

Continuous Integration with CruiseControl.Net

Part 3

Paul Grenyer

CruiseControl.Net

One of the first rules of writing is to write about something you know about. With the exception of the user guide for Aeryn [Aeryn] and Elephant [Elephant] I have never done this. I have always written about new techniques, operating systems and ideas that I have been exploring as the article develops. The article has been a key player in the learning process for me.

My articles about CruiseControl.Net [Part1][Part2] are no different. I'd only been using CruiseControl.Net on and off for about a month before I wrote the first article. I was no expert then and I'm still learning now.

So what's the main thing I have learnt? CruiseControl.Net [CCNet] has the potential to be a great application, but it's early days. There are a number of missing features, such as support for makefiles, sufficient error reporting from NAnt [NAnt] and, the feature I see asked for the most in the forums, a setting to remove and recreate the source code directory prior to version control checkout. This is all pretty basic stuff and I am at a loss as to why the writers of CruiseControl.Net have not added these features.

The CruiseControl.Net development team do have a patch submission procedure, but I need something that will work now and I don't want to use a non-standard build in the mean time. Luckily CruiseControl.Net does have a plug-in mechanism. The main missing feature, the ability to remove the source code directory prior to checkout, can be addressed by writing a plug-in.

In this article I'll describe how to create and install a plug-in for CruiseControl.Net, a couple of different ways to setup the appropriate Visual Studio project and write a new task block.

CruiseControl.Net Plug-ins

The limited instructions provided on the CruiseControl.Net website states the following:

1. Create a Class Library project to build the assembly that will contain your custom builder plug-in. The assembly that it produces should be named: 'ccnet.*.plugin.dll' (where the star represents the name you choose).
2. Add your new custom builder class.
3. The class must implement the `ThoughtWorks.CruiseControl.Core.ITask` interface (found in the `ccnet.core` assembly).
4. Mark your class with the `NetReflector.ReflectorType` attribute. The name argument supplied to the attribute is the name of the element/attribute that will appear in the configuration file.
5. Add any configuration properties that you need, marking them with the `NetReflector.ReflectorProperty` attributes accordingly. Note that the attribute names are case sensitive and must match exactly in the configuration.
6. Implement the `Run` method. The supplied `IntegrationResult` should provide you with everything that you need to know about the current build.
7. Compile the assembly.
8. Copy the assembly into the folder containing the `CruiseControl.NET` assemblies (or the current directory that you are running the `ccnet` server from).
9. Modify your `ccnet.config` file in accordance with the sample config file below.

Sample builder class:

```
using System;
using Exortech.NetReflector;
using ThoughtWorks.CruiseControl.Core;

namespace ThoughtWorks.CruiseControl.Sample.Builder
{
    [ReflectorType("mybuilder")]
    public class NAntBuilder : ITask
    {
        public void Run(IntegrationResult result)
        {
            Console.WriteLine("Hello World!");
        }
    }
}
```

Sample config File:

```
<cruisecontrol>
  <project name="myproject">
    <tasks>
      <mybuilder>
        <!-- include custom builder properties here -->
      </mybuilder>
    </tasks>
  </project>
</cruisecontrol>
```

Although the instructions are limited this is pretty much all there is to it. The only other details missing are how to setup a project and what the dependencies are. This is what I'll be covering next.

Setting Up a Visual Studio Project

This is an article about CruiseControl.Net and as Visual Studio is well understood by most Windows developers, I am not going to go into the details of creating the project. Instead I am going to concentrate more on the structure and dependencies of the project.

.Net Versions

When I started this article in May 2007 the current version of CruiseControl.net was 1.2.1.7 and used .Net 1.1. In the middle of June 2007 CruiseControl.Net 1.3, the first version to use .Net 2.0 was released. Plug-ins written in .Net 1.1 can of course be used by both versions, but plug-ins written in .Net 2.0 can only be used with version 1.3+. Equally, Microsoft Visual Studio 7.x can be used to write plug-ins for either version, but Microsoft Visual Studio 8.x can only be used to write plug-ins for version 1.3+.

