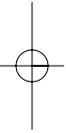
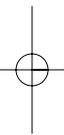
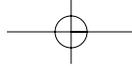


Part III

Integrating .NET Open Source Projects in Your Development





CHAPTER 9

ASpell.NET Case Study

Nothing you can't spell will ever work.

—Will Rogers

Introduction

There are three excuses ever present in software development:

1. We don't have time to automate the build.
2. We'll do the documentation later.
3. We don't have time to write tests.

I understand the importance of Time to Market and that the first release of a product is always the hardest to get out the door. However, I believe this “rush-to-market” approach to development is shortsighted and outdated. This philosophy could wind up costing you a lot more time in the end than if you spend a little time up front creating a solid development procedural atmosphere.

Chapters 4 through 7 deal directly with how to solve these problems. Writing documentation along the way is not difficult with NDoc. NAnt allows for an intuitive and solid scriptable build. Testing can actually be enjoyable with NUnit while you try to break your code with the test code you write.

This chapter is a simple but realistic case study of using Open Source tools in everyday development. To illustrate this I chose to create a new Open Source project: ASpell.NET. ASpell.NET (<http://aspell-net.sourceforge.net>) is a .NET wrapper around the GNU ASpell project (<http://aspell.sourceforge.net>). ASpell is a commercial-caliber spell checking component that supports twenty-some different language dictionaries.

NOTE: ASpell.NET is not a good candidate for cross-platform support because PInvoke is not very well supported on CLIs other than Microsoft's .NET.

Thorsten Maerz has created a Win32 port of ASpell, which I will wrap in a C# component. I believe this to be a good example because it includes PInvoke calls into unmanaged code. Realistically, this process of wrapping existing functionality and making it available to managed code will probably be done for quite a while by most corporations.

TIP: For a good book on .NET Interop, see:

.NET and COM: The Complete Interoperability Guide
by Adam Nathan from Sams Publishing

Adam has also created a wiki (<http://www.pinvoke.net>) and a Visual Studio.net add in (click the get visual studio add-in link on the Web site).

This example will check the code by using FxCop and NUnit. As we go along, we will utilize NDoc to output a useful help file for redistribution. In the end, an install will be created using MSI, and an iso file will be created using the custom mkisofs task created in Chapter 4. The iso file will then be uploaded to a server for distribution to a testing department or, depending on the extent of your testing, to customers. All this will happen automatically upon checking in code using Continuous Integration if the build succeeds (which also implies that the tests succeed).

I created ASpell.NET as a proof-of-concept to see how easy it would be to get ASpell working for .NET. ASpell.NET would make a great Web service. To eliminate the need to use pointers and the unsafe compiler option, I wrapped the ASpell dll in another C++ dll (AspellDll.dll). This allows ASpell.NET to use methods that have parameters that require pointers. So the base functionality for ASpell.NET is already there, but with no documentation or tests and without support for dictionaries other than the English dictionary. The source is available from the SourceForge project site, and you will be able to see that Log4Net also plays a part in ASpell.NET. We will use .NET's CulturalInfo to automatically detect which language dictionary to use. Finally, we will demonstrate the use in a somewhat real-world application similar to the WordPad app using #develop to create it as Windows form.

Test First Development

Really, to be fair, I should have developed the tests for ASpell.NET first, but I wanted to see if it was even feasible. Since it seems to be a useful component, now is the time to write the tests while the project is still pretty young. What is nice about developing the tests first is that it forces you to create your API and think about your contract with the outside world in a detailed way. After all, you cannot write the tests for components if you do not know what the signatures of the public methods are or what properties are available. For ASpell.NET, I did not have to spend too much time in the design of the component because it pretty much mimics ASpell's design. This was done for two reasons. First, I wanted ASpell.NET to be familiar to anyone who has used ASpell. Second, I realized the amount of man-hours and development put into ASpell and decided that I probably could not come up with a much better design.

I will start with the following tests:

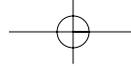
- What if I pass in a null or empty string to spell check?
- What if I use numerals or punctuation?
- What if the program can't find a dependant dll or dictionary?
- Can it support multiple cultures?
- What if there's nothing to return?

