8

Managed Code Analysis

In this chapter and the next, we describe the code analysis features included with Visual Studio Team System. These tools can quickly and easily inspect your code to find common mistakes, suggestions for improvements, and even violations of standards.

In this chapter, we'll focus on analyzing managed code, such as C# and VB.NET. If you're working with C or unmanaged C++, turn to Chapter 9, where you will learn how to analyze native code.

We begin by discussing the origins of the Static Code Analysis tool. We'll introduce Microsoft's .NET Design Guidelines for Class Library Developers and describe how it is related to the tools.

Then we will describe the tool itself and how to take advantage of its full integration with Team System. This includes enabling Static Code Analysis review for your projects, selecting rules to apply, and working with the results of the analysis.

However, using the IDE is not always an option and sometimes you need additional flexibility. The Static Code Analysis tool is available to you from the command line. You will also learn how to use the command line for code analysis and how to include code analysis with your automated builds.

The Static Code Analysis rules that ship with Team System will probably not be sufficient for the specific standards and practices of your own projects. To address this, we will also describe how you can create and integrate new custom rules. We begin by describing the mechanics of rules, introducing the new Introspection engine. You will then create an example rule using introspection and call it from the Visual Studio IDE.

The Need for Analysis Tools

Ensuring that developers follow best practices and write consistent code is a major challenge in today's software development projects. The act of documenting standards and practices is often skipped or overlooked. However, even in projects for which standards have been established, getting developers to read and follow those practices is another major challenge.

One of the best resources available for .NET developers is Microsoft's .NET Framework "Design Guidelines for Class Library Developers." These guidelines document Microsoft's (formerly) internal practices for developing class libraries and are freely available at `http://msdn.microsoft.com/library/en-us/cpgenref/html/cpconnetframeworkdesignguidelines.asp`.

The guidelines cover a wide range of subjects, including naming conventions, usage guidelines, and performance and security considerations. The importance of these guidelines cannot be overstated. When put into practice, they help ensure that your approach will be consistent with that of other developers. In addition, they have evolved over a number of years to reflect a considerable amount of knowledge, best practices, and lessons learned.

As useful as the design guidelines are, the reality of software creation is that many developers, due to lack of time or perhaps of interest, will not be familiar with their contents. The desire to automate the process of evaluating code for compliance with these guidelines led to the creation of *FxCop*.

Using Managed Code Analysis

FxCop is a tool used to analyze managed code against a library of rules. You can create rules for almost any purpose: naming conventions, security, attribute usage, and so on. FxCop contains nearly 200 rules, based on the .NET Framework Design Guidelines described earlier.

FxCop has been available from Microsoft on the GotDotNet.com site for several years. Before then, it had been used internally at Microsoft for analysis of their own frameworks to help ensure predictable and consistent interfaces. Previous versions of FxCop have been stand-alone applications, separated from the Visual Studio IDE.

With Team System, FxCop is now called the Managed Code Analysis tool, and is fully integrated with the IDE, enabling analysis to be performed with a simple build of your application. The FxCop heritage of Managed Code Analysis is generally hidden when you're using the IDE, but as you'll see, the FxCop name still appears when creating new rules and using command-line options.

We'll follow an example project through this chapter. Create a new C# Class Library project and name it "SampleLibrary." Rename the `Class1.cs` file to `PayCalculator.cs` and insert the following code, which, as you'll soon see, has several problems:

```
using System;

namespace SampleLibrary
{
    public class PayCalculator
    {
        public enum Pay_Level
        {
            EntryLevel = 20,
            Normal = 35,
            Senior = 50
            }

            public static int MaximumHours;
            public const double BONUS = 0.10;

            static PayCalculator()
            {
                MaximumHours = 100;
            }

            public static double ComputePayment(int hours, Pay_Level level)
            {
                if (hours > MaximumHours)
                {
                    throw new ArgumentOutOfRangeException("Employee works too much");
                }

                return ((int)level * hours);
            }
        }
    }
```

While this code will compile and run as expected, you can make several improvements to it, and the Code Analysis tool will help you find them.

Built-in Managed Code Analysis rules

As mentioned, Team System ships with nearly 200 rules for Managed Code Analysis, each helping to enforce the practices documented in the .NET Framework Design Guidelines and other practices recommended by Microsoft. This section briefly describes each of the eleven rule groups, so you have an understanding of when you might apply them to your projects.

The groups of rules included with Team System are described in the following table.

Rule Group (# of Rules)	Description
Design (60)	Typically focused on the interfaces and structure of code. These enforce proper implementation of common concepts such as classes, events, collections, namespaces, and parameters.
Globalization (7)	Practices to support the internationalization of code. This can include avoiding strings of literal text, correct use of `CultureInfo`, and formatting.
Interoperability (16)	Focused on the correct use of COM Interop. Includes rules for proper use of `PInvoke`, the `ComVisible` attribute, and marshalling.
Maintainability (3)	Rules to help make your code easier to maintain. Identifies potential problems such as complexity and overuse of inheritance.
Mobility (2)	Rules to help detect code that will not run effectively in mobile or disconnected environments

Table continued on following page

Rule Group (# of Rules)	Description
Naming (20)	Enforces naming standards as described in the Design Guidelines. Using these rules verifies that names of items such as assemblies, classes, members, and variables conform to standards. Some rules will even help to detect misspellings in your assigned names.
Performance (19)	These rules help to detect places in your code that may be optimized for performance. They detect a wide variety of wasteful or extraneous code.
Portability (2)	Rules to find code that might not be easily portable between operating environments.
Reliability (5)	The rules in this group will help to detect problems with your code that may lead to intermittent failures, including failure to dispose of objects, improper use of the garbage collector, bad threading use, and more. These rules can be extremely useful because intermittent errors are frequently the most difficult to identify and correct.
Security (24)	These rules help to identify insufficient or incorrect security practices. Rules exist to find missing attributes, improper use of permissions, and opportunities of SQL injection attacks.
Usage (40)	These rules cover a broad spectrum of recommended practices. Whereas the design group rules typically involve API structure, these rules govern the methodologies of code. Practices include proper exception management, handling of arithmetic overflow, serialization, and inheritance.

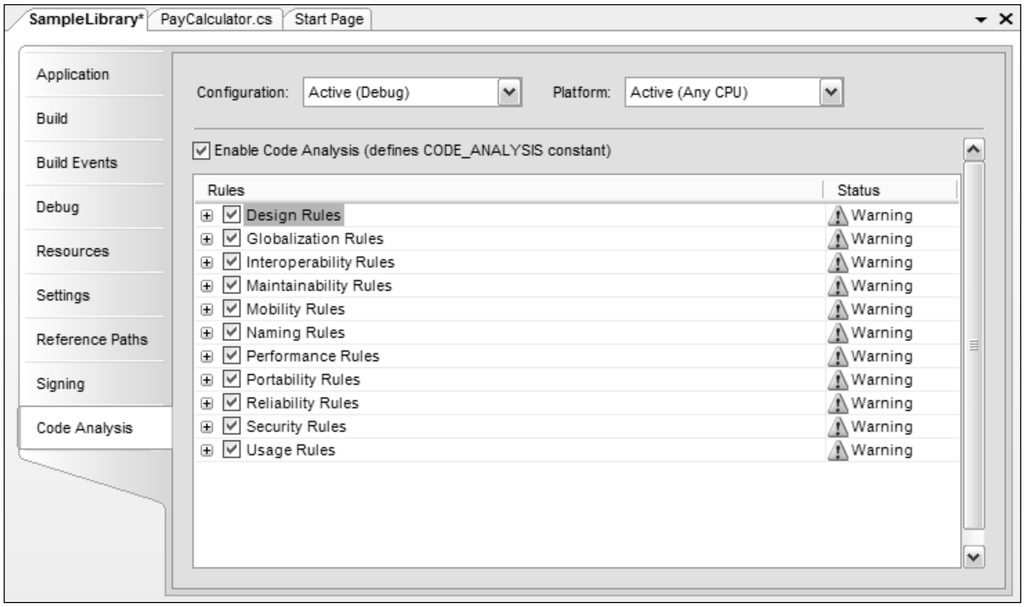
Of course, the rules that ship with Team System are only a starting point. Microsoft and others will certainly make additional rules available, and you can add your own custom rules and rule groups as well. You'll learn how to create custom Managed Code Analysis rules later in this chapter.

