Appendix A

New Features in MSBuild 4.0

With MSBuild 4.0, there were a large number of new features added. By far, the biggest feature that was added is support for building Visual C++ projects. The other features that were added were more foundational, just to make MSBuild 4.0 better than MSBuild 3.5. In this appendix, we will discuss those features in detail. Most, but not all of the content in this appendix has been covered previously in this book.

Support for Visual C++

As mentioned previously, the support for Visual C++ is by far the biggest new feature of MSBuild 4.0. Many changes were required to get Visual C++ projects using MSBuild. If you need to learn more about these features, then you should read the chapters in the "MSBuild in Visual C++" section that is dedicated to this area.

New Command-Line Switches

With MSBuild 4.0, there are two new command-line switches and one new parameter for the Console Logger. The two new switches are outlined in Table A-1.

TABLE A-1 New Command-Line Switches

Switch	Description
/preprocess (/pp)	You can use this switch to output the full canonical MSBuild file that the MSBuild engine uses to execute the build. The result will inline all the imports. This is very useful to see what elements are being used for the build.
/detailedSummary (/ds)	You can use this switch to include a detailed summary at the end of the build.

Along with these two switches, you can pass the DisableConsoleColor parameter to the Console Logger in order to display everything in the same color.

New Reserved Properties

As shown in Table A-2, a number of new reserved properties have been introduced with MSBuild 4.0.

TABLE A-2 New Reserved Properties

Name	Description
MSBuildExtensionsPath64	The full path to where MSBuild 64-bit extensions are located. This is typically under the Program Files folder. For 32-bit machines, this value will be empty.
MSBuildLastTaskResult	This contains a value of *true* if the last executed task was a success and *false* if it ended in a failure. If a task fails, typically the build stops unless you specified `ContinueOnError="true"`.
MSBuildProgramFiles32	The full path where MSBuild 32-bit extensions are located. This is typically under the Program Files folder.
MSBuildThisFile	Contains the file name, including the extension, to the file that contains the property usage. This differs from MSBuildProjectFile in that MSBuildProjectFile always refers to the file that was invoked, not any imported file name.
MSBuildThisFileDirectory	The path of the folder of the file which uses the property. This is useful if you need to define any items whose location you know relative to the targets file.
MSBuildThisFileDirectoryNoRoot	This is the same as MSBuildThisFileDirectory without the root (for example, InsideMSBuild\Ch02 instead of C:\InsideMSBuild\Ch02).
MSBuildThisFileExtension	The extension of the file referenced by MSBuildThisFile.
MSBuildThisFileFullPath	The full path to the file that contains the usage of the property.
MSBuildThisFileName	The name of the file, excluding the extension, to the file that contains usage of the property.
MSBuildOverrideTasksPath	MSBuild 4.0 introduces override tasks, which are tasks that force themselves to be used instead of any other defined task with the same name, and this property points to a file that contains the overrides. The override tasks feature is used internally to help MSBuild 4.0 work well with other versions of MSBuild.

For more information on reserved properties, you can see the section entitled "Reserved Properties," in Chapter 2, "MSBuild Deep Dive, Part 1."

BeforeTargets and *AfterTargets*

As we have seen throughout the book, you can now place either the *BeforeTargets* or *AfterTargets* attribute on the *Target* element. For example, consider the following basic example.

```
<Target Name="CustomBeforeBuild" BeforeTargets="Build">
    <Message Text="CustomBeforeBuild"/>
  </Target>
```

```
<Target Name="CustomAfterBuild" AfterTargets="Build">
  <Message Text="CustomAfterBuild"/>
</Target>
```

In this case, we have declared two targets, CustomBeforeBuild and CustomAfterBuild. We used the *BeforeTargets* and *AfterTargets* attributes to inject these targets before and after the Build target, respectively.

ImportGroup

With previous versions of MSBuild, there was no way to group a set of related imports. Now you can achieve this with the *ImportGroup* element. You might be interested in using the *ImportGroup* element when you have a set of related imports that you want to conditionally import. With MSBuild 4.0, you can place the condition on *ImportGroup* itself, whereas with previous versions you would have had to put the condition on all *Import* elements. For example, consider this snippet.

```
<ImportGroup Condition=" '$(ImportCustomFiles)'=='true' ">
  <Import Project="custom01.proj"/>
  <Import Project="custom02.proj"/>
  <Import Project="custom03.proj"/>
</ImportGroup>
```

Here, the three *Import* elements would be processed only if the value for the ImportCustomFiles property is set to *true*.

Import Wildcard

With previous versions of MSBuild, if you wanted to import a file, you had to specify the name and location of the file. You could not dynamically load a set of files based on a path wildcard. With MSBuild 4.0, however, you can do just this. For example, let's say that your project had an Imports folder where you wanted to drop different MSBuild files that should be imported into the build. You could easily achieve this with the following line of code.

```
<Import Project="Imports\*"/>
```

This *Import* element would import every file in the Imports folder.