To perform these tests, I will add a reference to NUnit into the build and use it to create the tests. Listing 9.1 shows the simplest test if a null is passed into ASpell.NET.

Listing 9.1 NUnit Test

```
[TestFixture]
public class AspellNullTest
{
    private aspell.net.SpellChecker spellCheck;

    [SetUp]
    public void Init()
    {
        spellCheck = new aspell.net.SpellChecker();
    }
}
```



246 Chapter 9 ASpell.NET Case Study

```
[Test]
[ExpectedException(typeof(aspell.net.ASpellNullException))]
public void NullCheck()
{
    foreach(CultureInfo ci in spellCheck.CultureInfos)
    {
        spellCheck.CultureInfo = ci;
        spellCheck.checkword(null);
    }
}

[Test]
[ExpectedException(typeof(aspell.net.ASpellNullException))]
public void NullSuggest()
{
    foreach(CultureInfo ci in spellCheck.CultureInfos)
    {
        spellCheck.CultureInfo = ci;
        spellCheck.suggest(null);
    }
}
}
```

NOTE: NUnit version 2.1.4 was used for these tests.

This test will take no time at all. The hardest test will be removing the dictionaries and dependant dlls. In designing the tests, we must create them in a way that will not require you to rewrite them for every culture.

NAnt Build

I will use the NAnt master build, simple dll, simple exe, and winform exe templates all taken from Chapter 4. Listing 9.2 shows the master build file.

NOTE: NAnt 0.8.2 was used for this chapter.

Listing 9.2 ASpell.NET's Build Projects File

```
<projects>
  <!-- ASpell.Net -->
  <project>
    <name>aspell.net\aspell.net</name>
  </project>
  <!-- Tests -->
  <project>
    <name>tests\nunit\nunit</name>
    <dependencies>
      <dependency>aspell.net</dependency>
    </dependencies>
  </project>
  <project>
    <name>tests\ConsoleAppTester\ConsoleAppTester</name>
    <dependencies>
      <dependency>aspell.net</dependency>
    </dependencies>
  </project>
  <!-- Examples -->
  <project>
    <name>examples\WindowsApp\WindowsApp</name>
    <dependencies>
      <dependency>aspell.net</dependency>
    </dependencies>
  </project>
  <project>
    <name>examples\WordPadSpell\WordPadSpell</name>
    <dependencies>
      <dependency>aspell.net</dependency>
    </dependencies>
  </project>
</projects>
```

Notice that all the example projects are dependent upon the ASpell.NET project. Listing 9.3 shows that after the transform of the project file and building all the subprojects, the setup project is run, and the iso file is created. At this point, the file could be distributed a number of different ways, but Listing 9.3 uses SCP (secure copy) to upload the file to a server.

248 Chapter 9 ASpell.NET Case Study

Listing 9.3 ASpell.NET's Master Build File

```

<project name="Master Build" basedir="." default="build">
  <sysinfo verbose='true' />

  <!-- General Build Properties -->
  <property name="debug" value="true" />
  <property name="define" value="DEBUG;TRACE" />
  <property name="build.dir" value="C:\book" />
  <property name="refassemblies" value="${build.dir}\refassemblies "
/>
  <property name="isofile" value="release.iso" />

  <!-- MSI Properties -->
  <property name="product.name" value="ASpell.NET" />
  <property name="company.name" value="OpenSource.NET" />
  <property name="msi.version" value="1.0.2" />
  <property name="msi.guid.product" value="{D9C16B65-BD89-44f5-AEC8-
16775D4A3619}" />
  <property name="msi.guid.upgrade" value="{42D979E5-E2E8-45c6-89D4-
378353848479}" />
  <!-- Location to output the complete msi -->
  <property name="output.dir" value="${build.dir}\output" />

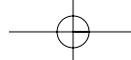
  <target name='build'>

    <exec program=iNantHelper.exei commandline=iprojects.xmlî
output=iprojects.txtî basedir=i.i />

    <!-- After applying Helper application transform pass the target
to the subprojects -->
    <foreach item='Line' property='filename' in='projects.txt'>
      <nant buildfile='${build.dir}\${filename}.build' target='build'
/>
    </foreach>

    <msi
      sourcedir="${output.dir}"
      license="license.rtf"
      output="${company.name}.${product.name}.msi"
      debug="true"
      verbose="true"
    >