Dependencies

CruiseControl.Net plug-ins are inherently dependent on the following assemblies:

ThoughtWorks.CruiseControl.Core.dll

This is the main CruiseControl.Net assembly and contains the classes and interfaces that make up the source control, task and publisher blocks.

NetReflector.dll

.Net reflector [NetReflector] provides the mechanism that CruiseControl.Net uses to take the XML parameters from `ccnet.config` and insert them into block class properties. It is a third party assembly, but supplied with CruiseControl.Net.

log4net.dll

Log4Net [Log4Net] is a well known third party logging library for .Net and this is exactly what CruiseControl.Net uses it for.

ThoughtWorks.CruiseControl.Remote.dll

The CruiseControl.Net `remote` assembly is not strictly a plug-in dependency. However, if the CruiseControl.Net `core` Visual Studio project is included in a solution the `remote` project must also be included as the `core` project references it.

Simple Solution

The easiest way to create a solution for a CruiseControl.Net plug-in is as follows:

1. Create an appropriately named C# assembly project and solution with Visual Studio. The name of the assembly must be in the form `ccnet.*.plugin.dll` where the star represents the name of the plug-in.
2. Create a `lib` folder and copy into it the assemblies that CruiseControl.Net is dependent on. Make sure you get the right versions of the assemblies for the version of .Net you are using. Both the release and source code zips of CruiseControl.Net contain all the required assemblies, and they are also located in the subversion repository.
3. Add the dependencies as references to the project.
4. Check the solution, the `lib` folder and its contents into a source control system.

Full Solution

CruiseControl.Net is an evolving project, so plug-ins should not be fixed to a completely static version and ideally space shouldn't be taken up in the source control system with pre-built assemblies.

CruiseControl.Net is one of the best organised projects I've seen. It uses a subversion [SVN] repository and releases are copied to the `tags` directory as recommended by the creators of subversion. Therefore if a plug-in project is also using subversion, a link can be created so that the appropriate CruiseControl.Net release tag is checked out as part of the plug-in working copy (see sidebar). This means that any patches can be picked up easily, space isn't taken up in the plug-in repository with the CruiseControl.Net code and when the next release comes out the tag link can be easily moved or added. If subversion is not being used the source from the appropriate CruiseControl.Net source code zip can be checked into the plug-in repository instead and used as a stable reference.

Sidebar: Subversion Links

The recommended way to organise a subversion repository is into three directories:

```
branches
tags
trunk
```

The main development goes on in `trunk`, any specialist branches are developed in `branches` and releases are copied to `tags`. CruiseControl.Net follows this recommendation. The final .Net 1.1 release is 1.2.1.7 and is located here:

```
http://ccnet.svn.sourceforge.net/svnroot/ccnet/tags/1.2.1.7
```

The current release (at time of writing) is 1.3.0.2918, uses .Net 2.0 and is located here:

```
http://ccnet.svn.sourceforge.net/svnroot/ccnet/tags/1.3.0.2918
```

Following the subversion recommendation a CruiseControl.Net plug-in should have the directory structure above and all development should be done in `trunk`. CruiseControl.Net plug-ins are dependent on the CrusieControl.Net assemblies and its third party assemblies. These can be committed to the repository or a link to the CrusieControl.Net release source can be created and the assemblies built from the source directory. The source code isn't actually copied into the plug-in repository, it is only checked out at the same time.

A good place to link the CruiseControl.Net source code is:

```
trunk/thirdparty/ccnet/
```

with a further subdirectory for .Net 1.1 version and the .Net 2.0 version:

```
trunk/thirdparty/ccnet/1.2.1.7
trunk/thirdparty/ccnet/1.3.0.2918
```

This enables you to build your plug-in for either .Net version just by pointing your build system at the appropriate sources.

