Continuous Integration: From Theory to Practice

A practical guide for implementing Continuous Integration in a .NET 2.0 Development Environment

Author: Blog: Email address:	Carel Lotz http://dotnet.org.za/cjlotz carel.lotz@gmail.com
Document version: Date last modified:	1.8 15 December 2007

Version	Date	Description
1.0	31 August 2007	Initial publication
1.1	17 September 2007	Added Part 7: The QtpBuild and updated Part 1, Part 2, Part 5 and Part 6 to "flow" with the new content added by this build.
1.2	28 September 2007	Changed the QtpBuild to use incremental builds to install/uninstall the application and to run the tests from Mercury Quality Center.
1.3	07 November 2007	Fixed the RunQtpInstallInput property in Part2: Common Build Targets as the incremental builds for the QtpBuild were not firing correctly.
1.4	07 December 2007	Reorganised the content into a Theory and Practice sections. Added content to the Theory section illustrating the benefits, best practices etc. for Continuous Integration.
1.5	11 December 2007	Included NDepend into the CodeStatisticsBuild and upgraded the CruiseControl.NET Configuration to v1.3 to make use of Build queues. Also removed duplication of CC.NET configuration through the use of DTD entity references.
1.6	12 December 2007	Added the GetEnvironment target to InitialTargets; Changed environment specific builds of DeploymentBuild to also depend upon the Environment.txt file
1.7	13 December 2007	Fixed the environment specific builds by removing the static Environment property and by passing the Environment property created from reading the Environment.txt file to the MSBuild task in the BuildCode target.
1.8	15 December 2007	Fixed reference to MbUnit and added additional items to be removed in the CleanSolution Target.

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 2.0 development tooling that includes Visual Studio 2005, MSBuild, MSBuild.Community.Tasks, CruiseControl.NET, Subversion, FxCop, NUnit, NCover, NDepend, SandCastle, InstallShield and Mercury Quick Test Professional.

Theory

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 (aka "*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

During Q1 2007 I spend so time refactoring our Continuous Integration (CI) process at work. Our current build was causing quite a few headaches - from not showing the build stats correctly to being too slow etc. I decided to redo the whole CI process and thought I'd document our build process as a series of blog posts to serve as a reference guide for other people wanting to do the same thing. The CI process uses MSBuild, VS 2005, InstallShield, Mercury QuickTest Professional, NDepend, NCover and various open source tools like CruiseControl.NET, FxCop, NUnit, Sandcastle and Subversion.

The series consists out of the following posts:

- 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, CodeStatisticsBuild, CodeDocumentationBuild** and **QtpBuild**
- Part 3 covers the DeveloperBuild
- Part 4 covers the DeploymentBuild
- Part 5 covers the CodeStatisticsBuild
- Part 6 covers the CodeDocumentationBuild
- Part 7 covers the QtpBuild

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

This Practice section simply consolidates all the information available in these blog entries.

Resources

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

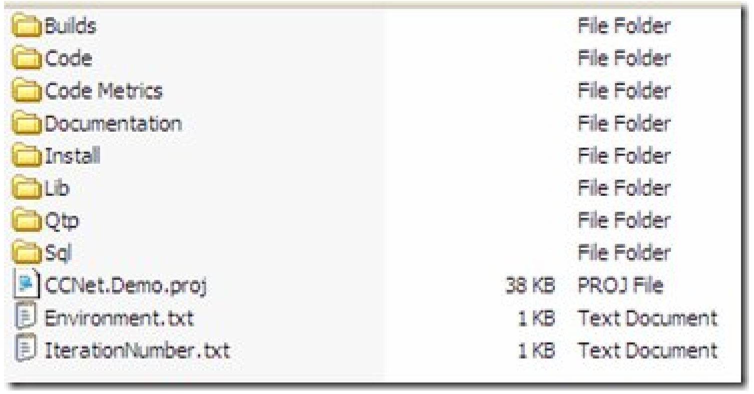
1. Book: Deploying .NET Applications: Learning MSBuild and ClickOnce
2. Channel 9 Wiki: MSBuild.Links
3. Channel 9 Wiki: MSBuild.HomePage
4. MSDN Library: MSBuild Overview
5. MSDN Library: MSBuild Reference
6. Code: MSBuild Community Tasks

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

Before delving into the CI process itself, I need to provide some background on our application to create the context for our CI process. The application, developed using VS 2005, is a stand-alone WinForms application with integration points to various other systems/applications via web services/third party API's. We use Subversion for source control and some third party libraries/controls from vendors like Infragistics, Xpo. We have tried to develop the application using Agile practices like TDD and we are currently running on 1 week iterations. At the end of each iteration we produce a build (.msi) that is released to our testers. We treat all our databases as just-another-code-artifact and have scripts that allow us to create the databases' structure and content. We are able to at any point in time setup a new development environment by simple getting the latest version from our repository.

To facilitate all of this, we have standardized on using the following directory structure for our solution:



Builds		File Folder
Code		File Folder
Code Metrics		File Folder
Documentation		File Folder
Install		File Folder
Lib		File Folder
Qtp		File Folder
Sql		File Folder
CCNet.Demo.proj	38 KB	PROJ File
Environment.txt	1 KB	Text Document
IterationNumber.txt	1 KB	Text Document

- **Builds** folder contains a separate folder for every build and its build artefacts.
- **Code** folder contains all our VS 2005 projects as a flat hierarchy of sub-folders. All the .sln files reside in the **Code** root folder. We have a **Code\Deploy** sub-folder to which all the non-unit test project outputs are compiled.
- **Code Metrics** folder contains all the metrics that we generate for our system like the code coverage results.
- **Documentation** folder contains the help file settings and MSDN style help file we generate from our XML code comments.
- **Install** folder contains our InstallShield .ism file and various merge modules and other files required to create a .msi install for our system.
- **Lib** folder contains all the third party libraries (Infragistics, NUnit, TypeMock, XPO etc.) that are used as project references.
- **Qtp** folder contains the RunQTP.vbs script file and a **TestCases** sub-folder that contains a list of test suite files. Each test suite file contains a list of individual Qtp test cases to run.
- **Sql** folder contains a sub-folder for every database that we use. Each database sub-folder contains all the .sql script files to create the structure and content for the database as well as batch files to automate this creation process using osql/sqlcmd. We support both sql 2000 + sql 2005.

Process

One of the requirements for our CI process is that a developer should be able harvest the same process as the build server to work in his/her sandbox/private workspace 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 from within both the developer sandbox/private workspace and build server.

Our build needs to do the following tasks:

1. Get the latest source from the repository
2. Build the databases
3. Compile the source
4. Run the unit tests
5. Produce code coverage results
6. Produce FxCop results
7. Produce NDepend results
8. Build MSDN style help documentation
9. Backup the databases
10. Version the source code
11. Build a msi
12. Deploy the msi
13. Run the regression test pack using QTP
14. Notify QTP testers via e-mail
15. Tag the repository

Not all of these tasks need to be completed on a continuous basis. Some (like steps 9-15) are only required to run once weekly as part of the iteration build to create the .msi. We also found that creating MSDN style documentation as well as running code coverage and code analysis on every check-in caused our solution build, which consists out of 83 projects, to take way too long. We therefore decided to create a staged build/build pipeline and to split our CI build process into 5 separate builds on two separate servers.

1. **DeveloperBuild** (Runs on BuildServer) - 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, compiling the source code and for running the tests to get quicker build times.
2. **DeploymentBuild** (Runs on BuildServer) - The deployment build is run once weekly at 12:30 on Friday afternoons to create the .msi installation that is deployed to our QTP testers. In addition to creating and deploying the .msi it also tags the repository to create a tag for the work done for the iteration.
3. **CodeStatisticsBuild** (Runs on BuildServer) - The statistics build is set to run on successful completion of the deployment build and produces the Code Coverage, FxCop and NDepend results for the week. The developers can still evaluate the code coverage, FxCop and NDepend results continuously in their sandbox, but we incur the overhead of this on the build server only once a week. This way we can have the stats to spot trends on a weekly basis.
4. **CodeDocumentationBuild** (Runs on BuildServer) - The documentation build is set to be invoked manually and creates MSDN style help documentation from the XML code comments.

5. **QtpBuild** (Runs on QtpServer) - The QtpBuild runs on a dedicated machine as the regression test pack takes quite some time to run through (3 hours in our scenario). The build triggers on the completion of a successful DeploymentBuild on the normal build server.

As evident, the first server (aka the **BuildServer**) will do all builds except for the QtpBuild. The second server (aka the **QtpServer**) will only run the regression test pack using QTP.

Build Server Tools

These are the tools that are used by our CI process and that needs to be installed on our build server running Windows Server 2003:

1. Visual Studio 2005 Team Edition for Software Developers to run Managed Code Analysis (aarrgghhhh!). You can use the standalone version of FxCop, but there are some differences between this and the version used for VS 2005 Code Analysis. Unfortunately you therefore need to install the complete VS 2005 Team Edition for Software Developers.
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.
4. NDepend v2.6 for additional code metrics.
5. MSBuild.Community.Tasks for some additional MSBuild tasks.
6. NUnit 2.0 2.4.5 to run the unit tests.
7. .NET Framework 2.0
8. .NET Framework SDK 2.0
9. CruiseControl .NET 1.3
10. InstallShield Standalone Build v12 - The standalone build is the InstallShield build engine without the GUI and you are allowed to install this on a build server for free granted you have a valid InstallShield license.
11. Subversion 1.4.5 for source control.
12. HTML Help Workshop for building HTML help 1.x files.
13. Visual Studio .NET Help Integration Kit 2003 for building help files that can integrate into Visual Studio.
14. Sandcastle June 2007 CTP to generate the help file content from the XML code comments.
15. Sandcastle Help File Builder (SHFB) to set the options for the help file and to bootstrap the document creation process.
16. Sandcastle Presentation File Patches to solve some issues with the Sandcastle June CTP. Extract the contents of the ZIP archive to the Sandcastle installation directory.
17. Mercury QuickTest Professional 8.2 for automated regression testing.

Next Steps

Well, that takes care of the all the background information. The next post will delve into all the common build targets that are used by the **DeveloperBuild, DeploymentBuild, CodeStatisticsBuild** and **CodeDocumentationBuild.** Keep watching this space...

B.t.w, I'm a big fan of ReSharper and the MsBuild/NAnt support of ReSharper is oh so sweet!!

Part 2: Common Build Targets

This is the second post in a series where I document how to setup a Continuous Integration (CI) process using open source tools like CruiseControl.NET, NCover, NUnit, Subversion and Sandcastle together with MSBuild and VS 2005. Part 2 covers the common build targets and tasks that are used by the **DeveloperBuild, DeploymentBuild, CodeStatisticsBuild, CodeDocumentationBuild** and **QtpBuild.**

MSBuild Scripts

The whole build process is defined in the CCNet.Demo.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>

   <!-- Environment Settings -->
   <PropertyGroup>
      <DBServer Condition=" '$(DBServer)' == '' ">(local)</DBServer>
      <DBMS Condition=" '$(DBMS)' == '' ">sql2005</DBMS>
      <SqlCmdRunner Condition=" '$(DBMS)' == 'sql2000' " >osql</SqlCmdRunner>
      <SqlCmdRunner Condition=" '$(DBMS)' == 'sql2005' " >sqlcmd</SqlCmdRunner>
      <SmtpServer>smtp.yourdomain.co.za</SmtpServer>
   </PropertyGroup>

   <!-- 3rd Party Program Settings -->
   <PropertyGroup>
      <SubversionPath>C:\Program Files\Subversion\bin</SubversionPath>
      <SubversionCmd>$(SubversionPath)\svn.exe</SubversionCmd>
      <NCoverPath>C:\Program Files\NCover\</NCoverPath>
      <NCoverExplorerPath>C:\Program Files\NCoverExplorer\</NCoverExplorerPath>
      <NDependPath>C:\Program Files\NDepend\</NDependPath>
      <NUnitPath>C:\Program Files\NUnit 2.4.5\bin\</NUnitPath>
      <NUnitCmd>$(NUnitPath)nunit-console.exe</NUnitCmd>
      <SandCastleHFBPath>C:\Program Files\EWSoftware\Sandcastle Help File Builder\</SandCastleHFBPath>
      <SandCastleHFBCmd>$(SandCastleHFBPath)SandcastleBuilderConsole.exe</SandCastleHFBCmd>
      <InstallShieldPath>C:\Program Files\Macrovision\IS 12 StandaloneBuild</InstallShieldPath>
      <InstallShieldCmd>$(InstallShieldPath)\IsSaBld.exe</InstallShieldCmd>
```

