Visual Studio 2010 Package Development Fundamentals

VS 2010 Package Development

Table of Contents

Whom This Book Is For

This book is for those .NET developers, who work with Visual Studio 2010 and are interested in enhancing their development environment with powerful extensions. If you recognize yourself as you're in one or more of the roles listed below, this book is intended to be a primary resource for you.

- You're a professional software developer. Creating Visual Studio extensions adds measurable (business) value to your or your company's everyday activities.
- You're a developer or an architect with special projects. You feel the project team performance can be enhanced by custom tools integrated into Visual Studio and tailored to the project's special needs.
- You're a member of a development team with tools that right now run out of Visual Studio. Integrating these tools into the Visual Studio IDE would improve your team's performance and developer experience and make team members happy.
- You are a developer of an ISV company or a developer wanting to become an ISV. You have concrete business drivers to create products integrated into Visual Studio and find the extensibility market for them.
- You are a student or a hobbyist, and you are looking for exciting things. Extending Visual Studio is one of them.

There are many ways to extend Visual Studio such as customization, macros, add-ins, and more, but without doubt the most powerful is package development. This book intends to treat the fundamentals of Visual Studio package development in a didactic way to provide you the best learning curve ever. After reading it you'll be familiar with the extensibility architecture of Visual Studio and with the most important concepts of package development. You'll understand how packages are connected to the functionality offered by Visual Studio and how you can turn your ideas into extensibility components.

Even if you have never created any kind of Visual Studio extension, you can immediately start package development with the help of this book. If you have some experience with writing Visual Studio add-ins, you will discover that packages offer you more power than add-ins ever did.

If you have already developed Visual Studio packages, this book provides you a comprehensive

overview of the fundamental concepts; you can also give it to your fellows or team members as a self-paced study guide.

What You Need to Use This Book

The books title tells you it's about Visual Studio 2010. Although the essential package development concepts have not changed enormously since Visual Studio 2008, the samples you are going to build run together with Visual Studio 2010. So you need to install either the Professional or the Ultimate edition of Visual Studio 2010. If you have not bought Visual Studio 2010 yet, you can get information about evaluation copies from Microsoft's official home page.

Visual Studio 2010 has free Express editions, but these do not support extensibility options, so you cannot use them with this book.

You cannot develop packages with Visual Studio out of the box. You need to install the Visual Studio 2010 SDK that can be downloaded from the Visual Studio Extensibility Development Center (http://msdn.com/vsx). The download and installation process takes about 3-5 minutes.

When you have Visual Studio 2010 and VS 2010 SDK installed on your computer, you're ready to start.

Visual Studio 2010 runs on several operating systems. The samples I provide in this book were written on a computer with Windows 7 x64 Ultimate operating system and Visual Studio 2010 Ultimate.

What This Book Covers

As the title of this book suggests, it is dedicated to Visual Studio 2010 Package development. The APIs available to develop Visual Studio extensions provide you several hundred of objects and interfaces with over thousand methods, and almost hundred extensibility points.
Instead of treating all of them, this book intends to give you a very detailed overview of the package development fundamentals. The book contains seven chapters suggested to read from the first to the last without skipping any of them:

- Chapter 1 talks about the architecture of Visual Studio from the extensibility point of view and explains you what options you have to add your own functionality to the Integrated Development Environment (IDE).
- Chapter 2 explains the basic concepts of Visual Studio Packages. In this chapter you create your first package and dive into the source code to discover how these concepts are represented. You will also learn how packages are built, deployed and how you can debug them.
- Chapter 3 provides an overview of the command handling architecture. You'll learn how package commands are merged into the IDE and how you can design the menu and toolbar items for your commands.
- Chapter 4 treats the window management architecture of Visual Studio and gives you a comprehensive overview of designing and implementing tool windows. You are going to create

a sample and improve it step-by-step while you discover the concepts, options and programming patterns of tool windows.

- Chapter 5 talks about services, one of the most important concepts in extending Visual Studio. You'll learn the service architecture of the IDE. Besides consuming the common IDE services you are going to create your own services to be consumed by other packages.
- Chapter 6 gives you an overview about the Development Tools Extensibility (DTE) object model that contains several dozen automation objects to access Visual Studio services. The DTE API is a part of Visual Studio — and not the part of Visual Studio 2010 SDK — from the first version of Visual Studio .NET (released in April, 2002). This chapter also treats how can you extend the automation model with your package-specific objects.
- Chapter 7 explains the concept of options pages to allow integrating your own package-specific configuration information into the Options dialog.

VS 2010 Package Development

Chapter 1 : Fundamentals

I have spent a lot of time with preparing a book about Visual Studio Extensibility, focusing on Visual Studio Package Development. I have made proposals for several book publishers, but I did not manage to get a contract, most of them found such a book is not profitable. I decided to share the four chapters of the book that I've already written. They are the followings:

- Chapter 1: Visual Studio Packages
- Chapter 2: Commands, Menus and Toolbars
- Chapter 3: Window Management and Tool Windows
- Chapter 4: Services

I hope, you will find these chapters useful.

The majority of Visual Studio's functions you use in your everyday work (such as programming languages, editors, designers and debuggers) are provided by Visual Studio Integration Packages, or shortly by packages. Some call them VSIP packages but the VSIP acronym is overloaded: while the first two letters means "Visual Studio" the last two may mean either "Integration Package" or "Industry Partner" and unfortunately both terms are frequently used. To avoid ambiguity hereinafter you'll meet the term package or VSPackage.

Developing packages means you can extend Visual Studio on the same way as its developer team at Microsoft. Adding new functions through packages is actually programming a new part of Visual Studio just like if you were the member of the team. You can use the full power and integrate any functionality you miss from the IDE!

In this chapter you will create a very simple package called FirstLook to get a feeling that first steps are easy. Then you'll learn the basic concepts behind packages and dive into the FirstLook project's structure and source code to have a closer look at the implementation of those concepts. At the end of this chapter you'll be familiar with the following:

- Building a package with the VSPackage Wizard
- The idea of a package
- The on-demand package loading mechanism and the idea of package siting
- Registration of packages and information stored in the registry
- The Visual Studio Experimental Hive and using it for debugging packages
- The package build process
- Deploying VSPackages

VS 2010 Package Development

This chapter will not teach you how to build a specific functionality into a package and does not cover the API used to develop packages. The focus is on understanding the concepts, architectural considerations behind VSPackages to let you take a look behind the scenes and get acquainted with package mechanisms. These concepts will be very useful when you are about creating your own packages.

Building a Simple Package

There are a few important concepts to understand if you want to develop a package. In order treat them in the right context you build a very simple functional package to touch the surface of those concepts and then jump into the details.

A Visual Studio package is a class library containing the types responsible for the package infrastructure and functionality. In order Visual Studio recognize the compiled class library as package, encapsulated types should have specific metadata information and some additional steps are required after compilation. So, even if you could start building a package from an empty class library, it is much easier to use the VSPackage Wizard installed with Visual Studio SDK.

Start a new project with the File|New|Project command. The IDE displays the **New File** dialog to select the desired project type. You can find the Visual Studio Integration Package project type under the Other Project Types category in the Extensibility folder as Figure 1 illustrates.

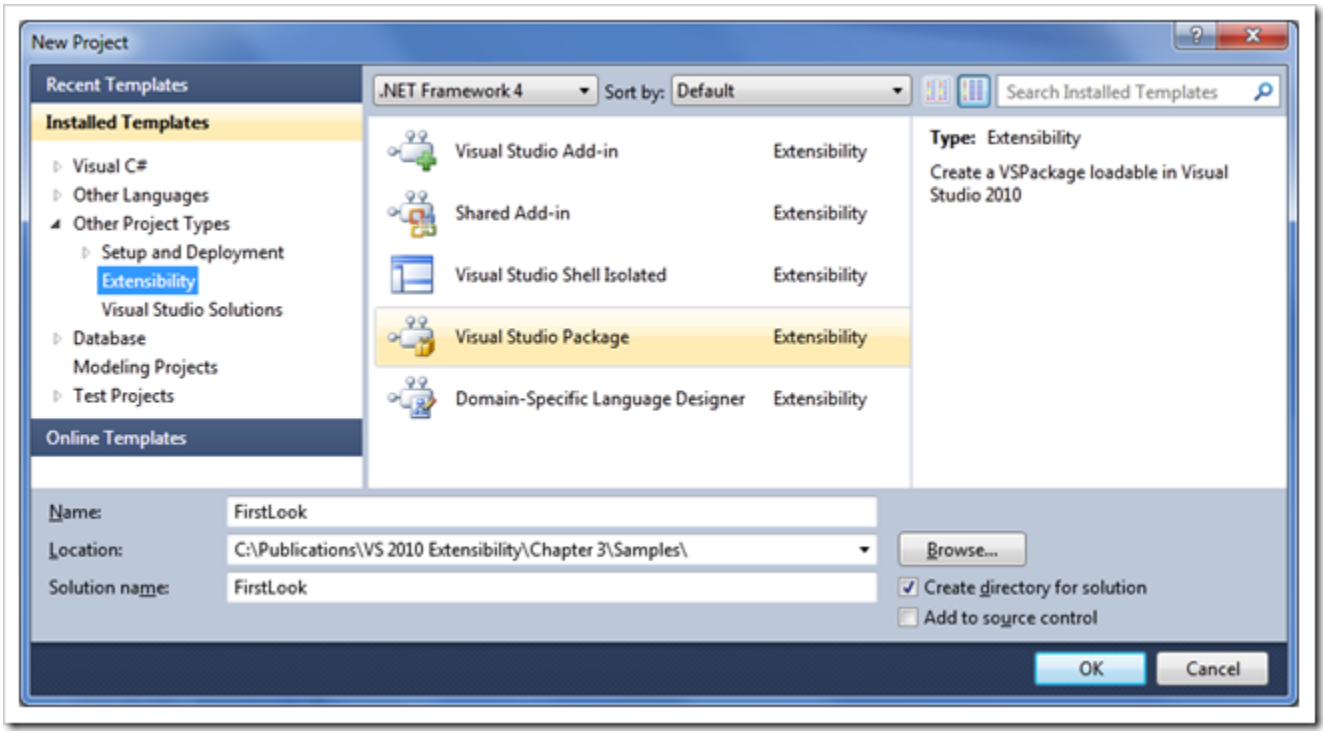


Figure 1: *The New Project dialog with the Extensibility project types*

VS 2010 Package Development

Should you not find this project type or many other project types in the Extensibility folder means that Visual Studio SDK is not — or not properly — installed on your machine. Install it according to the setup notes in order to go on with building the package.

Give the **FirstLook** name to the package in order to be able to follow code details later in this chapter. Clicking the OK button starts the Visual Studio Integration Package Wizard (henceforward VSPackage Wizard is used, it is shorter) which welcomes you with the dialog in Figure 2.

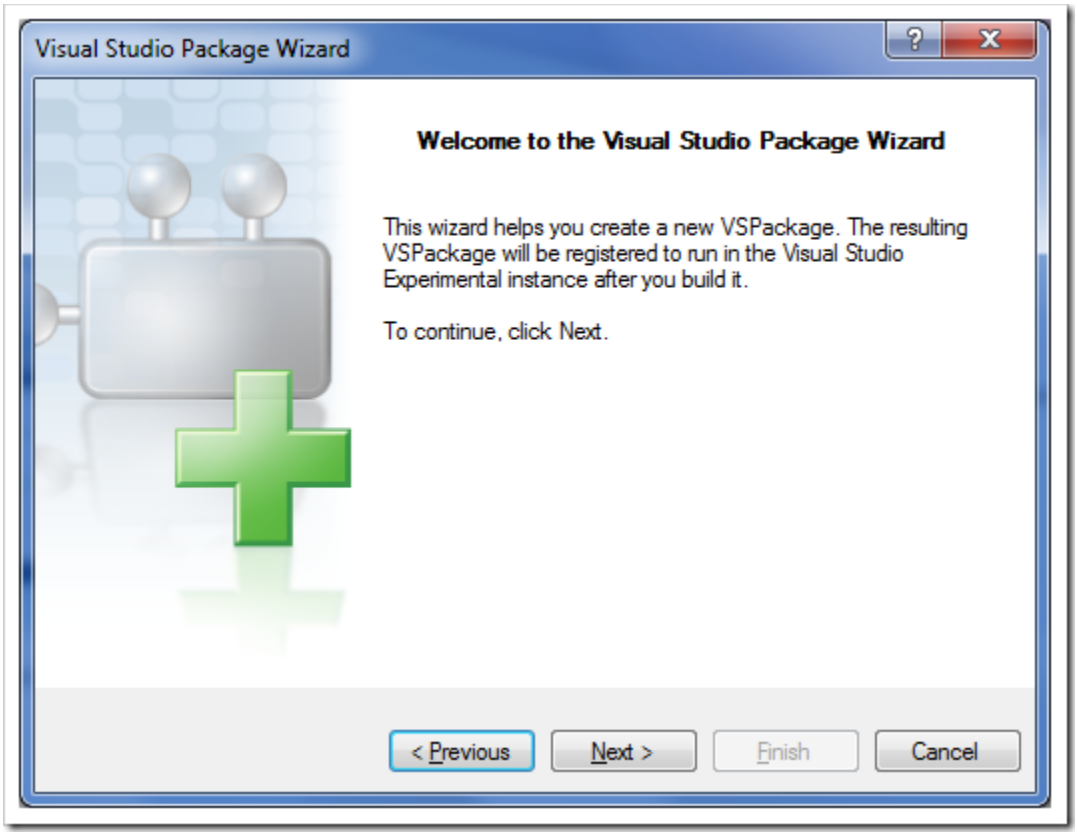


Figure 2: *The Welcome page of the VSPackage Wizard*

Click the Next button to go on specifying the package parameters and you get to the Select a Programming Language page of the wizard as Figure 3 shows.

VS 2010 Package Development

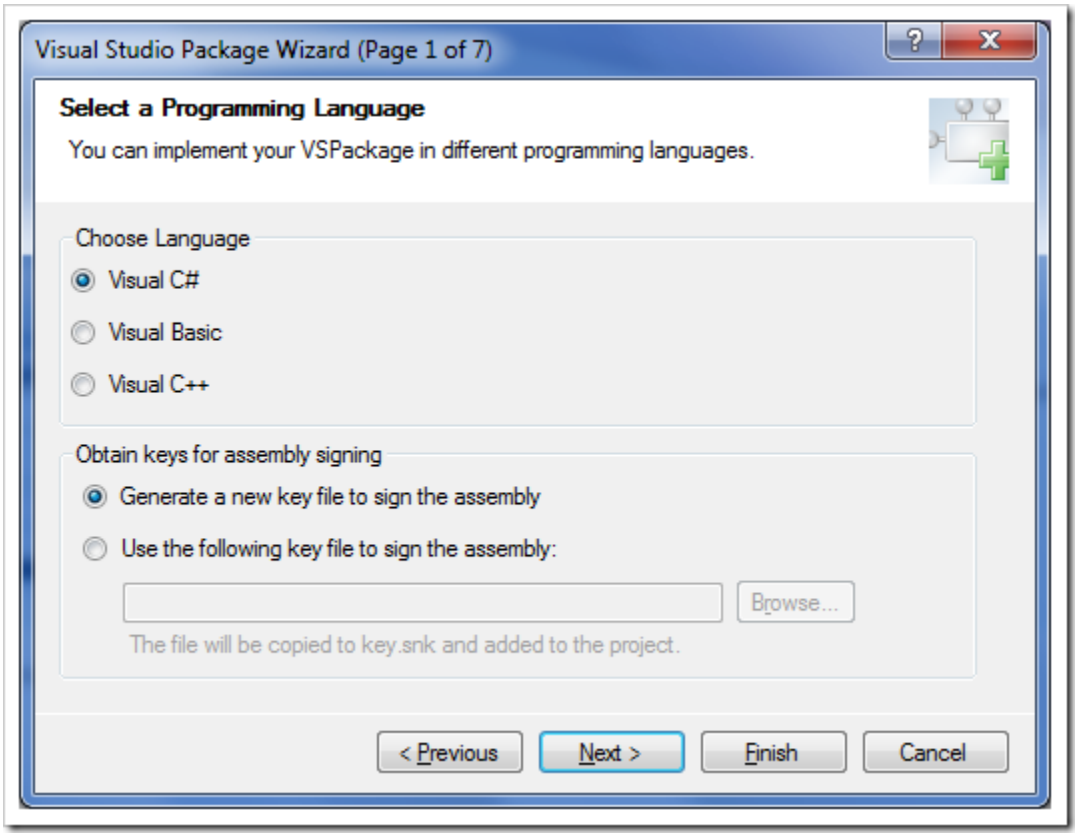


Figure 3: *VSPackage Wizard lets you select the programming language*

As it was mentioned earlier you create the code in C#. Packages are strongly named assemblies, so you need to sign the class library assembly with a key. For this project the wizard creates the signing key. Click Next and you get to the Basic VSPackage Information page as Figure 4 shows.

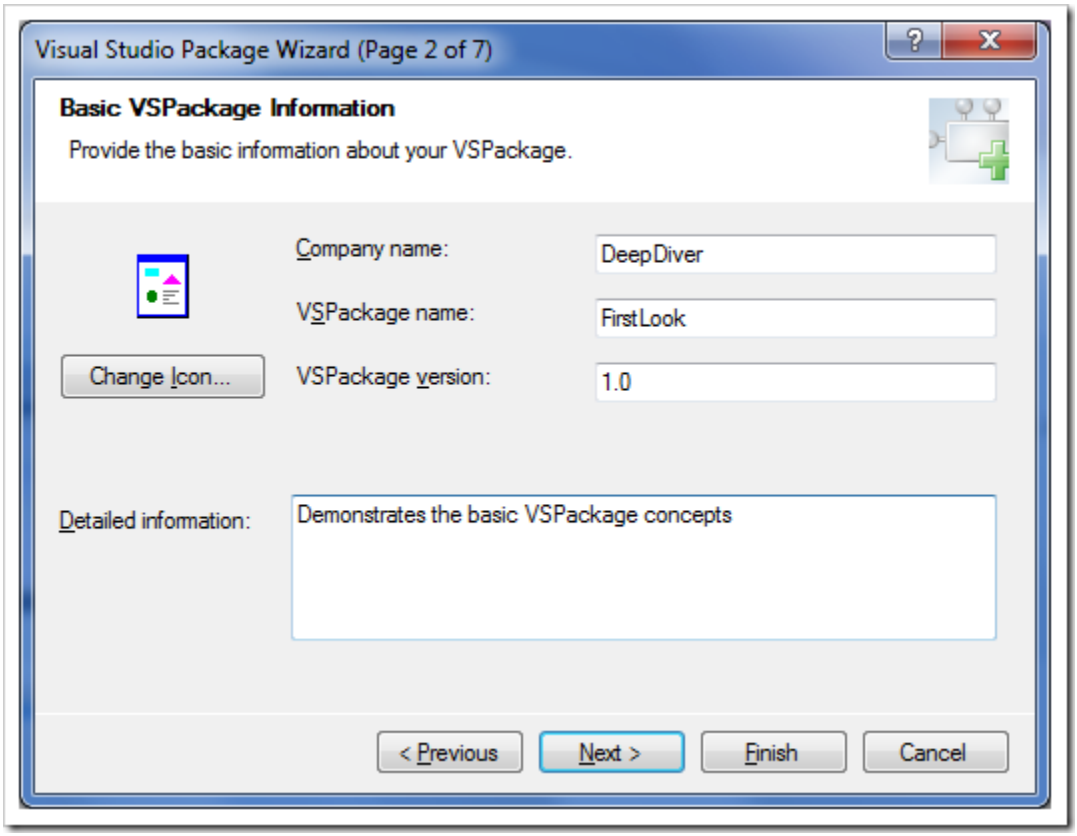


Figure 4: *The wizard asks for the basic package information*

The information you provide here will be used in the source code generated for the package and will be displayed in the Visual Studio About dialog. The Company name will be used in the namespace of generated types as well as VSPackage name which also names the class representing the package in code. VSPackage version is additional information to give a way for distinguishing separate package releases.

Text typed in the Detailed information field will be displayed in the About dialog and can supply the user with more information about what the package does.

When you click the Next button the wizard moves you to the VSPackage Options page — as can be seen in Figure 5 — to set a few more code generation options.

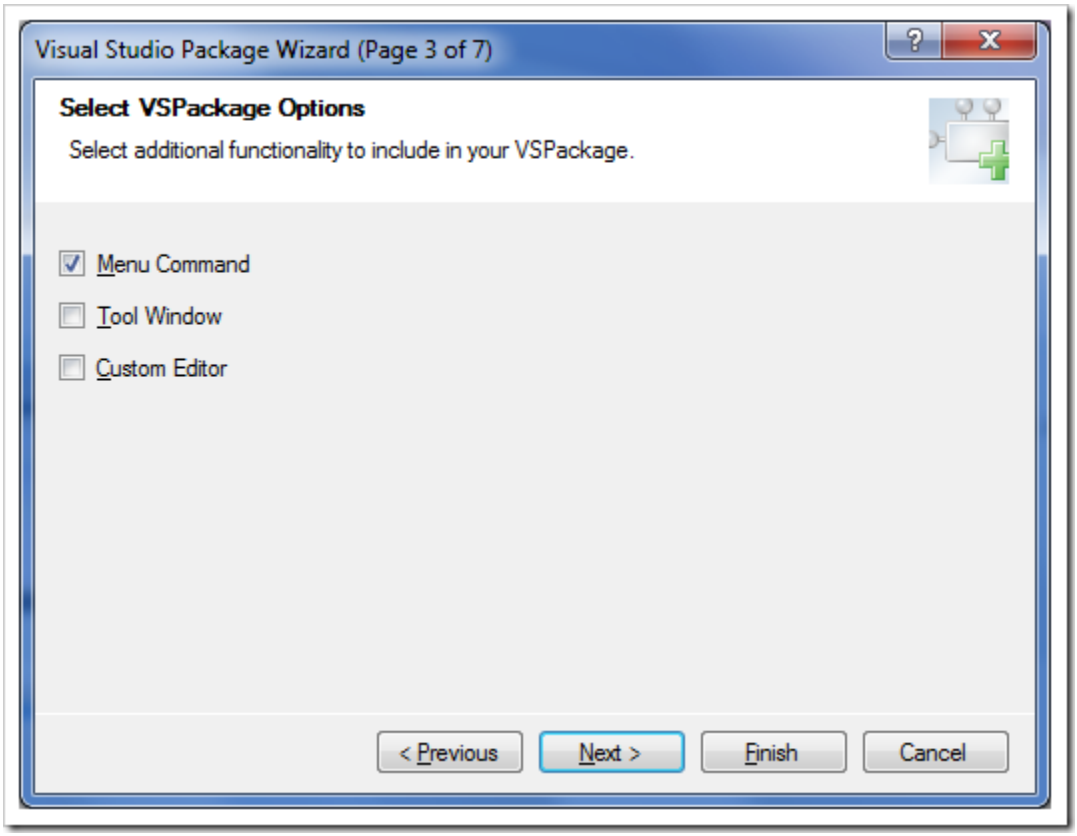


Figure 5: *You can select a few code generation options*

In this sample you are going to create only a menu command that pops up a message on the screen, so set the Menu Command option. Should you select the other two options the VSPackage Wizard would create some more code for a simple tool window or a rich text editor. Please, let those options remain unchecked. With the Next button the wizard goes to the page where we can specify a few details about the menu command to create. This page is shown in Figure 6.

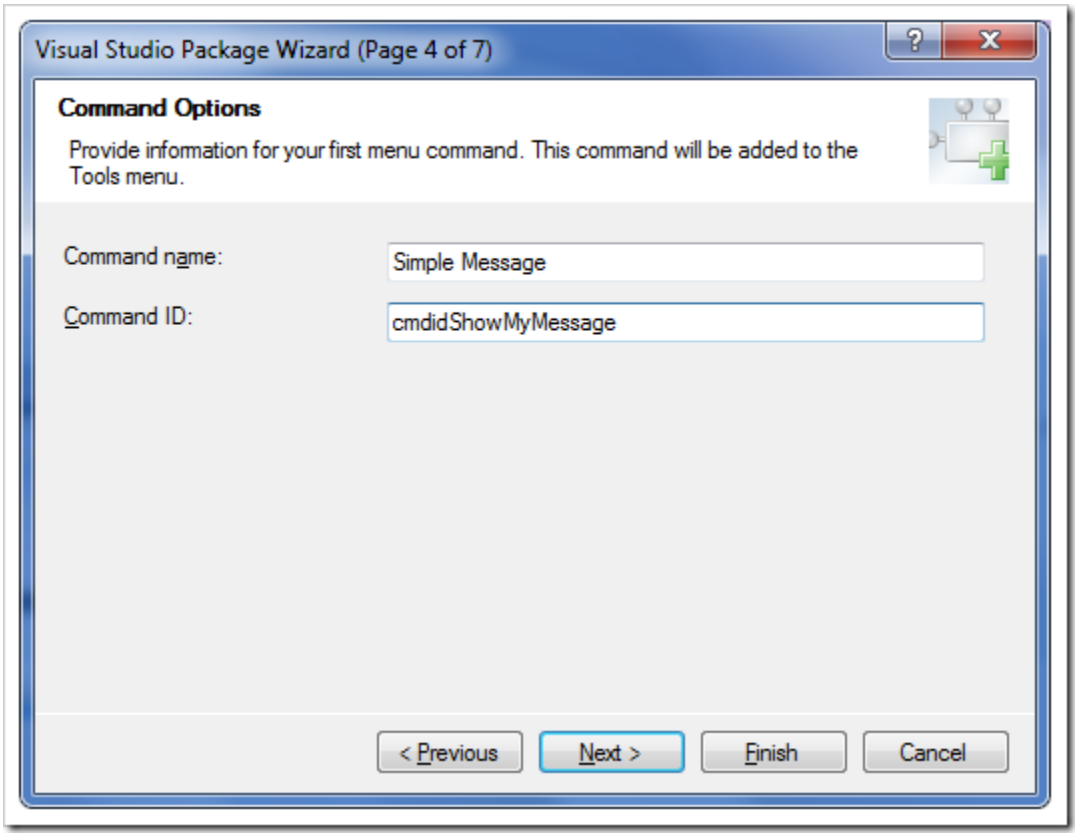


Figure 6: *Command options are specified here*

The command will be added to the Tools menu of Visual Studio, in Command name you can specify the text to be displayed for the menu item. According to the internal command handling architecture each command has an identifier. The Command ID field supplies a name for this identifier and VSPackage Wizard will generate an ID value behind this name. With clicking Next the wizard moves to the Test Project Options page as shown in Figure 7.

