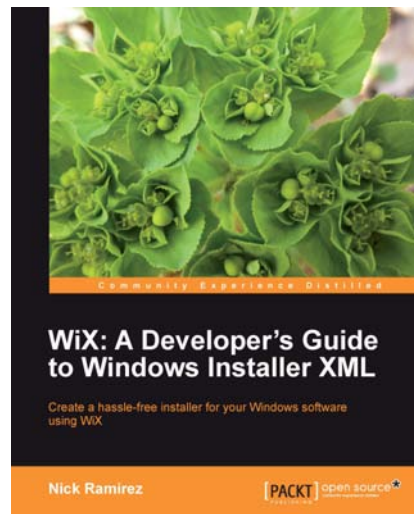


# WiX: A Developer's Guide to Windows Installer XML

Nick Ramirez



## Chapter No.1 "Getting Started"

## In this package, you will find:

A Biography of the author of the book

A preview chapter from the book, Chapter NO.1 "Getting Started"

A synopsis of the book's content

Information on where to buy this book

## About the Author

**Nick Ramirez** is a software engineer working at Sophos in Columbus, Ohio. He previously worked with open source solutions like Linux and PHP before moving to C#, WiX, and other Windows technologies. As a member of the engineering team, he has helped to develop install code for the company's enterprise software.

I would like to thank especially my fiancée, Heidi, for her patience while I worked on this book. I'd also like to thank the Packt team for their hard work and guidance along the way. Another big thanks to Martin Oberhammer and Neil Sleightholm for their WiX expertise and help in filling in the gaps.

**For More Information:**

[www.PacktPub.com/wix-a-developers-guide-to-windows-installer-xml/book](http://www.PacktPub.com/wix-a-developers-guide-to-windows-installer-xml/book)

# WiX: A Developer's Guide to Windows Installer XML

Since Rob Mensching offered up the WiX toolset as the first open source project from Microsoft in 2004, it has been quietly gaining momentum and followers. Today, thousands use it to build Windows Installer packages from simple XML elements. Gone are the days when you would have had to pay for software to build an installer for you. Now, you can do it yourself for cheap.

Not only that, but WiX has matured into a fairly slick product that's sufficiently easy to use. Best of all, it has all of the bells and whistles you want including the functionality to add user interface wizards, Start Menu shortcuts, control Windows services, and read and write to the Registry.

*WiX: A Developer's Guide to Windows Installer XML* gives you the knowledge to start building sophisticated installers right away, even if you have no prior experience doing so. Each chapter gets straight to the point, giving you hands on experience, so you'll master the technology quickly.

## What This Book Covers

Chapter 1, Getting Started, explains how after downloading and installing the WiX toolset, you'll start using it right away to create a simple installer. Then, you'll see how to add a basic user interface to it, install it with logging turned on, and view its internal database.

Chapter 2, Creating Files and Directories, gives you deeper understanding of how files are installed and what is the best way to organize them in your project. You'll then use the tool `heat.exe` to generate WiX markup. Last, you'll learn about copying and moving files, and installing special-case files.

Chapter 3, Putting Properties and AppSearch to Work, gets you introduced to Windows Installer properties as you create your own and use those that are built in. Later, you'll check the end user's computer for specific files, directories, Registry keys, and INI file settings using AppSearch.

Chapter 4, Improving Control with Launch Conditions and Installed States, allows you to leverage conditional statements to set prerequisites for running your installer and to exclude particular features or components from the install. You'll also discover how to check the action and installed state of your features and components.

Chapter 5, Understanding the Install Sequence, allows you to get a clear picture of how the whole installation process works as you examine the order and meaning of installer

**For More Information:**

[www.PacktPub.com/wix-a-developers-guide-to-windows-installer-xml/book](http://www.PacktPub.com/wix-a-developers-guide-to-windows-installer-xml/book)

actions. You will then create custom actions and add them to this builtin sequence to extend the functionality. Then, you'll learn the basics of using the Deployment Tools Foundation library for writing custom action code in C#.

Chapter 6, Adding a User Interface, after giving you a quick introduction to the standard dialog wizards that come with the WiX toolset, allows you to start building your own from scratch. You'll learn all of the required elements to display dialogs and to link them together. You'll also see how to build dialogs for displaying errors and user exits.

Chapter 7, Using UI Controls, gives you hands on experience with each type of UI control including buttons, textboxes, and progress bars.

Chapter 8, Tapping into Control Events, breathes life into your UI controls by having them publish and subscribe to events.

Chapter 9, Working from the Command Line, compiles your code from the command line and then links and binds object files into an MSI. You'll learn to use preprocessor variables and conditional statements and how to create custom preprocessor extensions.

Chapter 10, Accessing the Windows Registry, allows you to read from the Windows Registry and add keys and values to it at install time. You'll also learn how to remove existing keys, copy values, and set permissions.

Chapter 11, Controlling Windows Services, installs Windows services and issues start, stop, and remove commands to them. You'll learn to set the service's user account, add service dependencies, and set failure recovery.

Chapter 12, Localizing Your Installer, creates localized installers for different languages and teaches how light.exe, the WiX linker, plays a role. You'll then learn how to make a single multi-language installer.

Chapter 13, Upgrading and Patching, allows you to learn how to plan for and implement a major upgrade of your product and how to make small updates using patch files.

**For More Information:**

**[www.PacktPub.com/wix-a-developers-guide-to-windows-installer-xml/book](http://www.PacktPub.com/wix-a-developers-guide-to-windows-installer-xml/book)**

# 1

## Getting Started

**Windows Installer XML (WiX)** is a free XML markup from Microsoft that is used to author installation packages for Windows-based software. The underlying technology is Windows Installer, which is the established standard for installing desktop-based applications to any Windows operating system. It is used by countless companies around the world. Microsoft uses it to deploy its own software including Microsoft Office and Visual Studio. In fact, Microsoft uses WiX for these products.

Windows Installer has many features, but how do you leverage them? How do you even know what they are? This book will help you by making you more familiar with the wide range of capabilities that are available. WiX makes many of the arcane and difficult to understand aspects of Windows Installer technology simple to use. This book will teach you the WiX syntax so that you can create a professional-grade installer that's right for you.

In this chapter, we will cover the following:

- Getting WiX and using it with Visual Studio
- Creating your first WiX installer
- Examining an installer database with Orca
- Logging an installation process
- Adding a simple user interface

## Introducing Windows Installer XML

In this section, we'll dive right in and talk about what WiX is, where to get it, and why you'd want to use it when building an installation package for your software. We'll follow up with a quick description of the WiX tools and the new project types made available in Visual Studio.

**For More Information:**

[www.PacktPub.com/wix-a-developers-guide-to-windows-installer-xml/book](http://www.PacktPub.com/wix-a-developers-guide-to-windows-installer-xml/book)

## What is WiX?

Although it's the standard technology and has been around for years, creating a Windows Installer, or **MSI** package, has always been a challenging task. The package is actually a relational database that describes how the various components of an application should be unpacked and copied to the end user's computer.

In the past you had two options:

- You could try to author the database yourself – a path that requires a thorough knowledge of the Windows Installer API.
- You could buy a commercial product like InstallShield to do it for you. These software products will take care of the details, but you'll forever be dependent on them. There will always be parts of the process that are hidden from you.

WiX is relatively new to the scene, offering a route that exists somewhere in the middle. Abstracting away the low-level function calls while still allowing you to write much of the code by hand, WiX is an architecture for building an installer in ways that mere mortals can grasp. Best of all, it's free. As an open source product, it has quickly garnered a wide user base and a dedicated community of developers. Much of this has to do not only with its price tag but also with its simplicity. It can be authored in a simple text editor (such as Notepad) and compiled with the tools provided by WiX. As it's a flavor of XML, it can be read by humans, edited without expensive software, and lends itself to being stored in source control where it can be easily merged and compared.

The examples in this first chapter will show how to create a simple installer with WiX using Visual Studio. However, later chapters will show how you can build your project from the command line using the compiler and linker from the WiX toolset. The WiX source code is available for download, so you can be assured that nothing about the process will be hidden if you truly need to know.

## Is WiX for you?

To answer the question "Is WiX for you?" we have to answer "What can WiX *do for you*?" It's fairly simple to copy files to an end user's computer. If that's all your product needs, then the Windows Installer technology might be overkill. However, there are many benefits to creating an installable package for your customers, some of which might be overlooked. Following is a list of features that you get when you author a Windows Installer package with WiX:

- All of your executable files can be packaged into one convenient bundle, simplifying deployment.

- Your software is automatically registered with **Add/Remove Programs**.
- Windows takes care of uninstalling all of the components that make up your product when the user chooses to do so.
- If files for your software are accidentally removed, they can be replaced by right-clicking on the MSI file and selecting **Repair**.
- You can create different versions of your installer and detect which version has been installed.
- You can create patches to update only specific areas of your application.
- If something goes wrong while installing your software, the end user's computer can be rolled back to a previous state.
- You can create **Wizard**-style dialogs to guide the user through the installation.

