Continuous Integration: From Theory to Practice, 2nd Edition

A practical guide for implementing Continuous Integration in a .NET 3.5 Development Environment

Author: Blog: Email address:	Carel Lotz http://dotnet.org.za/cjlotz carel.lotz@gmail.com
Document version: Date last modified:	1.1 7 April 2008

Version	**Date**	**Description**
1.0	16 January 2008	Initial publication
1.1	07 April 2008	Fixed the broken CodeCoverage target when using MbUnit to pass in the assemblies to profile via the command line

Introduction

Continuous Integration (CI) is a popular incremental integration process whereby each change made to the system is integrated into a latest build. These integration points can occur continuously on every commit to the repository or on regular intervals like every 30 minutes. They should however not take longer than a day i.e. you need to integrate at least once daily.

In this guide I take a closer look at CI. The guide is divided into two main sections: theory and practice. In the *Theory* section, I consider some CI best practices; look at the benefits of integrating frequently and reflect on some recommendations for introducing CI into your environment. The *Practice* section provides an in-depth example of how to implement a CI process using .NET 3.5 development tooling that includes Visual Studio 2008, MSBuild, MSBuild Community Tasks, CruiseControl.NET, Subversion, FxCop, TypeMock, NUnit, MbUnit, NCover, NDepend, Sandcastle and Windows Installer Xml (WiX).

What's New in the 2nd Edition?

For readers of the first edition of the guide, the most notable differences between the second edition and the first edition of the guide are:

1. Updated to use VS 2008, .NET 3.5 and MSBuild 3.5 (including new MSBuild features like parallel builds and multi-targeting).
2. All tools (NUnit, NDepend, NCover etc.) are now stored in a separate **Tools** folder and kept under version control. The only development tools a developer needs to install are VS 2008, SQL Server 2005 and Subversion. The rest of the tools are retrieved form the mainline along with the latest version of the source code.
3. Added the CruiseControl.NET configuration to version control and created a **single step setup** process for the build server. This **greatly** simplifies the process of setting up a new build server.
4. Changed from using InstallShield to WiX for creating a Windows installer (msi).
5. Added support for running MbUnit tests in addition to the NUnit tests.
6. Added support for running standalone FxCop in addition to running VS 2008 Managed Code Analysis.
7. Added targets to test the install and uninstall of the Windows installer created.
8. Consolidated the CodeDocumentationBuild to become part of the DeploymentBuild.
9. Removed the QTP integration. If you want to still integrate QTP, refer to the QtpBuild in the first edition of the guide.
10. Used the latest version of all the tools available.

Theory

In this section I consider some CI best practices; look at the benefits of integrating frequently and reflect on some recommendations for introducing CI into your development environment.

References

I used the following references:

- [McConnell] *Code Complete, 2nd Edition* by Steve McConnell.
- [Fowler] *Continuous Integration* by Martin Fowler.
- [Miller] *Using Continuous Integration? Better do the "Check In Dance"* by Jeremy Miller.
- [Elssamadisy] *Patterns of Agile Practice Adoption: The Technical Cluster* by Amr Elssamadisy.
- [Duvall] *Continuous Integration Anti-Patterns* by Paul Duvall.

Practices

Martin Fowler presents the following set of practices for CI [Fowler]:

- **Maintain a single source repository** - Use a decent source code management system and make sure its location is well known as the place where everyone can go to get source code. Also ensure that everything is put into the repository (test scripts, database schema, third party libraries etc.)
- **Automate the build** - Make sure you can build and launch your system using MSBuild/NAnt scripts in a single command. Include everything into your build. As a simple rule of thumb: anyone should be able to bring in a clean machine, check the sources out of the repository and issue a single command to have a running system on their machine.
- **Make your build self-testing** - Include automated tests in your build process.
- **Everyone commits every day** - "*A commit a day keeps the integration woes away*" [Duvall]. Frequent commits encourage developers to break down their work into small chunks of a few hours each. Before committing their code, they need to update their working copy with the mainline, resolve any conflicts and ensure that everything still works fine. Jeremy Miller refers to this process as the "Check In Dance" [Miller].
- **Every commit should build the mainline on an integration machine** - Use a CI server like or CruiseControl.NET or Team Foundation Build. If the mainline build fails, it needs to be fixed right away.
- **Keep the build fast** - If the build takes too long to execute, try and create a staged build/build pipeline where multiple builds are done in a sequence. The first build (a.k.a "*commit build*") executes quickly and gives other people confidence that they can work with the code in the repository. Further builds can run additional, slower running unit tests, create code metrics, check the code against coding standards, create documentation etc.
- **Test in a clone of the production environment** - Try to set up your test environment to be as exact a mimic of your production environment as possible. Use the same versions of third party software, the same operating system version etc. Consider using virtualization to make it easy to put together test environments.

- **Make it easy for anyone to get the latest executable** - Make sure that there is a well known place where people can find the latest executable.
- **Everyone can see what's happening** - Make the state of the mainline build as visible as possible. Use various feedback mechanisms to relate build status information [Duvall]. Some fun examples include using lava lamps, a big screen LCD and an ambient orb.
- **Automate deployment** - Have scripts that allow you to automatically deploy into different environments, including production. For web applications, consider deploying a trial build to a subset of users before deploying to the full user base.

Jeremy Miller adds the following advice [Miller]:

- **Check in as often as you can -** Try breaking down your workload into meaningful chunks of code and integrate these pieces of code when you have a collection of code in a consistent state. Checking in once a week seriously compromises the effectiveness of a CI process.
- **Don't leave the build broken overnight** - Developers need to be immediately notified upon a build breakage and make it a top priority to fix a broken build.
- Don't ever check into a busted build.
- If you are working on fixing the build, let the rest of the team know.
- Every developer needs to know how to execute a build locally and troubleshoot a broken build.

Paul Duvall also warns against some additional CI anti-patterns that tend to produce adverse effects [Duvall]. The ones that have not been covered above are:

- **The cold shoulder of spam feedback** - Team members sometimes quickly become inundated with build status e-mails (success and failure and everything in between) to the point where they start to ignore messages. Try to make the feedback succinctly targeted so that people don't receive irrelevant information.
- **Don't delay feedback with a slow machine** - Get a build machine that has optimal disk speed, processor, and RAM resources for speedy builds.

Benefits

Numerous benefits result from integrating continuously [McConnell]:

- **Errors are easier to locate** - New problems can be narrowed down to the small part that was recently integrated.
- **Improved team morale** - Programmers see early results from their work.
- **Better customer relations** - Customers like signs of progress and incremental builds provide signs of progress frequently.
- **More reliable schedule estimates & more accurate status reporting** - Management gets a better sense of progress than the typical "*coding is 99% percent complete"* message.
- **Units of the system are tested more fully** - As integration starts early in the project the code is exercised as part of the overall system more frequently.
- **Work that sometimes surfaces unexpectedly at the end of a project is exposed early on**

Where do I start?

Here are some steps to consider for introducing a CI process [Fowler]:

- **Get the build automated** - Get everything into source control and make sure you can build the whole system with a single command.
- **Introduce automated testing in the build** - Identify major areas where things go wrong and start adding automated tests to expose these failures.
- **Speed up the build** - Try aiming at creating a build that runs to completion within ten minutes. Constantly monitor your build and take action as soon as your start going slower than the ten minute rule.
- **Start all new projects with CI from the beginning**

Recommended Reading

The following books detail some excellent techniques that can be applied in your CI environments to keep it running smoothly.

1. Continuous Integration: Improving Software Quality and Reducing Risk by Paul Duvall.
2. Software Configuration Management Patterns by Steve Berczuk and Brad Appleton.
3. Refactoring Databases by Scott Ambler and Pramodkumar Sadalage.
4. Visual Studio Team System: Better Software Development for Agile Teams by Will Stott and James Newkirk.

Practice

The Practice section highlights the steps required for creating a CI process for a sample .NET application. The application is developed using VS 2008, Subversion and various agile practices like CI and TDD. All databases are scripted into various .sql script files and treated as file artefacts under source control.

The Practice section is divided into the following chapters:

- Part 1 covers the background, requirements, process and tools required for the whole CI process.
- Part 2 covers the common build targets and tasks that are used by the **DeveloperBuild, DeploymentBuild** and **CodeStatisticsBuild** as well as showing how to automate the setup of your build server.
- Part 3 covers the DeveloperBuild
- Part 4 covers the DeploymentBuild
- Part 5 covers the CodeStatisticsBuild

I end off the guide by showing you how to use some additional community extensions to add some further panache to your CI build.

Resources

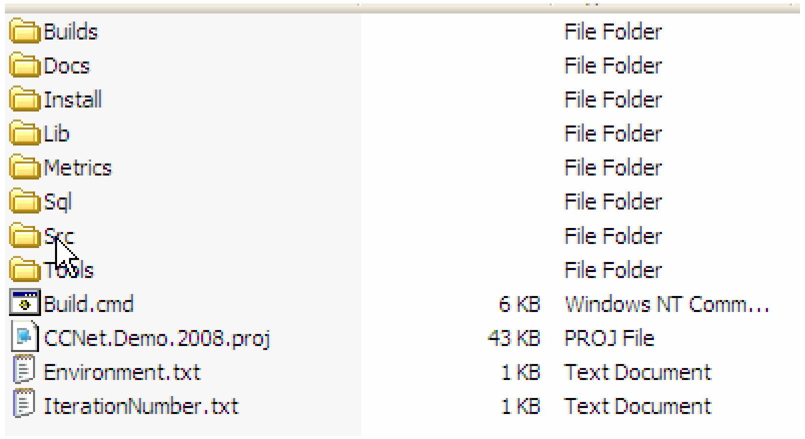
I found the following resources helpful for getting to grips with the power of MSBuild:

1. Book: Deploying .NET Applications: Learning MSBuild and ClickOnce
2. Blog: MSBuild Team Blog
3. .NET Developer Journal: MSBuild – What It Does and What You Can Expect in the Future
4. MSDN Magazine: Automate Releases With MSBuild and Windows Installer XML
5. Channel 9 Wiki: MSBuild.Links
6. Channel 9 Wiki: MSBuild.HomePage
7. MSDN Library: MSBuild Overview
8. MSDN Library: MSBuild Reference
9. Code: MSBuild Community Tasks

The documentation for CruiseControl.NET was sufficient for setting up the build server to use CC.NET with MSBuild.

Part 1: Requirements, Process and Tools

One of the most important requirements for any CI process is to put everything under source control in your repository. This allows a developer to get up and running by simply retrieving the latest/well-known version from the mainline in your repository. As all the tools are also kept in the repository, the developer will automatically have the right version of all the dependencies to start building new features/fix bugs. To support this requirement, the following directory structure is used for a project:



Builds		File Folder
Docs		File Folder
Install		File Folder
Lib		File Folder
Metrics		File Folder
Sql		File Folder
Src		File Folder
Tools		File Folder
Build.cmd	6 KB	Windows NT Comm...
CCNet.Demo.2008.proj	43 KB	PROJ File
Environment.txt	1 KB	Text Document
IterationNumber.txt	1 KB	Text Document

- **Builds** folder contains a separate folder for every CruiseControl.NET (CC.NET) build and its build artefacts as well as a **CruiseControl.NET** folder containing all the project specific CC.NET configurations (custom style sheets, server configuration etc.) for setting up the build server. Please note that the physical build logs for every build are not stored in the repository by reside only on the build server.
- **Docs** folder contains the help file settings and MSDN style help file that is generated from the XML code comments.
- **Install** folder contains the WiX install project file and various merge modules and other files required to create a Windows installer for the application.
- **Lib** folder contains all the third party libraries (NHibernate, Spring.NET etc.) that are used as file references.
- **Metrics** folder contains all the metrics generated for the system like the code coverage results.
- **Sql** folder contains a sub-folder for every database. Each database sub-folder contains all the sql script files to create the structure and content for the database as well as some batch files to automate this creation process using osql/sqlcmd. Both SQL Server 2000 + SQL Sever 2005 are supported.
- **Src** folder contains all the VS 2008 projects as a flat hierarchy of sub-folders. All the .sln files reside in the **Src** root folder whilst all non-unit test project outputs are compiled into a **Src\Bin** sub-folder.

- **Tools** folder contains a separate folder for all the tools (NUnit, NCover, NDepend etc.) that are used by the project. A developer simply needs to get the latest version from the repository to run these tools.[1]

Process

Another important requirement for the CI process is that a developer should be able to harvest the same build process as the build server whilst working in his/her private workspace/sandbox before committing their changes to the repository. The build server will obviously use a few additional tasks for deployment, but the same compile/test process should be re-usable by both the developer and the build server.

The build needs to complete the following tasks:

1. Get the latest source from the repository
2. Build the databases
3. Build the source code
4. Run the unit tests
5. Generate NCover code coverage results
6. Generate FxCop code analysis results
7. Generate NDepend code metrics results
8. Build MSDN style help documentation from the XML code comments
9. Backup the databases
10. Version the source code
11. Sign the assemblies
12. Build a Windows Installer
13. Deploy the Windows Installer
14. Notify QA via e-mail
15. Tag the repository

Not all of these tasks need to be run on a continuous basis within the developer sandbox. You may also find that running code coverage and code analysis on every check-in causes your build to take longer than the 10-minutes per build rule. I therefore decided to create a staged build/build pipeline and to split the CI build process into 3 separate builds. Depending on your own setup, you may choose to combine all three builds into a single/two build(s).

1. **DeveloperBuild** - This build is set to monitor the repository every 60 seconds for changes. It builds the databases, compiles the source code and runs the unit tests. It does incremental builds for building the databases and for compiling the source code to get quicker build times.

[1] Unfortunately TypeMock and WiX integrate with VS itself. If you want to use these tools and their VS integration abilities you have to run their installers. For WiX it is easy enough to author the installer just using Xml and not the VS integration. However, if you use TypeMock and you want to debug your unit tests running in VS, you will have to install TypeMock. As an alternative, consider using Rhino.Mocks as a mock framework as it does not require you to install anything.

2. **DeploymentBuild** - The deployment build is forced when required to create the Windows installer file that is deployed to QA. The build also creates the help documentation that is included as part of the installation. In addition to creating and deploying the installer it also tags the repository with the version number of the build created.
3. **CodeStatisticsBuild** - The statistics build is set to run every morning at 03:00 am and produces the daily Code Coverage, FxCop and NDepend results. The developers still evaluate the code coverage, FxCop and NDepend results continuously in their sandbox, but we incur the overhead of running this on the build server early in the morning when there is no other activity on the build server. This way we have the stats to spot trends on a daily basis.

