

How to setup a .NET Development Tree

Mike Roberts - ThoughtWorks. - <http://mikeroberts.thoughtworks.net/blog/>

In my career I've setup quite a few development trees for .NET projects. What do I mean by a 'development tree'?

- It is a directory structure
- containing:
 - source files
 - tools and dependencies
 - references to external tools and dependencies
- checked into source control
- that is atomically integratable
- to produce a set of artifacts

A good development tree should:

- be easily integratable on new environments
- require little maintainance
- but be easily maintainable when it does require maintenance
- support, but not hamper, developer productivity
- have consistent behaviour

These are rather qualitative ideals, but give some direction about where we want to head. In this article I show how to develop a good 'boilerplate' development tree structure for .NET projects that other people can use.

Part 1 - Setting up Source Control

So, lets start building our development tree. Feel free to join in. :)

The first thing you need is a Source Control environment. This may sound simple, but even at this stage I have seen some strange things happen on projects.

Here are some 'must haves':

- Your source control server **must** be fast. Your developers are often going to be waiting for your source control to do things, so don't scrimp on hardware. Specifically:
 - **Do** use decent, modern, hardware.
 - **Don't** use network shares to store files - in my experience it will slow your source control by about 10x. Instead invest in some locally redundant disks (RAID 5 is OK, RAID 0+1 is better), and a backup strategy.
 - **Don't** put your Source Control server on the other side of the world from your team. Keep it local, and make sure your network isn't getting bogged down. Obviously with distributed

teams this may not be possible, but if your team isn't distributed, don't distribute your hardware.

- **Don't** be tight on hard disk space. Get about as much as you think you might need in 3-5 years. Disk space **really is** cheap and having lots of it means that people can worry about producing software, and not about whether they are going over quota.
- Give developers write access to the code they need to work on. If you trust them to write code, you should trust them to be able to edit their own work without having to go through slow processes. Other team's code may be a different matter.
- Put each development tree in its own folder under source control - don't try and 'save' work or space by merging them. It really will save you headaches and time. See the 'hard disk space' point.
- Make sure new source control clients can be set up **fast** and correctly. Document what needs to be done for each project on a Wiki. If your Source Control Client setup takes over 10 minutes, or is more than a page of manual work, change it. If necessary throw away your current Source Control software and start again.
- Make sure basic source control operations are quick, simple and well understood. All developers should be easily able to do all of the following operations - if they don't know how or if these processes are cumbersome or slow to execute, then change them (again, if necessary consider changing your Source Control software)
 - Check out from nothing
 - Get updates
 - Find differences between server and local versions
 - Revert local versions
 - Commit changes
- Your Source Control system must be **consistently trustworthy** - if developers are losing changes or files are becoming corrupted, fix it.
- Your Source Control server should support the following more advanced operations, which developers should be able to perform if necessary:
 - Labelling (tagging)
 - Branching (parallel, independent, development of integratable code lines)
 - Automation (be driven by a process, not just a person)

The above points I believe are all necessary for an effective development project. For an 'excellent' project I recommend the following:

- A good source control server can happily accommodate 100 developers. I recommend the following kind of system:
 - UNIX/Linux based - Most good Source Control software is written primarily for a UNIX/Linux environment so don't support edge cases.
 - At least dual-CPU (I like the idea of one CPU being able to do work, and one doing I/O, but I'm sure that's rather a simplistic model these days)
 - At least 1GB RAM - if your often-accessed source is already cached you should get a speed up.

- Don't run anything else on the machine apart from a Source Control server. If you do (e.g. source control reporting), invest in extra processors and monitor what impact those extra applications are having.
- Use 1 disk set for applications and checkpoints/journals, and a separate disk set for your actual data.
- If cash is fairly easily available in your organisation, use [Perforce](#). I've been using it on and off for 4 years now and it never ceases to amaze me how fast and stable it is. It also requires almost zero maintenance.
- Otherwise use [Subversion](#). It is free, and better than any other SCM system I've tried apart from Perforce.
- If you are using Visual SourceSafe, I strongly urge you to migrate away from it. It is renowned for not being scalable and is also prone to file corruption. If you are not experienced with UNIX, or any other SCM tool apart from VSS, I have heard good things about [SourceGear Vault](#).
- Use clean and simple setups for your 'meta' trees. In Perforce, putting all projects in one 'depot' is perfectly reasonable, and use similar ideas for other tools.