Solution Import Files

With previous versions of MSBuild, it was simply not possible to extend the build process of a solution file. This is because solution files (.sln) are not in MSBuild format. Unfortunately, this is still the case with this release, but there is a little-known feature that can be used to extend the build process for solution files. When you build a solution file, it will be converted

to an in-memory MSBuild project file, and then that will be built. This in-memory file contains import declarations that automatically import files from a known location into the build process. I've outlined what the solution build file looks like here.

```
<Project ToolsVersion="4.0" xmlns="http://schemas.microsoft.com/developer/msbuild/2003"
         DefaultTargets="Build"
         InitialTargets="ValidateSolutionConfiguration;
                         ValidateToolsVersions;ValidateProjects">
  <Import Project="$(MSBuildExtensionsPath)\$(MSBuildToolsVersion)\SolutionFile\
ImportBefore\*"
          Condition="'$(ImportByWildcardBeforeSolution)' != 'false' and
     exists('$(MSBuildExtensionsPath)\$(MSBuildToolsVersion)\SolutionFile\ImportBefore')" />
  <Import Project="C:\InsideMSBuild\AppxA\before.AppxA.sln.targets"
          Condition="exists('C:\InsideMSBuild\AppxA\before.AppxA.sln.targets')" />

  <!--
  Solution build element here
  -->
  <Import Project="$(MSBuildExtensionsPath)\$(MSBuildToolsVersion)\SolutionFile\
ImportAfter\*"
          Condition="'$(ImportByWildcardBeforeSolution)' != 'false' and
      exists('$(MSBuildExtensionsPath)\$(MSBuildToolsVersion)\SolutionFile\ImportAfter')" />
  <Import Project="C:\InsideMSBuild\AppxA\after.AppxA.sln.targets"
          Condition="exists('C:\InsideMSBuild\AppxA\after.AppxA.sln.targets')" />
</Project>
```

As you can see, there are essentially two sets of imports: one to pull files from the $(MSBuildExtensionsPath)\$(MSBuildToolsVersion)\SolutionFile\Importxxx\ folders and the other to pull files from the same directory as the solution, if those files match a specific pattern. The difference between these approaches is that if you place a file inside the $(MSBuildExtensionsPath)\$(MSBuildToolsVersion)\SolutionFile\ImportBefore\ folder, it will be picked up for every build on that machine. The same applies to the ImportAfter folder. If you create a file matching the pattern before.*{SolutionName}*.sln.targets or after.*{SolutionName}* .sln.targets (where *{SolutionName}* is the name of the solution), then it will be used only for that particular solution. Let's see this at work. In the samples with this book, you will find an AppxA solution file. In the same folder as the AppxA.sln file, I have created the following two files.

before.AppxA.sln.targets

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

  <Target Name="FromBeforeAppxA" AfterTargets="Build">
    <Message Text="FromBeforeAppxA" Importance="high"/>
  </Target>

</Project>
```

after.AppxA.sln.targets

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

  <Target Name="FromAfterAppxA" AfterTargets="Build">
```

```
    <Message Text="FromAfterAppxA" Importance="high"/>
  </Target>

</Project>
```

In this case, for both files I've created a new target and used the *AfterTargets="Build"* attribute value to inject the target into the build process. When I build this solution, the result is shown in Figure A-1.

```
C:\InsideMSBuild\AppxA>msbuild AppxA.sln /nologo
Build started 10/10/2010 1:05:57 PM.
Project "C:\InsideMSBuild\AppxA\AppxA.sln" on node 1 (default targets).
ValidateSolutionConfiguration:
  Building solution configuration "Debug|x86".
Project "C:\InsideMSBuild\AppxA\AppxA.sln" (1) is building "C:\InsideMSBuild\AppxA\WindowsFormsApp
lication1\WindowsFormsApplication1.csproj" (2) on node 1 (default targets).
ResolveAssemblyReferences:
  A TargetFramework profile exclusion list will be generated.
CoreResGen:
  All outputs are up-to-date.
GenerateTargetFrameworkMonikerAttribute:
Skipping target "GenerateTargetFrameworkMonikerAttribute" because all output files are up-to-date
with respect to the input files.
CoreCompile:
Skipping target "CoreCompile" because all output files are up-to-date with respect to the input fi
les.
CopyFilesToOutputDirectory:
  WindowsFormsApplication1 -> C:\InsideMSBuild\AppxA\WindowsFormsApplication1\bin\Debug\WindowsFor
  msApplication1.exe
Done Building Project "C:\InsideMSBuild\AppxA\WindowsFormsApplication1\WindowsFormsApplication1.cs
proj" (default targets).

FromBeforeAppxA:
  FromBeforeAppxA
FromAfterAppxA:
  FromAfterAppxA
Done Building Project "C:\InsideMSBuild\AppxA\AppxA.sln" (default targets).


Build succeeded.
    0 Warning(s)
    0 Error(s)
```

FIGURE A-1 Solution import result

From this figure, you can see that the targets from the before and after targets files were successfully injected into the solution build process.