Tools

These are the tools that are used by the CI process:

1. Visual Studio 2008 Team Edition for Software Developers to run Managed Code Analysis. You can use the standalone version of FxCop, but there are some minor differences between this and the version used by VS 2008 Code Analysis. I include a target for running the standalone version of FxCop as well.
2. NCover v1.5.8 for Code coverage.
3. NCoverExplorer v1.4.0.7 and NCoverExplorer.Extras v1.4.0.5 to create code coverage reports.[2]
4. NDepend v2.6 for additional code metrics.
5. MSBuild.Community.Tasks for additional MSBuild tasks. I'm using the latest nightly build.
6. NUnit 2.0 2.4.5 / MbUnit 2.4.1 to run the unit tests.
7. TypeMock 4.1 / Rhino.Mocks 3.3 as mock frameworks.
8. WiX 3.0 for creating the Windows Installer (.msi).
9. Subversion 1.4.5 for source control.
10. Sandcastle October 2007 CTP to generate the help file content from the XML code comments.
11. Sandcastle Help File Builder (SHFB) to set the options for the help file and to bootstrap the document creation process.
12. Sandcastle Presentation File Patches to solve some issues with the presentation styles of the Sandcastle October CTP.

To setup a build server, you need to do the following:

1. Install VS 2008
2. Install SQL Server 2005
3. Install Subversion
4. Install CruiseControl.NET
5. Get the latest version from source control
6. Run the following command:

```
msbuild ccnet.demo.2008.proj /tv:3.5 /t:InstallCCNetConfig.
```

[2] These were the last free versions of NCover and NCoverExplorer. Since v2.0, NCover and NCoverExplorer have become COTS.

I'll provide more information on setting up the build server in Part 2.

To setup a developer PC, you need to do the following:

1. Install VS 2008
2. Install SQL Server 2005
3. Install Subversion
4. Get the latest version from source control
5. (Optional) Install TypeMock and WiX. As mentioned TypeMock and WiX integrate with VS 2008. If you want to make use of these tools and their VS integration features you need to still run their installers manually.

Next Steps

Well, that takes care of the all the background information. The next chapter will delve into all the common build targets that are used by the **DeveloperBuild, DeploymentBuild** and **CodeStatisticsBuild** and also show how to automate the process of setting up your build server. By the way, I'm a big fan of ReSharper and the MSBuild/NAnt support of ReSharper is excellent!

Part 2: Common Build Targets and Setting up the Build Server

Part 2 covers the common build targets and tasks that are shared and used by the **DeveloperBuild, DeploymentBuild** and **CodeStatisticsBuild** as well as showing you how to set up your build server.

MSBuild Scripts

The whole build process is defined in the CCNet.Demo.2008.proj build file:

Common Property Groups

```
<Project DefaultTargets="BuildCode;BuildTests" InitialTargets="GetPaths;GetProjects;GetEnvironment"
xmlns="http://schemas.microsoft.com/developer/msbuild/2003">

   <!-- ASCII Constants -->
   <PropertyGroup>
      <NEW_LINE>%0D%0A</NEW_LINE>
      <TAB>%09</TAB>
      <DOUBLE_QUOTES>%22</DOUBLE_QUOTES>
      <SPACE>%20</SPACE>
   </PropertyGroup>

   <!-- Misc Settings -->
   <PropertyGroup>
      <Major>1</Major>
      <Minor>0</Minor>
      <Company>YourCompany Ltd</Company>
      <ProductName>CCNetDemo2008</ProductName>
      <BuildTargets Condition=" '$(BuildTargets)' == '' ">Build</BuildTargets>
   </PropertyGroup>

   <!-- Environment Settings -->
   <PropertyGroup>
      <Domain>yourdomain.co.za</Domain>
      <SmtpServer>smtp.$(Domain)</SmtpServer>
      <WikiUrl>http://yourwikiserver/wiki</WikiUrl>
      <BuildMasterEmail>buildmaster@$(Domain)</BuildMasterEmail>
      <TempWorkingFolder>$(Temp)\$(ProductName)</TempWorkingFolder>
      <DBMS Condition=" '$(DBMS)' == '' ">sql2005</DBMS>
      <DBServer Condition=" '$(DBServer)' == '' ">(local)</DBServer>
      <SqlCmdRunner Condition=" '$(DBMS)' == 'sql2000' " >osql</SqlCmdRunner>
      <SqlCmdRunner Condition=" '$(DBMS)' == 'sql2005' " >sqlcmd</SqlCmdRunner>
      <DBBackupFolder Condition=" '$(DBMS)' == 'sql2000' ">$(ProgramFiles)\Microsoft SQL Server\MSSQL\Backup</DBBackupFolder>
      <DBBackupFolder Condition=" '$(DBMS)' == 'sql2005' ">$(ProgramFiles)\Microsoft SQL Server\MSSQL.1\MSSQL\Backup</DBBackupFolder>
      <Deployme ntFolder Condition=" '$(DeploymentFolder)' == '' ">$(TempWorkingFolder)</DeploymentFolder>
```

```xml
    <LocalInstallFolder Condition=" '$(LocalInstallFolder)' == '' ">$(TempWorkingFolder)\Install</LocalInstallFolder>
    <LocalUninstallFolder Condition=" '$(LocalUninstallFolder)' == '' ">$(TempWorkingFolder)\Uninstall</LocalUninstallFolder>
    <CCNetInstallPath Condition=" '$(CCNetInstallPath)' == '' ">$(ProgramFiles)\CruiseControl.NET</CCNetInstallPath>
  </PropertyGroup>

  <!-- Solution Folders -->
  <PropertyGroup>
    <SrcFolder>$(MSBuildProjectDirectory)\Src</SrcFolder>
    <SqlFolder>$(MSBuildProjectDirectory)\Sql</SqlFolder>
    <LibFolder>$(MSBuildProjectDirectory)\Lib</LibFolder>
    <ToolsFolder>$(MSBuildProjectDirectory)\Tools</ToolsFolder>
    <DocsFolder>$(MSBuildProjectDirectory)\Docs</DocsFolder>
    <InstallFolder>$(MSBuildProjectDirectory)\Install</InstallFolder>
    <InstallBinFolder>$(InstallFolder)\Bin</InstallBinFolder>
    <BinFolder>$(SrcFolder)\Bin</BinFolder>
    <BuildsFolder>$(MSBuildProjectDirectory)\Builds</BuildsFolder>
    <MetricsFolder>$(MSBuildProjectDirectory)\Metrics</MetricsFolder>
  </PropertyGroup>

  <!-- Solution Files -->
  <PropertyGroup>
    <KeyFile>$(SrcFolder)\$(ProductName).snk</KeyFile>
    <IterationNumberFile>$(MSBuildProjectDirectory)\IterationNumber.txt</IterationNumberFile>
    <EnvironmentFile>$(MSBuildProjectDirectory)\Environment.txt</EnvironmentFile>
    <LastCodeAnalysisSucceededFile>LastCodeAnalysisSucceeded</LastCodeAnalysisSucceededFile>
    <InstallBuildEmailFile>$(TEMP)\InstallBuildEmailFile.htm</InstallBuildEmailFile>
    <InstallBuildEmailTemplate>$(InstallFolder)\InstallBuildEmailTemplate.htm</InstallBuildEmailTemplate>
    <TestResultFile>TestResult</TestResultFile>
    <FxCopResultFile>CodeAnalysisLog.xml</FxCopResultFile>
    <NCoverResultFile>Coverage.xml</NCoverResultFile>
    <NCoverLogFile>Coverage.log</NCoverLogFile>
    <NCoverSummaryFile>CoverageSummary.xml</NCoverSummaryFile>
    <NCoverHtmlReport>CoverageSummary.html</NCoverHtmlReport>
    <NDependProjectFile>NDependProject.xml</NDependProjectFile>
    <NDependResultFile>NDependMain.xml</NDependResultFile>
    <SandCastleHFBProject>$(ProductName).shfb</SandCastleHFBProject>
    <DBBackupScript>$(SqlFolder)\BackupDB.cmd</DBBackupScript>
    <MsiInstallFile Condition=" '$(MsiInstallFile)' == '' ">$(ProductName).msi</MsiInstallFile>
    <SolutionName Condition=" '$(SolutionName)' == '' ">$(ProductName).sln</SolutionName>
  </PropertyGroup>

  <!-- 3rd Party Program Settings -->
  <PropertyGroup>
    <SubversionPath>C:\Program Files\Subversion\bin</SubversionPath>
    <SubversionCmd>$(SubversionPath)\svn.exe</SubversionCmd>
    <NCoverPath>$(ToolsFolder)\NCover\</NCoverPath>
    <NCoverCoverLib>$(ToolsFolder)\NCover\CoverLib.dll</NCoverCoverLib>
```

```xml
        <NCoverExplorerPath>$(ToolsFolder)\NCoverExplorer\</NCoverExplorerPath>
        <NDependPath>$(ToolsFolder)\NDepend\</NDependPath>
        <NDependOutputPath>$(MetricsFolder)\NDependOut</NDependOutputPath>
        <NUnitPath>$(ToolsFolder)\NUnit\bin\</NUnitPath>
        <NUnitCmd>$(NUnitPath)nunit-console.exe</NUnitCmd>
        <MbUnitPath>$(ToolsFolder)\MbUnit\</MbUnitPath>
        <MbUnitCmd>$(MbUnitPath)mbunit.cons.exe</MbUnitCmd>
        <MbUnitReportFormat>/report-type:Xml</MbUnitReportFormat>
        <MbUnitTasksPath>$(MbUnitPath)</MbUnitTasksPath>
        <SandcastlePath>$(ToolsFolder)\Sandcastle</SandcastlePath>
        <SandcastleHFBPath>$(ToolsFolder)\SHFBuilder\</SandcastleHFBPath>
        <SandcastleHFBCmd>$(SandcastleHFBPath)SandcastleBuilderConsole.exe</SandcastleHFBCmd>
        <MSBuildCommunityTasksPath>$(ToolsFolder)\MSBuild.Community.Tasks</MSBuildCommunityTasksPath>
        <NCoverExplorerTasksPath>$(ToolsFolder)\NCoverExplorer.Extras</NCoverExplorerTasksPath>
        <WixPath>$(ToolsFolder)\WiX\bin\</WixPath>
        <WixTargetsPath>$(WixPath)Wix.targets</WixTargetsPath>
        <WixTasksPath>$(WixPath)WixTasks.dll</WixTasksPath>
        <WixToolPath>$(WixPath)</WixToolPath>
        <TypeMockAutoDeploy Condition=" '$(TypeMockAutoDeploy)' == '' ">false</TypeMockAutoDeploy>
        <TypeMockAutoDeploy Condition=" '$(CCNetProject)' != '' ">true</TypeMockAutoDeploy>

        <CoverageExclusions>
            <CoverageExclusion>
                <ExclusionType>Namespace</ExclusionType>
                <Pattern>^$(ProductName).*\.Properties$</Pattern>
                <IsRegex>true</IsRegex>
            </CoverageExclusion>
            <CoverageExclusion>
                <ExclusionType>Namespace</ExclusionType>
                <Pattern>^$(ProductName).*\.Resources$</Pattern>
                <IsRegex>true</IsRegex>
            </CoverageExclusion>
            <CoverageExclusion>
                <ExclusionType>Class</ExclusionType>
                <Pattern>Resources</Pattern>
            </CoverageExclusion>
        </CoverageExclusions>
    </PropertyGroup>

    <!-- Subversion Settings -->
    <PropertyGroup>
        <SvnServerPath>http://svn.$(Domain)/$(ProductName)</SvnServerPath>
        <SvnTrunkFolder>$(SvnServerPath)/trunk</SvnTrunkFolder>
        <SvnTagsFolder>$(SvnServerPath)/tags</SvnTagsFolder>
    </PropertyGroup>

    <!-- Overriding .csproj Project settings -->
```

```
128    <PropertyGroup>
129       <OutputPath Condition=" '$(OutputPath)' == '' ">$(BinFolder)</OutputPath>
130       <Configuration Condition=" '$(Configuration)' == '' ">Debug</Configuration>
131       <Platform Condition=" '$(Platform)' == '' ">AnyCPU</Platform>
132       <RunCodeAnalysis Condition=" '$(RunCodeAnalysis)' == '' ">false</RunCodeAnalysis>
133       <CodeAnalysisRules>-Microsoft.Globalization#CA1300;-Microsoft.Globalization#CA1301;-Microsoft.Globalization#CA1302;-
Microsoft.Naming#CA1701;-Microsoft.Naming#CA1702;-Microsoft.Naming#CA1703;-Microsoft.Naming#CA1704;-Microsoft.Naming#CA1726;-
Microsoft.Usage#CA2243</CodeAnalysisRules>
134    </PropertyGroup>
135
136    <ItemGroup>
137       <Developers Include="developer1@$(Domain);
138                           developer2@$(Domain)" />
139    </ItemGroup>
140
141    <!-- Imports -->
142    <Import Project="$(MSBuildToolsPath)\Microsoft.CSharp.targets" />
143    <Import Project="$(ToolsFolder)\TypeMock\TypeMock.MSBuild.Tasks"/>
144    <Import Project="$(ToolsFolder)\MSBuild.Community.Tasks\MSBuild.Community.Tasks.Targets"/>
145    <Import Project="$(ToolsFolder)\NCoverExplorer.Extras\NCoverExplorer.MSBuildTasks.targets"/>
146
147    <!-- Add our CleanSolution task to the general Clean task -->
148    <PropertyGroup>
149       <CleanDependsOn>
150          $(CleanDependsOn);
151          CleanSolution
152       </CleanDependsOn>
153    </PropertyGroup>
```

Looking at the script, it defines various property groups that define properties for ASCII constants, Environment settings, 3rd Party Program Settings, Solution Folders, Solution Files, Version Information etc. These properties are used by the different Targets within the project. The project also imports the targets containing the additional MSBuild tasks provided by TypeMock, NCoverExplorer and the MSBuild.Community.Tasks library. Some other observations:

1. Line 1: The **InitialTargets** for the build is set to GetPaths, GetProjects and GetEnvironment. The targets specified within the InitialTargets will always be executed before any other target within the build file.
2. Line 1: The **DefaultTargets** for the build file is set to BuildCode and BuildTests which implies that if no targets are specified, these targets will be executed.
3. Lines 40-51: We create some properties that map to the directory structure of the project.
4. Lines 54-73: We create some properties that map to the different files that are used within the project like the NDepend project file, NCover log file etc.
5. Lines 99-100: As we do not install TypeMock on the build server, we need to make use of its AutoDeploy feature to allow it to run successfully. The **TypeMockAutoDeploy** property is set to true if the build is being run from CC.NET on the build server (i.e. CCNetProject != null) or if explicitly specified as a property from the MSBuild command line.