There is no need to create the `thirdparty` directory or the `ccnet` directory. The version directories *must not* be created or a locked error will be given when the linked code is checked out. To create the link:

1. Open a command prompt and move to the `trunk` directory.
2. On Windows set the default text editor for Subversion:

```
set SVN_EDITOR=notepad
```

3. Execute the following command:

```
svn propedit svn:externals .
```

The full stop at the end is very important as it specifies that the property (see subversion manual for more details) is set on the current (`trunk`) directory. If the plug-in goes into continuous integration the chances are that only the `trunk` directory from the plug-in workspace will be checked out so the property must go there or the CruiseControl.Net source won't get checked out with it.

4. Enter the following into the default text editor instance that is launched and then save and close it:

```
thirdparty/ccnet/1.2.1.7 http://ccnet.svn.sourceforge.net/svnroot/ccnet/tags/1.2.1.7  
thirdparty/ccnet/1.3.0.2918 http://ccnet.svn.sourceforge.net/svnroot/ccnet/tags/1.3.0.2918
```

5. Execute `svn update` to checkout the CruiseControl.Net source.
6. Execute `svn ci -m"<suitable commit message>"` to commit the property change to the repository.

The property can be checked by executing `svn propget svn:externals` and should give the following output:

```
thirdparty/ccnet/1.2.1.7 http://ccnet.svn.sourceforge.net/svnroot/ccnet/tags/1.2.1.7  
thirdparty/ccnet/1.3.0.2918 http://ccnet.svn.sourceforge.net/svnroot/ccnet/tags/1.3.0.2918
```

If necessary the link can be removed by executing: `svn propdel svn:externals`.

Be very careful not to cancel a subversion command while setting a property or checking code out as this can cause repository locks that are very difficult to remove.

Assuming that either a link to a CruiseControl.Net release has been created or the source code checked into the plug-in repository the following steps should be taken to setup a CruiseControl.Net plug-in solution in Visual Studio:

1. Create an appropriately named C# assembly project and solution with Visual Studio. The name of the assembly must be in the form `ccnet.*.plugin.dll` where the star represents the name of the plug-in.
2. Create a second assembly project for unit tests called `ccnet.*.plugin.test.dll` where the star represents the name of the plug-in.

3. Add `core.csproj` from `<ccnet source>/project/core` and `Remote.csproj` from `<ccnet source>/project/Remote` to the solution.

Make sure you choose the CruiseControl.Net source directory that goes with version of .Net and Visual Studio that you are using:

.Net Version	CruiseControl.Net version	Visual Studio Version
1.1	1.2.1.7	7.x
2	1.3.0.2918	8.0

4. Add `core` as a *project* reference and `NetReflector.dll` from `<ccnet source>/lib` as an *assembly* reference to the `ccnet.*.plugin` project.
5. Add `ccnet.*.plugin` and `core` as a *project* reference and `NetReflector.dll` from `<ccnet source>/lib` as an *assembly* reference to the `ccnet.*.plugin.test` project.
6. Add `nunit.framework.dll` from `<ccnet source>/tools/nunit` as an *assembly* reference to the `ccnet.*.plugin.test` project.

The `nunit.framework` assembly from the CruiseControl.Net source should be used to ensure you have an assembly that uses the correct version of .Net.

7. Create an empty directory called `xls` at the solution root.

The `core` project has a post build step that copies some `xls` files from the root of the CruiseControl.Net solution to the output directory. It uses a Visual Studio environment variable to determine the root of the solution. Building the project from another solution confuses it. Without the `xls` directory the project will fail to build due to failure of its post build step. It's a hack, but it solves the problem.

The solution will now build. The CruiseControl.Net source code, unfortunately, generates a number of warnings. You might also want to create a NAnt script. This is useful if you're using, for example, Mono or something other than Visual Studio. NAnt is also useful if you want to build your plug-in for multiple versions of .Net.

Getting the Plug-in into Continuous Integration

This is an article series primarily about Continuous Integration, so naturally the first thing to do is to place the plug-in under Continuous Integration. In Part I of this series I described how to create a CruiseControl.Net configuration file with a project block, source control block, tasks blocks and an email publisher block. The project structure described above also requires an NUnit block.