If after reading all of this you are thinking 'Nice ideas, but we don't have the time or money to do any of this', then think how much it would really cost you to (say) invest in a new Linux server and Subversion, and how much money you are losing through lack of productivity. Its also a lot simpler than you think. Why not try out Subversion for half a day with a [good book](#)?

In the next part we'll start looking at some code.

Part 2 - Setting up the Solution

In Part 1 we looked at making sure we had our Source Control story straight. With that sorted out, we can start creating some files to put in it.

First some terminology, I'm going to use the word 'Project' to define the thing that all the files in our development tree go to make up. It is more than just a Visual Studio Project. I'm going to use an example project called *Sycamore*.

Next, I'm going to assume you are using Visual Studio 2003. Pretty much everything we are going to look at will work without Visual Studio, but I'll assume you have it anyway.

First, we want to make a new folder. We'll put it in our **meta root** - a place where you check projects out from Source Control. On my machine, this folder is C:\devel but on your machine (and anyone else's) it might be different. You should never assume the concrete location of **meta roots**.

Call the new folder pretty much the same thing as your project. I say 'pretty much' since I like to remove capitals and spaces, but its really up to you. Our folder will be called sycamore. It is the **root** of our development tree. All source code for this project will exist somewhere under this **root**. Any tool or library dependencies that exist outside the scope of this **root** will have to be managed carefully. There will be **no** source under this **root** that belongs to any other project.

To start with we are just going to create a Visual Studio-compilable 'solution'. A solution contains source code, so we're going to create a sub-folder of sycamore called src. We will have other sub-directories later, but they will contain other things, so its good to separate the source out into its own location.

In src we create our new Visual Studio solution. To make things easy, I'm going to call it *Sycamore*. Unfortunately, Visual Studio **doesn't** make it easy - it wants to put it in a another sub-folder called Sycamore so once I've created an empty solution I'll close it down and move the Solution.sln file into the src folder. We can delete the extra Sycamore folder that Visual Studio created for us.

Next to create some VS Projects. In this part, I'm going to keep it simple - just a single command line application in one project. We'll be creating some more projects later. I like projects to have their own folder under src. We never merge VS projects into one folder and never put their 'project roots' anywhere other than src.

In Visual Studio I create a C# Console Application called *SycamoreConsole*. Its location on my machine is C:\devel\sycamore\src, but the location on

your's will depend on your **meta root**. VS creates the project for us and creates a class called *Class1*.

We're also going to change some of the project properties. First the Assembly Name. In later parts we'll talk about dll names, but for console applications, pick something short and obvious. We're going to call our's sycamore. For the Default Namespace, I like to use the convention *OrganisationName.ProjectName.VSProjectName*, so in our case I'm going to use the Namespace *SherwoodForest.Sycamore.SycamoreConsole*. Save these properties and go back to the *Class1* window.

First, set the namespace to be as you just set in the Project properties. Now rename the class to something sensible (we'll use HelloWorld for now), and don't forget to rename the file to match. (I recommend you use [ReSharper](#) which will do the file rename for you.) I also like to delete all the unnecessary comments. Sticking with tradition we'll add the statement `Console.WriteLine("Hello World");` to the main method. Compile, run and make sure everything does as expected.

We're done for now. We may be making baby steps, but we are already seeing some definitions and patterns emerge:

- The **root** is the upper most point of our development tree.
- All files belonging to a project exist under the **root**.
- No files belonging to any other project exist under the **root**.
- The **root** itself is resident in a **meta root** which can change from machine to machine.
- All source code resides under a src sub-folder.
- The project .sln file is saved in the src folder.
- All Visual Studio Projects exist in their own sub-folders under the src folder.
- Visual Studio Project folders are atomic, and should be named identically to the project they contain.
- The default namespace for a Visual Studio project should be *OrganisationName.ProjectName.VSProjectName*.

In the next part we'll look at what we have to do to get this project into Source Control.

Part 3 - Adding files to Source Control

A quick recap. So far we have made sure we have a good source control environment and have created a Visual Studio Solution with a well structured folder setup. But we haven't checked those files into our Source Control server yet - we'd better fix that.