```xml
</PropertyGroup>

<!-- Solution Folders -->
<PropertyGroup>
   <CodeFolder>$(MSBuildProjectDirectory)\Code</CodeFolder>
   <SqlFolder>$(MSBuildProjectDirectory)\Sql</SqlFolder>
   <LibFolder>$(MSBuildProjectDirectory)\Lib</LibFolder>
   <DocFolder>$(MSBuildProjectDirectory)\Documentation</DocFolder>
   <DocOutputFolder>$(DocFolder)\Help</DocOutputFolder>
   <InstallFolder>$(MSBuildProjectDirectory)\Install</InstallFolder>
   <CodeOutputFolder>$(CodeFolder)\Deploy</CodeOutputFolder>
   <SqlScriptsFolder>$(SqlFolder)\Scripts</SqlScriptsFolder>
   <CodeMetricsFolder>$(MSBuildProjectDirectory)\Code Metrics</CodeMetricsFolder>
   <QtpFolder>$(MSBuildProjectDirectory)\Qtp</QtpFolder>
   <QtpInstallFolder>$(QtpFolder)\Install</QtpInstallFolder>
   <QtpUninstallFolder>$(QtpFolder)\Uninstall</QtpUninstallFolder>
   <QtpTestCasesFolder Condition=" '$(QtpTestCasesFolder)' == '' ">$(QtpFolder)\TestCases</QtpTestCasesFolder>
   <QtpResultsFolder Condition=" '$(QtpResultsFolder)' == '' ">$(QtpFolder)\TestResults</QtpResultsFolder>
   <QtpDeployFolder Condition=" '$(QtpDeployFolder)' == '' ">\\someserver\qtpshare</QtpDeployFolder>
</PropertyGroup>

<!-- Solution Files -->
<PropertyGroup>
   <KeyFile>..\YourPrivateKey.snk</KeyFile>
   <SolutionName>CCNet.Demo.sln</SolutionName>
   <IterationNumberFile>$(MSBuildProjectDirectory)\IterationNumber.txt</IterationNumberFile>
   <EnvironmentFile>$(MSBuildProjectDirectory)\Environment.txt</EnvironmentFile>
   <LastTestRunSucceededFile>LastTestRunSucceeded</LastTestRunSucceededFile>
   <LastCodeAnalysisSucceededFile>LastCodeAnalysisSucceeded</LastCodeAnalysisSucceededFile>
   <InstallBuildEmailFile>$(Temp)\InstallBuildEmailFile.htm</InstallBuildEmailFile>
   <InstallBuildEmailTemplate>$(InstallFolder)\InstallBuildEmailTemplate.htm</InstallBuildEmailTemplate>
   <NUnitFile>TestResult.xml</NUnitFile>
   <FxCopFile>CodeAnalysisLog.xml</FxCopFile>
   <NCoverFile>Coverage.xml</NCoverFile>
   <NCoverLogFile>Coverage.log</NCoverLogFile>
   <NCoverSummaryFile>CoverageSummary.xml</NCoverSummaryFile>
   <NCoverHtmlReport>CoverageSummary.html</NCoverHtmlReport>
   <NDependProjectFile>NDependProject.xml</NDependProjectFile>
   <NDependResultsFile>NDependMain.xml</NDependResultsFile>
   <SandCastleHFBProject>CCNet.Demo.shfb</SandCastleHFBProject>
   <DBBackupScript>$(SqlScriptsFolder)\Create DB Backups automated build.cmd</DBBackupScript>
   <QtpRunTestsScriptFile>$(QtpFolder)\RunQTP.vbs</QtpRunTestsScriptFile>
   <QtpResultsSummaryFile>QtpResultsSummary.xml</QtpResultsSummaryFile>
   <QtpLastInstallSucceededFile>Qtp.LastInstallSucceeded</QtpLastInstallSucceededFile>
</PropertyGroup>

<!-- Subversion Settings -->
```

```xml
  <PropertyGroup>
     <SvnLocalPath>$(MSBuildProjectDirectory)</SvnLocalPath>
     <SvnServerPath>https://svn.yourdomain.co.za/ccnet.demo</SvnServerPath>
     <SvnTrunkFolder>$(SvnServerPath)/trunk</SvnTrunkFolder>
     <SvnTagsFolder>$(SvnServerPath)/tags</SvnTagsFolder>
  </PropertyGroup>

  <!-- InstallShield Settings -->
  <PropertyGroup>
     <MsiInstallFile>CCNet.Demo.msi</MsiInstallFile>
     <InstallShieldProjectFile>$(InstallFolder)\CCNet.Demo.ism</InstallShieldProjectFile>
     <SetMsiProductVersionScript>$(InstallFolder)\SetMsiProductVersion.vbs</SetMsiProductVersionScript>
     <InstallShieldInputFolder>$(InstallFolder)\Binaries\InstallFiles</InstallShieldInputFolder>
     <InstallShieldOutputFolder>$(Temp)\CCNet.Demo\Install</InstallShieldOutputFolder>
     <InstallShieldBuildFolders>PROJECT_ASSISTANT\SINGLE_MSI_IMAGE\DiskImages\DISK1</InstallShieldBuildFolders>
     <InstallShieldMergeModulesFolder>$(InstallFolder)\Merge Modules</InstallShieldMergeModulesFolder>
  </PropertyGroup>

  <!-- Misc Settings -->
  <PropertyGroup>
     <Major>1</Major>
     <Minor>0</Minor>
     <BuildTargets Condition=" '$(BuildTargets)' == '' ">Build</BuildTargets>
     <DeploymentBuild Condition=" '$(DeploymentBuild)' == '' ">false</DeploymentBuild>
     <QCServer Condition=" '$(QCServer)' == '' ">http://qcserver.yourdomain.co.za/qcbin</QCServer>
  </PropertyGroup>

  <!-- Overriding .csproj Project settings -->
  <PropertyGroup>
     <OutputPath Condition=" '$(OutputPath)' == '' ">$(CodeOutputFolder)</OutputPath>
     <Configuration Condition=" '$(Configuration)' == '' ">Debug</Configuration>
     <Platform Condition=" '$(Platform)' == '' ">AnyCPU</Platform>
     <RunCodeAnalysis Condition=" '$(RunCodeAnalysis)' == '' ">false</RunCodeAnalysis>
  </PropertyGroup>

  <!-- Only import the other targets here after the property definitions above have been defined -->
  <Import Project="$(MSBuildBinPath)\Microsoft.CSharp.targets" />
  <Import Project="C:\Program Files\TypeMock\TypeMock.NET\TypeMock.MSBuild.Tasks"/>
  <Import Project="$(MSBuildExtensionsPath)\MSBuildCommunityTasks\MSBuild.Community.Tasks.Targets"/>
  <Import Project="$(MSBuildExtensionsPath)\NCoverExplorerTasks\NCoverExplorer.MSBuildTasks.targets"/>

  <!-- Add our CleanSolution task to the general Clean task -->
  <PropertyGroup>
     <CleanDependsOn>
        $(CleanDependsOn);
        CleanSolution
     </CleanDependsOn>
```

```
127    </PropertyGroup>
128
129    <ItemGroup>
130       <SqlProjects Include="$(SqlFolder)\**\*.proj"/>
131
132       <Developers Include="
133                   developer1@yourdomain.co.za;
134                   developer2@yourdomain.co.za;" />
135
136       <QTPTesters Include=
137                   "tester1@yourdomain.co.za;
138                    tester2@yourdomain.co.za" />
139
140       <RunQtpInstallInput Include="$(QtpDeployFolder)\$(MsiInstallFile)" />
141       <RunQtpInstallOutput Include="$(QtpInstallFolder)\$(QtpLastInstallSucceededFile)" />
142
143    </ItemGroup>
```

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 for TypeMock (our mocking framework), NCover, NCoverExplorer and the MSBuild.Community.Tasks.

Some other observations:

1. 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.
2. 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 target within the build file.
3. Lines 122-127: 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

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

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

```
 <Target Name="GetProjects">

    <!-- Get all the projects associated with the solution -->
    <GetSolutionProjects Solution="$(CodeFolder)\$(SolutionName)">
       <Output TaskParameter="Output" ItemName="SolutionProjects" />
    </GetSolutionProjects>

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

    <!-- Add Code folder to all solution project paths -->
    <RegexReplace Input="@(Projects)" Expression="(CCNet.Demo.*)\\" Replacement="Code\$1\\" Count="-1">
       <Output  TaskParameter="Output" ItemName="CSProjects"/>
    </RegexReplace>

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

    <!-- Resolve the code projects -->
    <CreateItem Include="@(CSProjects)"
                Exclude="@(TestProjects);
                         @(SqlProjects)">
       <Output TaskParameter="Include" ItemName="CodeProjects"/>
    </CreateItem>

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

```
        <Message Text="SqlProjects:$(NEW_LINE)$(TAB)@(SqlProjects->'%(RecursiveDir)%(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 we use different .sln files that contain different projects we want the build to figure out by itself which projects should be compiled given a .sln file is specified via the **SolutionName** property. Remember, that as mentioned in Part 1, the developer should be able to execute the same sub-set of build processes as the build server to compile and test the code in his/her sandbox. The GetProjects target uses the **GetSolutionProjects**, **RegexMatch** and **RegexReplace** tasks from the **MSBuildCommunityTasks** to parse out and filter the different project files from a .sln file according to our naming conventions used.

The rational for using different solution files is developer productivity. Some of our solution files ignore the Presentation Layer projects, some ignore the Unit Test projects etc. This allows a developer to work on a smaller subset of projects which makes working on slow developer PC's less frustrating. Of course 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    <Target Name="GetEnvironment">
2
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        <CreateProperty Value="@(EnvironmentFileContents->'%(Identity)')">
10          <Output TaskParameter="Value" PropertyName="Environment"/>
11       </CreateProperty>
12
13       <Message Text="Running build for Environment = $(Environment)" importance="high" />
14
15    </Target>
```

Observations:

- Lines 4-6: The environment to make the build for is read from the Solution Folder\Environment.txt file. This allows us to build for a different environment by simply changing the file - we **do not** want to change the build server to compile for different environment.
- Lines 9-11: The contents of the file is assigned to the **Environment** property that is used to build for a specific environment (see **Environment Specific Builds** section above)

GetIterationNumber

```
    <Target Name="GetIterationNumber">

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

    <!-- Assign file contents to IterationNumber property -->
    <CreateProperty Value="@(IterationNumberFileContents->'%(Identity)')">
      <Output TaskParameter="Value" PropertyName="IterationNumber"/>
    </CreateProperty>

  </Target>
```

The GetIterationNumber target reads the current iteration number from a file and assigns it to an **IterationNumber** property. We use the IterationNumber as the Build number when versioning our assemblies, databases and msi installation. The rational for storing the number in a file is to allow the build to automatically increase this number when the DeploymentBuild runs through successfully at the end of every iteration.

GetRevisionNumber

```
<Target Name="GetRevisionNumber">

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

</Target>
```

The GetRevisionNumber target reads the current revision number from of our Subversion repository and assigns it to a **Revision** property. It does this by using the **SvnInfo** task defined within the MSBuildCommunityTasks. We use the Revision number when versioning our assemblies, databases and msi installation.