```



```
<properties>
  <property name="ProductName" value="${product.name}" />
  <property name="ProductVersion" value="${msi.version}"
/>
  <property name="Manufacturer" value="${company.name}" />
  <property name="ProductCode" value="${msi.guid.product}"
/>
  <property name="UpgradeCode" value="${msi.guid.upgrade}"
/>
</properties>

<directories>
  <directory name="D__BINDIRECTORY" foldername="bin"
root="TARGETDIR" />
</directories>
<components>
  <component name="C__MainFiles" id="{301CC44C-A3A4-4674-
AE04-23D91F156301}" attr="2"
    directory="TARGETDIR" feature="F__DefaultFeature">
    <key file="Test.xml" />
    <fileset basedir="${build.dir}">
      <includes name="*. *" />
    </fileset>
  </component>
</components>
<features>
  <feature name="F__DefaultFeature" title="${product.name}
Main Feature" display="1" typical="true" directory="TARGETDIR">
    <description>${product.name} core
files.</description>
  </feature>
</features>
</msi>

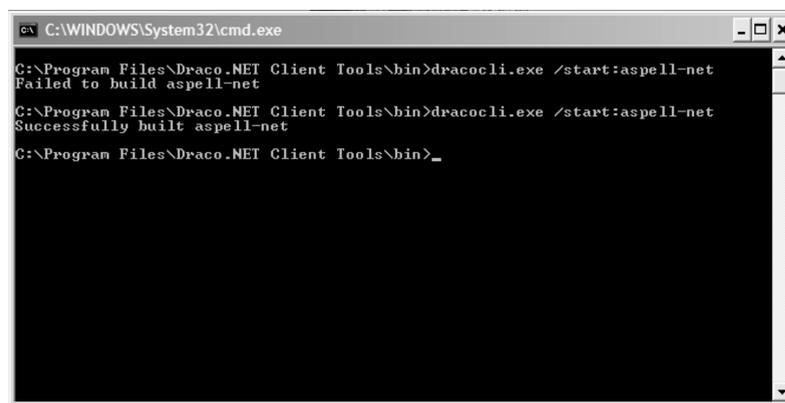
  <mkisofs isofilename='${build.dir}\${isofile}'
inputdir='output.dir' />

  <scp file='${build.dir}\${isofile}' server="ReleaseServer"
path="~" />

</target>
</project>
```

Figure 9-1 is the assurance that the NAnt build, complete with NUnit and NDoc integration, is working.

Before continuing any further, we should set up a Source Code Management (SCM) system.



```
C:\WINDOWS\System32\cmd.exe
C:\Program Files\Draco.NET Client Tools\bin>dracocli.exe /start:aspell-net
Failed to build aspell-net
C:\Program Files\Draco.NET Client Tools\bin>dracocli.exe /start:aspell-net
Successfully built aspell-net
C:\Program Files\Draco.NET Client Tools\bin>_
```

Figure 9-1 ASpell.NET's Build Output.

Subversion

I chose to use Subversion (SVN) with ASpell.NET, even though SourceForge does not support it, because it is easy to use and has some nice features that CVS does not support.

NOTE: TortoiseSVN version 0.21.0 Build 277 and SVN version 0.32.1 were used in this chapter.

First, a repository must be created, which is simple, as Figure 9-2 demonstrates.

Next I will simply import ASpell.NET into the repository. Figure 9-3 shows how to import a Subversion project.

As you can see, using Subversion is very simple and not intrusive in the development cycle. Another great feature of Subversion is that the repository can be easily compressed and moved to a different machine, even if that machine runs a different operating system.