Property Functions

There are many times that you need to perform a simple operation on a property. For example, you might want to extract a substring from a property or compute its length. With previous versions of MSBuild, you had to use, or create, a task for this. With MSBuild 4.0, you can use property functions. For more information on property functions, see Chapter 3, "MSBuild Deep Dive, Part 2."

Item Functions

With MSBuild 4.0, you can now perform simple operations on item lists. For example, you can compute the number of elements that a list contains, extract the list of distinct elements, get a list of the elements with a given metadata value, and so on. These operations are known as *item functions*. With previous versions of MSBuild, you were forced to use batching or custom tasks to achieve the same results. For more information on item functions, see Chapter 3.

Inline Tasks

With MSBuild 4.0, if you need to create a custom task, you are not forced to precompile that into an assembly and then reference that during your build. You can simply declare the task in code within the project file itself. MSBuild will take care of compiling it for you. We covered inline tasks extensively in Chapter 4, "Custom Tasks."

Cancellable Builds

If you need to cancel a build from the command prompt, you would typically press the Ctrl+C keys. With previous versions of MSBuild, this would abruptly end the process, and your tasks were not given a chance to gracefully handle the cancellation. With MSBuild 4.0, however, a new interface has been introduced, called Microsoft.Build.Framework .ICancelableTask. If your task implements this interface when a build is cancelled, then the *Cancel()* method will be called on the task in order to give it a chance to exit gracefully.

YieldDuringToolExecution

With MSBuild 4.0 comes a new property on the *ToolTask* abstract class, YieldDuringToolExecution. If your tool task has a value of *true* for YieldDuringToolExecution, then the process that launches that tool task will be asynchronous so that other projects can continue to build. For more information on using this property, see Chapter 2.

New Object Model

With MSBuild 4.0, a new object model has been introduced that simplifies the process needed in order to consume the MSBuild application programming interface (API). The MSBuild Object Model is outside the scope of this book. For more information take a look at the MSDN docs on MSBuild.

Debugger

For a while, people have wanted an MSBuild debugger. Now, with MSBuild 4.0, there is a hidden debugger that you can turn on. This is not a supported feature, but you might find it helpful in any case. In order to turn on this feature, we have to walk through a few steps. First, you have to enable the Just My Code option, which can be found on the Tools, Options menu under the Debugging node (see Figure A-2).

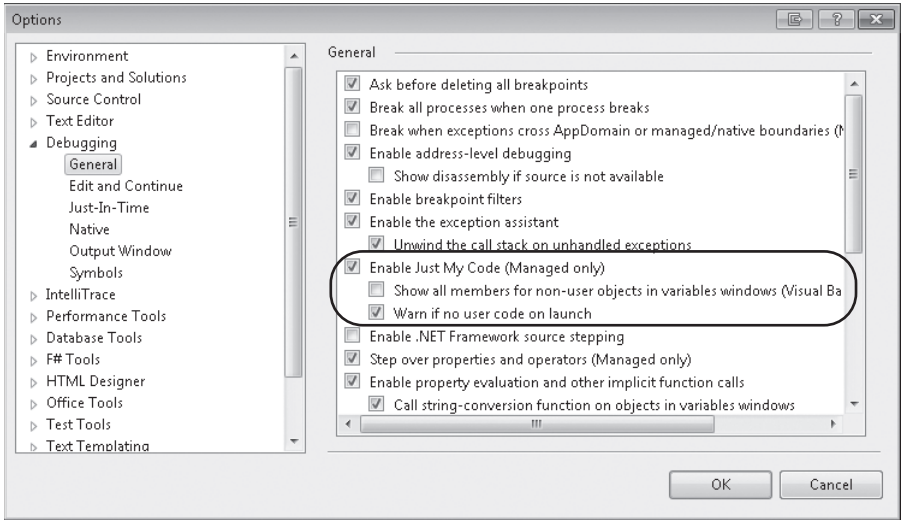

FIGURE A-2 Just My Code setting

After you have set that, you have to change the value of a registry key, which will enable you to pass a /debug switch on the command line to msbuild.exe. From an administrator command prompt, execute the command `reg add "HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\MSBuild\4.0" /v DebuggerEnabled /d true`. The result should be the same as that depicted in Figure A-3.

```
C:\InsideMSBuild>reg add "HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\MSBuild\4.0" /v DebuggerEnabled /d
true
The operation completed successfully.
```

FIGURE A-3 Registry key update

Note If you have a 64-bit machine, then you should run the following commands to ensure that debugging is enabled for both the 32- and 64-bit versions of msbuild.exe.

```
C:\windows\system32\reg add "HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\MSBuild\4.0" /v
DebuggerEnabled /d true

C:\windows\SysWOW64\reg add "HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\MSBuild\4.0" /v
DebuggerEnabled /d true
```

After you have set this key, when you execute the command `msbuild.exe /?`, you should see the /debug switch appear in the usage information as shown in Figure A-4.