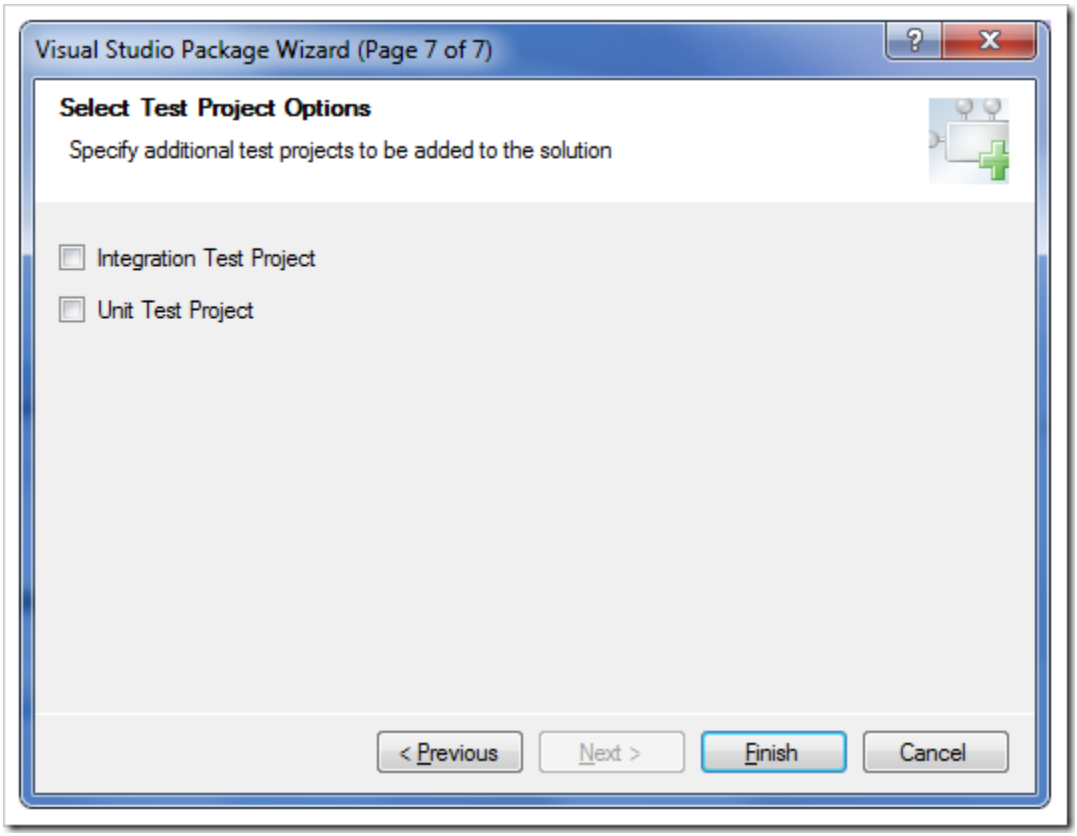


Figure 7: *VSPackage Wizard asks for test project options*

The wizard can create unit test for the package which check if its functional units work properly. The wizard also can create an integration test project for you, in which packages are tested within the context of a Visual Studio instance.

For the sake of simplicity here you do not create any tests, so clear the options — by default both are checked. Now you have set all parameters the wizard uses to generate the package project, click on the Finish button.

In a few seconds the wizard generates the package project ready to build and run. Taste the pudding you have just cooked! With the Build ð Rebuild Solution function you can compile the package and carry out all other steps required to run the package with Visual Studio. So rebuild the project and start with Ctrl + F5 (Debug|Start Without Debugging).

You might be surprised as a new instance of Visual Studio is started with “Experimental Instance” in its window caption.

Note: If this is the first time you have started the Experimental Instance, the Choose Default Environment Settings dialog appears just like when you first time launch Visual Studio after installation.

VS 2010 Package Development

This is an instance of Visual Studio that hosts the **FirstLook** package — later you're going to learn the concept behind. The menu command implemented by this freshly generated package can be seen in the Tools menu as Figure 8 shows it.

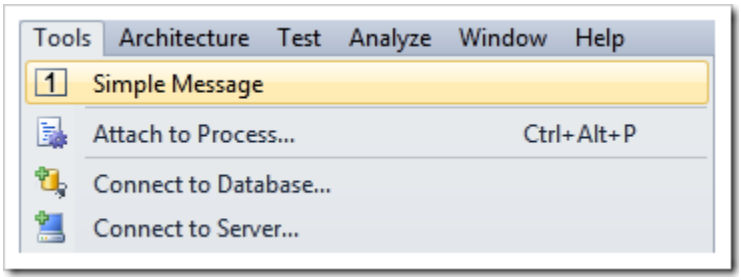


Figure 8: *The menu command item appears in the Tools menu*

When you click the Simple Message command, it pops up a message box telling you it was displayed from the **FirstLook** package.

The package also registered some branding information that can be seen in the Help|About dialog as Figure 9 shows.

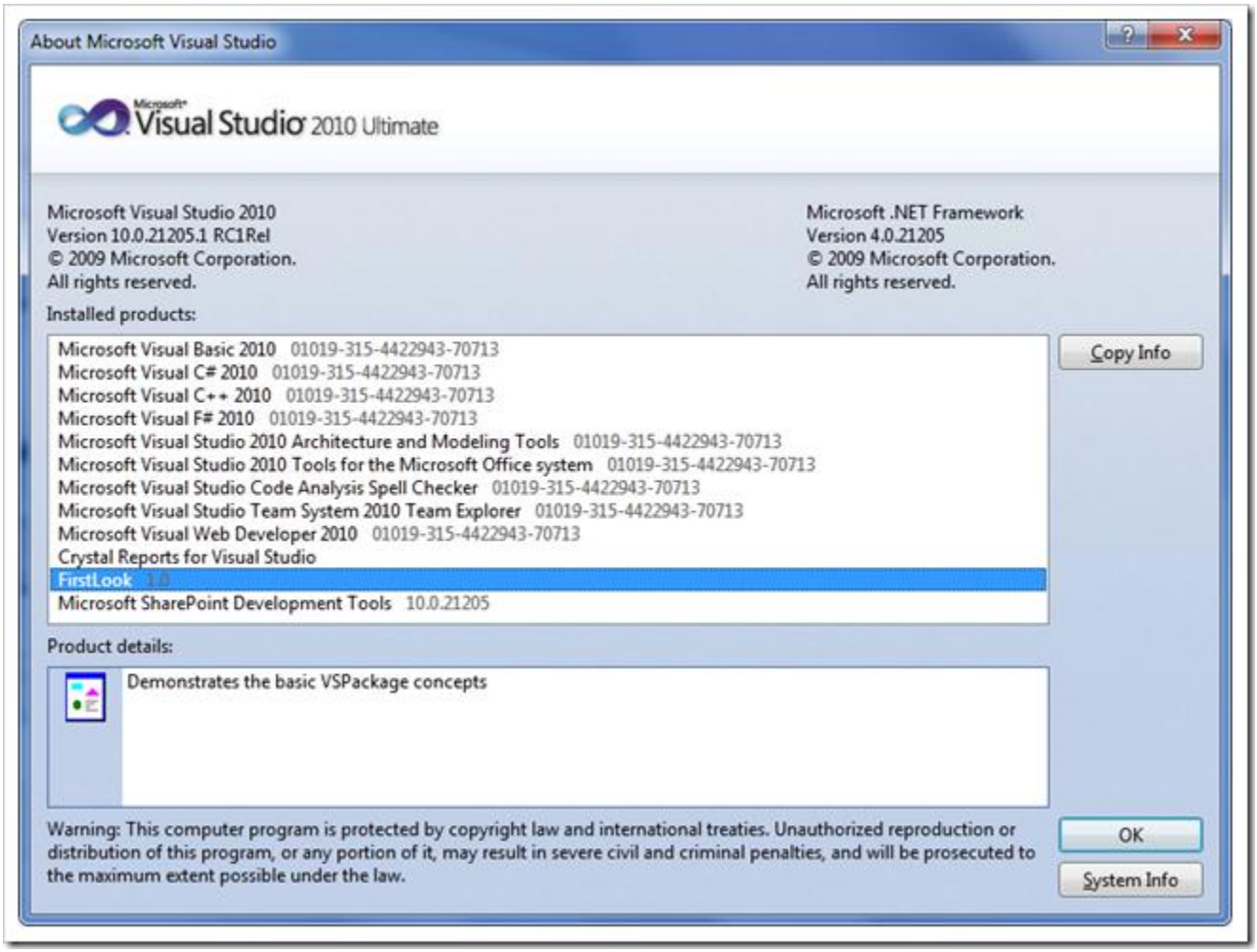


Figure 9: *Branding information of the **FirstLook** package*

Nothing call tell more about your VSPackage than its source code. But before deep-diving into it, let's treat important concepts about packages.

Concepts behind Visual Studio Packages

A VSPackage is the principal architectural unit of Visual Studio. As you already know, Visual Studio IDE itself is the Shell hosting a set of VSPackages working together with each other and with the Shell. The basic responsibility of a package is to provide a common container for extensibility objects. So, a VSPackage is a software module that is a unit not only from architectural point of view but also from deployment, security and licensing aspects.

Developers — including the developers of Visual Studio — create VSPackages to provide some extension to the VS IDE and group them into modules according to their functionality. These extensions can be:

- **Services**. These are software objects that offer functionality for other objects in the same package or even for other packages. For example, the C# Language Service (as its name also tells) is a service.
- **User interface elements**. Menus, toolbars, and windows can be used by developers to initiate some actions in the user interface, interact with, display messages, information and figures, and so on.
- **Editors**. During development you write and modify program text to create applications. This task is the responsibility of an editor. Visual Studio 2010 has its own core text editor and you can extend it or even create your own editors.
- **Designers**. Application creation is not just simply a text-typing activity. There are many visual tools known as designers that allow an alternative representation and design of modules, components, parts or even full applications. Well-known examples are the Windows Forms designer, The WPF Forms designer, the ASP.NET page designer or the Data Table designer.
- **Projects**. When developing applications you generally work with a large number of files. A project is an organization of source files and resources. A project not just simply stores these files but also defines operations with them, allows building, debugging and deploying the products created from source files.

It is natural in .NET that you can divide your functional units into separate assemblies where consumer assemblies will reference other service assemblies. The same principle works for VSPackages: an assembly containing a package can reference other assemblies that may contain not just helper types but even extensibility object types.

An assembly — as the smallest physical deployment unit in .NET — may contain more than one VSPackage. Although generally only one package is encapsulated in an assembly, you may have many reasons to group more packages in one assembly — including deployment considerations.

The Package Load Key

The previous versions of Visual Studio checked the package before loading them into the process space of the Shell. Any VSPackage should have been "sealed" with a so-called Package Load Key (PLK) and this key was verified during package load time. PLK was not a digital signature or a full hash, because it was calculated from a few information fields in the package. PLK could have been requested from Microsoft through a webpage: the developer specified a few well-defined attributes of the package and some logic calculated the PLK value. This value was embedded as a resource into the assembly representing the package.

Every time the package was loaded, the Shell checked the PLK against the package attributes it had been created from. Should have been this check fail, the Shell would have refused loading the package. This PLK mechanism did not mean that a developer had to request a new PLK for each package modification. While no basic information the PLK had been generated from was changed, the package continued to load.

Although this concept had been seemed useful, in the real life in had no real advantage. To be honest, in most of the cases it was the root cause of deployment issues, in many scenarios it raised more problems than it solved.

In the new Visual Studio 2010 the Shell does not use the Package Load Key to check packages before loading them into the memory.

On-demand Loading of Packages

You can imagine that complex packages like the C#, VB, F# or C++ languages with all of their "accessories" could consume many system resources by terms of memory and CPU. If you do not use them they do not press the CPU, but they might use the memory if they sit within the Visual Studio process space. If you create a project using F# you actually do not need services belonging to other languages, so why to load them into the memory at all?

The architects of Visual Studio implemented the package load mechanism so that packages are loaded into the memory at the first time when an event requiring the presence of the package is raised. These events can be one of the followings:

- **Command activation**. The user (or some running code) activates a menu or toolbar command (or even a command that cannot be accessed from UI) served by a package which has not been loaded yet. It does not matter if the user has clicked on a menu item or the running code has activated it with a "virtual click" the result is the same.
- **Object or service request**. The Shell is about to use an object or a service in a package not loaded yet. For example a tool window should be displayed or a service function is about to be executed.
- **Context change**. The Shell can enter into certain user interface contexts. For example, when you start debugging a project, the Shell enters into the Debugging context. When a solution with a single project is loaded, the Shell enters into the **SolutionHasSingleProject** context. You can declare that a package should be loaded as the Shell enters in a certain context. Visual Studio has a few predefined contexts, but you can also define your own ones.

So, if you do not need a package during the whole IDE session, it does not consume the memory at all. Should you click on a menu item activating a command sitting in a package which has not been loaded yet the IDE will immediately load and initialize it. Should you ask for a tool window in a package not in the memory yet, the IDE will start loading it.

Binding package loading with a context change is generally required where your package want to subscribe for events raised in Visual Studio. You cannot bind the event either to command activation or object or service requests, because in order your package could work you have to subscribe to events. The code to create subscriptions is generally put into the initialization code. But you cannot run any code belonging to a package while it is not loaded into the memory! In this case you declare that the best context (the latest possible) to load the package. If your package logic requires, you can specify the **NoSolutionExists** context. Visual Studio enters into

this context immediately when the Shell is loaded and ready to function, so packages bound to this context load at Visual Studio startup time.

Note: Be frugal with system resources if you have to load a package at Visual Studio startup time. Allocate only resources that are indispensable to carry out the required initialization at startup time.

Package Siting

When you develop a package it is an independent piece of code. When it is loaded into Visual Studio it becomes an organic part of the IDE:

- Your package can access services and objects provided by the Shell and other packages.
- The Shell and other packages can access the objects and services you proffer them.

The process of making a package physically integrated into the Shell is called *siting*. While the package is not sited, its functions cannot be used from outside. From the same reason the package can be only partially initialized, because it cannot touch any objects or services through the Shell. As soon as the package gets sited, it is ready to finish its initialization and be fully functional. Siting happens when Visual Studio loads the package.

The object type representing your package must implement an interface called IVsPackage (you are going to learn details later in this chapter) and must have a default constructor.

Loading a package contains the following steps:

- The Shell creates an instance of the object type representing your package by invoking its default constructor.
- The Shell calls the SetSite method of the object instance — IVsPackage defines this method — and passes a so-called service provider instance which can be used by the package to query objects in order to access services implemented by the Shell or other packages.
- The Shell allows your package to finish its initialization. In the first step during the construction call the service object did not have any contact with the Shell. After siting all initialization steps invoking Shell services can be carried out.

Although siting physically integrates your package functionally with Visual Studio, functional integration may require some additional — and sometimes complex — steps depending on what your package is intended to do.

Package Registration

Visual Studio must keep track of packages in order to load and use them. Of course, the most flexible would be some kind of discovery where the Shell can look around only in the file system to guess out which entities represent packages. The .NET framework supports and intensively uses metadata (attributes) that can represent the information you can use for this purpose. You

can even load an assembly so that only the metadata part is read from the file system and put into the memory. Although you can imagine this mechanism could work, it is not used by the Shell in this way.

The reason is that the roots of Visual Studio go back to the COM era. Packages are COM objects and can be created not only in managed code but also in native code using the Win32 API with any language including C++, Delphi and others. So, not surprisingly Visual Studio uses the registry to keep information about packages.

Although the registry is used to store configuration information about Visual Studio settings, developers and package users are not bothered with registration issues. The new deployment mechanism (through VSIX files) implemented in Visual Studio 2010 takes away this task. When Visual Studio is started, its discovery mechanism collects the information to be entered into the registry and does all the registration process on the fly. Developers perceive they do installation only by copying files and this resembles to the great mechanism we are using with the .NET Framework.

Package Information in the Registry

Packages are loaded into the memory on-demand. To effectively use this approach, Visual Studio stores the registration information about packages in a way so that this information could be accessed from the direction of objects to be used. For example, one kind of object that could trigger package loading is a tool window. A tool window has its own identity. When it is about to be created in order to be displayed, Visual Studio turns into the registry and addresses the corresponding key with the tool window identity. Under that key there is information stored about the identity of the package owning and implementing the tool window. Using this identity Visual Studio finds the registry key containing the package information and loads the package into the memory accordingly. The identity of packages and objects is represented by a GUID.

Figure 10 shows a few registry keys that store package-related information for Visual Studio.

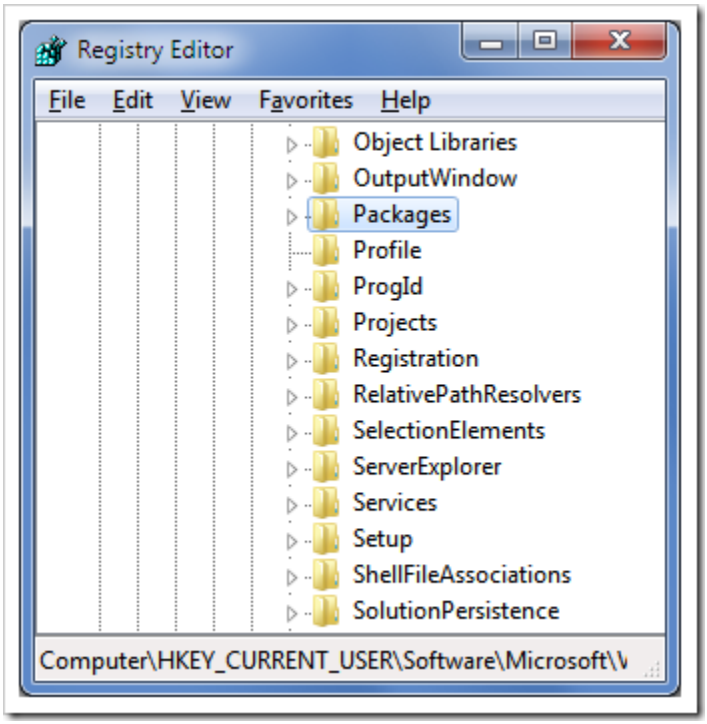


Figure 10: *Package and object information in the registry*

Assume that Visual Studio is about to create a service instance. It uses the Services registry key and according to the service identity it finds the package. Figure 11 illustrates the service key of the Class Designer Service. At the left side you can see the subkey named by the GUID identifying the service. At the right side you see the corresponding package ID in the default value under the service key.

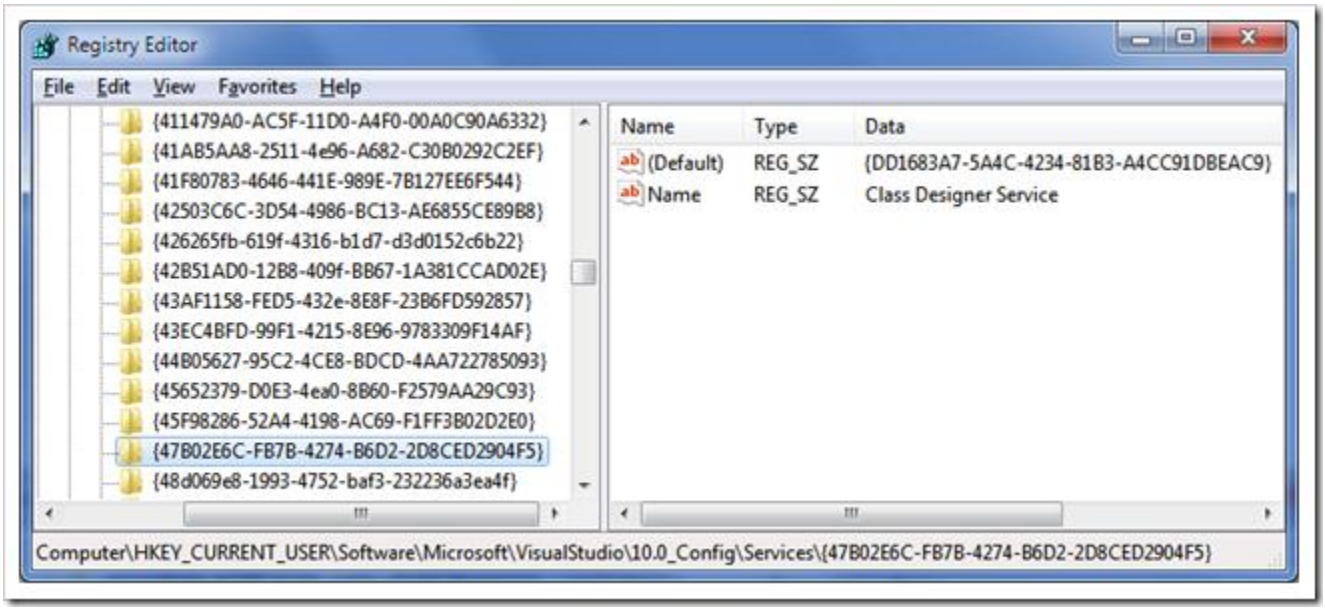


Figure 11: *Service information in the registry*

Visual Studio uses the same approach to find owner packages for any other objects.

There is one important thing to treat about package information in the registry. Packages can add their own user interface items to the Visual Studio IDE in form of menus and toolbars. If this UI information were created during the package initialization time, Visual Studio would need to load the package at startup time in order to present the package-dependent UI to the users. This approach would demolish the whole concept of on-demand loading.

Instead, the package uses a declarative way to define the user interface that should be displayed in the IDE at startup time. This information is encapsulated into the package assembly as an embedded resource. During the registration process Visual Studio extracts this resource information and merges it with its standard menus. Each user interface element — the commands they represent — is associated with the identity (GUID) of the owner package, so the Shell knows which package to load when a command is activated.

Visual Studio Experimental Instance

When you started the FirstLook package a new instance of Visual Studio was launched with the "Experimental Instance" text in its caption. What is that Visual Studio instance and how did it get to the computer?

Visual Studio Experimental Instance is a test bed to run and debug Visual Studio packages during the development and test phase. It is not a new installation of Visual Studio. It is the same **devenv.exe** file you use normally. But why you need it?

As it's been mentioned earlier the package is registered — information is written into the system registry — in order it could integrate with Visual Studio. Every time you build and run a package some information goes into the registry. When you modify the package it might affect also the information in the registry. You can imagine what an unexpected confusion it can lead that you always modify the registry under the Visual Studio instance you actually use to develop and debug a package? What if your package does not work or even worse it prevents Visual Studio start correctly or even causes a crash? How would you fix the pollution of registry?

That is the point when Visual Studio Experimental Instance comes into the picture. The Experimental Instance is simply another instance of Visual Studio that picks up its settings from a different place — including configuration files and registry keys.

The **devenv.exe** keeps its configuration settings in the system registry in the **CURRENT_USER** hive under the **Software\Microsoft\VisualStudio\10.0_Config** key. When Visual Studio runs it reads out the configuration information from these keys by default. With the **/rootsuffix** command line parameters the root key used by **devenv.exe** can be changed.

The VSPackage Wizard sets up the package project so that the **devenv.exe** uses the **/rootsuffix Exp** command line parameter when running or debugging the package. By doing it so **devenv.exe** will use the **Software\Microsoft\VisualStudio\10.0Exp_Config** registry key under the **CURRENT_USER** registry hive. So running the package with the Start Debugging (F5) or Start Without Debugging (Ctrl+F5) functions will launch the Visual Studio Experimental Instance using this registry key.

The build process of a package copies the package binaries and the so-called VSIX manifest information to a well-known location under the current user's application data folder. When the Experimental Hive starts, it discovers the package in the folder and uses the information found there to enter the package information into the registry key consumed by the Experimental Instance.

Using the Experimental Instance prevents you polluting the registry of the Visual Studio instance used for the normal development process. However, making mistakes, recompilations and faulty package registrations does not prevent putting junk or leaving orphaned information in the Experimental Instance registry. Making an appropriate cleanup could be very complex and dangerous if you would do it by delving in the registry.

Cleaning up packages from the Experimental Instance's registry is quite easy! The Visual Studio SDK installs the **CreateExpInstance.exe** utility and adds it to the Visual Studio 2010 SDK menu items under the Tools folder with the name "Reset the Microsoft Visual Studio 2010 Experimental Instance". Running this utility will reset the registry key belonging to the Experimental Hive to the state after the installation of VS SDK.

Note: If you develop Visual Studio packages you can get into situations several times where your package which worked before seems suddenly stop working. In majority of these cases the cause is the pollution of Visual Studio registry. There can be many orphaned objects in the registry as a result of continuous modification and rebuild of packages, and that can cause problems. If you run in such a situation, reset the Experimental Instance and then build your package with the Rebuild All function of Visual Studio. This procedure often helps.

Diving into the Package Source Code

You have already created a **FirstLook** project with the VSPackage wizard and by now you have a good overview about the basic concepts behind packages. Now we go into details and look how those concepts and ideas are reflected in the source code.

When the VSPackage Wizard generated the code according to the parameters we specified on the wizard pages, it made a lot of work at the background. The wizard carried out the following activities:

- Generation of a class library project in C#.
- Adding references to this project for the interoperability assemblies required to access Visual Studio functionality.