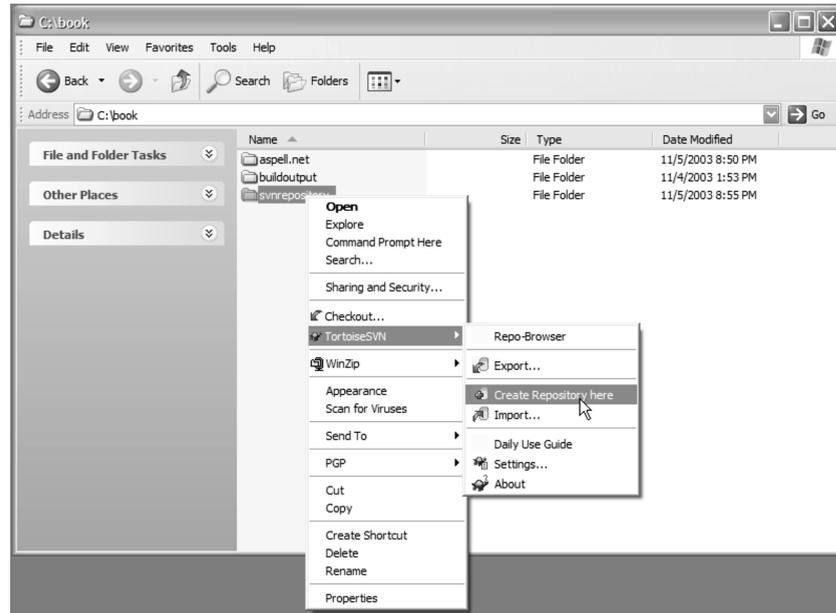


Figure 9-2 Creating an SVN repository with TortoiseSVN.

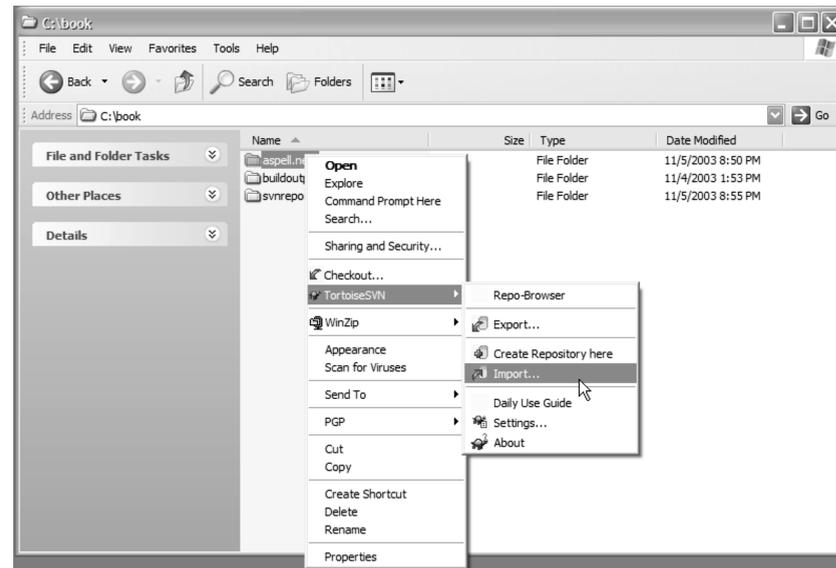


Figure 9-3 Importing ASpell.NET using TortoiseSVN.

Draco.NET

I choose Draco.NET because it is so easy to install and simple to use. Although CruiseControl.NET does offer nice features for a large development team, I wanted to eliminate any complexity to emphasize the concepts of a complete case study. Listing 9.4 is the configuration for Draco.NET to build ASpell.NET. Figure 9-4 shows the email notification sent from the initial import's triggering of the build.