If you do not see the /debug switch show up in the usage information, then you have not initialized the registry key correctly.

```
  /preprocess[:file]
                      Creates a single, aggregated project file by
                      inlining all the files that would be imported during a
                      build, with their boundaries marked. This can be
                      useful for figuring out what files are being imported
                      and from where, and what they will contribute to
                      the build. By default the output is written to
                      the console window. If the path to an output file
                      is provided that will be used instead.
                      (Short form: /pp)
                      Example:
                        /pp:out.txt

  /detailedsummary
                      Shows detailed information at the end of the build
                      about the configurations built and how they were
                      scheduled to nodes.
                      (Short form: /ds)

  /debug
                      Causes a debugger prompt to appear immediately so that
                      Visual Studio can be attached for you to debug the
                      MSBuild XML and any tasks and loggers it uses.

  @<file>             Insert command-line settings from a text file. To specify
                      multiple response files, specify each response file
                      separately.

  /noautoresponse     Do not auto-include the MSBuild.rsp file. (Short form:
                      /noautorsp)

  /nologo             Do not display the startup banner and copyright message.

  /version            Display version information only. (Short form: /ver)

  /help               Display this usage message. (Short form: /? or /h)

Examples:

       MSBuild MyApp.sln /t:Rebuild /p:Configuration=Release
       MSBuild MyApp.csproj /t:Clean
                            /p:Configuration=Debug;TargetFrameworkVersion=v3.5
```

FIGURE A-4 msbuild.exe usage information showing /debug switch

After you have enabled the /debug switch, you can just append it to a build. For example, in the samples with this appendix, I have included a project called WindowsFormsApplication1. To debug the build for that project, you can execute the command `msbuild WindowsFormsApplication1.csproj /debug`. After you execute this command, you will see the dialog shown in Figure A-5.

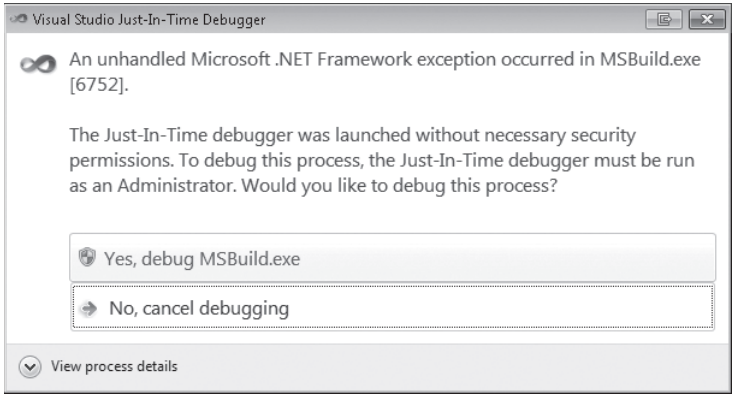

FIGURE A-5 JIT Debugger dialog

In this dialog, you need to click the Yes, Debug MSBuild.exe option. You will need Administrator rights to do this. Following this dialog, you'll see another dialog, shown in Figure A-6.

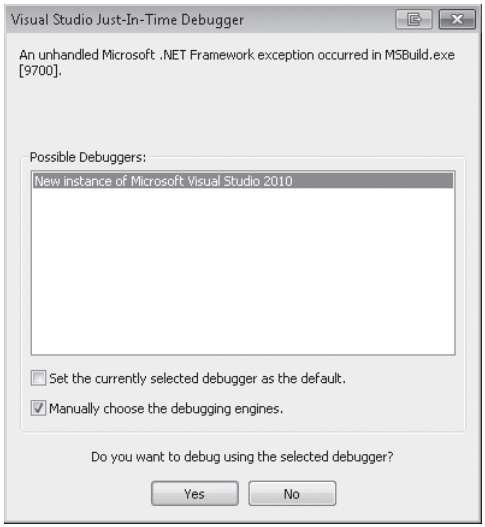

FIGURE A-6 JIT Debugger Selection dialog

When this dialog appears, you should select the Manually Choose The Debugging Engines check box. The reason that we want to do this is because we want to enable only managed debugging. Mixed-mode debugging will slow down the performance of this. In order to do this, after you click Yes in the JIT Debugger Selection dialog, you will be presented with yet another dialog. This one is shown in Figure A-7.