Your Source Control administrator will probably tell you where to make your initial check-in of your new project, but I suggest you think about **simplicity** for a moment:

- If you're using Perforce, consider using 1 depot as the 'server side' equivalent of your meta root. You don't lose any security options, and you gain in that developers may already have this depot mapped in their client so won't need to change any source control configuration.
- If you're using Subversion, just use one repository for all the projects in your department (see [here](#) for a good explanation why.) Use a new directory for your new project (and probably check it in to a 'trunk' sub-directory, but you can always move it later.)
- If you're using CVS, its fairly standard to create a new CVSROOT for each project, and I would recommend it. Note that you'll have to setup any extra permissions and triggers that you use as standard. I've seen organisations make good use of [GForge](#) to manage their CVS server.
- For other Source Control systems, follow similar guidelines

Once you've figured out the source control location of your new project, don't be too hasty about checking in. Its worth taking a moment to decide what you actually want to check in. Files you **don't** want to include are:

- Build output folders - don't check in the bin or obj sub-folders of your VS project folders
- Any Solution .suo or VS Project .user files - these are user and environment specific and should not be checked in
- Any Resharper, or other third-party tool output. (Resharper generates a SolutionName.resharperoptions file and a _ReSharper.SolutionName folder, neither of which you need to save)

Not checking these files in is good, but making sure no-one else ever checks them in later by mistake is even better. CVS and Subversion both offer such functionality through .cvsignore files and svn:ignore properties respectively. With Perforce, you can use [Triggers](#), but this is not as elegant a solution.

Moving back to our *Sycamore* example, I'm going to use a Subversion server to check in our work. First of all I delete all the temporary files we discussed above. Then I'm going to use the svn command line tool, but you could use [TortoiseSVN](#) or [AnkhSVN](#) instead. My command line looks like:

```
c:\devel\sycamore>svn import -m "Initial Sycamore Import" .  
file:///c:/svn-repos/sycamore/trunk
```

Once the initial checkin is complete I'm going to delete my 'sycamore' folder and then checkout from Subversion the folder we just imported to get a local versioned folder. After that I reload the solution in Visual Studio and compile. This recreates the temporary files.

I then set the svn:ignore value for src to be *.suo, ReSharper.Sycamore and *.resharperoptions. The svn:ignore for VS Project dirs should be set to *.user, bin and obj. You should be able to test you've captured everything by doing a svn status in the root folder and only seeing output for merging the properties of the src and VS Project directories. Make sure to commit these property updates.

To see exactly the state of Sycamore as it currently stands, download a zip file from [here](#).

To summarise this part:

- Pick a Source Control location that is simple for everyone to use.
- When checking in your project directory, make sure not to include build artifacts or temporary environment files.
- If possible, configure your Source Control to make sure no-one can check in such files in the future.

In the next part we'll be adding an automated build for our project using NAnt.

Part 4 - Adding an Automated Build

At this point we have a basic Visual Studio solution checked into Source Control. Now its time to automate how we build this solution.

Most of the time .NET developers will work solely within the Visual Studio environment, compiling their solution with the in-built compiler, and running tests using [TestDriven.NET](#) (more on testing to come...). But relying solely on Visual Studio as a way to produce build artifacts and run your tests isn't enough. For instance:

- How do you run scheduled or triggered builds for your project? Using the command line version of Visual Studio (devenv.com) provides you with only basic command line features.
- Visual Studio's 'pre-' and 'post-' build events provide some build scripting beyond just compiling code, but such scripting is limited in scope and expressiveness.

The current 'de-facto' automated build tool for .NET projects is [NAnt](#). NAnt is based on the Java build tool [Ant](#) and has similar strengths (integration with lots of useful tools, few dependencies) and also its weaknesses (being defined in XML means large build scripts quickly get hard to maintain). .NET 2 and Visual Studio 2005 will come with their own build scripting tool, [MSBuild](#), which is very similar to NAnt. Investing in NAnt now should give you a build script you can easily convert to MSBuild later, should you want to.

NAnt is a tool that can be installed on every developers machine. However, I like to check NAnt into the project tree for some simple reasons:

- It saves the manual steps of everyone copying it to their machine, and installing it. (Remember - manual steps take time and are a possible point of error.)
- NAnt changes between versions, and such changes can effect the behaviour of a build. Making sure that everyone has the same version of NAnt when everyone is manually installing it can be tricky, and is time consuming when you want to upgrade the version of NAnt everyone uses.
- Many projects use their own 'custom' NAnt tasks. Storing these in source control along with the project's own version of NAnt makes distribution to team members painless.
- It is not a large tool, so the overhead of storing it in source control should not be a problem.