NUnit Task Block

All software projects should be unit tested. CruiseControl.Net is one of the best and most completely unit tested projects I've seen. Any plug-in should also be unit tested. The solution creation described above includes the creation of a project for unit tests. Running unit tests as part of the build is an important part of Continuous Integration. CruiseControl.Net is intended primarily (but not exclusively) for .Net projects, therefore it has a block that supports the most well know and popular .Net testing framework: NUnit [NUnit].

The only required NUnit task block parameter is a list of assemblies to run. However, just specifying assemblies assumes that the `nunit-console.exe` executable, which is used to run the tests and to generate the output xml file containing the results, is in the path. There are a few different versions of NUnit. To make matters more complicated there are separate .Net 1.1 and .Net 2.0 versions and you cannot have them both installed at the same time. To ensure the correct version of NUnit is used it should also be checked into the repository. CruiseControl.Net includes NUnit in its repository and it can be accessed as shown below:

```
<nunit>
  <path>thirdparty\ccnet\1.3.0.2918\tools\nunit\nunit-console.exe</path>
  <assemblies>
    <assembly>ccnet.marauder.plugin.tests\bin\Debug\
      ccnet.marauder.plugin.tests.dll</assembly>
    <assembly>ccnet.marauder.plugin.tests\bin\Release\
      ccnet.marauder.plugin.tests.dll</assembly>
    <assembly>build\net-1.1\ccnet.marauder.plugin.tests.dll</assembly>
    <assembly>build\net-2.0\ccnet.marauder.plugin.tests.dll</assembly>
  </assemblies>
</nunit>
```

Bear in mind that assemblies created with both .Net 1.1 and .Net 2.0 can be run using the .Net 2 version of NUnit, but assemblies built with .Net 2.0 can only be run using the .Net 2.0 version.

There are also parameters for setting the name of the xml output file and the timeout. The result of running the tests can be viewed using the CruiseControl.Net Dashboard via the NUnit Details link in the Build view and are included in the email if an email publisher block is specified.



Complete CruiseControl.Net Configuration for CCMarauder

```
<cruisecontrol>
  <project name="CCMarauder">
    <workingDirectory>c:\temp\ccnet\ccmarauder\</workingDirectory>
    <artifactDirectory>c:\temp\ccnet\ccmarauder</artifactDirectory>
    <sourcecontrol type="svn">
      <trunkUrl>http://ccmarauder.tigris.org/svn/ccmarauder/trunk/</trunkUrl>
      <workingDirectory>c:\temp\ccnet\ccmarauder</workingDirectory>
      <executable>C:\Program Files\CollabNet Subversion\svn.exe</executable>
    </sourcecontrol>
    <tasks>
      <devenv>
        <solutionfile>ccnet.marauder.plugin.sln</solutionfile>
        <configuration>Debug</configuration>
        <executable>C:\Program Files\Microsoft Visual Studio .NET...
          ...2003\Common7\IDE\devenv.com</executable>
        <buildtype>Rebuild</buildtype>
        <buildTimeoutSeconds>300</buildTimeoutSeconds>
      </devenv>
      <devenv>
        <solutionfile>ccnet.marauder.plugin.sln</solutionfile>
        <configuration>Release</configuration>
        <executable>C:\Program Files\Microsoft Visual Studio .NET
          2003\Common7\IDE\devenv.com</executable>
        <buildtype>Rebuild</buildtype>
        <buildTimeoutSeconds>300</buildTimeoutSeconds>
      </devenv>
      <nunit>
        <path>C:\Program Files\NUnit 2.4.1\bin\nunit-console.exe</path>
        <assemblies>
          <assembly>ccnet.marauder.plugin.test\bin\Debug\
            ccnet.marauder.plugin.test.dll</assembly>
          <assembly>ccnet.marauder.plugin.test\bin\Release\...
            ccnet.marauder.plugin.test.dll</assembly>
          <assembly>build\ccnet.marauder.plugin.test.dll</assembly>
        </assemblies>
      </nunit>
    </tasks>
    <publishers>
      <xmllogger logDir="buildlogs" />
      <email from="paul.grenyer@gmail.com"
        mailhost="mailhost.zen.co.uk" includeDetails="TRUE">

      <users>
        <user name="Paul Grenyer" group="buildmaster"
          address="paul.grenyer@gmail.com"/>
        <user name="ccmarauder Developers" group="developers"
          address="continuousintegration@ccmarauder.tigris.org"/>
      </users>
      <groups>
        <group name="developers" notification="change"/>
        <group name="buildmaster" notification="always"/>
      </groups>
    </email>
  </publishers>
</project>
</cruisecontrol>
```

Adding a Source Control task to the Plug-in

Now that the infrastructure is in place, the development of the plug-in itself can begin.