CleanSolution

```
  <Target Name="CleanSolution">

    <Message Text="Cleaning Solution Output"/>

    <!-- Create item collection of custom artifacts produced by the build -->
    <CreateItem Include=
                 "$(CodeOutputFolder)\**\*.*;
                  $(SqlFolder)\*.LastUpdateSucceeded;
                  @(TestProjects->'%(RootDir)%(Directory)bin\$(Configuration)\*$(NUnitFile)');
                  @(TestProjects->'%(RootDir)%(Directory)bin\$(Configuration)\*$(LastTestRunSucceededFile)');
                  @(TestProjects->'%(RootDir)%(Directory)bin\$(Configuration)\*$(NCoverFile)');
                  @(TestProjects->'%(RootDir)%(Directory)bin\$(Configuration)\*$(NCoverLogFile)')">
      <Output TaskParameter="Include" ItemName="SolutionOutput" />
    </CreateItem>
```

```

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

      <!-- Change the default BuildTarget to include a Clean before a Build -->
      <CreateProperty Value="Clean;$(BuildTargets)">
         <Output TaskParameter="Value" PropertyName="BuildTargets"/>
      </CreateProperty>

   </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.

SvnVerify

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

The SvnVerify target verifies that the details required for integrating with the Subversion repository has been specified for the build.

Common CruiseControl.NET 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. We 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='developer1@yourdomain.co.za'/>
               <user name='Developer2' group='developers' address='developer2@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://wikiserver.yourdomain.co.za/wiki' />
      </externalLinks>">

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

   <!ENTITY svn "<sourcecontrol type='svn'>
        <executable>C:\Program Files\Subversion\bin\svn.exe</executable>
        <trunkUrl>https://svn.yourdomain.co.za/ccnet.demo/trunk</trunkUrl>
        <workingDirectory>C:\Projects\CCNet.Demo</workingDirectory>
        <username>fred</username>
        <password>password</password>
      </sourcecontrol>">

   <!ENTITY svnrevert "<exec>
            <executable>C:\Program Files\Subversion\bin\svn.exe</executable>
            <buildArgs>revert C:\Projects\CCNet.Demo --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. The next post will highlight all the targets required for a **DeveloperBuild**. Keep watching this space

Part 3: DeveloperBuild

This is the third post in a series where I document how to setup a Continuous Integration (CI) process using open source tools like CruiseControl.NET, NCover, NUnit, Subversion and Sandcastle together with MSBuild and VS 2005. Part 3 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 can therefore be 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/test 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 sub-folders beneath the **Sql** folder. We also have batch files that create the database by running these script files using osql/sqlcmd. We support both Sql Server 2000 + 2005 . We also support running the scripts on a dedicated database server machine.

```
<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 database projects. The **SqlCmdRunner** 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. For each database, we have created a MSBuild project file that looks as follows:

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

  <PropertyGroup>
     <SqlCmdRunner Condition=" '$(SqlCmdRunner)' == '' ">sqlcmd</SqlCmdRunner>
     <DBServer Condition=" '$(SqlCmdRunner)' == '' ">(local)</DBServer>
     <DeploymentBuild Condition=" '$(DeploymentBuild)' == '' ">false</DeploymentBuild>
  </PropertyGroup>
  <PropertyGroup Condition=" '$(Configuration)|$(Platform)' == 'Debug|AnyCPU' ">
```

```
 9        <OutputPath>bin\Debug\</OutputPath>
10     </PropertyGroup>
11     <PropertyGroup Condition=" '$(Configuration)|$(Platform)' == 'Release|AnyCPU' ">
12        <OutputPath>bin\Release\</OutputPath>
13     </PropertyGroup>
14
15     <ItemGroup>
16        <Sql Include="*.sql" />
17        <Sql Include="Install Database1.bat" />
18        <SqlOutput Include="$(TEMP)\Database1.LastUpdateSucceeded" />
19     </ItemGroup>
20
21     <Target Name="Build"
22             Inputs="@(Sql)"
23             Outputs="@(SqlOutput)">
24
25        <Exec Command="%22Install Database1.bat%22 $(SqlCmdRunner) $(DBServer)"/>
26
27        <Touch Files="@(SqlOutput)" AlwaysCreate="true"/>
28
29     </Target>
30
31     <Target Name="Rebuild"
32             DependsOnTargets="Clean;Build"/>
33
34     <Target Name="Clean">
35        <Delete Files="@(SqlOutput)"/>
36     </Target>
37
38  </Project>
```

The project file is really self explanatory. It provides Targets for the standard Clean, Build and ReBuild targets which allows us to compile and build it using the MSBuild task in the same way as any .csproj file.   Also notice the use of incremental builds at Lines 21-23. All the .sql script files and the batch file are combined into a **Sql** group and used as inputs for the Build target. This is compared to the **SqlOutput** group that contains a single Database1.LastUpdateSucceeded file. If the filestamps of the items in the Sql group is older than the .LastUpdateSucceeded file, MSBuild will execute the Build target using its incremental build logic. Line 27 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

```
<Target Name="BuildCode">

   <!-- Build the assemblies -->
   <MSBuild Projects="@(CodeProjects)"
            Targets="$(BuildTargets)"
Properties="Environment=$(Environment);Configuration=$(Configuration);Platform=$(Platform);RunCodeAnalysis=$(RunCodeAnalysis)">
```

```
 7          <Output TaskParameter="TargetOutputs"
                    ItemName="CodeAssemblies"/>
 9       </MSBuild>
10
11       <!-- Add the compiled code assemblies to master list of all compiled assemblies for the build -->
12       <CreateItem Include="@(CodeAssemblies)">
13          <Output TaskParameter="Include" ItemName="CompiledAssemblies"/>
14       </CreateItem>
15
16       <!-- If code analysis was run, create an item collection of the FxCop result files -->
17       <CreateItem Include="%(CodeAssemblies.FullPath).$(FxCopFile)"
18                   Condition=" '$(RunCodeAnalysis)' == 'true' ">
19          <Output TaskParameter="Include" ItemName="FxCopResults"/>
20       </CreateItem>
21
22       <Message Text="FxCopResults:$(NEW_LINE)$(TAB)@(CodeAssemblies->'%(FullPath).$(FxCopFile)', '$(NEW_LINE)$(TAB)')$(NEW_LINE)"
23                Condition=" '$(RunCodeAnalysis)' == 'true' "
24                Importance="high"/>
25
26    </Target>
```

The compile of the code projects is straightforward. We use the same incremental build functionality used by VS 2005. The list of assemblies compiled are added to a **CodeAssemblies** and **CompiledAssemblies** item group for later use. The **CompiledAssemblies** group is a master list of compiled assemblies that will contain all compiled assemblies - including the test projects. Lines 16-24 can be ignored for now as it is used only by the **CodeStatisticsBuild**.

BuildTests

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

```
20      </CreateItem>
21
22    </Target>
```

The compile of the test projects is basically a mirror of the BuildCode target. The list of assemblies compiled are added to a **TestAssemblies** and the **CompiledAssemblies** item group for later use. Lines 16-20 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

```
<Target Name="Test"
        DependsOnTargets="BuildTests"
        Inputs="@(TestAssemblies)"
        Outputs="@(TestAssemblies->'%(FullPath).$(LastTestRunSucceededFile)')">

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

   <!-- Remove the success file for the projects being tested -->
   <Delete Files="@(TestAssemblies->'%(FullPath).$(LastTestRunSucceededFile)')"/>

   <TypeMockRegister Company="YourCompany" License="xxxxx"/>
   <TypeMockStart Target="2.0" />

   <NUnit Assemblies="@(TestAssemblies)"
          ToolPath="$(NUnitPath)"
          WorkingDirectory="%(TestAssemblies.RootDir)%(TestAssemblies.Directory)"
          OutputXmlFile="@(TestAssemblies->'%(FullPath).$(NUnitFile)')"
          ContinueOnError="true">
      <Output TaskParameter="ExitCode" ItemName="NUnitExitCodes"/>
   </NUnit>

   <TypeMockStop/>

   <!-- Build an item collection of the test projects with their individual test results.
        The XmlQuery task creates the attributes of the xml node as additional meta-data items. -->
   <XmlQuery XmlFileName="%(TestAssemblies.FullPath).$(NUnitFile)"
             XPath="/test-results">
      <Output TaskParameter="Values" ItemName="NUnitResults"/>
   </XmlQuery>
```

```
31
32    <Message Text="NUnitResults:$(NEW_LINE)$(TAB)@(NUnitResults->'%(name)[%(failures)]', '$(NEW_LINE)$(TAB)')$(NEW_LINE)"
Importance="high"/>
33
34    <!-- Write a success file for the project if the tests for the project passed -->
35    <Touch Files="%(NUnitResults.name).$(LastTestRunSucceededFile)"
36           Condition=" '%(NUnitResults.failures)' == '0' "
37           AlwaysCreate="true">
38    </Touch>
39
40    <!-- Copy the test results for the CCNet build before a possible build failure (see next step) -->
41    <CallTarget Targets="CopyTestResults" />
42
43    <!-- Fail the build if any test failed -->
44    <Error Text="Test error(s) occured" Condition=" '%(NUnitExitCodes.Identity)' != '0'"/>
45
46 </Target>
```

Observations:

- Line 1-4: The outputs of the BuildTest target is used as inputs for the Test target. This is compared in Line 4 to each compiled test assemblies' .LastTestRunSucceededFile to allow MSBuild to determine if the Test target should be executed at all using its incremental build functionality. This will prevent running the Test target if no test assemblies were recompiled.
- Line 10: Before running the tests, the .LastTestRunSucceededFile for every compiled test project is deleted.
- Line 12-13 + Line 22: We use **TypeMock** as our mocking framework and it requires us to initialise and stop the framework before running the tests.
- Line 15-21: We use the NUnit task from the **MSBuildCommunityTasks** 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 **NUnitExitCodes**.
- Line 41: We need to copy the test results for CC.NET on the build server to merge and display as part of the build report.
- Line 44: After running the tests we use the list of exit codes in **NUnitExitCodes** to fail the build if we have an exit code of non-zero.

To support incremental testing, we want to create the .LastTestRunSucceededFile only for test projects that succeeded. In that way we can prevent tests from being re-run for projects that were not changed. We therefore need some way to associate an exit code with its test project. To the rescue comes the very handy **XmlQuery** task from the MSBuildCommunityTasks. We use it to parse the NUnit output that is stored in xml format. The test results summary for a NUnit project is stored as attributes on the root tests-results node. The **XmlQuery** task will store these attributes as custom meta-data in its **Values** output property. We then use this meta-data in lines 34-37 to only touch the .LastTestRunSucceededFiles for those projects for which the failures attribute == 0.

CopyTestResults

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

```xml
        <!-- Create item collection of test results -->
        <CreateItem Include="%(TestAssemblies.FullPath).$(NUnitFile)">
           <Output TaskParameter="Include" ItemName="NUnitResults"/>
        </CreateItem>

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

        <CreateItem Include="$(CCNetArtifactDirectory)\*.$(NUnitFile)">
           <Output TaskParameter="Include" ItemName="ExistingNUnitResults"/>
        </CreateItem>

        <Delete Files="@(ExistingNUnitResults)"/>
        <Copy SourceFiles="@(NUnitResults)"
              DestinationFolder="$(CCNetArtifactDirectory)"
              ContinueOnError="true"/>

     </Target>

```

As we only want to copy the test results if the tests were run on the buildserver, 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

```xml
     <Target Name="CodeCoverage">

        <!-- Find all Presentation assemblies -->
        <RegexMatch Input="@(CodeAssemblies)" Expression=".[\.](CCNet.Demo.Presentation).*[\.]dll$">
           <Output TaskParameter="Output" ItemName="PresentationAssemblies"/>
        </RegexMatch>

        <!-- Filter out the Presentation assemblies from the coverage analysis -->
        <CreateItem Include="@(CodeAssemblies)"
                    Exclude="@(PresentationAssemblies)">
           <Output TaskParameter="Include" ItemName="CodeCoverageAssemblies"/>
        </CreateItem>

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

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

        <TypeMockRegister Company="YourCompany" License="xxxxxxx"/>

```

```
22       <TypeMockStart Target="2.0"
23                      Link="NCover"
24                      ProfilerLaunchedFirst="true" />
25
26       <NCover
27           ToolPath="$(NCoverPath)"
28           CommandLineExe="$(NUnitCmd)"
29           CommandLineArgs="$(DOUBLE_QUOTES)%(TestAssemblies.FullPath)$(DOUBLE_QUOTES) /nologo"
30           CoverageFile="%(TestAssemblies.FullPath).$(NCoverFile)"
31           LogLevel="Normal"
32           LogFile="%(TestAssemblies.FullPath).$(NCoverLogFile)"
33           WorkingDirectory="%(TestAssemblies.RootDir)%(TestAssemblies.Directory)"
34           ExcludeAttributes=""
35           RegisterProfiler="false"
36           Assemblies="@(CodeCoverageAssemblies)"
37           ContinueOnError="true"/>
38
39       <TypeMockStop/>
40
41       <!-- Create Item collection of all the coverage results -->
42       <CreateItem Include="%(TestAssemblies.FullPath).$(NCoverFile)">
43          <Output TaskParameter="Include" ItemName="NCoverResults"/>
44       </CreateItem>
45
46       <Message Text="NCoverResults:$(NEW_LINE)$(TAB)@(NCoverResults->'%(FileName).$(NCoverFile)', '$(NEW_LINE)$(TAB)')$(NEW_LINE)"
Importance="low"/>
47
48       <!-- Merge all the coverage results into a single summary Coverage file -->
49       <NCoverExplorer ToolPath="$(NCoverExplorerPath)"
50                       ProjectName="$(MSBuildProjectName)"
51                       OutputDir="$(Temp)"
52                       Exclusions=""
53                       CoverageFiles="@(NCoverResults)"
54                       SatisfactoryCoverage="75"
55                       ReportType="ModuleClassSummary"
56                       HtmlReportName="$(CodeMetricsFolder)\$(NCoverHtmlReport)"
57                       XmlReportName="$(CodeMetricsFolder)\$(NCoverSummaryFile)"
58                       FailMinimum="False"/>
59
60    </Target>
61
```

Observations:

- Line 3-12: We create a **CodeCoverageAssemblies** group of the assemblies that we want to create coverage statistics for. We filter out any Presentation assembly as we currently have not tests to verify these projects.

- Line 18: Before running the tests, the old **NCover** coverage file for every test project is deleted.
- Line 20-24 + Line 38: We use **TypeMock** as our mocking framework and 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.
- Line 26-36: We use the **NCover** MSBuild task 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. We need to do this asTypeMock needs to register itself with NCover (see lines 22-24).
- Line 48-57: 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.

Compliments to Grant Drake for the excellent work he has done/is still doing on **NCoverExplorer**.

NDepend

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    <Target Name="NDepend">
2
3       <Message Text="Running NDepend For:$(CodeMetricsFolder)\$(NDependProjectFile)" Importance="low"/>
4
5       <Exec Command="$(DOUBLE_QUOTES)$(NDependPath)NDepend.Console.exe$(DOUBLE_QUOTES)
$(DOUBLE_QUOTES)$(CodeMetricsFolder)\$(NDependProjectFile)$(DOUBLE_QUOTES)"
6             ContinueOnError="false" />
7
8    </Target>
9
```

Observations:

- Lines 5-6: 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.