To add NAnt to your project tree, first download and unpack its [binary zip file](#) (I'm going to use NAnt 0.85 RC1, available [here](#).) Then, copy the bin folder to your project directory. I like to put all build-time tools in a sub-folder of my project root called tools, and then put the contents of NAnt's

bin folder in tools\nant. Before going any further, commit NAnt to your project's source control, making sure to include in the commit message the version of NAnt you are using. Later on, this will help you decide whether you want to upgrade to a new version.

You tell NAnt what to do using a *build script*. The standard for naming NAnt build scripts is *ProjectName.build*. The build script is a gateway into our project, so I like to save it in the root folder. You can edit your build script with Visual Studio - create it as a 'solution item' (Right click on the solution icon in Solution Explorer and choose *Add new item...* or *Add existing item...*). If you follow the instructions [here](#) and [here](#) you'll even get IntelliSense! (Thanks to Serge van de Oever and Craig Boland for writing it up.)

Our first NAnt build script will just compile our project. There are several ways to do this, and I'm going to use the [<solution>](#) task:

```
<?xml version="1.0" ?>
<project name="nant" default="compile"
xmlns="http://nant.sf.net/schemas/nant.xsd">
  <target name="compile">
    <solution solutionfile="src\Sycamore.sln"
configuration="debug" />
  </target>
</project>
```

I like to use the [<solution>](#) task for a couple of reasons:

- For developers to work in Visual Studio we need to define how to compile our project in Visual Studio using its 'references' system. [<solution>](#) lets us re-use all this work in 1 line of script. If we were to use the [<CSC>](#) task instead we would need to maintain a separate set of compile definitions (which would be time-consuming and might not match the Solution / VS Project setup).
- Using [<solution>](#) rather than the [<exec>](#) task calling out to devenv.com is less resource intensive, gives more appropriate feedback and also allows us to run builds on machines without Visual Studio installed (it just needs the .NET SDK.) If you have a problem using [<solution>](#) you can always quickly replace it with an [<exec>](#) to devenv.com

To run your build, save the build script, open a command prompt and change to your project's root folder. Then just enter tools\nant\NAnt. You should see output like:

```
NAnt 0.85 (Build 0.85.1793.0; rc1; 28/11/2004)
Copyright (C) 2001-2004 Gerry Shaw
http://nant.sourceforge.net
Buildfile: file:///c:/devel/sycamore/Sycamore.build
Target(s) specified: compile
```

```
compile:  
[solution] Starting solution build.  
[solution] Building 'SycamoreConsole' [debug] ...
```

BUILD SUCCEEDED

Total time: 0.2 seconds.

Woohoo - a successful build! We have something new, that works, so submit the build script (and your changes to the Solution file that include the build script) to source control.

The current state of Sycamore is available [here](#).

To summarise this part:

- Add an automated build system to your project.
- Use NAnt to automate your .NET 1.1 and earlier projects.
- Check the NAnt distribution into your development tree.
- Create a build script and save it in your development tree.
- Use the <solution> task to compile your project.

In the next part we'll be adding some more features to our automated build.

Part 5 - Extending the Automated Build

In the last part we started using NAnt to automate a build for our project. In this part we'll add some more build functionality.

When we added the compile target we used the `<solution>` task to compile our solution. However, we also specified which 'Build Configuration' to use. Build Configurations are a Visual Studio feature that allow you to build your project in different ways. The most common differences are between 'Debug' and 'Release' (2 configurations that Visual Studio always creates for you.) With a Debug build, the Visual Studio compiler is configured to create the .pdb files we use for debugging (it gives us line numbers in exception stack traces, that kind of thing.) The 'Release' configuration doesn't have these files generated, but it does produce assemblies more geared towards production than development.

However, there are a whole bunch of other things you can configure for different build configurations. Right-click on a project in Visual Studio, select Properties, then look at everything that appears under 'Configuration Properties' - all of those items can change for different Build Configurations. We're interested in the 'Output Path' property, and I'll explain why.

When we tell NAnt to compile the Debug Build Configuration of our solution, it tries to invoke the C# compiler to produce all the files that appear under the bin\Debug folder for each VS Project. There's a problem with this though - if we already have the Solution open in Visual Studio, VS will have locks on those files once they reach a certain size. That means that our NAnt compile will fail since it can't overwrite the assemblies. Anyway, it would be cleaner if we could separate out our 'automated' build from our 'interactive' build.

Thankfully, Build Configurations let us do this and still use the `<solution>` task. We do this by creating a **new** Build Configuration which we will just use for automated builds, and change where it outputs its files to.