Many people today simply expect that your installer will have these features. Not having them could be seen as a real deficit. For example, what is a user supposed to do when they want to uninstall your product but can't find it in the **Add/Remove Programs** list and there isn't an uninstall shortcut? They're likely to remove files haphazardly and wonder why you didn't make things easy for them.

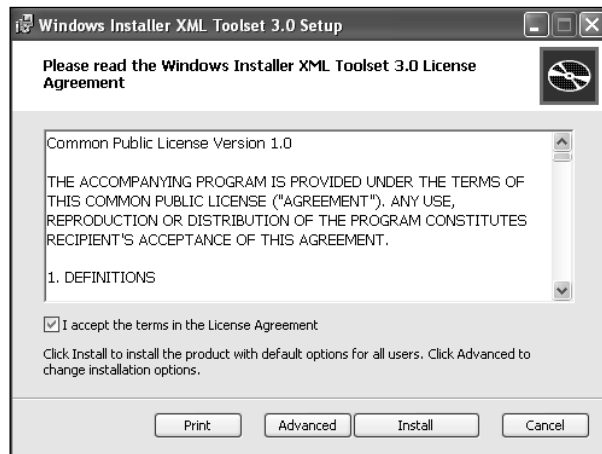
Maybe you've already figured that Windows Installer is the way to go, but why WiX? One of my favorite reasons is that it gives you greater control over how things work. You get a much finer level of control over the development process. Commercial software that does this for you also produces an MSI file, but hides the details about how it was done. It's analogous to crafting a web site. You get much more control when you write the HTML yourself as opposed to using WYSIWYG software.

Even though WiX gives you more control, it doesn't make things overly complex. You'll find that making a simple installer is very straightforward. For more complex projects, the parts can be split up into multiple XML source files to make it easier to work with. Going further, if your product is made up of multiple products that will be installed together as a suite, you can compile the different chunks into libraries that can be merged together into a single MSI. This allows each team to isolate and manage their part of the installation package.

WiX is a stable technology, having been first released to the public in 2004, so you don't have to worry about it disappearing. It's also had a steady progression of version releases. The most current version is updated for Windows Installer 4.5 and the next release will include changes for Windows Installer 5.0, which is the version that comes preinstalled with Windows 7. These are just some of the reasons why you might choose to use WiX.

## Where can I get it?

You can download WiX from the Codeplex site, <http://wix.codeplex.com/>, which has both stable releases and source code. The current release is version 3.0. Once you've downloaded the WiX installer package, double-click it to install it to your local hard drive.



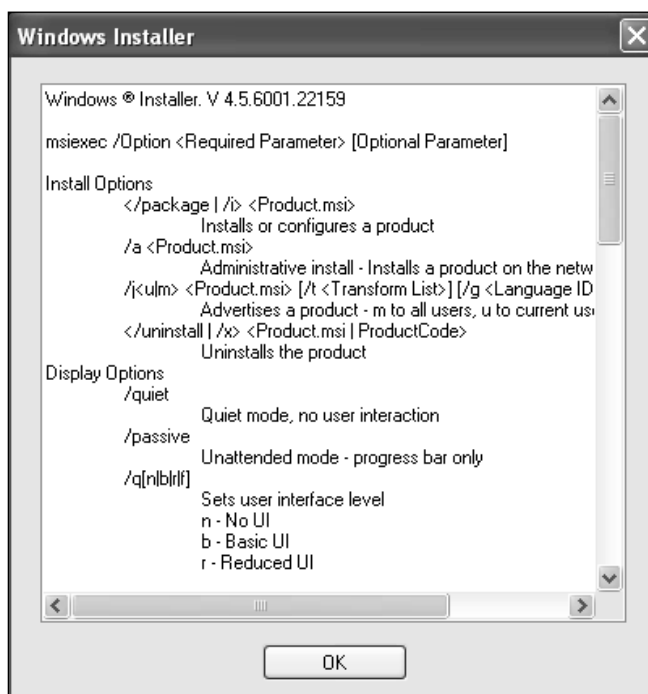
This installs all of the necessary files needed to build WiX projects. You'll also get the WiX SDK documentation and the settings for Visual Studio IntelliSense, highlighting and project templates. Version 3 supports Visual Studio 2005 and Visual Studio 2008, Standard edition or higher.

WiX comes with the following tools:

Tool	What it does
Candle.exe	Compiles WiX source files (.wxs) into intermediate object files (.wixobj).
Light.exe	Links and binds .wixobj files to create final .msi file. Also creates cabinet files and embeds streams in MSI database.
Lit.exe	Creates WiX libraries (.wixlib) that can be linked together by Light.
Dark.exe	Decompiles an MSI file into WiX code.
Heat.exe	Creates a WiX source file that specifies components from various inputs.
Melt.exe	Converts a "merge module" (.msm) into a component group in a WiX source file.
Torch.exe	Generates a transform file used to create a software patch.
Smoke.exe	Runs validation checks on an MSI or MSM file.
Pyro.exe	Creates an patch file (.msp) from .wixmsp and .wixmst files.
WixCop.exe	Converts version 2 WiX files to version 3.



In order to use some of the functionality in WiX, you may need to download a more recent version of Windows Installer. You can check your current version by viewing the help file for `msiexec.exe`, which is the Windows Installer service. Go to your **Start Menu** and select **Run**, type `cmd` and then type `msiexec /?` at the prompt. This should bring up a window like the following:



If you'd like to install a newer version of Windows Installer, you can get one from the Microsoft Download Center website. Go to:

<http://www.microsoft.com/downloads/en/default.aspx>

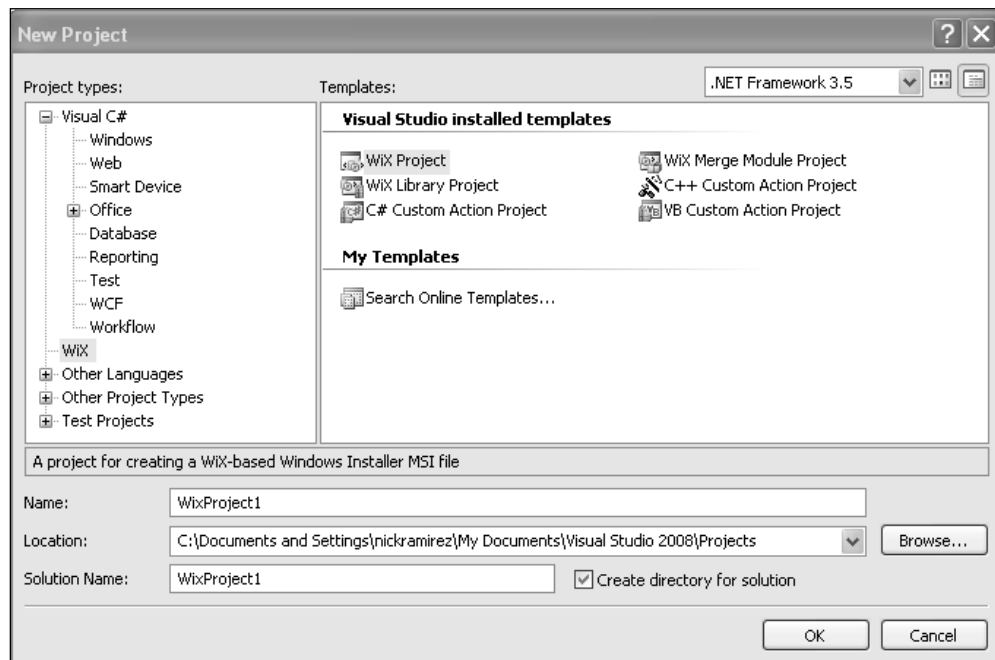
Search for **Windows Installer**. The current version for Windows XP, Vista, Server 2003, and Server 2008 is 4.5. Windows 7 and Windows Server 2008 R2 can support version 5.0. Each new version is backwards compatible and includes the features from earlier editions.

## Votive

The WiX toolset provides files that update Visual Studio to provide new WiX IntelliSense, syntax highlighting, and project templates. Together these features, which are installed for you along with the other WiX tools, are called **Votive**. You must have Visual Studio 2005 or 2008 (Standard edition or higher). Votive won't work on the Express versions. If you're using Visual Studio 2005, you may need to install an additional component called **ProjectAggregator2**. Refer to the WiX site for more information:

<http://wix.sourceforge.net/votive.html>

After you've installed WiX, you should see a new category of project types in Visual Studio, labeled under the title **WiX**. To test it out, open Visual Studio and go to **File | New | Project**. Select the category **WiX**.