6. Lines 102-117: We define a **CoverageExclusions** ItemGroup that contains the regular expression patterns and names of types that NCoverExplorer should ignore when calculating the code coverage result for the solution. We choose to ignore all the namespaces containing the VS generated resource and property files and also all classes with the name of Resources.
7. Lines 128-134: We define some overriding settings to be used for our .csproj files. This includes the **CodeAnalysisRules** that we want to ignore on a solution wide basis as well as redirecting the output of the projects to the **BinFolder.** It is important to note that the Microsoft.CSharp.targets is imported **after** (see line 142) we have defined the overriding .csproj settings. This ensures that these settings take precedence over the default settings defined in Microsoft.CSharp.targets as ours are defined first.
8. Lines 148-153: We add our own CleanSolution task to the **CleanDependsOn** property. The **Clean** target defined in Microsoft.Common.targets depends on the targets defined within the CleanDependsOn property. By adding our own CleanSolution target to the property we ensure that whenever somebody builds the project with the **Clean** target, our CleanSolution target will also be executed thus allowing us to clean up the temporary files created by our own build.

General Purpose Targets

The following general purpose targets are used by the build:

GetPaths

```
<!-- GetPaths -->
<Target Name="GetPaths">
   <GetFrameworkSdkPath>
      <Output
          TaskParameter="Path"
          PropertyName="FrameworkSdkPath" />
   </GetFrameworkSdkPath>

   <GetFrameworkPath>
      <Output
          TaskParameter="Path"
          PropertyName="FrameworkPath" />
   </GetFrameworkPath>

   <Message Text="SdkPath: $(FrameworkSdkPath)$(NEW_LINE)FrameworkPath: $(FrameworkPath)$(NEW_LINE)VStudioPath: $(VSINSTALLDIR)" Importance="high" />
</Target>
```

The GetPaths target stores the .NET Framework and Framework SDK paths into 2 properties for later use by using the GetFrameworkSdkPath and GetFrameworkPath tasks.

GetProjects

```
<!-- GetProjects -->
<Target Name="GetProjects">

   <!-- Get all the projects associated with the solution -->
```

```xml
       <GetSolutionProjects Solution="$(SrcFolder)\$(SolutionName)">
          <Output TaskParameter="Output" ItemName="SolutionProjects" />
       </GetSolutionProjects>

       <!-- Filter out solution folders and non .csproj items -->
       <RegexMatch Input="@(SolutionProjects)" Expression=".[\.]csproj$">
          <Output TaskParameter="Output" ItemName="CSProjects"/>
       </RegexMatch>

       <!-- Resolve the test projects -->
       <RegexMatch Input="@(CSProjects)" Expression=".[\.](UnitTests|IntegrationTests)[\.]csproj$">
          <Output TaskParameter="Output" ItemName="TestProjects"/>
       </RegexMatch>

       <!-- Resolve the projects -->
       <ItemGroup>
          <SqlProjects Include="$(SqlFolder)\**\*.proj"/>
          <InstallProjects Include="$(InstallFolder)\**\*.wixproj"/>

          <CodeProjects Include="@(CSProjects)"
                        Exclude="@(TestProjects);
                                 @(SqlProjects)" />
       </ItemGroup>

       <Message Text="$(NEW_LINE)Resolved the following solution projects:" Importance="high" />
       <Message Text="CodeProjects:$(NEW_LINE)$(TAB)@(CodeProjects->'%(RecurseDir)%(FileName)%(Extension)', '$(NEW_LINE)$(TAB)')" Importance="high"/>
       <Message Text="TestProjects:$(NEW_LINE)$(TAB)@(TestProjects->'%(RecurseDir)%(FileName)%(Extension)', '$(NEW_LINE)$(TAB)')" Importance="high"/>
       <Message Text="SqlProjects:$(NEW_LINE)$(TAB)@(SqlProjects->'%(RecurseDir)%(FileName)%(Extension)', '$(NEW_LINE)$(TAB)')" Importance="high"/>
       <Message Text="InstallProjects:$(NEW_LINE)$(TAB)@(InstallProjects->'%(RecurseDir)%(FileName)%(Extension)', '$(NEW_LINE)$(TAB)')" Importance="high"/>

    </Target>
```

The GetProjects target parses any .sln file to get a list of projects to operate on. As you may use different .sln files that contain different projects we want the build to figure out by itself which projects should be compiled given any .sln file specified via the **SolutionName** property. Remember, that as mentioned in Part 1, the developer should be able to execute the same build processes as the build server to compile and test the code in his/her sandbox. The GetProjects target uses the **GetSolutionProjects** and **RegexMatch** tasks from the **MSBuild.Community.Tasks** to parse out and filter the different project files from a .sln file according to the naming conventions used. The results are stored in different item groups to be used later on in the build.

The rational for using different solution files is that of developer productivity. Some solution files may ignore for instance the Integration Tests that typically take longer to execute. This allows a developer to work on a smaller subset of projects which makes working on slow developer PC's less frustrating. It goes without saying that no commit back into the repository is allowed without the developer ensuring that the main .sln file that the build server uses runs through successfully in his/her developer sandbox.

GetEnvironment

```
1     <!-- GetEnvironment -->
2     <Target Name="GetEnvironment">
3        <!-- Read the the Environment file contents -->
4        <ReadLinesFromFile File="$(EnvironmentFile)">
5           <Output TaskParameter="Lines" ItemName="EnvironmentFileContents"/>
6        </ReadLinesFromFile>
7
8        <!-- Assign file contents to Environment property -->
9        <PropertyGroup>
10          <Environment>@(EnvironmentFileContents->'%(Identity)')</Environment>
11       </PropertyGroup>
12
13       <Message Text="Running build for Environment = $(Environment)" importance="high" />
14    </Target>
```

Observations:

- Lines 4-6: The environment to make the build for is read from the Environment.txt file. This allows us to build for a different environment by simply changing the file - we **do not** want to change something on the build server to compile for a different environment. We want to change a file that is stored and retrieved from source control to remember for what environment a specific build was made.
- Line 10: The contents of the file are assigned to the **Environment** property that is used later on to build for a specific environment.

GetIterationNumber

```
<!-- GetIterationNumber -->
<Target Name="GetIterationNumber">
   <!-- Read the the iteration number file contents -->
   <ReadLinesFromFile File="$(IterationNumberFile)">
      <Output TaskParameter="Lines" ItemName="IterationNumberFileContents"/>
   </ReadLinesFromFile>

   <PropertyGroup>
      <IterationNumber>@(IterationNumberFileContents->'%(Identity)')</IterationNumber>
   </PropertyGroup>

   <Message Text="Running build for Iteration = $(IterationNumber)" Importance="high" />
</Target>
```

The GetIterationNumber target reads the current iteration number from a file and assigns it to an **IterationNumber** property. The IterationNumber is used as the Build number when versioning the assemblies, databases and msi installation.

GetRevisionNumber

```
<!-- GetRevisionNumber -->
<Target Name="GetRevisionNumber">
   <!-- Get the revision number of the local working copy -->
   <SvnInfo LocalPath="$(MSBuildProjectDirectory)">
      <Output TaskParameter="Revision" PropertyName="Revision"/>
   </SvnInfo>

   <Message Text="Running build for Revision = $(Revision)" importance="high" />
</Target>
```

The GetRevisionNumber target reads the current revision number from the Subversion repository and assigns it to a **Revision** property. It does this by using the **SvnInfo** task defined within the MSBuild.Community.Tasks. We use the Revision number when versioning our assemblies.

CleanSolution

```
<!-- CleanSolution -->
<Target Name="CleanSolution">
   <Message Text="Cleaning Solution Output"/>

   <!-- Create item collection of custom artifacts produced by the build -->
   <ItemGroup>
      <SolutionOutput Include=
                "$(BinFolder)\**\*.*;
                 $(SqlFolder)\**\*.LastUpdateSucceeded;
                 @(TestProjects->'%(RootDir)%(Directory)bin\$(Configuration)\*$(TestResultFile)*');
                 @(TestProjects->'%(RootDir)%(Directory)bin\$(Configuration)\*$(NCoverResultFile)');
                 @(TestProjects->'%(RootDir)%(Directory)bin\$(Configuration)\*$(NCoverLogFile)')" />
   </ItemGroup>

   <!-- Delete all the solution created artifacts -->
   <Delete Files="@(SolutionOutput)"/>

   <PropertyGroup>
      <BuildTargets>Clean;$(BuildTargets)</BuildTargets>
   </PropertyGroup>

   <Message Text="BuildTargets: $(BuildTargets)" Importance="low" />
</Target>
```

The CleanSolution target takes care of removing all the temporary files created by our build. This includes the temporary files that we create to support incremental builds. It also adds the **Clean** target to the default list of **BuildTargets** to execute. The BuildTargets property is used as the Targets to execute when calling the MSBuild task to compile the code - more on this and incremental builds later on in Part 3.

Subversion Targets

The following targets are used for Subversion integration by the build:

```
<!-- SvnVerify -->
<Target Name="SvnVerify">

   <Error Text="No UserName or Password for accessing the repository has been specified"
          Condition=" '$(SvnUsername)' == '' Or '$(SvnPassword)' == '' " />

</Target>

<!-- CommitChanges -->
<Target Name="CommitChanges"
        DependsOnTargets="SvnVerify">

   <PropertyGroup>
      <SvnMessage Condition=" '$(SvnMessage)' == '' ">Automatic commit by $(ComputerName) as part of a build</SvnMessage>
   </PropertyGroup>

   <!-- Commit to the repository -->
   <SvnCommit Targets="."
              Username="$(SvnUserName)"
              Password="$(SvnPassword)"
              Message="$(SvnMessage)" />
</Target>

<!-- GetLatestChanges -->
<Target Name="GetLatestChanges"
        DependsOnTargets="SvnVerify">

   <SvnUpdate RepositoryPath="$(SvnTrunkFolder)"
              LocalPath="$(MSBuildProjectDirectory)"
              Username="$(SvnUserName)"
              Password="$(SvnPassword)"/>

</Target>

<!-- TagRepository -->
<Target Name="TagRepository"
        DependsOnTargets="SvnVerify">

   <PropertyGroup>
      <SvnTag Condition=" '$(SvnTag)' == '' And '$(AppVersion)' != '' ">Build-$(AppVersion)</SvnTag>
      <SvnMessage Condition=" '$(SvnMessage)' == '' ">Automatic tag by $(ComputerName) as part of a build</SvnMessage>
   </PropertyGroup>

```

```
    <Error Text="SvnTag property has not been specified" Condition=" '$(SvnTag)' == '' "/>

    <!-- Tag the repository -->
    <SvnCopy SourcePath="$(SvnTrunkFolder)"
             DestinationPath="$(SvnTagsFolder)/$(SvnTag)"
             Message="$(SvnMessage)"
             Username="$(SvnUserName)"
             Password="$(SvnPassword)" />

    <Message Text="Tag created at $(SvnTagsFolder)/$(SvnTag)"  Importance="high"/>
  </Target>
```

The **SvnVerify** target verifies that the user details required for integrating with the Subversion repository has been specified for the build and is used by the other Subversion targets. The **GetLatestChanges**, **CommitChanges** and **TagRepository** targets use different Subversion tasks from the MSBuild.Community.Tasks to update the working copy with the latest changes; commit the working copy's changes to the repository and to tag the repository with a specific revision number respectively.

Miscellaneous Targets

The following miscellaneous targets are used by the build:

Install

```
  <!-- Install -->
  <Target Name="Install">

    <Error Text="No LatestVersionFolder has been specified" Condition=" '$(LatestVersionFolder)' == ''" />

    <!-- Copy the latest install from the deployment server -->
    <Copy SourceFiles="$(LatestVersionFolder)\$(MsiInstallFile)"
          DestinationFolder="$(LocalInstallFolder)"/>

    <!-- Launch the install in silent mode -->
    <Exec Command="msiexec /i $(DOUBLE_QUOTES)$(LocalInstallFolder)\$(MsiInstallFile)$(DOUBLE_QUOTES) /passive /l*
$(DOUBLE_QUOTES)$(LocalInstallFolder)\Install.log$(DOUBLE_QUOTES)"/>

    <!-- Copy the msi to the Uninstall folder to use for later uninstallation -->
    <Copy SourceFiles="$(LocalInstallFolder)\$(MsiInstallFile)"
          DestinationFolder="$(LocalUninstallFolder)"/>
  </Target>
```

The Install target automates the process of copying the latest version of the Windows installer file from the deployment server and running the installer in silent mode. This provides a way to test the installation as well as automatically installing the application for any automated regression testing tool like QTP.

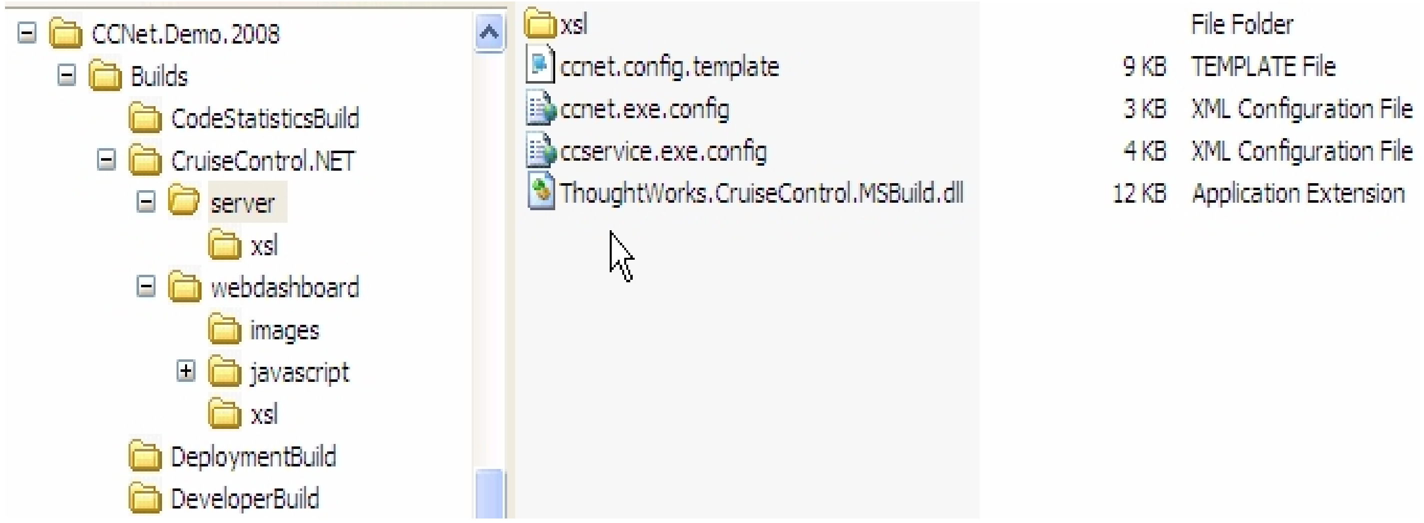
Uninstall

```
    <!-- Uninstall -->
    <Target Name="Uninstall">

      <!-- Launch the Uninstall in silent mode -->
      <Exec Command="msiexec /x $(DOUBLE_QUOTES)$(LocalUninstallFolder)\$(MsiInstallFile)$(DOUBLE_QUOTES) /passive /l*
$(DOUBLE_QUOTES)$(LocalUninstallFolder)\Uninstall.log$(DOUBLE_QUOTES)"
            ContinueOnError="true" />

    </Target>
```

The Uninstall target simply automates the process of uninstalling the application by launching the Windows installer file in silent mode with the /x parameter.