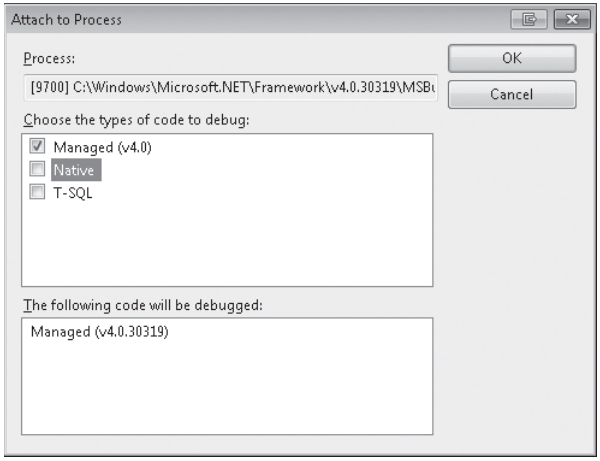

FIGURE A-7 Debugger Selection dialog

You'll be happy to learn that after you click OK, you won't see any more dialogs. You should now see Microsoft Visual Studio open, with a breakpoint set on the Project node.

Now that we have hit a breakpoint in the build file, we can do familiar things such as stepping over, stepping into, stepping out, and so on. While debugging, you can take a look

at the Locals window. If you don't see that, then enable it with the menu option Debug, Windows, Locals. Take a look at this in action in Figure A-8.

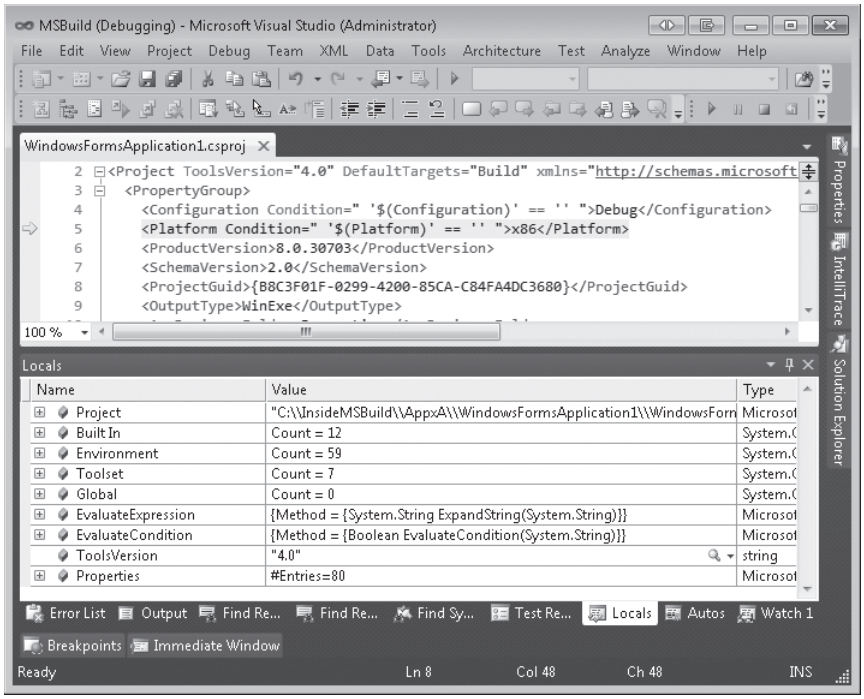

FIGURE A-8 Debugging MSBuild

By using the Locals window, or Watch window, you can examine the different values during the build process. You can also use the Immediate window to view values and make changes during execution. Unfortunately, the syntax that you will have to use in the Immediate window is not the same syntax that you would normally use in MSBuild. For example, to get the value of a property or item, you would use the *EvaluateExpression* method. To get the value of the Configuration property, use the command `EvaluateExpression ("$(Configuration)")`. To see the values in the Compile item list, use the command `EvaluateExpression("@(Compile)")`. In order to set the value of a property or create a new one, you can use the *Project.SetProperty* method. You could implement this with the following code: Project.SetProperty("NewProperty","test"). This would create or update a property named NewProperty to the value *"test"*. Since this is an unsupported feature, there is not that much information about it, and it is still unpolished, hence the many dialogs. However, it can be very useful when you are building a project and don't understand what is going on.

Appendix B

Building Large Source Trees

When you are dealing with a large number of projects (say, more than 100), you need to organize your projects and have a build process that is efficient, yet flexible enough to meet the needs of each of the projects. In this appendix, I describe one means for organizing your source code, as well as an approach for integrating a build process into that structure. The structure I describe won't suit every team or every product, but the important ideas to take away are how to modularize your build process and how to introduce common build elements into all products that are being built.

You can organize your source into trees of related projects, with the most common projects at the top. This organization assumes that projects need to build any of the projects beneath them in the tree and potentially sibling projects, but they should not build projects directly that exist in the nodes above them. For example, Figure A-1 shows the dependency relationships of several fictitious projects.

