, are a problem: they're overly-complex, may not be able to be tested thoroughly, and represent a very real risk in terms of code quality. The "bad fix" number is especially significant because it represents the probability that some future fix to the code will actually introduce a new problem simply because of the code's complexity; high numbers such as 40% means the code may well be impossible to maintain into the future, and that calls for immediate management action to decide what to do—*now.*



BNTNETWinApp
- AddressClass
  - Address.cs
    - Address
    - Address
    - CalculateDistance
    - City
    - City
    - GetTimeZone
    - PostalCode
    - PostalCode
    - State
    - State
    - Street
    - Street
- BNTNETWinApp
- CustomerClass
  - Customer.vb
    - CustomerID
    - CustomerID
    - EmailAddress
    - EmailAddress
    - Name
    - Name
    - New
    - New
    - Password
    - Password
- ShoppingCartClass
  - ShoppingCart.vb
    - AddItem
    - CalculateCartTotal
    - DeleteItem

BNTNETWinApp 403 metrics

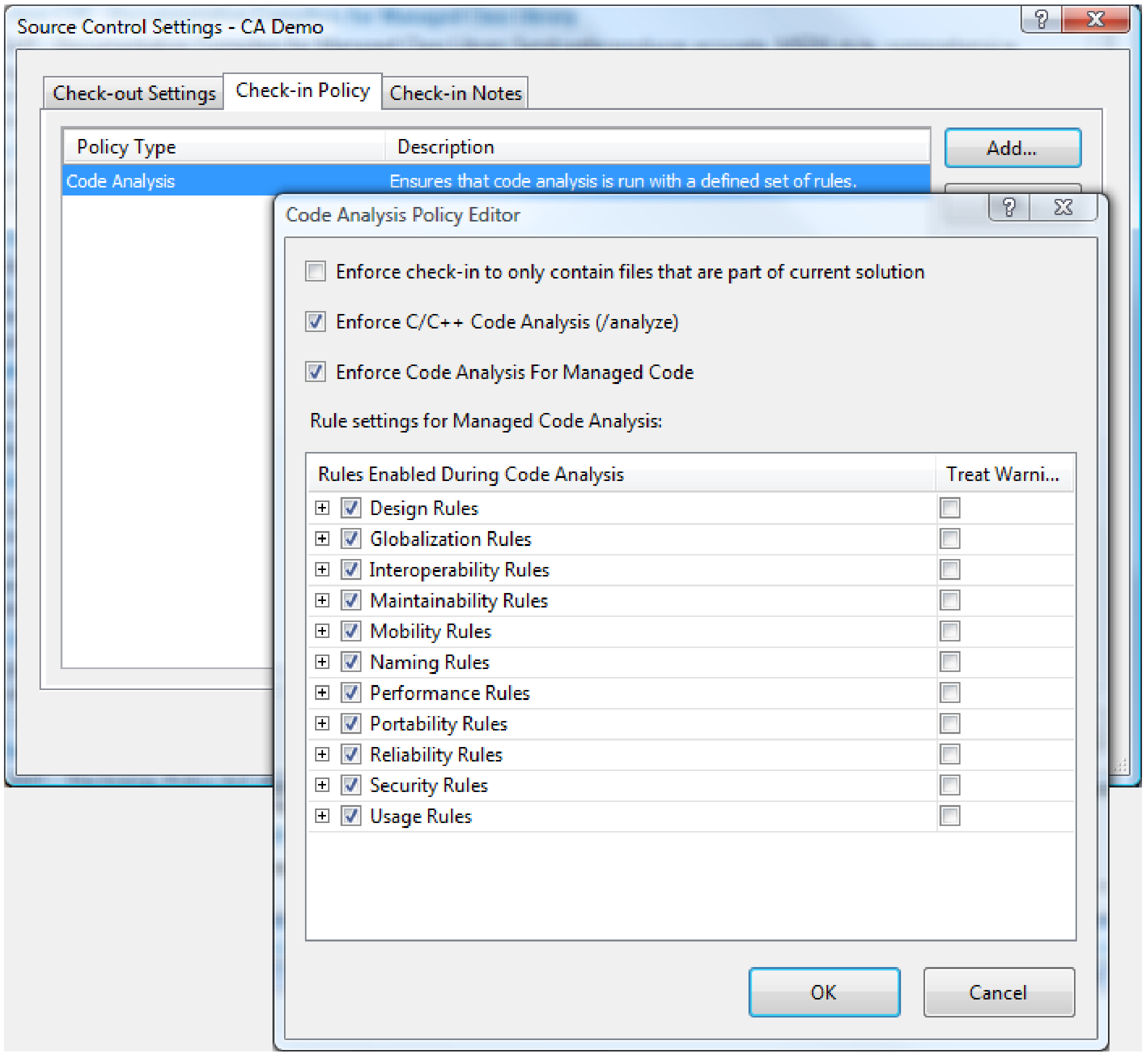
Method	File	Project	Complexity	Bad Fix %	Understanding	Lines
GetTimeZone	Address.cs	AddressClass	51	40	High to untestable	47
LoadConfigSettings	frmBNTNETWinAppMain.vb	BNTNETWinApp	18	10	Moderate	96
ReadSQLFile	frmItemDetail.vb	BNTNETWinApp	14	10	Moderate	108
cmdUpdateList_Click	frmBrowseItems.vb	BNTNETWinApp	13	10	Moderate	103
frmOrderInquiry_Activated	frmOrderInquiry.vb	BNTNETWinApp	11	10	Moderate	150
LoadItemList	frmOrder.vb	BNTNETWinApp	10	5	Simple to moderate	72
CheckAvail	frmOrder.vb	BNTNETWinApp	9	5	Simple to moderate	78
frmEditItem_Activated	frmEditItem.vb	BNTNETWinApp	9	5	Simple to moderate	101
ValidateItemInfo	frmEditItem.vb	BNTNETWinApp	9	5	Simple to moderate	60
CheckAvailability	frmOrder.vb	BNTNETWinApp	9	5	Simple to moderate	73
ValidateSupplierInfo	frmEditSupplier.vb	BNTNETWinApp	9	5	Simple to moderate	55
ReportOrders	frmReports.vb	BNTNETWinApp	9	5	Simple to moderate	81
ValidateCustomerInfo	frmEditCustomer.vb	BNTNETWinApp	9	5	Simple to moderate	55
frmSettings_Load	frmSettings.vb	BNTNETWinApp	8	5	Simple to moderate	41
UpdateSupplierList	frmItemDetail.vb	BNTNETWinApp	8	5	Simple to moderate	99
cmdExecute_Click	frmReportCustomerRemote.vb	BNTNETWinApp	6	5	Simple to moderate	60
frmBrowseCustomers_Activated	frmBrowseCustomers.vb	BNTNETWinApp	6	5	Simple to moderate	77
GetHiveRegistryKey	Registry.vb	BNTNETWinApp	6	5	Simple to moderate	12
cmdOk_Click	frmLogin.vb	BNTNETWinApp	5	5	Simple to moderate	80
cmdRunReport_Click	frmReportOrders.vb	BNTNETWinApp	5	5	Simple to moderate	23
InsertOrder	frmOrder.vb	BNTNETWinApp	5	5	Simple to moderate	62
cmdClose_Click	frmReportCustomerRemote.vb	BNTNETWinApp	5	5	Simple to moderate	25
frmBrowseShippers_Activated	frmBrowseShippers.vb	BNTNETWinApp	5	5	Simple to moderate	64
cmdOK_Click	frmEditItem.vb	BNTNETWinApp	5	5	Simple to moderate	37
frmBrowseSuppliers_Activated	frmBrowseSuppliers.vb	BNTNETWinApp	5	5	Simple to moderate	75
cmdUpdateList_Click	frmOrder.vb	BNTNETWinApp	5	5	Simple to moderate	98