Enabling Managed Code Analysis

By default, code analysis is disabled for projects in Visual Studio. To enable analysis, open your project's Properties window and select Code Analysis from the left-hand side tabs. You will then see a collapsed list of rules, as shown in Figure 8-1.

To enable and configure Code Analysis for ASP.NET applications, select Website ➪ Code Analysis Configuration. Code Analysis may also be enabled, but not configured, from the Build page of the ASP.NET project's Property Pages.

Check the box labeled Enable Code Analysis. You can also expand the Rules groups to see specific rules. Rules or entire groups of rules can be disabled by unchecking their boxes. Save your settings via Save Selected Items on the File menu or by pressing Control+S.



SampleLibrary* | PayCalculator.cs | Start Page

Application
Build
Build Events
Debug
Resources
Settings
Reference Paths
Signing
Code Analysis

Configuration: Active (Debug)   Platform: Active (Any CPU)

☑ Enable Code Analysis (defines CODE_ANALYSIS constant)

Rules	Status
☑ Design Rules	Warning
☑ Globalization Rules	Warning
☑ Interoperability Rules	Warning
☑ Maintainability Rules	Warning
☑ Mobility Rules	Warning
☑ Naming Rules	Warning
☑ Performance Rules	Warning
☑ Portability Rules	Warning
☑ Reliability Rules	Warning
☑ Security Rules	Warning
☑ Usage Rules	Warning

Figure 8-1

In addition, each rule can be set to either Warning (the default) or Error. Warnings serve as an advisory that something may need to be corrected, but they will not prevent the project's build from succeeding. You may want to set certain rules or groups of rules to Error if they are critically important, thus preventing a build when those rules are violated. Double-click on the entry under the Status column to toggle the value between Warning and Error. As with enabling rules, this can be done for specific rules or entire groups of rules.

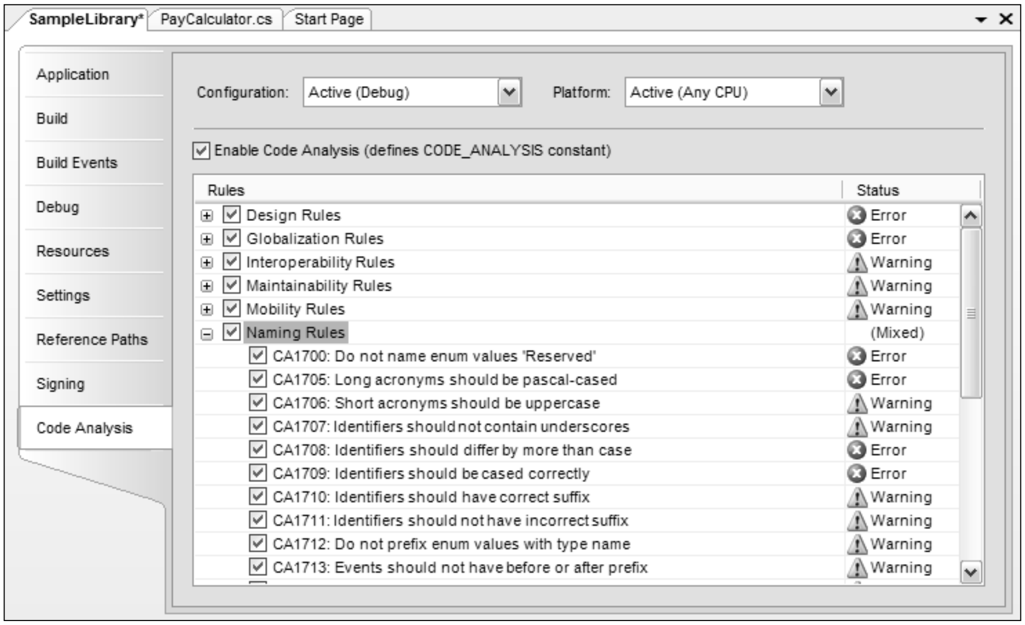
Figure 8-2 illustrates how to enable and disable specific rules and how each can be set to Warning or Error as necessary.

Finally, you can specify different sets of code analysis properties for each configuration. By default, settings apply to the Active build configuration, but you can be more specific. For example, you may wish to treat certain critical rules as errors in your Release builds, but as warnings in Debug. You might instead decide to disable code analysis entirely for your Release builds. Simply choose a build type from the Configuration drop-down menu and then review your settings. To make changes affecting all build configurations, select the All Configurations option, and then modify and save your settings.

Executing static code analysis

Once you have enabled code analysis and configured the rules to reflect your development standards, code analysis will be performed each time you build your project. Go ahead and build your sample project now.

You can also execute code analysis on your project by choosing Build ➪ Run Code Analysis on [Project Name].



Configuration: Active (Debug)   Platform: Active (Any CPU)

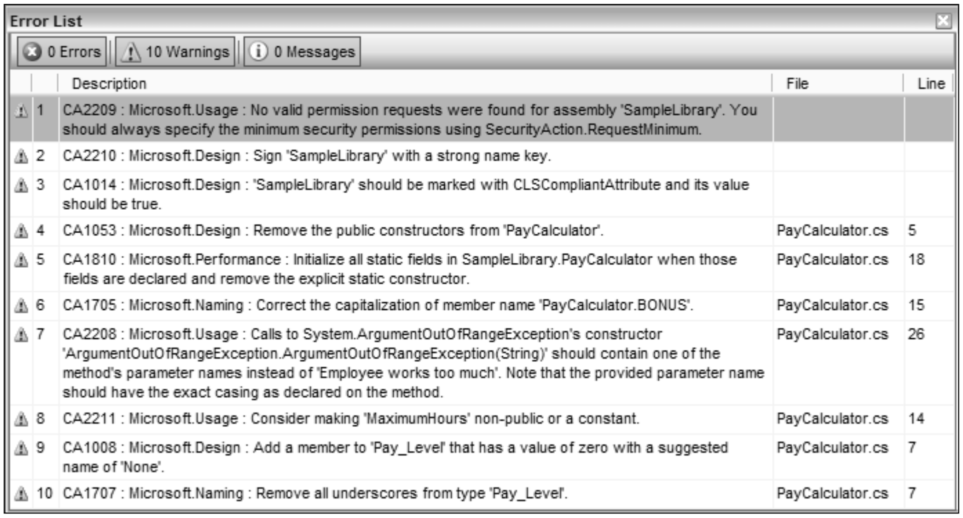
☑ Enable Code Analysis (defines CODE_ANALYSIS constant)