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

```

```xml
      <triggers>
         <filterTrigger startTime="12:30" endTime="14:00">
            <trigger type="intervalTrigger" seconds="60" />
            <weekDays>
               <weekDay>Friday</weekDay>
            </weekDays>
         </filterTrigger>
      </triggers>

      <state type="state" />

      &svn;

      <labeller type="defaultlabeller"/>

      <!-- NB! Do not remove the log files from the previous build as a prebuild action as incremental builds
           will not re-create the test run logs for projects that were not updated and thus retested -->

      <tasks>
         <msbuild>
           <executable>C:\WINDOWS\Microsoft.NET\Framework\v2.0.50727\MSBuild.exe</executable>
           <workingDirectory>C:\Projects\CCNet.Demo</workingDirectory>
           <projectFile>CCNET.Demo.proj</projectFile>
           <buildArgs>/noconsolelogger /v:normal /p:SolutionName=CCNet.Demo.sln;Configuration=Debug;DeploymentBuild=false;RunCodeAnalysis=false;DBMS=sql2005;DBServer=(local)</buildArgs>
           <targets>BuildAll,Test</targets>
           <timeout>1800</timeout> <!-- 30 minutes -->
           <logger>C:\Program Files\CruiseControl.NET\server\ThoughtWorks.CruiseControl.MSBuild.dll</logger>
         </msbuild>
      </tasks>

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

         <statistics>
            <statisticList>
               <firstMatch name="Svn Revision" xpath="//modifications/modification/changeNumber" />
            </statisticList>
         </statistics>

         &pub;
      </publishers>

```

```
        &links;
    </project>
```

Again, the file is self explanatory if you are familiar with the different CruiseControl.NET configuration options. The only thing of interest to note is the filter trigger that prevents the build from being fired from 12:30-14:00 on a Friday. This is the window period at the end of an iteration in which the DeploymentBuild and CodeStatisticsBuild are executed - more on this later.

Next Steps

Well, that takes care of all the targets used by the DeveloperBuild. The next post will highlight all the targets required for a **DeploymentBuild**. Keep watching this space 🙂

Part 4: DeploymentBuild

This is the fourth post in a series where I document how to setup a Continuous Integration (CI) process using open source tools like CruiseControl.NET, NCover, NUnit, Sandcastle and Subversion together with MSBuild and VS 2005. Part 4 covers the DeploymentBuild and the targets and tasks used by the DeploymentBuild.

MSBuild Scripts

The DeploymentBuild happens at the end of every iteration - once a week as we have 1 week iterations. It creates a complete .msi package that is distributed to our testers.

Environment specific builds

We need to support deploying 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.) To manage these differences, we have created a MSBuild project file that copies the environment specific version of our .config files to the output directory based on the value set for an **Environment** property. Here is an extract from this project file:

```
<Project DefaultTargets="Build" xmlns="http://schemas.microsoft.com/developer/msbuild/2003">
   <ItemGroup>
      <ConfigFiles Include="App.config" />
      <ConfigFiles Include="ObjectFactories.config" />
      <ConfigFiles Include="DataSources.config" />
      <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>
   <ItemGroup>
      <EnvironmentSetting Include="..\..\Environment.txt" />
      <BindingFiles Include="..\Xml\Bindings\*.xml" />
      <ConfigOutput Include="$(TEMP)\CCNet.Demo.Startup.LastUpdateSucceeded" />
   </ItemGroup>
   <Import Project="$(MSBuildBinPath)\Microsoft.CSharp.targets" />
   <PropertyGroup>
      <Environment Condition=" '$(Environment)' == '' ">DEV</Environment>
```

```
27    </PropertyGroup>
28    <Target Name="Build" Inputs="@(EnvironmentSetting);@(EnvironmentConfigFiles);@(ConfigFiles);@(BindingFiles)" Outputs="@(ConfigOutput)">
29       <Copy SourceFiles="@(BindingFiles)" DestinationFolder="$(OutputPath)" />
30       <!-- Copy the environment unspecific files -->
31       <Copy SourceFiles="@(ConfigFiles)" DestinationFolder="$(OutputPath)" />
32       <!-- Copy the environment specific files -->
33       <Copy SourceFiles="@(EnvironmentConfigFiles)" DestinationFolder="$(OutputPath)" Condition=" '%(EnvironmentConfigFiles.Env)' == 
'$(Environment)' " />
34       <Touch Files="@(ConfigOutput)" AlwaysCreate="true" />
35    </Target>
36    <Target Name="Rebuild" DependsOnTargets="Clean;Build" />
37    <Target Name="Clean">
38       <Delete Files="@(ConfigOutput)" />
39    </Target>
40 </Project>
```

Observations:

- Lines 6-17: We create an item group **EnvironmentConfigFiles** with environment specific custom meta-data. We use this meta-data (see Line 32) to figure out what environment files to copy.
- Lines 28-39: We add the standard Clean, Build and Rebuild targets to allow us to compile the project as any standard .csproj. The "compile" in effect only copies the relevant files to the output directory.
- Lines 28+34: We use incremental builds to only build the project if it is outdated.

Version

```
    <Target Name="Version"
            DependsOnTargets="GetIterationNumber;GetRevisionNumber"
            Condition=" '$(DeploymentBuild)' == 'true' ">

       <!-- Assign IterationNumber to the Build property -->
       <CreateProperty Value="$(IterationNumber)">
          <Output TaskParameter="Value" PropertyName="Build"/>
       </CreateProperty>

       <!-- Add 1 to the Revision for the VersionInfo update commit -->
       <Add Numbers="$(Revision);1">
          <Output TaskParameter="Result" PropertyName="Revision"/>
       </Add>

       <CreateProperty Value="$(Major).$(Minor).$(Build).$(Revision)">
          <Output TaskParameter="Value" PropertyName="AppVersion"/>
       </CreateProperty>

       <Message Text="AppVersion number generated: $(AppVersion)" Importance="high"/>

```

```
21       <!-- Generate the GlobalAssemblyInfo.cs file -->
22       <AssemblyInfo CodeLanguage="CS"
23            OutputFile="$(CodeFolder)\GlobalAssemblyInfo.cs"
24            AssemblyConfiguration="$(Configuration)"
25            AssemblyCompany="YourCompany Ltd"
26            AssemblyProduct="CCNet.Demo"
27            AssemblyCopyright="Copyright (c) 2006-2007 YourCompany Ltd"
28            AssemblyTrademark=""
29            AssemblyCulture=""
30            CLSCompliant="true"
31            AssemblyVersion="$(AppVersion)"
32            AssemblyFileVersion="$(AppVersion)" />
33
34    </Target>
35
```

Observations:

- Line 2: The target depends on the **GetIterationNumber** and **GetRevisionNumber** targets described in Part 2
- Lines 11-12: The **Revision** property is incremented to cater for the futher commit (see TagRepository target) that will follow because of the version and iteration number updates.
- Lines 15-17: An **AppVersion** property in the format Major.Minor.Build.Revision is created.
- Lines 22-32: The AssemblyInfo task from the MSBuildCommunityTasks is used to create an GlobalAssemblyInfo.cs file that is used when compiling all the .csproj files.

BackupDatabases

```
 1    <Target Name="BackupDatabases"
 2            Condition=" '$(DeploymentBuild)' == 'true' ">
 3
 4       <Error Text="No DBBackupFolder has been specified" Condition="'$(DBBackupFolder)' == ''" />
 5
 6       <Exec Command="$(DOUBLE_QUOTES)$(DBBackupScript)$(DOUBLE_QUOTES) $(SqlCmdRunner) $(DBServer)"
 7             WorkingDirectory="$(SqlScriptsFolder)"/>
 8
 9       <CreateItem Include="$(DBBackupFolder)\Database1.bak">
10          <Output TaskParameter="Include" ItemName="DBBackupFiles"/>
11       </CreateItem>
12
13    </Target>
```

Observations:

- Lines 6-7: A batch file is executed to create a backup of all the databases for inclusion in the .msi package.
- Lines 9-11: All the database backup files are added to a **DBBackupFiles** group for later use.

Sign

```
1    <Target Name="Sign"
2            Condition=" '$(DeploymentBuild)' == 'true' ">
3
4       <!-- Sign the assemblies -->
5       <Exec Command="$(DOUBLE_QUOTES)$(FrameworkSdkPath)Bin\sn.exe$(DOUBLE_QUOTES) -Ra
$(DOUBLE_QUOTES)%(CompiledAssemblies.FullPath)$(DOUBLE_QUOTES) $(KeyFile)"/>
6
7    </Target>
```

Observations:

- Line 5: As we use delay signing in our development sandboxes, we resign our assemblies before deployment with our private key.

Package

```
1    <Target Name="Package"
2            DependsOnTargets="BackupDatabases"
3            Condition=" '$(DeploymentBuild)' == 'true' ">
4
5       <Error Text="No InstallShieldInputFolder has been specified" Condition="'$(InstallShieldInputFolder)' == ''" />
6       <Error Text="No InstallShieldProjectFile has been specified" Condition="'$(InstallShieldProjectFile)' == ''" />
7
8       <CreateItem Include=
9                     "$(CodeOutputFolder)\**\*.*;
10                     $(LibFolder)\*.*;"
11                  Exclude=
12                     "$(CodeOutputFolder)\*.cmd;
13                      $(CodeOutputFolder)\*.$(NUnitFile);
14                      $(CodeOutputFolder)\*.$(LastTestRunSucceededFile);
15                      $(CodeOutputFolder)\*.$(FxCopFile);
16                      $(CodeOutputFolder)\*.$(LastCodeAnalysisSucceededFile);
17                      $(CodeOutputFolder)\*.$(NCoverFile);
18                      $(CodeOutputFolder)\*.$(NCoverLogFile);
19                      $(CodeOutputFolder)\Binding\**;
20                      $(CodeOutputFolder)\Logs\**;
21                      $(CodeOutputFolder)\Utilities\**;
22                      $(CodeOutputFolder)\**\.svn\**;
25                      $(LibFolder)\nunit*;
26                      $(LibFolder)\TypeMock*;
27                      $(LibFolder)\**\.svn\**">
28          <Output TaskParameter="Include" ItemName="MsiFiles"/>
29       </CreateItem>
30
31       <CreateItem Include="$(InstallShieldInputFolder)\**\*.*">
32          <Output TaskParameter="Include" ItemName="OldMsiFiles"/>
```

```
33       </CreateItem>
34
35       <Delete Files="@(OldMsiFiles)"/>
36
37       <Copy SourceFiles="@(MsiFiles)"
38             DestinationFiles="@(MsiFiles->'$(InstallShieldInputFolder)\%(RecursiveDir)%(FileName)%(Extension)')" />
39       <Copy SourceFiles="@(DBBackupFiles)"
40             DestinationFiles="@(DBBackupFiles->'$(InstallShieldInputFolder)\Database\%(RecursiveDir)%(FileName)%(Extension)')" />
41
42       <!-- Set the version on the InstallShield Project -->
43       <Exec Command="$(DOUBLE_QUOTES)$(SetMsiProductVersionScript)$(DOUBLE_QUOTES)
$(DOUBLE_QUOTES)$(InstallShieldProjectFile)$(DOUBLE_QUOTES) $(AppVersion)"/>
44
45       <!-- Call InstallShield to create the msi -->
46       <Exec Command="$(DOUBLE_QUOTES)$(InstallShieldCmd)$(DOUBLE_QUOTES) -p $(DOUBLE_QUOTES)$(InstallShieldProjectFile)$(DOUBLE_QUOTES) -o
$(DOUBLE_QUOTES)$(InstallShieldMergeModulesFolder)$(DOUBLE_QUOTES) -t $(DOUBLE_QUOTES)$(FrameworkPath)$(DOUBLE_QUOTES) -b
$(DOUBLE_QUOTES)$(InstallShieldOutputFolder)$(DOUBLE_QUOTES)" />
47
48       <CreateItem Include="$(InstallShieldOutputFolder)\$(InstallShieldBuildFolders)\$(MsiInstallFile)">
49          <Output TaskParameter="Include" ItemName="MsiInstallFileFullPath"/>
50       </CreateItem>
51
52    </Target>
```

Observations:

- Lines 8-29: A **MsiFiles** group is created that collects all the files required for building our .msi from the various folders.
- Lines 37-40: The files are copied to the working directory used by our InstallShield project.
- Line 43: Before building the msi, we run the **SetMsiProductVersion.vbs** file that automatically sets the version number of the msi to align with the **AppVersion** property.
- Line 46: InstallShield Standalone Build is invoked to build the msi
- Lines 48-50: The msi output is assigned to a **MsiInstallFileFullPath** property for later use.

Here is the contents of the SetMsiProductVersion.vbs file. It uses the InstallShield automation interface to set the ProductVersion property on the InstallShield project.