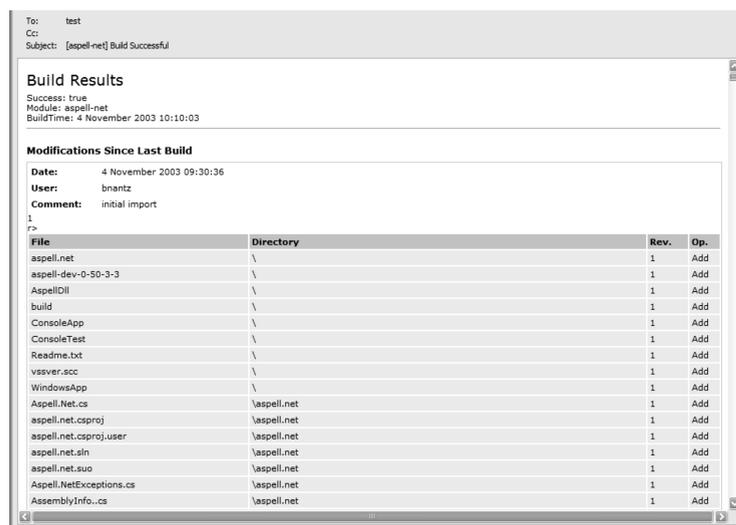


Figure 9-4 Draco.NET's Email Notification.

Listing 9.4 Draco.NET's Configuration

```
<?xml version="1.0" encoding="utf-8"?>
<configuration>
  <configSections>
    <section name="draco"
type="Chive.Draco.Config.ConfigurationSection, Draco"/>
  </configSections>
  <system.diagnostics>
    <switches>
      <add name="TraceLevelSwitch" value="4"/>
    </switches>
  </system.diagnostics>
</configuration>
```

```

    <trace autoflush="true" indentsize="4">
      <listeners>
        <remove name="Default"/>
      </listeners>
    </trace>
  </system.diagnostics>
  <system.runtime.remoting>
    <application>
      <service>
        <wellknown mode="Singleton"
objectUri="Draco" type="Chive.Draco.DracoRemote, Draco"/>
      </service>
      <channels>
        <channel ref="tcp" port="8086"/>
      </channels>
    </application>
  </system.runtime.remoting>
  <draco xmlns="http://www.chive.com/draco">
    <pollperiod>60</pollperiod>
    <quietperiod>60</quietperiod>
    <mailserver>mail.securityint.com</mailserver>
    <fromaddress>brian.nantz@somewhere.com</fromaddress>
    <builds>
      <build>
        <name>aspell-net</name>
        <pollperiod>10</pollperiod>
        <quietperiod>30</quietperiod>
        <notification>
          <email>
            <recipient>brian.nantz@somewhere.com</recipient>
            </email>
            <file>
              <dir>C:\book\BuildOutput</dir>
            </file>
          </notification>
        <nant>
          <buildfile>build\master.build</buildfile>
        </nant>
        <svn>
          <url>file:///C:/book/svnrepository</url>
        </svn>
        <ignorechanges>

```

```
                <ignore comment="autobuild"/>
            </ignorechanges>
        </build>
    </builds>
</draco>
</configuration>
```

Again, in Listing 9.4, all the comments have been removed. These comments more than point you in the right direction with helpful examples. For example, there is a section for each source control that it supports (i.e., Subversion, Visual Source Safe, etc.), and you just have to uncomment it and changed the values (like username, paths, passwords) to fit your environment. By just uncommenting the proper notification XML node, you can now receive build results in multiple formats. All of the supported SCMs are also very configurable and well documented. Notice that you can ignore certain checkins to SCM (if you are triggering builds off of checkins and are not scheduled). You can also potentially monitor multiple source repositories. In Listing 9.4, only the svn XML tag is used to monitor a single Subversion repository.

Next we will add a new example client for greater stressing of the component.

Adding Functionality

In the original proof-of-concept, I created two example applications. The first example was a C# Console Application to take a word input and if needed return the suggestions for the correct spelling (Figure 9-5).

The second application was a simple VB.NET application (Figure 9-6) that showed checking a single word in a text box and returning the results in a right-click context menu.