To do this for Sycamore, I open up Visual Studio's 'Configuration Manager' (right click on the Solution, choose 'Configuration Manager'), and create a new configuration (open the drop-down menu, select '<New...>'). I'm going to call the new configuration **AutomatedDebug** and copy settings from the 'Debug' configuration (leave the 'create new project configuration(s)' box checked.) Close the dialog, and then bring up the properties for 'SycamoreConsole'. Select the 'Build' 'Configuration Properties' section, and make sure 'AutomatedDebug' is selected in the Configuration drop-down. Select the 'Output Path' box and change its value to `..\..\build\Debug\SycamoreConsole`. Switch Visual Studio back to the normal 'Debug' configuration which we use for interactive builds.

Finally, edit the build script, and change the 'configuration' argument of the `<solution>` task to be AutomatedDebug. It should now look like this:

```
<target name="compile">
    <solution solutionfile="src\Sycamore.sln"
configuration="AutomatedDebug" />
</target>
```

So what have we actually done here? If you run NAnt, you should see the following lines in your build output:

```
compile:
```

```
[solution] Starting solution build.
```

```
[solution] Building 'SycamoreConsole' [AutomatedDebug] ...
```

This tells us that NAnt is using the new Build Configuration. Now, look in the build\Debug\SycamoreConsole folder - you should see our compiled .exe file (and a .pdb file since we are compiling with debug options.)

That tells us *what* is happening, but *why* have we put these files in this directory? We use the build folder as another of our 'top level' project folders. It will contain all the build artifacts (assemblies, test reports, etc.) that we produce in the automated build. It will not contain any files that aren't generated by the build, so we don't need to check it into Source Control, and we can safely delete it whenever we want. Under build we will have a number of sub-folders, and so far we created one called Debug that will contain all of our Debug compilation artifacts. We put the artifacts for each VS Project in its own folder, with the same name as the VS Project it belongs to.

I said we could safely delete this folder, so let's add another NAnt target that will do this:

```
<target name="clean">
    <delete dir="build" if="{directory::exists('build')}" />
</target>
```

I also said we didn't need to check the build folder into Source Control, so we can also add it to our list of excluded files. With Subversion, I do this by editing the svn:ignore property of the project root folder.

Finally for this part, we're going to create a batch file that developers can use to kick off the build. Its very simple, having just the following line:

```
@tools\nant\NAnt.exe -buildfile:Sycamore.build %*
```

I like calling this file 'go.bat' since the targets in the build script tend to be

'action' type words. Since its closely associated with the build script, put it in the project root. Note that we specify which build script to use - change this for your project. To use this file, just pass the target to run as an option, so to delete the build folder, just enter go clean.

Note that this batch file really is just meant as a bootstrap for convenience. I've seen plenty of projects use a combination of batch files and NAnt / Ant scripts to configure a build system. This is a **bad** idea for several reasons:

- Batch files are significantly less manageable or powerful than NAnt, and tend to get very 'hacky' very quickly.
- Your build behaviour really is one distinct concept and NAnt can handle all of it - splitting it across technologies isn't necessary.
- **Don't** go down the road of having multiple batch files to launch builds for different environments. I'm yet to see a project that managed to pull this off in a clean, manageable way. Apart from anything else it is redundancy, and introduces more manual work and possibilities for error. Instead, use logic in your NAnt script to use different property values for different environments (hopefully I'll get on to build configuration and refactoring concepts in the future.)

If you run your default target, it should still be successful. If you have all your ignored files and directories setup correctly you should have 4 files to commit - the build script, the build script launcher (go.bat), the solution, and the VS Project for Sycamore Console. I'm going to check in these changes and call it a day for this part.

The current state of Sycamore is available [here](#).

To summarise this part:

- Use a top-level, transient, folder called build as the target folder of your automated build.
- Create a new Visual Studio Build Configuration for your automated NAnt Builds. This Build Configuration should output to build.
- Setup a clean target to delete your transient files.
- Create a simple build bootstrap batch file.
- Don't put any kind of build logic in this build bootstrap - leave that all in the NAnt build script.

In the next part we'll start to add some unit tests.

Part 6 - Adding Unit Tests

By now we have some source code checked in to our Source Control server. Its got a structured folder hierarchy and we're being careful about how we check specific files in (and ignore others). We're combining Visual Studio and NAnt to have a simple yet powerful automated build that works closely with the changes we make during interactive development.