The layout of a CruiseControl.Net plug-in ought to match the layout of the CruiseControl.Net project, for consistency if nothing else. Create a `sourcecontrol` folder in the `ccnet.marauder.plugin` project in Visual Studio (this will create a corresponding directory on the disk) and create an `Svn.cs` file in it. Check the file and folder into your version control system.

CruiseControl.Net uses Net Reflector [NetReflector] to allow its blocks to be dynamically instantiated from the XML in its configuration file and have the task's properties initialised with the values specified in the XML. Therefore a `Exortech.NetReflector` using directive is required.

Usually, custom tasks implement the `ITask` interface. However, we want to add functionality to an existing task. The ability to remove the source directory is missing from all CruiseControl.Net source control tasks. I am going to describe how to add it to the Subversion task and therefore will inherit the new task from that. The task also needs a name to identify it in the CruiseControl.Net configuration file. "marauder-svn" is sufficiently descriptive and unique. The name of the task is specified using the `ReflectorType` attribute. The basic class declaration looks like this:

```
using Exortech.NetReflector;

namespace ccnet.marauder.plugin.sourcecontrol
{
    [ReflectorType("marauder-svn")]
    class Svn : ThoughtWorks.CruiseControl.Core.Sourcecontrol.Svn
    {
        // ...
    }
}
```

Not everyone will want the new functionality so it needs to be easily enabled and disabled. The easiest way to do this is with a property based flag that is initialised from the configuration file. The `ReflectorProperty` attribute is used for this:

```
[ReflectorType("marauder-svn")]
class Svn : ThoughtWorks.CruiseControl.Core.Sourcecontrol.Svn
{
    private bool removeWorkingDir = false;

    [ReflectorProperty("removeWorkingDirectory", Required=false)]
    public bool RemoveWorkingDir
    {
        get { return removeWorkingDir; }
        set { removeWorkingDir = value; }
    }

    // ...
}
```

At this stage it's helpful and useful to add a test. Create a `sourcecontrol` folder in the `ccnet.marauder.plugin.tests` project in Visual Studio (this will create a corresponding directory on the disk) and create an `SvnTest.cs` file in it. Check the file and folder into your version control system.

All CruiseControl.Net blocks have tests that check that settings defined by the XML in the configuration file are correctly initialised in the blocks' properties. A number of using directives are needed in order to write the test:

```
using ccnet.marauder.plugin.sourcecontrol
Defines marauder-svn.
```

```
using ThoughtWorks.CruiseControl.Core.Util
Defines CruiseControl.Net utility classes such as Time.
```

```
using Exortech.NetReflector
Defines classes to take XML configuration and inserts it into block class properties.
```

```
using NUnit.Framework
Defines classes and attributes for the NUnit framework.
```

The test fixture and test function are defined as follows:

```
using ccnet.marauder.plugin.sourcecontrol;
using ThoughtWorks.CruiseControl.Core.Util;
using Exortech.NetReflector;
using NUnit.Framework;
```

```
namespace ccnet.marauder.plugin.tests.sourcecontrol
{
    [TestFixture]
    public class SvnTest
    {
        [Test]
        public void PopulateFromFullySpecifiedXml()
        {
            // ...
        }
    }
}
```

This is all that is required to create a test and, even though it does not actually test anything yet, if it is committed to source control, continuous integration will pick it up and run the test. Try it.