Figure 3.7: DevPartner code analysis metrics.

Code Analysis Policy in Visual Studio

Visual Studio supports a feature called Code Analysis Check-In Policies. Essentially, these are rule sets implemented by Visual Studio and its Team Foundation Server, which analyze all code checked into the Team Foundation Server's source code repository. These rules are designed to help ensure that only higher-quality code is checked into the repository, and when code fails to meet a rule, the developer receives feedback that helps guide corrective actions.

Figure 3.8 illustrates part of this feature. You can see that the source control Check-In Policy includes a specific policy called Code Analysis that is comprised of numerous rule sets which are enforced on check-in.



Policy Type	Description
Code Analysis	Ensures that code analysis is run with a defined set of rules.

- [ ] Enforce check-in to only contain files that are part of current solution
- [x] Enforce C/C++ Code Analysis (/analyze)
- [x] Enforce Code Analysis For Managed Code

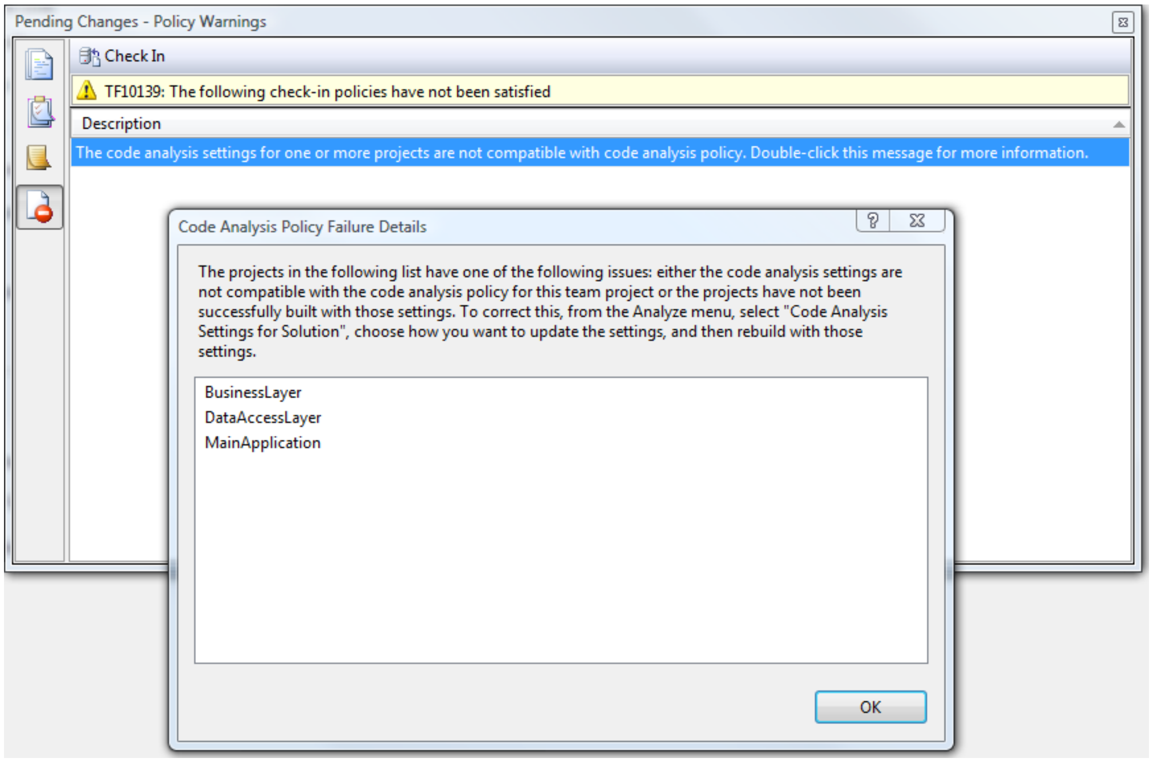
Rule settings for Managed Code Analysis:

Rules Enabled During Code Analysis	Treat Warni...
✓ Design Rules	☐
✓ Globalization Rules	☐
✓ Interoperability Rules	☐
✓ Maintainability Rules	☐
✓ Mobility Rules	☐
✓ Naming Rules	☐
✓ Performance Rules	☐
✓ Portability Rules	☐
✓ Reliability Rules	☐
✓ Security Rules	☐
✓ Usage Rules	☐

Figure 3.8: Code Analysis check-in policy.

The Team Foundation Server policy can also be migrated to individual projects, which allows smaller projects to use the rule sets without having to check the code into the change control system. Local (non-Team) projects can also have their own distinct rule sets.