Rules	Status
⊞ ☑ Design Rules	Error
⊞ ☑ Globalization Rules	Error
⊞ ☑ Interoperability Rules	Warning
⊞ ☑ Maintainability Rules	Warning
⊞ ☑ Mobility Rules	Warning
⊟ ☑ Naming Rules	(Mixed)
☑ CA1700: Do not name enum values 'Reserved'	Error
☑ CA1705: Long acronyms should be pascal-cased	Error
☑ CA1706: Short acronyms should be uppercase	Warning
☑ CA1707: Identifiers should not contain underscores	Warning
☑ CA1708: Identifiers should differ by more than case	Error
☑ CA1709: Identifiers should be cased correctly	Error
☑ CA1710: Identifiers should have correct suffix	Warning
☑ CA1711: Identifiers should not have incorrect suffix	Warning
☑ CA1712: Do not prefix enum values with type name	Warning
☑ CA1713: Events should not have before or after prefix	Warning

Figure 8-2

The output window will include details about your build, including results from calling code analysis. After the build, the Error List window may appear, displaying a number of warnings and possibly some errors. The Error List does not automatically open if there are only warnings. If you do not see the Error List, choose View ➪ Error List.

Figure 8-3 is a screenshot of the Error List, displaying code analysis results for the SampleLibrary assembly.



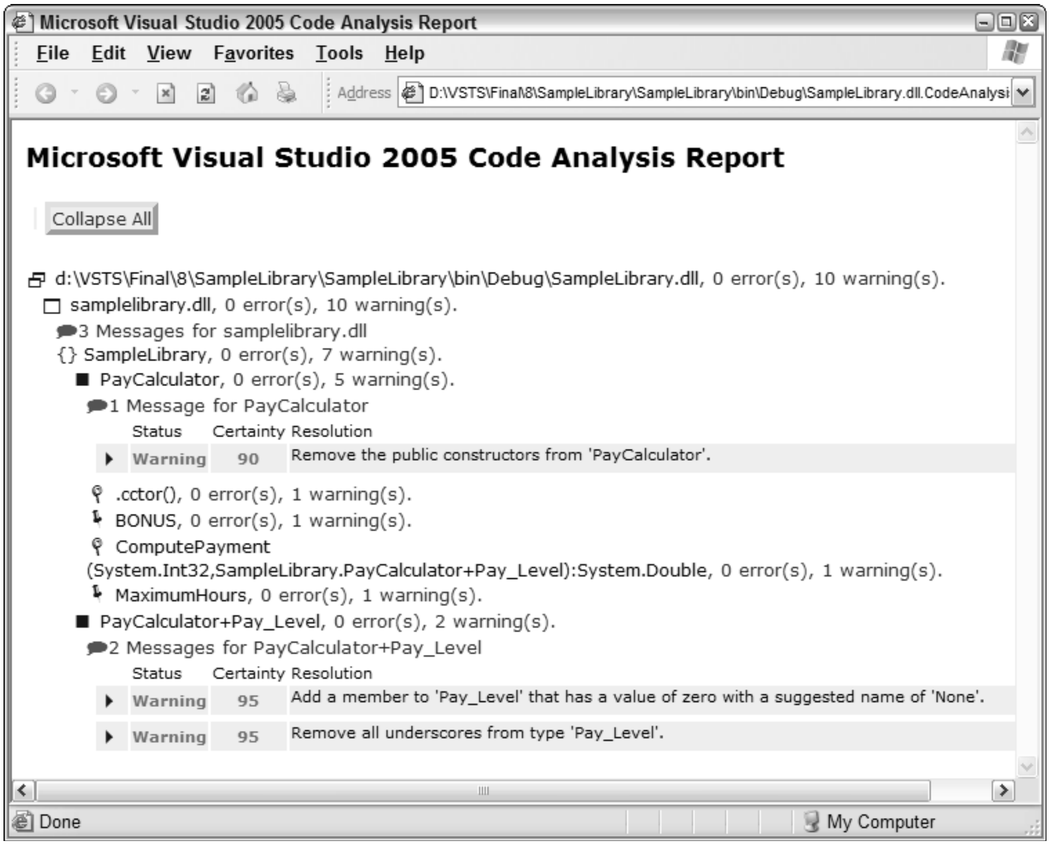
	Description	File	Line
1	CA2209 : Microsoft.Usage : No valid permission requests were found for assembly 'SampleLibrary'. You should always specify the minimum security permissions using SecurityAction.RequestMinimum.		
2	CA2210 : Microsoft.Design : Sign 'SampleLibrary' with a strong name key.		
3	CA1014 : Microsoft.Design : 'SampleLibrary' should be marked with CLSCompliantAttribute and its value should be true.		
4	CA1053 : Microsoft.Design : Remove the public constructors from 'PayCalculator'.	PayCalculator.cs	5
5	CA1810 : Microsoft.Performance : Initialize all static fields in SampleLibrary.PayCalculator when those fields are declared and remove the explicit static constructor.	PayCalculator.cs	18
6	CA1705 : Microsoft.Naming : Correct the capitalization of member name 'PayCalculator.BONUS'.	PayCalculator.cs	15
7	CA2208 : Microsoft.Usage : Calls to System.ArgumentOutOfRangeException's constructor 'ArgumentOutOfRangeException.ArgumentOutOfRangeException(String)' should contain one of the method's parameter names instead of 'Employee works too much'. Note that the provided parameter name should have the exact casing as declared on the method.	PayCalculator.cs	26
8	CA2211 : Microsoft.Usage : Consider making 'MaximumHours' non-public or a constant.	PayCalculator.cs	14
9	CA1008 : Microsoft.Design : Add a member to 'Pay_Level' that has a value of zero with a suggested name of 'None'.	PayCalculator.cs	7
10	CA1707 : Microsoft.Naming : Remove all underscores from type 'Pay_Level'.	PayCalculator.cs	7

Figure 8-3

Analysis of the SampleLibrary code indicates 10 potential rule violations. Each item in the list has a full description indicating how your code is in violation of a rule. The Error List has File and Line columns that indicate, when appropriate, specific source files and code related to each warning. Some warnings do not relate to specific code, but perhaps to a lack of an attribute or security setting. In such cases, there will be no value in the File column. Others may refer directly to problem code, perhaps naming violations or performance issues. You can double-click on the warning and the code editor will switch to the related code.

Each time you run code analysis, the results are stored in an XML file. This file is named `<Project Name>.CodeAnalysisLog.xml` and is located in your project's `\bin\Debug or \bin\Release` directory, depending on the current build configuration. For the SampleLibrary project, the file will be `SampleLibrary.dll.CodeAnalysisLog.xml`.

If you open the file from within the IDE, you will see the raw, unformatted XML. However, the XML has an associated XSL template that formats the data into HTML, similar to what is shown in Figure 8-4. To see this view, open the XML file with Internet Explorer. To customize rendering, you can supply your own XSL templates. We recommend you make a copy of the included template and modify the copy to suit your needs. The base template is found in your Visual Studio installation directory as `\Team Tools\Static Analysis Tools\FxCop\Xml\CodeAnalysisReport.xsl`.