One thing worth testing is that the base class (`*.Core.Sourcecontrol.Svn`) properties are still populated correctly. The easiest way to do that is to copy the SVN CruiseControl.Net test into the plug-in test:

```
[Test]
public void PopulateFromFullySpecifiedXml()
{
    string xml = @"
        <marauder-svn>
            <executable>c:\svn\svn.exe</executable>
            <trunkUrl>svn://myserver/mypath</trunkUrl>
            <timeout>5</timeout>
            <workingDirectory>c:\dev\src</workingDirectory>
            <username>user</username>
            <password>password</password>
            <tagOnSuccess>true</tagOnSuccess>
            <tagBaseUrl>svn://myserver/mypath/tags</tagBaseUrl>
            <autoGetSource>true</autoGetSource>
        </marauder-svn>";

    Svn svn = (Svn) NetReflector.Read(xml);
    Assert.AreEqual(@"c:\svn\svn.exe", svn.Executable);
    Assert.AreEqual("svn://myserver/mypath", svn.TrunkUrl);
    Assert.AreEqual(new Timeout(5), svn.Timeout);
    Assert.AreEqual(@"c:\dev\src", svn.WorkingDirectory);
    Assert.AreEqual("user", svn.Username);
    Assert.AreEqual("password", svn.Password);
    Assert.AreEqual(true, svn.TagOnSuccess);
    Assert.AreEqual(true, svn.AutoGetSource);
    Assert.AreEqual("svn://myserver/mypath/tags", svn.TagBaseUrl);
    Assert.AreEqual(true, svn.RemoveWorkingDir);
}
```

The code above is not an exact cut and paste from CruiseControl.Net test. The root XML tag has been modified so that it instantiates the Svn block from the plug-in, rather than the CruiseControl.Net block and a new test has been added to test the value of RemoveWorkingDirectory. The default value of removeWorkingDirectory is false, so currently the test will fail. Commit the file to source control to see this.

To fix this a new element needs to be added to the XML. The xml string defined in the test function represents, although is not the same as, the XML used in the CruiseControl.Net configuration file. This is where the element must be added:

```
string xml = @"
    <marauder-svn>
      <executable>c:\svn\svn.exe</executable>
      <trunkUrl>svn://myserver/mypath</trunkUrl>
      <timeout>5</timeout>
      <workingDirectory>c:\dev\src</workingDirectory>
      <username>user</username>
      <password>password</password>
      <tagOnSuccess>true</tagOnSuccess>
      <tagBaseUrl>svn://myserver/mypath/tags</tagBaseUrl>
      <autoGetSource>true</autoGetSource>
      <removeWorkingDirectory>true</removeWorkingDirectory>
    </marauder-svn>";
```

Commit the modification to source control to see the tests pass.

CruiseControl.Net blocks also have tests for the minimum required XML:

```
[Test]
public void SpecifyFromMinimallySpecifiedXml()
{
    string xml = @"<marauder-svn/>";
    svn = (Svn) NetReflector.Read(xml);
    Assert.AreEqual("svn.exe", svn.Executable);
    Assert.AreEqual(false, svn.RemoveWorkingDir);
}
```

One test that is missing from CruiseControl.Net is the element present in the XML, but with a value of false. For completeness this should also be added to the plug-in tests:

```
[Test]
public void SpecifyFalseInXml()
{
    string xml = @"
    <marauder-svn>
      <removeWorkingDirectory>false</removeWorkingDirectory>
    </marauder-svn>";
    Svn svn = (Svn)NetReflector.Read(xml);
    Assert.AreEqual(false, svn.RemoveWorkingDir);
}
```

Implementing the Task Block

As shown in the sample builder class above, a task block is usually implemented by the `ITask` interface. The tasks that the task block performs go into the `Run` method. Our custom Subversion source control block does not implement `ITask`. It implements `ISourceControl` instead, which is the equivalent interface for source control blocks:

```
namespace ThoughtWorks.CruiseControl.Core
{
    [TypeConverter(typeof(ExpandableObjectConverter))]
    public interface ISourceControl
    {
        Modification[] GetModifications( IIntegrationResult from,
                                         IIntegrationResult to);

        void LabelSourceControl(IIntegrationResult result);
        void GetSource(IIntegrationResult result);

        void Initialize(IProject project);
        void Purge(IProject project);
    }
}
```