Setting up the Build Server

The automation of the build server setup is an important piece in the CI puzzle that, if neglected, may lead to a lengthy and frustrating manual process of trying to set up CC.NET with the correct style sheets and running trial builds to eventually get all the different tools integrated correctly into the CC.NET dashboard running on the build server. To avoid this, I automate the setup of a new build server by firstly storing all the correct CC.NET configurations (style sheets, server configuration etc.) in source control using the following directory structure:



xsl		File Folder
ccnet.config.template	9 KB	TEMPLATE File
ccnet.exe.config	3 KB	XML Configuration File
ccservice.exe.config	4 KB	XML Configuration File
ThoughtWorks.CruiseControl.MSBuild.dll	12 KB	Application Extension

The CruiseControl.NET directory structure mirrors the exact directory structure used by CC.NET when installed on a machine. Within the relevant directories I store the style sheets and additional CC.NET configuration that I simply copy over an existing CC.NET installation to establish my own custom configuration. In addition to copying my own CC.NET artefacts, I make use of two template files:

1. ccnet.config.template - Stored in the **server** directory, this file contains my CC.NET server configuration for the DeveloperBuild, DeploymentBuild and CodeStatisticsBuild. Within the template file, I keep a few tokens that are replaced using the InstallCCNetConfig task.

2. dashboard.config.template – Stored in the **webdashboard** directory this file contains my CC.NET web dashboard settings with a few tokens that are replaced using the InstallCCNetConfig task.

The second piece of automating the setup is the InstallCCNetConfig task.

```
<!-- InstallCCNetConfig -->
<Target Name="InstallCCNetConfig">

   <Message Text="Installing CCNet Configuration on $(ComputerName)" Importance="low"/>

   <Error Text="The CCNetInstallPath directory for $(ComputerName) does not exist!" Condition=" !Exists('$(CCNetInstallPath)')"/>

   <ItemGroup>
      <CCNetFiles Include="$(BuildsFolder)\CruiseControl.NET\**\*.*"
                  Exclude="$(BuildsFolder)\CruiseControl.NET\**\.svn\**;
                           $(BuildsFolder)\CruiseControl.NET\**\*.template" />
   </ItemGroup>

   <!-- Copy the CCNet Configuration -->
   <Copy SourceFiles="@(CCNetFiles)"
         DestinationFiles="@(CCNetFiles->'$(CCNetInstallPath)\%(RecursiveDir)%(FileName)%(Extension)')" />

   <!-- Create the tokens for the ccnet.config template -->
   <ItemGroup>
      <CCNetConfigTokens Include="BuildServer">
         <ReplacementValue>$(ComputerName)</ReplacementValue>
      </CCNetConfigTokens>
      <CCNetConfigTokens Include="ProjectDir">
         <ReplacementValue>$(MSBuildProjectDirectory)</ReplacementValue>
      </CCNetConfigTokens>
      <CCNetConfigTokens Include="YourDomain">
         <ReplacementValue>$(Domain)</ReplacementValue>
      </CCNetConfigTokens>
      <CCNetConfigTokens Include="SvnTrunkUrl">
         <ReplacementValue>$(SvnTrunkFolder)</ReplacementValue>
      </CCNetConfigTokens>
      <CCNetConfigTokens Include="SvnUserName">
         <ReplacementValue>$(SvnUserName)</ReplacementValue>
      </CCNetConfigTokens>
```

```
35          <CCNetConfigTokens Include="SvnPassword">
36             <ReplacementValue>$(SvnPassword)</ReplacementValue>
37          </CCNetConfigTokens>
38          <CCNetConfigTokens Include="MSBuildProjectFile">
39             <ReplacementValue>$(MSBuildProjectFile)</ReplacementValue>
40          </CCNetConfigTokens>
41          <CCNetConfigTokens Include="WikiUrl">
42             <ReplacementValue>$(WikiUrl)</ReplacementValue>
43          </CCNetConfigTokens>
44          <CCNetConfigTokens Include="BuildMasterEmail">
45             <ReplacementValue>$(BuildMasterEmail)</ReplacementValue>
46          </CCNetConfigTokens>
47          <CCNetConfigTokens Include="SmtpServer">
48             <ReplacementValue>$(SmtpServer)</ReplacementValue>
49          </CCNetConfigTokens>
50          <CCNetConfigTokens Include="BuildQueue">
51             <ReplacementValue>$(ProductName)</ReplacementValue>
52          </CCNetConfigTokens>
53       </ItemGroup>
54
55       <!-- Create the ccnet.config file by replacing the tokens in the template file -->
56       <TemplateFile Template="$(BuildsFolder)\CruiseControl.NET\server\ccnet.config.template"
57                     OutputFilename="$(CCNetInstallPath)\server\ccnet.config"
58                     Tokens="@(CCNetConfigTokens)" />
59
60       <!-- Create the tokens for the dashboard.config template -->
61       <ItemGroup>
62          <DashBoardTokens Include="BuildServer">
63             <ReplacementValue>$(ComputerName)</ReplacementValue>
64          </DashBoardTokens>
65          <DashBoardTokens Include="YourDomain">
66             <ReplacementValue>$(Domain)</ReplacementValue>
67          </DashBoardTokens>
68       </ItemGroup>
69
70       <!-- Create the dashboard.config file by replacing the tokens in the template file -->
71       <TemplateFile Template="$(BuildsFolder)\CruiseControl.NET\webdashboard\dashboard.config.template"
72                     OutputFilename="$(CCNetInstallPath)\webdashboard\dashboard.config"
73                     Tokens="@(DashBoardTokens)" />
74    </Target>
75
```

Observations:

- Lines 9-16: We create an item group containing all our custom CC.NET artefacts and copy these artefacts over the existing CC.NET installation.

- Lines 20-58: We create our own custom values for all the tokens of the ccnet.config.template file and call the **TemplateFile** task from the MSBuild.Community.Tasks to replace the tokens with our values and store the result in the ccnet.config file for use by CC.NET. It is important to make sure that the ccservice.exe or ccnet.exe is not running as these will keep a file lock onto the ccnet.config file and prevent the **TemplateFile** task from replacing the file. Alternatively, if you have an existing ccnet.config file with configuration that you do not want to loose, point the **CCNetInstallPath** property to a temporary directory where all the content will be written and then manually edit and copy the files over to the your CC.NET directory on the server.
- Lines 60-73: We create our own custom values for all the tokens of the dashboard.config.template file and call the **TemplateFile** task again to replace the tokens with our values and store the result in the dashboard.config file for use by the CC.NET web dashboard.

Taking all of the above in account, setting up the build server is now as easy as doing the following:

1. Installing VS 2008
2. Installing SQL Server 2005
3. Installing Subversion
4. Installing CruiseControl.NET
5. Getting the latest version from source control
6. Running the following command:

   ```
   msbuild ccnet.demo.2008.proj /tv:3.5 /t:InstallCCNetConfig
   ```

   OR if you chose to install CC.NET into a custom directory or do not want to overwrite your existing CC.NET configuration

   ```
   msbuild ccnet.demo.2008.proj /tv:3.5 /t:InstallCCNetConfig /p:CCNetInstallPath=”<Your CCNet Install Directory>”
   ```

If you make use of virtualization in your environment, you can easily enough create an image with steps 1-4 installed on, thus making the setup of a build server a quick and effortless task.

Common Build Server Configuration

I am not going to spend time delving into the different CruiseControl.NET configuration options. I'm going to leave that for the reader to further explore. One aspect I would like to highlight though is the ability to use DTD entities to prevent duplication across the different xml configuration elements used by your builds. I use this feature to create the following common entities that are shared between the different CC.NET build configurations.

```
<!DOCTYPE cruisecontrol [
   <!ENTITY pub "<xmllogger />

         <email from='buildserver@yourdomain.co.za' mailhost='smtp.yourdomain.co.za' includeDetails='true'>
            <users>
               <user name='BuildGuru' group='buildmaster' address='buildmaster@yourdomain.co.za'/>
            </users>
            <groups>
               <group name='developers' notification='change'/>
```

```
                  <group name='buildmaster' notification='always'/>
               </groups>
            </email>">

      <!ENTITY links "<externalLinks>
            <externalLink name='Wiki' url='http://yourwikiserver/wiki' />
         </externalLinks>">

      <!ENTITY header "<webURL>http://buildserver.yourdomain.co.za/ccnet</webURL>
         <workingDirectory>C:\Projects\CCNet.Demo.2008</workingDirectory>">

      <!ENTITY svn "<sourcecontrol type='svn'>
           <executable>C:\Program Files\Subversion\bin\svn.exe</executable>
           <trunkUrl>http://svn.yourdomain.co.za/YourProduct/trunk</trunkUrl>
           <workingDirectory>C:\Projects\CCNet.Demo.2008</workingDirectory>
         </sourcecontrol>">

      <!ENTITY svnrevert "<exec>
               <executable>C:\Program Files\Subversion\bin\svn.exe</executable>
               <buildArgs>revert C:\Projects\CCNet.Demo.2008 --recursive</buildArgs>
            </exec>">
   ]>
```

Instead of duplicating these elements, you can now simply reference it by including the entity reference, i.e. **&svnrevert;** or **&svn;** within the different build configurations.

Next Steps

Finally! That takes care of all the common targets used by the build and shows you how to automate the job of setting up your build server. In the next chapter we will take a closer look at all the targets required for a **DeveloperBuild**.

Part 3: DeveloperBuild

This chapter covers the DeveloperBuild and the targets and tasks used by the DeveloperBuild.

MSBuild Scripts

As mentioned in Part 1, the developer should be able to compile and test the code in his/her sandbox using the same targets as the build server. The build targets of the DeveloperBuild are therefore continuously executed by the developers within their sandboxes.

Before delving into these targets, it is important to understand how MSBuild does incremental builds. Please read the documentation here to gain a good understanding of the topic. Other important MSBuild concepts to understand are batching and transforms. All the build targets of the DeveloperBuild make use of incremental builds to build databases and code. The benefit of course is quicker build times.

BuildDatabases

All our databases are scripted into various .sql script files to allow us to create the databases structure and content of any database for a specific revision of the Subversion repository. The script files are stored per database in sub-folders beneath the **Sql** folder. We also have batch files that automate the creation of the database by running these script files using osql/sqlcmd. As mentioned previously, both SQL Server 2000 + 2005 are supported. We also support running the scripts on any dedicated database server machine.

```
<!-- BuildDatabases -->
<Target Name="BuildDatabases">

   <MSBuild Projects="@(SqlProjects)"
            Targets="$(BuildTargets)"
            Properties="Configuration=$(Configuration);Platform=$(Platform);SqlCmdRunner=$(SqlCmdRunner);DBServer=$(DBServer)"/>

</Target>
```

The **SqlProjects** item group contains all the different database projects. The **SqlCmdRunner** (osql/sqlcmd) is used to run against either SQL Server 2000/2005 whilst the **DBServer** property is used to create the database on a separate database server instance. For each database, we have created MSBuild project files that look as follows:

```
<Project DefaultTargets="Build" xmlns="http://schemas.microsoft.com/developer/msbuild/2003">

   <PropertyGroup>
      <SqlCmdRunner Condition=" '$(SqlCmdRunner)' == '' ">sqlcmd</SqlCmdRunner>
      <DBServer Condition=" '$(DBServer)' == '' ">(local)</DBServer>
      <DBName Condition=" '$(DBName)' == '' ">CCNetDemo2008</DBName>
   </PropertyGroup>

   <ItemGroup>
```

```
10       <Sql Include="*.sql" />
11       <Sql Include="Install.cmd" />
12       <SqlOutput Include="$(DBName).DB.LastUpdateSucceeded" />
13    </ItemGroup>
14
15    <Target Name="Build"
16            Inputs="@(Sql)"
17            Outputs="@(SqlOutput)">
18
19       <Exec Command="Install.cmd $(DBName) $(SqlCmdRunner) $(DBServer)"/>
20       <Touch Files="@(SqlOutput)" AlwaysCreate="true"/>
21    </Target>
22
23    <Target Name="Rebuild"
24            DependsOnTargets="Clean;Build"/>
25
26    <Target Name="Clean">
27       <Delete Files="@(SqlOutput)"/>
28    </Target>
29
30 </Project>
```

The project file is really self explanatory. It provides Targets for the standard Clean, Build and ReBuild targets which allow us to compile and build it using the MSBuild task in the same way as any regular .csproj file. Also notice the use of incremental builds at Lines 15-17. All the .sql script files and the batch file are combined into a **Sql** item group and used as inputs for the Build target. This is compared to the **SqlOutput** item group that contains a single CCNetDemo2008.DB.LastUpdateSucceeded file. If the filestamps of the items in the Sql group are older than the .LastUpdateSucceeded file, MSBuild will execute the Build target using its incremental build logic. Line 20 executes the Touch target to set the access and modification time for the .LastUpdateSucceeded file to later than the input files after a successful creation of the database.

BuildCode

```
<!-- Build Code -->
<Target Name="BuildCode">
   <!-- Build the assemblies -->
   <MSBuild Projects="@(CodeProjects)"
            Targets="$(BuildTargets)"
            BuildInParallel="true"
Properties="Environment=$(Environment);Configuration=$(Configuration);Platform=$(Platform);RunCodeAnalysis=$(RunCodeAnalysis);CodeAnalysisRul
es=$(CodeAnalysisRules)">
      <Output TaskParameter="TargetOutputs"
              ItemName="CodeAssemblies"/>
   </MSBuild>

   <!-- Add the compiled code assemblies to master list of all compiled assemblies for the build -->
   <ItemGroup>
      <CompiledAssemblies Include="@(CodeAssemblies)" />
```

```
15
16          <!-- If code analysis was run, create an item collection of the FxCop result files -->
17          <FxCopResults Include ="%(CodeAssemblies.FullPath).$(FxCopResultFile)"
18                        Condition=" '$(RunCodeAnalysis)' == 'true' " />
19       </ItemGroup>
20
21       <Message Text="FxCopResults:$(NEW_LINE)$(TAB)@(CodeAssemblies->'%(RecurseDir)%(FileName)%(Extension).$(FxCopResultFile)',
'$(NEW_LINE)$(TAB)')$(NEW_LINE)"
22                Condition=" '$(RunCodeAnalysis)' == 'true' "
23                Importance="high"/>
24    </Target>
```

Observations:

- Line 4-10: We compile the code projects using the MSBuild task and the same incremental build functionality used by VS 2008. We pass on parameters like the Environment, Configuration, Platform and CodeAnalysisRules to the compiler. The list of assemblies compiled is added to a **CodeAssemblies** and **CompiledAssemblies** item group for later use. The **CompiledAssemblies** group is a master list of all compiled assemblies that will include the test projects. Line 6 illustrates how to make use of the new parallel builds feature of MSBuild to use multiple CPU cores to speed up the compilation of your code.
- Lines 15-22: If code analysis was run, we add the FxCop results to an **FxCopResults** item group. More on this later on in the **CodeStatisticsBuild**.