Microsoft Visual Studio 2005 Code Analysis Report
File
Edit
View
Favorites
Tools
Help
Address
D:\VSTS\Final\8\SampleLibrary\SampleLibrary\bin\Debug\SampleLibrary.dll.CodeAnalysi
Collapse All
Done
My Computer

Microsoft Visual Studio 2005 Code Analysis Report

d:\VSTS\Final\8\SampleLibrary\SampleLibrary\bin\Debug\SampleLibrary.dll, 0 error(s), 10 warning(s).
samplelibrary.dll, 0 error(s), 10 warning(s).
3 Messages for samplelibrary.dll
{} SampleLibrary, 0 error(s), 7 warning(s).
PayCalculator, 0 error(s), 5 warning(s).
1 Message for PayCalculator

Status	Certainty	Resolution
Warning	90	Remove the public constructors from 'PayCalculator'.

.cctor(), 0 error(s), 1 warning(s).
BONUS, 0 error(s), 1 warning(s).
ComputePayment (System.Int32,SampleLibrary.PayCalculator+Pay_Level):System.Double, 0 error(s), 1 warning(s).
MaximumHours, 0 error(s), 1 warning(s).
PayCalculator+Pay_Level, 0 error(s), 2 warning(s).
2 Messages for PayCalculator+Pay_Level

Status	Certainty	Resolution
Warning	95	Add a member to 'Pay_Level' that has a value of zero with a suggested name of 'None'.
Warning	95	Remove all underscores from type 'Pay_Level'.

Figure 8-4

Working with rule violations

Several issues should be addressed in the sample `PayCalculator` class. For each warning or error, you need to determine whether the rule actually applies to your project or a specific section of code. If it does, you need to modify the project to address the issue; otherwise, you may choose to ignore the rule. In this section, we'll describe how to act on identified issues and how to ignore, or suppress, a given rule.

We'll immediately go into the code and make corrections as necessary, but your organization or project may require the use of work items to track any changes. Or, alternatively, perhaps you don't have time to immediately address an identified problem but would like to use a work item as a reminder. Fortunately, you can easily create work items directly from Code Analysis rule violations. Simply right-click on the warning or error and choose Create Work Item from the menu. Choose the correct Team Project and you will be shown the New Work Item dialog. Make any necessary changes and save your new work item.

Correcting problems

Looking through the Error List shown in Figure 8-3, you should see item CA1810, with a description of "Initialize all static fields in SampleLibrary.PayCalculator when those fields are declared and remove the explicit static constructor."

Right-click on this warning and choose Show Error Help. This will display the documentation for the rule that triggered this warning, including suggestions for resolving the issue. We are currently assigning the value of 100 to `MaximumHours` inside the static constructor of `PayCalculator`. The rule's Help text states that our code may perform more efficiently if we make that assignment when the variable is defined.

To address this issue, double-click on this warning and you'll be brought to the static constructor of the `PayCalculator` class. Change the code to assign the value in the declaration as follows:

```
    public static int MaximumHours = 100;
```

Next, delete the static `PayCalculator` constructor entirely. Build the project and look at the Error List window. The specific warning should no longer be in the list. Your code is better already!

There is another easy problem to correct. Many of the code analysis rules relate to standard naming conventions. Find the warning "Remove all underscores from type Pay_Level" and double-click. The rule helps to enforce the naming convention that underscores should not be used in type names. Use the built-in refactoring support to rename it (see Chapter 10 for details). Right-click on the `Pay_Level` enumeration and choose Refactor ➪ Rename. Change the name to `PayLevel` and apply.

Another warning says "Remove the public constructors from PayCalculator." There aren't any constructors in the code, so why is this violation included? If you don't specify a constructor, a parameterless public constructor is compiled into your code automatically. The `PayCalculator` class has only static members, so creating an instance is not going to be useful. This rule lets you know that you should specifically prevent this by adding a private constructor. Add the following code to `PayCalculator`:

```
    private PayCalculator() { }
```

Once you make this change and rebuild, a new warning will appear in the Error List window: "Mark PayCalculator as sealed." This indicates that because all members of the class are static and there are no public constructors, you can improve runtime efficiency by sealing the class. Because a sealed class cannot be inherited from, .NET can make some assumptions about the class, which optimizes performance. Mark the `PayCaclulator` class definition sealed as follows:

```
public sealed class PayCalculator
```

In C# 2.0, you can go one step further. Instead of using `sealed`*, you could use the new* `static` *keyword to mark the entire class as static. This can be applied only to classes that have only static members and no instance constructors, public or private. In our example, you would need to remove the private instance constructor in order to use the* `static` *keyword.*

Rules can also help ensure that you're using the Framework correctly. You can see from the warning "Calls to System.ArgumentOutOfRangeException's constructor... should contain one of the method's parameter names..." that the rule has detected that you might not be creating the `ArgumentOutOfRangeException` correctly. To fix this, change the line that throws the exception to the following:

```
        if (hours > MaximumHours)
        {
            throw new ArgumentOutOfRangeException("hours", "Employee works too much");
        }
```

One of the remaining warnings, "SampleLibrary should be marked with CLSCompliantAttribute and its value should be true," is a fairly common suggestion. Consider addressing this when creating a reusable library assembly that might be consumed by code of more than one .NET language. CLS compliance specifies that your assembly must meet the common structure and syntax supported by all .NET languages as defined in the Common Language Specification (CLS). Keep in mind that there may be times when CLS compliance is not possible, such as when exposing unsigned types.

To address this warning, open `AssemblyInfo.cs` and add the following line:

```
[assembly: System.CLSCompliant(true)]
```

The `assembly:` notation is used because the attribute applies to the entire assembly, and not to a specific class or member. Other assembly-level attributes can be found in the `AssemblyInfo.cs` file.

Another change to the `AssemblyInfo.cs` file will satisfy the "No valid permission requests were found for assembly SampleLibrary" warning. Best practices for security indicate that assemblies should always specify the permissions they require. The `SampleLibrary` does not require any special permissions, so you'll add an attribute to indicate this. You'll need the `System.Security.Permissions` namespace, so add the `using` statement to `AssemblyInfo.cs`:

```
using System.Security.Permissions;
```

Then add the attribute:

```
[assembly: SecurityPermission(SecurityAction.RequestMinimum, UnmanagedCode=false)]
```

Now, build the project. The violations you corrected should no longer generate messages in the Error List. We'll address some of the other warnings in the next section.

Suppressing messages

Team System ships with many rules, but not all of them are appropriate for every project. There is a chance that some rules will trigger warnings that simply don't apply to certain parts of your project. To prevent these irrelevant messages from recurring, right-click on the rule violation and choose Suppress Message(s).

When you suppress a message, Visual Studio automatically adds an attribute to your code to indicate that a rule should not apply. The `SuppressMessage` attribute can be applied to a code construct, such as a field, method, or class, and to an entire assembly.

Suppressing a message is not the same as disabling a rule. Suppression prevents the specific violation of a rule from recurring, but other violations of the same rule will still be identified. You should disable a rule only if you're certain it could never be meaningfully applied to any part of your project.

Let's continue with the SampleLibrary example and use message suppression to clean up more of the code analysis violation messages.