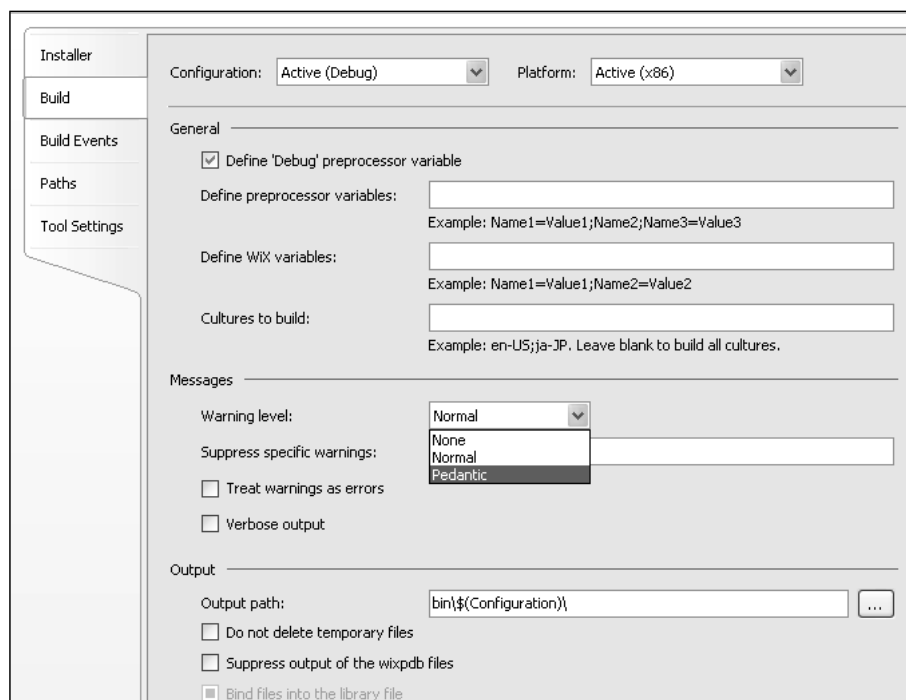
There are six new project templates:

- **WiX Project:** Creates a Windows Installer package from one or more WiX source files
- **WiX Library Project:** Creates a .wixlib library
- **C# Custom Action Project:** Creates a .NET custom action in C#

- **WiX Merge Module Project:** Creates a merge module
- **C++ Custom Action Project:** Creates an unmanaged C++ custom action
- **VB Custom Action Project:** Creates a VB.NET custom action

Using these templates is certainly easier than creating them on your own with a text editor. To start creating your own MSI installer, select the template **WiX Project**. This will create a new `.wxs` (WiX source file) for you to add XML markup to. Once we've added the necessary markup, you'll be able to build the solution by selecting **Build Solution** from the **Build** menu or by right-clicking on the project in the **Solution Explorer** and selecting **Build**. Visual Studio will take care of calling `candle.exe` and `light.exe` to compile and link your project files.

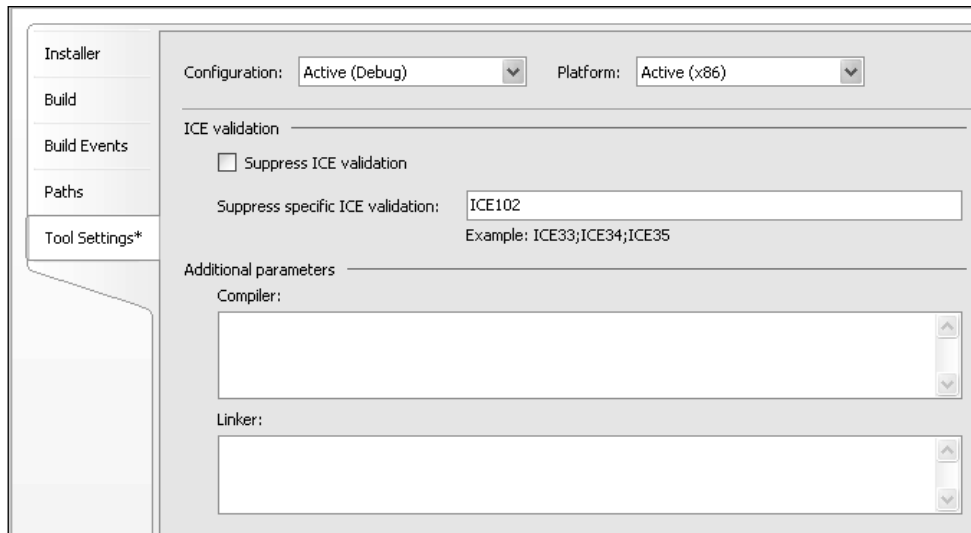
If you right-click on your WiX project in the **Solution Explorer** and select **Properties**, you'll see several screens where you can tweak the build process. One thing you'll want to do is set the amount of information that you'd like to see when compiling and linking the project and how non-critical messages are treated. Refer to the following screenshot:



Here, we're selecting the level of messages that we'd like to see. To see all warnings and messages, set the **Warning Level** to **Pedantic**. You can also check the **Verbose output** checkbox to get even more information. Checking **Treat warnings as errors** will cause warning messages that normally would not stop the build to be treated as fatal errors.

You can also choose to suppress certain warnings. You'll need to know the specific warning message number, though. If you get a build-time warning, you'll see the warning message, but not the number. One way to get it is to open the WiX source code (available at <http://wix.codeplex.com/SourceControl/list/changesets>) and view the `messages.xml` file in the Wix solution. Search the file for the warning and from there you'll see its number. Note that you can suppress warnings but not errors.

Another feature of WiX is its ability to run validity checks on the MSI package. Windows Installer uses a suite of tests called **Internal Consistency Evaluators** (ICEs) for this. These checks ensure that the database as a whole makes sense and that the keys on each table join correctly. Through Votive, you can choose to suppress specific ICE tests. Use the **Tools Setting** page of the project's properties as shown in the following screenshot:



In this example, ICE test 102 is being suppressed. You can specify more than one test by separating them with semicolons. To find a full list of ICE tests, go to MSDN's **ICE Reference** web page at:

<http://msdn.microsoft.com/en-us/library/aa369206%28VS.85%29.aspx>

The **Tool Settings** screen also gives you the ability to add compiler or linker command-line flags. Simply add them to the text boxes at the bottom of the screen. We will discuss command-line arguments for Candle and Light later in the book.

## A word about GUIDs

In various places throughout WiX, you'll be asked to provide a **GUID**, which is a **Globally Unique Identifier**. This is so that when your product is installed on the end user's computer, references to it can be stored in the Windows Registry without the chance of having name conflicts. By using GUIDs, Windows Installer can be sure that every software application, and even every component of that software, has a unique identity on the system.

Each GUID that you create on your computer is guaranteed to be different from a GUID that someone else would make. Using this, even if two pieces of software, both called "Amazing Software", are installed on the same computer, Windows will be able to tell them apart.

Visual Studio 2008 provides a way to create a GUID. Go to **Tools | Create GUID** and copy a new GUID using the **Registry Format**. WiX can accept a GUID with or without curly brackets around it as 01234567-89AB-CDEF-0123-456789ABCDEF or {01234567-89AB-CDEF-0123-456789ABCDEF}.

Be sure to only use uppercase letters, though. In this book, I'll display real GUIDs, but you should not reuse them as then your components will not be guaranteed to be unique.

## Your first WiX project

To get started, download the WiX Toolset. It can be found at:

<http://wix.codeplex.com/>

Once you've downloaded and installed it, open Visual Studio and select **New Project | WiX | WiX Project**. This will create a solution with a single `.wxs` (WiX source) file. Visual Studio will usually call this file `Product.wxs`, but the name could be anything as long as it ends with `.wxs`.

Even the most minimal installer must have the following XML elements:

- an XML declaration
- a `wix` element that serves as the root element in your XML document
- a `Product` element that is a child to the `wix` element, but all other elements are children to it

- a Package element
- a Media element
- at least one Directory element with at least one child Component element
- a Feature element

## XML declaration and Wix element

Every WiX project begins with an XML declaration and a wix element.

```
<?xml version="1.0" encoding="UTF-8"?>
<Wix xmlns="http://schemas.microsoft.com/wix/2006/wi">

</Wix>
```

The xmlns, or XML namespace, just brings the core WiX elements into the local scope of your document. At the bottom of the file, you'll have to close the wix element, of course. Otherwise, it's not valid XML. The wix element is the root element of the document. It comes first and last. All other elements will be nested inside of it.

At this point, you could also add the RequiredVersion attribute to the wix element. Given a WiX toolset version number, such as "3.0.5419.0", it won't let anyone compile the .wxs file unless they have that version or higher installed. If, on the other hand, you're the only one compiling your project, then it's no big deal.

## Product element

Next, add a Product element.

```
<Wix ... >
  <Product Id="3E786878-358D-43AD-82D1-1435ADF9F6EA"
    Name="Awesome Software"
    Language="1033"
    Version="1.0.0.0"
    Manufacturer="Awesome Company"
    UpgradeCode="B414C827-8D81-4B4A-B3B6-338C06DE3A11">
  </Product>
</Wix>
```

This is where you define the characteristics of the software you're installing: its name, language, version, and manufacturer. The end user will be able to see these properties by right-clicking on your MSI file, selecting **Properties** and viewing the **Summary** tab. Most of the time, these values will stay the same from one build of

your project to the next. The exception is when you want to increment the software's version or indicate that it's an upgrade of a previous installation. In that case you need only change the `Version`, and sometimes `Id`, attribute. We'll talk more about upgrading previous installations later on in the book.