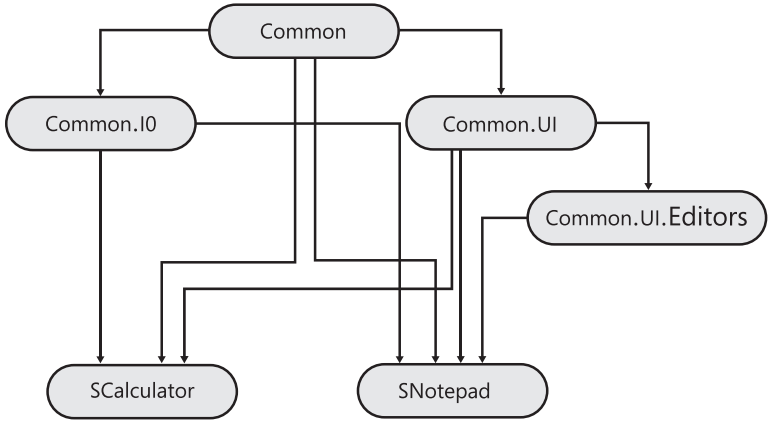

FIGURE A-1 Project dependencies

Here we have two products, SCalculator and SNotepad, and four libraries that they depend on. We could organize these projects into a tree similar to Figure A-2.

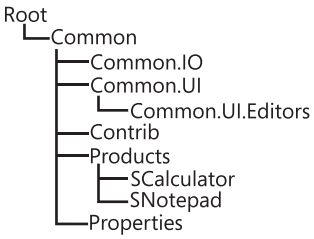

FIGURE A-2 SCalculator and SNotepad organizational tree

Because all the projects depend on the Common project, it is directly under the Root node. The SCalculator and SNotepad projects are placed inside the Products folder.

What you need here is a strategy that allows developers working on specific subtrees to build the pieces that they need, but not necessarily the entire structure. You can achieve this by using a convention in which each folder contains three MSBuild files:

- NodeName.setting
- NodeName.traversal.targets
- dirs.proj

NodeName is the name of the current node—for example, Root, Common, or Common .UI.Editors. The NodeName.setting file contains any settings (captured as properties or items) that are used during the build process. For example, settings here might include BuildInParallel, Configuration, or Platform. The NodeName.traversal.targets file contains the targets that are used to build the projects. Finally, the dirs.proj file maintains a list of projects (in the ProjectFiles item) that need to be built for that subtree.

The NodeName.setting and NodeName.traversal.targets files will always import the top-level corresponding files—root.setting and root.traversal.targets. These top-level files contain the global settings and targets, and the node-level files are where customizations can be injected. In many cases, these node-level files need to import only the root file. The code block that follows shows the contents of the root.traversal.targets file. Fundamentally, there are three targets in this file: Build, Rebuild, and Clean. The properties and other targets are there simply to support these three targets. This file uses the ProjectFiles item, which is declared in the dirs.proj file for that specific directory. The requirements for the dirs.proj file are to do the following:

1. Define all projects to be built using ProjectFiles.
2. Import the NodeName.setting file towards the top.
3. Import the NodeName.targets file near the bottom.

```
<Project xmlns="http://schemas.microsoft.com/developer/msbuild/2003"
         ToolsVersion="4.0">

  <!-- Targets used to build all the projects -->

  <PropertyGroup>
    <BuildDependsOn>
      $(BuildDependsOn);
      CoreBuild
    </BuildDependsOn>
  </PropertyGroup>

  <Target Name="Build" DependsOnTargets="$(BuildDependsOn)" />
  <Target Name="CoreBuild">
```

```
    <!--
    Properties BuildInParallel and SkipNonexistentProjects
    should be defined in the .setting file.
    -->
    <MSBuild Projects="@(ProjectFiles)"
             BuildInParallel="$(BuildInParallel)"
             SkipNonexistentProjects="$(SkipNonexistentProjects)"
             Targets="Build"
             />
  </Target>

  <PropertyGroup>
    <RebuildDependsOn>
      $(RebuildDependsOn);
      CoreRebuild
    </RebuildDependsOn>
  </PropertyGroup>
  <Target Name="Rebuild" DependsOnTargets="$(RebuildDependsOn)" />
  <Target Name="CoreRebuild">
    <MSBuild Projects="@(ProjectFiles)"
             BuildInParallel="$(BuildInParallel)"
             SkipNonexistentProjects="$(SkipNonexistentProjects)"
             Targets="Rebuild"
             />
  </Target>

  <PropertyGroup>
    <CleanDependsOn>
      $(CleanDependsOn);
      CoreClean
    </CleanDependsOn>
  </PropertyGroup>
  <Target Name="Clean" DependsOnTargets="$(CleanDependsOn)" />
  <Target Name="CoreClean">
    <MSBuild Projects="@(ProjectFiles)"
             BuildInParallel="$(BuildInParallel)"
             SkipNonexistentProjects="$(SkipNonexistentProjects)"
             Targets="Clean"
             />
  </Target>
</Project>
```

The dirs.proj file should include all projects in that directory, as well as all projects in subdirectories. It can include normal MSBuild projects, like C# or Microsoft Visual Basic .NET projects, or other dirs.proj projects (for subdirectories). This file should not include projects that exist in directories above it. The dirs.proj file should assume that required projects that are higher in the directory structure are already built.