BuildTests

```
 1    <!-- BuildTests -->
 2    <Target Name="BuildTests">
 3       <!-- Build the assemblies -->
 4       <MSBuild Projects="@(TestProjects)"
 5                Targets="$(BuildTargets)"
 6                BuildInParallel="true"
 7
Properties="Environment=$(Environment);Configuration=$(Configuration);Platform=$(Platform);RunCodeAnalysis=$(RunCodeAnalysis);CodeAnalysisRul
es=$(CodeAnalysisRules)">
 8          <Output TaskParameter="TargetOutputs"
 9                  ItemName="TestAssemblies"/>
10       </MSBuild>
11
12       <!-- Add the compiled test assemblies to master list of all compiled assemblies for the build -->
13       <ItemGroup>
14          <CompiledAssemblies Include="@(TestAssemblies)" />
15
16          <!-- If code analysis was run, create an item collection of the FxCop result files -->
17          <FxCopResults Include ="%(TestAssemblies.FullPath).$(FxCopResultFile)"
18                        Condition=" '$(RunCodeAnalysis)' == 'true' " />
```

```
19        </ItemGroup>
20
21        <Message Text="FxCopResults:$(NEW_LINE)$(TAB)@(TestAssemblies->'%(RecurseDir)%(FileName)%(Extension).$(FxCopResultFile)',
'$(NEW_LINE)$(TAB)')$(NEW_LINE)"
22                 Condition=" '$(RunCodeAnalysis)' == 'true' "
23                 Importance="high"/>
24     </Target>
```

The compile of the test projects is basically a mirror of the BuildCode target. The list of compiled test assemblies are added to the **TestAssemblies** and the master **CompiledAssemblies** list for later use. Lines 17-23 can be ignored for now as it is used only by the **CodeStatisticsBuild**.

BuildAll

```
<Target Name="BuildAll"
        DependsOnTargets="BuildDatabases;BuildCode;BuildTests"/>

```

The BuildAll target just aggregates the different build targets.

Test

```
<!-- Test -->
<Target Name="Test"
        DependsOnTargets="BuildTests">

   <Message Text="$(NEW_LINE)Running Tests for:$(NEW_LINE)$(TAB)@(TestAssemblies->'%(FileName)', '$(NEW_LINE)$(TAB)')$(NEW_LINE)"
            Importance="high"/>

   <TypeMockRegister Company="YourCompany" License="xxxxxxxxxxxxxxxxx" AutoDeploy="$(TypeMockAutoDeploy)"/>
   <TypeMockStart Target="2.0" />

   <!-- For running NUnit
   <NUnit Assemblies="@(TestAssemblies)"
          ToolPath="$(NUnitPath)"
          WorkingDirectory="%(TestAssemblies.RootDir)%(TestAssemblies.Directory)"
          OutputXmlFile="@(TestAssemblies->'%(FullPath).$(TestResultFile).xml')"
          ContinueOnError="true">
      <Output TaskParameter="ExitCode" ItemName="TestExitCodes"/>
   </NUnit>
   -->

   <!-- As the MbUnit MSBuild Task does not provide an ExitCode to test, we run MbUnit from the command line -->
   <Exec Command="$(DOUBLE_QUOTES)$(MbUnitCmd)$(DOUBLE_QUOTES) %(TestAssemblies.Filename)%(TestAssemblies.Extension)
$(MbUnitReportFormat) /verbose /report-name-format:%(TestAssemblies.Filename)%(TestAssemblies.Extension).$(TestResultFile) /report-
folder:$(DOUBLE_QUOTES)%(TestAssemblies.RootDir)%(TestAssemblies.Directory)"
```

```
23            WorkingDirectory="%(TestAssemblies.RootDir)%(TestAssemblies.Directory)"
24            ContinueOnError="true">
25         <Output TaskParameter="ExitCode" ItemName="TestExitCodes"/>
26      </Exec>
27
28      <TypeMockStop Undeploy="$(TypeMockAutoDeploy)"/>
29
30      <!-- Copy the test results for the CCNet build before a possible build failure (see next step) -->
31      <CallTarget Targets="CopyTestResults" />
32
33      <!-- Fail the build if any test failed -->
34      <Error Text="Test error(s) occured" Condition=" '%(TestExitCodes.Identity)' != '0'"/>
35   </Target>
```

Observations:

- Lines 8-9 + Line 28: If you use **TypeMock** as mocking framework you need to initialise and stop the framework before running the tests. As we do not install TypeMock on the build server, we need to make use of its AutoDeploy feature to allow it to run successfully. The **TypeMockAutoDeploy** property is set to true if the build is being run from CC.NET on the build server (i.e. CCNetProject != null) or if explicitly specified via this property from the MSBuild command line.
- Lines 12-18: We use the NUnit task from the **MSBuild.Community.Tasks** to run the tests using NUnit. The **ContinueOnError** property is set to true to allow us to execute all tests. The output (**ExitCode**) for each test run is added to a list of **TestExitCodes**.
- Lines 21-26: If you use MbUnit as your xUnit test framework, we invoke the MbUnit console using the Exec task as the MbUnit MSBuild task does not provide an ExitCode we can use to test for success or failure. The output (**ExitCode**) for each test run is added to a list of **TestExitCodes**.
- Line 31: We need to copy the test results for CC.NET on the build server to merge and display as part of the CC.NET build report.
- Line 34: After running the tests we use the list of exit codes in **TestExitCodes** to fail the build if any of the tests failed.

CopyTestResults

```
<!-- CopyTestResults -->
<Target Name="CopyTestResults"
        Condition=" '$(CCNetProject)' != '' ">

   <!-- Create item collection of test results -->
   <ItemGroup>
      <TestResults Include="%(TestAssemblies.FullPath).$(TestResultFile)*" />
      <ExistingTestResults Include="$(CCNetArtifactDirectory)\*.$(TestResultFile)*" />
   </ItemGroup>

   <Message Text="TestResults:$(NEW_LINE)$(TAB)@(TestResults->'%(FullPath)', '$(NEW_LINE)$(TAB)')$(NEW_LINE)" Importance="low"/>

   <Delete Files="@(ExistingTestResults)"/>
   <Copy SourceFiles="@(TestResults)"
         DestinationFolder="$(CCNetArtifactDirectory)"
```

```
            ContinueOnError="true"/>

    </Target>
```

As we only want to copy the test results if the tests were run on the build server, we test on the **CCNetProject** property to determine if the build was invoked via CC.NET. The **CCNetProject** property is one of the properties populated by CC.NET when using the MSBuild CC.NET task. We create a group for the test results and copy it to the **CCNetArtifactDirectory** on the build server for merging into the build results.

CodeCoverage

```
    <!-- CodeCoverage -->
    <Target Name="CodeCoverage">
       <Message Text="$(NEW_LINE)Running CodeCoverage for:$(NEW_LINE)$(TAB)@(CodeAssemblies->'%(FileName)',
'$(NEW_LINE)$(TAB)')$(NEW_LINE)"
                Importance="high"/>

       <!-- Remove the old coverage files for the projects being analysed -->
       <Delete Files="@(TestAssemblies->'%(FullPath).$(NCoverResultFile)')"/>

       <!-- Register NCover -->
       <Exec Command="regsvr32 /s $(DOUBLE_QUOTES)$(NCoverCoverLib)$(DOUBLE_QUOTES)"
            ContinueOnError="true"/>

       <TypeMockRegister Company="YourCompany" License="xxxxxxxxxxxxxx" AutoDeploy="$(TypeMockAutoDeploy)"/>

       <TypeMockStart Target="2.0"
                     Link="NCover"
                     ProfilerLaunchedFirst="true" />

       <!-- Using NUnit
       <NCover
           ToolPath="$(NCoverPath)"
           CommandLineExe="$(NUnitCmd)"
           CommandLineArgs="$(DOUBLE_QUOTES)%(TestAssemblies.FullPath)$(DOUBLE_QUOTES) /nologo"
           CoverageFile="%(TestAssemblies.FullPath).$(NCoverResultFile)"
           LogLevel="Normal"
           LogFile="%(TestAssemblies.FullPath).$(NCoverLogFile)"
           WorkingDirectory="%(TestAssemblies.RootDir)%(TestAssemblies.Directory)"
           ExcludeAttributes="CoverageExcludeAttribute"
           Assemblies="@(CodeAssemblies)"
           RegisterProfiler="false"
           ContinueOnError="true"/>
        -->

       <!-- Using MbUnit -->
       <NCover
           ToolPath="$(NCoverPath)"
```

```
37           CommandLineExe="$(MbUnitCmd)"
38           CommandLineArgs="$(MbUnitReportFormat) $(DOUBLE_QUOTES)%(TestAssemblies.FullPath)$(DOUBLE_QUOTES) /verbose /report-name-
format:%(TestAssemblies.Filename)%(TestAssemblies.Extension).$(TestResultFile) /report-
folder:$(DOUBLE_QUOTES)%(TestAssemblies.RootDir)%(TestAssemblies.Directory)"
39           CoverageFile="%(TestAssemblies.FullPath).$(NCoverResultFile)"
40           LogLevel="Normal"
41           LogFile="%(TestAssemblies.FullPath).$(NCoverLogFile)"
42           WorkingDirectory="%(TestAssemblies.RootDir)%(TestAssemblies.Directory)"
43           ExcludeAttributes="CoverageExcludeAttribute"
44           Assemblies="@(CodeAssemblies)"
45           RegisterProfiler="false"
46           ContinueOnError="true"/>
47
48       <TypeMockStop Undeploy="$(TypeMockAutoDeploy)"/>
49
50       <!-- Unregister NCover -->
51       <Exec Command="regsvr32 /u /s $(DOUBLE_QUOTES)$(NCoverCoverLib)$(DOUBLE_QUOTES)"
52             ContinueOnError="true"/>
53
54       <!-- Create Item collection of all the coverage results -->
55       <ItemGroup>
56          <NCoverResults Include="%(TestAssemblies.FullPath).$(NCoverResultFile)" />
57       </ItemGroup>
58
59       <Message Text="NCoverResults:$(NEW_LINE)$(TAB)@(NCoverResults->'%(FileName).$(NCoverResultFile)', '$(NEW_LINE)$(TAB)')$(NEW_LINE)"
Importance="low"/>
60
61       <!-- Merge all the coverage results into a single summary Coverage file -->
62       <NCoverExplorer ToolPath="$(NCoverExplorerPath)"
63                       ProjectName="$(ProductName)"
64                       OutputDir="$(Temp)"
65                       Exclusions="$(CoverageExclusions)"
66                       CoverageFiles="@(NCoverResults)"
67                       SatisfactoryCoverage="80"
68                       ReportType="ModuleClassSummary"
69                       HtmlReportName="$(MetricsFolder)\$(NCoverHtmlReport)"
70                       XmlReportName="$(MetricsFolder)\$(NCoverSummaryFile)"
71                       FailMinimum="False"/>
72    </Target>
```

Observations:

- Line 6: Before running the tests, the old **NCover** coverage file for every test project is deleted.
- Lines 10-11 + 51-52: As we do not install NCover on the machine but run it from the Tools folder, we need to register and unregister the NCover CoverLib.dll before and after running the unit tests to ensure that NCover is able to generate the code coverage statistics successfully.

- Line 13-17 + Line 48: As with running the unit tests, if we use **TypeMock** as our mocking framework it requires us to initialise and stop the framework before running the tests to produce the coverage stats. This time we need to link **NCover** as profiler as well using the **TypeMockStart** task.
- Line 20-31 + 35-46: We use the **NCover** MSBuild task and either NUnit/MbUnit from the **NCoverExplorer.Extras** to create the coverage statistics. We set the **RegisterProfiler** property to false to not let the task register the CoverLib.dll used by NCover as we have done this already. Also note the use of the **CoverageExcludeAttribute**. NCover will ignore all code sections from code coverage analysis that have been annotated with this attribute.
- Line 62-71: We use the **NCoverExplorer** MSBuild task from the **NCoverExplorer.Extras** to merge the coverage results and produce a xml summary file and html summary report. Also notice the use of the **CoverageExclusions** item group defined in Part 2 that contain some regular expression patterns to namespaces and class names that NCoverExplorer should ignore when calculating the overall coverage result.

CodeMetrics

NDepend runs against a set of assemblies that are stored within a NDepend project file. The NDepend project file also contains various other settings that NDepend uses to do things like customize the report outputs, resolve assembly references etc. We therefore first use VisualNDepend to create a project file containing all our assemblies that we want to analyze and we store this file within the **CodeMetricsFolder**.

```
1    <!-- CodeMetrics -->
2    <Target Name="CodeMetrics">
3       <Message Text="Running NDepend For:$(MetricsFolder)\$(NDependProjectFile)" Importance="low"/>
4
5       <Exec Command="$(DOUBLE_QUOTES)$(NDependPath)NDepend.Console.exe$(DOUBLE_QUOTES)
$(DOUBLE_QUOTES)$(MetricsFolder)\$(NDependProjectFile)$(DOUBLE_QUOTES)"
6             ContinueOnError="false" />
7    </Target>
```

Observations:

- Line 2: We invoke NDepend using the Exec task passing it the NDepend project file stored in the Code Metrics folder. By default NDepend stores the output in a **NDependOut** sub-folder beneath the folder in which the project file is situated in.