- Creating resources used by the package and other resources used by the registration process.
- Adding new MSBuild targets to the project to support the build and registration process of the package.
- Generation of types responsible for implementing the package functionality.
- Setting up the debug properties of the project to start Visual Studio Experimental Instance.

Table 1 summarizes the source files in the **FirstLook** project.

Table 1: *FirstLook source files generated by the VSPackage wizard*

Source File	Description
source.extension.vsixmanifest	This is the so-called VSIX manifest file which plays vital role in the discovery and registration mechanism of Visual Studio extensions.
FirstLook.vsct	The so-called command table file storing the definition of the menus and commands to be merged into the Visual Studio IDE during the registration process.
FirstLookPackage.cs	Class implementing the simple functionality of the package.
GlobalSupressions.cs	Attributes used to suppress messages coming from the static code analysis.
Guids.cs	GUID values used to identify the package and command objects within the package.
Key.snk	Signing key used to generate the strong name for the package assembly.
PkgCmdID.cs	Constants for identifying command values.
Resources.resx	Resource file to store your functional package resources — resources you use according to the functions you implement in the package.
VSPackage.resx	Resource file to store package infrastructure resources — those resources which are used by Visual Studio to integrate your package into the IDE.

The wizard added several assemblies to the class library project. Table 2 summarizes their roles. Their names start with the **Microsoft.VisualStudio**, this prefix is omitted in the table for the sake of clarity.

Table 2: *Interoperability assembly references in the project*

Source File	Description
~.Shell.Interop	This assembly defines several hundreds of core interoperability types (interfaces, structures, enumerations, classes, etc.).
~.Shell.Interop.8.0, ~.Shell.Interop.9.0 and *~.Shell.Interop.10.0*	There are COM types new in VS 2005, VS 2008 and VS 2010 IDEs. The interoperability wrappers of them are defined in these assemblies where the 8.0 suffix is for VS 2005, the 9.0 suffix for VS 2008 while the 10.0 suffix for VS 2010.
~.Shell.Interop.Immutable.10.0	A few abstract and enumeration types used for package registration attributes.
~.OLE.Interop	There are a few hundred of standard OLE types and interfaces. This assembly provides wrapper types for them.
~.Shell.9.0	The core types of the Managed Package Framework. There is a separate version for VS 2008 with the 9.0 suffix. If you work with VS 2008, you should use this assembly.
~.TextManager.Interop	Interoperability assembly with wrapper types to access text editor and text management functionality. Although this assembly is referenced by the project, it is not used in the package you've created.

All assemblies having Interop in their names contain only proxy type definitions to access the core Visual Studio COM service interface and object types.

Package Type Definition

Now, let's see the source code of the package! The wizard added many useful comments to the generated source files. In the code extracts listed here those commands are cut out to make the listings shorter and improve the readability of the code. The indentation has also been changed a bit for the same purpose.

Listing 1 shows the source code of the most important file in our project named FirstLookPackage.cs. This file implements the type representing our package:

Listing 1: *FirstLookPackage.cs*

```
1. using System;
2. using System.ComponentModel.Design;
3. using System.Diagnostics;
```

```
4. using System.Globalization;
5. using System.Runtime.InteropServices;
6. using Microsoft.VisualStudio.Shell;
7. using Microsoft.VisualStudio.Shell.Interop;
8.
9. namespace DeepDiver.FirstLook
10. {
11.   [PackageRegistration(UseManagedResourcesOnly = true)]
12.   [InstalledProductRegistration("#110", "#112", "1.0", IconResourceID =
   400)]
13.   [ProvideMenuResource("Menus.ctmenu", 1)]
14.   [Guid(GuidList.guidFirstLookPkgString)]
15.   public sealed class FirstLookPackage : Package
16.   {
17.     public FirstLookPackage()
18.     {
19.       Trace.WriteLine(string.Format(CultureInfo.CurrentCulture,
   "Entering constructor for: {0}", this.ToString()));
20.     }
21.
22.     protected override void Initialize()
23.     {
24.       Trace.WriteLine(string.Format(CultureInfo.CurrentCulture,
   "Entering Initialize() of: {0}", this.ToString()));
25.       base.Initialize();
26.
27.       // Add our command handlers for menu (commands must exist in the
   .vsct file)
28.       OleMenuCommandService mcs =
   GetService(typeof(IMenuCommandService)) as OleMenuCommandService;
29.       if (null != mcs)
30.       {
31.         // Create the command for the menu item.
32.         CommandID menuCommandID = new
   CommandID(GuidList.guidFirstLookCmdSet,
   (int)PkgCmdIDList.cmdidShowMyMessage);
33.         MenuCommand menuItem = new MenuCommand(MenuItemCallback,
   menuCommandID);
34.         mcs.AddCommand(menuItem);
35.       }
36.     }
37.
38.     private void MenuItemCallback(object sender, EventArgs e)
39.     {
40.       // Show a Message Box to prove we were here
41.       IVsUIShell uiShell = (IVsUIShell)GetService(typeof(SVsUIShell));
42.       Guid clsid = Guid.Empty;
43.       int result;
44.       Microsoft.VisualStudio.ErrorHandler.ThrowOnFailure(uiShell.ShowMe
   ssageBox(
45.                   0,
46.                   ref clsid,
47.                   "FirstLook",
48.                   string.Format(CultureInfo.CurrentCulture, "Inside
   {0}.MenuItemCallback()", this.ToString()),
```

```
                    string.Empty,
                    0,
                    OLEMSGBUTTON.OLEMSGBUTTON_OK,
                    OLEMSGDEFBUTTON.OLEMSGDEFBUTTON_FIRST,
                    OLEMSGICON.OLEMSGICON_INFO,
                    0,        // false
                    out result));
        }
    }
}
```

The **FirstLookPackage** class becomes a working package by inheriting the behavior defined in the Package class of the **Microsoft.VisualStudio.Shell** namespace and by using the attributes decorating the class definition.

The **Package** base class implements the **IVsPackage** interface that is required by Visual Studio in order to take an object into account as a package. This interface provides a few methods managing the lifecycle of a package and also offers methods to access package related objects like tool windows, options pages, and automation objects. One of the most important of them is the **SetSite** method having the following signature:

```
int SetSite(IServiceProvider psp);
```

Visual Studio calls this method immediately after the package has been instantiated by its default constructor. The **psp** parameter is an instance of **System.IServiceProvider** and this object is the key in keeping contact between the package and the IDE: any time the package requests a service object from its context—from the IDE—the **psp** instance is used at the back, however, the implementation of **Package** hides it from our eyes.

The overridden **Initialize** method is called after the package has been successfully sited. This method has to do all the initialization steps that require access to services provided by the Shell or other packages. Should you move this code to the package constructor you would get a **NullReferenceException** because at that point all attempts to access the Shell would fail as the package is not sited yet and actually has no contact with any shell objects.

The package constructor should do only inexpensive initialization that you would put normally to a constructor. Any other kind of initialization activities should be put to the overridden **Initialize** method. If you have some other expensive initialization activity that can be postponed, you should do them right when there's no more time to delay them.

In this case the **Initialize** method binds the single menu command provided by the **FirstLookPackage** with its event handler method called **MenuItemCallback**:

```
protected override void Initialize()
{
  Trace.WriteLine(string.Format(CultureInfo.CurrentCulture, "Entering
  Initialize() of: {0}", this.ToString()));
  base.Initialize();
```

```
5.
6.    // Add our command handlers for menu (commands must exist in the .vsct
   file)
7.    OleMenuCommandService mcs = GetService(typeof(IMenuCommandService)) as
   OleMenuCommandService;
8.    if (null != mcs)
9.    {
10.       // Create the command for the menu item.
11.       CommandID menuCommandID = new
   CommandID(GuidList.guidFirstLookCmdSet,
   (int)PkgCmdIDList.cmdidShowMyMessage);
12.       MenuCommand menuItem = new MenuCommand(MenuItemCallback,
   menuCommandID);
13.       mcs.AddCommand(menuItem);
14.    }
15. }
```

First it calls the **Initialize** method of the base class — **Package** in this case. Omitting the call to the base implementation would prevent the package running correctly. Look at the call of **GetService** in Line 7! If you could select a method that is especially very important when creating Visual Studio Extensions, probably the **GetService** method is that one! This method is implemented by the **Package** class — many other Managed Package Framework objects also implement this method — in order to request service objects from the environment. **GetService** has one type parameter — it's called service address — that retrieves a service object implementing the service interface specified by the address type.

So, Line 7 obtains an **OleMenuCommandService** instance that you can use to bind event handlers to so-called command objects. In Line 11 and 12 a **CommandID** instance is created to address the command to be put to the Tools menu. In Line 12 a **MenuCommand** instance is created to assigns the **MenuItemCallback** method as a response for the command specified with the **CommandID** instance. Line 13 entitles the menu command service to handle events related to the command. The result of this short initialization code is that your package handles the event when the user clicks the Simple Message menu item in the Tools menu by executing the **MenuItemCallback** method. In the next chapter you will find all nitty-gritty details about the command handling concepts used in Visual Studio and there you will learn much more about the initialization approach used here.

The **MenuItemCallback** method uses the **IVsUIShell** service to pop up a message box from within the IDE.

Registration Attributes

By now you know that packages are registered with Visual Studio in order to support the on-demand loading mechanism and allow merging menus and toolbars into the user interface of the IDE. The information to be registered is created during the build process from attributes assigned to the package class:

```
1. [PackageRegistration(UseManagedResourcesOnly = true)]
```

```
2. [InstalledProductRegistration("#110", "#112", "1.0", IconResourceID =
   400)]
3. [ProvideMenuResource("Menus.ctmenu", 1)]
4. [Guid(GuidList.guidFirstLookPkgString)]
5. public sealed class FirstLookPackage : Package
6. {
7.   // --- Package body omitted
8. }
```

Packages are COM objects and so they must have a GUID uniquely identifying them. The **Guid** attribute is used by the .NET framework to assign this GUID value to a type. All attributes above except **Guid** are derived from the **RegistrationAttribute** class which is the abstract root class for all attributes taking a role in package registration. Table 3 describes the attributes decorating the **FirstLookPackage**:

Table 3: *Registration attributes in FirstLookPackage*

Source File	Description
PackageRegistration	Adding this attribute to a class, the build process will handle it as a package and looks for other attributes to prepare the package registration according to your intention. In the example this attribute sets the **UseManagedResourcesOnly** flag to tell that all resources used by the package are described in the managed package and not in a satellite DLL.
InstalledProductRegistration	This attribute is responsible to provide information to be displayed in the Help|About dialog in the IDE. The constructor of this attribute requires four arguments with the following meanings: • The first and second strings provide the name and the description of the package. The "#" characters indicate that these values should be looked up in the package resources with the IDs following "#". • The third "1.0" parameter is the product ID (version number). • The fourth parameter (**IconResourceID**) tells which icon to use for the package. All resources (name, description and icon) should be defined in the VSPackage.resx file.
ProvideMenuResource	This attribute is to create registry entries about menu and toolbar items provided by the package. Visual Studio uses the embedded resources here to merge the package menus into the Visual Studio menus. The attribute has two parameters. The first is the so-called resourceID. This value is set to **Menus.ctmenu** as during the build process the VSCT

compiler uses this value by default when adding the binary representation of the .vsct file to the package resources. The second parameter is called versionID and it plays important role in the caching mechanism of resources. In the next chapter you are going to examine the role of this attribute in details.

There are many other registration attribute beside the ones in the table above, later in the book we are going to meet a few of them. You are not constrained to apply only the registration attributes defined by the Managed Package Framework, and you can define your own attributes. Any of them including yours are handled on the same way by the build process as the predefined ones.

The Command Table

The wizard generated a file named **FirstLook.vsct**. It is an XML file and the file extension refers to the acronym coming from the "Visual Studio Command Table" expression. The schema of the XML file defines the command table owned by the package.

The command table is transformed into a binary format during the build process and is embedded into the package assembly as a resource. During the registration phase the ID of this resource is put into the registry. When Visual Studio starts, loads this binary resource information and merges it with the menus of the IDE including toolbars and context menus.

In order to avoid menu merges every time Visual Studio is launched, the IDE uses a cache mechanism and carries out the merge process only once for each package.

The next chapter will treat this mechanism and the structure of the command table in details. Listing 2 shows you the command table described in the **FirstLook.vsct** file.

Listing 2: *FirstLook.vsct*

```
1. <?xml version="1.0" encoding="utf-8"?>
2. <CommandTable xmlns="http://schemas.microsoft.com/VisualStudio/2005-10-
   18/CommandTable"
3.               xmlns:xs="http://www.w3.org/2001/XMLSchema">
4.   <Extern href="stdidcmd.h"/>
5.   <Extern href="vsshlids.h"/>
6.   <Extern href="msobtnid.h"/>
7.
8.   <Commands package="guidFirstLookPkg">
9.     <Groups>
10.       <Group guid="guidFirstLookCmdSet" id="MyMenuGroup"
   priority="0x0600">
11.         <Parent guid="guidSHLMainMenu" id="IDM_VS_MENU_TOOLS"/>
12.       </Group>
13.     </Groups>
```

```
14.
15.      <Buttons>
16.        <Button guid="guidFirstLookCmdSet" id="cmdidShowMyMessage"
   priority="0x0100" type="Button">
17.          <Parent guid="guidFirstLookCmdSet" id="MyMenuGroup" />
18.          <Icon guid="guidImages" id="bmpPic1" />
19.          <Strings>
20.            <CommandName>cmdidShowMyMessage</CommandName>
21.            <ButtonText>Simple Message</ButtonText>
22.          </Strings>
23.        </Button>
24.      </Buttons>
25.
26.      <Bitmaps>
27.        <Bitmap guid="guidImages" href="Resources\Images_32bit.bmp"
   usedList="bmpPic1, bmpPic2, bmpPicSearch, bmpPicX, bmpPicArrows"/>
28.      </Bitmaps>
29.    </Commands>
30.
31.    <Symbols>
32.      <GuidSymbol name="guidFirstLookPkg" value="{d55758eb-6581-48fe-
   930b-f3536f43b6f0}" />
33.      <GuidSymbol name="guidFirstLookCmdSet" value="{05da2180-8d8e-4822-
   913b-b6a9012c4b2b}">
34.        <IDSymbol name="MyMenuGroup" value="0x1020" />
35.        <IDSymbol name="cmdidShowMyMessage" value="0x0100" />
36.      </GuidSymbol>
37.      <GuidSymbol name="guidImages" value="{49ba23b3-1631-483d-a095-
   003cb157f55d}" >
38.        <IDSymbol name="bmpPic1" value="1" />
39.        <IDSymbol name="bmpPic2" value="2" />
40.        <IDSymbol name="bmpPicSearch" value="3" />
41.        <IDSymbol name="bmpPicX" value="4" />
42.        <IDSymbol name="bmpPicArrows" value="5" />
43.      </GuidSymbol>
44.    </Symbols>
45.
46. </CommandTable>
```

In this listing all comments placed into the generated file are omitted for saving space. However, it is worth to read those comments to have a better understanding of the command table structure.

The **.vsct** file tells a lot about how Visual Studio is architected, how it solves the coupling of functions (commands) and user interface elements.

- Commands (actions to execute) are separated from the user interface element triggering the command. The same command can be assigned to different menus and toolbars; they will use the same action.
- Commands used together can be grouped and simply merged into existing menus by using the command group representation. It is much easier then coupling commands with hosting menus one-by-one.

- Elements are identified by symbols rather than using explicit values. This makes the coupling less error-prone: values of symbols must be defined only once, and the VSCT compiler can check for mistyping.

The root element of a **.vsct** file is the **CommandTable** element. As you can see all related elements are defined by the *http://schemas.microsoft.com/VisualStudio/2005-10-18/CommandTable* namespace. No doubt, the most important element is **Commands**, because this node defines commands, their initial layout and behavior.

Any command in the VS IDE must belong to the IDE itself or to a package. To assign a command to the appropriate (owning) VSPackage, the package attribute of the **Commands** element must name the GUID of the corresponding package.

Commands node can have a few child elements; each has a very specific role.

Group elements define so-called command groups; each of them is a logical set of related commands that visually stand together. In the **FirstLook.vsct** file we have a **Group** element that holds only a **Button**. A button represents a piece of user interface element the user can interact with, in this case a menu item that can be clicked. The **Parent** element defines the relationship between elements, for example the **Button** element defined above is parented in the **Group**.

Toolbars and menus would be poor without icons helping the user to associate a small image with the function. The **Bitmap** nodes allow defining the visual elements (icons) used in menus.

The **Symbols** section is a central place in the command table file where you can define the identifiers to be used in the other parts of the **.vsct** file. You can use the **GuidSymbol** element to define the "logical container GUID" and the nested **IDSymbol** elements to provide (optional) identifiers within the logical container. The name and the value attribute of these elements do exactly what you expect: associate the symbol name with its value.

The VSPackage Wizard put the generated GUID values into the **FirstLook.vsct** file but they also can be found in the **Guids.cs** file. The **PkgCmdID.cs** file defines constant values for the **IDSymbol** values used by package commands. These three files must be kept consistent, so if you change a GUID or a command identifier, the changes should be tracked in the other files as well; otherwise your package will not work as expected.

Package Resource Files

The **FirstLook** project has two resource files: **Resources.resx** and **VSPackage.resx**. They both utilize the resource handling mechanism in the .NET Framework, but have different roles.

Resources.resx is to store functional resources that are consumed by the objects and services of your package. For instance, you can store error messages, prompt strings, UI elements, logos, and so on in this resource file and access them programmatically through the static members of

the Resources class generated by the **ResXFileCodeGenerator** custom tool attached to the **.resx** file.

VSPackage.resx can store resources just like **Resources.resx**, but its primary role is to embed package infrastructure resources. This resource file does not use the **ResXFileCodeGenerator** custom tool and so does not generate any helper class to access resources.

As you remember the package is decorated with the **InstalledProductRegistration** attribute which refers to resource identifiers 110, 112 and 400:

```
1. [InstalledProductRegistration("#110", "#112", "1.0", IconResourceID = 400)]
```

These IDs refer to string and icon resources in the **VSPackage.resx** file as shown in Figure 12.

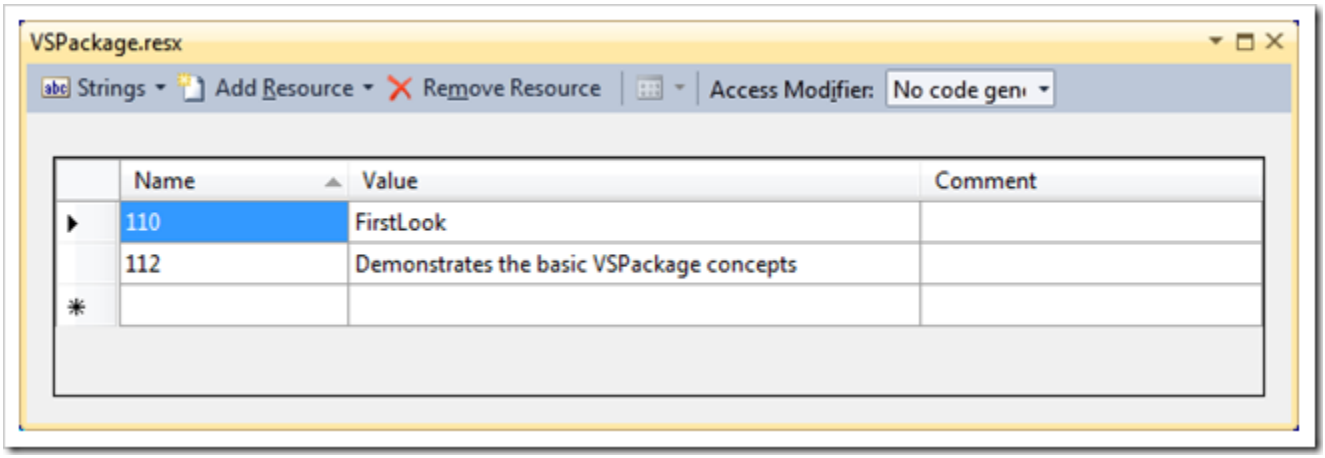


Figure 12: *String resources in VSPackage.resx*

Package resources will be extracted from the content of the **VSPackage.resx** file, so if you put them in the **Resources.resx** file, the package will not find the resource. Although you can put functional resources into **VSPackage.resx** file, their recommended place is the **Resources.resx** file.

The Package Build Process

Understanding the package build process can help a lot when you are about to debug or deploy your application. In this part you'll learn the steps of this process in details.

Building a package is not simply compiling the package source code into a .NET assembly. There are other important steps to complete in order to use the package either in the Experimental Instance or in its productive environment.

When the wizard generates the package it adds new build targets to the **.csproj** file of the corresponding class library. You can discover these entries by first unloading the project and

then editing the project file. If you want to try it, first right-click on the project file in Solution Explorer and use the Unload Project function, then activate the Edit FirstLook.csproj command also with right-click. When you scroll down to the bottom of the file you can discover the following entries:

```
1. <Import Project="$(MSBuildBinPath)\Microsoft.CSharp.targets" />
2. <Import
   Project="$(MSBuildExtensionsPath)\Microsoft\VisualStudio\v10.0\VSSDK\\uinput1
   ?
3.   Microsoft.VsSDK.targets" />
4.
```

The first Import entry can be found in any C# language projects to invoke the C# compiler and all the other tools (for example the resource compiler) to create the assemblies from the source project. The second Import entry is the one added by the VSPackage Wizard. The .targets file specified here contains Visual Studio SDK related build targets. If you would like to have a look at this file, you can find it in the *MSBuild\Microsoft\VisualStudio\v10.0\VSSDK* folder under Program Files. This book is not about MSBUILD, so you won't find more explanations about what the build targets describe and how they internally work, instead, here are the steps of the package build process:

- The Resources.resx and VSPackage.resx files are compiled into the corresponding .resource files.
- With the help of the vsct.exe utility the content of the .vsct file belonging to the package is compiled into a binary file called CTO file. This file format is used by Visual Studio when merging the package menu and command information into the IDE menu.
- The CTO file is merged into the VSPackage.resource file as a binary resource with the name of Menus.ctmenu.
- The C# compiler is invoked to compile the project source code just like normally when we build a standard class library. During this step the Resources.resources file and the VSPackage.resources files are embedded into the assembly.
- The CreatePkgDef.exe utility is executed and it scans the assembly for registration metadata information. Each registration attribute is translated to corresponding registry data. The utility is parameterized so that it creates a .pkgdef text file containing the information that is to be entered into the system registry.
- The binaries of the package (including the .dll and .pdb file), the corresponding .pkgdef file and the VSIX manifest file are zipped into a file with .vsix extension. This .vsix file is the installation kit of the package. You will find this file just beside the binaries (in the bin\Debug or bin\Release or maybe in another folder depending on the build configuration). With double clicking on this file you can start the Visual Studio Extension Installer utility.
- The .vsix file is installed in your LocalAppData folder under the Extensions subfolder of the Visual Studio Experimental Instance. The location of this folder depends on your user account and profile type. For example, if your user name is jsmith, you have installed Windows 7 on your C: drive and you have a local profile the build process will look for the *C:\Users\jsmith\AppData\Local\Microsoft\VisualStudio* folder and install the .vsix file under the *10.0Exp\Extensions* subfolder. The extension will be enabled (setting a key in the registry).

As a result of the build process the package is available in the Experimental Instance. The next time you start the Experimental Instance, it scans the Extensions folder, uses the .pkgdef file of your package to create the appropriate registry settings, the package's menus gets merged into the IDE, so your package is ready to run.

Debugging Visual Studio Packages

Anyone who develops software creates programming mistakes. Majority of them can be caught during a simple or more complex code review. Many of them are obvious and you can find them in the code after observing the faulty behavior. There are a few of them which cannot be easily caught without using a debugger.

Developing VSPackages is the same story. Sooner or later you find yourself debugging a package and searching for a bug. This book does not want to go into details about debugging techniques; this is definitely not its topic. However, you will learn how easy is to debug your package and what is going behind the scenes.

To debug or run a package you should set it as the startup project. If your package is the only project is the solution, it is already marked so. If you have more projects in the solution, you should mark any VSPackage project as the startup project.

Independently if you run a package with or without debugging, the Visual Studio Experimental Instance is started. You can check it on the project property pages on the Debug tab as Figure 13 shows it for the FirstLook project.