One of the warnings states "Correct the capitalization of member name PayCalculator.BONUS." Assume that your organization has different naming conventions for constants and you know that this rule will not apply to this `BONUS` constant. Right-click on the message and choose Suppress Message. The message will be crossed out in the Error List and the `PayCalculator` class will be modified to include the following attribute immediately before the declaration of `BONUS`:

```
[System.Diagnostics.CodeAnalysis.SuppressMessage(
    "Microsoft.Naming",
    "CA1705:LongAcronymsShouldBePascalCased",
    MessageId = "Member")]
```

The next time Code Analysis is run, the engine will recognize this attribute. Moreover, even when the CA1705 rule is violated at this point, no message will be created. Messages for any other violations of this rule elsewhere in the code will still be reported as normal.

In many cases, especially in ASP.NET applications, Visual Studio will automatically generate helper or wrapper code. Previous versions of FxCop had difficulty working with such generated code and often flagged many warnings — for example, naming convention violations, which would have to be investigated and generally excluded. Fortunately, the .NET Framework 2.0 offers a new attribute, `GeneratedCodeAttribute`*, that the Managed Code Analysis tool uses to identify code that it does not need to analyze.*

Two more messages don't apply to the project. "Consider making MaximumHours non-public or a constant" reminds us that external users of the class could change its value. This is the behavior you want, so right-click on the message and choose Suppress Message. The message "Add a member to PayLevel that has a value of zero with a suggested name of None" also does not apply, as all employees are required to have an employee level. Suppress this message as well.

As you can see, suppressing messages can quickly add a number of attributes to your code. If you find that you always suppress a given message, it is probably better to exclude the rule altogether; then your code will not require the additional `SuppressMessage` attributes. However, as noted above, use caution when doing this because you could unintentionally be missing valid violations that should be addressed.

A new warning appeared after we corrected the `ArgumentOutOfRangeException` issue earlier. Rule CA1303 warns that we should not embed literal strings in our code. The recommended approach is to use a resource table with all such strings and to load each value using `System.Resources .ResourceManager`. Too keep this example simple, we'll suppress this message as well.

The warning "Sign 'SampleLibrary' with a strong name key" applies to the overall assembly. If you know that you'll never use this assembly in the Global Assembly Cache and will have no other need for strong names, you can suppress this message. However, because there is no specific code to which the `SuppressMessage` attribute can be applied, a new file, `GlobalSuppressions.cs`, will be added to the project with the following code:

```
[assembly: System.Diagnostics.CodeAnalysis.SuppressMessage(
    "Microsoft.Design",
    "CA2210:AssembliesShouldHaveValidStrongNames")]
```

Build the project and you should now see an empty Error List. This indicates all enabled Code Analysis rules have either been passed or suppressed.

The effect of assembly-level suppression is basically the same as if you had excluded the rule altogether. The advantage of the attribute-based approach is that it is easy to see which rules have been suppressed project-wide by viewing the GlobalSuppressions.cs file. In addition, you could add comments to that file to indicate the reason for suppressing the rule to other developers. Excluding a rule by not selecting it in the Code Analysis section of the project's properties has the same effect, but does not offer a way to document why certain exclusions were made.

Using the Command-Line Analysis Tool

Like the versions of FxCop that preceded Team System, a command-line interface is available for static code analysis. This tool, called `FxCopCmd.exe`, can be found in your Visual Studio 2005 installation directory under `Team Tools\Static Analysis Tools\FxCop`.

`FxCopCmd` can perform any of the code analysis functions that are available to you in the Team System IDE. In fact, the IDE uses `FxCopCmd` under the covers to execute analysis and generate reports.

FxCopCmd options

The following are some of the options that `FxCopCmd.exe` supports:

Option	Description
/f[ile]: <directory/file>	Assembly file(s) or directory(ies) to analyze. If a directory is used without a filename, Code Analysis will try to analyze all files in that directory with `.dll` or `.exe` extensions. You can specify this option more than once. It is required unless you specify a project file with the `/project` option.
/r[ule]:<directory/file>	A rule assembly file or a directory to browse for rule assemblies. If a directory without a filename is supplied, Code Analysis will look for rules in any files with a `.dll` extension. You can specify this option more than once.
/r[ule]id:<[+|-] Category#CheckId>	Enables or disables a specific rule, supplying its `Category` and `CheckId` values — for example, /rid: +!Microsoft.Usage#CA2225.
/o[ut]:<file>	Names a file in which the results of the analysis will be stored in XML form. Required unless the `/console` option is used.
/p[roject]:<file>	Loads a project file that contains the settings for FxCopCmd to use (discussed in the next section). Required if you do not use both the `/file` and `/rules` options.
/t[ypes]:<type list>	Used to constrain analysis to only the specified type(s). Supply a list of comma-delimited type names. Wildcards can be used to specify multiple types. Optional.
/i[mport]:<directory/file>	Loads analysis reports or project files to exclude items from the current test that appear as excluded in the imported file. You may specify a file or a directory. If a directory is specified, Code Analysis will attempt to load all files with an `.xml` extension. Optional.
/s[ummary]	Displays a summary after analysis. Optional.
/v[erbose]	Gives more detailed status output. Optional.
/q[uiet]	Suppresses output of status details. Optional.
/u[pdate]	Saves the results of the current analysis to the specified project file. Ignored if you do not supply the `/project` option. Optional.
/c[onsole]	Uses the console to display the analysis results. This is required unless you have specified the `/out` option.
/c[onsole]xsl:<file>	Applies an XSL file to transform XML output before displaying.
/plat[form]:<directory>	Location of platform assemblies. Optional.
/d[irectory]: <directory>	Location to search for assembly dependencies. Optional.
/help (or) /?	Help about command-line options.

Notice that most of the commands have long and short forms available. For example `/summary` and `/s` are equivalent. Arguments support using wildcards (*) to specify multiple items. Arguments with spaces in them must be surrounded with double quotes.

For example, to conduct analysis of a single assembly `CustomLibrary.dll`, use the following command:

```
FxCopCmd /f:CustomLibrary.dll /o:"FxCop Results.xml" /s
```

The `/f` (or `/file`) argument indicates which assembly to analyze and the `/o` (or `/output`) option indicates that analysis output should be stored as XML in `FxCop Results.xml`. Finally, the `/s` (or `/summary`) option will display a short summary of the results of the analysis.

FxCopCmd project files

`FxCopCmd`'s command-line options offer a good deal of flexibility, but to fine-tune your analysis, you should consider using a *project file*. A project file enables you to set options such as targets and rule assemblies, exclusions, and output preferences. You can then simply use the `/project` option to tell `FxCopCmd` to use those settings instead of supplying a detailed list of arguments.

We recommend you create a default `FxCopCmd` project file that you can copy and customize for each project. Create a new file named `EmptyCodeAnalysisProject.fxcop` and enter the contents as follows:

```
<?xml version="1.0" encoding="UTF-8"?>
<FxCopProject Version="8" Name="Temporary FxCop Project">
        <ProjectOptions>
                <SharedProject>False</SharedProject>
                <CompressProjectFile DefaultTargetCheck="True" DefaultRuleCheck=
"True">True</CompressProjectFile>
                <PermitAnalysis>True</PermitAnalysis>
        </ProjectOptions>
        <Targets>
        <Target Name="$(TargetFile)" Analyze="True" AnalyzeAllChildren="True" />
        </Targets>
        <RuleFiles>
        </RuleFiles>
        <FxCopReport Version="8" LastAnalysis="2004-04-20 22:08:53Z">
        </FxCopReport>
</FxCopProject>
```

Copy this to a new file and add your project's settings. The rules and files specified in your project file serve as the basis for `FxCopCmd` execution. Additional rules and target files can be specified on the command line with the `/rules` and `/file` options.