CruiseControl .NET Configuration

Now that we have covered all the targets required for the DeveloperBuild, let's have a quick look at the setup of the DeveloperBuild in the ccnet.config file used by CruiseControl.NET. The configuration is self explanatory if you are familiar with the different CruiseControl.NET configuration options. Note the use of the DTD entities covered in Part 2 to prevent the duplication of CC.NET xml configuration.

```
<project name="DeveloperBuild" queue="YourProduct" queuePriority="1">
    &header;
    <category>Continuous</category>
    <artifactDirectory>C:\Projects\CCNet.Demo.2008\Builds\DeveloperBuild\Artifacts</artifactDirectory>

```

```
   <triggers>
      <intervalTrigger />
   </triggers>

   <state type="state" />

   &svn;

   <labeller type="defaultlabeller"/>

   <tasks>
      <msbuild>
        <executable>C:\WINDOWS\Microsoft.NET\Framework\v3.5\MSBuild.exe</executable>
        <workingDirectory>C:\Projects\CCNet.Demo.2008</workingDirectory>
        <projectFile>CCNet.Demo.2008.proj</projectFile>
        <buildArgs>/noconsolelogger /v:normal /m:2 /tv:3.5 /p:TargetFrameworkVersion=v3.5</buildArgs>
        <targets>BuildAll,Test</targets>
        <timeout>600</timeout> <!-- 10 minutes -->
        <logger>C:\Program Files\CruiseControl.NET\server\ThoughtWorks.CruiseControl.MSBuild.dll</logger>
      </msbuild>
   </tasks>

   <publishers>
      <merge>
         <files>
            <file>C:\Projects\CCNet.Demo.2008\Builds\DeveloperBuild\Artifacts\*.TestResult.xml</file>
         </files>
      </merge>

      <statistics>
         <statisticList>
            <firstMatch name='Revision' xpath='//modifications/modification/changeNumber' />
            <statistic name='TestCount' xpath='sum(//report-result/counter/@run-count)'/>
            <statistic name='TestFailures' xpath='sum(//report-result/counter/@failure-count)'/>
            <statistic name='TestIgnored' xpath='sum(//report-result/counter/@ignore-count)'/>
            <statistic name='TestAsserts' xpath='sum(//report-result/counter/@assert-count)'/>
            <statistic name='TestTime' xpath='sum(//report-result/counter/@duration)'/>
         </statisticList>
      </statistics>

      &pub;
   </publishers>

   &links;
</project>
```

Observations:

- Lines 17-25: The MSBuild CC.Net task is used to launch MSBuild. Of special interest is the different parameters passed onto MSBuild in Line 21. The **/m:2** parameter indicates that we want MSBuild to make use of its new parallel builds feature to execute the build on 2 processors. The **/tv:3.5** parameter indicates that we want to use MSBuild 3.5 and not MSBuild 2.0. The **/p:TargetFrameworkVersion=v3.5** indicates that we want to make use of the multi-targeting features of VS 2008 and that we want to target the .NET 3.5 framework.
- Lines 35-44: I'll cover the statistics gathered here in the last chapter of the Practice section.

Next Steps

Well, that takes care of all the targets used by the DeveloperBuild. The next chapter will highlight all the targets required for a **DeploymentBuild**.

Part 4: DeploymentBuild

This chapter covers the DeploymentBuild and the targets and tasks used by the DeploymentBuild.

MSBuild Scripts

The DeploymentBuild is triggered whenever a Windows installer (.msi package) is required. Before creating the package, the MSDN style documentation is generated from the XML code comments and this is included along with the application in the Windows installer that is distributed to QA.

Before automating the creation of the MSDN style documentation, I used Sandcastle Help File Builder's (SHFB) GUI to create a .shfb project file that contains all the settings required to build the help file. Settings like the assemblies to document, the help file style to use, what member accessibility (public/private/internal/protected) to document, namespaces to document etc. can all be set via the SHFB GUI. I then added the .shfb file to the **\Docs** folder and verified the documentation output by compiling the project using the SHFB GUI.

BuildDocumentation

```
<!-- BuildDocumentation -->
<Target Name="BuildDocumentation"
        DependsOnTargets="BuildCode"
        Condition=" '$(Configuration)' == 'Release' ">

   <Message Text="Building the documentation using Sandcastle" Importance="low"/>

   <!-- Setup Sandcastle on the PC -->
   <Message Text="Extracting Sandcastle onto $(ComputerName)" Importance="low"/>
   <Unzip ZipFileName="$(ToolsFolder)\sandcastle.zip"
          TargetDirectory="$(SandcastlePath)"
          Condition=" !Exists('$(SandcastlePath)')"/>

   <!-- Build source code docs -->
   <Exec Command="$(DOUBLE_QUOTES)$(SandcastleHFBCmd)$(DOUBLE_QUOTES)
$(DOUBLE_QUOTES)$(DocsFolder)\$(SandCastleHFBProject)$(DOUBLE_QUOTES)"
         ContinueOnError="true">
      <Output TaskParameter="ExitCode" PropertyName="SFHBExitCode"/>
   </Exec>

   <!-- Remove Sandcastle from the PC -->
   <Message Text="Removing Sandcastle from $(ComputerName)" Importance="low"/>
   <RemoveDir Condition=" Exists('$(SandcastlePath)') " Directories="$(SandcastlePath)"/>

   <Message Text="SFHBExitCode: $(SFHBExitCode)" Importance="high"/>
   <Error Text="Error occured whilst building the help file" Condition=" '$(SFHBExitCode)' != '0' "/>
</Target>
```

Observations:

- Line 1-3: The target depends on the **BuildCode** target (see Part 3) and only runs if the code is compiled in Release mode. This is required as we only generate the XML code comments as part of doing release builds.
- Lines 9-12: I ran into problems when I added Sandcastle to the Tools folder as SHFB does not ignore the Subversion .svn folder and tries to merge the content of the \icons in the .svn folder as well. I got around the problem by creating a zip archive of the complete installed Sandcastle folder. As only the zip archive is added to Subversion and not the individual Sandcastle folders, we automatically circumvent the .svn folder issue. I then use the **Unzip** task from the MSBuild.Community.Tasks to automatically install Sandcastle by extracting the archive if the Sandcastle directory does not yet exist on the machine.
- Line 15: The SHFB Console runner is called using our settings file to generate the documentation. The output is stored in the **\Docs\Help** folder.
- Line 22: If Sandcastle was installed by the build, we remove if from the PC as soon as the build finishes. This ensures that we always use the latest version of Sandcastle in source control as the zip archive will always be extracted before a documentation run.

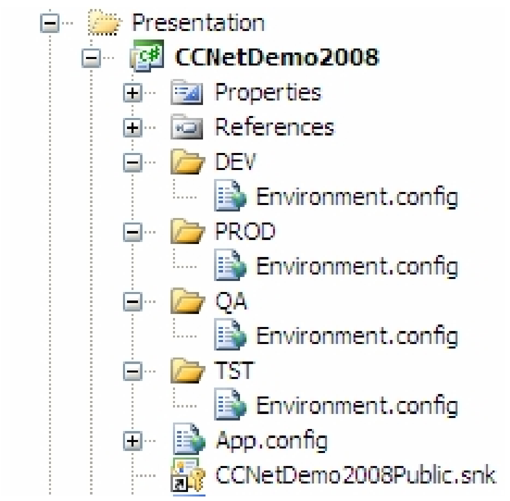
Environment specific builds

We need to support the deployment of our application into different environments (DEV, TST, QA and PROD). Within these environments, we have different configuration settings for our application (e.g. web-service URL's, logging settings etc.) In order to manage these differences, we start by moving all environment specific settings out of our App.Config file into a separate Environment.config file. We then reference this file from within the AppSettings section of our App.Config file:

```
<appSettings file="Environment.config">
   <clear />
   <add key="email.SmtpMailServer" value="smtp.yourdomain" />
   <add key="email.SmtpTimeOut" value="1200000" />
   <add key="email.EnableSSL" value="false" />
</appSettings>
```

Lastly we edit the MSBuild project file of our start-up project to copy the correct version of the Environment.config file to the output directory based on the value set for the global **Environment** property. If you remember correctly, the Environment property is read using the GetEnvironment target covered in Part 2 and passed to the MSBuild task as part of the BuildCode and BuildTests targets covered in Part 3.

Here is the layout of the project within Visual Studio:



Here is an extract from the project file:

```
<ItemGroup>
   <EnvironmentConfigFiles Include="DEV\Environment.config">
      <Env>DEV</Env>
   </EnvironmentConfigFiles>
   <EnvironmentConfigFiles Include="PROD\Environment.config">
      <Env>PROD</Env>
   </EnvironmentConfigFiles>
   <EnvironmentConfigFiles Include="QA\Environment.config">
      <Env>QA</Env>
   </EnvironmentConfigFiles>
   <EnvironmentConfigFiles Include="TST\Environment.config">
      <Env>TST</Env>
   </EnvironmentConfigFiles>
</ItemGroup>
<PropertyGroup>
  <Environment Condition=" '$(Environment)' == '' ">DEV</Environment>
</PropertyGroup>
<Import Project="$(MSBuildToolsPath)\Microsoft.CSharp.targets" />
<!-- To modify your build process, add your task inside one of the targets below and uncomment it.
     Other similar extension points exist, see Microsoft.Common.targets.
<Target Name="BeforeBuild">
</Target>
-->
<Target Name="AfterBuild">
  <!-- Copy the environment specific files -->
```

```
26     <Copy SourceFiles="@(EnvironmentConfigFiles)" DestinationFolder="$(OutputPath)" Condition=" '%(EnvironmentConfigFiles.Env)' ==
'$(Environment)' " />
27   </Target>
```

Observations:

- Lines 1-17: We create an item group **EnvironmentConfigFiles** with environment specific custom meta-data. We use this meta-data (see Line 26) to figure out what environment files to copy by matching it against the value of the **Environment** property.

Version

```
1     <!-- Version -->
2     <Target Name="Version"
3             DependsOnTargets="GetIterationNumber;GetRevisionNumber">
4        <PropertyGroup>
5           <AppVersion>$(Major).$(Minor).$(IterationNumber).$(Revision)</AppVersion>
6        </PropertyGroup>
7
8        <Message Text="AppVersion number generated: $(AppVersion)" Importance="high"/>
9
10       <ItemGroup>
11          <AssemblyInfoTokens Include="AssemblyVersion">
12             <ReplacementValue>$(AppVersion)</ReplacementValue>
13          </AssemblyInfoTokens>
14          <AssemblyInfoTokens Include="AssemblyConfiguration">
15             <ReplacementValue>$(Configuration)</ReplacementValue>
16          </AssemblyInfoTokens>
17          <AssemblyInfoTokens Include="Company">
18             <ReplacementValue>$(Company)</ReplacementValue>
19          </AssemblyInfoTokens>
20          <AssemblyInfoTokens Include="Product">
21             <ReplacementValue>$(ProductName)</ReplacementValue>
22          </AssemblyInfoTokens>
23       </ItemGroup>
24
25       <TemplateFile Template="$(SrcFolder)\GlobalAssemblyInfo.cs.template"
26                     OutputFilename="$(SrcFolder)\GlobalAssemblyInfo.cs"
27                     Tokens="@(AssemblyInfoTokens)"/>
28    </Target>
```

Observations:

- Line 2: The target depends on the **GetIterationNumber** and **GetRevisionNumber** targets described in Part 2.
- Line 5: We create an **AppVersion** property in the format Major.Minor.IterationNumber.Revision.
- Lines 10-23: We create an **AssemblyInfoTokens** item group containing the values for the tokens within our GlobalAssemblyInfo.cs.template file.

- Lines 25-26: We use the **TemplateFile** task from the MSBuild.Community.Tasks to create a GlobalAssemblyInfo.cs file that is linked to all our .csproj files.

Here is the content of the GlobalAssemblyInfo.cs.template file:

```
using System;
using System.Diagnostics.CodeAnalysis;
using System.Reflection;

[assembly: AssemblyConfiguration("${AssemblyConfiguration}")]
[assembly: AssemblyCompany("${Company}")]
[assembly: AssemblyProduct("${Product}")]
[assembly: AssemblyCopyright("Copyright (c) 2007 ${Company}")]
[assembly: CLSCompliant(true)]
[assembly: AssemblyVersion("${AssemblyVersion}")]
[assembly: AssemblyFileVersion("${AssemblyVersion}")]

[module: SuppressMessage("Microsoft.Design", "CA2210:AssembliesShouldHaveValidStrongNames")]

[SuppressMessage("Microsoft.Design", "CA1050:DeclareTypesInNamespaces", Justification = "To support coverage exclusions using TestDriven.NET we have to put it in without a namespace")]
[AttributeUsage(AttributeTargets.All)]
public sealed class CoverageExcludeAttribute : Attribute
{
}
```

BackupDatabases

```
1    <!-- BackupDatabases -->
2    <Target Name="BackupDatabases">
3       <Error Text="No DBBackupFolder has been specified" Condition="'$(DBBackupFolder)' == ''" />
4
5       <Exec Command="$(DOUBLE_QUOTES)$(DBBackupScript)$(DOUBLE_QUOTES) $(SqlCmdRunner) $(DBServer)"
6             WorkingDirectory="$(SqlFolder)"/>
7
8       <ItemGroup>
9          <DBBackupFiles Include="$(DBBackupFolder)\$(ProductName).bak" />
10      </ItemGroup>
11   </Target>
```

Observations:

- Lines 5-6: A batch file is executed to create a backup of all the databases for inclusion in the .msi package.
- Line 9: All the database backup files are added to a **DBBackupFiles** group for later use.

Sign

```
1     <!-- Sign -->
2     <Target Name="Sign">
3        <ItemGroup>
4           <SignInputs Include="$(BinFolder)\$(ProductName)*.dll;
5                                $(BinFolder)\$(ProductName)*.exe;
6                                $(BinFolder)\**\$(ProductName)*.resources.dll"
7                       Exclude="*.LastUpdateSucceeded;
8                                $(BinFolder)\*vshost.exe;" />
9        </ItemGroup>
10
11       <Message Text="Sign inputs:$(NEW_LINE)$(TAB)@(SignInputs->'%(FullPath)', '$(NEW_LINE)$(TAB)')$(NEW_LINE)"
12                Importance="high"/>
13
14       <!-- Sign the assemblies and resources -->
15       <Exec Command="$(DOUBLE_QUOTES)$(FrameworkSdkPath)Bin\sn.exe$(DOUBLE_QUOTES) -Ra
$(DOUBLE_QUOTES)%(SignInputs.FullPath)$(DOUBLE_QUOTES) $(DOUBLE_QUOTES)$(KeyFile)$(DOUBLE_QUOTES)"/>
16    </Target>
```

Observations:

- Lines 4-8: We create an item group consisting out of our assemblies and resource files to sign.
- Line 15: As we use delay signing in our development sandboxes, we resign our assemblies before deployment with our private key.

Package

```
1     <!-- Package -->
2     <Target Name="Package"
3             DependsOnTargets="BackupDatabases;Sign">
4
5        <ItemGroup>
6           <InstallerInputs Include=
7                            "$(BinFolder)\$(ProductName)*.dll;
8                             $(BinFolder)\$(ProductName)*.exe;
9                             $(BinFolder)\**\$(ProductName)*.resources.dll;
10                            $(LibFolder)\*.dll;
11                            $(DocsFolder)\Help\*.chm"
12                           Exclude=
13                            "$(BinFolder)\*.$(TestResultFile)*;
14                             $(BinFolder)\*.$(FxCopResultFile);
15                             $(BinFolder)\*.$(LastCodeAnalysisSucceededFile);
16                             $(BinFolder)\*.$(NCoverResultFile);
17                             $(BinFolder)\*.$(NCoverLogFile);
18                             $(LibFolder)\**\.svn\**" />
19
20          <OldInstallerInputs Include="$(InstallBinFolder)\**\*.*"
```

```
21                               Exclude="$(InstallBinFolder)\**\.svn\**"/>
22        </ItemGroup>
23
24        <Delete Files="@(OldInstallerInputs)"/>
25
26        <Copy SourceFiles="@(InstallerInputs)"
27              DestinationFiles="@(InstallerInputs->'$(InstallBinFolder)\%(RecursiveDir)%(FileName)%(Extension)')" />
28        <Copy SourceFiles="@(DBBackupFiles)"
29              DestinationFiles="@(DBBackupFiles->'$(InstallBinFolder)\%(RecursiveDir)%(FileName)%(Extension)')" />
30
31        <MSBuild Projects="@(InstallProjects)"
32                 Targets="$(BuildTargets)"
33
Properties="Configuration=$(Configuration);DefineSolutionProperties=false;WixTasksPath=$(WixTasksPath);WixTargetsPath=$(WixTargetsPath);WixTo
olPath=$(WixToolPath)">
34           <Output TaskParameter="TargetOutputs"
35                   ItemName="InstallerOutputs"/>
36        </MSBuild>
37
38        <Message Text="Installer Outputs:$(NEW_LINE)$(TAB)@(InstallerOutputs->'%(FullPath)', '$(NEW_LINE)$(TAB)')$(NEW_LINE)"
39                 Importance="high"/>
40     </Target>
```

Observations:

- Lines 6-18: We create an **InstallerInputFiles** item group that contains all the files required for building our Windows installer from the various folders.
- Lines 20-24: We create an **OldInstallerInputFiles** item group that contains the previous installation files where after we delete these files.
- Lines 26-29: The new installation files and database backups are copied to the working directory used by our WiX install project.
- Line 31: As a WiX project file is in MSBuild format, we create our installer by simply using the MSBuild task to compile the project file. The outputs of the build are stored in an **InstallerOutputs** item group for later use.