We need to modify the standard behaviour of the CruiseControl.Net Subversion source control block so that the working directory is removed before the source is checked out. Looking at the CruiseControl.net Subversion class reveals that the source is checked out in the following method:

```
public override void GetSource(IIntegrationResult result)
{
    if (! AutoGetSource) return;

    if (DoesSvnDirectoryExist(result))
    {
        UpdateSource(result);
    }
    else
    {
        CheckoutSource(result);
    }
}
```

This method is virtual (this is indicated by the use of the `override` keyword and the lack of a compiler error), so all we need to do is override this method in our subclass, add our functionality and then call the above `GetSource` method. There is one further consideration. Looking at the above method closely reveals that if `AutoGetSource` is set to `false`, the method does nothing. Our overridden method needs to act in the same way:



```
[ReflectorType("marauder-svn")]
public class Svn :
    ThoughtWorks.CruiseControl.Core.Sourcecontrol.Svn
{
    // ..

    public override void GetSource(IIntegrationResult result)
    {
        if (AutoGetSource)
        {
            // Add new functionality here.
            base.GetSource(result);
        }
    }
}
```

Before the working directory can be removed, we need to know what it is. Several CruiseControl.Net task blocks have a working directory property. Usually if this is an absolute path it is used as is. If it is a relative path then the project working directory is appended to the beginning. The CruiseControl.Net subversion source control block has an example of how to create the working directory:

```
string workingDirectory =
    result.BaseFromWorkingDirectory(WorkingDirectory);
```

WorkingDirectory is a property of the subclass. result is a reference to an object that implements the IIntegrationResult interface and gives access to all sorts of project settings, including the global working directory path, and is passed to all task blocks.

Once the working directory path has been ascertained, RemoveWorkingDirectory needs to be checked and if true the working directory removed:



```
using System.IO;

// ...

public override void GetSource(IIntegrationResult result)
{
    if (AutoGetSource)
    {
        string workingDirectory =
            result.BaseFromWorkingDirectory(WorkingDirectory);

        if (RemoveWorkingDir)
        {
            Log.Info("Removing working directory.");
            DeleteDirectory(new DirectoryInfo(workingDirectory));
        }

        base.GetSource(result);
    }
}

private void DeleteDirectory(DirectoryInfo dirInfo)
{
    foreach (DirectoryInfo subDirInfo in dirInfo.GetDirectories())
    {
        DeleteDirectory(subDirInfo);
    }

    foreach (FileInfo fileInfo in dirInfo.GetFiles())
    {
        FileAttributes fileAttri =
            File.GetAttributes(fileInfo.FullName);
        if ((fileAttri & FileAttributes.ReadOnly) != 0)
        {
            File.SetAttributes(fileInfo.FullName, fileAttri &
                ~FileAttributes.ReadOnly);
        }

        Log.Debug(string.Format("Removing: {0}", fileInfo.FullName));
        File.Delete(fileInfo.FullName);
    }

    Log.Debug(string.Format("Removing: {0}", dirInfo.FullName));
    Directory.Delete(dirInfo.FullName);
}
}
```

The details of the `DeleteDirectory` method are beyond the scope of this article, but explained in detail in [Visiting Files and Directories in C# \[VFDC#\]](#). `Log` is a `log4net` logger available in a base class.

As mentioned previously, `CruiseControl.Net` source control blocks do not create the source code directory if it does not exist. Adding this functionality is straight forward:

```
public override void GetSource(IIntegrationResult result)
{
    if (AutoGetSource)
    {
        string workingDirectory =
            result.BaseFromWorkingDirectory(WorkingDirectory);

        if (RemoveWorkingDir)
        {
            Log.Info("Removing working directory.");
            DeleteDirectory(new DirectoryInfo(workingDirectory));
        }

        if (!DirectoryExists(workingDirectory))
        {
            Log.Info("Creating working directory.");
            CreateDirectory(workingDirectory);
        }

        base.GetSource(result);
    }
}

...

private bool DirectoryExists(string path)
{
    return Directory.Exists(path);
}

private void CreateDirectory(string path)
{
    Directory.CreateDirectory(path);
}
```

That completes the implementation of our custom Subversion source control block. The next step is to add more unit tests. However, this would involve:

- Applying an extract method to `GetSource` so that the new functionality can be called without calling the base class version of `GetSource`.
- Creating a class to call the `File` and `Directory` methods and extracting its interface so that a mock can be passed in when testing.