Figure 3.9 shows the feedback offered to developers when something fails a check-in policy. They're told which projects have problems. The integration between the Team Foundation Server and Visual Studio is actually interesting. Basically, developers can attempt to build their solution against a rule set. Any failures are described in detail within the IDE. If a developer fixes the problems and successfully builds the project against the rule set, it can be checked into Team Foundation Server under the same rule set.


Pending Changes - Policy Warnings
Check In
Description
Code Analysis Policy Failure Details
OK

TF10139: The following check-in policies have not been satisfied

The code analysis settings for one or more projects are not compatible with code analysis policy. Double-click this message for more information.

The projects in the following list have one of the following issues: either the code analysis settings are not compatible with the code analysis policy for this team project or the projects have not been successfully built with those settings. To correct this, from the Analyze menu, select "Code Analysis Settings for Solution", choose how you want to update the settings, and then rebuild with those settings.

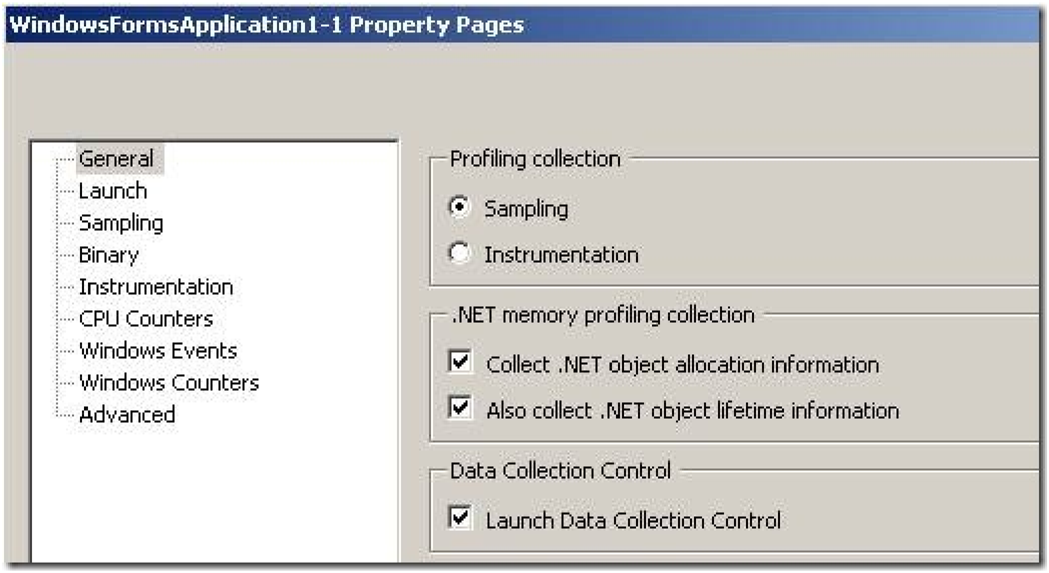
BusinessLayer
DataAccessLayer
MainApplication

Figure 3.9: Can't check in non-compliant code.

Note that third parties can extend this functionality or replace it with their own functionality in order to provide more varied rules and more complex code analysis.

Performance and Security Analysis Topics and Techniques

Visual Studio Team System's bundled tools for analyzing code performance and security are somewhat less mature than the basic code metrics features, and this is an area where a robust third-party add-in market has flourished for years. Team System does include a Performance Wizard, which is a profiling tool. The Wizard runs in two modes, as illustrated in Figure 3.10: Instrumentation and Sampling. The Wizard is a fairly basic performance tool that extends Visual Studio's capabilities; third-party tools, which have been on the market for longer, in most cases, extend Visual Studio in a similar fashion and often provide more detailed performance information.



WindowsFormsApplication1-1 Property Pages

Profiling collection

- [● Sampling]
- ○ Instrumentation

.NET memory profiling collection

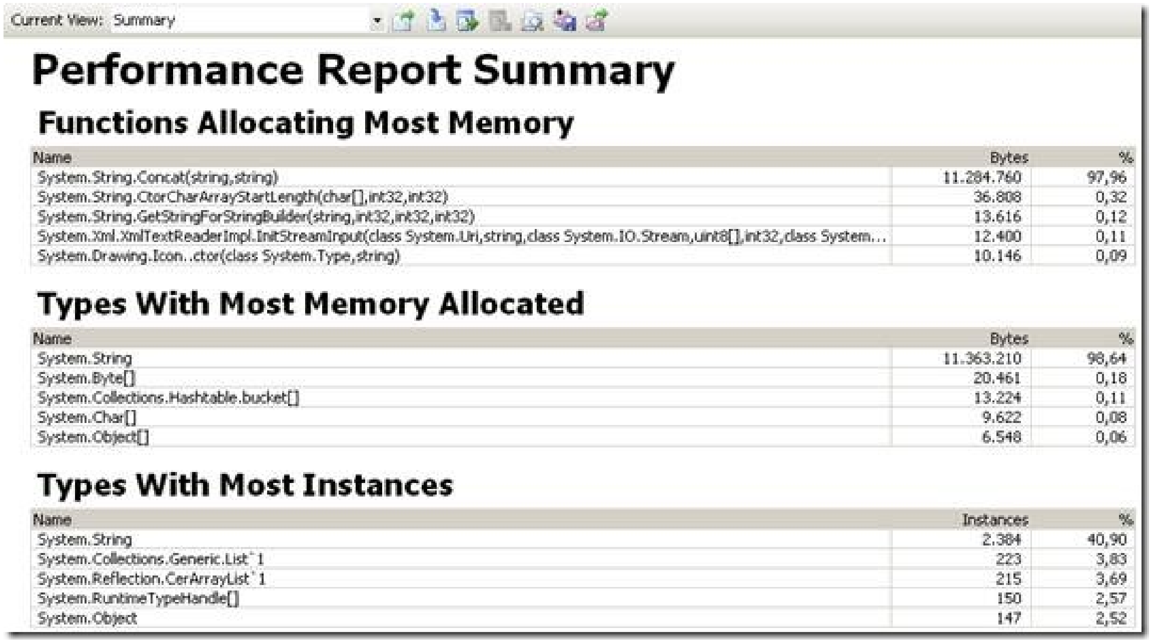
- [✓] Collect .NET object allocation information
- [✓] Also collect .NET object lifetime information

Data Collection Control

- [✓] Launch Data Collection Control

Figure 3.10: The Visual Studio Performance Wizard.