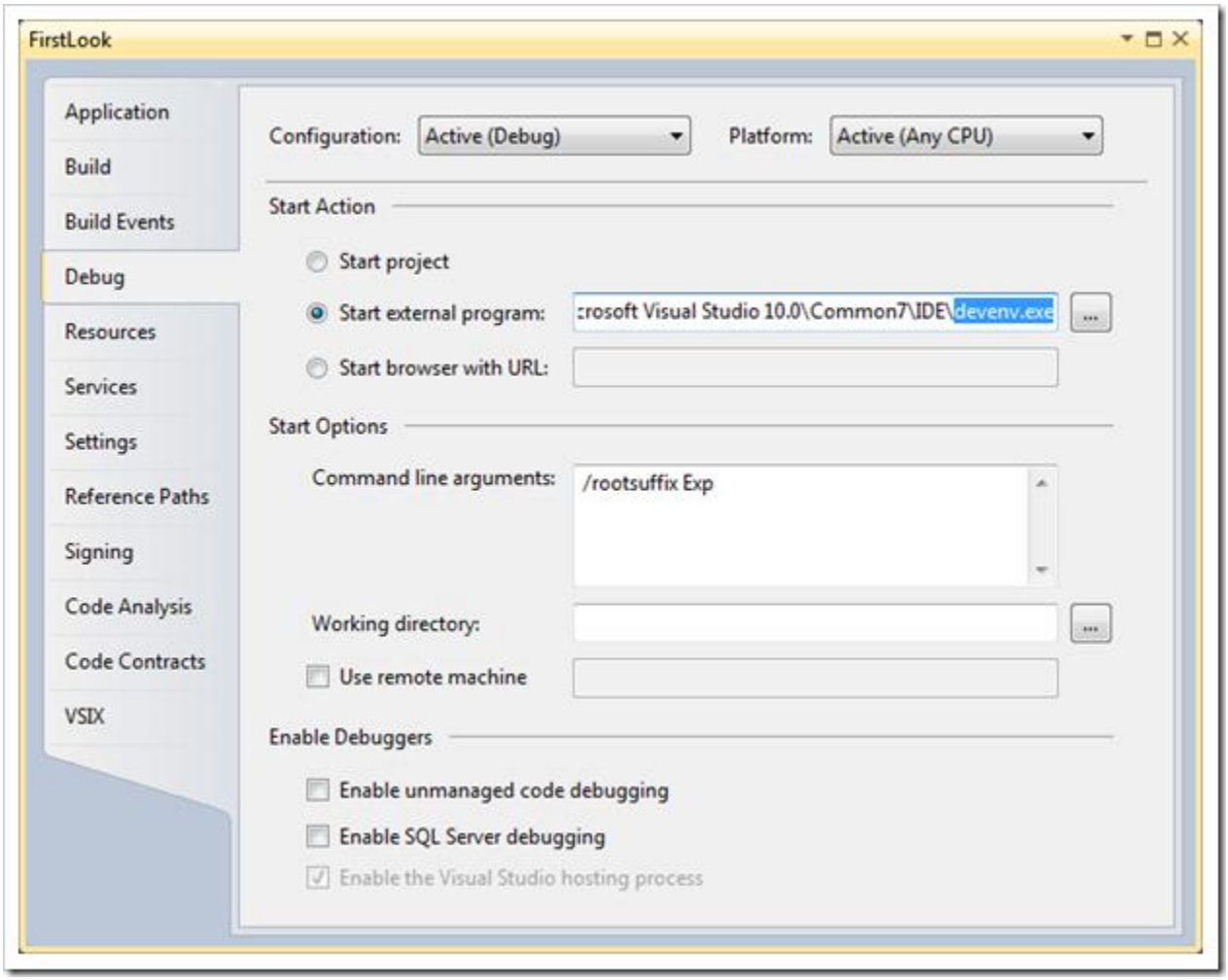


Figure 13: *Debug properties*

You can see that devenv.exe is selected as the startup project and it is launched with the **/rootsuffix Exp** command line parameters. As you have learnt before this command line starts the Experimental Instance.

When you start the project with the Start Debugging (F5) function Visual Studio attaches the debugger to the Experimental Instance and so you can set breakpoints in Visual Studio. As your package running in the Experimental Instance reaches a breakpoint, you are taken back to the debug view as Figure 14 illustrates it. In this case a breakpoint was set within the Initialize method of the **FirstLookPackage** class.

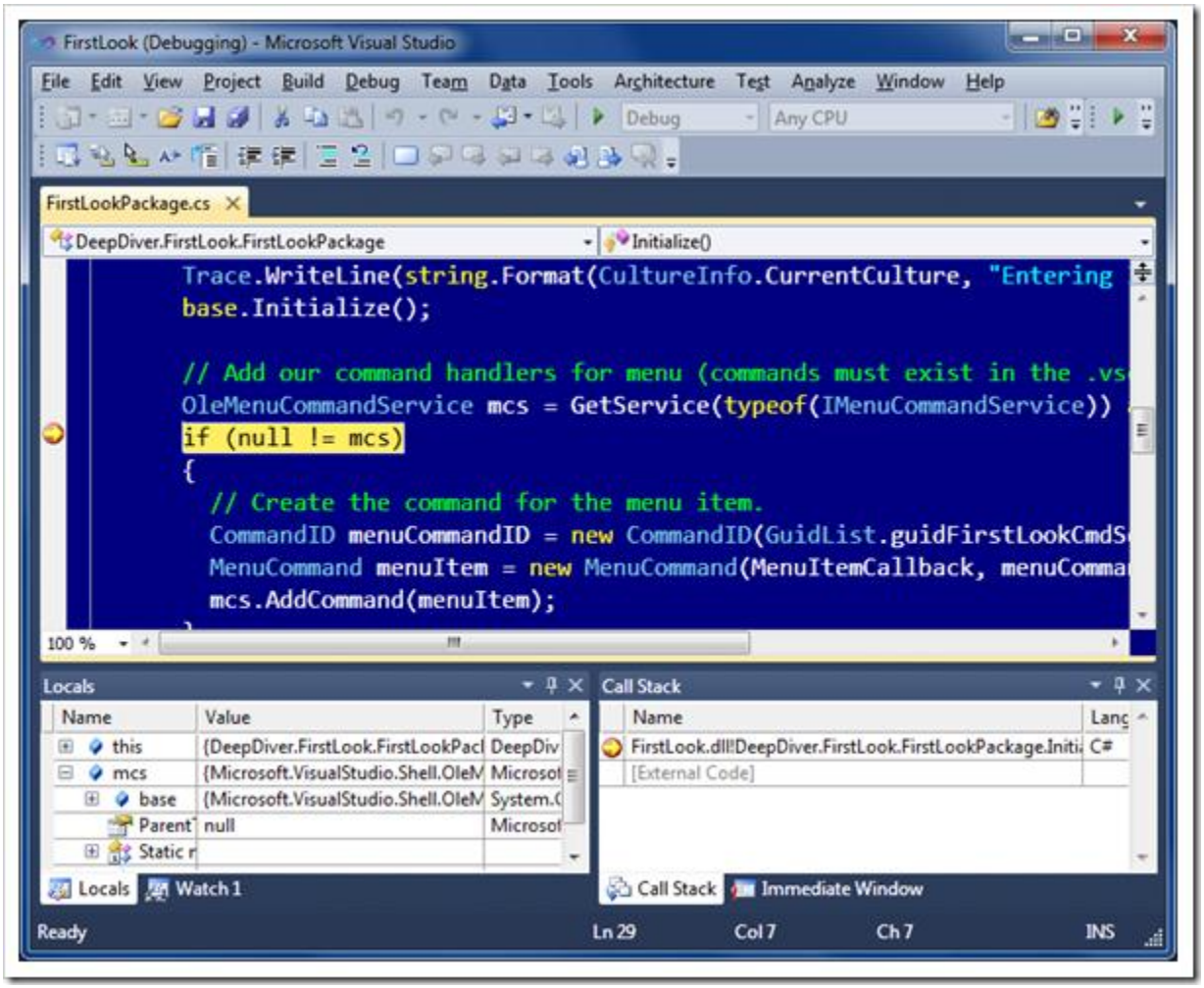


Figure 14: *The debugger in action*

You can use the same techniques for debugging a VSPackage as for any other applications. All debugging features of Visual Studio are accessible: you can watch variables, evaluate expressions, set up conditional breakpoints, and so on.

There are cases when you would like to trace your application without a debugger using trace messages. You can follow this practice with Visual Studio. The simplest one is writing to the Debug pane of the Output window. You can learn about this topic in Chapter 3.

Deploying a Package

It is very comfortable to use the Experimental Instance while developing a package. The build process takes care about setting up your package to work, so you can use either the Start

Debugging or Start Without Debugging commands to try what you've created. However, when your package is ready for distribution you should care about deployment questions.

Package Deployment in the Past

With Visual Studio versions preceding 2010 developers had to do some extra activities to prepare packages for deployment and it has a few potential pitfalls. The two main issues were that you needed to obtain a so-called Package Load Key (PLK) through a web page and take care of entering the required entries into the registry to allow Visual Studio recognize and integrate your package.

¾ Any change in package information like name, GUID, company or version required obtaining a new PLK. The Experimental Hive (this is what is now called Experimental Instance) did not check the PLK by default, so it often happened that developers faced with a wrong (missing or not renewed) PLK only after the installation kit was built and tested in the production Visual Studio environment.

¾ While the build process automatically registered the package under the Experimental Hive, developers had to create their own registration mechanism in the installation kit. It was not difficult, but because it was not automatic, forgotten registration updates could have led to annoying issues.

The VSIX installation

The new deployment mechanism built into Visual Studio 2010 removes this pain and provides an easy and straightforward way for package deployment.

Generally the easiest form of deploying an artifact is if you have an installation kit. The package build process as treated earlier creates this installation kit as a .vsix file containing the package binaries and some additional information. You can distribute your package by simply distributing the .vsix file to your customers. When they receive it, the Visual Studio Extension Installer utility can be started with double clicking on the .vsix file as Figure 15 illustrates.

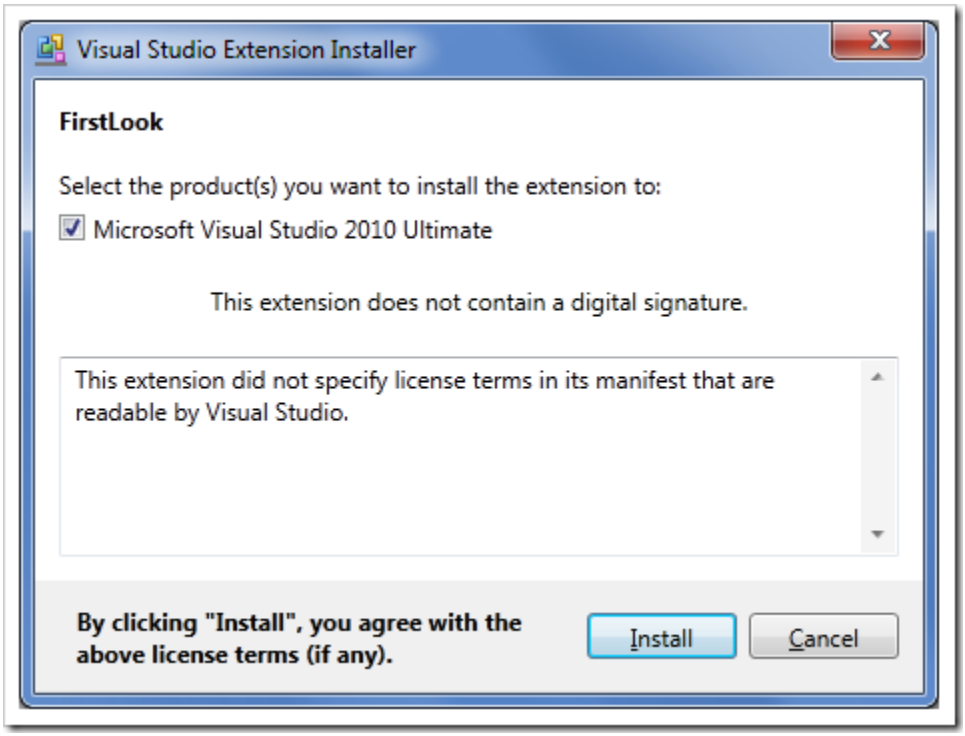


Figure 15: *Installing a VSIX file*

When you click install the content of the VSIX file will be installed to the specified Visual Studio instance.

If you create a package for a broad set of customers or for the community, you can upload the VSIX file to the Visual Studio Gallery. The new Shell of Visual Studio contains a great tool called Extension Manager that is able to search this gallery for extensions, install or remove them, and keep track of installed extensions as well as managing their updates. Figure 16 shows a screenshot of the Extension Manager browsing the extensions available on Visual Studio Gallery.

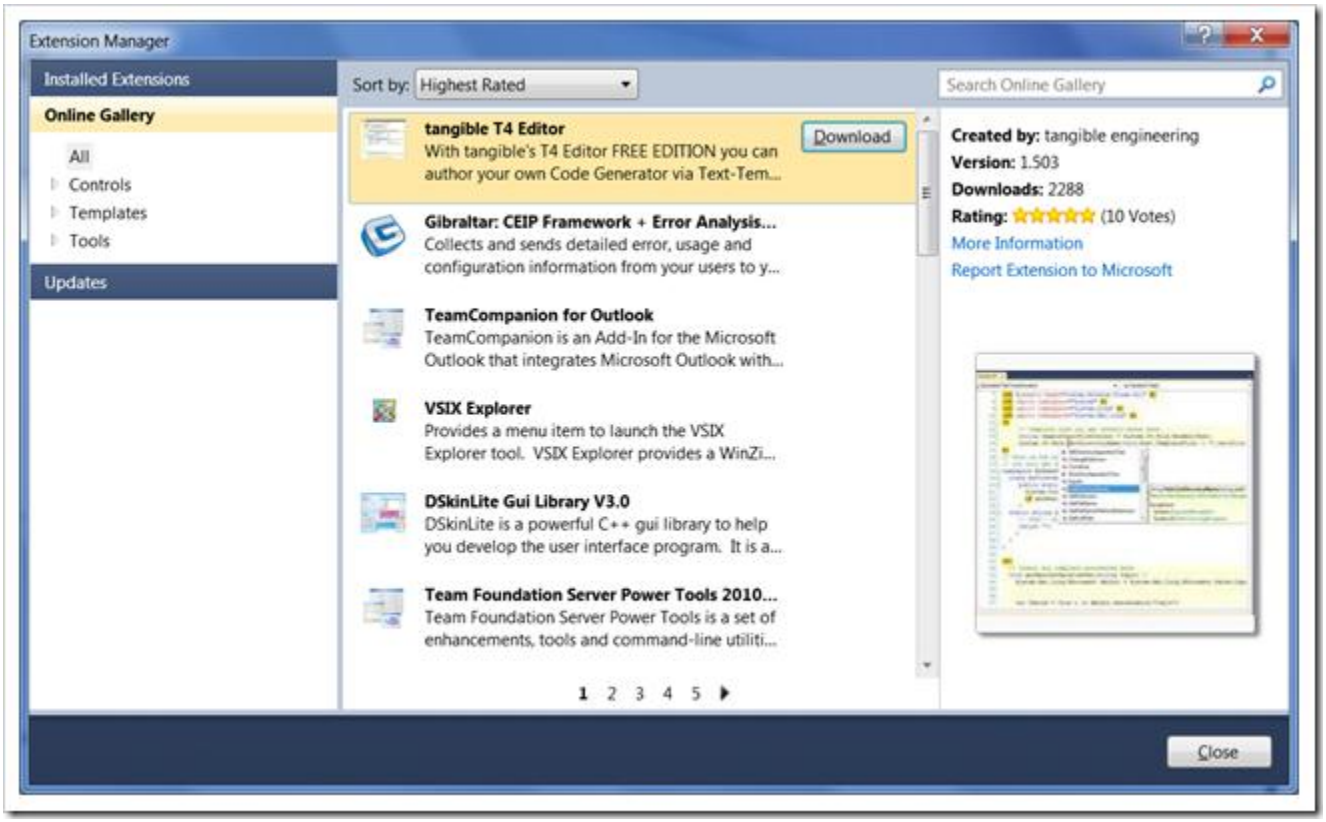


Figure 16: *Browsing Visual Studio Gallery with the Extension Manager*

You can select any of the components while browsing, and on the right pane of the window you find some more details about the highlighted item. With the More Information link you will be directed to component's home page on the Visual Studio Gallery. If you like this component, you can get it with the Download button just as others can obtain your uploaded components. The Extension Manager is the recommended way to obtain extensions. Because it runs within a Visual Studio instance, you can use it to install a separate set of components for your development environment and for the Experimental Instance. When using the Visual Studio Extension Installer utility your components will be installed under the normal development environment by default.

It was mentioned earlier that the build process packages the binaries and some other files into the VSIX file. In order the installation process could understand your .vsix installation file you need to create a so-called VSIX manifest file that is the soul of the installation kit. This file describes the metadata that is used as the set of instructions about what, where and how should be put during the setup. The VSPackage Wizard automatically creates this manifest for you and names the file as source.extension.vsixmanifest. You are probably not surprised that the manifest is an XML file with its own schema. When the FirstLook package was generated the wizard created the manifest shown in Listing 3:

Listing 3: *source.extension.vsixmanifest*

```
1. <?xml version="1.0" encoding="utf-8"?>
2. <Vsix Version="1.0.0" xmlns="http://schemas.microsoft.com/developer/vsx-
   schema/2010">
3.   <Identifier Id="d55758eb-6581-48fe-930b-f3536f43b6f0">
4.     <Name>FirstLook</Name>
5.     <Author>DeepDiver</Author>
6.     <Version>1.0</Version>
7.     <Description xml:space="preserve">Demonstrates the basic VSPackage
   concepts</Description>
8.     <Locale>1033</Locale>
9.     <InstalledByMsi>false</InstalledByMsi>
10.          <SupportedProducts>
11.            <VisualStudio Version="10.0">
12.              <Edition>Pro</Edition>
13.            </VisualStudio>
14.          </SupportedProducts>
15.          <SupportedFrameworkRuntimeEdition MinVersion="4.0" MaxVersion="4.0"
   />
16.        </Identifier>
17.        <References>
18.        </References>
19.        <Content>
20.          <VsPackage>|FirstLook;PkgdefProjectOutputGroup|</VsPackage>
21.        </Content>
22.      </Vsix>
23.
```

The root element of the manifest structure is the VSIX element that uses the *http://schemas.microsoft.com/developer/vsx-schema/2010* namespace. The manifest contains three sections:

- **Identifier** is used to uniquely define the different installation packages. The information here is used by the setup mechanism to manage the initial setup and the updates using the **Id** attribute and **Version** element values. This section also contains information describing the package and attributes taken into account during the setup process.
- **References** element contains a collection of dependencies. Each item is a **Reference** element which defines a dependency on another product. In the FirstLook.vsix sample the package contains one dependency.
- The **Content** element is a collection of content items that are packed in the payload. It our case the content is a VSPackage where the value of the item is used to generate the .pkgdef file containing the package information.

When running the Visual Studio Extension Installer or using the Extension Manager, the VSIX manifest is used to determine how the VSIX package should be set up. The VSPackage content type tells the installer that the related FirstLook.pkgdef file will contain the information to be put into the registry in order to register the COM object representing a Visual Studio package. The FirstLook.pkgdef file that has been created during the build process contains the information in Listing 4:

Listing 4: *FirstLook.pkgdef*

```
1. [$RootKey$\InstalledProducts\FirstLookPackage]
2. @="#110"
3. "Package"="{d55758eb-6581-48fe-930b-f3536f43b6f0}"
4. "PID"="1.0"
5. "ProductDetails"="#112"
6. "LogoID"="#400"
7. [$RootKey$\Packages\{d55758eb-6581-48fe-930b-f3536f43b6f0}]
8. @="DeepDiver.FirstLook.FirstLookPackage, FirstLook, Version=1.0.0.0,
   Culture=neutral, PublicKeyToken=8e5c6425e9b83cf4"
9. "InprocServer32"="$WinDir$\SYSTEM32\MSCOREE.DLL"
10.      "Class"="DeepDiver.FirstLook.FirstLookPackage"
11.      "CodeBase"="$PackageFolder$\FirstLook.dll"
12.      [$RootKey$\Menus]
13.      "{d55758eb-6581-48fe-930b-f3536f43b6f0}"=", Menus.ctmenu, 1"
```

The content of this file resembles to the content of a .reg file that can be exported from or imported to the Windows registry. However, the .pkgdef file contains a few tokens closed between dollar signs. The values of these tokens are passed by the context of the .pkgdef file to the entity processing the file content.

For example, if you use the Visual Studio Extension Installer utility to process the .vsix file, the utility extracts the payload into the *Microsoft\VisualStudio\10.0\Extensions* subfolder under the *LocalAppData* folder of your user profile. The files are put not directly into the Extension folder but into the subfolder calculated from the **Author**, **Name** and **Version** elements of the manifest's Identity section. In this case the payload includes the FirstLook.pkgdef and FirstLook.dll files beside a few others.

When Visual Studio starts, it recognizes that a new .pkgdef file is under the Extensions folder and processes it. It substitutes the **$RootKey$** token with the corresponding registry root of Visual Studio 2010, **$Windir$** with the current Windows installation folder, **$PackageFolder$** with the encapsulating folder of the .pkgdef file. After Visual Studio startup finishes, all information required to find and load the package is entered into the registry. When the first action demanding the package is executed, Visual Studio can pick up and initialize it.

Summary

A VSPackage is the principal architectural unit of Visual Studio, a container for extensibility objects. It is also a unit from deployment, security and licensing aspects. Packages are not loaded immediately as Visual Studio starts, they are read into the memory on-demand at the first time when any of their objects or services is about to be used.

The process of integrating a package physically into the Shell is called siting. While the package is not sited, its functions cannot be used from outside. As soon as the package gets sited, it is ready to finish its initialization and be fully functional. Siting happens when Visual Studio loads the package.

VS 2010 Package Development

Visual Studio keeps track of packages installed through registration, and package information is stored in the system registry under a specific Visual Studio key. With command line parameters this registration key can be suffixed in order to use another configuration set — even with separate package registration parameters.

The Visual Studio SDK sets up the Visual Studio Experimental Instance which is a test bed to run and debug Visual Studio packages during the development and test phases. The Experimental Instance is not a separate Visual Studio installation, it uses the same devenv.exe file but with different configuration settings.

VSPackages use a build process that contains some additional steps in order to prepare the packages for debugging or deployment. The easiest way to create a package is running the VSPackage Wizard which sets up the build process appropriately. During this process package infrastructure resources — like to so-called command table — are embedded into the package assembly, the package installation kit is created and installed under the Experimental Instance.

During the development phase packages run inside the process space of the Experimental Instance and the same debug techniques can be used for tracing and troubleshooting as for any other .NET applications.

VSPackage deployment in Visual Studio 2010 became really simple related to the preceding versions. The package installation kit is represented by a VSIX file that can be distributed directly to the users of your package or — and this way opens up brand new opportunities — uploaded to the Visual Studio Gallery. The Extension Manager built into the IDE can be used to browse, install, and remove VSPackages (and many other kinds of extensions) as well as to keep track of them.

VS 2010 Package Development

Chapter 2: Commands, Menus and Toolbars

Almost all packages are created to allow the user interact with through the corresponding user interface. This interaction generally means the user can click on a menu or toolbar item and activates a function of the package.

From the user interface perspective it is pretty easy to imagine what a menu, a menu item or a toolbar is. Developers either creating Windows Forms or ASP.NET applications can understand these ideas and know their semantics. WFP developers even meet with the concept of command binding. Coming from the world of Windows Forms application programming, most developers put an equation sign between the menu item and the event handler for that item. Using the same approach for Visual Studio simply will not work: the model behind the IDE is more generic and from this aspect is more complex.

In this chapter you'll cruise around the concepts related to command handling. Command is a central concept of user interaction in Visual Studio — and as you are going to see, it is different from the idea most Windows Forms or ASP.NET programmers thinks about.

After reading this chapter you will be familiar with the following things:

- Important concepts behind the command handling mechanism, like menu item, command, context, command binding and state.
- The way the visibility and state of commands is determined by Visual Studio IDE. The CommandState sample will provide you a demonstration about how to enable or disable commands, and how to set their visual properties programmatically.
- The role and structure of the Visual Studio Command Table (.vsct) file. The BasicVSCTSample and the AdvancedVSCTSample packages will show you a few simple tasks to carry out with a .vsct file:
    - Creating a main menu
    - Using command groups and submenus
    - Decorating commands with icons and using command flags
    - Creating toolbars and menu controllers
- The concept and usage of visibility contexts. The VisibilityContextSample will demonstrate how to set up commands to be visible only in specific contexts.
- Command routing used by Visual Studio to direct commands to an appropriate command target.

Commands are indispensable and frequently used entities in Visual Studio. Although the fundamental ideas behind them and many practical details are treated in this chapter, you will find a lot of command-related information also in the subsequent chapters.

VS 2010 Package Development

Basic Command Handling Concepts

You generally create a VSPackage (and it is true for many other kind of software artifacts) in order to encapsulate functions. A part of these functions can be used from outside — for instance, by services your package offers for the Shell and for other packages. A part of these functions even can be used by users — assuming you provide appropriate interaction points to invoke them. These functions are called commands in the Visual Studio Extensibility terminology. A few samples are opening a solution, printing a source file, copying the selected text in the editor to the clipboard, adding a new file to the project hierarchy, and so on.

A command can be invoked by users only if you provide a way to do that. The most obvious way is to create a menu item or a button that can be clicked by the mouse or triggered by a keypress. There are other alternatives, for example you can provide a way where the user can type something in a console-like way to activate a command. If you do not think it is a real alternative, just look what the Command Window does in Visual Studio — visit the View ð Other Windows ð Command Window function!

Of course you have the "clickable" form of user interface representing commands in the Visual Studio IDE as menu and toolbar items. Each of them has the same function: when the user activates them, Visual Studio invokes the command bound to the item. From this perspective there is no difference if a command has been invoked from a menu or a toolbar.

- Menus are generally displayed in a row at the top of the IDE and they generally provide a visual hierarchy of the items executing commands. Other user interface elements of the IDE — like tool windows, document windows, window frames — also can have their own context menus that pop up when the user right-clicks on them.
- Toolbars hold a set of visual controls — typically organized in a row — that have the same function as menu items: execute commands. Toolbars can have a variety of control types, like buttons, text boxes, combo boxes and labels.

In this book the term of menu item is mentioned for user interface items that can be used to invoke a command independently of whether that is a menu item or a control on a toolbar.

Commands also can be executed programmatically. For example, the DTE object which is the root of the automation object model contains a method named ExecuteCommand that accepts a command name string and an optional parameter string — and as you guess it executes the action behind the command.

When pressing a button or triggering a command invocation action on any other kind of control, an event is raised that initiates the command action.

In the traditional Windows Forms programming many developers have only the idea of menu and toolbar items and event handler methods. Often they attach the same event handler method to more menu and toolbar items and handle the state of the UI separately. For example, if the

same functionality is used by a menu item and a toolbar item they attach the same event handler method but handle the enabled/disabled state of the menu item and the toolbar item separately.

Visual Studio makes a clear separation on the concept of menu items and commands.

A command is responsible for determining its state (name, visibility, enabled, disabled, etc.) and executing the command triggered. A menu item is responsible for presenting the visual properties of a command and providing a way the user can trigger the execution of the command.