```
Option Explicit: Dim oIPWI, arrayVersionFields, UpperBound

Set oIPWI = CreateObject("SAAuto12.ISWiProject")

' Open the project by reading the project location from the command line
oIPWI.OpenProject WScript.Arguments.Item(0)

' Read the ProductVersion from the commandline and write it back to the project
oIPWI.ProductVersion = WScript.Arguments.Item(1)

oIPWI.SaveProject
oIPWI.CloseProject
```

Deploy

```
<Target Name="Deploy"
        Condition=" '$(DeploymentBuild)' == 'true' ">

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

   <CreateProperty Value="$(DeploymentFolder)\$(AppVersion)">
      <Output TaskParameter="Value" PropertyName="InstallFolder"/>
   </CreateProperty>

   <Message Text="Install file: $(InstallFolder)\$(MsiInstallFile)" Importance="high" />

   <Copy SourceFiles="@(MsiInstallFileFullPath)"
         DestinationFolder="$(InstallFolder)"/>

   <!-- Copy the msi onto the Qtp machine for regression testing -->
   <Copy SourceFiles="@(MsiInstallFileFullPath)"
         DestinationFolder="$(QtpInstallFolder)"/>


   <!-- Create the tokens for the InstallBuildEmail template -->
   <CreateItem Include="AppVersion" AdditionalMetadata="ReplacementValue=$(AppVersion)">
      <Output TaskParameter="Include" ItemName="EmailTokens"/>
   </CreateItem>
   <CreateItem Include="InstallFileFullPath" AdditionalMetadata="ReplacementValue=$(InstallFolder)\$(MsiInstallFile)">
      <Output TaskParameter="Include" ItemName="EmailTokens"/>
   </CreateItem>

   <!-- Create the InstallBuildEmail by replacing the tokens in the template file -->
   <TemplateFile Template="$(InstallBuildEmailTemplate)"
                 OutputFilename="$(InstallBuildEmailFile)"
                 Tokens="@(EmailTokens)" />

   <ReadLinesFromFile File="$(InstallBuildEmailFile)">
      <Output TaskParameter="Lines" ItemName="EmailBody"/>
   </ReadLinesFromFile>

   <!-- Send e-mail notification of the new build -->
   <Mail SmtpServer="$(SmtpServer)"
         To="@(Developers);@(QTPTesters)"
         Priority="High"
         IsBodyHtml="true"
         From="$(CCNetProject)@somewhere.co.za"
```

```
43              Subject="Build $(AppVersion) ready for testing."
44              Body="@(EmailBody)" />
45
46     </Target>
```

Observations:

- Lines 12-13: The msi is copied to the deployment server.
- Lines 16-17: The msi is copied to the qtp server.
- Lines 20-26: An **EmailTokens** item group is created to use to create an automatic install mail from a template file.
- Lines 29-31: The **TemplateFile** task from the MSBuildCommunityTasks 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 33-35: The contents of the completed mail template file is read into an **EmailBody** item.
- Lines 38-44: The **Mail** task from the MSBuildCommunityTasks is used to send an automatic confirmation of the new build. The mail contains a link to the location from where it can be installed.

Here is the contents 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 ${AppVersion} is ready for installation.&nbsp; You can install it from <A href="file://${InstallFileFullPath}">this location</A>.</SPAN></FONT></DIV><DIV><FONT face="Trebuchet MS" size=2></FONT>&nbsp;</DIV></BODY></HTML>
```

CommitBuildChanges

```
 1     <Target Name="CommitBuildChanges"
 2             DependsOnTargets="SvnVerify">
 3
 4        <!-- Commit all the build changes -->
 5
 6        <CreateProperty Value="Automatic commit of the build changes made by the buildserver.">
 7           <Output TaskParameter="Value" PropertyName="SvnCommitMessage"/>
 8        </CreateProperty>
 9
10        <!-- Use svn directly as the SvnCommit task from MSBuildCommunityTasks requires Targets property to be set -->
11        <Exec Command="$(DOUBLE_QUOTES)$(SubversionCmd)$(DOUBLE_QUOTES) commit $(DOUBLE_QUOTES)$(SvnLocalPath)$(DOUBLE_QUOTES) --username $(SvnUsername) --password $(SvnPassword) --message $(DOUBLE_QUOTES)$(SvnCommitMessage)$(DOUBLE_QUOTES)"/>
12
13     </Target>
```

Observations:

- Line 11: We use the Subversion command line to commit the changes made by the build back into the repository.

TagRepository

After successfully completing all of the above mentioned targets, we want to increment the iteration number, commit the changes made by the **DeploymentBuild** and tag our Subversion repository using the **AppVersion** created.

```
 1   <Target Name="TagRepository"
 2           DependsOnTargets="SvnVerify;Version;"
 3           Condition=" '$(DeploymentBuild)' == 'true' ">
 4
 5      <!-- Increment the iteration number before the commit -->
 6      <Add Numbers="$(IterationNumber);1">
 7         <Output TaskParameter="Result" PropertyName="NextIterationNumber"/>
 8      </Add>
 9
10      <!-- Write out new iteration number to disk -->
11      <WriteLinesToFile File="$(IterationNumberFile)"
12                        Lines="$(NextIterationNumber)"
13                        Overwrite="true"/>
14
15      <!-- Commit all build changes -->
16      <CallTarget Targets="CommitBuildChanges" />
17
18      <!-- Tag the repository at the end of an iteration -->
19      <SvnCopy SourcePath="$(SvnTrunkFolder)"
20               DestinationPath="$(SvnTagsFolder)/Build-$(AppVersion)"
21               Message="Automatic tag by the buildserver for the successful IterationBuild $(IterationNumber)." />
22
23      <Message Text="Tag created at $(SvnTagsFolder)/Build-$(AppVersion)"  Importance="high"/>
24
25   </Target>
26
```

Observations:

- Lines 6-13: The iteration number is incremented and written out to file to commit back into the repository.
- Lines 16: All the changes made on the build server by the DeploymentBuild, is committed back into the repository. This includes the GlobalAssemblyInfo.cs, InstallShield project file and the IterationNumber file.
- Lines 19-23: After successfully comitting these changes, the **SvnCopy** task from the MSBuildCommunityTasks is used to tag the repository.

CruiseControl.NET Configuration

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

```
<project name="DeploymentBuild" queue="CCNet.Demo" queuePriority="2">
```

```
        &header;
        <category>Deployment</category>
        <artifactDirectory>C:\Projects\CCNet.Demo\Builds\DeploymentBuild\Artifacts</artifactDirectory>

        <triggers>
           <scheduleTrigger time="12:30" buildCondition="ForceBuild">
              <weekDays>
                 <weekDay>Friday</weekDay>
              </weekDays>
           </scheduleTrigger>
        </triggers>

        &svn;

        <state type="state" />

        <labeller type="defaultlabeller"/>

        <!-- Remove the log files from the previous build -->
        <prebuild>
           <exec>
              <executable>clean.bat</executable>
              <baseDirectory>C:\Projects\CCNet.Demo\Builds\DeploymentBuild</baseDirectory>
           </exec>
        </prebuild>

        <tasks>
           <msbuild>
              <executable>C:\WINDOWS\Microsoft.NET\Framework\v2.0.50727\MSBuild.exe</executable>
              <workingDirectory>C:\Projects\CCNet.Demo</workingDirectory>
              <projectFile>CCNET.Demo.proj</projectFile>
              <buildArgs>/noconsolelogger /v:normal
/p:SolutionName=CCNet.Demo.sln;Configuration=Release;Platform=AnyCPU;DeploymentBuild=true;RunCodeAnalysis=false;DBMS=sql2005;DBServer=(local)
;SvnUserName=fred;SvnPassword=password;DBBackupFolder="C:\Program Files\Microsoft SQL
Server\MSSQL.1\MSSQL\Backup";DeploymentFolder="\\someserver\installations";QtpDeployFolder=\\someserver\QtpShare;DocumentationFile=</buildArg
s>
              <targets>GetEnvironment,Version,BuildDatabases,BackupDatabases,BuildCode,Sign,Package,Deploy,TagRepository</targets>
              <timeout>1200</timeout> <!-- 20 minutes -->
              <logger>C:\Program Files\CruiseControl.NET\server\ThoughtWorks.CruiseControl.MSBuild.dll</logger>
           </msbuild>
        </tasks>

        <publishers>
           <statistics>
              <statisticList>
                 <firstMatch name="Svn Revision" xpath="//modifications/modification/changeNumber" />
              </statisticList>
```

```
45          </statistics>
46
47          <!-- Revert changes that may still exist because the build failed -->
48          &svnrevert;
49
50          &pub;
51       </publishers>
52
53       &links;
54    </project>
55
```

Observations:

- Lines 6-12: The build is set to start at 12:30 every Friday.
- Lines 28-38: The MSBuild task is used to execute all the DeploymentBuild targets.
- Line 48: To cater for scenarios where the build fails, we need to revert all 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.

Next Steps

Well, that takes care of all the targets used by the DeploymentBuild. The next post will highlight all the targets required for the **CodeStatisticsBuild**. Keep watching this space

Part 5: CodeStatisticsBuild

This is the fifth post in a series where I document how to setup a Continuous Integration (CI) process using open source tools like CruiseControl.NET, NCover, NUnit, Subversion and Sandcastle together with MSBuild and VS 2005. Part 5 covers the CodeStatisticsBuild and the targets and tasks used by the CodeStatisticsBuild.

MSBuild Scripts

The CodeStatisticsBuild happens at the end of every iteration - once a week as we have 1 week iterations. Its fires on successful completion of the DeploymentBuild. The purpose of the CodeStatisticsBuild is to generate metrics on our code base that we can use on a weekly basis to spot 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 already been defined for the 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 NDepend 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.