You typically begin by sampling your entire application, searching for performance spikes. Once you find them, you investigate specific spikes by using instrumentation. Figure 3.11 shows the performance report from a very simplistic application, indicating that almost 98% of the application's performance was taken by the String.Concat() method.



Current View: Summary

Performance Report Summary

Functions Allocating Most Memory

Name	Bytes	%
System.String.Concat(string,string)	11.284.760	97,96
System.String.CtorCharArrayStartLength(char[],int32,int32)	36.808	0,32
System.String.GetStringForStringBuilder(string,int32,int32,int32)	13.616	0,12
System.Xml.XmlTextReaderImpl.InitStreamInput(class System.Uri,string,class System.IO.Stream,uint8[],int32,class System...	12.400	0,11
System.Drawing.Icon..ctor(class System.Type,string)	10.146	0,09

Types With Most Memory Allocated

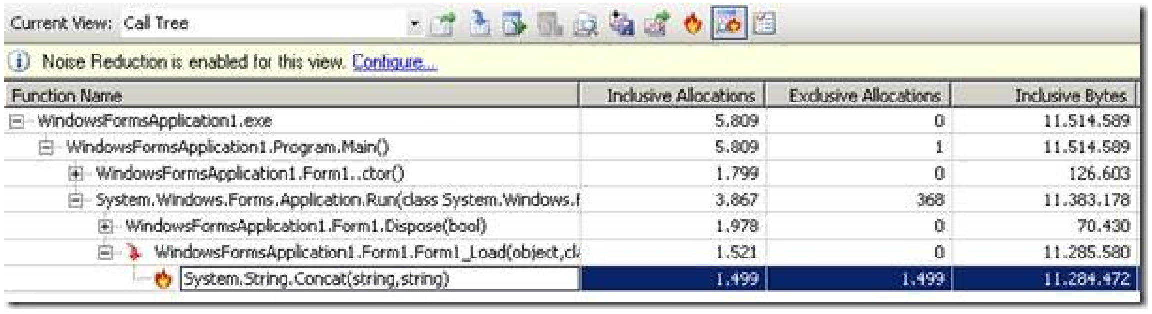
Name	Bytes	%
System.String	11.363.210	98,64
System.Byte[]	20.461	0,18
System.Collections.Hashtable.bucket[]	13.224	0,11
System.Char[]	9.622	0,08
System.Object[]	6.548	0,06

Types With Most Instances

Name	Instances	%
System.String	2.384	40,90
System.Collections.Generic.List`1	223	3,83
System.Reflection.CerArrayList`1	215	3,69
System.RuntimeTypeHandle[]	150	2,57
System.Object	147	2,52

Figure 3.11: An example performance report.

From that report, you could right-click that method and see the code that is actually calling it. This functionality is designed to help you track down lower-performance code throughout the project. As Figure 3.12 shows, Visual Studio Team Edition actually helps identify lower-performance calls in a call tree by using a “flame” icon.


Current View: Call Tree

Noise Reduction is enabled for this view. Configure...

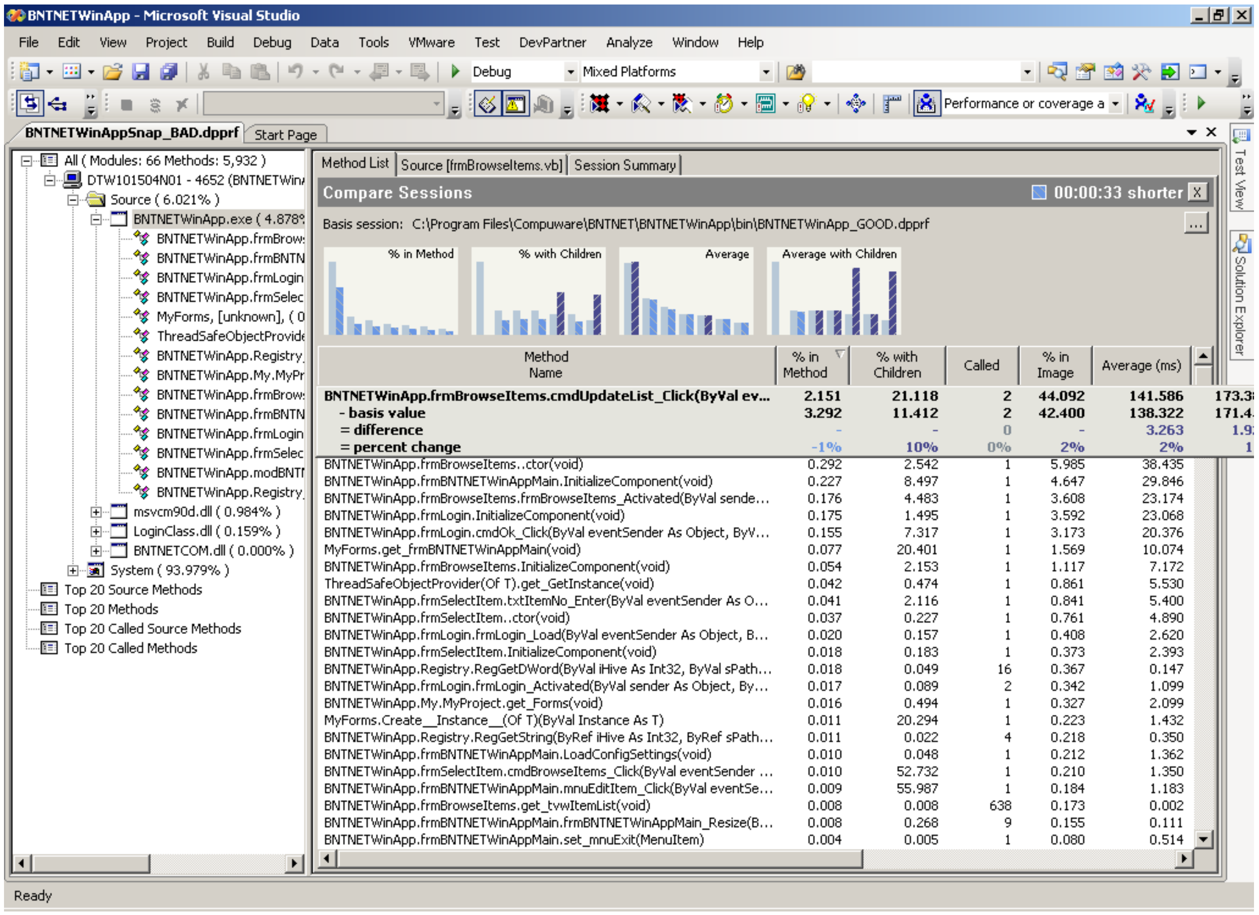
Function Name	Inclusive Allocations	Exclusive Allocations	Inclusive Bytes
WindowsFormsApplication1.exe	5.809	0	11.514.589
WindowsFormsApplication1.Program.Main()	5.809	1	11.514.589
WindowsFormsApplication1.Form1..ctor()	1.799	0	126.603
System.Windows.Forms.Application.Run(class System.Windows.F	3.867	368	11.383.178
WindowsFormsApplication1.Form1.Dispose(bool)	1.978	0	70.430
WindowsFormsApplication1.Form1.Form1_Load(object,cl	1.521	0	11.285.580
System.String.Concat(string,string)	1.499	1.499	11.284.472