If you build a project that has a reference to a project that is higher in the directory structure and that project is out of date, the out-of-date project will be built automatically. As a result, the dirs.proj file doesn't have to specify to build higher-level projects. Also, for massive builds, it is better to use file references instead of project references. With this approach, if

you switch to project references, you do not have to modify your build process, only your references. Here are the contents of the root.setting file:

```
<Project xmlns="http://schemas.microsoft.com/developer/msbuild/2003"
         ToolsVersion="4.0">
  <!--
  Global properties defined in this file
  -->
  <PropertyGroup>
    <BuildInParallel
      Condition="'$(BuildInParallel)'==''">true</BuildInParallel>
    <SkipNonexistentProjects
      Condition="'$(SkipNonexistentProjects)'==''">false</SkipNonexistentProjects>
  </PropertyGroup>

</Project>
```

This file contains only two properties: BuildInParallel and SkipNonexistentProjects. It is important to note that these properties use conditions to ensure that any preexisting values are not overwritten, which allows these properties to be customized easily. The next code block contains the contents of the dirs.proj file for the Root directory.

```
<Project xmlns="http://schemas.microsoft.com/developer/msbuild/2003"
         ToolsVersion="4.0">

  <!-- Insert any customizations for settings here -->

  <Import Project="root.setting"/>

  <!-- Define all ProjectFiles here -->
  <ItemGroup>
    <ProjectFiles Include="Common\dirs.proj"/>
  </ItemGroup>

  <Import Project="root.traversal.targets"/>

  <!-- Insert any customizations for targets here -->

</Project>
```

This dirs.proj file meets all three conditions listed earlier. If any customizations for values in the root.setting file need to be specified, they would be placed above the Import element for that file, and any customizations for targets would be placed after the Import element for that file. This dirs.proj file defines the ProjectFiles item to include just the Common\dirs .proj file, which is responsible for building its contents. There are no other projects in the Root folder that need to be built. See the contents of the Common\dirs.proj file, shown in the next snippet.

```
<Project xmlns="http://schemas.microsoft.com/developer/msbuild/2003"
         ToolsVersion="4.0">
```

```
  <!-- Insert any customizations for settings here -->
  <PropertyGroup>
    <SkipNonexistentProjects>true</SkipNonexistentProjects>
  </PropertyGroup>


  <Import Project="common.setting"/>

  <!-- Define all ProjectFiles here -->
  <ItemGroup>
    <ProjectFiles Include="Common.csproj"/>
    <ProjectFiles Include="Common.IO\dirs.proj"/>
    <ProjectFiles Include="Common.UI\dirs.proj"/>
    <ProjectFiles Include="Products\dirs.proj"/>
  </ItemGroup>

  <Import Project="common.traversal.targets"/>

  <!-- Insert any customizations for targets here -->

  <PropertyGroup>
    <BuildDependsOn>
      CommonPrepareForBuild;
      $(BuildDependsOn);
      CommonBuildComplete;
    </BuildDependsOn>
  </PropertyGroup>

  <Target Name="CommonPrepareForBuild">
    <Message Text="CommonPrepareForBuild executed"
             Importance="high"/>
  </Target>
  <Target Name="CommonBuildComplete">
    <Message Text="CommonBuildComplete executed"
             Importance="high"/>
  </Target>
</Project>
```

This file overrides the SkipNonexistentProjects property, setting it to *True*. The ProjectFiles item is populated with four values, three of which are dirs.proj files, and a couple of targets are added to the build dependency list. If you build the Common\dirs.proj file with the command `msbuild.exe dirs.proj /t:Build`, you will see that all the projects are built and that the custom targets execute. I will not include the results here because of space limitations, but the source for these files is available with the samples for this book.

In this appendix, we have looked at a few key recommendations that you can use to create better build processes for your products. As with all best practices, there will be situations in which these rules may not apply 100 percent of the time and need to be bent a little. The best way to learn which of these practices works for you is simply to try each one out for yourself.

I would like to thank Dan Moseley, from the MSBuild team, and Brian Kretzler for their invaluable help on this.

Appendix C

Upgrading from Team Foundation Build 2008

As part of the upgrade process from Team Foundation Server 2008 to Team Foundation Server 2010, all build definitions will be automatically upgraded to use the upgrade template. This build process template provides the ability for Team Foundation Build 2010 to run build processes based on the Team Foundation Build 2008 TFSBuild.proj project file.