This means a command object can be bound to zero, one or more menu items. The command knows its state and can report it to the related menu items: developers have one central location to handle the state of the command independently of how many menu items make that accessible. They only have to deal with the state of the command; menu items adjust their appearance by themselves. Also there is only one location for the command action code independently of the number of interaction points which can trigger the command.

So the idea of a menu (or toolbar) item and the command behind them are separated. There is another twist in the story: a command does not own the code that is intended to run when the command is invoked or when the status of the command is queried.

A command itself is a logical entity that can be forwarded to so-called command targets which know how to handle the semantics of a specific command. There is a command routing model in the IDE that forwards a command to a command target. The target either can do something with the request related to a command (for example, sets the command state disabled, execute the command, etc.) or can pass back the command as not supported (the target does not know what to do with that). The target even can pass the command to other command targets.

To make it easier to follow, let me tell you an example. We have Cut, Copy and Paste menu items in the Edit menu and on the standard toolbar of Visual Studio. Moreover, these items are added to a number of context menus — each of them can be accessed at least from three different locations. The items are bound to commands having the logical name of Cut, Copy and Paste. There is not a single object in Visual Studio that knows how to execute those operations. The IDE forwards these commands to command targets depending on the current context. For example, when the text editor is focused the IDE sends the commands to the text editor. When you are in the Properties window, commands are sent there. When you edit an .aspx page visually, the designer receives these operation commands. So, the text editor, the Properties window and the ASP.NET page designer all are command targets. They can decide if they support the command — they understand what the command means — and how to respond to that. If they support the command they are also able to execute the corresponding action.

Figure 1 visualizes the logical relation among basic entities that are involved in the process of executing a command, and Table 1 describes them:

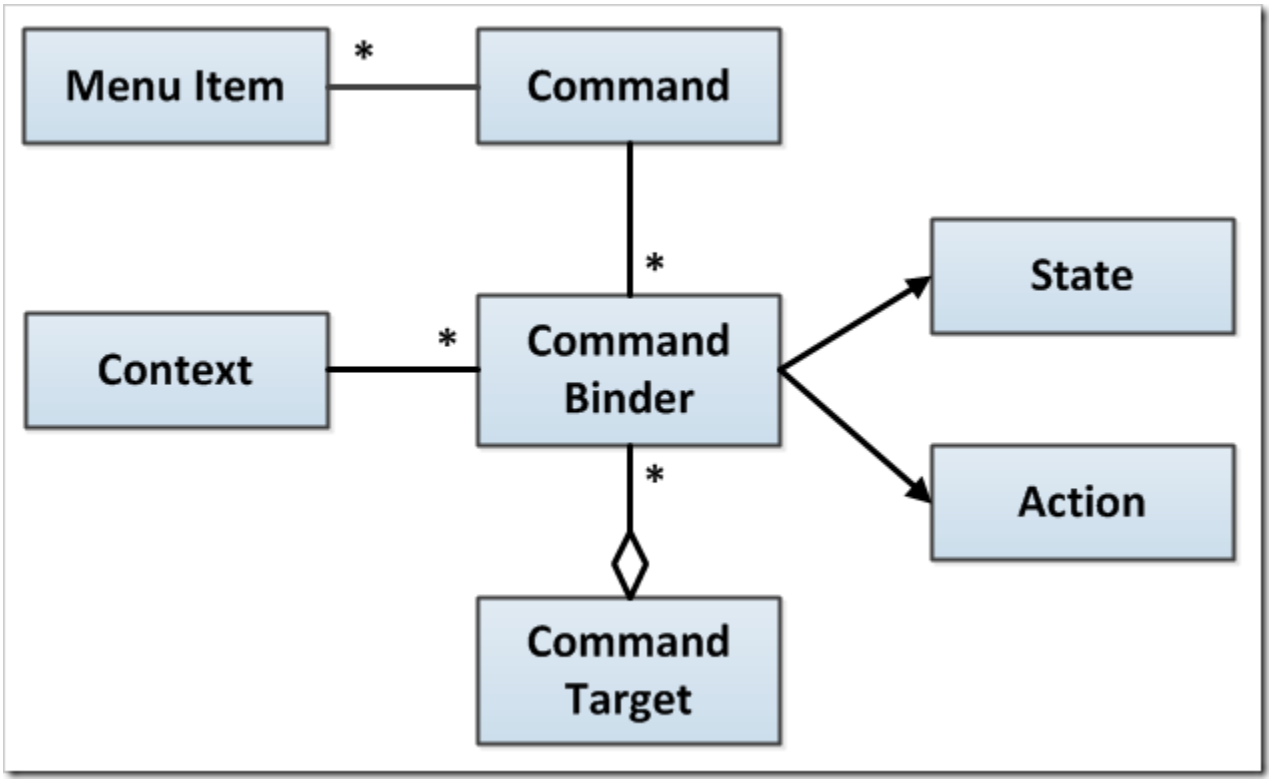


Figure 1: *Logical diagram of basic command handling entities*

Table 1: *Summary of basic command handling concepts*

Entity	*Responsibility*
Menu Item (toolbar item)	Provides a user interface to invoke commands. Allows feedback about the command state to the user.
Context	A logical context representing the state of the environment. The context determines the status of the command. For example, if there is no text copied to the clipboard, the **Paste** function is not enabled.
State	Commands are logical entities and they have states that influence if commands are allowed executing their related action or not and also how commands are visualized. Commands can be enabled or disabled and they can have shown or hidden states.
Action	The physical action to be invoked when an enabled command is executed by the

	command target.
Command	A logical entity representing an action that can be queried about its state within the current context, and can be resulted in executing some code. The command itself is just a logical entity, and it does not own the code responsible for status query and action execution.
Command Target	An entity that knows how to execute a status query and action belonging to a command entity. It can route, execute or even refuse actions of the command received.
Command Binder	An entity responsible for understanding a specific command received by the command target in the current context. The Command Binder invokes physically the action bound to the command.

The best way of getting more information about how these concepts work in practice is to create an example and diving into the code.

The best way of getting more information about how these concepts work in practice is to create an example and diving into the code.

Physical Representation of Command Handling Concepts

You are going to use the FirstLook package sample code introduced in the previous chapter as the first example, because this is one of the smallest which can demonstrate how the logical concepts are represented in code. This sample application adds a command named Simple Message to the Tools menu. The command displays a message box when the user clicks on the related menu item.

First let's have a look the code with the visual representations of commands. In the FirstLook.vsct file you can see the items for defining and placing the menu item:

```
1. <Groups>
2.   <Group guid="guidFirstLookCmdSet" id="MyMenuGroup" priority="0x0600">
3.     <Parent guid="guidSHLMainMenu" id="IDM_VS_MENU_TOOLS"/>
4.   </Group>
5. </Groups>
6.
7. <Buttons>
8.   <Button guid="guidFirstLookCmdSet" id="cmdidShowMyMessage"
   priority="0x0100"
9.     type="Button">
10.     <Parent guid="guidFirstLookCmdSet" id="MyMenuGroup" />
```

```
      <Icon guid="guidImages" id="bmpPic1" />
      <Strings>
        <CommandName>cmdidShowMyMessage</CommandName>
        <ButtonText>Simple Message</ButtonText>
      </Strings>
    </Button>
</Buttons>
```

By this definition the Simple Message menu item is represented by a Button element that is parented by a Group element. The Group, Button and Parent elements have guid and id attributes that are used as compound identifiers. The Parent elements create the relationships so that the Group is inserted into the Tools menu, the Button is added to the Group so as a result you'll have the Simple Message item added to the Tools menu.

The (guid, id) pairs are the compound keys of command table elements. The guid represents a logical container of elements belonging together and the id represents the unique element identifier within that container. Here the identifier of the Button element is actually the identifier of the command represented by this Button node. The command here is used implicitly with its ID. This nature of the compound key is also represented by the Symbols section where GuidSymbol nodes have IDSymbol child nodes describe this logical containment:

```
<GuidSymbol name="guidFirstLookCmdSet" value="{4423366f-0518-4fd9-832f-9efd21a9013b}">
  <IDSymbol name="MyMenuGroup" value="0x1020" />
  <IDSymbol name="cmdidShowMyMessage" value="0x0100" />
</GuidSymbol>
```

Let's see how this command is represented in the code. The Initialize method of the FirstLookPackage class contains the following code:

```
protected override void Initialize()
{
  Trace.WriteLine(string.Format(CultureInfo.CurrentCulture,
    "Entering Initialize() of: {0}", this.ToString()));
  base.Initialize();
  OleMenuCommandService mcs = GetService(typeof(IMenuCommandService))
    as OleMenuCommandService;
  if (null != mcs)
  {
    CommandID menuCommandID = new
  CommandID(GuidList.guidFirstLookCmdSet,
      (int)PkgCmdIDList.cmdidShowMyMessage);
    MenuCommand menuItem = new MenuCommand(MenuItemCallback,
  menuCommandID);
    mcs.AddCommand(menuItem);
  }
}
```

The package is a command target, so it can receive command notifications and execute actions of commands it understands. This behavior is defined by the IOleCommandTarget interface and the Microsoft.VisualStudio.Shell.Package class that is the ancestor of FirstLookPackage implements this interface. Although the package declares and understands only a single command, it receives many command notifications. However, it will refuse all commands except the one it knows. This command target behavior is delegated — the Package class implements it this way — to an object instance of type OleMenuCommandService. In Line 06 and 07 the GetService method is addressed with the type of IMenuCommandService to retrieve this OleMenuCommandService instance. You can add commands to this OleMenuCommandService instance to handle. When the package as a command target receives a command notification, it is processed by the command service instance. The command service checks if the command received is on its list or not. If the command is on the list, the required action is executed; otherwise command execution is refused. The sender object is notified about the command execution result (executed or refused).

Lines 10-13 contain the steps required to add a command to the list of the command service. First a CommandID instance is created which is a simple wrapper type for the compound key of a command. It uses a pair of System.Guid and System.UInt32 values to identify a command. When you check the identifiers used as the two construction parameters you can guess they are the same as the ones used for the Button node in the FirstLook.vsct file.

Line 12 creates a MenuCommand instance. MenuCommand is defined in the System.ComponentModel.Design namespace in the System.dll assembly and is the part of the .NET Framework and not the part of Visual Studio. It represents a part of the Command Binding entity id Figure 1. In Line 12 you construct it using the MenuItemCallback method delegate and the identifier of the command you have created in Line 10 and 11. MenuCommand — among the others — has an Invoke method to execute the menu action, and properties like Enabled and Visible to get or set the status of the related command item. The command service instance leverages on these properties and method to implement its behavior.

Line 13 adds the MenuCommand instance to the container of the command service object.

Now you can create Table 2 for associating the types for the roles summarized in Table 1.

Table 2: *Summary of types implementing the command handling concepts*

Entity	***Implementation***
Menu Item (toolbar item)	Menu and toolbar items do not appear as concrete .NET types in the package, they are defined in the .vsct file of the package project.
Command	In the FirstLook sample command states are not used. However, the properties of

State	MenuCommand — such as Enabled and Visible — represent the state.
Command	The command as a standalone physical entity is not used. The concept of command is represented by a CommandID instance that is used in several places. For example, a MenuCommand refers to a command by its CommandID property which is a CommandID instance.
Command Target	Objects implementing the IOleCommandTarget interface. In the FirstLook sample this is the FirstLookPackage class that implements the interface indirectly through the Package class. However, the package delegates this responsibility to its command service instance with the type of OleMenuCommandService.
Command Binder	In the FirstLook sample binders are represented by MenuCommand instances.

You have seen the implementation of a very simple command. It's time to look other examples demonstrating more aspects of Visual Studio's command handling architecture.

Command State and Visibility

Each command has a state in the current Visual Studio context. This state is built up from two orthogonal factors:

- The availability of a command (enabled or disabled) tells if it is allowed to be invoked within the context or not. For example, if you do not have any text selected in the active text editor, the Copy or Cut commands should not be allowed being invoked.
- The visual properties of a command tell if the menu and toolbar items related to the command are visible and what the displayed label or icon are.

Although these factors are independent from each other, the availability of a command is also reflected visually: menu items of disabled commands are grayed out.

The properties determining the state of commands can be set dynamically. Visual Studio updates the user interface asynchronously. It sends status requests to the appropriate command targets and asks them to tell the status of specific commands. This is done in Visual Studio idle time with an algorithm that sends status queries only for commands that have a chance to be displayed in the current context.

There is one thing that needs some consideration: how the initial state of a command is set up? Packages are on-demand loaded, so there is a part of the command lifecycle when their owner package is not loaded, but menu items representing them are already merged with Visual Studio menus. In this part of their life Visual Studio does not send status requests to the command target of unloaded commands, because it actually would require the command to be loaded.

Visual Studio uses the following approach to handle the visibility status of commands:

- While a command is not flagged in the command table (.vsct file) as one that changes any of its visual properties during its life — this flag wears the remarkable DynamicVisibility name —, the explicit visual status of the command is never queried and never updated, the initial state set up in the command table is used during the whole IDE session. Of course, you should initialize these commands to be visible (this is their default state); otherwise they are not very useful.
- If a command is flagged to dynamically change its visual properties, the initial settings in the command table are used unless the package is loaded. After the package gets sited the appropriate command target object is queried for the status of the command.

Command availability is not part of the explicit visual state. The enabled or hidden state of a command is maintained independently of the DynamicVisibility flag is used or not. While the package of the command is not loaded, the initial availability state defined in the command table is used. As soon as the package gets loaded the command target is queried for availability even if explicit visual state is not queried.

The Command State Sample

To follow the concepts above you'll examine a package example named CommandState. This package adds five menu items to the Tools menu as indicated in Figure 2.

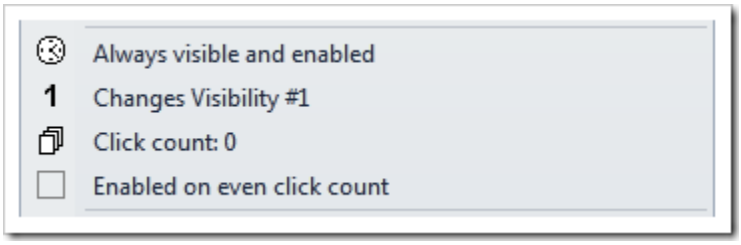


Figure 2: *Menu items of the CommandState package*

There is a hidden fifth menu item with the text "Changes Visibility #2". When you click any of the Changes Visibility items, #1 and #2 are swapped. The "Click count: 0" item counts every click and indicates this counter in the menu text. The bottom menu item changes its availability. It is enabled when the click count above is an even number; otherwise it is disabled. So clicking 3 times on the Changes Visibility items and three times on the Click count item leads to the command status result that is illustrated in Figure 3.

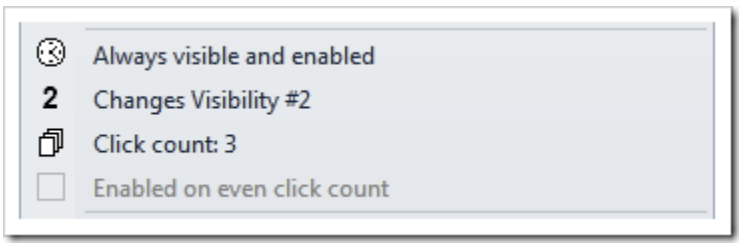


Figure 3: *Menu items after changing their visibility state*

Listing 1 shows the command table defining the layout of the menu and initial state of commands.

Listing 1: *CommandState.vsct*

```
1. <?xml version="1.0" encoding="utf-8"?>
2. <CommandTable xmlns="http://schemas.microsoft.com/VisualStudio/2005-10-
   18/CommandTable"
3.   xmlns:xs="http://www.w3.org/2001/XMLSchema">
4.   <Extern href="stdidcmd.h"/>
5.   <Extern href="vsshlids.h"/>
6.   <Extern href="msobtnid.h"/>
7.   <Commands package="guidCommandStatePkg">
8.     <Groups>
9.       <Group guid="guidCommandStateCmdSet" id="MyMenuGroup"
   priority="0x0600">
10.        <Parent guid="guidSHLMainMenu" id="IDM_VS_MENU_TOOLS"/>
11.      </Group>
12.    </Groups>
13.
14.    <Buttons>
15.      <Button guid="guidCommandStateCmdSet"
   id="AlwaysVisibleAndEnabled"
16.        priority="0x0100" type="Button">
17.        <Parent guid="guidCommandStateCmdSet" id="MyMenuGroup" />
18.        <Icon guid="guidOfficeIcon" id="msotcidClock" />
19.        <Strings>
20.          <CommandName>AlwaysVisibleAndEnabled</CommandName>
21.          <ButtonText>Always visible and enabled</ButtonText>
22.        </Strings>
23.      </Button>
24.
25.      <Button guid="guidCommandStateCmdSet" id="ChangesVisibility1"
26.        priority="0x0200" type="Button">
27.        <Parent guid="guidCommandStateCmdSet" id="MyMenuGroup" />
28.        <Icon guid="guidOfficeIcon" id="msotcid1" />
29.        <CommandFlag>DynamicVisibility</CommandFlag>
30.        <Strings>
31.          <CommandName>ChangesVisibility1</CommandName>
32.          <ButtonText>Changes Visibility #1</ButtonText>
33.        </Strings>
34.      </Button>
35.      <Button guid="guidCommandStateCmdSet" id="ChangesVisibility2"
36.        priority="0x0300" type="Button">
37.        <Parent guid="guidCommandStateCmdSet" id="MyMenuGroup" />
38.        <Icon guid="guidOfficeIcon" id="msotcid2" />
39.        <CommandFlag>DynamicVisibility</CommandFlag>
40.        <CommandFlag>DefaultInvisible</CommandFlag>
41.        <CommandFlag></CommandFlag>
```

```
42.          <Strings>
43.            <CommandName>ChangesVisibility2</CommandName>
44.            <ButtonText>Changes Visibility #2</ButtonText>
45.          </Strings>
46.        </Button>
47.
48.        <Button guid="guidCommandStateCmdSet" id="ChangesText"
49.          priority="0x0400" type="Button">
50.          <Parent guid="guidCommandStateCmdSet" id="MyMenuGroup" />
51.          <Icon guid="guidOfficeIcon" id="msotcidPaperStack" />
52.          <CommandFlag>TextChanges</CommandFlag>
53.          <Strings>
54.            <CommandName>ChangesText</CommandName>
55.            <ButtonText>Click count: 0</ButtonText>
56.          </Strings>
57.        </Button>
58.
59.        <Button guid="guidCommandStateCmdSet" id="ChangesEnabledState"
60.          priority="0x0500" type="Button">
61.          <Parent guid="guidCommandStateCmdSet" id="MyMenuGroup" />
62.          <Icon guid="guidOfficeIcon" id="msotcidBlank" />
63.          <Strings>
64.            <CommandName>ChangesEnabledState</CommandName>
65.            <ButtonText>Enabled on even click count</ButtonText>
66.          </Strings>
67.        </Button>
68.      </Buttons>
69.    </Commands>
70.
71.    <Symbols>
72.      <GuidSymbol name="guidCommandStatePkg"
73.        value="{ff5ab687-afbb-46c1-8542-17510542549a}"/>
74.      <GuidSymbol name="guidCommandStateCmdSet"
75.        value="{ab0f163e-cd16-404d-a378-60423d214c32}">
76.        <IDSymbol name="MyMenuGroup" value="0x1020" />
77.        <IDSymbol name="AlwaysVisibleAndEnabled" value="0x0100" />
78.        <IDSymbol name="ChangesVisibility1" value="0x0101" />
79.        <IDSymbol name="ChangesVisibility2" value="0x0102" />
80.        <IDSymbol name="ChangesText" value="0x0103" />
81.        <IDSymbol name="ChangesEnabledState" value="0x0104" />
82.      </GuidSymbol>
83.    </Symbols>
84.
85. </CommandTable>
```

Each menu item representing a command is set up a bit differently than the others. Button elements can define flags determining their behavior in CommandFlag elements as highlighted in the listing. Table 3 describes the behavior defined by the initial settings of Button elements.

Table 3: *Behavior of Button elements*

Button element ID	***Description***

AlwaysVisibleAndEnabled	This element does not define any flags to handle the visibility properties dynamically, so its visibility state is always the default (visible). Even if the package is loaded, its visibility state is never queried.
ChangesVisibility1	By using the DynamicVisibility command flag this item is visible unless the package is loaded. When the package gets sited, the command target is queried for visibility status.
ChangesVisibility2	This element combines the DefaultInvisible flag with DynamicVisibility. As a result, it behaves exactly like ChangesVisibility1 except that its initial state is invisible.
ChangesText	Here the TextChanges command flag is used. While the package is not loaded the string specified in the ButtonText element is used as the label of the command. When the package is loaded Visual Studio asks the package for the label text before displaying the command.
ChangesEnabledState	Similarly to the AlwaysVisibleAndEnabled command this element does not define any flags to handle visibility state dynamically. However, you change the availability of the command programmatically and the appearance of the menu item follows it. This is because as soon as the package gets loaded availability status is queried before displaying the menu item.

The command table defines only the initial visual state of commands and some of them are changed during runtime. Listing 2 contains the source code for the CommandStatePackage class that defines this behavior:

Listing 2: *CommandStatePackage.cs*

```
1. using System;
2. using System.Runtime.InteropServices;
3. using System.ComponentModel.Design;
4. using System.Text;
5. using Microsoft.VisualStudio.Shell.Interop;
6. using Microsoft.VisualStudio.Shell;
7.
8. namespace DeepDiver.CommandState
9. {
10.    [PackageRegistration(UseManagedResourcesOnly = true)]
11.    [InstalledProductRegistration("#110", "#112", "1.0", IconResourceID =
   400)]
12.    [ProvideMenuResource("Menus.ctmenu", 1)]
13.    [Guid(GuidList.guidCommandStatePkgString)]
14.    public sealed class CommandStatePackage : Package
15.    {
```

```
16.     private OleMenuCommandService _CommandService;
17.     private OleMenuCommand _ChangesVisibility1;
18.     private OleMenuCommand _ChangesVisibility2;
19.     private OleMenuCommand _ChangesText;
20.     private OleMenuCommand _ChangesEnabledState;
21.     private int _ClickCount;
22.
23.     protected override void Initialize()
24.     {
25.       base.Initialize();
26.       _CommandService = GetService(typeof(IMenuCommandService)) as
   OleMenuCommandService;
27.       RegisterCommand(CmdIDs.AlwaysVisibleAndEnabled,
   AlwaysVisibleCallback);
28.       _ChangesVisibility1 =
29.         RegisterCommand(CmdIDs.ChangesVisibility1,
   ChangesVisibilityCallback);
30.       _ChangesVisibility2 =
31.         RegisterCommand(CmdIDs.ChangesVisibility2,
   ChangesVisibilityCallback);
32.       _ChangesVisibility2.Visible = false;
33.       _ChangesText =
34.         RegisterCommand(CmdIDs.ChangesText, ChangesTextCallback);
35.       _ChangesEnabledState =
36.         RegisterCommand(CmdIDs.ChangesEnabledState,
   ChangesEnabledStateCallback);
37.     }
38.
39.     private void AlwaysVisibleCallback(object caller, EventArgs args)
40.     {
41.       OutputCommandString("'Always visible and enabled' command
   executed.");
42.     }
43.
44.     private void ChangesVisibilityCallback(object caller, EventArgs
   args)
45.     {
46.       _ChangesVisibility1.Visible = !_ChangesVisibility1.Visible;
47.       _ChangesVisibility2.Visible = !_ChangesVisibility1.Visible;
48.       OutputCommandString("Visibility of buttons #1 and #2 swapped.");
49.     }
50.
51.     private void ChangesTextCallback(object caller, EventArgs args)
52.     {
53.       _ChangesText.Text = String.Format("Click count: {0}",
   ++_ClickCount);
54.       _ChangesEnabledState.Enabled = _ClickCount % 2 == 0;
55.       OutputCommandString("Current click counter is " + _ClickCount);
56.     }
57.
58.     private void ChangesEnabledStateCallback(object caller, EventArgs
   args)
59.     {
60.       OutputCommandString("Command is enabled at click count " +
   _ClickCount);
```

```
61.     }
62.
63.     private OleMenuCommand RegisterCommand(uint id, EventHandler
   callback)
64.     {
65.       if (_CommandService == null) return null;
66.       var menuCommandID = new
   CommandID(GuidList.guidCommandStateCmdSet, (int)id);
67.       var menuItem = new OleMenuCommand(callback, menuCommandID);
68.       _CommandService.AddCommand(menuItem);
69.       return menuItem;
70.     }
71.
72.     private void OutputCommandString(string text)
73.     {
74.       // --- Build the string to write on the debugger and Output
   window.
75.       var outputText = new StringBuilder();
76.       outputText.AppendFormat("CommandStatePackage: {0} ", text);
77.       // --- Get a reference to IVsOutputWindow.
78.       var outputWindow = GetService(typeof(SVsOutputWindow)) as
   IVsOutputWindow;
79.       if (outputWindow == null) return;
80.
81.       // --- Get the window pane for the general output.
82.       var guidGeneral =
   Microsoft.VisualStudio.VSConstants.GUID_OutWindowDebugPane;
83.       IVsOutputWindowPane windowPane;
84.       if (Microsoft.VisualStudio.ErrorHandler.Failed(
85.         outputWindow.GetPane(ref guidGeneral, out windowPane)))
86.       {
87.         return;
88.       }
89.       // --- As the last step, write to the output window pane
90.       windowPane.OutputString(outputText.ToString());
91.       windowPane.Activate();
92.     }
93.   }
94. }
```

The package sets up command bindings in the overridden Initialize methods. Commands use the event handler methods having a Callback suffix. The RegisterCommand helper method carries out the necessary steps to bind a command handler to its logical command. Each command generates a diagnostic message to the Debug pane of the Output window in Visual Studio, OutputCommandString implements this functionality. Output window handling is treated in details in *Chapter 3: Window Management and Tool Windows* so there you will find more explanation about how OutputCommandString method works.