While these applications tested the functionality and demonstrated the cross-language capability, they really were not real-world useable applications. There are a few things that ASpell.NET needs to support before even a quality beta cycle. First, I wanted to add Log4Net to the project, even though it is somewhat simple to support good behavior if the system starts to error. This is one example where it may be useful to associate the Logger configuration with a dll. This probably warrants some changes to the code. In the meantime, ASpell.NET should just document the Log4Net requirement in the calling executables configuration file. Second, if I want to check

```

C:\WINDOWS\System32\cmd.exe
C:\dev\books\opencsharpdev\part II\chapter 9\code\aspell.net\tests\ConsoleAppTester\bin\Debug>ConsoleTest.exe oahago
lago
ago
hag
oho
imago
hrgo
Haag
Hugo
Oahu
Ohio
OIag
haagie
quahog
Haggai
agog
hgi
Oage
outgo
hog
aho y
oho s
hnaigo
Ag
Hg
hgy
hagar
Hager
Hague
anigo
C:\dev\books\opencsharpdev\part II\chapter 9\code\aspell.net\tests\ConsoleAppTester\bin\Debug>_

```

Figure 9-5 Console Test Application.

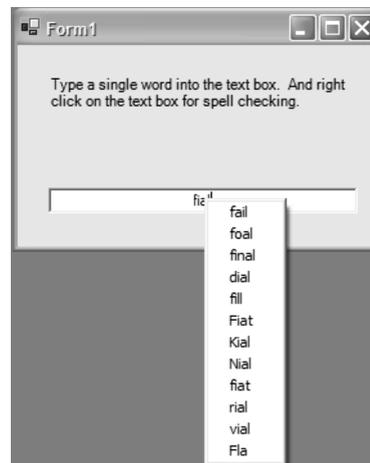


Figure 9-6 VB.NET Windows Form Test Application.

spelling, it usually is not a single textbox field but a whole document like email or a text file, so I decided to create an application much like WordPad that is a Rich Text Format-aware editor application. Figure 9-7 shows using SharpDevelop, which is the only Open Source editor I am aware of that has a Windows Forms designer.

256 Chapter 9 ASpell.NET Case Study

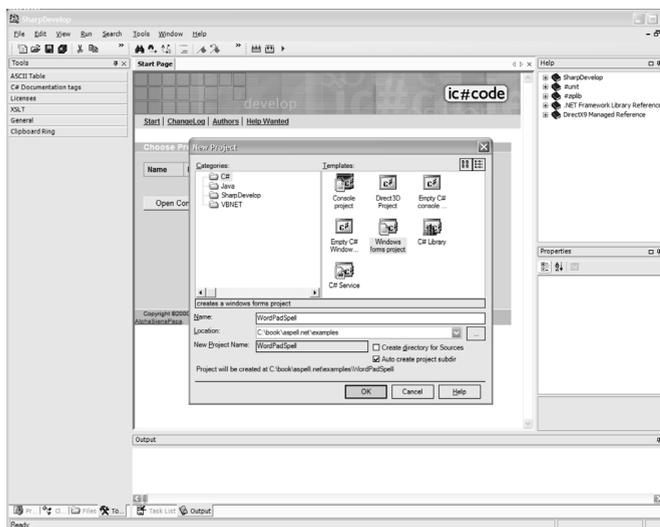


Figure 9-7 SharpDevelop WordpadSpell Project.

While eventually I would like all the functionality of WordPad, this example application only allows for spell checking and opening a text file for spell checking. Figure 9-8 shows WordpadSpell in action.

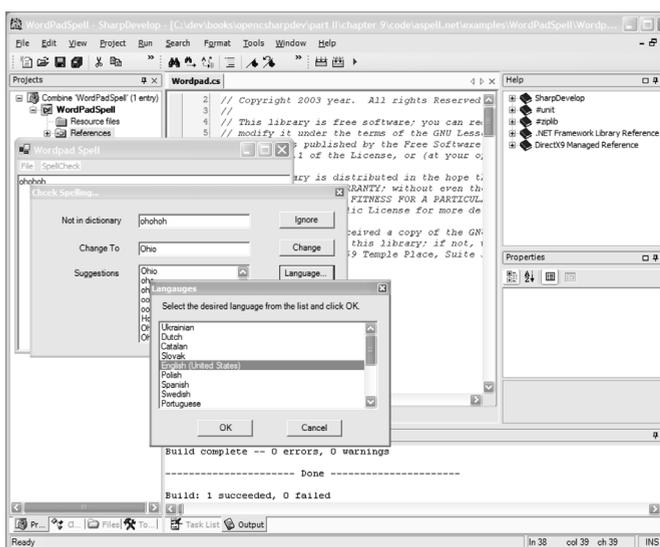


Figure 9-8 WordpadSpell Example.