Although the upgrade template requires little to no additional effort during the upgrade process, it should be considered a short-term solution until you can switch to the default template or create a customized process template. The upgrade template is lacking a number of features that exist in the default template, such as:

- Source and symbol server support
- Test impact analysis

Upgrade Process

There are five high-level steps to the upgrade from Team Foundation Build 2008 to Team Foundation Build 2010:

1. Upgrade the server to Team Foundation Server 2010. This will modify all the build definitions to use the upgrade template.
2. Install Team Build 2010 on one or more build controllers. Team Build 2008 didn't have a concept of build controllers so you will need to identify machines to act as build controllers. Refer to Chapter 13, "Team Build Quick Start," for more information about setting up build controllers.
3. Upgrade each of your build agents from Team Build 2008 to Team Build 2010. This requires uninstalling Team Build 2008 and installing Team Build 2010. Refer to Chapter 13 for more information about installing Team Build 2010 and setting up build agents.
4. Review build definitions and configure if required.
5. Run test builds to verify the upgraded build definitions and infrastructure.

Upgrade Template

The upgrade template has significantly fewer process parameters than the default template, as can be seen in Figure C-1.

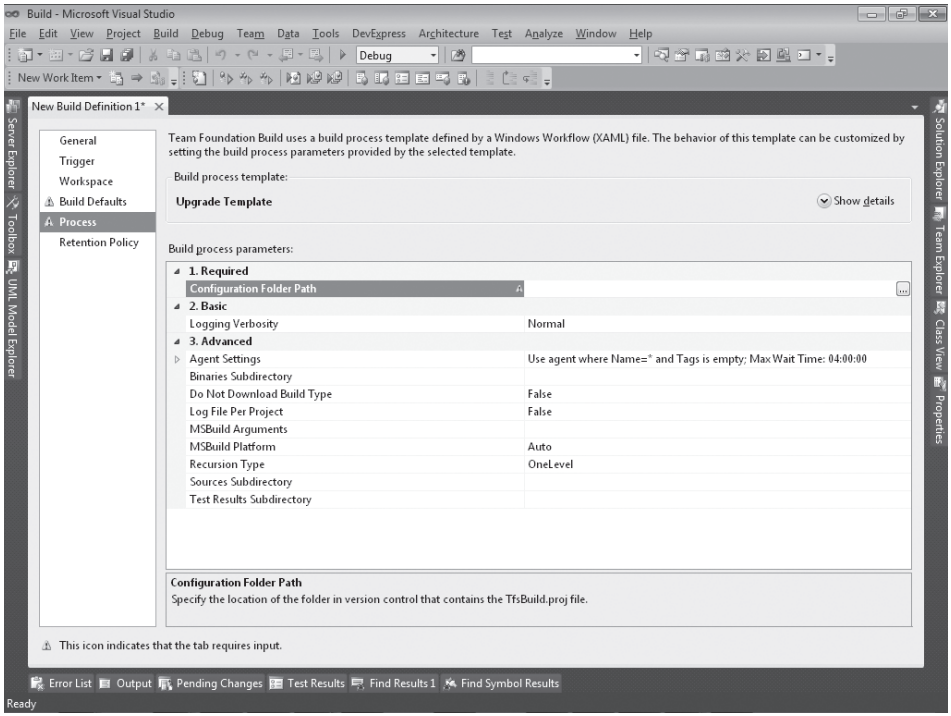

FIGURE C-1 Upgrade template process parameters

The most important parameter is Configuration Folder Path, which contains the version control path to the folder containing TFSBuild.proj. Team Build will download the latest version of this folder to the build agent, but this action can be suppressed by setting the Do Not Download Build Type parameter to *True* (in which case the existing folder from the previous build on the build agent will be used). By default, Team Build will download only immediate children of the Configuration Folder Path. This can be changed, however, by setting the Recursion Type parameter to *Full*.

The Logging Verbosity parameter allows the build's logging verbosity to be specified. In addition, the upgrade template supports splitting the MSBuild logs per project by setting the Log File Per Project parameter to *True*.

The Agent Settings parameter defines the Name Filter, Tags Filter, Tag Comparison, and timeouts to use when selecting and executing on a build agent.

The Sources Subdirectory, Binaries Subdirectory, and Test Results Subdirectory parameters can be used to override the name of the directory (relative to the build agent's working directory) where sources, binaries, and test results (respectively) are placed.

The MSBuild Platform parameter can be used to specify whether to always use the 32-bit version of MSBuild (by selecting X86) or to automatically choose the 32- or 64-bit versions of MSBuild based on the operating system's bitness (by selecting Auto).

Finally, you can specify any additional arguments to be passed to MSBuild by specifying them in the MSBuild Arguments parameter.