In the FirstLook sample at the beginning of this chapter you could see a MenuCommand instance was used as the command binder, whilst in this example OleMenuCommand instances are used. OleMenuCommand derives from MenuCommand and this kind of object is passed as the sender argument in the event handler methods. Each OleMenuCommand instances represent the

command binders in private fields, and later these are used to change command status. The RegisterCommand method is implemented so that it retrieves the binder object.

AlwaysVisibleCallback simply generates an output message.

You can observe that both **_ChangesVisibility1** and **_ChangesVisibility2** are bound to ChangesVisibilityCallback that is a very simple method swapping the current visibility states of these two commands:

```
1. private void ChangesVisibilityCallback(object caller, EventArgs args)
2. {
3.    _ChangesVisibility1.Visible = !_ChangesVisibility1.Visible;
4.    _ChangesVisibility2.Visible = !_ChangesVisibility1.Visible;
5.    OutputCommandString("Visibility of buttons #1 and #2 swapped.");
6. }
```

The menu binder triggering the event is passed in the caller argument, so you could write the code above like this:

```
1. private void ChangesVisibilityCallback(object caller, EventArgs args)
2. {
3.    var command = caller as OleMenuCommand;
4.    if (command == null || (command.CommandID.ID !=
   CmdIDs.ChangesVisibility1 &&
5.       command.CommandID.ID != CmdIDs.ChangesVisibility2))
6.    {
7.       return;
8.    }
9.    _ChangesVisibility1.Visible = command.CommandID.ID ==
   CmdIDs.ChangesVisibility2;
10.    _ChangesVisibility2.Visible = !_ChangesVisibility1.Visible;
11.    OutputCommandString("Visibility of buttons #1 and #2 swapped.");
12. }
```

You need an additional step to make the swapping commands work correctly: you have to initialize the visibility state of **_ChangesVisibility2** to hidden in the **Initialize** method:

```
1. _ChangesVisibility1 =
2.    RegisterCommand(CmdIDs.ChangesVisibility1, ChangesVisibilityCallback);
3. _ChangesVisibility2 =
4.    RegisterCommand(CmdIDs.ChangesVisibility2, ChangesVisibilityCallback);
5. _ChangesVisibility2.Visible = false;
```

If you omit the last line, the package seems working properly. But click on the Always visible and enabled item first and then drop down the Tools menu. You will observe some unexpected thing as Figure 4 shows.

Figure 4: *Both Changes Visibility commands are visible!*

What's wrong with the command logic? If you invoke the Always visible and enabled item first, it causes the package to load. The Initialize method runs and creates the binders for the Changes Visibility items. By default these binders are created with their Visibility property set to true. Because the package is loaded, the IDE uses the package to query the visibility status when you drop down the **Tools** menu and this time both commands are visible.

The ChangesText command also uses a very simple logic:

```
1. private void ChangesTextCallback(object caller, EventArgs args)
2. {
3.    _ChangesText.Text = String.Format("Click count: {0}", ++_ClickCount);
4.    _ChangesEnabledState.Enabled = _ClickCount%2 == 0;
5.    OutputCommandString("Current click counter is " + _ClickCount);
6. }
```

It uses the _ClickCount member variable to count the number of clicks and updates the label of the command through the Text property. This command controls the availability of the ChangesEnabledState command. Try to omit the TextChanges flag from the command table and run the package. You can use the ChangesText command — as you click the availability status of ChangesEnabledState switches between enabled and hidden — but the count of click is not displayed.

You do not need an event handler method for the ChangesEnabledState command, because its state is handled by the ChangesTextCallback method, but in this example the handler is used to display a trace message.

Commands and Visibility

When you move around in Visual Studio you can see that a few toolbars, menus and menu items are visible or invisible depending on the context you are in. For example, the Project and Debug menu items cannot be seen while there is no project open. Similarly, you cannot reach the Team menu unless you connect to a Team Server.

Visual Studio shows or hides commands according to the current context. Please note, showing or hiding commands means to make them visible or invisible. If a command is visible,

availability state also influences the appearance — disabled commands are dimmed — but availability is totally independent of visibility.

Commands can be defined in different locations — they logically belong to one of the following entities:

- **The VS IDE environment.** All commands defined by the Visual Studio IDE are always visible.
- **Packages registered in the IDE.** Packages may define commands and decide whether to show or hide a certain command. Please remember that Visual Studio IDE is a combination of the core IDE (Shell) and the packages registered with the IDE. When you install Visual Studio many packages also get registered.
- **The active project.** In Visual Studio you work with solutions that contain projects. There can be only one active project (the currently selected project) determined by either the active editor file or the item selected in the Solution Explorer. All the other projects (if there are any others in the solution) are inactive. Only the commands belonging to the active project are visible.
- **Active editor.** When you edit a source file in the text editor or a form with the form editor or creating a class diagram, you work with an editor. If you have documents open you always have one active editor (and might have a set of inactive ones). The active editor can define its own commands (think about the Windows Forms editor). The commands defined by the active editors are visible, but any commands of other (inactive) editors are hidden. There are editors that support a variety of file types (e.g. the Image Editor). There can be commands that are available for one file type but not for another file type. For instance, you can set the transparent pixels for an icon but not for a .BMP file in the image editor. It is always the responsibility of the editor to show or hide a command depending on the file type currently used.
- **Tool windows.** Packages can register tool windows and tool windows can have their own commands. When you register a package providing a tool window with Visual Studio, the related commands are visible by default. Of course, to make them appear on the screen you must show up the tool window.

Any time when you are in a concrete Visual Studio context all visible commands from the locations above are merged. So, the union of IDE environment commands, package commands, active project commands, active editor commands (depending on the file type) and visible tool window commands determine the set of visible commands.

You may feel that something is missing. As it has been treated here commands are visible or hidden for a package, editor or tool window. An example was also mentioned earlier that the Project and Debug menus are not available unless a file or a project is opened. How does this fit into the picture? How does Visual Studio handle this thing? How can a package make a command visible or hidden depending on whether a project is open or closed?

Visual Studio allows further control about command visibility. The IDE defines visibility contexts and commands can be bound to them. The most frequently used contexts are summarized in Table 4.

Table 4: *User interface context definitions*

Context name	*Description*
NoSolution	No solution is opened in the IDE (the Solution Explorer tool window is empty).
SolutionExists	There is an existing solution opened in the IDE. This can be an empty solution, a solution created by opening a single file, a solution with one or more project. Commands bound to this context are visible if you can see a root solution in the Solution Explorer.
EmptySolution	There is a solution opened in the IDE and this solution is empty (does not contain any item).
SolutionHasSingleProject	There is a solution opened in the IDE and this solution contains exactly one project of any type.
SolutionHasMultipleProjects	There is a solution opened in the IDE and this solution contains multiple projects loaded.
SolutionBuilding	The current solution or any of its projects is being built. The IDE stays in this context unless the build process finishes.
Debugging	The IDE is in Debug mode: its debugger is attached to a running process.
DesignMode	The IDE is in design mode (not in Debug mode).
FullScreenMode	The IDE is running in Full screen mode (it can be activated with the View | Full Screen function.
Dragging	Currently there is an active drag/drop operation in the Visual Studio IDE.

When a command is bound to a set of visibility contexts it is visible only when Visual Studio IDE is in one of the contexts bound to it. You can define your own contexts and notify Visual Studio about entering into the specified context. Commands can be also bound to your custom contexts. Later in this chapter you are going to see a few code samples about handling visibility contexts.

The Visual Studio Command Table

You have already seen the Visual Studio Command Table in a few examples in the previous and also in this chapter. The command table is defined in the .vsct file of the package project and is compiled to a binary resource which is embedded into the package infrastructure resources

during the build process. When deploying the package the command table is merged into the menus and toolbars of the IDE.

In this section you dive into the details of the command table structure and create a few more samples to understand the basic concepts. Of course, all nitty-gritty details cannot be handled, but the information here is enough that you can to discover the advanced aspects of the command table by yourself.

Note: The .vsct file was introduced as a new format for the command table substituting the .ctc file in the previous versions of Visual Studio SDK. Visual Studio 2005 SDK used a textual format (using files with .ctc extension). Editing and understanding a .ctc file was not a simple task. With the release of Visual Studio 2008 SDK Microsoft created the new XML-based file format with the .vsct extension and a new compiler to produce the binary .cto format.

The main advantage of using the .vsct file is that it is editable just like other XML files and so has all the great XML editing features like automatic generation of closing tags or IntelliSense based on the VSCT XML schema. Due to the fact XML format has so many advantage the old .ctc format has been deprecated. Microsoft recommends using the VSCT compiler to generate the .cto files, although CTC is still supported.

VSCT File Structure

When describing the elements of an XML file one alternative is to provide an XSD schema. It tells a lot about the syntax and the semantics and is said to be readable by either a human or a machine. There is no doubt that machines can easily understand an XSD, but for understanding concepts behind a certain XML file XSD is not the best way. Instead, here you will be shown small examples to help you understand the structure of the command file.

The root element of this structure is the CommandTable element:

```
1. <?xml version="1.0" encoding="utf-8"?>
2. <CommandTable
3.   xmlns="http://schemas.microsoft.com/VisualStudio/2005-10-
   18/CommandTable"
4.   xmlns:xs="http://www.w3.org/2001/XMLSchema">
5.   <!-- Content of the command table -->
6. </CommandTable>
```

The elements of the command table are defined by the http://schemas.microsoft.com/VisualStudio/2005-10-18/CommandTable namespace. You will rarely create the startup state of a command table by hand, because the VSPackage wizard does it for you when you ask it to create an initial command or tool window. When you create the command table by hand, do not forget about specifying this namespace otherwise your command table will not compile. In the code examples later the namespace information will be omitted just for the sake of brevity.

The CommandTable itself uses a few child elements to define the content of the table:

```
<CommandTable xmlns="..." xmlns:xs="...">
  <Extern/>
  <Include/>
  <Define/>
  <Commands/>
  <CommandPlacements>
  <VisibilityConstraints/>
  <KeyBindings/>
  <UsedCommands/>
   <Symbols/>
</CommandTable>
```

For a developer new to Visual Studio Extensibility, the Extern, Commands and Symbols elements are the most important and, of course, the most frequently used ones. When the VSPackage wizard creates a package with a simple menu command, you get something similar (the wizard injects a lot of comments to explain the content but here those are omitted):

```
<?xml version="1.0" encoding="utf-8"?>
<CommandTable xmlns="..." xmlns:xs="...">
  <Extern href="stdidcmd.h"/>
  <Extern href="vsshlids.h"/>
  <Extern href="msobtnid.h"/>

  <Commands package="guidSimpleCommandPkg">
    <Groups>
      <Group guid="guidSimpleCommandCmdSet" id="MyMenuGroup"
priority="0x0600">
        <Parent guid="guidSHLMainMenu" id="IDM_VS_MENU_TOOLS"/>
      </Group>
    </Groups>

    <Buttons>
      <Button guid="guidSimpleCommandCmdSet" id="cmdidMyFirstCommand"
        priority="0x0100" type="Button">
        <Parent guid="guidSimpleCommandCmdSet" id="MyMenuGroup" />
        <Icon guid="guidImages" id="bmpPic1" />
        <Strings>
          <CommandName>cmdidMyFirstCommand</CommandName>
          <ButtonText>My First Command</ButtonText>
        </Strings>
      </Button>
    </Buttons>

    <Bitmaps>
      <Bitmap guid="guidImages" href="Resources\Images_32bit.bmp"
        usedList="bmpPic1, bmpPic2, bmpPicSearch, bmpPicX,
bmpPicArrows"/>
    </Bitmaps>
  </Commands>

  <Symbols>
```

```
33.     <GuidSymbol name="guidSimpleCommandPkg"
34.       value="{2291da24-92e5-4ea4-bdb7-72a9b5ac7d59}" />
35.     <GuidSymbol name="guidSimpleCommandCmdSet"
36.       value="{a982b107-4ad4-437e-b2bc-cdf2708aa376}">
37.       <IDSymbol name="MyMenuGroup" value="0x1020" />
38.       <IDSymbol name="cmdidMyFirstCommand" value="0x0100" />
39.     </GuidSymbol>
40.     <GuidSymbol name="guidImages" value="{5c3faf04-8190-48c4-a6e9-
   71f04f1848e5}" >
41.       <IDSymbol name="bmpPic1" value="1" />
42.       <IDSymbol name="bmpPic2" value="2" />
43.       <IDSymbol name="bmpPicSearch" value="3" />
44.       <IDSymbol name="bmpPicX" value="4" />
45.       <IDSymbol name="bmpPicArrows" value="5" />
46.     </GuidSymbol>
47.   </Symbols>
48.
49. </CommandTable>
```

You already know that commands and other elements of the command table objects are uniquely identified by a compound key composed from a GUID and a 32-bit unsigned integer. Command table elements have references to each other and the unique identifiers are used to describe this reference. The parts of the key are defined by the guid and id attributes, respectively.

In a .vsct file you generally use most element identifiers at least twice: once for identifying an object and at least once for referencing it. If you would use the GUID and uint values directly those would not provide the best readability. If you had to type the same GUID value several times it would make the hand-edited .vsct file very fragile, as humans are not very good in typing, comparing and checking GUID values.

The Symbols section is a central place in the command table file where you can define the identifiers to be used in the other parts of the .vsct file. You can use the GuidSymbol element to define the logical container represented by the GUID and the nested IDSymbol elements to provide (optional) identifiers within the logical container. The name and the value attribute of these elements do exactly what you expect: associate the symbol name with its value. The GUID and uint values of the compound key are checked together, the VSCT compiler takes care of that the uint value should be defined by an IDSymbol nested in the corresponding GuidSymbol elements value.

In the example above there are three GUID containers defined. The first is an empty container (with the symbolic name of guidSimpleCommandPkg) the last two contain a few nested ID elements. The identifiers defined here are referenced in the upper parts of the command table definition as in the following extract:

```
1. <Commands package="guidSimpleCommandPkg">
2.   <Groups>
3.     <Group guid="guidSimpleCommandCmdSet" id="MyMenuGroup"
   priority="0x0600">
4.       <!-- ... -->
```

```
    </Group>
  </Groups>

  <Buttons>
    <Button guid="guidSimpleCommandCmdSet" id="cmdidMyFirstCommand"
       priority="0x0100" type="Button">
       <!-- ... -->
    </Button>
  </Buttons>

  <Bitmaps>
    <Bitmap guid="guidImages" href="Resources\Images_32bit.bmp"
       usedList="bmpPic1, bmpPic2, bmpPicSearch, bmpPicX,
  bmpPicArrows"/>
  </Bitmaps>
</Commands>
```

Objects defined by the Visual Studio IDE are frequently referenced in the command table. For example, when you add a menu item to the Tools menu, somehow you must refer to the Tools menu so that the VSCT compiler can understand it. Of course, the Tools menu is identified on the same way as any other command table elements:

```
<Group guid="guidSimpleCommandCmdSet" id="MyMenuGroup" priority="0x0600">
  <Parent guid="guidSHLMainMenu" id="IDM_VS_MENU_TOOLS"/>
</Group>
```

Here the Parent element defines the location where the logical command group declared by the Group element should be placed. The guidSHLMainMenu defines the logical container of the main menu bar of the IDE and the IDM_VS_MENU_TOOLS is the identifier of the Tools menu within the logical container. As you guess, there are thousands of GUIDs and IDs related to Visual Studio IDE elements.

The Extern element is the key to access them:

```
<?xml version="1.0" encoding="utf-8"?>
<CommandTable xmlns="..." xmlns:xs="...">
  <Extern href="stdidcmd.h"/>
  <Extern href="vsshlids.h"/>
  <Extern href="msobtnid.h"/>
  <!-- ... -->
</CommandTable>
```

When the VSCT compiler processes the .vsct file it can run a preprocessor on its content. This preprocessor is by default the C++ preprocessor that allows including files defining symbols and macros. The Extern element is one of the elements directing the preprocessor. The href attribute names the file with .h (standard C++ header file) extension. After the preprocess phase the symbols in the #define pragmas and in the macro definitions of referenced files can be used anywhere just like GUID and ID values defined in the Symbols section of the command table.

The files named in the href attributes are searched in the include path passed to the VSCT compiler. This is the VisualStudioIntegration\Common\Inc folder under the root installation folder of VS SDK by default when you create your package with the VSPackage wizard. The wizard also creates three Extern elements. Table 5 describes the role of the referenced files.

Table 5: *External file references in the VSCT file*

File	*Content*
stdidcmd.h	This file represents the command IDs for all commands exposed by Visual Studio. IDs include the visible (or hidden) menu command IDs prefixed with cmdid, standard editor commands with ECMD_ prefix and a few others.
vsshlids.h	This file collects command IDs for the menus provided by the Visual Studio Shell.
msobtnid.h	This file represents IDs of the standard Microsoft Office commands (many of them, like Cut, Copy and Paste are also used in VS IDE).

If you look into the header files, you can find the definitions for guidSHLMainMenu and IDM_VS_MENU_TOOLS in the vsshlids.h file. The preprocessor understands both the DEFINE_GUID macro and the #define pragma.

```
1. ...
2. DEFINE GUID (guidSHLMainMenu,
3.   0xd309f791, 0x903f, 0x11d0, 0x9e, 0xfc, 0x00, 0xa0, 0xc9, 0x11, 0x00,
   0x4f);
4. ...
5. #define IDM_VS_MENU_TOOLS 0x0085
6. ...
```

The DEFINE_GUID macro here sets the guidSHLMainMenu to the value of {d309f791-903f -11d0-9efc-00a0c911004f}. Any time you know a GUID value but you do not know which symbolic name is used for that value, search the VisualStudioIntegration\Common\Inc folder to find if the GUID value is defined there or not. If necessary add a new Extern directive to the command table.

Command Definitions

No doubt, the most important element of the .vsct file is Commands. Its role is to define commands, their initial layout and behavior:

```
1. <Commands package="...">
2.   <Groups/>
3.   <Menus/>
4.   <Buttons/>
```

```
5.    <Combos/>
6.    <Bitmaps/>
7. </Commands>
```

Any command in the IDE must belong to the IDE itself or to a package. A package assembly can implement one or more packages. To assign a command to the appropriate (owning) VSPackage, the package attribute of the Commands element must name the GUID of the corresponding package. Generally you have only one package in one assembly so you put the package ID generated by the VSPackage wizard into this attribute:

```
1. <Commands package="guidSimpleCommandPkg">
2.    <!-- ... -->
3. </Commands>
```

As you obviously guessed Visual Studio uses the package ID here to find out which package is to be loaded to handle a command. There are several types of nested child elements in Commands, and each of them has a specific role.

The Groups element defines so-called command groups using a nested Group element. A command group is a logical set of related commands that generally stand together. For example, the Build Solution, Rebuild Solution and Clean Solution menu items form a logical group: they stand together in the Build menu and in the context menu of a Solution item as shown in Figure 5.

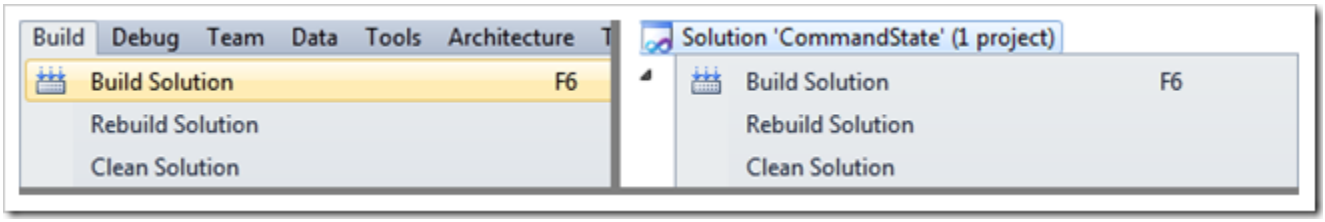


Figure 5: *Visual effect of a Group element*

Instead of putting individual commands to an existing menu you can place them into a group and that group can be placed to the menu. Even if you have only one element to add to a menu, you must create a Group element for it and put the item into this group. Visual Studio menu groups also have related symbolic IDs in the header files, so you can add a command to one of the predefined groups. Each command in the logical set formed by the group appears in the related menu.

You can nest Menu elements as children of Menus to define a single menu. A single menu is a placeholder for items and of course, you also can put submenu items there.

Menus can have different appearance and behavior. The most frequent forms are:

- **Standard menus** (for example the File, Edit and View menus in the IDE).
- **Context menus**: They show up when right-clicking on the object they provide a command context for.
- **Toolbars**: standard toolbars where commands are organized in rows with icons — and/or text labels — representing them.

The Buttons element is a container for nested Button elements. A Button represents a piece of user interface the user can interact with. The name is a bit confusing, because when saying the word button you generally associate it with the pushbutton. Here, Button relates to a menu or toolbar item. In .vsct you can define a few types of Button items:

- **Standard buttons** represent a simple menu items which execute a command.
- **Menu buttons** display a submenu.
- **Dropdown buttons** allow functions like **Undo** and **Redo** on the main toolbar of the IDE.
- **"Swatch" buttons** display color choices such as those in a font color dialog.

Not surprisingly Combos is a container for Combo elements. A Combo defines a set of commands that appear in a combo box.

Toolbars and menus would be poor without icons helping the user to associate a small image with the function. The Bitmaps section allows defining the visual elements (icons) used in menus. The section describes Bitmap elements that are uniquely identified. A bitmap can come from an external file or from a package resource.

When referring to bitmaps, you can use a few formats like .bmp, .gif, .png. Right now you cannot use all formats in the same way. For example, you may have problems with the 32-bit .bmp formats (alpha channel information for partial transparency). If you use 120 DPI in your display settings, the original 16x16 pixel images will be stretched to a 20x20 pixel size. If you use .png format, stretching is smooth without disturbing artifacts. In some cases only a few formats are accepted: for example for icons representing tool windows on their tabs, .png format is not welcomed (no icon appears), only 24-bit .bmp with fuchsia as the transparent color.

If you have problems with icon images, try a few formats and you can generally find the one expected by VS IDE.

Working with Commands and Bitmaps

To successfully define commands appearing in the VS IDE menus you have to use at least one Group and one Button element. If you want to add icons, you need at least one additional Bitmap element. If you prefer your own menu instead of inserting a command group in the middle of a Visual Studio menu, at least one Menu element should be defined. To establish the layout and the relation among the elements you have to look behind their attributes and child elements.

Menu, Group, Button and other elements have common attributes and child element with very similar semantics summarized in Table 6.

Table 6: *Common attributes of command table elements*

Attribute	*Description*
guid	The GUID part of the element identifier. This attribute is required.
id	The uint part of the element identifier. This attribute is required.
priority	An optional numeric value determining the order of the element. The lower this value is the closer the element is to the position of the first element. Visual Studio IDE sorts elements according to their priority. However, you cannot know the priority value of all other elements, so exact position is not guaranteed.
type	An optional stereotype of the element determining its layout and behavior. The Group element does not have this attribute (as it is a behavior-less element). The possible values and the meaning of this attribute changes according to the element type.

There are child elements connecting UI elements together and adding properties which fine-tune layout and behavior. Except of Bitmap all other types of Commands children have the child elements in Table 7.

Table 7: *Common child elements*

Child element	*Description*
Parent	Optional parent of the element. A command element can be attached to one or more menu items. Here you can define zero or one Parent element. If you want to attach a command to more than one menu item, the CommandPlacement element is for you. The Parent element has the guid and id attribute to uniquely name the parent the element belongs to. For a Button the parent element must refer to a command Group. For a Group, the parent must refer to a Menu.
Annotation	This element allows adding optional comment information to the element. Annotations can have either simple text or a nested structure.

Table 8 summarizes child elements extending the layout and behavior of Menu, Button and Combo.

Table 8: *Child elements influencing behavior*

Child element	*Description*
CommandFlag	This child element can be applied zero, one or more times to its parent element. As the name indicates it sets flags influencing the layout and behavior of its parent. The VSCT Schema Reference enumerates all the command flags that can be applied to a particular element. Several flags have effect only when combining with other flags. For example, applying the DynamicVisibility and DefaultInvisible flags causes a menu item to be hidden at Visual Studio startup time.
Strings	This element is a container for children representing string information of UI elements. At least the ButtonText child of Strings should be applied to display a caption for the element. Possible child elements include ButtonText, ToolTipText, MenuText, CommandName and others. For details see the VSCT Schema reference.
Icon	Available only for Button. Optionally defines the icon associated with the button.

The Bitmaps section of Commands provides a place to define bitmaps you want to use in menu and toolbar items. Each definition must be in a Bitmap element. The concept of Bitmap covers one bitmap with a physical size of 16x16 pixels or a bitmap strip that has 16x*N* pixels where *N* is a multiple of 16 (number of pixels in a column). Bitmap handles the simple 16x16 pixel bitmap as a bitmap strip, where *N* equals 1. A Bitmap is identified with a GUID that must be a separate GUID from the package ID, command set IDs, and so on. When you refer to a certain item in the bitmap strip you do it by using a one-based index for the bitmap in the strip.

Just as in case of commands, menu IDs you can use the Symbols section of the command table to define IDs related to a bitmap. For example when you create a new Tool window package With the VSPackage wizard, the following entries are related to bitmaps:

```
1. <?xml version="1.0" encoding="utf-8"?>
2. <CommandTable xmlns="..." xmlns:xs="...">
3.   <!-- Extern elements -->
4.   <Commands package="...">
5.     <!-- Menus, Groups and Buttons -->
6.     <Bitmaps>
7.       <Bitmap guid="guidImages" href="Resources\Images_32bit.bmp"
8.         usedList="bmpPic1, bmpPic2, bmpPicSearch, bmpPicX,
   bmpPicArrows"/>
9.     </Bitmaps>
10.   </Commands>
11.   <Symbols>
12.     <GuidSymbol name="guidImages" value="{...}" >
```

```
13.         <IDSymbol name="bmpPic1" value="1" />
14.         <IDSymbol name="bmpPic2" value="2" />
15.         <IDSymbol name="bmpPicSearch" value="3" />
16.         <IDSymbol name="bmpPicX" value="4" />
17.         <IDSymbol name="bmpPicArrows" value="5" />
18.       </GuidSymbol>
19.    </Symbols>
20. </CommandTable
```

The Symbols section defines a GUID used only by the Bitmap definition (guidImages) and symbolic names for the one-based bitmap indexes within the strip. In the example above bmpPicSearch is the third picture within the strip.