So far though we only have 1 source file and shockingly no tests. We need to change this.

To do this we're going to create 2 new assemblies - one application DLL, and one DLL for unit tests. .NET won't allow you to use .exe assemblies as references for other projects, so a unit test DLL can only reference another DLL. Its slightly off-topic but because of this reason I try to keep my .exe projects as small as possible (because any classes in them can't be unit tested) and have nearly all code in a DLL.

So let's create our new Application DLL. I'm going to call it *Core*. Following the conventions we set down in part 2, the VS Project Folder is stored in src and we change the default namespace to SherwoodForest.Sycamore.Core. Before closing the Project Properties window though, there are 2 more things to change.

Firstly, for DLLs I like to use the naming convention that the Assembly has the same name as the default namespace. Also, following what we did in the previous part, create an 'AutomatedDebug' configuration, based on the 'Debug' Configuration, except with the output path of ..\..\build\Debug\Core. Make sure your **Solution** build configurations are all mapped correctly. We won't need the 'Class1' which VS automatically creates, so delete it.

We follow exactly the same procedure for our Unit Test DLL, giving the VS Project the (not particularly original, nevertheless informative) name of *UnitTests*. Save everything and make sure you can compile in Visual Studio and using your build script.

Before we write a test, we need to setup our project with [NUnit](#). There's a few hoops to go through here but we only have to do it once for our project. Firstly, download NUnit - I'm going to be using NUnit 2.2.2 for this example. Download the binary zip file, not the MSI. While its downloading, open up your Global Assembly Cache (or GAC) - it will be in C:\Windows\Assembly, or something similar. Look to see if you have any NUnit assmblies in it. If you do, try to get rid of them by uninstalling any previous versions of NUnit from your computer.

Why are we worrying about not using the GAC and MSI's? Well, for pretty much exactly the reasons as we specified for NAnt, we want to use NUnit from our development tree . The problem is that if we have any NUnit

assemblies in the GAC, they will take priority over the NUnit in our development tree. We could go through being explicit about the versions of NUnit each assembly requires, but that's a lot of hassle. It's easier just not to make NUnit a system wide tool, and this means getting it out of the GAC. *(Mike Two, one of the NUnit authors, is probably going to shoot me for suggesting all of this. If you want to make NUnit a system tool then that will work too, you just have a few more hoops to jump through.)*

By now your NUnit download should be complete. Extract it, take the bin folder and put it next to the nant folder in your project's tools folder. Rename it to nunit.

To create test fixtures in our *UnitTests* VS Project, we need to reference the nunit.framework assembly. This introduces a new concept - that of third party code dependencies. To implement these, I like to have a new top-level folder in my project root called lib. Do this in your project and copy the nunit.framework.dll file from the NUnit distribution to the new folder. Once you've done that, add lib\nunit.framework.dll as a Reference to your *UnitTests* project.

Because of the previous step we now have the same file (nunit.framework.dll) copied twice in our development tree. It's worth doing this because we have a clear separation between **code dependencies** (in the lib folder) and **build time tools** (in the tools folder). We could delete the entire tools folder and the solution would still compile in Visual Studio. This is an example of making things clean and simple. It uses more disk space, but remember what we said back in Part 1 about that?

So finally we can actually write a test! For Sycamore, I'm going to add the following as a file called TreeRecogniserTest.cs to my *UnitTests* project:

```
using NUnit.Framework;
using SherwoodForest.Sycamore.Core;

namespace SherwoodForest.Sycamore.UnitTests
{
    [TestFixture]
    public class TreeRecogniserTest
    {
        [Test]
        public void ShouldRecogniseLarchAs1()
        {
            TreeRecogniser recogniser = new TreeRecogniser();
            Assert.AreEqual(1, recogniser.Recognise("Larch"));
        }
    }
}
```

To implement this, I add *Core* as a Project Reference to *UnitTests* and create a new class in *Core* called *TreeRecogniser*:

```

namespace SherwoodForest.Sycamore.Core
{
    public class TreeRecogniser
    {
        public int Recognise(string treeName)
        {
            if (treeName == "Larch")
            {
                return 1;
            }
            else
            {
                return 0;
            }
        }
    }
}

```

I can then run this test by using [TestDriven.NET](#) within the IDE, or by using the NUnit GUI and pointing it at `src\UnitTests\bin\Debug\SherwoodForest.Sycamore.UnitTests.dll`. The tests should pass in either case.