Listing 9.5, a sample command-line application from the ASpell.NET distribution, shows that the API for ASpell.NET is fairly simple. The WordpadSpell is responsible for the logic to parse things down to a single word to pass to ASpell.NET for checking.

Listing 9.5 A Simple Use of ASpell.NET

```
try
{
    string word = args[0].ToString();
    aspell.net.SpellChecker spellCheck = new
aspell.net.SpellChecker();
    if(!spellCheck.checkword(word))
    {
        System.Collections.ArrayList suggestions =
spellCheck.suggest(word);
        foreach(string suggestion in suggestions)
        {
            Console.WriteLine(suggestion);
        }
    }
    else
    {
        Console.WriteLine(word + " is correct");
    }
}
catch(Exception ex)
{
    System.Diagnostics.Debug.WriteLine(ex.ToString());
    Console.Write(ex.ToString());
}
```

Adding a NAnt build script is a simple copy and paste and a quick edit of the template file. When we add the project to Subversion (Figure 9-9), Draco.NET will be triggered to perform a build. The installation and iso file are created, and the successful result is emailed out.

258 Chapter 9 ASpell.NET Case Study

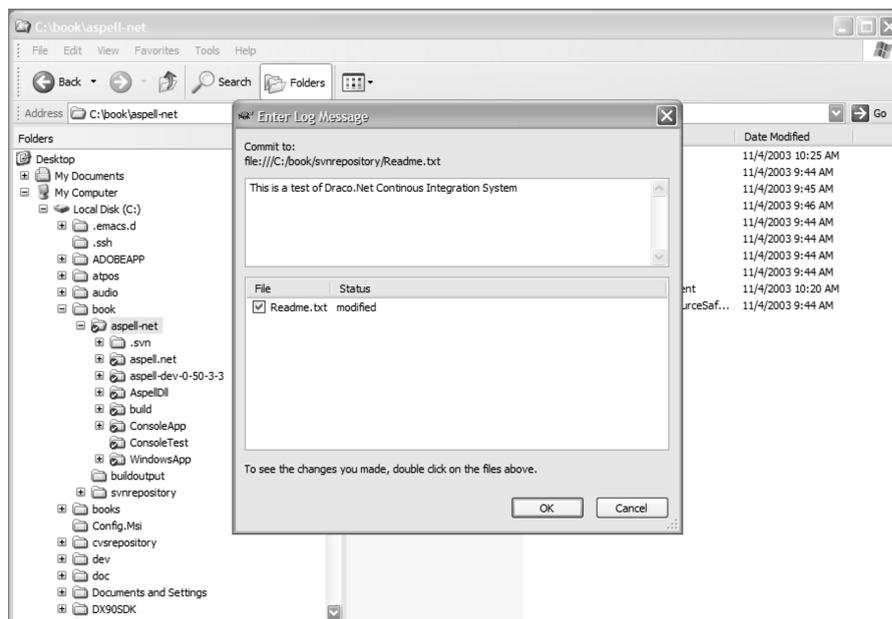


Figure 9-9 Adding to SVN.

Summary

While this application is admittedly simple compared to what you develop day in and day out, it is large enough to demonstrate that the process of using these Open Source tools has great benefits. Most real projects can be broken up into smaller projects that are not much more daunting than this one. These tools do not require all that much overhead to the development process. As the project or number of developers increases, the value of these tools increases in more of an exponential rather than linear proportion. These tools could easily be (and actually already are) used in SourceForge's projects to remotely monitor CVS in a multi-developer project and to automatically build and expose integration issues.