The Bitmap element's href attribute names the file where the bitmap strip can be found. This folder is relative to the location of the .vsct file. The usedList attribute enumerates the symbolic indices that can be used to identify bitmaps in the strip. If you want to set a button icon to the third image in the strip, you should use the following Icon child element in a Button:

```
1. <Button ...="">
2.    <Icon guid="guidImages" id="bmpPicSearch" >
3. </Button>
```

You are allowed to use stocked icons. For example, the CommandState sample uses such icons:

```
1. <Icon guid="guidOfficeIcon" id="msotcidClock" />
```

The guidOfficeIcon symbol is defined in the vsshlids.h file and is a logical container for office icons. The msotcidClock and thousands of other IDs can be found in the msobtnid.h file. If you are looking for stocked icons, definitely this is the place you should search for them. The msobtnid.h contains comments that help you to identify icons you are looking for.

Basic VSCT Samples

By now you have an understanding of the essential VSCT concepts, moreover, you have seen a few samples. In order to illustrate the structure and usage of command table element you are going to build a few more samples. You start from a very simple package created with the help of VSPackage wizard using a simple menu command. The package does not have any real function, because it is just a container for menu resources, the package class definition is quite simple as Listing 3 shows.

Listing 3: *The BasicVSCTSamplePackage definition*

```
1. // --- BasicVSCTSamplePackage.cs
2.
3. using System.Runtime.InteropServices;
4. using Microsoft.VisualStudio.Shell;
5.
6. namespace DeepDiver.BasicVSCTSample
```

```
7. {
8.   [PackageRegistration(UseManagedResourcesOnly = true)]
9.   [InstalledProductRegistration(false, "#110", "#112", "1.0",
   IconResourceID = 400)]
10.   [ProvideMenuResource("Menus.ctmenu", 1)]
11.   [Guid(GuidList.guidBasicVSCTSamplePkgString)]
12.   public sealed class BasicVSCTSamplePackage : Package
13.   {
14.   }
15. }
16.
17. // --- Guids.cs
18.
19. namespace DeepDiver.BasicVSCTSample
20. {
21.   static class GuidList
22.   {
23.     public const string guidBasicVSCTSamplePkgString =
24.       "41fe6025-c174-4d02-af4f-ea948a272830";
25.   }
26. }
```

You are going to work only with the BasicVSCTSample.vsct file. You start from simple scenario and transform the .vsct file to discover new usage opportunities. You can use the VSPackage wizard to create the sample, or open the BasicVSCTSample project downloaded from the book's website.

Creating a Main Menu Level Command

The VSPackage wizard puts menu items in a fixed place: in the Tools menu if you create a simple command or into the View menu if you create a simple tool window. However, when you create a package you often would like to put your own package-specific menu in the main menu bar of Visual Studio. Now you are going to learn how to do that.

You create a new main level menu VSCTSample with two menu command items. This task requires the following steps:

- **Step1**: Create a GuidSymbol element with children for the symbols you are going to use. Nest this element in the Symbols section.
- **Step 2**: Create a Menu element representing the main level menu. Put this item into the Menus section and set up the Parent of this menu to a main menu level menu group.
- **Step 3**: Create a Group element representing the logical group holding the menu command items. Set the Parent of this group to the Menu created in the previous step.
- **Step 4**: Create two Button elements representing the commands and set their Parent to the Group created previously.

Following the steps the .vsct file looks like in Listing 4.

Listing 4: *Creating main menu commands*

VS 2010 Package Development

```
1. <?xml version="1.0" encoding="utf-8"?>
2. <CommandTable xmlns="http://schemas.microsoft.com/VisualStudio/2005-10-
   18/CommandTable"
3.   xmlns:xs="http://www.w3.org/2001/XMLSchema">
4.   <Extern href="stdidcmd.h"/>
5.   <Extern href="vsshlids.h"/>
6.   <Extern href="msobtnid.h"/>
7.
8.   <Commands package="guidBasicVSCTSamplePkg">
9.     <Menus>
10.      <Menu guid="guidBasicVSCTSampleCmdSet" id="TopLevelMenu"
   priority="0x100"
11.        type="Menu">
12.        <Parent guid="guidSHLMainMenu" id="IDG_VS_MM_BUILDDEBUGRUN" />
13.        <Strings>
14.          <ButtonText>VSCTSample</ButtonText>
15.          <CommandName>VSCTSample</CommandName>
16.        </Strings>
17.      </Menu>
18.    </Menus>
19.
20.    <Groups>
21.      <Group guid="guidBasicVSCTSampleCmdSet" id="TopLevelMenuGroup"
22.        priority="0x0600">
23.        <Parent guid="guidBasicVSCTSampleCmdSet" id="TopLevelMenu"/>
24.      </Group>
25.    </Groups>
26.
27.    <Buttons>
28.      <Button guid="guidBasicVSCTSampleCmdSet" id="FirstCommand"
   priority="0x0100"
29.        type="Button">
30.        <Parent guid="guidBasicVSCTSampleCmdSet" id="TopLevelMenuGroup"
   />
31.        <Strings>
32.          <CommandName>FirstCommand</CommandName>
33.          <ButtonText>First Command</ButtonText>
34.        </Strings>
35.      </Button>
36.      <Button guid="guidBasicVSCTSampleCmdSet" id="SecondCommand"
37.        priority="0x0101" type="Button">
38.        <Parent guid="guidBasicVSCTSampleCmdSet" id="TopLevelMenuGroup"
   />
39.        <Strings>
40.          <CommandName>SecondCommand</CommandName>
41.          <ButtonText>Second Command</ButtonText>
42.        </Strings>
43.      </Button>
44.    </Buttons>
45.
46.  </Commands>
47.
48.  <Symbols>
49.    <GuidSymbol name="guidBasicVSCTSamplePkg"
50.      value="{41fe6025-c174-4d02-af4f-ea948a272830}" />
```

```
51.      <GuidSymbol name="guidBasicVSCTSampleCmdSet"
52.        value="{234580c4-8a2c-4ae1-8e4f-5bc708b188fe}">
53.        <IDSymbol name="TopLevelMenu" value="0x0100" />
54.        <IDSymbol name="TopLevelMenuGroup" value="0x0200" />
55.        <IDSymbol name="FirstCommand" value="0x0300" />
56.        <IDSymbol name="SecondCommand" value="0x0301" />
57.      </GuidSymbol>
58.    </Symbols>
59.
60. </CommandTable>
```

As you see, you have to write quite a lot for such a simple task. In the menu definition the IDs used in the Parent elements are highlighted. If you build and run your package, you can discover the new menu located just after the View menu group and before the Tools menu as Figure 6 shows.

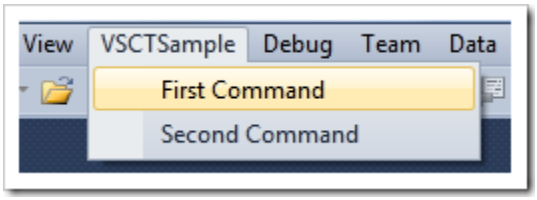


Figure 6: *New main menu item*

When you open a solution the Build menu gets visible and VSCTSample menu goes between Build and Debug as it can be seen in Figure 7.

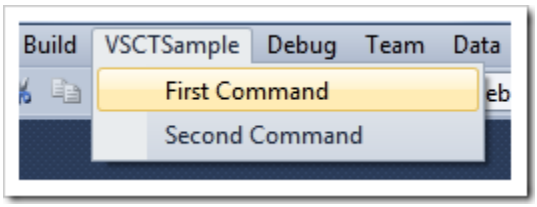


Figure 7: *VSCTSample is located between Build and Debug*

This is due to the Parent definition of Menu and the priority you assigned to it:

```
1. <Menu guid="guidBasicVSCTSampleCmdSet" id="TopLevelMenu" priority="0x100"
   type="Menu">
2.   <Parent guid="guidSHLMainMenu" id="IDG_VS_MM_BUILDDEBUGRUN" />
3.   <Strings>
4.     <ButtonText>VSCTSample</ButtonText>
5.     <CommandName>VSCTSample</CommandName>
6.   </Strings>
7. </Menu>
```

The IDG_VS_MM_BUILDDEBUGRUN is the identifier of a logical group representing the Build and Debug menus on the main VS IDE menu bar. The priority value of 0x100 sets our menu to be shown before the Debug menu. If you change priority to 0x700 (and rebuild the package and run) VSCTSample moves after the Debug menu, as Figure 8 shows.

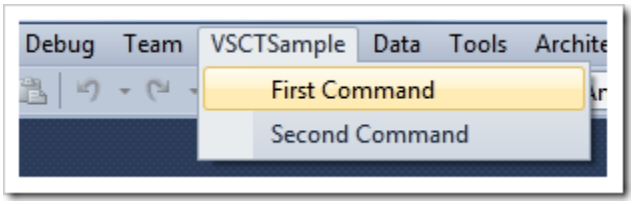


Figure 8: *Changing the location of VSCTSample item*

Looking into the Button definition you may say there is no need for a Group definition: let's set the Parent attribute of Buttons directly to the Menu item like in the following example:

```
1. <Button ...="">
2.   <Parent guid=" guidBasicVSCTSampleCmdSet " id="TopLevelMenu" />
3.   <!-- ... -->
4. </Button>
```

When running the package, no VSCTSample menu will appear. The cause of this phenomenon is the fact that buttons cannot have menus as parents. They must have command groups as parents to be displayed in a menu. Because your VSCTSample menu does not have any child, it is not displayed.

Separating Command Groups in a Menu

As it's been treated, a command group is a logical container for commands belonging together. This kind of grouping also can be used for visual effects. If you put more than one command group in a menu, a separator is created to visually emphasize the separation of command groups. Let's add two more commands with a separate group to the .vsct file you've created above:

- **Step 1**: Add new symbols representing the two new buttons and the group for new commands.
- **Step 2**: Create new Group definition
- **Step 3**: Define two new Button elements parented in the newly created Group.

After the third step your new .vsct file looks like in Listing 5. Please note, only new parts added in the steps above are indicated in this listing:

Listing 5: *Separating Command Groups in a Menu*

```
1. <?xml version="1.0" encoding="utf-8"?>
2. <CommandTable xmlns="..." xmlns:xs="...">
3.   <!-- Extern elements -->
4.   <Commands package="...">
5.     <!-- Menus section unchanged -->
6.
7.     <Groups>
8.       <!-- New group added -->
9.       <Group guid="guidBasicVSCTSampleCmdSet" id="TopLevelMenuGroup2"
10.         priority="0x0600">
11.         <Parent guid="guidBasicVSCTSampleCmdSet" id="TopLevelMenu"/>
12.       </Group>
13.     </Groups>
14.
15.     <Buttons>
16.       <!-- New buttons added -->
17.       <Button guid="guidBasicVSCTSampleCmdSet" id="ThirdCommand"
   priority="0x0100"
18.         type="Button">
19.         <Parent guid="guidBasicVSCTSampleCmdSet"
   id="TopLevelMenuGroup2" />
20.         <Strings>
21.           <CommandName>ThirdCommand</CommandName>
22.           <ButtonText>Third Command</ButtonText>
23.         </Strings>
24.       </Button>
25.       <Button guid="guidBasicVSCTSampleCmdSet" id="FourthCommand"
26.         priority="0x0101" type="Button">
27.         <Parent guid="guidBasicVSCTSampleCmdSet"
   id="TopLevelMenuGroup2" />
28.         <Strings>
29.           <CommandName>FourthCommand</CommandName>
30.           <ButtonText>Fourth Command</ButtonText>
31.         </Strings>
32.       </Button>
33.     </Buttons>
34.   </Commands>
35.
36.   <Symbols>
37.     <GuidSymbol name="guidHowToPackageCmdSet" value="{...}" >
38.       <!-- New IDSymbols added -->
39.       <IDSymbol name="TopLevelMenuGroup2" value="0x0201" />
40.       <IDSymbol name="ThirdCommand" value="0x0302" />
41.       <IDSymbol name="FourthCommand" value="0x0303" />
42.     </GuidSymbol>
43.     <!-- Other Guids for the package -->
44.   </Symbols>
45.
46. </CommandTable>
```

Running the package after these modifications you can see the VSCTSample menu where the two command groups are separated, as shown in Figure 9.

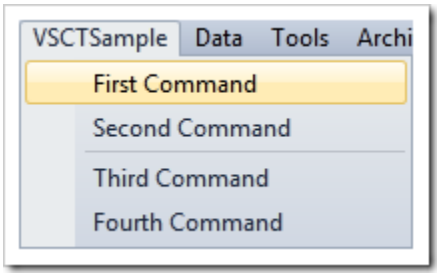


Figure 9: *Separating commands into groups*

Adding Icons to the Menu Items

If you want to add icons to the menu items representing a command, you have to define a Bitmap node and assign it to the Button elements with their Icon property:

- **Step 1**: Create IDs for the bitmap strip and each individual icon in the strip.
- **Step 2**: Create the Bitmap element and set it up to use the icons in the strip.
- **Step 3**: Add Icon properties to the buttons.

After this change the .vsct file changes as indicated in Listing 6.

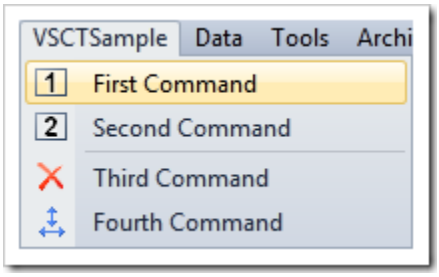
Listing 6: *Adding Icons to the Buttons*

```
1. <?xml version="1.0" encoding="utf-8"?>
2. <CommandTable xmlns="..." xmlns:xs="...">
3.   <!-- Extern elements -->
4.   <Commands package="...">
5.     <!-- Menus section unchanged -->
6.     <!-- Groups section unchanged -->
7.     <Buttons>
8.       <Button guid="guidBasicVSCTSampleCmdSet" id="FirstCommand"
   priority="0x0100"
9.         type="Button">
10.         <!-- Icon added, other children unchanged -->
11.         <Icon guid="guidImages" id="bmpPic1" />
12.       </Button>
13.       <Button guid="guidBasicVSCTSampleCmdSet" id="SecondCommand"
14.         priority="0x0101" type="Button">
15.         <!-- Icon added, other children unchanged -->
16.         <Icon guid="guidImages" id="bmpPic2" />
17.       </Button>
18.       <Button guid="guidBasicVSCTSampleCmdSet" id="ThirdCommand"
   priority="0x0100"
19.         type="Button">
20.         <!-- Icon added, other children unchanged -->
21.         <Icon guid="guidImages" id="bmpPicX" />
22.       </Button>
23.       <Button guid="guidBasicVSCTSampleCmdSet" id="FourthCommand"
24.         priority="0x0101" type="Button">
25.         <!-- Icon added, other children unchanged -->
```

```
26.          <Icon guid="guidImages" id="bmpPicArrows" />
27.        </Button>
28.      </Buttons>
29.
30.      <Bitmaps>
31.        <!-- A new Bitmap added -->
32.        <Bitmap guid="guidImages" href="Resources\Images_24bit.bmp"
33.          usedList="bmpPic1, bmpPic2, bmpPicSearch, bmpPicX,
   bmpPicArrows"/>
34.      </Bitmaps>
35.    </Commands>
36.
37.    <Symbols>
38.      <!-- New GuidSymbol section added -->
39.      <GuidSymbol name="guidImages" value="{...}" >
40.        <IDSymbol name="bmpPic1" value="1" />
41.        <IDSymbol name="bmpPic2" value="2" />
42.        <IDSymbol name="bmpPicSearch" value="3" />
43.        <IDSymbol name="bmpPicX" value="4" />
44.        <IDSymbol name="bmpPicArrows" value="5" />
45.      </GuidSymbol>
46.      <!-- Other Guids for the package -->
47.    </Symbols>
48.
49. </CommandTable>
```

As you can see, separate identifiers with bmp prefix are defined to index icons in the bitmap strip. The Bitmap element references the physical resource and in the usedList attribute enumerates the symbols for the icons which can be referenced in Button definitions. Button elements use the Icon node to name the bitmap with guidImages and the appropriate strip index with its symbolic name.

Running the package with this .vsct file results the menu in Figure 10.



Listing 10: *Adding Icons to the buttons*

Playing with CommandFlag Values

Menu item behavior can be influenced by using so-called command flags. To discover a few of them you modify two buttons with CommandFlag elements. Add a TextOnly flag to the first command. Although the first command has an associated Icon, the TextOnly command flag sets

the menu item appearance to show only the text and not the icon. Add a DynamicVisibility and a DefaultDisabled flag to the third command. These flags initialize the command in disabled state by default. Listing 7 shows the changes.

Listing 7: *Applying CommandFlags*

```
<Buttons>
  <Button guid="guidBasicVSCTSampleCmdSet" id="FirstCommand"
priority="0x0100"
    type="Button">
    <!-- CommandFlag added, other children unchanged -->
    <CommandFlag>TextOnly</CommandFlag>
  </Button>
  <Button guid="guidBasicVSCTSampleCmdSet" id="ThirdCommand"
priority="0x0100"
    type="Button">
    <!-- CommandFlag added, other children unchanged -->
     <CommandFlag>DynamicVisibility</CommandFlag>
     <CommandFlag>DefaultDisabled</CommandFlag>
  </Button>
  <!-- Other buttons are unchanged -->
</Buttons>
```

After building and running the package you can recognize the effect of command flag applied as Figure 11 shows.

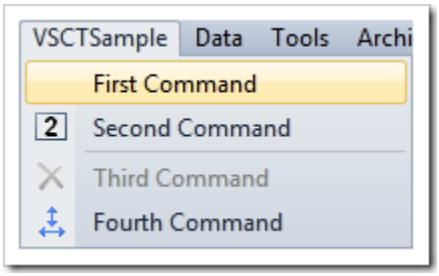


Figure 11: *Using CommandFlag elements to modify behavior*

Creating Submenus

You have already seen how to create a new main menu item and separate items visually by using command groups. Now you make a little modification to use submenus for grouping related commands. Remove the command flags you have added in Listing 7 and change the VSCT content:

- **Step 1**: Rename the TopLevelMenuGroup2 to SubMenuGroup (and all references). You turn the group of third and fourth command into a submenu group.
- **Step 2**: Add a new menu representing the submenu and attach it to the existing TopLevelMenuGroup. Name it SubMenu and create a symbol for it.
- **Step 3**: Attach the SubMenuGroup to the newly created submenu.

Listing 8 indicates the changes you have done.

Listing 8: *Moving commands into a submenu*

```
<?xml version="1.0" encoding="utf-8"?>
<CommandTable xmlns="...">
  <!-- Extern section unchanged -->
  <Commands package="guidHowToPackagePkg">
    <Menus>
      <!-- New menu added -->
      <Menu guid="guidBasicVSCTSampleCmdSet" id="SubMenu" priority="0x200"
        type="Menu">
        <Parent guid="guidBasicVSCTSampleCmdSet" id="TopLevelMenuGroup" />
        <Strings>
          <ButtonText>Other Commands</ButtonText>
          <CommandName>Other Commands</CommandName>
        </Strings>
      </Menu>
    </Menus>
    <Groups>
      <!-- Group changed to SubMenuGroup and attached to SubMenu -->
      <Group guid="guidBasicVSCTSampleCmdSet" id="SubMenuGroup"
        priority="0x0600">
        <Parent guid="guidBasicVSCTSampleCmdSet" id="SubMenu"/>
      </Group>
    </Groups>

    <Buttons>
      <!-- We attached these two buttons to SubMenuGroup -->
      <Button guid="guidBasicVSCTSampleCmdSet" id="ThirdCommand" priority="0x0100"
        type="Button">
        <Parent guid="guidBasicVSCTSampleCmdSet" id="SubMenuGroup" />
        <Icon guid="guidImages" id="bmpPicX" />
        <Strings>
          <CommandName>ThirdCommand</CommandName>
          <ButtonText>Third Command</ButtonText>
        </Strings>
      </Button>
      <Button guid="guidBasicVSCTSampleCmdSet" id="FourthCommand"
        priority="0x0101" type="Button">
        <Parent guid="guidBasicVSCTSampleCmdSet" id="SubMenuGroup" />
        <Icon guid="guidImages" id="bmpPicArrows" />
        <Strings>
          <CommandName>FourthCommand</CommandName>
          <ButtonText>Fourth Command</ButtonText>
        </Strings>
      </Button>
    </Buttons>

  </Commands>

```

```
48.    <Symbols>
49.      <!-- We add a SubMenu and changed SubMenuGroup -->
50.      <GuidSymbol name="guidBasicVSCTSampleCmdSet" value="...">
51.        <IDSymbol name="SubMenu" value="0x0101" />
52.        <IDSymbol name="SubMenuGroup" value="0x0201" />
53.      </GuidSymbol>
54.    </Symbols>
55. </CommandTable>
```

After building and running the package with the modified command table, you can see the submenu as Figure 12 shows.

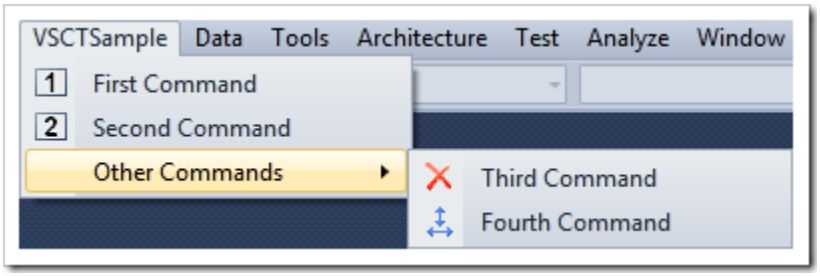


Figure 12: *Commands moved to a submenu*

If you carefully examine the command table, you can observe the following nesting: TopLevelMenu --> TopLevelMenuGroup --> SubMenu --> SubMenuGroup --> Button elements. As you see, menus can be nested to each other through Group elements. Should you break this nesting the menu would not display as expected.

Adding Shortcut Keys to Menu Items

The command table allows you to bind shortcut keys to commands, and here you are going to examine this feature. Till this time you've used all commands in the BasicVSCTSample package that have no code behind to execute. In order to demonstrate that shortcut keys are working, add code to commands to display the name of the command invoked. Listing 9 shows the code of BasicVSCTSamplePackage:

Listing 9: *Command handling in BasicVSCTSample*

```
1. using System;
2. using System.ComponentModel.Design;
3. using System.Runtime.InteropServices;
4. using System.Text;
5. using Microsoft.VisualStudio.Shell;
6. using Microsoft.VisualStudio.Shell.Interop;
7.
8. namespace DeepDiver.BasicVSCTSample
9. {
10.    [PackageRegistration(UseManagedResourcesOnly = true)]
```

```
11.   [InstalledProductRegistration(false, "#110", "#112", "1.0",
   IconResourceID = 400)]
12.   [ProvideMenuResource("Menus.ctmenu", 1)]
13.   [Guid(GuidList.guidBasicVSCTSamplePkgString)]
14.   public sealed class BasicVSCTSamplePackage : Package
15.   {
16.     private OleMenuCommandService _CommandService;
17.
18.     protected override void Initialize()
19.     {
20.       base.Initialize();
21.       _CommandService = GetService(typeof(IMenuCommandService)) as
   OleMenuCommandService;
22.       RegisterCommand(CmdIDs.FirstCommand, "First Command",
   CommandHandlerCallback);
23.       RegisterCommand(CmdIDs.SecondCommand, "Second Command",
   CommandHandlerCallback);
24.       RegisterCommand(CmdIDs.ThirdCommand, "Third Command",
   CommandHandlerCallback);
25.       RegisterCommand(CmdIDs.FourthCommand, "Fourth Command",
   CommandHandlerCallback);
26.     }
27.
28.     private void CommandHandlerCallback(object caller, EventArgs args)
29.     {
30.       var command = caller as OleMenuCommand;
31.       if (command == null) return;
32.       OutputCommandString(command.Text + " has been invoked");
33.     }
34.
35.     private void RegisterCommand(uint id, string initialText,
36.       EventHandler callback)
37.     {
38.       if (_CommandService == null) return;
39.       var menuCommandID = new
   CommandID(GuidList.guidBasicVSCTSampleCmdSet, (int)id);
40.       var menuItem = new OleMenuCommand(callback, menuCommandID) { Text
   = initialText };
41.       _CommandService.AddCommand(menuItem);
42.     }
43.
44.     private void OutputCommandString(string text)
45.     {
46.       // --- Build the string to write on the debugger and Output
   window.
47.       var outputText = new StringBuilder();
48.       outputText.AppendFormat("BasicVSCTSamplePackage: {0} ", text);
49.       var outputWindow = GetService(typeof(SVsOutputWindow)) as
   IVsOutputWindow;
50.       if (outputWindow == null) return;
51.       var guidGeneral =
   Microsoft.VisualStudio.VSConstants.GUID_OutWindowDebugPane;
52.       IVsOutputWindowPane windowPane;
53.       if (Microsoft.VisualStudio.ErrorHandler.Failed(
54.         outputWindow.GetPane(ref guidGeneral, out windowPane)))
```

```
55.         {
56.           return;
57.         }
58.         windowPane.Activate();
59.         windowPane.OutputStringThreadSafe(outputText.ToString());
60.       }
61.     }
62. }
```

Each command invokes the CommandHandlerCallback method as a response for their invocation. This method uses the OutputCommandString method to display the name of the command. The RegisterCommand method is used to bind the commands to the code to execute and setup their Text property. Because you do not use the TextChanges command flag the values set here never get reflected in the menu items.