For example, here is a simple project file that specifies a target assembly, `SampleLibrary.dll`, and includes one rule assembly, the default Code Analysis naming conventions assembly:

```
<?xml version="1.0" encoding="UTF-8"?>
<FxCopProject Version="8" Name="Sample Library Code Analysis Project">
        <ProjectOptions>
                <SharedProject>False</SharedProject>
                <CompressProjectFile DefaultTargetCheck="True" DefaultRuleCheck=
                 "True">True</CompressProjectFile>
                <PermitAnalysis>True</PermitAnalysis>
        </ProjectOptions>
        <Targets>
                <Target Name="C:\SampleLibrary\bin\Debug\SampleLibrary.dll"
Analyze="True"
                 AnalyzeAllChildren="True" />
        </Targets>
        <RuleFiles>
                <RuleFile Name="$(FxCopDir)\Rules\NamingRules.dll" Enabled="True"
                 AllRulesEnabled="True" />
        </RuleFiles>
        <FxCopReport Version="8" LastAnalysis="2004-04-20 22:08:53Z">
        </FxCopReport>
</FxCopProject>
```

Save the above to a file named `SampleLibrary.fxcop`. To execute Code Analysis for SampleLibrary using this project file, use the following command:

```
FxCopCmd /p:SampleLibrary.fxcop /o:"FxCop Results.xml" /s
```

Build process code analysis integration

You have now seen how to use `FxCopCmd` from the command line to analyze your code and report potential defects. However, with the full integration of code analysis with the Team System IDE, why would you need to use `FxCopCmd`?

One of the main reasons is for automated batch operations. A common use of `FxCopCmd` is to enable automated code analysis from a build process. You can do this with Team System's Team Build, Visual Studio 2005's MSBuild, or one of many other build automation packages available, such as NAnt.

Whichever package you use for your periodic builds, the same basic flow will apply:

1. Build the assemblies you wish to analyze (e.g., by invoking the C# or VB.Net compilers).
2. Call `FxCopCmd.exe`, specifying the `/f[ile]` option pointing to your freshly built assemblies.
3. If `FxCopCmd` outputs any notices of violations, configure your build to stop and display appropriate messages.
4. Review the reasons for the build failure.
5. If any of the rules are false positives that can be safely ignored, update your Code Analysis project file to exclude them. Save your project file and rebuild.
6. If there are any remaining warnings, make the appropriate code changes to comply with the rules.

With Team Build, you can easily enable managed code analysis. Open a build configuration by choosing Team ➪ Team Project Settings, and then click Build Configurations. Open Build Steps and check the Perform Static Analysis option. The next time your build runs, a post-build step will invoke the Code Analysis tool and include the results with the build report.

By integrating Code Analysis with your builds, you can ensure that your entire team's work is being evaluated against a consistent set of rules. You will quickly discover when a developer has added nonstandard code. Developers will quickly learn those rules and practices because they don't want to be the person responsible for "breaking the build."

Creating Code Analysis Rules

Team System includes many code analysis rules, but no matter how comprehensive the rules from Microsoft are, they can never fully cover the specific requirements of your own projects. Perhaps you have specific naming conventions or a standard way to load database connection strings. In many cases, you can create a custom code analysis rule to help diagnose the issue and help developers take corrective action.

Reflection and Introspection

Many static analysis tools use simple source-code inspection to identify issues. However, with FxCop, Microsoft decided to leverage the inherent functionality of .NET itself as the basis for creating rules. A very useful feature of .NET is called *reflection*. Using reflection, you can programmatically inspect other assemblies, classes, and members. You can even invoke methods or access fields, public or private, given appropriate security settings. Reflection is done without establishing a link to the target assembly at compilation time, a practice known as *late binding*.

Initial versions of FxCop relied on reflection as the basis for rules. However, a newer option is available, called *introspection*. Similar to reflection, introspection can inspect a target assembly to discover its types, and details about those types. It can also invoke members of those types. Introspection does this in a much faster manner than reflection and supports multi-threaded operations. Furthermore, introspection does not lock the files under analysis, a problem suffered by previous versions of FxCop that needed to use reflection. Given the clear advantages of introspection over reflection, Microsoft has leveraged introspection with the rules that are shipped with Team System. We'll also use introspection in this section for our custom rule.

Creating a new rule

Creating a new rule can be challenging, so we will walk through the creation of one in this section. We'll continue working with the SampleLibrary created earlier in this chapter. You'll recall that when you ran code analysis on the SampleLibrary, a number of potential issues were flagged. There is actually another problem with the code that is not detected by the set of rules included with Team System.

In this section, we'll create a fairly simple rule to help correct a potentially serious issue. Exposing constant values from an assembly is a normal and expected practice, but with .NET there is a surprising side effect. When a second assembly references a source assembly that exposes a constant value, the value of that constant is actually stored directly in the IL of the referencing assembly. This means that even when you change and recompile the original assembly, the value is not changed in the referencing assembly. This can lead to extremely difficult-to-diagnose problems, and will require you to recompile all referencing assemblies even though those assemblies have not changed.

To address this, we'll create a new rule, `AvoidExposingPublicConstants`, which searches a target assembly for publicly visible constant values. Begin by creating a new C# Class Library project named "CustomCodeAnalysisRules."

Code Analysis loads designated assemblies and searches them for rule classes. Code Analysis rules implement a core interface called `IRule`. Rules that use introspection also implement `IIntrospectionRule`. However, these and other related interfaces and classes are wrapped by the helpful `BaseIntrospectionRule` class. We'll use this class as the basis for our own rules. To use this base class, add a reference in the `CustomCodeAnalysisRules` project to the `FxCopSdk.dll` and `Microsoft.Cci.dll` assemblies found in the `\Team Tools\Static Analysis Tools\FxCop` directory.

Creating a base rule

As mentioned before, most of the included code analysis rules inherit, typically indirectly, from a base class called `BaseIntrospectionRule`. While each custom rule could inherit directly from this class, it's easier to create a common base class that inherits from `BaseIntrospectionRule`. This is because the constructor to `BaseIntrospectionRule` requires three arguments. The first argument is the name of the rule, and the second is the name of the XML file containing rule data. The final argument is a reference to the rule assembly type.

If you created a rule assembly with multiple rules, each rule would have to supply those three arguments each time. However, with a new base class, you can abstract away the last two arguments and keep your rule code streamlined.

Create a new file called `BaseStaticAnalysisRule.cs` and add the following code:

```
using System;
using Microsoft.FxCop.Sdk.Introspection;

namespace CustomCodeAnalysisRules
{
    public abstract class BaseStaticAnalysisRule : BaseIntrospectionRule
    {
        protected BaseStaticAnalysisRule(string name) :
            base(name,
                 "CustomCodeAnalysisRules.Rules",
                 typeof(BaseStaticAnalysisRule).Assembly ) { }
    }
}
```

Because the values of the second and third parameter to the `BaseIntrospectionRule` constructor will be the same for all rules in your assembly, you use this simple class as a wrapper in which those values can be set. The second argument, `CustomCodeAnalysisRules.Rules`, needs further explanation and is described in the section "Creating Rules. XML."