CopyCodeAnalysisResults

If you look at the **BuildCode** and **BuildTests** targets that the **BuildAll** target depends on, you will see 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    <Target Name="CopyCodeAnalysisResults"
2            Condition=" '$(CCNetProject)' != '' ">
3
4       <Message Text="FxCopResults:$(NEW_LINE)$(TAB)@(FxCopResults->'%(Identity)', '$(NEW_LINE)$(TAB)')$(NEW_LINE)" Importance="low"/>
5
6       <CreateItem Include="$(CCNetArtifactDirectory)\*.$(FxCopFile)">
7          <Output TaskParameter="Include" ItemName="ExistingFxCopResults"/>
8       </CreateItem>
9
10      <Delete Files="@(ExistingFxCopResults)"/>
11      <Copy SourceFiles="@(FxCopResults)"
12            DestinationFolder="$(CCNetArtifactDirectory)"
13            ContinueOnError="true"/>
14
15   </Target>
```

Observations:

- Line 2: We test on the **CCNetProject** property to determine if the build was invoked via CC.NET. The **CCNetProject** property is one of the properties created and passed on by CC.NET when using the MSBuild CC.NET task.
- Lines 6-10: The existing FxCop results are deleted.

- Line 11-13: The new set of FxCop results are copied to the **CCNetArtifactDirectory**.

CopyCoverageResults

If you look at the **CodeCoverage** target, 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    <Target Name="CopyCodeCoverageResults"
2            Condition=" '$(CCNetProject)' != '' ">
3
4      <CreateItem Include="$(CCNetArtifactDirectory)\$(NCoverSummaryFile)">
5         <Output TaskParameter="Include" ItemName="ExistingNCoverResults"/>
6      </CreateItem>
7
8      <Delete Files="@(ExistingNCoverResults)"/>
9      <Copy SourceFiles="$(CodeMetricsFolder)\$(NCoverSummaryFile)"
10           DestinationFolder="$(CCNetArtifactDirectory)"
11           ContinueOnError="true"/>
12
13   </Target>
```

Observations:

- Line 2: We test on the **CCNetProject** property to determine if the build was invoked via CC.NET.
- Lines 4-8: The existing NCover summary file is deleted.
- Line 9-11: The new NCover summary file is copied to the **CCNetArtifactDirectory**.

CopyNDependResults

If you look at the **NDepend** target, you will see notice that the NDepend results are created in a **NDependOutFolder**. Of these results, we are 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 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    <Target Name="CopyNDependResults"
2            Condition=" '$(CCNetProject)' != '' ">
3
4      <CreateItem Include="$(CCNetArtifactDirectory)\NDepend\$(NDependResultsFile)">
5         <Output TaskParameter="Include" ItemName="ExistingNDependResults"/>
6      </CreateItem>
7
8      <Delete Files="@(ExistingNDependResults)"/>
9
```

```
10        <Copy SourceFiles="$(NDependOutputFolder)\$(NDependResultsFile)"
11              DestinationFolder="$(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.
- Lines 4-6: The existing NDepend results file is deleted.
- Line 9-11: The new NDepend results file is copied to the **CCNetArtifactDirectory**.

CruiseControl.NET Configuration

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

```
<project name="CodeStatisticsBuild" queue="CCNet.Demo" queuePriority="3">
   &header;
   <category>CodeStatistics</category>
   <artifactDirectory>C:\Projects\CCNet.Demo\Builds\CodeStatisticsBuild\Artifacts</artifactDirectory>

   <triggers>
      <projectTrigger serverUri="tcp://buildserver.yourdomain.co.za:21234/CruiseManager.rem" project="DeploymentBuild">
         <triggerStatus>Success</triggerStatus>
         <innerTrigger type="intervalTrigger" seconds="30" buildCondition="ForceBuild"/>
      </projectTrigger>
   </triggers>

   <state type="state" />

   <labeller type="defaultlabeller"/>

   <!-- Remove the log files from the previous build as we are creating a complete new build -->
   <prebuild>
      <exec>
         <executable>clean.bat</executable>
         <baseDirectory>C:\Projects\CCNet.Demo\Builds\CodeStatisticsBuild</baseDirectory>
      </exec>
   </prebuild>

   <tasks>
      <msbuild>
         <executable>C:\WINDOWS\Microsoft.NET\Framework\v2.0.50727\MSBuild.exe</executable>
         <workingDirectory>C:\Projects\CCNet.Demo</workingDirectory>
```

```
29             <projectFile>CCNET.Demo.proj</projectFile>
30             <buildArgs>/noconsolelogger /v:normal /p:FxCopDir="C:\Program Files\Microsoft Visual Studio 8\Team Tools\Static Analysis
Tools\FxCop";SolutionName=CCNet.Demo.sln;Configuration=Release;Platform=AnyCPU;DeploymentBuild=false;RunCodeAnalysis=true;Environment=DEV;Doc
umentationFile=</buildArgs>
31             <targets>BuildAll,CopyCodeAnalysisResults,CodeCoverage,CopyCodeCoverageResults,NDepend,CopyNDependResults</targets>
32             <timeout>3600</timeout>  <!-- 1 hour -->
33             <logger>C:\Program Files\CruiseControl.NET\server\ThoughtWorks.CruiseControl.MSBuild.dll</logger>
34          </msbuild>
35       </tasks>
36
37       <publishers>
38          <merge>
39             <files>
40                <file>C:\Projects\CCNet.Demo\Builds\CodeStatisticsBuild\Artifacts\*.CodeAnalysisLog.xml</file>
41                <file>C:\Projects\CCNet.Demo\Builds\CodeStatisticsBuild\Artifacts\CoverageSummary.xml</file>
42                <file>C:\Projects\CCNet.Demo\Builds\CodeStatisticsBuild\Artifacts\NDepend\*.xml</file>
43             </files>
44          </merge>
45
46          <statistics>
47             <statisticList>
48                <firstMatch name="Svn Revision" xpath="//modifications/modification/changeNumber" />
49                <firstMatch name="Coverage" xpath="//coverageReport/project/@coverage" />
50                <firstMatch name="ILInstructions" xpath="//ApplicationMetrics/@NILInstruction" />
51                <firstMatch name="LinesOfCode" xpath="//ApplicationMetrics/@NbLinesOfCode" />
52                <firstMatch name="LinesOfComment" xpath="//ApplicationMetrics/@NbLinesOfComment" />
53                <statistic name='FxCop Warnings'
xpath="count(//FxCopReport/Targets/Target/Modules/Module/Namespaces/Namespace/Types/Type/Members/Member/Messages/Message/Issue[@Level='Warnin
g'])" />
54                <statistic name='FxCop Errors'
xpath="count(//FxCopReport/Targets/Target/Modules/Module/Namespaces/Namespace/Types/Type/Members/Member/Messages/Message/Issue[@Level='Critic
alError'])" />
55             </statisticList>
56          </statistics>
57
58          &pub;
59       </publishers>
60
61       &links;
62    </project>
63
```

Observations:

- Lines 6-11: The build is set to trigger on successful completion of the DeploymentBuild
- Lines 18-23: Before starting the build, the previous build's log files are removed

- Lines 38-44: The NCover, FxCop and NDepend results are merged into the build results
- Lines 47-55: The Statistics publisher is used to publish some additional metrics - see this post for more details.

Next Steps

Finally! That takes care of all the targets used by the CodeStatisticsBuild. The next post will highlight all the targets required for the **CodeDocumentationBuild**. Keep watching this space 🙂

Part 6: CodeDocumentationBuild

This is the sixth post in a series where I document how to setup a Continuous Integration (CI) process using open source tools like CruiseControl.NET, NCover, NUnit, Subversion and Sandcastle together with MSBuild and VS 2005. Part 6 covers the CodeDocumentationBuild and the targets and tasks used by the CodeDocumentationBuild. The CodeDocumentationBuild shows how to create MSDN style help documentation from the XML code comments.

Tools

I used the following tools to generate the MSDN style documentation:

1. HTML Help Workshop for building HTML help 1.x files.
2. Visual Studio .NET Help Integration Kit 2003 for building help files that can integrate into Visual Studio.
3. Sandcastle June 2007 CTP to generate the help file content from the XML code comments.
4. Sandcastle Help File Builder (SHFB) to set the options for the help file and to bootstrap the document creation process.
5. Sandcastle Presentation File Patches to solve some issues with the Sandcastle June CTP. Extract the contents of the ZIP archive to the Sandcastle installation directory.

Process

Before automating the process via MSBuild and CC.NET, I used SHFB's GUI to create a .shfb help file project that contains all the settings required to build our help file. Settings like the assemblies to document, the help file styles to use, member accessibility (public/private/internal/protected) to document, namespaces to document etc. can all be set via the GUI. I then added the .shfb file to a **\Documentation** folder (see Part 1 for our directory structure) and verified the documentation output by compiling the project using the SHFB GUI.

MSBuild Scripts

Getting the project integrated into MSBuild was quite easy as SHFB comes with a console runner that can be used to bootstrap the whole document generation process. I simply had to add a **BuildDocumentation** target and a few additional settings to our build file to get it running:

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

      <!-- Build source code docs -->
      <Exec Command="$(DOUBLE_QUOTES)$(SandCastleHFBCmd)$(DOUBLE_QUOTES)
$(DOUBLE_QUOTES)$(DocFolder)\$(SandCastleHFBProject)$(DOUBLE_QUOTES)" />

   </Target>

```

Observations:

- Line 1-3: The **BuildDocumentation** 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.
- Line 6: The SHFB Console runner is called using our settings file to generated the documentation. The output is stored in the **\Documentation\Help** folder.

CruiseControl.NET Configuration

As the document generation process can take quite a while to complete, I decided to create a separate **CodeDocumentationBuild** CC.NET project to generate the documentation. This way we don't incur the overhead of generating the documentation on every build. Here is the CruiseControl.NET Configuration:

```
<project name="CodeDocumentationBuild" queue="CCNet.Demo" queuePriority="4">
   &header;
   <category>CodeDocumentation</category>
   <artifactDirectory>C:\Projects\CCNet.Demo\Builds\CodeDocumentationBuild\Artifacts</artifactDirectory>

   <triggers />

   &svn;

   <state type="state" />

   <labeller type="defaultlabeller"/>

   <!-- Remove the log files from the previous build as we are creating a complete new build -->
   <prebuild>
      <exec>
         <executable>clean.bat</executable>
         <baseDirectory>C:\Projects\CCNet.Demo\Builds\CodeDocumentationBuild</baseDirectory>
      </exec>
   </prebuild>

   <tasks>
      <msbuild>
         <executable>C:\WINDOWS\Microsoft.NET\Framework\v2.0.50727\MSBuild.exe</executable>
         <workingDirectory>C:\Projects\CCNet.Demo</workingDirectory>
         <projectFile>CCNET.Demo.proj</projectFile>
         <buildArgs>/noconsolelogger /v:normal
/p:SolutionName=CCNet.Demo.sln;Configuration=Release;Platform=AnyCPU;DeploymentBuild=false;RunCodeAnalysis=false</buildArgs>
         <targets>BuildDocumentation,CommitBuildChanges</targets>
         <timeout>3600</timeout>  <!-- 1 hour -->
         <logger>C:\Program Files\CruiseControl.NET\server\ThoughtWorks.CruiseControl.MSBuild.dll</logger>
      </msbuild>
```

```
32      </tasks>
33
34      <publishers>
35
36         <statistics>
37            <statisticList>
38               <firstMatch name="Svn Revision" xpath="//modifications/modification/changeNumber" />
39            </statisticList>
40         </statistics>
41
42         <!-- Revert changes that may still exist because the build failed -->
43         &svnrevert;
44
45         &pub;
46      </publishers>
47
48      &links;
49   </project>
50
```

Observations:

- Line 6: The build makes use of a force trigger, i.e. it needs to be manually invoked.
- Line 27: The solution is compiled in **Release** mode to generate the XML code comments but **RunCodeAnalysis** is turned off to prevent running FxCop that is also turned on in Release mode.
- Line 28: The **BuildDocumentation** target is invoked to build the documentation and the latest version of the help file is committed to the repository by calling the **CommitChanges** target. As we generate a single .chm help file this is a feasible approach. For projects where a MSDN style documentation web site is created, a better approach would be to deploy the web site contents onto a server for publication.

Next Steps

As you can see it was quite easy to add the document generation to our build. I also found this handy resource that shows you how to use all the possible XML code documentation tags to create decent code documentation. The next post will highlight all the targets required for the **QtpBuild**. Keep watching this space 🙂

Part 7: QtpBuild

This is the seventh post in a series where I document how to setup a Continuous Integration (CI) process using open source tools like CruiseControl.NET, NCover, NUnit, Subversion and Sandcastle together with MSBuild and VS 2005. Part 7 covers the QtpBuild and the targets and tasks used by the QtpBuild. The QtpBuild shows how to run the regression test pack using Mercury's QuickTest Professional.

MSBuild Scripts

As mentioned in Part 1, QTP runs on a dedicated QtpServer machine as the regression test pack takes quite some time to run through (3 hours in our scenario). This begs the question how to invoke Qtp remotely from the BuildServer. I considered the following options:

1. Remotely invoke Qtp using psexec from our BuildServer.
2. Remotely invoke Qtp using DCOM from our BuildServer. This seems possible using QTP's automation object model.
3. Instead of remotely invoking QTP, install CC.NET on the QtpServer machine itself and create a project that monitors and triggers a build once the DeploymentBuild has been completed on the BuildServer.

Depending on your environment scenario you may choose either option 1 or 2 as it requires less configuration, but in the end I chose to not use a remote invocation mechanism in favour of option 3. The down side is that I have to install CC.NET on the QtpServer as well. In addition to deploying the build onto our file server, the DeploymentBuild also copies the msi onto a share for the QtpBuild to use.

Uninstall

Before running the tests the old version of our application needs to be uninstalled.