If we run our automated NAnt build, everything should compile OK, and you should be able to see each of the VS Projects compiling in their AutomatedDebug Build Configuration. The tests aren't run yet, but that's what we'll be looking at next time. Even so, we are still at a check-in point. We have 2 new project folders to add, but remember the exclusion rules (*.user, bin and obj). Being a Subversion command-line user, I like to use the the -N (non recursive) flag of `svn add` to make sure I can mark the `svn:ignore` property before all the temporary files get added.

Also, don't forget to check in `tools\nunit` or the new `lib` folder.

The current state of Sycamore is available [here](#).

So let's wrap up this part then. We covered some new generic principles about projects and dependencies. We also looked at the specifics of using NUnit. Some concrete points to take away are:

- Set DLL Names to be the same as the default namespace
- Put your Unit Tests in a separate VS project called *UnitTests*
- Save NUnit in your development tree in its own folder under `tools`
- Put all DLLs your code depends on in a top level folder called `lib`. The only exceptions are system DLLs such as .NET Framework Libraries.

Part 7 - Automating Unit Tests

Last time we left our code with a dependency on a 3rd party library, multiple internal modules (VS Projects), and a passing test. Great! But how do we know the test passes? At the moment it requires us to have our 'interactive hat' on. It would be much better if we knew just by running our automated build. So let's do that.

Before we start, here is the current state of our build script:

```
<project name="nant" default="compile"
xmlns="http://nant.sf.net/schemas/nant.xsd">
  <target name="clean">
    <delete dir="build" if="{directory::exists('build')}" />
  </target>

  <target name="compile">
    <solution solutionfile="src\Sycamore.sln"
configuration="AutomatedDebug" />
  </target>
</project>
```

We're going to add a *test* target. Here's our first cut:

```
<target name="test">
  <exec program="nunit-console.exe" basedir="tools\nunit"
workingdir="build\Debug\UnitTests">
    <arg value="SherwoodForest.Sycamore.UnitTests.dll" />
  </exec>
</target>
```

Here we are using an `<exec>` task to run the NUnit Console application that's already in our development tree (that was handy, wasn't it? That's why we left all the NUnit binaries in our tree.) Some projects will use the `<nunit>` or `<nunit2>` tasks to run their tests from a build script, but this requires your version of NAnt and version of NUnit being in sync. Personally, I think the `<exec>` call looks pretty clean so I'm happy to use that rather than the tighter NUnit integration. And it means that later on if we update one of these 2 tools we don't have to worry about breaking this part of our build script.

The slightly tricky thing here is getting our directory specifications right. `<exec>`'s `basedir` attribute is the location of the actual .exe we want to run, and `workingdir` is the directory we want to run the application in. What might catch you out is that `workingdir` is relative to your NAnt base directory, not to the `basedir` attribute in the task specification.

Try running this target by entering `go test` from a command prompt in the project root. Did it work? What if you try `go clean test`? The problem is that we need to compile our code before we test our code. NAnt supports this

kind of problem through the depends target attribute and <call> task. Now we are entering the realm of much disagreement between build script developers. :) Which is the best option? And how should it be used? If you're new to NAnt, you'll probably want to skip the next few paragraphs.

depends specifies that for a target to run, all the targets in the depends list must have all run already. If they haven't, they will be run first, and then the requested target will run. <call> is much more like a traditional procedure call. So surely <call> is the best option, since we all know about procedure calls, right? Well, maybe, but the problem is that depends is really quite a clean way of writing things, especially when our script has multiple entry points. Also, traditionally, the behaviour of 'properties' have been a little strange when using <call>. depends though can get messy if every target has 7 different dependencies.

So, for better or worse, here's my current advice on this subject:

1. Use depends as the primary way of defining flow in your build script.
2. If a target has a depends value, **don't** give it a body. In other words a target should have task definitions, or dependencies, but **not** both. This is to try and get away from the 'dependency explosion' that Ant / NAnt scripts tend towards.
3. Use <call> only for the equivalent of an [extract method](#) refactoring. <call>ed targets should never have dependencies. Think very carefully about properties when using <call>.

We'll put this hot potato back on the fire now.