The `Product` element's `Id` attribute represents the so-called `ProductCode` of your software. It's always a unique number – a GUID – that Windows will use to uniquely identify your software (and tell if it's already installed on the computer). You can either hardcode it, like here, or just put an asterisk. That way, WiX will pick a new GUID for you each time you compile the project.

```
<Wix ... >
  <Product Id="*"
    Name="Awesome Software"
    Language="1033"
    Version="1.0.0.0"
    Manufacturer="Awesome Company"
    UpgradeCode="B414C827-8D81-4B4A-B3B6-338C06DE3A11">
  </Product>
</Wix>
```

The `Name` attribute defines the name of the software. In addition to being displayed in the MSI file's **Properties** page, it will also be shown in various places throughout the user interface of your installer – that is, once you've added a user interface, which we'll touch on at the end of this chapter.

The `Language` attribute is used to display error messages and progress information in the specified language to the user. It's a decimal language ID (LCID). A full list can be found on Microsoft's LCID page at:

<http://msdn.microsoft.com/en-us/global/bb964664.aspx>

The previous example used "1033", which stands for "English-United States". If your installer uses characters not found in the ASCII character set, you'll also need to add a `Codepage` attribute set to the code page that contains those characters. Don't worry too much about this now. We'll cover languages and code pages later in the book when we talk about localization.

The `Version` attribute is used to set the version number of your software. It can accept up to four digits separated by periods, although the last digit is ignored by Windows Installer during operations such as detecting previously installed versions of your application. Typically, when you make a big enough change to the existing software, you'll increment the number. Companies often use the `[MajorVersion].[MinorVersion].[Build].[Revision]` format, but you're free to use any numbering system you like.

The `Manufacturer` attribute tells the user who this software is from and usually contains the name of your company. This is another bit of information that's available via the MSI file's **Properties**.

The final attribute to consider is `UpgradeCode`. This should be set to a GUID and will identify your product across version releases. Therefore, it should stay the same even when the Product ID and Version change. Windows will use this number in its efforts to keep track of all the software installed on the machine. WiX has the ability to search for previously installed versions of not only your own software but also those created by others and it uses `UpgradeCode` to do it. Although, technically, this is an optional attribute, you should always supply it.

## Package element

Once you've defined your `Product` element, the next step is to nest a `Package` element inside. An example is shown:

```
<Wix ... >
  <Product ... >
    <Package Compressed="yes"
      InstallerVersion="301"
      Manufacturer="Awesome Company"
      Description="Installs Awesome Software"
      Keywords="Practice,Installer,MSI"
      Comments="(c) 2010 Awesome Company" />
  </Product>
</Wix>
```

Of the attributes shown in this example, only `Compressed` is really required. By setting `Compressed` to `yes`, you're telling the installer to package all of the MSI's resources into CAB files. Later, you'll define these CAB files with `Media` elements.

Technically, an `Id` attribute is also required, but by omitting it, you're letting WiX create one for you. You'd have to create a new one anyway since every time you change your software or the installer in any way, the "package" (the MSI file) has changed and so the ID must change. This really, in itself, emphasizes what the `Package` element is. Unlike the `Product` element, which describes the software that's in the installer, the `Package` element describes the installer itself. Once you've built it, you'll be able to right-click on the MSI and select **Properties** to see the attributes you've set here.

The `InstallerVersion` attribute can be set to require a specific version of `msiexec.exe` (the Windows Installer service that installs the MSI when you double-click on it) to be installed on the end user's computer. If they have an older version, Windows



Installer will display a `MessageBox` telling them that they need to upgrade. It will also prevent you from compiling the project unless you also have this version installed on your own computer. The value can be found by multiplying the major version by 100 and adding the minor version. So, for version 4.5 of `msiexec.exe`, you'd set `InstallerVersion` to "405".

The rest of the attributes shown provide additional information for the MSI file's **Properties** window. `Manufacturer` is displayed in the **Author** text field, `Description` is shown as **Subject**, `Keywords` show up as **Keywords**, and `Comments` show as **Comments**. It's usually a good idea to provide at least some of this information, if just to help *you* distinguish one MSI package from another.

## Media element

The files that you intend to install are bundled up into CAB files. You have the option of splitting your package into several parts or keeping it all in one. For each `Media` element that you add to your WiX markup, a new CAB file will be created. Generally, you should limit the number of files you put into a single CAB file to 64 K or less and no single file should be larger than 2 GB. You can find more information about the CAB file format at:

[http://msdn.microsoft.com/en-us/library/ee177956\(v=EXCHG.80\).aspx](http://msdn.microsoft.com/en-us/library/ee177956(v=EXCHG.80).aspx)

`Media` elements come nested inside the `Product` element alongside the `Package` element.

```
<Wix ... >
  <Product ... >
    <Package ... />
    <Media Id="1"
      Cabinet="media1.cab"
      EmbedCab="yes" />
  </Product>
</Wix>
```

Each `Media` element gets a unique `Id` attribute to distinguish it in the MSI **Media** table. It must be a positive integer. If the files that you add to your installation package don't explicitly state which CAB file they wish to be packaged into, they'll default to using a `Media` element with an `Id` of 1. Therefore, your first `Media` element should always use an `Id` of 1.

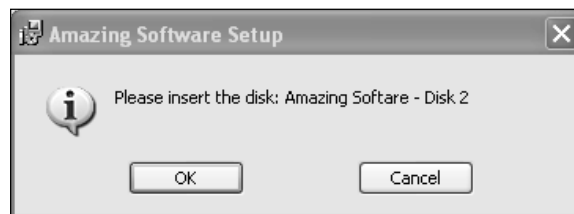
The `Cabinet` attribute sets the name of the CAB file. You won't actually see this unless you set `EmbedCab` to `no`, in which case the file won't be bundled inside the MSI package. This is atypical, but might be done to split the installation files onto several disks. Even this is becoming rare in the age of 4 GB DVDs and Internet downloads. Setting `EmbedCab` to `no` will produce a visible CAB file that must be provided alongside the MSI file during an installation.

If you do choose to split the installation up into several physical disks (or even virtual ISO images), you'll want to add the `DiskPrompt` and `VolumeLabel` attributes. In the following example, I've added two `Media` elements instead of one. I've also added a `Property` element above them, which defines a variable called `DiskPrompt` with a value of `Amazing Software - [1]`.

```
<Property Id="DiskPrompt"
  Value="Amazing Software - [1]" />
<Media Id="1"
  Cabinet="media1.cab"
  EmbedCab="no"
  DiskPrompt="Disk 1"
  VolumeLabel="Disk1" />

<Media Id="2"
  Cabinet="media2.cab"
  EmbedCab="no"
  DiskPrompt="Disk 2"
  VolumeLabel="Disk2" />
```

The `Property` element will be used as the text in the `MessageBox` the end user sees prompting them to insert the next disk. The text in the `DiskPrompt` attribute is combined with the text in the property's value, switched with `[1]`, to change the message for each subsequent disk. Make sure you give this property an `Id` of `DiskPrompt`.



So that Windows will know when the correct disk is inserted, the `VolumeLabel` attribute must match the "Volume Label" of the actual disk, which you'll set with whichever CD or DVD burning program you use. Once you've built your project, include the MSI file and first CAB file on the first disk. The second CAB file should then be written to a second disk.

Although we haven't described the `File` element yet, it's used to add a file to the installation package. To include one in a *specific* CAB file, add the `DiskId` attribute, set to the `Id` of the corresponding `Media` element. The following example includes a text file called `myFile.txt` in the `media2.cab` file:

```
<File Id="fileTXT"
      Name="myFile.txt"
      Source="myFile.txt"
      KeyPath="yes"
      DiskId="2" />
```

We'll discuss the `File` element in more detail later on in the chapter. If you're only using one `Media` element, you won't need to specify the `DiskId` attribute on your `File` elements.

## Directories

So, now we've defined the identity of the product, set up its package properties, and told the installer to create a CAB file to package up the things we'll eventually install. Then, how do you decide where your product will get installed *to* on the end user's computer? How do we set the default installation path, such as some folder under `Program Files`?