Figure 3.12: Hot-path tracking in Visual Studio.

Visual Studio Team System’s code analysis tool, which I discussed earlier in this chapter, also has rule sets for security. Many organizations rely on third-party code analysis tools that offer broader feature sets for code security; in some cases, you can obtain third-party security analysis tools that have been built to help support specific legislative and industry initiatives, such as the Payment Card Industry Data Security Standard (PCI DSS).

So does Team System give you everything you need? For smaller projects, possibly. However, its performance and code analysis tools are still fairly first-generation, and it’s worth your time to evaluate the third-party tools out there because they can often provide much more detailed information. In addition, they integrate with Visual Studio in the same way that Team System’s tools do, so you’re not making your workflow any more complicated. Companies such as Micro Focus, Red Gate, and others offer tools that typically surpass Team System in terms of features.

For example, consider the differences between Team System’s performance reports and those provided by Micro Focus’ product, which also integrates with Visual Studio (including the more commonly-used and less-expensive Standard and Professional editions). Figure 3.13 shows how the tool can break down the amount of time spent executing individual methods within the application, making it easy to spot those methods that are occupying the most time, and focusing some effort to improve their performance.


BNTNETWinApp - Microsoft Visual Studio
File Edit View Project Build Debug Data Tools VMware Test DevPartner Analyze Window Help
Debug
Mixed Platforms
Performance or coverage a
BNTNETWinAppSnap_BAD.dpprf
Start Page
Method List
Source [frmBrowseItems.vb]
Session Summary
Test View
Solution Explorer
% in Method
% with Children
Average
Average with Children
Ready

All (Modules: 66 Methods: 5,932)
DTW101504N01 - 4652 (BNTNETWin
Source (6.021%)
BNTNETWinApp.exe (4.878
BNTNETWinApp.frmBrow
BNTNETWinApp.frmBNTN
BNTNETWinApp.frmLogin
BNTNETWinApp.frmSelec
MyForms, [unknown], (0
ThreadSafeObjectProvide
BNTNETWinApp.Registry
BNTNETWinApp.My.MyPr
BNTNETWinApp.frmBrow
BNTNETWinApp.frmBNTN
BNTNETWinApp.frmLogin
BNTNETWinApp.frmSelec
BNTNETWinApp.modBNTI
BNTNETWinApp.Registry
msvcm90d.dll (0.984%)
LoginClass.dll (0.159%)
BNTNETCOM.dll (0.000%)
System (93.979%)
Top 20 Source Methods
Top 20 Methods
Top 20 Called Source Methods
Top 20 Called Methods