Deploy

```
1      <!-- Deploy -->
2      <Target Name="Deploy"
3              DependsOnTargets="Package">
4
5         <Error Text="No DeploymentFolder has been specified" Condition=" '$(DeploymentFolder)' == '' " />
6         <Error Text="No AppVersion has been specified" Condition=" '$(AppVersion)' == '' " />
7
8         <PropertyGroup>
9            <LatestVersionFolder>$(DeploymentFolder)\$(AppVersion)</LatestVersionFolder>
10        </PropertyGroup>
11
12        <Copy SourceFiles="@(InstallerOutputs)"
```

```
13             DestinationFolder="$(LatestVersionFolder)"/>
14
15       <Message Text="Install files:$(NEW_LINE)$(TAB)@(InstallerOutputs->'$(LatestVersionFolder)\%(FileName)',
'$(NEW_LINE)$(TAB)')$(NEW_LINE)"
16                Importance="high"/>
17    </Target>
```

Observations:

- Lines 5-6: We fail the build if no **AppVersion** or **DeploymentFolder** has been specified.
- Line 9: The installer location is stored in a **LatestVersionFolder** property for later use.
- Lines 12-13: We copy the Installer outputs to the LatestVersionFolder.

CreateBuildMail

```
1    <!-- CreateBuildMail -->
2    <Target Name="CreateBuildMail"
3            DependsOnTargets="Deploy">
4
5       <!-- Create the tokens for the InstallBuildEmail template -->
6       <ItemGroup>
7          <EmailTokens Include="AppVersion">
8             <ReplacementValue>$(AppVersion)</ReplacementValue>
9          </EmailTokens>
10         <EmailTokens Include="LatestVersionFolder">
11            <ReplacementValue>$(LatestVersionFolder)</ReplacementValue>
12         </EmailTokens>
13      </ItemGroup>
14
15      <!-- Create the InstallBuildEmail by replacing the tokens in the template file -->
16      <TemplateFile Template="$(InstallBuildEmailTemplate)"
17                    OutputFilename="$(InstallBuildEmailFile)"
18                    Tokens="@(EmailTokens)" />
19
20      <ReadLinesFromFile File="$(InstallBuildEmailFile)">
21         <Output TaskParameter="Lines" ItemName="EmailBody"/>
22      </ReadLinesFromFile>
23
24   </Target>
```

Observations:

- Lines 6-13: An **EmailTokens** item group is created to use to create an automatic install mail from a template file.
- Lines 16-18: The **TemplateFile** task from the MSBuild.Community.Tasks is used to replace the tokens within the mail template file with the values created within the **EmailTokens** item group. The completed mail template output is written to a temporary file.

- Lines 20-2: The contents of the completed mail template file are read into an **EmailBody** item for later use.

Here is the content of the InstallBuildEmailTemplate.htm file:

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN"><HTML><HEAD><META http-equiv=Content-Type content="text/html; charset=us-ascii"><META content="MSHTML 6.00.5730.11" name=GENERATOR></HEAD><BODY><DIV><FONT face="Trebuchet MS" size=2><SPAN class=710153311-12032007>Build ${AppVersion} is ready for installation.&nbsp; You can install it from <A href="file://${LatestVersionFolder}">this location</A>.</SPAN></FONT></DIV><DIV><FONT face="Trebuchet MS" size=2></FONT>&nbsp;</DIV></BODY></HTML>
```

SendBuildMail

```
1     <!-- SendBuildMail -->
2     <Target Name="SendBuildMail"
3             DependsOnTargets="CreateBuildMail">
4
5        <!-- Send e-mail notification of the new build -->
6        <Mail SmtpServer="$(SmtpServer)"
7              To="@(Developers)"
8              Priority="High"
9              IsBodyHtml="true"
10             From="$(ComputerName)@$(Domain)"
11             Subject="$(ProductName) Build $(AppVersion) is ready for testing."
12             Body="@(EmailBody)"
13             ContinueOnError="true"/>
14
15    </Target>
```

Observations:

- Line 3: The CreateBuildMail target will always first be executed as a dependency.
- Lines 6-13: We use the **Mail** task from the MSBuild.Community.Tasks to notify the developers and QA of the new build by sending them the template e-mail created in the CreateBuildMail target.

CruiseControl.NET Configuration

The build server is set use the following CC.NET configuration. The configuration is really self explanatory if you are familiar with the different CruiseControl.NET configuration options.

```
1    <project name="DeploymentBuild" queue="YourProduct" queuePriority="2">
2        &header;
3        <category>Deployment</category>
4        <artifactDirectory>C:\Projects\CCNet.Demo.2008\Builds\DeploymentBuild\Artifacts</artifactDirectory>
5
6        <triggers />
7
8        &svn;
```

```
 9
10       <state type="state" />
11
12       <labeller type="defaultlabeller"/>
13
14       <!-- Remove the log files from the previous build -->
15       <prebuild>
16          <exec>
17             <executable>clean.bat</executable>
18             <baseDirectory>C:\Projects\CCNet.Demo.2008\Builds\DeploymentBuild</baseDirectory>
19          </exec>
20       </prebuild>
21
22       <tasks>
23          <msbuild>
24             <executable>C:\WINDOWS\Microsoft.NET\Framework\v3.5\MSBuild.exe</executable>
25             <workingDirectory>C:\Projects\CCNet.Demo.2008</workingDirectory>
26             <projectFile>CCNet.Demo.2008.proj</projectFile>
27             <buildArgs>/noconsolelogger /v:normal /m:2 /tv:3.5
/p:TargetFrameworkVersion=v3.5;Configuration=Release;RunCodeAnalysis=false;SvnUserName=fred;SvnPassword=password</buildArgs>
28
<targets>Version,BuildCode,Sign,BuildDatabases,BuildDocumentation,BackupDatabases,Package,Deploy,SendBuildMail,CommitChanges,TagRepository</t
argets>
29             <timeout>600</timeout> <!-- 10 minutes -->
30             <logger>C:\Program Files\CruiseControl.NET\server\ThoughtWorks.CruiseControl.MSBuild.dll</logger>
31          </msbuild>
32       </tasks>
33
34       <publishers>
35          <statistics>
36             <statisticList>
37                <firstMatch name='Revision' xpath='//modifications/modification/changeNumber' />
38             </statisticList>
39          </statistics>
40
41          <!-- Revert changes that may still exist because the build failed -->
42          &svnrevert;
43
44          &pub;
45       </publishers>
46
47       &links;
48    </project>
```

Observations:

- Lines 6: The build is set to be forced manually.

- Lines 23-31: The MSBuild CC.Net task is used to launch MSBuild. We instruct MSBuild to make use of its new parallel builds feature to execute the build on 2 processors and to target the .NET 3.5 Framework.
- Line 42: To cater for scenarios where the build fails, we have to revert all the changes made on the build server. If the build ran through successfully, the effect of this revert will be nothing as there won't be any changes left to revert due to the successful execution of the CommitChanges target (see line 28).

Next Steps

Well, that takes care of all the targets used by the DeploymentBuild. The chapter will highlight all the targets required for the **CodeStatisticsBuild**.

Part 5: CodeStatisticsBuild

This chapter covers the CodeStatisticsBuild and the targets and tasks used by the CodeStatisticsBuild.

MSBuild Scripts

The CodeStatisticsBuild happens every morning at 03:00 am when there is no other activity on the build server. The purpose of the CodeStatisticsBuild is to generate metrics on the code base that we use to spot daily trends and identify areas of concern. We generate metrics using FxCop, NCover and NDepend.

The CodeStatisticsBuild is quite simple as it re-uses most of the targets that have been covered in Part 3: DeveloperBuild. To run FxCop, the **BuildAll** target of the DeveloperBuild is invoked with the **RunCodeAnalysis** property set to True. To run NCover, the **CodeCoverage** target of the DeveloperBuild is invoked. To run NDepend, the **CodeMetrics** target of the DeveloperBuild is invoked. The only additional tasks required for the CodeStatisticsBuild are tasks to copy the metrics to the **CCNetArtifactDirectory** to allow CC.NET to merge the results into the build report.

For people interested in using the standalone version of FxCop instead of VS 2008 Managed Code Analysis, I also include a **FxCop** target that generates the FxCop metrics using the standalone version of FxCop 1.36 stored in the Tools folder. Please note that there are differences between running standalone FxCop and Managed Code Analysis so the metrics generated will differ a bit.

FxCop

```
<!—CodeAnalysis using Standalone FxCop -->
<Target Name="CodeAnalysis"
        DependsOnTargets="BuildCode;BuildTests">

   <PropertyGroup>
      <FxCopDir>$(ToolsFolder)\FxCop</FxCopDir>
   </PropertyGroup>

   <ItemGroup>
      <FxCopRuleExclusions Include="$(CodeAnalysisRules)"/>
      <FxCopDirectories Include=
                      "$(DOUBLE_QUOTES)$(LibFolder)$(DOUBLE_QUOTES);
                       $(DOUBLE_QUOTES)$(ProgramFiles)\Reference Assemblies\Microsoft\Framework\v3.0$(DOUBLE_QUOTES);
                       $(DOUBLE_QUOTES)$(ProgramFiles)\Reference Assemblies\Microsoft\Framework\v3.5$(DOUBLE_QUOTES)"/>
   </ItemGroup>

   <Message Text="Running FxCop for$(NEW_LINE)$(TAB)@(CompiledAssemblies->'%(RecurseDir)%(FileName)%(Extension)',
'$(NEW_LINE)$(TAB)')$(NEW_LINE)" Importance="high"/>

   <!-- Run FxCop using FxCopCmd.exe and the same options as VS uses when running managed code analysis -->
   <Exec Command="$(DOUBLE_QUOTES)$(FxCopDir)\FxCopCmd.exe$(DOUBLE_QUOTES)
/out:$(DOUBLE_QUOTES)%(CompiledAssemblies.FullPath).$(FxCopResultFile)$(DOUBLE_QUOTES)
/file:$(DOUBLE_QUOTES)%(CompiledAssemblies.FullPath)$(DOUBLE_QUOTES) /ruleid:@(FxCopRuleExclusions->'%(Identity)', ' /ruleid:')
```

```
/directory:@(FxCopDirectories->'%(Identity)', ' /directory:') /rule:$(DOUBLE_QUOTES)$(FxCopDir)\Rules$(DOUBLE_QUOTES) /searchgac
/ignoreinvalidtargets /forceoutput /successfile /ignoregeneratedcode /saveMessagesToReport:Active /timeout:120"
21              ContinueOnError="true"/>
22
23        <ItemGroup>
24           <FxCopResults Include ="%(CompiledAssemblies.FullPath).$(FxCopResultFile)" />
25        </ItemGroup>
26
27        <Message Text="FxCopResults:$(NEW_LINE)$(TAB)@(CompiledAssemblies->'%(RecurseDir)%(FileName)%(Extension).$(FxCopResultFile)',
'$(NEW_LINE)$(TAB)')$(NEW_LINE)"
28                 Importance="high"/>
29     </Target>
```

Observations:

- Line 5-15: We create the necessary configuration settings for running standalone FxCop. This includes the rules to exclude and the reference directories to use for resolving assembly references.
- Line 20: We invoke FxCopCmd.exe using the same options that Managed Code Analysis uses when running from within VS 2008. The individual results of every assembly evaluated are stored in the same file used by Managed Code Analysis.
- Line 23: We create a **FxCopResults** item group containing the FxCop results for all the individual assemblies evaluated.

CopyFxCopResults

If you look at the **BuildCode** and **BuildTests** targets covered in Part 3 or the **FxCop** target just covered, you will notice that the FxCop results are added to a **FxCopResults** item group. The only thing left to do is therefore to copy these results for CC.NET to include in the build report.

```
1      <!-- CopyFxCopResults -->
2      <Target Name="CopyFxCopResults"
3              Condition=" '$(CCNetProject)' != '' ">
4
5         <Message Text="FxCopResults:$(NEW_LINE)$(TAB)@(FxCopResults->'%(Identity)', '$(NEW_LINE)$(TAB)')$(NEW_LINE)" Importance="low"/>
6
7         <ItemGroup>
8            <ExistingFxCopResults Include="$(CCNetArtifactDirectory)\*.$(FxCopResultFile)" />
9         </ItemGroup>
10
11        <Delete Files="@(ExistingFxCopResults)"/>
12        <Copy SourceFiles="@(FxCopResults)"
13              DestinationFolder="$(CCNetArtifactDirectory)"
14              ContinueOnError="true"/>
15
16     </Target>
```

Observations:

- Line 2: We test on the **CCNetProject** property to determine if the build was invoked via CC.NET on the build server. The **CCNetProject** property is one of the properties created and passed on by CC.NET when using the MSBuild CC.NET task.
- Lines 7-11: The existing FxCop results are deleted.
- Line 12-14: The new set of FxCop results are copied to the **CCNetArtifactDirectory** for inclusion into the build report.

CopyCoverageResults

If you look at the **CodeCoverage** target covered in Part 3, you will see notice that the NCover results are merged into a single **NCoverSummaryFile**. The only thing left to do is therefore to copy this file for CC.NET to include in the build report.