When you want to install to `C:\Program Files`, you can use a sort of shorthand. There are several directory names provided by Windows Installer that will be translated to their true paths at install time. For example, `ProgramFilesFolder` usually translates to `C:\Program Files`. Following is a list of these built-in directory properties:

Directory property	Actual path
<code>AdminToolsFolder</code>	Full path to directory containing administrative tools
<code>AppDataFolder</code>	Full path to Roaming folder for current user
<code>CommonAppDataFolder</code>	Full path to application data for all users
<code>CommonFiles64Folder</code>	Full path to 64-bit Common Files folder
<code>CommonFilesFolder</code>	Full path to Common Files folder for current user
<code>DesktopFolder</code>	Full path to Desktop folder
<code>FavoritesFolder</code>	Full path to Favorites folder for current user
<code>FontsFolder</code>	Full path to Fonts folder
<code>LocalAppDataFolder</code>	Full path to folder containing local (non-roaming) applications
<code>MyPicturesFolder</code>	Full path to Pictures folder
<code>NetHoodFolder</code>	Full path to NetHood folder

Directory property	Actual path
PersonalFolder	Full path to Documents folder for current user
PrintHoodFolder	Full path to PrintHood folder
ProgramFiles64Folder	Full path to 64-bit Program Files folder
ProgramFilesFolder	Full path to 32-bit Program Files folder
ProgramMenuFolder	Full path to Program Menu folder
RecentFolder	Full path to Recent folder
SendToFolder	Full path to SendTo folder for current user
StartMenuFolder	Full path to Start Menu folder
StartupFolder	Full path to Startup folder
System16Folder	Full path to 16-bit system DLLs folder
System64Folder	Full path to System64 folder
SystemFolder	Full path to System folder for current user
TempFolder	Full path to Temp folder
TemplateFolder	Full path to Template folder for current user
WindowsFolder	Full path to Windows folder

The easiest way to add your own directories is to nest them inside one of the predefined ones. For example, to create a new directory called `Install Practice` inside the Program Files folder, you could add it as a child to `ProgramFilesFolder`. To define your directory structure in WiX, use `Directory` elements:

```
<Wix ... >
  <Product ... >
    <Package ... />
    <Media ... />

    <Directory Id="TARGETDIR"
      Name="SourceDir">
      <Directory Id="ProgramFilesFolder">
        <Directory Id="MyProgramDir"
          Name="Install Practice" />
      </Directory>
    </Directory>

  </Product>
</Wix>
```

You should place your `Directory` elements inside of the top-level `Product` element. Other than that, there aren't any restrictions about exactly where inside `Product` they have to go. One thing to know is that you must start your `Directory` elements hierarchy with a `Directory` with an `Id` of `TARGETDIR` and a `Name` of `SourceDir`. This sets up the "root" directory of your installation. Therefore, be sure to always create it first and nest all other `Directory` elements inside.

By default, Windows Installer sets `TARGETDIR` to the local hard drive with the most free space—in most cases, the `C:` drive. However, you can set `TARGETDIR` to another drive letter during installation. You might, for example, set it with a `VolumeSelectCombo` user interface control. We'll talk about setting properties and UI controls later in the book.

A `Directory` element always has an `Id` attribute that will serve as a primary key on the `Directory` table. If you're using a predefined name, such as `ProgramFilesFolder`, use that for `Id`. Otherwise, you can make one up yourself. The previous example creates a new directory called `Install Practice` inside the `Program Files` folder. The `Id`, `MyProgramDir`, is an arbitrary value.

When creating your own directory, you must provide the `Name` attribute. This sets the name of the new folder. Without it, the directory won't be created and any files that were meant to go inside it will instead be placed in the parent directory—in this case, `Program Files`. Note that you do not need to provide a `Name` attribute for predefined directories.

You can nest more subdirectories inside your folders by adding more `Directory` elements. Here is an example:

```
<Directory Id="TARGETDIR"
  Name="SourceDir">
  <Directory Id="ProgramFilesFolder">
    <Directory Id="MyProgramDir"
      Name="Install Practice">
      <Directory Id="MyFirstSubDir"
        Name="Subdirectory 1">
        <Directory Id="MySecondSubDir"
          Name="Subdirectory 2" />
        </Directory>
      </Directory>
    </Directory>
  </Directory>
</Directory>
```

Here, a subdirectory called `Subdirectory 1` is placed inside the `Install Practice` folder. A second subdirectory, called `Subdirectory 2`, is then placed inside `Subdirectory 1`, giving us two levels of nested directories under `Install Practice`.

To put something inside a directory, use a `DirectoryRef` element. `DirectoryRef` takes only a single attribute: `Id`. This is your reference to the `Id` set on the corresponding `Directory` element. `DirectoryRef` elements, like `Directory` elements, are placed as children to the top-level `Product` element.

Using a `DirectoryRef` adds a layer of abstraction between where you define your directory structure and the files that will go into those directories. The following example adds a file (via the `Component` element, which we'll cover next) to the `MyProgramDir` directory.

```
<Directory Id="TARGETDIR"
  Name="SourceDir">
  <Directory Id="ProgramFilesFolder">
    <Directory Id="MyProgramDir"
      Name="Install Practice" />
  </Directory>
</Directory>

<DirectoryRef Id="MyProgramDir">
  <Component ... />
</DirectoryRef>
```

By using a `DirectoryRef`, we're able to separate the markup that adds files to directories from the markup that defines the directories. You can also add a component directly inside a `Directory` element:

```
<Directory Id="TARGETDIR"
  Name="SourceDir">
  <Directory Id="ProgramFilesFolder">
    <Directory Id="MyProgramDir"
      Name="Install Practice">
      <Component ... />
    </Directory>
  </Directory>
</Directory>
```

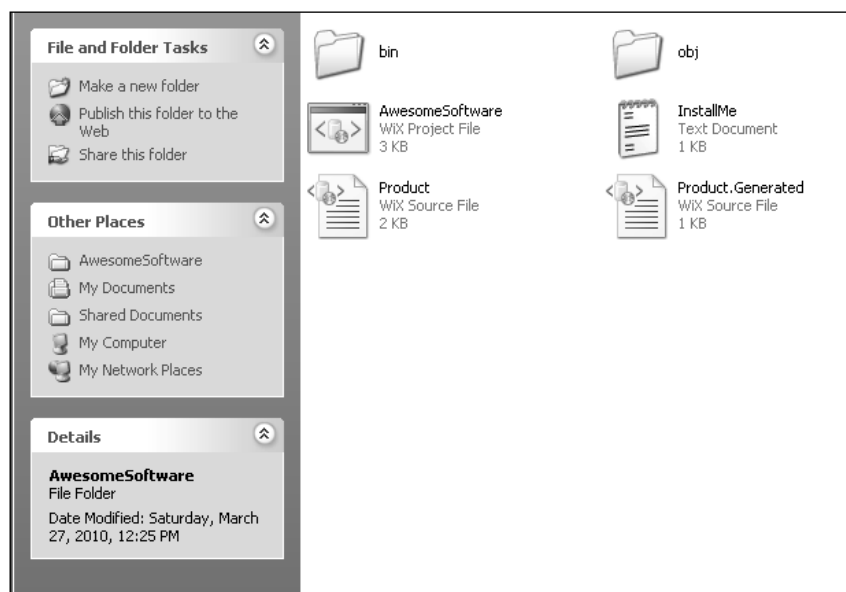
Placing components directly inside `Directory` elements is more straightforward, but it couples your directory structure more tightly to the files that you're installing. By using `DirectoryRef` elements, you're able to make the two pieces more modular and independent of one another.

## Components

Once you've mapped out the directories that you want to target or create during the installation, the next step is to copy files into them. Windows Installer expects every file to be wrapped up in a component before it's installed. It doesn't matter what type of file it is either. Each gets its own Component element.

Components, which always have a unique GUID, allow Windows to track every file that gets installed on the end user's computer. During an installation, this information is stored away in the Registry. This lets Windows find every piece of your product during an uninstallation so that your software can be completely removed. It also uses it to replace missing files during a "repair", which you can trigger by right-clicking on an MSI file and selecting **Repair**. You won't get an error by placing more than one file into a single component, but it is considered bad practice.

To really explain components, we'll need something to install. So, let's create a simple text file and add it to our project's directory. We'll call it `InstallMe.txt`. For our purposes, it doesn't really matter what's in the text file. We just need something for testing.



The `Component` element is used to uniquely identify each file that you plan to install. Each `Component` element gets a unique GUID via its `Guid` attribute. To create a GUID in Visual Studio, go to **Tools | Create GUID** and copy a new GUID using the Registry format. Be sure to make all of the letters uppercase. The `Id` attribute is up to you. It will serve as the primary key for the component in the MSI database, so each one must also be unique.

```
<Component Id="CMP_InstallMeTXT"
           Guid="E8A58B7B-F031-4548-9BDD-7A6796C8460D">

  <File Id="FILE_InstallMeTXT"
        Source="InstallMe.txt"
        KeyPath="yes" />
</Component>
```

Here, I've created a new component called `CMP_InstallMeTXT`. I've started it with `CMP_` to label it as a component. Although it isn't required, it helps to prefix components in this way so that it's always clear what sort of element it refers to.

The `File` element inside the component references the file that's going to be installed. Here, it's the `InstallMe.txt` file located in the current directory (which is the same directory as your WiX source file). You can specify a full or absolute path with the `Source` attribute.

You should always mark a `File` element as the `KeyPath` file and you should only ever include one `File` inside a `Component`. A `KeyPath` file will be replaced if it's missing when the user triggers a repair (Windows Installer documentation calls this *resiliency*). Placing more than one `File` element inside a single `Component` is, at least in most cases, not recommended. This is because only one file can be the `KeyPath`, so the other files wouldn't be covered by a repair. You would really only ever place more than one `File` in a `Component` if you *didn't* want the extra files to be resilient.