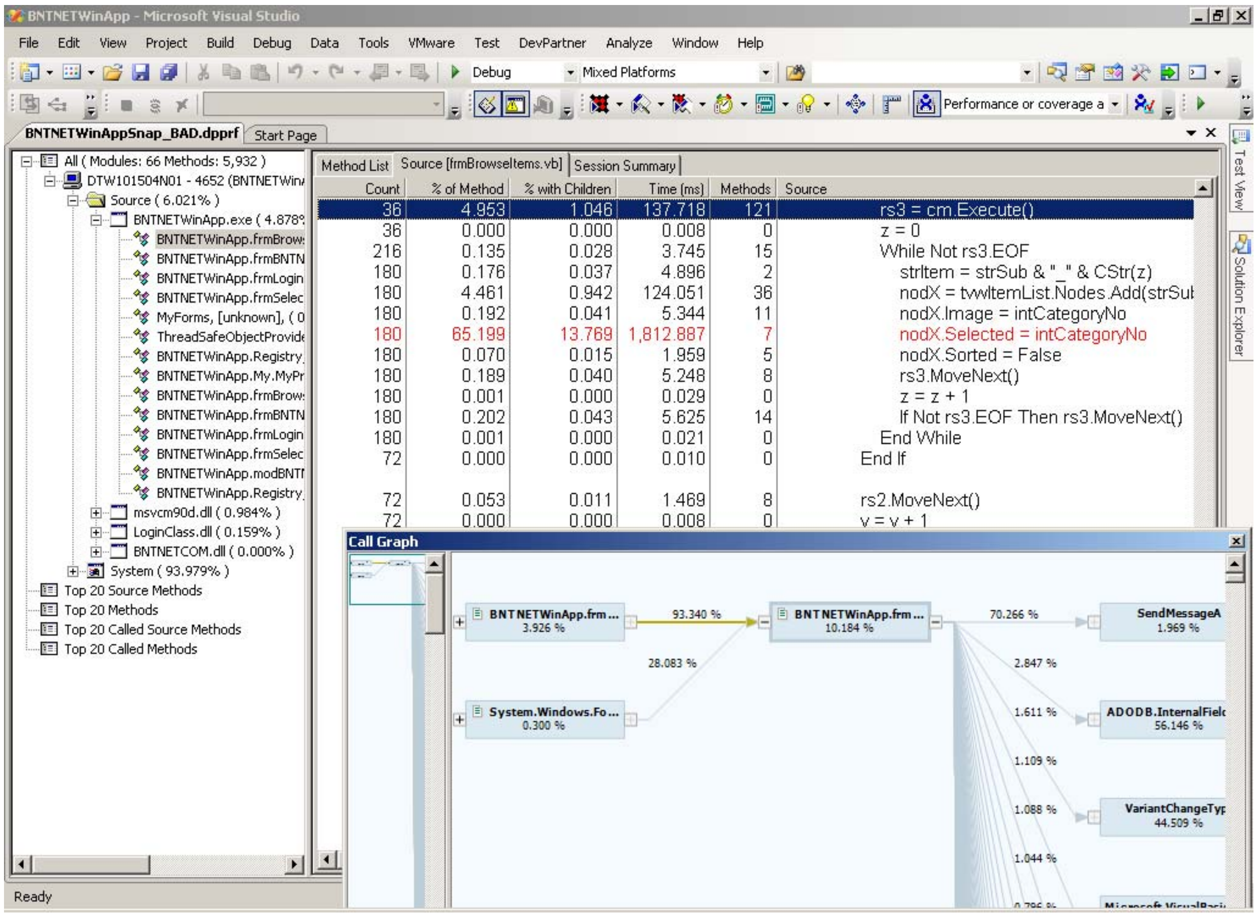
Compare Sessions 00:00:33 shorter

Basis session: C:\Program Files\Compuware\BNTNET\BNTNETWinApp\bin\BNTNETWinApp_GOOD.dpprf

Method Name	% in Method	% with Children	Called	% in Image	Average (ms)	
BNTNETWinApp.frmBrowseItems.cmdUpdateList_Click(ByVal ev...	**2.151**	**21.118**	**2**	**44.092**	**141.586**	**173.3**
- basis value	**3.292**	**11.412**	**2**	**42.400**	**138.322**	**171.4**
= difference	-	-	0	-	3.263	1.9
= percent change	-1%	10%	0%	2%	2%	1
BNTNETWinApp.frmBrowseItems..ctor(void)	0.292	2.542	1	5.985	38.435	
BNTNETWinApp.frmBNTNETWinAppMain.InitializeComponent(void)	0.227	8.497	1	4.647	29.846	
BNTNETWinApp.frmBrowseItems.frmBrowseItems_Activated(ByVal sende...	0.176	4.483	1	3.608	23.174	
BNTNETWinApp.frmLogin.InitializeComponent(void)	0.175	1.495	1	3.592	23.068	
BNTNETWinApp.frmLogin.cmdOk_Click(ByVal eventSender As Object, ByV...	0.155	7.317	1	3.173	20.376	
MyForms.get_frmBNTNETWinAppMain(void)	0.077	20.401	1	1.569	10.074	
BNTNETWinApp.frmBrowseItems.InitializeComponent(void)	0.054	2.153	1	1.117	7.172	
ThreadSafeObjectProvider(Of T).get_GetInstance(void)	0.042	0.474	1	0.861	5.530	
BNTNETWinApp.frmSelectItem.txtItemNo_Enter(ByVal eventSender As O...	0.041	2.116	1	0.841	5.400	
BNTNETWinApp.frmSelectItem..ctor(void)	0.037	0.227	1	0.761	4.890	
BNTNETWinApp.frmLogin.frmLogin_Load(ByVal eventSender As Object, B...	0.020	0.157	1	0.408	2.620	
BNTNETWinApp.frmSelectItem.InitializeComponent(void)	0.018	0.183	1	0.373	2.393	
BNTNETWinApp.Registry.RegGetDWord(ByVal iHive As Int32, ByVal sPath...	0.018	0.049	16	0.367	0.147	
BNTNETWinApp.frmLogin.frmLogin_Activated(ByVal sender As Object, By...	0.017	0.089	2	0.342	1.099	
BNTNETWinApp.My.MyProject.get_Forms(void)	0.016	0.494	1	0.327	2.099	
MyForms.Create__Instance__(Of T)(ByVal Instance As T)	0.011	20.294	1	0.223	1.432	
BNTNETWinApp.Registry.RegGetString(ByRef iHive As Int32, ByRef sPath...	0.011	0.022	4	0.218	0.350	
BNTNETWinApp.frmBNTNETWinAppMain.LoadConfigSettings(void)	0.010	0.048	1	0.212	1.362	
BNTNETWinApp.frmSelectItem.cmdBrowseItems_Click(ByVal eventSender ...	0.010	52.732	1	0.210	1.350	
BNTNETWinApp.frmBNTNETWinAppMain.mnuEditItem_Click(ByVal eventSe...	0.009	55.987	1	0.184	1.183	
BNTNETWinApp.frmBrowseItems.get_tvwItemList(void)	0.008	0.008	638	0.173	0.002	
BNTNETWinApp.frmBNTNETWinAppMain.frmBNTNETWinAppMain_Resize(B...	0.008	0.268	9	0.155	0.111	
BNTNETWinApp.frmBNTNETWinAppMain.set_mnuExit(MenuItem)	0.004	0.005	1	0.080	0.514	

Figure 3.13: Performance by method.