```
1    <Target Name="Uninstall"
2            Inputs="@(RunQtpInstallInput)"
3            Outputs="@(RunQtpInstallOutput)">
4
5       <!-- Launch the Uninstall in silent mode -->
6       <Exec Command="msiexec /x $(DOUBLE_QUOTES)$(QtpUninstallFolder)\$(MsiInstallFile)$(DOUBLE_QUOTES) /passive /l*
$(DOUBLE_QUOTES)$(QtpUninstallFolder)\Uninstall.log$(DOUBLE_QUOTES)"
7           ContinueOnError="true" />
8
9    </Target>
```

Observations:

- Lines 2-3: We use MSBuild's incremental builds feature to ensure that we do not uninstall the application if the latest version of the application is already installed on the machine. This is accomplished by comparing the timestamp of the **QtpLastInstallSucceededFile** to the timestamp of the latest msi package. If the msi package timestamp is later we know that we have a new version to install and thus need to do the uninstall.

- Line 4: The old version of our application is uninstalled by launching Windows Installer with the **/x** and **/passive** parameters. The /passive parameter ensures that no user interaction is required. We also log the uninstall output to a **uninstall.log** file. ContinueOnError is set to true for those scenarios where the machine hasn't got an installed version of our application

Install

Before running the tests the new version of our application needs to be installed. As mentioned earlier, the msi is copied into the **QtpInstallFolder** share by the Deploy target that runs as part of the DeploymentBuild.

```
1    <Target Name="Install"
2            Inputs="@(RunQtpInstallInput)"
3            Outputs="@(RunQtpInstallOutput)">
4
5       <Error Text="No QtpDeployFolder has been specified" Condition="'$(QtpDeployFolder)' == ''" />
6
7       <!-- Copy the latest install from the build server -->
8       <Copy SourceFiles="$(QtpDeployFolder)\$(MsiInstallFile)"
9             DestinationFolder="$(QtpInstallFolder)"/>
10
11      <!-- Launch the install in silent mode -->
12      <Exec Command="msiexec /i $(DOUBLE_QUOTES)$(QtpInstallFolder)\$(MsiInstallFile)$(DOUBLE_QUOTES) /passive /l*
$(DOUBLE_QUOTES)$(QtpInstallFolder)\Install.log$(DOUBLE_QUOTES) CLEAN_DATABASE=1"/>
13
14      <!-- Copy the msi to the Uninstall folder to use for later uninstallation -->
15      <Copy SourceFiles="$(QtpInstallFolder)\$(MsiInstallFile)"
16            DestinationFolder="$(QtpUninstallFolder)"/>
17
18      <Touch Files="@(RunQtpInstallOutput)" AlwaysCreate="true" />
19
20   </Target>
```

Observations:

- Lines 2-3: We use MSBuild's incremental builds feature to ensure that we do not uninstall the application if the latest version of the application is already installed on the machine. This is accomplished by comparing the timestamp of the **QtpLastInstallSucceededFile** to the timestamp of the latest msi package. If the msi package timestamp is later we know that we have a new version to install and thus need to do the install.
- Lines 8-9: The latest msi package is copied from the server to the local machine to use for installation.
- Line 12: The new version of our application is installed by launching Windows Installer with the **/i** and **/passive** parameters. The /passive parameter ensures that no user interaction is required. We also log the install output to an **install.log** file. Also notice the **CLEAN_DATABASE=1** statement that we pass onto the installer. As the installer is launched in passive mode, we need to set any installer options that would normally be selected by the user via the GUI now via the command line.
- Line 15-16: After installing the application, a copy of the msi is put into the **QtpUninstallFolder** to use for the next **Uninstall**.
- Line 18: We touch the timestamp of the **QtpLastInstallSucceededFile** to support future incremental builds.

Qtp Automation

Fortunately, the help file of Mercury's QTP product has some good examples of how to automate QTP using COM automation. I used extracts from their VBScript examples as a base for creating a **RunQTP.vbs** file that we use for doing our QTP automation. As QTP stores the result for every test in XML format, the idea is to create a single XML file that contains the combined results from all the individual tests and to merge this file into the CC.NET build report

```
Dim arTestCases, qcServer, qcUser, qcPassword
Dim testCasesFile, resultsDir, resultsSummaryFile
Dim fso, folder, subFolders, testResults, testCases
Dim qtApp, qtTest, qtResults
Dim testsPassed

testCasesFile = WScript.Arguments(0)
resultsDir = WScript.Arguments(1) & "\"
resultsSummaryFile = WScript.Arguments(2)
qcServer = WScript.Arguments(3)
qcUser = WScript.Arguments(4)
qcPassword = WScript.Arguments(5)

Set fso = CreateObject("Scripting.FileSystemObject")
Set testCases = fso.OpenTextFile(testCasesFile)
Set testResults = fso.CreateTextFile(resultsSummaryFile, True)

' Build an array of test cases to use
arTestCases = Split(testCases.ReadAll, vbCrLf)

testCases.Close
Set testCases = Nothing

' Launch QuickTest
Set qtApp = CreateObject("QuickTest.Application")
qtApp.Launch
qtApp.Visible = False
qtApp.Options.Run.RunMode = "Fast"
qtApp.Options.Run.ViewResults = False

Set qtResults = CreateObject("QuickTest.RunResultsOptions")

' Connect to Quality Center
qtApp.TDConnection.Connect qcServer, "CCNET.DEMO", "CCNet.Demo", qcUser, qcPassword, False
If Not qtApp.TDConnection.IsConnected Then
   WScript.Echo "Could not connect to Quality Center!"
   WScript.Quit 1
End If

testsPassed = True
```

```
testResults.WriteLine("<QtpResults>")

' Iterate through all the test cases
testsPassed = RunTests(arTestCases, resultsDir, testResults)

' Close the output file
testResults.WriteLine("</QtpResults>")
testResults.Close

Set fso = Nothing
Set testResults = Nothing

If Not qtApp Is Nothing Then
   ' Disconnect from Quality Center
   qtApp.TDConnection.Disconnect

   ' Close QuickTest Professional.
   qtApp.Quit
End If

' Release the created objects.
Set qtResults = Nothing
Set qtTest = Nothing
Set qtApp = Nothing

If (Err.Number > 0) Then
   WScript.Echo "Run-time error occured!! " & Err.Description
   WScript.Quit 1
ElseIf Not testsPassed Then
   WScript.Echo "Tests Failed!!"
   WScript.Quit 1
End If

Function RunTests(ByVal arTestCases, ByVal resultsDir, ByVal testResults)

Dim success, index

   On Error Resume Next
   success = True

   ' Loop through all the tests in the arTestCases.
   For index = 0 To UBound (arTestCases) - 1
      If Len(arTestCases(index)) > 0 Then
         ' Display Status
         WScript.Echo "Running Test Case: " & arTestCases(index)

         ' Open the test in read-only mode
         qtApp.Open arTestCases(index), True

         ' Get the test object and set the results location
         Set qtTest = qtApp.Test
         qtTest.Settings.Resources.ObjectRepositoryPath = "[QualityCenter] Subject\CCNet.Demo\Object Repository\CCNet.Demo_Obj_Rep.tsr"
         qtTest.Settings.Resources.Libraries.RemoveAll
         qtTest.Settings.Resources.Libraries.Add("[QualityCenter] Subject\CCNet.Demo\Functions\CCNet.Demo_Functions.txt")

         ' This statement specifies a test results location.
         qtResults.ResultsLocation = resultsDir & qtTest.Name

         ' Execute the test. Instruct QuickTest Professional to wait for the test to finish executing.
         qtTest.Run qtResults

         ' Display Status
         WScript.Echo vbTab & "Test Case finished: " & qtTest.LastRunResults.Status

         success = success And (qtTest.LastRunResults.Status = "Passed" Or qtTest.LastRunResults.Status = "Warning")

         ' Close the test.
         qtTest.Close

         ' Strip the unrequired results and append the test results to the overal result file
         FormatTestResults fso, testResults, qtResults.ResultsLocation
      End If
   Next

   RunTests = success

End Function

Sub FormatTestResults(ByVal fso, ByVal outputResults, ByVal testCaseDir)

Dim found, line, fileStream

   Set fileStream = fso.OpenTextFile(testCaseDir & "\Report\Results.xml")

   Do
      line = fileStream.ReadLine
      found = (InStr(1, line, "<Report", vbTextCompare) <> 0)
   Loop While (found = False)

   outputResults.WriteLine line
   Do
      outputResults.WriteLine(fileStream.ReadLine)
   Loop While (fileStream.AtEndOfStream = False)

```

```
135    fileStream.Close
136    Set fileStream = Nothing
137
138 End Sub
```

Observations:

- Lines 7-12: The script file receives the files and Quality Center (QC) credentials to operate on as command line arguments. The arguments include a file that contains the list of test cases to run; the directory to store the test results in; the results summary file that will be used to aggregate all the individual test results and lastly the QC server and QC user logon credentials.
- Lines 14-22: We use the **Scripting.FileSystemObject** to build an array that contains the names of all the test cases to run. As all the test cases are stored on our QC Server, we create a file that contains the server path of all the QC test cases to run, i.e. *[QualityCenter] Subject\CCNet.Demo\Executable Scripts\Create Intermediaries*. We store a single test case per line and we parse the file to create an array of test cases to run.
- Lines 25-38: We use the COM Automation model of QTP to start QTP and connect to the QC server. If we are unable to connect to the QC server, we exit the script with a non-zero exit code.
- Lines 41+48: The individual test results are added to a <QtpResults> tag within the QTP results summary file.
- Lines 44, 74-113: We iterate through the test cases contained within the array and run these using the QTP automation objects. For every QTP test, we open the test case, set the test result location and run the test to finally append the test result (stored in a Report\result.xml file) to the overall result summary file.
- Lines 111, 119-138: The xml file produced by QTP contains a lot of unrequired DTD information that we do not want to include in the test result summary file. We use the **FormatTestResults** procedure to strip this information from the file before appending the remainder of the test results to the overall test result summary file.
- Lines 66-68: If a Windows Script Error occurred, we indicate this to the calling process by exiting the script with a non-zero exit code.
- Lines 69-72: If one of the tests failed, we indicate this to the calling process by exiting the script with a non-zero exit code.

Qtp

As the **RunQTP.vbs** script file takes care of all of the QTP automation, all that we need to do is to invoke the script file using Windows Script Host.

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

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

   <Message Text="$(NEW_LINE)Running QTP Regression Tests" Importance="high"/>

   <CreateItem Include="$(QtpResultsFolder)\$(CCNetProject)">
      <Output TaskParameter="Include" ItemName="QtpOutputFolder"/>
   </CreateItem>

   <!-- Clear existing test results -->
   <RemoveDir Directories="$(QtpOutputFolder)"/>
```

```
15       <MakeDir Directories="$(QtpOutputFolder)"/>
16
17       <!-- Run the QTP tests -->
18       <Exec Command="cscript.exe /nologo $(DOUBLE_QUOTES)$(QtpTestCasesFolder)\$(CCNetProject).TestCases.txt$(DOUBLE_QUOTES)
$(DOUBLE_QUOTES)$(QtpOutputFolder)$(DOUBLE_QUOTES) $(DOUBLE_QUOTES)$(QtpOutputFolder)\$(QtpResultsSummaryFile)$(DOUBLE_QUOTES)
$(DOUBLE_QUOTES)$(QCServer)$(DOUBLE_QUOTES) $(QCUser) $(QCPassword)"
19             ContinueOnError="true">
20          <Output TaskParameter="ExitCode" ItemName="QtpExitCode"/>
21       </Exec>
22
23       <!-- Copy the QTP test results for the CCNet build before we possibly fail the build because of Qtp test failures -->
24       <CallTarget Targets="CopyQtpResults" />
25
26       <!-- Fail the build if any test failed -->
27       <Error Text="Qtp test error(s) occured" Condition=" '%(QtpExitCode.Identity)' != '0'"/>
28
29    </Target>
```

Observations:

- Lines 9-11: We create a unique test results output directory based on the name of the test suite that is being run.
- Lines 13-15: We clear the old test results by deleting the test results output folder.
- Line 18-21: Windows Script host is invoked to execute the **RunQTP.vbs** script file. The test cases file, test result folder and test results summary file as well as Quality Center credentials are passed as command line parameters. ContinueOnError is set to true and the result of the test run is stored in a **QtpExitCode** property.
- Line 24: The **CopyQtpResults** target (see below) is invoked manually to copy the test results for inclusion in the CC.NET build report.
- Line 27: We fail the build if any of the tests failed.