Once you've created your `File` and `Component` elements, you'll need to tell Windows Installer where they should be installed to. To do that, place them inside a `DirectoryRef` element that references one of your `Directory` elements—as shown in the following snippet:

```
<DirectoryRef Id="MyProgramDir">
  <Component Id="CMP_InstallMeTXT"
             Guid="E8A58B7B-F031-4548-9BDD-7A6796C8460D">

    <File Id="FILE_InstallMeTXT"
          Source="InstallMe.txt"
          KeyPath="yes" />
  </Component>
</DirectoryRef>
```



To add more files, simply create more `Component` and `File` elements. Of course, they don't all have to be installed to the same place. You might install some to the `MyProgramDir` folder that we're creating and others to a different folder. You always have to create a `Directory` element before you can place components in that directory. For example, you can't use the `AppDataFolder` property to place files in the Application Data directory until you've first added a `Directory` element for it.

## Files

As you've seen, the actual files inside components are declared with `File` elements. `File` elements can represent everything from simple text files to complex DLLs and executables. Remember, you should only place one file into each component. The following example would add a file called `SomeAssembly.dll` to the installation package:

```
<Component ... >
  <File Id="FILE_SomeAssemblyDLL"
        Name="Some Assembly.dll"
        Source="SomeAssembly.dll"
        KeyPath="yes" />
</Component>
```

A `File` element should always get the `Id`, `Source`, and `KeyPath` attributes. `Name` is optional and gives you a chance to set the name of the file to something user-friendly. Without it, the name will default to whatever it's called in the `Source` attribute. Notice that you should set the value to the file name plus file type extension. In earlier versions of WiX, you'd have to use a separate attribute if the name was longer than eight characters or the extension longer than three. Now, this attribute can handle longer names as well.

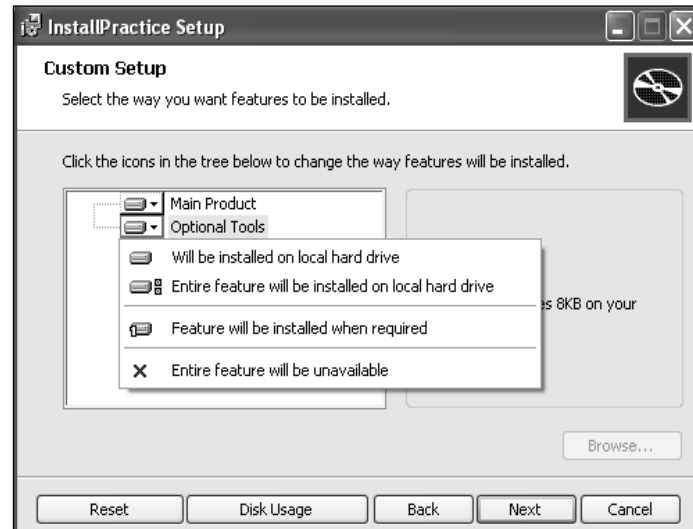
`Source` defines the path to the file during compilation. I've listed a relative path here, but you could also specify an absolute path. The `Id` attribute should be something unique, but you might consider starting it with `FILE` to make it clear that it refers to a `File` element. To mark a file as important (and that it should be replaced if it goes missing), set it as the `KeyPath` for the component. Since you should only ever place one file inside a component, in almost all cases that file should be the `KeyPath`.

There are quite a few optional attributes available for the `File` element. A few useful ones are:

- `Hidden`: Set to `yes` to have the file's `Hidden` flag set. The file won't be visible unless the user sets the directory's options to show hidden files.
- `ReadOnly`: Set to `yes` to have the file's `Read-only` flag set. The user will be able to read the file, but not modify it unless they change the file's properties.
- `Vital`: Set to `yes` to stop the installation if the file isn't installed successfully.

## Features

After you've defined your components and the directories that they'll be copied into, your next step is to define features. A **feature** is a group of components that the user can decide to install all at once. You'll often see these in an installation dialog as a list of modules, called a **feature tree**, where each is included or excluded from the installation. Here is an example of such a tree:



Every component must be included in a feature. Generally, you should group together components that rely on one another or that form a complete, self-sufficient unit. That way, if a feature is disabled, you won't have orphaned files (files that aren't being used) installed onto the computer. In some instances, if your product doesn't have any optional parts, you'll only want to create one feature.

If you've included a feature tree dialog, which we'll explain later in the book, like the one shown, the user can simply click a feature to exclude it. However, even without this, they can select features from the command line. The following command only installs the `Main Product` feature:

```
msiexec /i myInstaller.msi ADDLOCAL="MainProduct"
```

Here, we're using the `msiexec` program to launch an installer. The `/i` flag targets the MSI file to install. The `ADDLOCAL` property is set to the names of the features we want to include. If more than one, use commas to separate the names. To install all available features set `ADDLOCAL="ALL"`, as shown:

```
msiexec /i myInstaller.msi ADDLOCAL=ALL
```

To create a new feature in your WiX file, add a `Feature` element inside the `Product` element. The following example installs three components under the feature `Main Product`. Another feature called `Optional Tools` installs another component. Components are included in a feature with the `ComponentRef` element. The `Id` attribute of `ComponentRef` targets the `Id` attribute from the corresponding `Component` element.

```
<Feature Id="MainProduct"
        Title="Main Product"
        Level="1">
  <ComponentRef Id="CMP_MyAppEXE" />
  <ComponentRef Id="CMP_ReadMeTXT" />
  <ComponentRef Id="CMP_StartMenuShortcuts" />
</Feature>

<Feature Id="OptionalTools"
        Title="Optional Tools"
        Level="1">
  <ComponentRef Id="CMP_ToolsEXE" />
</Feature>
```

The `Feature` element's `Id` attribute uniquely identifies the feature and is what you'll reference when using the `ADDLOCAL` property on the command line. The `Title` attribute is used to set a user-friendly name that can be displayed on dialogs. Setting the `Feature` element's `Level` attribute to 1 means that feature will be included in the installation by default. The end user will still be able to remove it through the user interface or via the command line. If, on the other hand, `Level` is set to 0, that feature will be removed from the feature tree and the user won't be able to install it.

If you wanted to, you could create a more complex tree with features nested inside features. You could use this to create more categories for the elements in your product and give the user more options concerning what gets installed. You would want to make sure that all possible configurations function correctly. Windows Installer makes this somewhat manageable in that if a parent feature is excluded, its children features will be too. Here's an example of a more complex feature setup:

```
<Feature Id="MainProduct"
        Title="Main Product"
        Level="1">
  <ComponentRef Id="CMP_MyAppEXE" />
  <ComponentRef Id="CMP_StartMenuShortcuts" />

  <Feature Id="SubFeature1"
          Title="Documentation"
          Level="1">
```

```
<ComponentRef Id="CMP_ReadMeTXT" />
</Feature>
</Feature>

<Feature Id="OptionalTools"
        Title="Optional Tools"
        Level="1">
    <ComponentRef Id="CMP_ToolsEXE" />
</Feature>
```

Here, I've moved the `ReadMe.txt` file into its own feature called `Documentation` that's nested inside the `Main Product` feature. Disabling its parent feature (`Main Product`), will also disable it. However, you could enable `Main Product` and disable `Documentation`.

If you're going to use a feature tree, you have the ability to prevent the user from excluding a particular feature. Just set the `Absent` attribute to `disallow`. You might do this for the main part of your product where excluding it wouldn't make sense.

You might also consider adding the `Description` attribute, which can be set to a string that describes the feature. This could be displayed in your dialog alongside the feature tree, if you decide to use one. It would look something like this:

```
<Feature Id="ProductFeature"
        Title="Main Product"
        Description="Installs the Main Feature"
        Level="1">

    <ComponentRef Id="cmp_MyAppEXE" />
    <ComponentRef Id="cmp_ReadMeTXT" />
    <ComponentRef Id="cmp_StartMenuShortcuts" />
</Feature>
```

## Start Menu shortcuts

Having a working installer is good, but wouldn't it be nice to add some shortcuts for the application to the Start Menu? To do so, create a new component with a nested `Shortcut` element. Shortcuts are stored in their own table in the MSI database. First, add another `Directory` element that references the Start Menu via the built-in `ProgramMenuFolder` name:

```
<Directory Id="TARGETDIR"
        Name="SourceDir">
    <Directory Id="ProgramFilesFolder">
        <Directory Id="MyProgramDir"
```

```

        Name="Awesome Software" />
    </Directory>
    <Directory Id="ProgramMenuFolder">
        <Directory Id="MyShortcutsDir"
            Name="Awesome Software" />
    </Directory>
</Directory>

```

Here, I've added a reference to the Start Menu folder with the `ProgramMenuFolder` `Directory` element `Id`. I've then told the installer to create a new subfolder inside it called `Awesome Software`. Now, you can use a `DirectoryRef` element to reference your new shortcuts folder, as in the following code snippet:

```

<DirectoryRef Id="MyShortcutsDir">
    <Component Id="CMP_DocumentationShortcut"
        Guid="33741C82-30BF-41AF-8246-44A5DCFCF953">

        <Shortcut Id="DocumentationStartMenuShortcut"
            Name="Awesome Software Documentation"
            Description="Read Awesome Software Documentation"
            Target="[MyProgramDir]InstallMe.txt" />
    </Component>
</DirectoryRef>

```

Each `Shortcut` element has a unique identifier set with the `Id` attribute. The `Name` attribute defines the user-friendly name that gets displayed. `Description` is set to a string that describes the shortcut and will appear when the user moves their mouse over the shortcut link.