(Paragraph skippers, join back in here.) So back to our *test* target. What we want to say is that running the unit tests *depends* on compiling the code. So we'll add the attribute depends="compile" to the *test* target tag.

```
<target name="test" depends="compile" />
  <exec program="nunit-console.exe" basedir="tools\nunit"
workingdir="build\Debug\UnitTests">
    <arg value="SherwoodForest.Sycamore.UnitTests.dll" />
  </exec>
</target>
```

Now we're mixing up our dependencies and tasks though, breaking rule 2 above. We'll use an **extract dependency target** refactoring to split the target into 2 (note the second dependency on the *test* target):

```
<target name="test" depends="compile, run-unit-tests"
description="Compile and Run Tests" />

<target name="run-unit-tests">
```

```

        <exec program="nunit-console.exe" basedir="tools\nunit"
workingdir="build\Debug\UnitTests">
            <arg value="SherwoodForest.Sycamore.UnitTests.dll" />
        </exec>
</target>

```

There's something else we've done here - we've added a description to the *test* target. This is important - you should use the convention that targets with a description value are runnable by the user. If a user tries running a target **without** a description then that's down to them - they should be aware that the script may fail since dependencies have not been run. Users can easily see all the 'public' targets in a build script by doing `go - projecthelp` (the 'main' targets as NAnt calls them are our public targets.)

OK, we can run our tests, but where are the results? What we'd actually like is to use NUnit's XML output so that results can be picked up by another process, such as [CruiseControl.NET](#). Let's put this XML output somewhere in the build folder, since it's another one of our build artifacts. We'll update the *run-unit-tests* target as follows:

```

<target name="run-unit-tests">
    <mkdir dir="build\test-reports" />
    <exec program="nunit-console.exe" basedir="tools\nunit"
workingdir="build\Debug\UnitTests">
        <arg value="SherwoodForest.Sycamore.UnitTests.dll" />
        <arg value="/xml:..\..\test-reports\UnitTests.xml" />
    </exec>
</target>

```

We used the `/xml:` parameter for NUnit, and made sure the report output directory already existed.

One more thing, and then we'll be done. We already introduced the idea of a **build script refactoring** above when we split-up the *test* target. If you look at the current state of the build script though, you'll see there's plenty of scope for another refactoring - 'introduce variable', or **introduce script property** as we'll call it in the build script world. Look at all those places where we use the folder name `build`. Let's put that in a script property called `build.dir`. Now our script looks like:

```

<project name="nant" default="test"
xmlns="http://nant.sf.net/schemas/nant.xsd">
    <property name="build.dir" value="build" />

    <!-- User targets -->
    <target name="test" depends="compile, run-unit-tests"
        description="Compile and Run Tests" />

    <target name="clean" description="Delete Automated Build
artifacts">
        <delete dir="{build.dir}"
if="{directory::exists(property::get-value('build.dir'))}"/>

```

```

    </target>

    <target name="compile" description="Compiles using the
AutomatedDebug Configuration">
        <solution solutionfile="src\Sycamore.sln"
configuration="AutomatedDebug" />
    </target>

    <!-- Internal targets -->
    <target name="run-unit-tests">
        <mkdir dir="${build.dir}\test-reports" />
        <exec program="nunit-console.exe" basedir="tools\nunit"
workingdir="${build.dir}\Debug\UnitTests">
            <arg value="SherwoodForest.Sycamore.UnitTests.dll"
/>
            <arg value="/xml:..\..\test-reports\UnitTests.xml"
/>
        </exec>
    </target>
</project>

```

A lot of people will introduce a script level property whenever they introduce a new directory, file, etc. I advise you not to do this in your build script development since (I think) it hinders maintainability. Treat your build script like well maintained code - do the simplest thing that works, but refactor mercilessly. In terms of **introduce script property** you should really **only** be doing it once the same piece of information is used by multiple targets. For example, a lot of people would introduce a `src.dir` property out of principle, and in our case it would have the value `src`. But what would that gain us? In our build script we only ever use that directory name once, so its simpler just to leave it as a literal usage in the call to `<solution>`.

Notice in the last example we also added descriptions to all the targets we want to be public, and split the file up into (effectively) public and private targets. XML is not the cleanest language to develop in, but by thinking about simplicity and readability, you can make your build scripts more maintainable.

To summarise this part:

- Use the `<exec>` task to call NUnit within your build script.
- Use targets that just specify dependencies to create flow within your build script.
- Don't use dependencies with targets that specify tasks
- Split your targets into 'public' and 'private' targets by giving public targets a description.
- Use **build script refactorings** to simplify the structure of your NAnt file.
- Don't introduce unnecessary script properties