```
1    <!-- CopyCodeCoverageResults -->
2    <Target Name="CopyCodeCoverageResults"
3            Condition=" '$(CCNetProject)' != '' ">
4
5       <ItemGroup>
6          <ExistingCoverageResults Include="$(CCNetArtifactDirectory)\$(NCoverSummaryFile)" />
7       </ItemGroup>
8
9       <Delete Files="@(ExistingCoverageResults)"/>
10      <Copy SourceFiles="$(MetricsFolder)\$(NCoverSummaryFile)"
11            DestinationFolder="$(CCNetArtifactDirectory)"
12            ContinueOnError="true"/>
13
14   </Target>
```

Observations:

- Line 2: We test on the **CCNetProject** property to determine if the build was invoked via CC.NET on the build server.
- Lines 5-9: The existing NCover summary file is deleted.
- Line 10-12: The new NCover summary file is copied to the **CCNetArtifactDirectory** for inclusion into the build report.

CopyCodeMetricsResults

If you look at the **CodeMetrics** target covered in Part 3, you will see notice that the NDepend results are created in a **NDependOutFolder**. Of these results, we are only interested in the **NDependResultsFile** as it contains the combined metrics for the NDepend run. Unfortunately CC.NET currently is unable to display images as part of the web dashboard so the graphs created by NDepend like **Abstractness vs. Instability** cannot be displayed. This will hopefully be fixed in a future version of CC.NET. In the mean time, you can use this workaround if you absolutely want to include the images into your reports. I have opted to exclude images for now.

```
1    <!-- CopyCodeMetricsResults -->
2    <Target Name="CopyCodeMetricsResults"
3            Condition=" '$(CCNetProject)' != '' ">
4
```

```
5      <ItemGroup>
6         <ExistingMetrics Include="$(CCNetArtifactDirectory)\NDepend\$(NDependResultFile)" />
7      </ItemGroup>
8
9      <Delete Files="@(ExistingMetrics)"/>
10     <Copy SourceFiles="$(NDependOutputPath)\$(NDependResultFile)"
11           DestinationFolder="$(CCNetArtifactDirectory)\NDepend"
12           ContinueOnError="true"/>
13
14  </Target>
```

Observations:

- Line 2: We test on the **CCNetProject** property to determine if the build was invoked via CC.NET on the build server.
- Lines 5-9: The existing NDepend results file is deleted.
- Line 10-12: The new NDepend results file is copied to the **CCNetArtifactDirectory** for inclusion into the build report.

CruiseControl.NET Configuration

The build server is set use the following CC.NET configuration. The configuration is really self explanatory if you are familiar with the different CruiseControl.NET configuration options.

```
1   <project name="CodeStatisticsBuild" queue="YourProduct" queuePriority="3">
2      &header;
3      <category>CodeStatistics</category>
4      <artifactDirectory>C:\Projects\CCNet.Demo.2008\Builds\CodeStatisticsBuild\Artifacts</artifactDirectory>
5
6      <triggers>
7         <scheduleTrigger time="03:00" buildCondition="ForceBuild" />
8      </triggers>
9
10     &svn;
11
12     <state type="state" />
13
14     <labeller type="defaultlabeller"/>
15
16     <!-- Remove the log files from the previous build as we are creating a complete new build -->
17     <prebuild>
18        <exec>
19           <executable>clean.bat</executable>
20           <baseDirectory>C:\Projects\CCNet.Demo.2008\Builds\CodeStatisticsBuild</baseDirectory>
21        </exec>
22     </prebuild>
23
```

```
24       <tasks>
25          <msbuild>
26             <executable>C:\WINDOWS\Microsoft.NET\Framework\v3.5\MSBuild.exe</executable>
27             <workingDirectory>C:\Projects\CCNet.Demo.2008</workingDirectory>
28             <projectFile>CCNet.Demo.2008.proj</projectFile>
29             <buildArgs>/noconsolelogger /v:normal /m:2 /tv:3.5
/p:TargetFrameworkVersion=v3.5;Configuration=Release;RunCodeAnalysis=true;DocumentationFile=</buildArgs>
30             <targets>BuildAll,CopyFxCopResults,CodeCoverage,CopyCodeCoverageResults,CodeMetrics,CopyCodeMetricsResults</targets>
31             <timeout>1800</timeout>  <!-- 30 minutes -->
32             <logger>C:\Program Files\CruiseControl.NET\server\ThoughtWorks.CruiseControl.MSBuild.dll</logger>
33          </msbuild>
34       </tasks>
35
36       <publishers>
37          <merge>
38             <files>
39                <file>C:\Projects\CCNet.Demo.2008\Builds\CodeStatisticsBuild\Artifacts\*.CodeAnalysisLog.xml</file>
40                <file>C:\Projects\CCNet.Demo.2008\Builds\CodeStatisticsBuild\Artifacts\CoverageSummary.xml</file>
41                <file>C:\Projects\CCNet.Demo.2008\Builds\CodeStatisticsBuild\Artifacts\NDepend\*.xml</file>
42             </files>
43          </merge>
44
45          <statistics>
46             <statisticList>
47                <firstMatch name="Revision" xpath="//modifications/modification/changeNumber" />
48                <firstMatch name="Coverage" xpath="//coverageReport/project/@coverage" />
49                <firstMatch name="ILInstructions" xpath="//ApplicationMetrics/@NILInstruction" />
50                <firstMatch name="LinesOfCode" xpath="//ApplicationMetrics/@NbLinesOfCode" />
51                <firstMatch name="LinesOfComment" xpath="//ApplicationMetrics/@NbLinesOfComment" />
52                <statistic name='FxCop Warnings'
xpath="count(//FxCopReport/Targets/Target/Modules/Module/Namespaces/Namespace/Types/Type/Members/Member/Messages/Message/Issue[@Level='Warnin
g'])" />
53                <statistic name='FxCop Errors'
xpath="count(//FxCopReport/Targets/Target/Modules/Module/Namespaces/Namespace/Types/Type/Members/Member/Messages/Message/Issue[@Level='Critic
alError'])" />
54             </statisticList>
55          </statistics>
56
57          &pub;
58       </publishers>
59
60       &links;
61    </project>
```

Observations:

- Lines 7: The build is set to trigger every morning at 03:00 am.

- Lines 17-22: Before starting the build, the previous build's log files are removed
- Lines 25-33: The MSBuild CC.Net task is used to launch MSBuild. We instruct MSBuild to make use of its new parallel builds feature to execute the build on 2 processors and to target the .NET 3.5 Framework.
- Lines 37-43: The NCover, FxCop and NDepend results are merged into the build results
- Lines 45-55: The Statistics publisher is used to publish some additional metrics - see the next chapter for more details.

Next Steps

Finally! That takes care of all the targets used by the CodeStatisticsBuild. The last chapter will highlight some community extensions that you can use to add some further panache to your CC.NET CI builds.

Extending Your Build Using Community Extensions

CruiseControl.NET 1.1 introduced a very nice feature called the Statistics Publisher that you can use to collect and update statistics for each build. The statistics generated can be very useful for spotting trends in your code base. If you include the Statistics publisher in your CC.NET configuration (as we did), the statistics can be viewed via the same web dashboard that you use to browse your CC.NET build results. You will find an additional link for every project called **View Statistics** that links to a web page containing your build statistics for every build of the project you are browsing.

Out-of-the-box statistics include counters like the number of NUnit tests passed/failed/ignored; FxCop warnings/errors; build times etc. The statistics shown have limited xml file configurability, but you can add new counters based on *xpath expressions* to the data contained in your build logs. To include the NCover and NDepend metrics as well as the Subversion revision number, I added the following statistics to our ccnet.config file for our CodeStatisticsBuild:

```
<statistics>
   <statisticList>
      <firstMatch name="Revision" xpath="//modifications/modification/changeNumber" />
      <firstMatch name="Coverage" xpath="//coverageReport/project/@coverage" />
      <firstMatch name="ILInstructions" xpath="//ApplicationMetrics/@NILInstruction" />
      <firstMatch name="LinesOfCode" xpath="//ApplicationMetrics/@NbLinesOfCode" />
      <firstMatch name="LinesOfComment" xpath="//ApplicationMetrics/@NbLinesOfComment" />
   </statisticList>
</statistics>

```

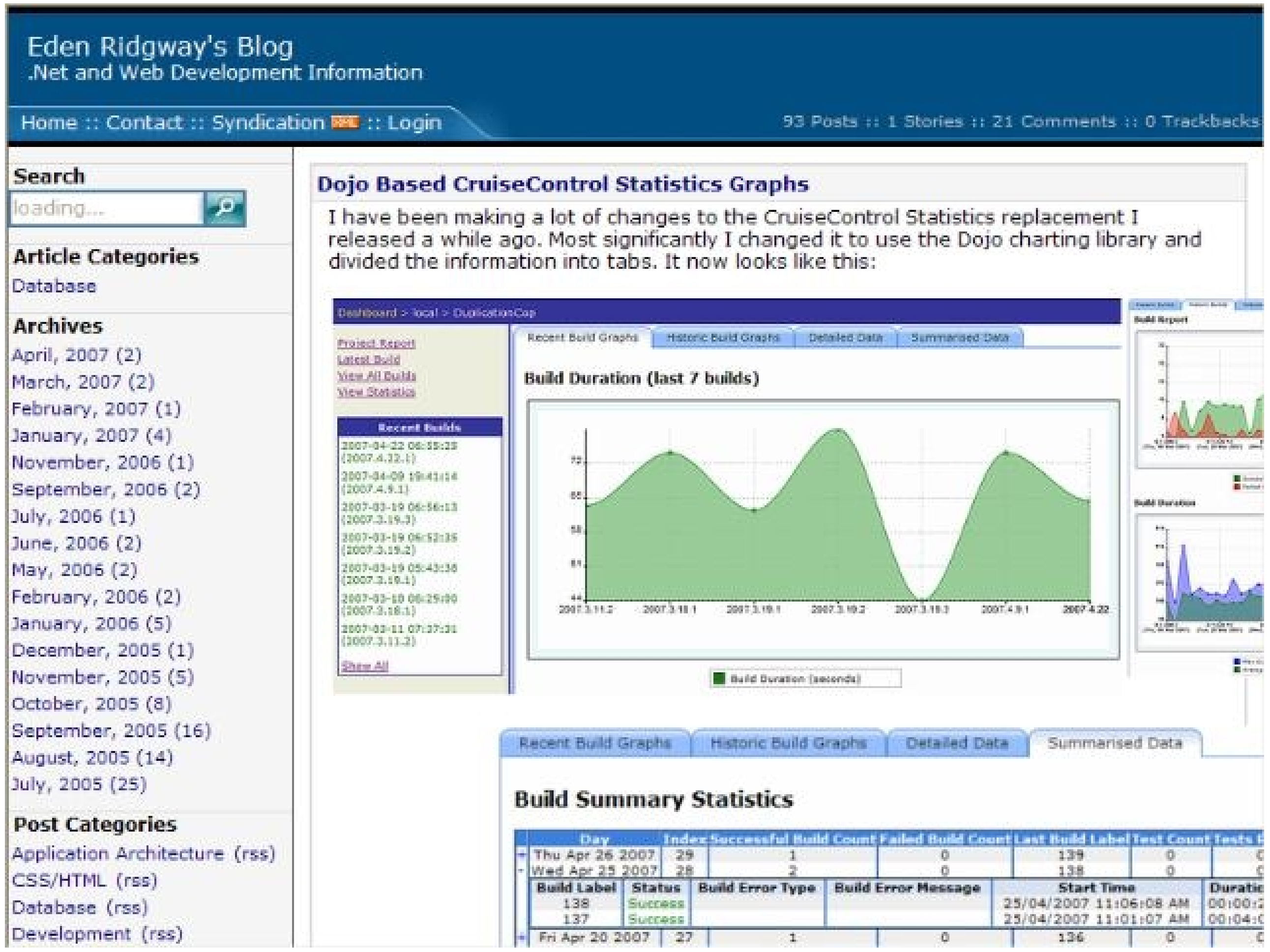
If you are using MbUnit as a test framework and you want to include the MbUnit pass/fail/ignore count, here is the snippet that you need to add to the ccnet.config:

```
<statistics>
   <statisticList>
      <statistic name='TestCount' xpath='sum(//report-result/counter/@run-count)'/>
      <statistic name='TestFailures' xpath='sum(//report-result/counter/@failure-count)'/>
      <statistic name='TestIgnored' xpath='sum(//report-result/counter/@ignore-count)'/>
      <statistic name='TestAsserts' xpath='sum(//report-result/counter/@assert-count)'/>
      <statistic name='TestTime' xpath='sum(//report-result/counter/@duration)'/>
  </statisticList>
</statistics>

```

Since its release the CC.NET community has cottoned onto the feature and there has been one or two very good utilities/extensions written for the statistics publisher.

1. The first extension is CCStatistics first created by Grant Drake, the author of the excellent NCoverExplorer, and later updated by Damon Carr to support CC.NET 1.3. CCStatistics is an application that will parse all your existing build logs and recreate the statistics. This is useful if you add some additional counters at a later stage and you wish to have your historical build logs also include the new counters.
2. The second extension is some very cool graphs for the different statistics created by Eden Ridgway. The old adage "A picture says a thousand words" is so true.

Source: http://www.ridgway.co.za/archive/2007/04/22/dojo-based-cruisecontrol-statistics-graphs.aspx



Eden Ridgway's Blog
.Net and Web Development Information

Home :: Contact :: Syndication :: Login   93 Posts :: 1 Stories :: 21 Comments :: 0 Trackbacks

Search

loading...

Article Categories

Database

Archives

April, 2007 (2)
March, 2007 (2)
February, 2007 (1)
January, 2007 (4)
November, 2006 (1)
September, 2006 (2)
July, 2006 (1)
June, 2006 (2)
May, 2006 (2)
February, 2006 (2)
January, 2006 (5)
December, 2005 (1)
November, 2005 (5)
October, 2005 (8)
September, 2005 (16)
August, 2005 (14)
July, 2005 (25)

Post Categories

Application Architecture (rss)
CSS/HTML (rss)
Database (rss)
Development (rss)

Dojo Based CruiseControl Statistics Graphs

I have been making a lot of changes to the CruiseControl Statistics replacement I released a while ago. Most significantly I changed it to use the Dojo charting library and divided the information into tabs. It now looks like this:

Recent Build Graphs | Historic Build Graphs | Detailed Data | Summarised Data

Build Summary Statistics

	Day	Index	Successful Build Count	Failed Build Count	Last Build Label	Test Count	Tests
+	Thu Apr 26 2007	29	1	0	139	0	0
-	Wed Apr 25 2007	28	2	0	138	0	0

Build Label	Status	Build Error Type	Build Error Message	Start Time	Duration
138	Success			25/04/2007 11:06:08 AM	00:00:2
137	Success			25/04/2007 11:01:07 AM	00:04:0

	Day	Index	Successful Build Count	Failed Build Count	Last Build Label	Test Count	Tests
+	Fri Apr 20 2007	27	1	0	136	0	0

I trust that you will find these extensions useful.

Final Remarks

Well, that concludes the guide. I hope you find it to be a useful resource for assisting you with creating your own CI process using MSBuild and the various tools used. If you have any questions, additional remarks or any suggestions, feel free to drop me an e-mail at carel.lotz@gmail.com.