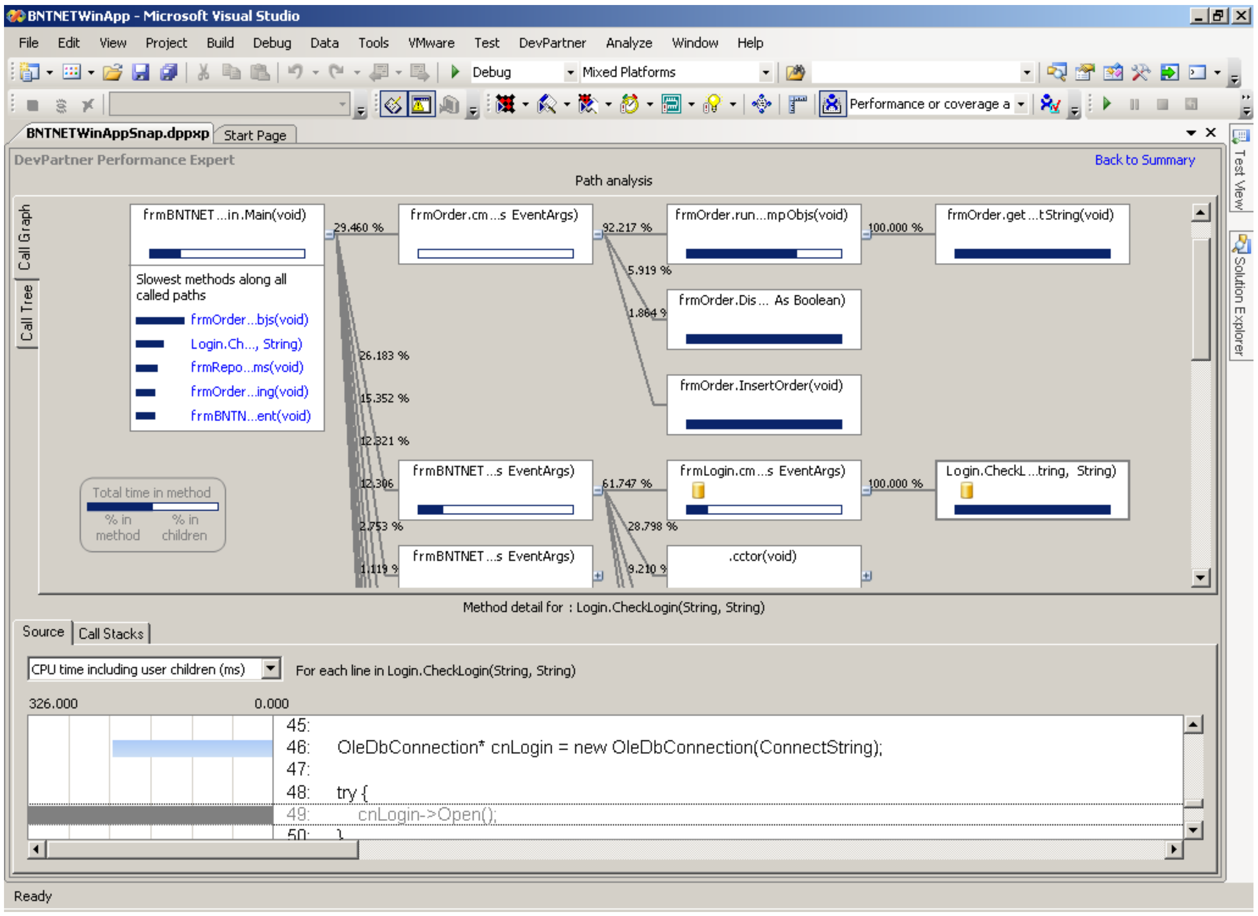
Figure 3.14 shows an even more detailed look, where performance is charted *per line of code.* This is an easy way to spot areas that should be targeted for improvement. I've worked on projects where the underlying problem turned out to be the database drivers we were using; there was almost no way for us to break into the drivers themselves, but the type of report in Figure 3.14 let us see that we were spending an unusual amount of time *on a single line of code* that was executing a relatively basic database query – the fact that we couldn't break that line of code down any more led us to suspect something "under the hood" as the cause of the problem.



Count	% of Method	% with Children	Time (ms)	Methods	Source
36	4.953	1.046	137.718	121	rs3 = cm.Execute()
36	0.000	0.000	0.008	0	z = 0
216	0.135	0.028	3.745	15	While Not rs3.EOF
180	0.176	0.037	4.896	2	strItem = strSub & "_" & CStr(z)
180	4.461	0.942	124.051	36	nodX = tvwItemList.Nodes.Add(strSul
180	0.192	0.041	5.344	11	nodX.Image = intCategoryNo
180	65.199	13.769	1,812.887	7	nodX.Selected = intCategoryNo
180	0.070	0.015	1.959	5	nodX.Sorted = False
180	0.189	0.040	5.248	8	rs3.MoveNext()
180	0.001	0.000	0.029	0	z = z + 1
180	0.202	0.043	5.625	14	If Not rs3.EOF Then rs3.MoveNext()
180	0.001	0.000	0.021	0	End While
72	0.000	0.000	0.010	0	End If
72	0.053	0.011	1.469	8	rs2.MoveNext()
72	0.000	0.000	0.008	0	v = v + 1

Figure 3.14: Analyzing performance for individual lines of code.

Figure 3.15 shows another way of looking at performance: by code path. This lets you trace your application's execution paths through various modules and calls and see the performance hit each one contributes to the application. In a highly-modularized application (and many applications written using good object-oriented techniques *are* highly modularized), this type of display can be invaluable in tracking down poor-performing code that is called upon frequently—contributing an unusually larger negative impact to the overall performance situation.


BNTNETWinApp - Microsoft Visual Studio
File Edit View Project Build Debug Data Tools VMware Test DevPartner Analyze Window Help
Debug
Mixed Platforms
Performance or coverage a
BNTNETWinAppSnap.dppxp
Start Page
DevPartner Performance Expert
Back to Summary
Path analysis
Call Graph
Call Tree
frmBNTNET...in.Main(void)
Slowest methods along all called paths
frmOrder...bjs(void)
Login.Ch..., String)
frmRepo...ms(void)
frmOrder...ing(void)
frmBNTN...ent(void)
Total time in method
% in method
% in children
29.460 %
26.183 %
15.352 %
12.321 %
12.306
2.753 %
1.119 %
frmOrder.cm...s EventArgs)
frmBNTNET...s EventArgs)
frmBNTNET...s EventArgs)
92.217 %
5.919 %
1.864 %
61.747 %
28.798 %
9.210 %
frmOrder.run...mpObjs(void)
frmOrder.Dis... As Boolean)
frmOrder.InsertOrder(void)
frmLogin.cm...s EventArgs)
.cctor(void)
100.000 %
100.000 %
frmOrder.get...tString(void)
Login.CheckL...tring, String)
Method detail for : Login.CheckLogin(String, String)
Source
Call Stacks
CPU time including user children (ms)
For each line in Login.CheckLogin(String, String)
326.000
0.000
Test View
Solution Explorer
Ready

```

    OleDbConnection* cnLogin = new OleDbConnection(ConnectString);

    try {
        cnLogin->Open();
    }
```

Figure 3.15: Analyzing performance by code paths.

So do you *need* these types of third-party performance tools? Only if you're struggling with performance, want to do something about it, and don't want to spend a lot of time going down dead ends in the search for a performance problem. The idea behind performance tuning is to spot the root cause of the problem quickly, and that's exactly what a good performance-analysis tool should help you do.

Reacting to Analytical Metrics and Guidance

With your analysis complete, what do you do with the results? Ideally, you start to take a look at your code with an eye toward risk and risk mitigation. You might also start looking at potential design changes to help reduce code complexity and increase maintainability. Code analysis is useless without a good follow-up reaction, so let's take the next few sections of this chapter to explore a list of action items.