CopyQtpResults

After running Qtp, all that remains to do is to copy the test results summary file for CC.NET to merge into the build report.

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

   <CreateItem Include="$(CCNetArtifactDirectory)\$(QtpResultsSummaryFile)">
      <Output TaskParameter="Include" ItemName="ExistingQtpResults"/>
   </CreateItem>

   <Delete Files="@(ExistingQtpResults)"/>
   <Copy SourceFiles="$(QtpOutputFolder)\$(QtpResultsSummaryFile)"
         DestinationFolder="$(CCNetArtifactDirectory)"
         ContinueOnError="true"/>

</Target>
```

Observations:

- Line 2: We test on the **CCNetProject** property to determine if the build was invoked via CC.NET. The **CCNetProject** property is one of the properties created and passed on by CC.NET when using the MSBuild CC.NET task.
- Lines 4-8: The existing Qtp results summary file is deleted.
- Line 9: The new Qtp results summary file is copied to the **CCNetArtifactDirectory**.

CruiseControl.NET Configuration

One dilemma you will face with the CC.NET configuration is that the current deployment of CC.NET does not include any style sheets for formatting the QTP results to display as part of the Web dashboard reports. QTP itself uses different style sheets to format its own xml test results, so one option is to tweak their style sheets to conform to the CC.NET style sheet requirements. Another option is to use the style sheets included by Owen Rogers in a post on the CC.NET Google user group - search for a post with the title "*QuickTest Pro Integration*". Regardless of what style sheets you choose, remember to include these style sheets into the dashboard.config to display it on the webdashboard and also in either the ccservice.exe.config/ccnet.exe.config for publishing via the EmailPublisher.

The QtpServer uses the following CC.NET configuration. The config is really self explanatory if you are familiar with the different CruiseControl.NET configuration options

```
<!DOCTYPE cruisecontrol [
   <!ENTITY pub "<statistics>
            <statisticList>
               <statistic name='Qtp Success' xpath='sum(//QtpResults/Report/Doc/Summary/@passed)' />
               <statistic name='Qtp Warnings' xpath='sum(//QtpResults/Report/Doc/Summary/@warnings)' />
               <statistic name='Qtp Errors' xpath='sum(//QtpResults/Report/Doc/Summary/@failed)' />
            </statisticList>
         </statistics>

         <xmllogger />

         <email from='qtpserver@yourdomain.co.za' mailhost='smtp.yourdomain.co.za' includeDetails='true'>
            <users>
               <user name='BuildGuru' group='buildmaster' address='developer1@yourdomain.co.za'/>
               <user name='Developer2' group='developers' address='developer2@yourdomain.co.za'/>
               <user name='QtpTester' group='qtptesters' address='qtptester1@yourdomain.co.za'/>
            </users>
            <groups>
               <group name='developers' notification='change'/>
               <group name='buildmaster' notification='always'/>
               <group name='qtptesters' notification='always'/>
            </groups>
         </email>">

   <!ENTITY links "<externalLinks>
         <externalLink name='Wiki' url='http://wikiserver.yourdomain.co.za/wiki' />
```

```xml
      </externalLinks>">

   <!ENTITY build "<tasks>
         <msbuild>
            <executable>C:\WINDOWS\Microsoft.NET\Framework\v2.0.50727\MSBuild.exe</executable>
            <workingDirectory>C:\Projects\CCNet.Demo</workingDirectory>
            <projectFile>CCNet.Demo.proj</projectFile>
            <buildArgs>/noconsolelogger /v:normal /p:QCUser=fred;QCPassword=password</buildArgs>
            <targets>Uninstall,Install,Qtp</targets>
            <timeout>14400</timeout>  <!-- 4 hours -->
            <logger>C:\Program Files\CruiseControl.NET\server\ThoughtWorks.CruiseControl.MSBuild.dll</logger>
         </msbuild>
      </tasks>">

   <!ENTITY header "<webURL>http://qtpserver.yourdomain.co.za/ccnet</webURL>
      <category>RegressionTesting</category>
      <workingDirectory>_dir</workingDirectory>">

]>
<cruisecontrol>

   <project name="TestSuite1" queue="Qtp.CCNet.Demo" queuePriority="1">
      &header;
      <artifactDirectory>_dir\Builds\TestSuite1\Artifacts</artifactDirectory>

      <triggers>
         <projectTrigger serverUri="tcp://buildserver.yourdomain.co.za:21234/CruiseManager.rem" project="DeploymentBuild">
            <triggerStatus>Success</triggerStatus>
            <innerTrigger type="intervalTrigger" seconds="30" buildCondition="ForceBuild"/>
         </projectTrigger>
      </triggers>

      <state type="state" />

      <labeller type="defaultlabeller"/>

      &build;

      <publishers>
         <merge>
            <files>
               <file>_dir\Builds\TestSuite1\Artifacts\QtpResultsSummary.xml</file>
            </files>
         </merge>

         &pub;
      </publishers>
```

```
 74
 75       &links;
 76    </project>
 77
 78    <project name="TestSuite2" queue="Qtp.CCNet.Demo" queuePriority="2">
 79       &header;
 80       <artifactDirectory>_dir\Builds\TestSuite2\Artifacts</artifactDirectory>
 81
 82       <triggers>
 83          <projectTrigger serverUri="tcp://qtpserver.yourdomain.co.za:21234/CruiseManager.rem" project="TestSuite1">
 84             <triggerStatus>Success</triggerStatus>
 85             <innerTrigger type="intervalTrigger" seconds="30" buildCondition="ForceBuild"/>
 86          </projectTrigger>
 87       </triggers>
 88
 89       <state type="state" />
 90
 91       <labeller type="defaultlabeller"/>
 92
 93       &build;
 94
 95       <publishers>
 96          <merge>
 97             <files>
 98                <file>_dir\Builds\TestSuite2\Artifacts\QtpResultsSummary.xml</file>
 99             </files>
100          </merge>
101
102          &pub;
103       </publishers>
104
105       &links;
106    </project>
107
108 </cruisecontrol>
```

Observations:

- Lines 4-6: We use the Statistics provider to include some additional statistics for every build that includes the number of QTP tests that failed, number of tests that passed and the number of tests that raised warnings.
- Line 35: The build uninstalls and installs the new version of the application that has been copied onto a share on the local machine before running Qtp.
- Line 52-57: We setup TestSuite1 to fire once a successful DeploymentBuild has been completed.
- Line 82-87: We setup TestSuite2 to fire once a successful TestSuite1 has completed.
- Line 69+98: The combined test results summary file is merged into the CC.NET build report.

Using the above mentioned configuration it is quite possible to setup additional QTP builds on the same and different machines to create a grid of machines to run through all of your test cases more quickly. Every build will run through a different set of test cases as identified by different Qtp test case files. In our setup at work we have 3 machines running through all our test cases concurrently. I leave that as a simple exercise for the reader to complete.

Next Steps

This is probably the most complex part of the whole CI process, but thankfully the COM automation interface provided by QTP makes it not too difficult to automate the whole regression test run. The next and final post will highlight some community extensions that you can use to add some further panache to your CI Build to make it even more useful. 🙂

Part 8: 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 config, 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 some 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="Svn 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:

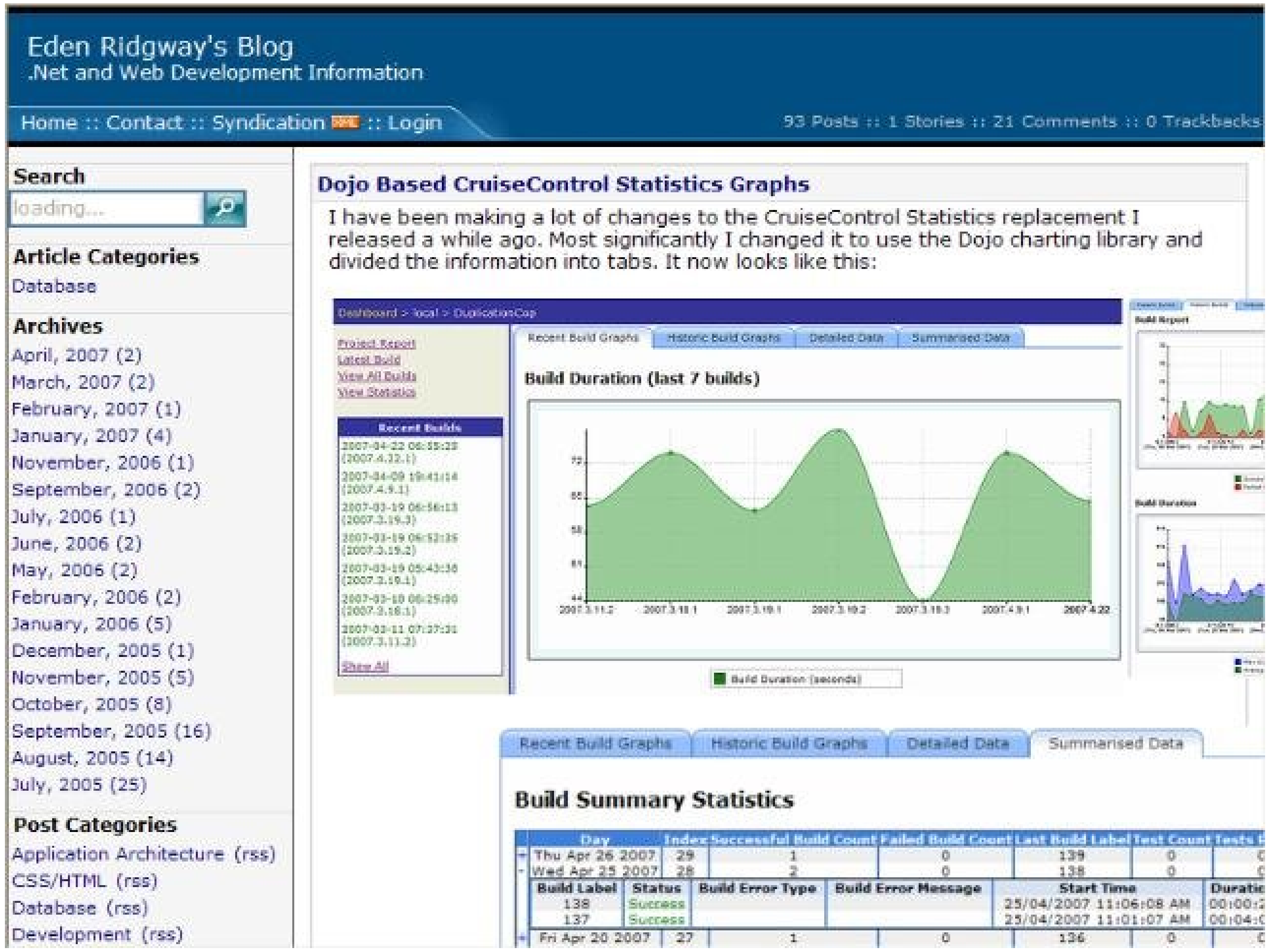
```
<statistics>
    <statisticList>
       <statistic name='TestCount' xpath='sum(//report-result/counter/@run-count)'/>
       <statistic name='TestFailures' xpath='sum(//report-result/counter/@failure-count)'/>
       <statistic name='TestIgnored' xpath='sum(//report-result/counter/@ignore-count)'/>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

Dashboard > local > DuplicationCop

Project Report
Latest Build
View All Builds
View Statistics

Recent Builds

2007-04-22 06:55:25 (2007.4.22.1)
2007-04-09 19:41:14 (2007.4.9.1)
2007-03-19 06:56:13 (2007.3.19.3)
2007-03-19 06:52:16 (2007.3.19.2)
2007-03-19 05:43:38 (2007.3.19.1)
2007-03-18 06:25:00 (2007.3.18.1)
2007-03-11 07:27:31 (2007.3.11.2)
Show All

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

Build Duration (last 7 builds)

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	Duratio
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

Hope you find these extensions useful 🙂

Final Remarks

Well, that concludes the series. I hope you find it to be a useful resource for assisting you with creating your own CI process. If you have any questions, additional remarks or suggestions, feel free to add it as comments to the individual blog posts or drop me an e-mail at carel.lotz@gmail.com.