Implementing the rule

Now that you have a base class to use for all of your custom rules, you can create a rule. A rule has two main components:

- **Rule implementation code:** This is the code that analyzes the target assembly and determines whether the standard or guideline it is trying to enforce has been violated.
- **Rule descriptive XML:** Having the implementation code is not enough. An embedded XML fragment is required in order to help Managed Code Analysis display the rule and provide details such as descriptions and resolutions to the user.

Before you can create the rule's implementation, you need to have an approach for evaluating and inspecting a target assembly. While there are many ways you could write code to do this, Microsoft has made the job much easier by including the `Microsoft.Cci` assembly with Team System. You'll learn what this assembly is and how to use it in the following section.

Using the Microsoft.Cci assembly

The `Microsoft.Cci` assembly, or Common Compiler Infrastructure, originated from Microsoft Research and contains classes that provide features for language-based tools, such as compilers. This assembly is especially helpful for code analysis because it offers many classes that map directly to common programming constructs such as classes, fields, members, and methods. You'll use these classes to inspect target assemblies to identify the places where your rule applies.

You may be familiar with the `System.CodeDom` *namespace. It is very useful for creating intermediate representations of programming constructs and then using a language provider to generate code in a desired language, such as VB.NET.* `Microsoft.Cci`*, conversely, offers additional features for reading and inspecting existing code, exactly the task we face when creating a code analysis rule.*

The following table lists the major classes offered by `Microsoft.Cci`, organized by programming concept.

Programming Concept	**Related** `Microsoft.Cci` **Classes**
Assembly	`CompilationUnit`
Namespace	`Namespace`
Types	`Class`, `Struct`, `Interface`
Type Member	`Member`
Member	`Method`, `Field`, `Property`, `Event`, `EnumNode`
Method	`Method`, `InstanceInitializer`, `StaticInitializer`
Statement	`Block`, `AssignmentStatement`, `If`, `For`, `ForEach`, `DoWhile`, `While`, `Continue`, `ExpressionStatement`, `VariableDeclaration`, `Return`, `Switch`, `Lock`
Expression	`Variable`, `AssignmentExpression`, `UnaryExpression`, `BinaryExpression`, `NaryExpression`, `Literal`, `Parameter`, `Local`
Exception-Related	`ExceptionHandler`, `Throw`, `Try`, `Catch`, `Finally`
Instructions and Operations	`Instruction`, `OpCode`

The members of the `Microsoft.Cci` namespace are organized in a hierarchical structure, with related classes organized under a parent type. For example, the `Member` class is the parent for the types of things you'd expect to have as class members, `Method`, `Property`, `Field`, `Event`, and others. If you have a `Method` instance, you can use its members to obtain references to the items it contains. For example, the `Instructions` property returns an `InstructionList` that you can use to loop through the operations of the method. Similarly, the `Method` class also has a `Pamameters` field, returning a `ParameterList` instance that can be used to inspect each parameter to the method.

To use the `Microsoft.Cci` assembly, add a reference to the `Microsoft.Cci.dll` assembly in the `\Team Tools\Static Code Analysis\FxCop` directory.

The IIntrospectionRule interface

As mentioned earlier, one of the abstractions our base class makes is the implementation of the `IIntrospectionRule` interface. This interface gives you a chance to specify the conditions under which you want your rule to be invoked. `IIntrospectionRule` contains the following members:

```
ProblemCollection Check(Member member);
ProblemCollection Check(Module module);
ProblemCollection Check(Parameter parameter);
ProblemCollection Check(Resource resource);
ProblemCollection Check(TypeNode type);
ProblemCollection Check(string namespaceName, TypeNodeList types);
```

These overloads of the `Check` method give you a chance to indicate that your rule should be called when a specific kind of programming construct is currently the focus of the code analysis engine. You do not need to implement all of the `Check` methods in your custom rules, only the ones that expose the constructs you need.

In our example, we're looking for constants in an assembly, so we need to observe the various members of each class, looking for those that are constants and exposed publicly. Therefore, we need to use the `Check(Member member)` overload. This method will be called each time the analysis engine finds any type member, be it a constant, method, field, property, or other member type.

Writing the rule implementation code

You now have a base class for your rule and an understanding of the `Microsoft.Cci` namespace and `IIntrospectionRule` methods that will help you write the implementation. Create a new class file, `AvoidExposingPublicConstants.cs`. First, add `using` statements for the namespaces you'll use:

```
using System;
using Microsoft.Cci;
using Microsoft.FxCop.Sdk;
using Microsoft.FxCop.Sdk.Introspection;
```

Now, create the class, inheriting from the `BaseStaticAnalysisRule` you created earlier:

```
namespace CustomCodeAnalysisRules
{
    public class AvoidExposingPublicConstants : BaseStaticAnalysisRule
    {
        public AvoidExposingPublicConstants() :
                base("AvoidExposingPublicConstants") {}

        public override ProblemCollection Check(Member member)
        {
            Field f = member as Field;
            if (f == null)
            {
                // Not a field
                return null;
            }

            if (member.DeclaringType is Microsoft.Cci.EnumNode)
            {
                // Inside an enumeration
                return null;
            }

            if (member.IsVisibleOutsideAssembly && f.IsLiteral)
            {
                // Is publicly visible and is a constant
                Problems.Add(new Problem(GetResolution(member.Name.Name)));
            }

            return Problems;
        }

        public override TargetVisibilities TargetVisibility
        {
            get { return TargetVisibilities.ExternallyVisible; }
        }
    }
}
```

The constructor only has to supply the name of the rule to the base class constructor, which will forward the name of your XML data store and the assembly type reference automatically to the `BaseIntrospectionRule`.

As we determined before, we need to implement the `Check(Member member)` overload from the `IIntrospectionRule` and search each member for constants. The first thing we do is attempt to convert the `Member` to a `Field` instance. If this fails, we know the member was not a field and we can move on to the next member. If it is a `Field`, we check the `Member` to determine whether it was declared inside of an enumeration. We're not interested in enumerations, so we also return `null` in this case.

Finally, we verify that the member is publicly visible with the `IsVisibleOutsideAssembly` property, and that it is a constant, or literal, value with the `IsLiteral` property. If these expressions are true, you have a publicly visible constant, and your rule has been violated.

When a rule has been violated, you must create a new `Problem` instance and add it to your rule's `Problems` collection, provided by the `BaseIntrospectionRule` class. The argument to a `Problem` constructor is a `Resolution` instance. The `BaseIntrospectionRule` class offers a `GetResolution` helper method that loads the resource data from the embedded XML data for the current rule. Arguments to `GetResolution` are automatically inserted into any placeholders such as `{0}` and `{1}`in the rule's resolution text, in the same manner as `String.Format`.

The new `Problem` is added to the `Problems` collection and the `Problems` collection is returned, indicating to the Code Analysis tool that a new violation has been added.

The final item in the rule implementation is the `TargetVisibility` property. This property is used by the Code Analysis tool to determine when items should be fed into the rule's `Check` method(s). The `TargetVisibilities` enumeration has values such as `All`, `ExternallyVisible`, `NotExternallyVisible`, and `Overridable` that can be combined to indicate when the rule should be tested. In our case, we only care about publicly visible members, so we return `TargetVisibilities.ExternallyVisible`.

Creating Rules.XML

With the implementation written, you now need to create an XML node that describes the rule and provides text to help the user understand and address rule violations. The outer `Rules` node specifies the name of the group of rules — for example, `Performance Rules`. It contains one or more `Rule` nodes, each describing a single rule.