This is quite involved with a number of steps and outside the scope of this article.

Using the plug-in with CruiseControl.Net

Finally the ultimate test: plugging the plug-in into a CruiseControl.Net server. To use, and test, the plug-in with CruiseControl.net:

1. Stop the service or command line running
2. Copy the plug-in assembly to the CruiseControl.Net service directory.
3. Edit the `ccnet.config` file (see below).
4. Restart the service or command line.

Adding a new task to the `ccnet.config` file is described at the beginning of this article. Using a source control task block is slightly different. The standard SVN block for the Aeryn project described in Part 1, is as follows:

```
<sourcecontrol type="svn">
  <trunkUrl>http://aeryn.tigris.org/svn/aeryn/trunk/</trunkUrl>
  <workingDirectory>c:\temp\ccnet\aeryn\working</workingDirectory>
  <executable>C:\Program Files\...\csvn.exe</executable>
</sourcecontrol>
```

It can be modified to use the plug-in like this:

```
<sourcecontrol type="marauder-svn">
  <trunkUrl>http://aeryn.tigris.org/svn/aeryn/trunk/</trunkUrl>
  <workingDirectory>c:\temp\ccnet\aeryn\working</workingDirectory>
  <executable>C:\Program Files\...\csvn.exe</executable>
  <removeWorkingDirectory>true</removeWorkingDirectory>
</sourcecontrol>
```

Changing the source control type to the attribute used to identify the plug-in SVN class tells CruiseControl.Net to load it instead of the standard SVN task. Adding the `removeWorkingDirectory` tags tells `marauder-svn` to invoke the new functionality. That's all there is to it.

There are a number of ways to test it. One easy one is to tail the CruiseControl.Net log (`ccnet.log` in the service directory), force a build and watch for the log messages:

```
[Aeryn:INFO] Building: Paul Grenyer triggered a build (ForceBuild)
[Aeryn:INFO] Removing working directory.
[Aeryn:INFO] Creating working directory.
[Aeryn:INFO] Starting build: ...
```

Another way to test is to open windows explorer (or equivalent) on the working directory and watch the source disappear and get rechecked out following a force a build.

Although the unit tests mean we should be confident that the new source control task works as expected, it is also worth checking what happens when `removeWorkingDirectory` is set to `false` and what happens when it is removed and the default value (`false`) is used.

Acknowledgments

Thank you to Jez Higgins for the pointers on `svn propset` and reviews. Thank you to Peter Hammond and Adrian Fagg for review and Caroline Hargreaves for proof reading.

References

[Aeryn] <http://www.aeryn.co.uk/>
[Elephant] <http://elephant.tigris.org/>
[Part 1] Integration with CruiseControl.Net – Part 1:
<http://www.marauder-consulting.co.uk/articles>
[Part 2] Integration with CruiseControl.Net – Part 2:
<http://www.marauder-consulting.co.uk/articles>
[CCNet] <http://ccnet.thoughtworks.com/>
[make] <http://www.gnu.org/software/make/>
[NAnt] <http://nant.sourceforge.net/>
[NetReflector] <http://sourceforge.net/projects/netreflector/>
[Log4Net] <http://logging.apache.org/log4net/>
[SVN] <http://subversion.tigris.org/>
[CCMaruader] <http://ccmaruader.tigris.org>
[mono] <http://www.mono-project.com/>
[KiwiDude] <http://www.kiwidude.com/dotnet/DownloadPage.html>
[NUnit] <http://www.nunit.org/>
[VFDC#] Visiting Files and Directories in C#:
<http://www.marauder-consulting.co.uk/articles>