The `Target` attribute defines the path on the end user's machine to the actual file being linked to. For that reason, you'll often want to use properties that update as they're changed instead of hardcoded values. In the previous example, the main installation directory is referenced by placing the `Id` of its corresponding `Directory` element in square brackets, which is then followed by the name of the file. Even if the path of `MyProgramDir` changes, it will still lead us to the `InstallMe.txt` file.

Two things that should accompany a shortcut are a `RemoveFolder` element and a `RegistryValue` element. `RemoveFolder` ensures that the new Start Menu subdirectory will be removed during an uninstall. It uses an `Id` attribute to uniquely identify a row in the MSI `RemoveFile` table and an `On` attribute to specify when to remove the folder. You can set `On` to `install`, `uninstall`, or `both`. You can specify a `Directory` attribute as well to set to the `Id` of a `Directory` element to remove. Without one, though, the element will remove the directory defined by the parent `DirectoryRef` element.

Let's add the `RemoveFolder` and `RegistryValue` elements to our component:

```
<DirectoryRef Id="MyShortcutsDir">
  <Component Id="CMP_DocumentationShortcut"
    Guid="33741C82-30BF-41AF-8246-44A5DCFCF953">

    <Shortcut Id="DocumentationStartMenuShortcut"
      Name="Awesome Software Documentation"
      Description="Read Awesome Software Documentation"
      Target="[MyProgramDir]InstallMe.txt" />

    <RemoveFolder Id="RemoveMyShortcutsDir"
      On="uninstall" />

    <RegistryValue Root="HKCU"
      Key="Software\Microsoft\AwesomeSoftware"
      Name="installed"
      Type="integer"
      Value="1"
      KeyPath="yes" />

  </Component>
</DirectoryRef>
```

The `RegistryValue` element is needed simply because every component must have a `KeyPath` item. Shortcuts aren't allowed to be `KeyPath` items as they aren't technically files. By adding a `RegistryValue`, a new item is added to the Registry and this is marked as the `KeyPath`. The actual value itself serves no other purpose. We will cover writing to the Registry in more detail later.

There's actually another reason for using a `RegistryValue` as the `KeyPath`. The shortcut we're creating is being installed to a directory specific to the current user. Windows Installer requires that you always use a registry value as the `KeyPath` item when doing this in order to simplify uninstalling the product when multiple users have installed it.

Another type of shortcut to add is one that uninstalls the product. For this, add a second `Shortcut` element to the same component. This shortcut will be different in that it will have its `Target` set to the `msiexec.exe` program, which is located in the `System` folder. The following example uses the predefined `System64Folder` directory name because it will automatically map to either the 64-bit or 32-bit `System` folder, depending on the end user's operating system.

By setting `Target` to the path of an executable, you're telling Windows to launch that program when the user clicks the shortcut. The `msiexec` program can remove software by using the `/x` argument followed by the `ProductCode` of the product you want to uninstall. The `ProductCode` is the `Id` specified in the `Product` element.

---

```

<DirectoryRef Id="MyShortcutsDir">
  <Component Id="CMP_DocumentationShortcut"
    Guid="33741C82-30BF-41AF-8246-44A5DCFCF953">

    <Shortcut Id="DocumentationStartMenuShortcut"
      Name="Awesome Software Documentation"
      Description="Read Awesome Software Documentation"
      Target="[MyProgramDir]InstallMe.txt" />

    <Shortcut Id="UninstallShortcut"
      Name="Uninstall Awesome Software"
      Description=
        "Uninstalls Awesome Software and all of its components"
      Target="[System64Folder]msiexec.exe"
      Arguments="/x [ProductCode]" />

    <RemoveFolder Id="RemoveMyShortcutsDir"
      On="uninstall" />

    <RegistryValue Root="HKCU"
      Key="Software\Microsoft\AwesomeSoftware"
      Name="installed"
      Type="integer"
      Value="1"
      KeyPath="yes" />

  </Component>
</DirectoryRef>

```

Notice that we don't have to use the exact GUID from the `Product` element to get the `ProductCode`. We can reference it using the built-in property called `ProductCode` surrounded by square brackets. If you'd like to add an icon to your shortcut, first add an `Icon` element as another child to the `Product` element. Then, reference that icon with the `Icon` attribute on the `Shortcut` element.

```

<Icon Id="icon.ico" SourceFile="myIcon.ico"/>
<DirectoryRef ... >
  <Component ... >
    <Shortcut Id="DocumentationStartMenuShortcut"
      Name="Awesome Software Documentation"
      Description="Read Awesome Software Documentation"
      Target="[MyProgramDir]InstallMe.txt"
      Icon="icon.ico" />

    <RemoveFolder ... />
    <RegistryValue ... />
  </Component>
</DirectoryRef>

```

Be sure to add the new component that contains the shortcuts to one of your features:

```
<Feature Id="ProductFeature"
        Title="Main Product"
        Level="1">
  <ComponentRef Id="CMP_InstallMeTXT" />
  <ComponentRef Id="CMP_DocumentationShortcut" />
</Feature>
```

## Putting it all together

Now that you've seen the different elements used to author an MSI package, here is the entire .wxs file:

```
<?xml version="1.0" encoding="UTF-8"?>
<Wix xmlns="http://schemas.microsoft.com/wix/2006/wi">

  <Product Id="3E786878-358D-43AD-82D1-1435ADF9F6EA"
          Name="Awesome Software"
          Language="1033"
          Version="1.0.0.0"
          Manufacturer="Awesome Company"
          UpgradeCode="B414C827-8D81-4B4A-B3B6-338C06DE3A11">

    <Package InstallerVersion="301"
            Compressed="yes" />
    <Media Id="1"
          Cabinet="medial.cab"
          EmbedCab="yes" />

    <!--Directory structure-->
    <Directory Id="TARGETDIR"
              Name="SourceDir">
      <Directory Id="ProgramFilesFolder">
        <Directory Id="MyProgramDir"
                  Name="Awesome Software" />
        <Directory Id="ProgramMenuFolder">
          <Directory Id="MyShortcutsDir"
                    Name="Awesome Software" />
        </Directory>
      </Directory>
    </Directory>

    <!--Components-->
```



```
<DirectoryRef Id="MyProgramDir">
  <Component Id="CMP_InstallMeTXT"
    Guid="E8A58B7B-F031-4548-9BDD-7A6796C8460D">
    <File Id="FILE_InstallMeTXT"
      Source="InstallMe.txt"
      KeyPath="yes" />
    </Component>
  </DirectoryRef>

<!--Start Menu Shortcuts-->
<DirectoryRef Id="MyShortcutsDir">
  <Component Id="CMP_DocumentationShortcut"
    Guid="33741C82-30BF-41AF-8246-44A5DCFCF953">

    <Shortcut Id="DocumentationStartMenuShortcut"
      Name="Awesome Software Documentation"
      Description="Read Awesome Software Documentation"
      Target="[MyProgramDir]InstallMe.txt" />

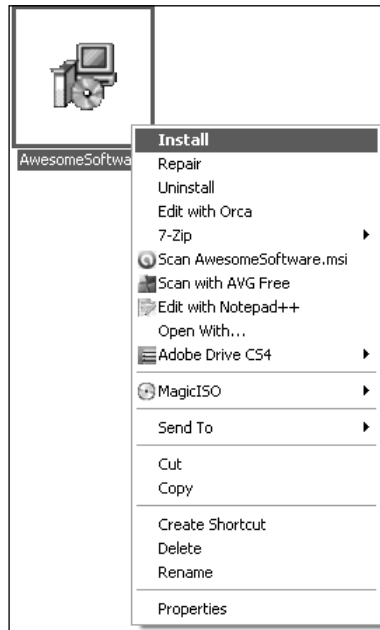
    <Shortcut Id="UninstallShortcut"
      Name="Uninstall InstallPractice"
      Description="Uninstalls Awesome Software"
      Target="[System64Folder]msiexec.exe"
      Arguments="/x [ProductCode]" />

    <RemoveFolder Id="RemoveMyShortcutsDir"
      On="uninstall" />

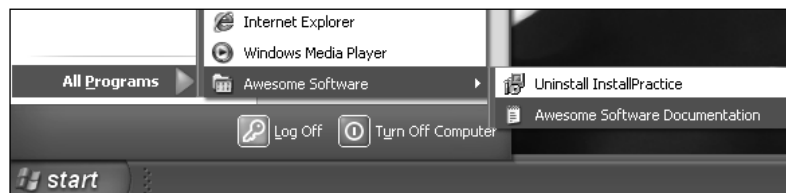
    <RegistryValue Root="HKCU"
      Key="Software\Microsoft\AwesomeSoftware"
      Name="installed"
      Type="integer"
      Value="1"
      KeyPath="yes" />
    </Component>
  </DirectoryRef>

<!--Features-->
<Feature Id="ProductFeature"
  Title="Main Product"
  Level="1">
  <ComponentRef Id="CMP_InstallMeTXT" />
  <ComponentRef Id="CMP_DocumentationShortcut" />
</Feature>
</Product>
</Wix>
```

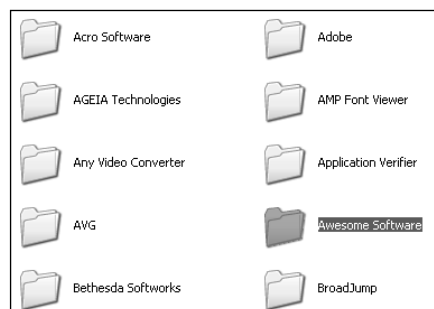
Compile the project in Visual Studio and you should get a new MSI file.