Establish Code Priority

Can you imagine building a house by putting up the walls first, then pouring the foundation? It's kind of backwards. Builders know that everything depends upon the foundation, so they build that first—and inspect it, fix it if necessary, and so forth. Code analysis can help you do the same thing with your code: Components that support many dependencies should be created and unit-tested as thoroughly as practical *first* so that you can get them absolutely correct before building other components on top of their foundation.

Note

Development methodologies such as Agile, which focus on short time boxes, are ideal for establishing code priority. You can have a complete development cycle of just a few weeks that focuses on various foundation components, then begin a new cycle that builds on top of those. This setup gives a complete opportunity for design, testing, and so forth to those critical foundation components.

Of course, you can establish priority in this fashion only if code metrics are known to you in advance of actually *writing* the code—meaning you would have to be aware of these interdependencies during the design phase of your project. Once you're actually looking at live metrics from code that is being, or has been, written, can those metrics in any way shape your priorities?

Of course. Riskier modules may be scheduled for earlier testing or you might realign other project priorities—such as developing test data—so that riskier modules can move to the "front of the line" for testing. You may rearrange development priorities, perhaps opting to build "test harness" modules so that riskier modules can be tested more thoroughly and earlier, or you may reschedule development so that everything depending upon a particularly risky module gets developed and tested earlier.

You might schedule additional code reviews of riskier modules as well, and potentially have multiple peer reviews with an eye toward reducing complexity and catching bugs earlier. Just *knowing* that a particular module of code is riskier gives you a tremendous array of options for helping to mitigate that risk.

Establish Metrics Guidelines

Another thing you might consider is giving developers goals for their metrics. You want to avoid arbitrary numbers, here, though. Simply stating that "no element may have an inheritance depth of more than 6" is kind of arbitrary and might not actually bring you any business value; it's easy to find intrinsic elements of the Framework itself with deeper inheritance than that. You might, however, state that, "elements should have an inheritance depth of less than 3 *on top of* intrinsic elements, and any element with deeper inheritance will be subject to additional reviews."

Note
The "3" in my example is actually completely arbitrary; it is used for the purpose of illustration only, and not a best practice or recommendation.

The idea is to help developers (and designers) make smarter decisions when it comes to mitigating risk. By helping to manage complex interdependencies between code, you can help reduce the number of potential code defects and increase the code quality.

Note
This discussion is not in any way meant to suggest that code interdependency is a bad thing; on the contrary, it's the very basis of object-oriented programming and can save tremendous amounts of time. The point here is that you want to *manage* interdependencies. You want to recognize that high levels of dependency create more complexity, which leads to more difficult debugging and the risk of ripple effects from low-level changes. You may want, from a business perspective, to establish guidelines that communicate the amount of interdependency risks you're willing to tolerate, and use those guidelines to drive "flatter," rather than "deeper" designs.

A smarter and more complex guideline might further analyze dependencies such as inheritance and assign "deeper" code to more experienced developers. That way, those components upon which many other components rely will have been created by someone with more experience, who can hopefully produce code that can be safely depended upon and will need to change less frequently.

Address and Monitor Troublesome Metrics

When you're looking at a code metrics report with red icons—or low levels of maintainability—that's something you can address. Can the code be broken into several components, each of which is less complex individually? Do you simply need to assign more experienced developers to those more-complex components? Should you plan for more time in unit testing and debugging?

Metrics are a way of identifying *risky* areas of code—not necessarily *bad* sections of code. Complex projects may have components that are inherently less easy to maintain; that's fine, so long as you accept, from a business perspective, that risk and put plans in place to mitigate it. Some ideas for general directions in dealing with troublesome metrics:

- Perform additional code reviews to determine whether the complexity can be reduced
- Review the design to determine whether the number of interdependencies can be reduced without compromising the project timeline or costs
- Reassign developers based on their experience in handling complex code so that more experienced developers are handling or overseeing the more complex components

- Plan time for additional testing and debugging for complex code
- Review complex code against coding standards—in some cases, a lack of standards compliance is what leads to more complex code
- Review coding standards to make sure you're not unintentionally creating more complex code through well-meaning but misguided coding conventions

The bottom line is that you need to do *something*, from a management perspective, to deal with code that has above-average complexity.

Test Suite Development Guidance

Your code analysis should absolutely help drive your test suite plans—both for unit testing and integration testing. Here are just a few ways you might go:

- Schedule riskier, more complex modules for earlier testing—potentially even incomplete integration testing, if possible, just to get a head start on bugs.
- Develop test data for riskier modules first so that those modules can be tested earlier and more thoroughly.
- Use cyclomatic complexity numbers to ensure that unit and integration tests are thoroughly testing the software. You might even assign identifiers to each code path within a module so that test plans can specifically test each identified path—this is one way to be sure that you're testing *everything.*
- When changes are made to underlying components, you can look at inheritance trees and other statistics so that you know what needs to be retested in an effort to spot any negative ripple effects.

Note

Visual Studio Team System's Code Metrics window does help display inheritance trees and dependency trees because it has to calculate those trees in order to determine class coupling and depth of inheritance metrics. Third-party add-ins can also produce more detailed dependency reports, which can help identify the chain of modules that need to be tested when a heavily depended-upon module changes.

Control of Scope Creep

Good code metrics can also help you respond to changing requirements—especially in Agile-type methodologies where change is embraced. If a changing requirement will require reprogramming in a critical, highly depended-upon module, you can use code metrics to quickly identify the situation and make the appropriate estimates for additional time and cost. Change requests for a less depended-upon module are always less costly to implement simply because there is less need for additional integration testing or re-testing. In some cases, a change request that involves a heavily depended-upon module may drive a design change so that dependencies can be broken up and the overall cost of the change reduced.

Analysis Complete

You've completed your code analysis and peer review. You know what areas of the code you need to prioritize in terms of testing and risk mitigation, so what's next?

What's next is addressing coding errors, the subject of our next chapter. In it, we'll cover taxonomy of coding errors, many of which can have profound security and performance impacts. We'll also look at ways of avoiding and addressing those errors, using both automated tools and manual effort, and we'll look at specific techniques to detect and correct errors.