Add a new XML file to the project. Name the file `Rules.xml` and enter the following content:

```
<?xml version="1.0" encoding="utf-8" ?>
<Rules FriendlyName="Custom Code Analysis Rules">
  <Rule TypeName="AvoidExposingPublicConstants" Category="Wrox.Custom"
   CheckId="CS0001">
    <Name>Avoid exposing public constants</Name>
    <Description>The values of public constants are compiled into any referencing
assemblies.  Should that value change, it is not sufficient to recompile the source
assembly because that value will also be stored in those referencing assemblies.
Avoid public constants for this reason.</Description>
    <Resolution>Change public constant '{0}' to a readonly variable, or mark it as
     private or internal.</Resolution>
    <MessageLevel Certainty="99">Warning</MessageLevel>
    <FixCategories>NonBreaking</FixCategories>
    <Url>/Custom/AvoidExposingPublicConstants.html</Url>
    <Email>yourname@yourcompany.com</Email>
    <Owner>Contact Person's Name</Owner>
  </Rule>
</Rules>
```

You must embed the XML into the rule assembly, or the Code Analysis tool will not be able to load the XML and your rule will fail. Set this by right-clicking on the XML field and choosing Properties. Under the Advanced section, find the Build Action property and select Embedded Resource. When you build the assembly, the XML will be included in the meta-data.

The `Rule` node has a `TypeName` attribute, which should match the name of the rule class; a `Category`, which is used when displaying violations; and a `CheckId`, which uniquely identifies that rule — for example, in `SuppressMessage` attributes. `Name` is a short but friendly version of the rule name. `Description` contains the full description of what the rule is detecting.

`Resolution` is the full text shown to users to help them correct the violation. It may contain placeholders, such as `{0}`, which will automatically be replaced with values from the implementation code, as discussed in the previous section. This is extremely useful to help users quickly identify where problems exist. `Resolution` also supports an optional `Name` attribute, which enables you to specify multiple resolutions for the same rule, which can be selected at analysis time by your rule. To do so, instead of using the `GetResolution` method, use `GetNamedResolution`, supplying the name you wish to match.

`MessageLevel` provides a `Certainty` that the rule is applicable, with values from 0 to 99. A 99 indicates there is little doubt the rule has been violated and should be addressed. A lower value means violations of the rule are difficult to detect with great certainty. Use this value to indicate to the user how likely it is that a specific member or assembly has violated the rule. The element value of the `MessageLevel` can be any of the `Microsoft.Tools.FxCop.Sdk.MessageLevel` enumeration values, including `Information`, Warning, CriticalWarning, Error, or CriticalError. Use these to indicate the relative severity of violating a rule. You can see this and the MessageLevel value in practice when you open the XML report from a Code Analysis run, as shown in Figure 8-4.

FixCategories indicate whether the changes needed to correct a rule violation should generally be considered breaking or nonbreaking. Values come from the Microsoft.Tools.FxCop.Sdk.FixCategories enumeration and can be Breaking, NonBreaking, or DependsOnFix. DependsOnFix ties back to the concept of multiple named resolutions. For example, a custom rule has two named resolutions, each used for different rule-violation scenarios. One resolution is easy to implement and considered nonbreaking, but the other is complex to correct, requiring a breaking change.

The Url is the path to an optional file that will show full details of the rule to the user, beyond what is given in the IDE. Email is the optional address of a contact person for help on the rule. Owner is the optionally provided name of a contact person for the rule.

Deploying a rule

You now have a complete rule assembly with embedded XML containing the supporting data for the contained rule(s). The easiest way to get Team System to use the contained rules is to move the assembly into the \Team Tools\Static Code Analysis\FxCop\Rules subdirectory of your Visual Studio installation directory. This will cause the IDE to recognize the rule assembly and read the contained rules so you can select them for inclusion.

However, you need to make a one-time change in order to get the Code Analysis engine to load custom rule assemblies at analysis time. MSBuild needs to be configured to load the custom rule assemblies so they can be applied. To do this, open the file Microsoft.CodeAnalysis.Targets in the C:\Program Files\MSBuild\Microsoft\VisualStudio\v8.0\Code Analysis directory. Find the CodeAnalysisRuleAssemblies node and replace it with the following line:

```
<CodeAnalysisRuleAssemblies
    Condition="'$(FxCopRuleAssemblies)'==''">$(FxCopDir)\rules
</CodeAnalysisRuleAssemblies>
```

Now, every time you run code analysis, any custom rule assemblies in the \FxCop\Rules\ directory will be loaded, and any enabled rules will be invoked.

A useful way to debug new rules is to create a single solution containing both the custom rule project and a sample target project with code that violates the rule. Open the Properties window for the rule assembly project and choose the Build Events tab. Add a post-build event command line to copy the rule assembly from the source project to the \Team Tools\Static Code Analysis\FxCop\Rules directory.

A problem with this approach is that if you open the Code Analysis properties window in either project, the custom rule assembly will be loaded and locked by the IDE. When this happens, you'll need to close and reopen Visual Studio. However, this approach will generally make your rule debugging process much easier.

Learning from existing rules

You've now seen how to create your own rules and integrate them into the Code Analysis tool of Team System. You will certainly find many uses for additional rules, but before you begin creating them you should invest some time learning from examples.

Our recommended approach for those wishing to implement custom rules is to look at how the rules that are included with Team System were written. While you don't have direct access to the source code for these rules, there is a tool that can help. "Reflector," written by Lutz Roeder and available at `www.aisto.com/roeder/dotnet/`, uses the power of .NET's reflection services to peer inside any assembly and generate an approximation of the source code. The target assembly can be any assembly, including those from Microsoft.

After obtaining Reflector, find the existing rules files in your Visual Studio 2005 installation directory under `Team Tools\Static Analysis Tools\FxCop\Rules`. Using Reflector, open one of the rule assemblies, such as `PerformanceRules.dll`. You can then navigate to the `Microsoft.Tools.FxCop.Rules.Performance` namespace, where you will see all of the rules in the Performance category.

Opening each rule will show you the details you've learned earlier in this chapter. The rules inherit from base helper classes, just as you saw with `AvoidExposingPublicConstants` and `BaseStaticAnalysisRule`. Opening the Resources node for any rule assembly enables you to view the XML data that was embedded in the assembly. Opening members such as the `Check` methods will show you code that, while not exactly original, will give you enough detail to determine how you might accomplish the same tasks in your own rules.

Summary

This chapter demonstrated the need for static analysis tools, introducing you to the .NET Framework Design Guidelines for Class Library Developers. These guidelines are a very important resource that Microsoft has made freely available and which are the basis for Team System's included Code Analysis rules.

We described the Managed Code Analysis tool, including how it now integrates with Visual Studio Team System and enables rule analysis to be performed with a simple build. You learned how to configure and execute analysis and how to work with the resulting rule violation messages.

To support projects using a repeatable build process or that need additional flexibility, you learned how to use the command-line Managed Code Analysis tool, and how to create `FxCopCmd` project files to store settings.

Finally, you walked through the process of creating your own rule and integrating it with your analysis process. You created the simple `AvoidExposingPublicConstants` rule as an example of how to use the new introspection engine to create new Managed Code Analysis rules in Team System.

In the next chapter, we introduce the Unmanaged Code Analysis tool. It is similar in purpose to the one described in this chapter, but is instead used for static analysis of native code.