You can double-click it or right-click and select **Install** to install the software. Doing so should create a subfolder for your program in the Start Menu.



You should also find a new folder under Program Files.



To uninstall the software, you have several options:

- Use the uninstall shortcut from the Start Menu
- Right-click on the MSI file and select **Uninstall**
- Uninstall it from Add/Remove Programs
- From a command prompt, navigate to the directory where the MSI file is and use the command:

```
msiexec /x AwesomeSoftware.msi
```

## Adding a User Interface

Although you'll eventually want to add your own dialogs to gather information from the user that's important for your own application, you may want to use one of WiX's built-in dialog sequences in the meantime. All of them are stored in an assembly called `WixUIExtension.dll`. You can add a reference to this file with Visual Studio's **Add a Reference** screen. The file exists in WiX's Program Files folder. You may have to navigate to `C:\Program Files\Windows Installer XML v3\bin`.

Once you've added the new reference, add the following line to your WiX source file. It doesn't matter exactly where, as long as it's a child to the `Product` element:

```
<UIRef Id="WixUI_Minimal" />
```

This will insert the `Minimal` dialog set into your installation sequence. It shows a single dialog screen containing a license agreement and **Install** button. Feel free to try any of the other dialog sets. Just replace `WixUI_Minimal`, with one of the other names in the `UIRef` element. `WixUI_Advanced` and `WixUI_InstallDir` require some further setup to really work properly. You can try out the following attributes:

- `WixUI_Advanced`
- `WixUI_FeatureTree`
- `WixUI_InstallDir`
- `WixUI_Mondo`

We will explore these standard dialogs in more detail later and also explain how to create your own.

## Viewing the MSI database

I mentioned before that an MSI file is really a sort of relational database. WiX does all the work of creating tables, inserting rows, and matching up keys in this database. However, as we progress through the rest of the book, I encourage you to explore how it looks behind the scenes. For example, we discussed the `File` and `Component` elements. Sure enough, there are two tables called `File` and `Component` in the MSI that contain the definitions you've set with your XML markup. To get inside the installer, you'll need a tool called Orca.

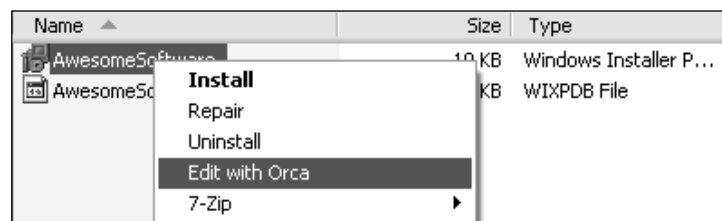
## Orca.exe

Once you've compiled your project in Visual Studio, you'll have a working MSI package that can be installed by double-clicking it. If you'd like to see the database inside, install the MSI viewer, **Orca.exe**. Orca is provided as part of the Windows SDK and despite the icon of a whale on the shortcut, stands for **One Really Cool App**. You can find versions of the SDK at Microsoft's Windows Development Center website:

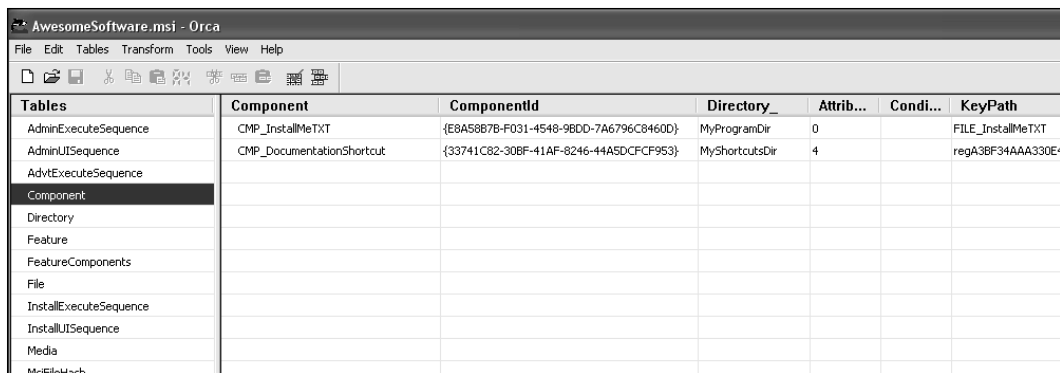
<http://msdn.microsoft.com/en-us/windows/bb980924.aspx>

After you've installed the SDK (specifically, the .NET tools that are included), you can find the installer for Orca—`Orca.msi`—in the Microsoft SDKs folder in Program Files. On my machine, it can be found in  
`C:\Program Files\Microsoft SDKs\Windows\v7.0\Bin.`

Install Orca and then right-click on your MSI file and select **Edit with Orca**.



Orca lets you view the database structure of your installer. This can be a big help in troubleshooting problems or just to get a better idea about how different elements work together. Following is a screenshot of the `Component` database:



Tables	Component	ComponentId	Directory	Attrib...	Condi...	KeyPath
AdminExecuteSequence	CMP_InstallMeTXT	{E8A58B7B-F031-4548-9BDD-7A6796C8460D}	MyProgramDir	0		FILE_InstallMeTXT
AdminUISequence	CMP_DocumentationShortcut	{33741C82-30BF-41AF-8246-44A5DCFCF953}	MyShortcutsDir	4		regA3BF34AAA330E4
AdvtExecuteSequence						
Component						
Directory						
Feature						
FeatureComponents						
File						
InstallExecuteSequence						
InstallUISequence						
Media						
ModuleHash						

If you wanted to, you could edit your MSI package directly with Orca. This is helpful when learning or trying out different concepts. You'll need to know exactly which tables and rows to modify. Sometimes, though, you'll be able to just change a single value and check its effect.

## Turning logging on during installation

If you get into trouble with your installer, it may help to run it with logging turned on. To do so, install your package from a command prompt using `msiexec` with the arguments `/l*v` and the name of a file to write the log to. For example, if you had an installer called `myInstaller.msi`, you could use this command to write a log during the installation to a file called `myLog.txt`:

```
msiexec /i myInstaller.msi /l*v myLog.txt
```

Every event that occurs during installation will be recorded here. It works for uninstalls too. Simply use the `/x` argument instead of `/i`. The log can be pretty helpful, but also very verbose. If your installer fails midway through, you might try searching the log for the text `return value 3`. This indicates that an action returned a status of failure. Often, you'll also see a specific MSI error code. You can find its meaning by searching for that number in the **MSI SDK Documentation** help file that comes with WiX.

## Other resources

If you have specific questions about WiX, you'll find additional resources at the following websites:

- WiX users Mailing List:

[http://sourceforge.net/mailarchive/forum.php?forum\\_name=wix-users](http://sourceforge.net/mailarchive/forum.php?forum_name=wix-users)

- Microsoft Windows Installer documentation:

[http://msdn.microsoft.com/en-us/library/cc185688\(VS.85\).aspx](http://msdn.microsoft.com/en-us/library/cc185688(VS.85).aspx)

## Summary

In this chapter, we discussed downloading the WiX toolset and its various features. Creating a simple MSI package is relatively easy. There are only a handful of XML elements needed to get started. As we explore more complex setups, you'll be introduced to elements that are more specialized.

Throughout the rest of this book, I'll make references to the structure of the MSI database. Orca is an excellent tool for seeing this structure yourself. Although this book focuses on WiX and not the underlying Windows Installer technology, it helps sometimes to see how the mechanics of it work. You may find it useful to consult Microsoft's MSI documentation too, which can be found online or in a help file provided by WiX, to get a deeper understanding of the properties and constructs we will discuss.

## Where to buy this book

You can buy WiX: A Developer's Guide to Windows Installer XML from the Packt Publishing website: <https://www.packtpub.com/wix-a-developers-guide-to-windows-installer-xml/book>.

Free shipping to the US, UK, Europe and selected Asian countries. For more information, please read our [shipping policy](#).

Alternatively, you can buy the book from Amazon, BN.com, Computer Manuals and most internet book retailers.



[www.PacktPub.com](http://www.PacktPub.com)

**For More Information:**

[www.PacktPub.com/wix-a-developers-guide-to-windows-installer-xml/book](https://www.PacktPub.com/wix-a-developers-guide-to-windows-installer-xml/book)