While shortcut keys are generally assigned to menu items, in Visual Studio shortcuts are assigned to commands. Because the same command can be used in different contexts, shortcuts of commands are also interpreted in different contexts. It might be so that in a certain context a command does not have a keyboard shortcut while in another it has. In a third context the command may even have a different shortcut.

The VSCT schema has an element called KeyBinding that is put into the KeyBindings container like in the following example:

```
1. <?xml version="1.0" encoding="utf-8"?>
2. <CommandTable xmlns="..." xmlns:xs="...">
3.   <Commands package="...">
4.     <!-- Command information goes here -->
5.   </Commands>
6.
7.   <KeyBindings>
8.     <KeyBinding guid=".." id=".." ... />
9.     <!-- More key bindings -->
10.   </KeyBindings>
11.
12. </CommandTable>
```

Please note, that KeyBindings is located outside of the Command element, directly within the CommandTable element. In the BasicVSCTSample you can add shortcut keys as defined in Listing 10.

Listing 10: *Shortcut key definitions in BasicVSCTSample*

```
1. <KeyBindings>
2.   <KeyBinding guid="guidBasicVSCTSampleCmdSet" id="FirstCommand"
   editor="guidVSStd97"
3.     key1="VK_F6" mod1="Control Alt" key2="VK_F1" />
4.   <KeyBinding guid="guidBasicVSCTSampleCmdSet" id="SecondCommand"
   editor="guidVSStd97"
5.     key1="VK_F6" mod1="Control Alt" key2="VK_F2" />
```

```
6. </KeyBindings>
```

You can assign not only a simple hotkey for a command but also two. The key1 and mod1 attributes set the first hotkey, key2 and mod2 the second one. These hotkeys are not alternatives of each other: you have to press the first then the second to activate the command.

For example, in the code above for FirstCommand the first key is Ctrl+Alt+F7, the second F1. In values of key1 and key2 attributes you can use alphanumeric characters available on the keyboard, VK_ constants (IntelliSense will offer you as you type) or two digit hexadecimal constants with 0x prefix.

The values of mod1 and mod2 specify the modification keys. You can use the Control, Alt and Shift values or a combination of them where the value items are separated by a space.

With the guid and id attributes you can bind the specified key combination to the appropriate command. The editor attribute defines the context for a specific binding. The guidVSStd97 value defines that the command is global, so it is available in anywhere in Visual Studio.

Shortcut keys are displayed in menu items as Figure 13 shows.

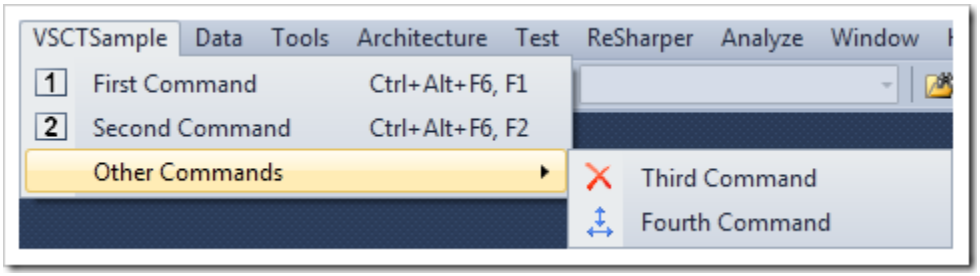


Figure 13: *Shortcut keys are displayed*

Open the Output window and use the First Command and Second Command menu items with their keyboard shortcuts. You can see the trace messages in the Output window indicating that the commands are executed.

Advanced VSCT Samples

You have just scratched the surface of the command table features as you created a main menu item and played with groupings, icons and submenus. In this section you dive a bit deeper and look after other useful samples: you are going to build toolbars, menu controllers and examine visibility contexts.

You are going to examine a new package (AdvancedVSCTSample) that was created with VSPackage wizard. You can download it from the book's website. The package is non-functional, it is only a container for resources, so it has an empty body.

VS 2010 Package Development

Creating a Toolbar

In many cases a toolbar represents commands in a more useful way than a menu does. Here you are going to turn a menu above into a toolbar and then play with a few options. The steps to create the toolbar with the commands are very similar to the ones you've used when creating a main menu item.

- **Step1**: Create a GuidSymbol element with children for the symbols we are going to use. Nest this element in the Symbols section.
- **Step 2**: Create a Menu element representing the toolbar menu. Put this item into the Menus section and set up the Parent pointing to the menu item itself.
- **Step 3**: Create a Group element representing the logical group holding the four toolbar command items. Set the Parent of this group to the Menu created in the previous step.
- **Step 4**: Create four Button elements representing the commands and set their Parent to the Group created previously.

After these steps the .vsct file should look like in Listing 11.

Listing 11: *Creating a toolbar*

```
1. <?xml version="1.0" encoding="utf-8"?>
2. <CommandTable xmlns="...">
3.   <!-- Extern section unchanged -->
4.   <Commands package="guidAdvancedVSCTSamplePkg">
5.     <Menus>
6.       <Menu guid="guidAdvancedVSCTSampleCmdSet" id="VSCTToolbar"
   priority="0x0000"
7.         type="Toolbar">
8.         <Parent guid="guidAdvancedVSCTSampleCmdSet" id="VSCTToolbar" />
9.         <CommandFlag>DefaultDocked</CommandFlag>
10.         <Strings>
11.           <CommandName>VSCTSampleToolbar</CommandName>
12.           <ButtonText>VSCT Sample Toolbar</ButtonText>
13.         </Strings>
14.       </Menu>
15.     </Menus>
16.
17.     <Groups>
18.       <Group guid="guidAdvancedVSCTSampleCmdSet" id="VSCTToolbarGroup"
19.         priority="0x0600">
20.         <Parent guid="guidAdvancedVSCTSampleCmdSet" id="VSCTToolbar"/>
21.       </Group>
22.     </Groups>
23.
24.     <Buttons>
25.       <Button guid="guidAdvancedVSCTSampleCmdSet" id="FirstCommand"
   priority="0x0100"
26.         type="Button">
27.         <Parent guid="guidAdvancedVSCTSampleCmdSet"
   id="VSCTToolbarGroup" />
28.         <Strings>
```

```
29.           <CommandName>FirstCommand</CommandName>
30.           <ButtonText>First Command</ButtonText>
31.         </Strings>
32.       </Button>
33.       <Button guid="guidAdvancedVSCTSampleCmdSet" id="SecondCommand"
34.         priority="0x0101" type="Button">
35.         <Parent guid="guidAdvancedVSCTSampleCmdSet"
   id="VSCTToolbarGroup" />
36.         <Strings>
37.           <CommandName>SecondCommand</CommandName>
38.           <ButtonText>Second Command</ButtonText>
39.         </Strings>
40.       </Button>
41.       <Button guid="guidAdvancedVSCTSampleCmdSet" id="ThirdCommand"
   priority="0x0102"
42.         type="Button">
43.         <Parent guid="guidAdvancedVSCTSampleCmdSet"
   id="VSCTToolbarGroup" />
44.         <Strings>
45.           <CommandName>ThirdCommand</CommandName>
46.           <ButtonText>Third Command</ButtonText>
47.         </Strings>
48.       </Button>
49.       <Button guid="guidAdvancedVSCTSampleCmdSet" id="FourthCommand"
50.         priority="0x0103" type="Button">
51.         <Parent guid="guidAdvancedVSCTSampleCmdSet"
   id="VSCTToolbarGroup" />
52.         <Strings>
53.           <CommandName>FourthCommand</CommandName>
54.           <ButtonText>Fourth Command</ButtonText>
55.         </Strings>
56.       </Button>
57.     </Buttons>
58.
59.   </Commands>
60.
61.   <Symbols>
62.     <GuidSymbol name="guidAdvancedVSCTSamplePkg" value="..." />
63.     <GuidSymbol name="guidAdvancedVSCTSampleCmdSet" value="...">
64.       <IDSymbol name="VSCTToolbar" value="0x0100" />
65.       <IDSymbol name="VSCTToolbarGroup" value="0x0200" />
66.       <IDSymbol name="FirstCommand" value="0x0300" />
67.       <IDSymbol name="SecondCommand" value="0x0301" />
68.       <IDSymbol name="ThirdCommand" value="0x0302" />
69.       <IDSymbol name="FourthCommand" value="0x0303" />
70.     </GuidSymbol>
71.   </Symbols>
72. </CommandTable>
```

The key in this scenario is the Menu element that uses the Toolbar value in its type attribute. Standard menus have parent groups defined by Visual Studio. However, for toolbars the concept of parent menu or menu group has no meaning just like as a nested toolbars does not. As a convention, you set the Parent of a toolbar to the same GUID and ID pair as used for the toolbar identification. The first time the toolbar is shown up you want it to be docked so you set a

CommandFlag with the value of DefaultDocked. The Button elements now are parented into the Group element that points to the Menu instance.

Now, you can build the project and see how the toolbar works. Start debugging and make the toolbar visible by enabling our toolbar with View ð Toolbars ð VSCTSampleToolbar menu item. The toolbar will be shown docked right under the standard toolbar as shown in Figure 14.

Figure 14: *Toolbar with button placeholders*

Note: In previous Visual Studio versions this .vcst file would have resulted in a toolbar with four button placeholders without any text. To achieve the save visual properties you can see in Figure 3-14 the TextOnly command flag should have been assigned to the Button elements. Visual Studio 2010 changes this behavior. By default, when a toolbar button does not have an icon, its text is displayed even if the TextOnly command flag is not used.

Well, this toolbar resembles to a standard menu, the original concept of a toolbar seems lost. So, let's move on using icons. To set up button icons you are going to use exactly the same code changes as used in Listing 3-6. Now the result shown in Figure 15 rather resembles to a toolbar than the first one.

Figure 15: *Toolbar buttons with text*

With toolbars you can use some other controls. Let's see a few examples!

Menu Controllers on Toolbars

By now you have seen examples for two kinds of menus: standard menus and submenus. In toolbars you can use a third kind of menu called menu controller: this is a split-button drop-down menu. You can change the toolbar above so that the last three buttons go into a menu controller just like in Figure 16.

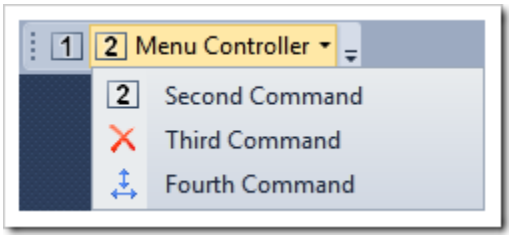


Figure 16: *Menu controller with three items*

Creating a menu controller is very similar to creating a submenu. To setup the items as in **Error! Reference source not found.** the following steps should be carried out:

- **Step 1**: Create IDs for a new Menu element and a new Group.
- **Step 2**: Create a Menu item with type of MenuController.
- **Step 3**: Create a Group item for encapsulate the three Button controls that will appear in the menu controller.
- **Step 4**: Attach the button items to the Group created in the third step.

Listing 12 summarizes the changes to establish the menu controller.

Listing 12: *Using a menu controller*

```
1. <?xml version="1.0" encoding="utf-8"?>
2. <CommandTable xmlns="...">
3.   <!-- Extern section unchanged -->
4.   <Commands package="guidAdvancedVSCTSamplePkg">
5.     <Menus>
6.       <!-- New item representing the menu controller -->
7.       <Menu guid="guidAdvancedVSCTSampleCmdSet" id="VSCTMenuController"
8.       priority="0x0200" type="MenuController">
9.         <Parent guid="guidAdvancedVSCTSampleCmdSet"
   id="VSCTToolbarGroup"/>
10.         <CommandFlag>IconAndText</CommandFlag>
11.         <Strings>
12.           <ButtonText>VSCT Menu Controller</ButtonText>
13.           <CommandName>Menu Controller</CommandName>
14.         </Strings>
15.       </Menu>
16.     </Menus>
17.
18.     <Groups>
19.       <!-- New group for the items in the menu controller -->
20.       <Group guid="guidAdvancedVSCTSampleCmdSet"
   id="VSCTControllerGroup"
21.         priority="0x0600">
22.         <Parent guid="guidAdvancedVSCTSampleCmdSet"
   id="VSCTMenuController"/>
23.       </Group>
24.     </Groups>
25.
26.     <Buttons>
27.       <!-- We leave the first button as it is -->
28.       <!-- We change the 2nd 3rd and 4th button as indicated -->
29.       <Button guid="guidHowToPackageCmdSet" id="cmdSecondCommand"
30.         priority="0x0101" type="Button">
31.         <Parent guid="guidAdvancedVSCTSampleCmdSet"
   id="VSCTControllerGroup" />
32.         <CommandFlag>IconAndText</CommandFlag>
33.         <Icon guid="guidImages" id="bmpPic2" />
```

```
34.          <Strings>
35.            <CommandName>cmdidSecondCommand</CommandName>
36.            <ButtonText>Second Command</ButtonText>
37.          </Strings>
38.        </Button>
39.        <!-- Code for the 3rd and 4th button is similar -->
40.      </Buttons>
41.      <!-- Bitmaps section unchanged -->
42.    </Commands>
43.
44.    <Symbols>
45.      <GuidSymbol name="guidHowToPackageCmdSet" value="..." >
46.        <!-- We use these two new IDs -->
47.        <IDSymbol name="VSCTMenuController" value="0x0101" />
48.        <IDSymbol name="VSCTControllerGroup" value="0x0201" />
49.      </GuidSymbol>
50.      <!-- Other Guids of the package -->
51.    </Symbols>
52. </CommandTable>
```

The Menu item for a menu controller can be defined so that the type attribute is set to MenuController. The Parent of the menu is the command group representing the four command buttons as it was used in the previous example.

You attach the newly created group to the Menu item above and then connect the last three Button elements to this Group. You also change the CommandFlag elements for the new menu controller item and for the tree involved button to IconAndText in order to make it dropdown-menu-like.

IconAndText is the default value for the buttons attached to a menu controller, so if you leave them for buttons, the toolbar appearance will not change. However, for the menu controller itself the default mode displays only the icons. If you had omitted the IconAndText flag from the controller definition we could see the toolbar like in Figure 17.

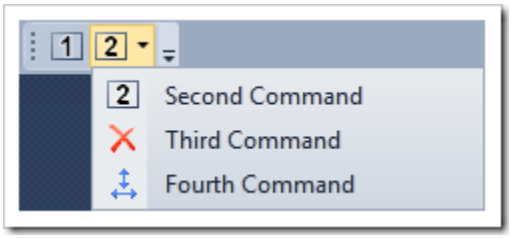


Figure 17: *Menu controller with icon only*

Combo Boxes

Combo boxes provide you an enhanced form of input by allowing a combination of hand-typed text selection with picking items from a list. Visual Studio also allows combo boxes on the user interface. The command table file (.vsct) has Combo elements to represent UI of commands with

combo boxes instead of standard menu items or buttons. There are four stereotypes of combos; each of them is identified by a type name used in the .vsct file.

- **DropDownCombo**: This type does not let the user type into the combo box; they can only pick an item from a list. After selecting an item the string value of the selected element is returned.
- **IndexCombo**: This type is the same as a DropDownCombo in that it allows only picking up items from a list. However, an IndexCombo returns the zero-based index of the selected value on the list and not the value itself.
- **MRUCombo**: This combo type allows the user to type into the edit box. The history of strings entered is automatically persisted by the IDE on a per-user or per-machine basis for the last 16 items.
- **DynamicCombo**: This combo allows the user to type into the edit box or pick from the list. The list of choices is managed dynamically by a command event handler method.

Handling combo boxes is more complicated than working with simple menu items. A part of the complexity comes from the fact that combo boxes use two commands behind. The first command is executed as you select an item from the dropdown list. The second command is used to fill up the list of items.

Working with Visibility Contexts

In the beginning of this chapter you have already learnt about the concept of visibility contexts, and in Table 4 you can see the most frequently used ones. In this section you are going to create a small sample package demonstrating concepts treated there.

You can examine a package named VisibilityContextSample and similarly to the previous samples this package is only a container for the menu and does not provide any real function. The package adds a menu that looks like as shown in Figure 18.

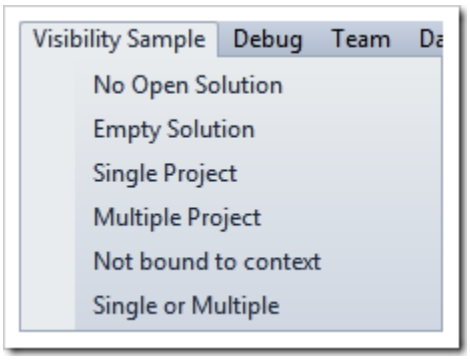


Figure 18: *Menus defined by the VisibilityContextSample package*

The command table producing this menu can be found in Listing 13 and uses exactly the same approach that can be found in Listing 4.

Listing 13: *VisibilityContextSample initial command table*

```
1. <?xml version="1.0" encoding="utf-8"?>
2. <CommandTable xmlns="http://schemas.microsoft.com/VisualStudio/2005-10-
   18/CommandTable" xmlns:xs="http://www.w3.org/2001/XMLSchema">
3.   <Extern href="stdidcmd.h"/>
4.   <Extern href="vsshlids.h"/>
5.   <Extern href="msobtnid.h"/>
6.   <Commands package="guidVisibilityContextSamplePkg">
7.     <Menus>
8.       <Menu guid="guidVisibilityContextSampleCmdSet" id="TopLevelMenu"
   priority="0x100"
9.         type="Menu">
10.         <Parent guid="guidSHLMainMenu" id="IDG_VS_MM_BUILDDEBUGRUN" />
11.         <Strings>
12.           <ButtonText>Visibility Sample</ButtonText>
13.           <CommandName>Visibility Sample</CommandName>
14.         </Strings>
15.       </Menu>
16.     </Menus>
17.
18.     <Groups>
19.       <Group guid="guidVisibilityContextSampleCmdSet"
   id="TopLevelMenuGroup"
20.         priority="0x0600">
21.         <Parent guid="guidVisibilityContextSampleCmdSet"
   id="TopLevelMenu"/>
22.       </Group>
23.     </Groups>
24.
25.     <Buttons>
26.       <Button guid="guidVisibilityContextSampleCmdSet" id="NoSolution"
   priority="0x0100"
27.         type="Button">
28.         <Parent guid="guidVisibilityContextSampleCmdSet"
   id="TopLevelMenuGroup" />
29.         <Strings>
30.           <CommandName>cmdNoSolution</CommandName>
31.           <ButtonText>No Open Solution</ButtonText>
32.         </Strings>
33.       </Button>
34.
35.       <Button guid="guidVisibilityContextSampleCmdSet"
   id="EmptySolution"
36.         priority="0x0101" type="Button">
37.         <Parent guid="guidVisibilityContextSampleCmdSet"
   id="TopLevelMenuGroup" />
38.         <Strings>
39.           <CommandName>cmdEmptySolution</CommandName>
40.           <ButtonText>Empty Solution</ButtonText>
41.         </Strings>
42.       </Button>
43.
44.       <Button guid="guidVisibilityContextSampleCmdSet"
   id="SingleProject"
45.         priority="0x0102" type="Button">
```

```
46.         <Parent guid="guidVisibilityContextSampleCmdSet"
   id="TopLevelMenuGroup" />
47.         <Strings>
48.           <CommandName>cmdSingleProject</CommandName>
49.           <ButtonText>Single Project</ButtonText>
50.         </Strings>
51.       </Button>
52.
53.       <Button guid="guidVisibilityContextSampleCmdSet"
   id="MultipleProject"
54.         priority="0x0103" type="Button">
55.         <Parent guid="guidVisibilityContextSampleCmdSet"
   id="TopLevelMenuGroup" />
56.         <Strings>
57.           <CommandName>cmdMultipleProject</CommandName>
58.           <ButtonText>Multiple Project</ButtonText>
59.         </Strings>
60.       </Button>
61.
62.       <Button guid="guidVisibilityContextSampleCmdSet" id="NotBound"
   priority="0x0104"
63.         type="Button">
64.         <Parent guid="guidVisibilityContextSampleCmdSet"
   id="TopLevelMenuGroup" />
65.         <Strings>
66.           <CommandName>cmdNotBound</CommandName>
67.           <ButtonText>Not bound to context</ButtonText>
68.         </Strings>
69.       </Button>
70.
71.       <Button guid="guidVisibilityContextSampleCmdSet"
   id="SingleOrMultiple"
72.         priority="0x0105" type="Button">
73.         <Parent guid="guidVisibilityContextSampleCmdSet"
   id="TopLevelMenuGroup" />
74.         <Strings>
75.           <CommandName>cmdSingleOrMultiple</CommandName>
76.           <ButtonText>Single or Multiple</ButtonText>
77.         </Strings>
78.       </Button>
79.     </Buttons>
80.
81.   </Commands>
82.
83.   <Symbols>
84.     <GuidSymbol name="guidVisibilityContextSamplePkg" value="..." />
85.     <GuidSymbol name="guidVisibilityContextSampleCmdSet" value="...">
86.       <IDSymbol name="TopLevelMenu" value="0x1000" />
87.       <IDSymbol name="TopLevelMenuGroup" value="0x1020" />
88.       <IDSymbol name="NoSolution" value="0x0100" />
89.       <IDSymbol name="EmptySolution" value="0x0101" />
90.       <IDSymbol name="SingleProject" value="0x0102" />
91.       <IDSymbol name="MultipleProject" value="0x0103" />
92.       <IDSymbol name="NotBound" value="0x0104" />
93.       <IDSymbol name="SingleOrMultiple" value="0x0105" />
```

```
94.      </GuidSymbol>
95.    </Symbols>
96.
97. </CommandTable>
```

If you apply this .vsct file for a package you will get the menu in the Figure 18, but nothing extra. Bind the commands to specific visibility contexts! First, add DynamicVisibility command flag to all buttons. Without this flag the sample would not work. Second, add the VisibilityConstraints section as in Listing 14 to the command table.

Listing 14: *VisibilityContextSample initial command table*

```
1. <VisibilityConstraints>
2.   <VisibilityItem guid="guidVisibilityContextSampleCmdSet"
   id="NoSolution"
3.     context="UICONTEXT_NoSolution"/>
4.   <VisibilityItem guid="guidVisibilityContextSampleCmdSet"
   id="EmptySolution"
5.     context="UICONTEXT_EmptySolution"/>
6.   <VisibilityItem guid="guidVisibilityContextSampleCmdSet"
   id="SingleProject"
7.     context="UICONTEXT_SolutionHasSingleProject"/>
8.   <VisibilityItem guid="guidVisibilityContextSampleCmdSet"
   id="MultipleProject"
9.     context="UICONTEXT_SolutionHasMultipleProjects"/>
10.   <VisibilityItem guid="guidVisibilityContextSampleCmdSet"
   id="SingleOrMultiple"
11.     context="UICONTEXT_SolutionHasSingleProject"/>
12.   <VisibilityItem guid="guidVisibilityContextSampleCmdSet"
   id="SingleOrMultiple"
13.     context="UICONTEXT_SolutionHasMultipleProjects"/>
14. </VisibilityConstraints>
```

Each VisibilityItem node binds the specified command to a visibility context defined by the context attribute. Symbols starting with the UICONTEXT_ prefix define the contexts as summarized in Table 3-4. The NotBound button is not assigned with any contexts, so it is always visible (it is displayed in every context). The SingleOrMultiple command is bound to two contexts (represented by the last two VisibilityItem nodes), so it is displayed when there is a solution with one or more projects open.

We can check in a few steps how these constraints work. First, when you open Visual Studio you can see the initial state of the menu as shown in Figure 19.

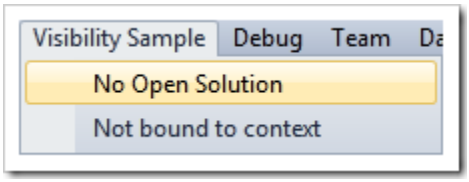


Figure 19: *Initial menu state after starting Visual Studio*

Now, carry out the following steps and see the results in the corresponding figures:

- Create an empty solution (Figure 20)
- Add a C# class library (or any other kind of project) to the solution (Figure 21)
- Add another project to the existing solution (Figure 22)

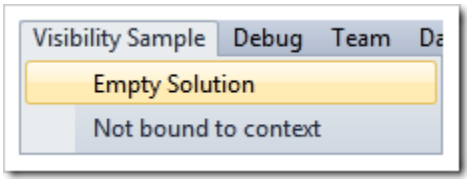


Figure 20: *Empty solution created*

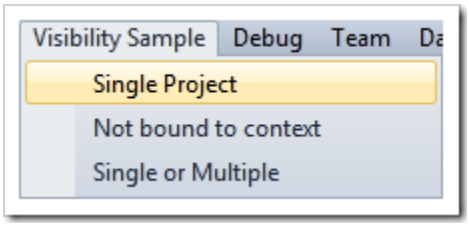


Figure 21: *A new project is added to the solution*

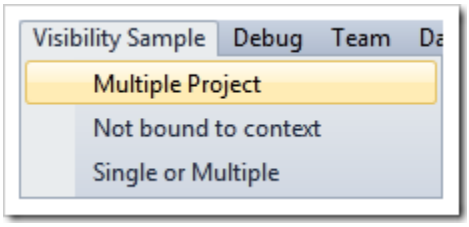


Figure 22: *Another project is added to the solution*

As you see, visibility contexts work as expected.

Command Routing and Nested Contexts

The Visual Studio IDE, its registered packages, and objects owned by packages (editors, tool windows, and so on) define a huge set of executable commands. A command can be executed by a number of entities depending on the current context, for instance by the active editor window, the currently selected project, active tool window, and so on.

There are many objects at the same time in the IDE which are command targets and so are able to receive and execute commands. Visual Studio has a well-defined routing architecture that lays down the rules how commands are executed in a certain context — in which order command targets are offered to process commands. This algorithm starts form the innermost context and goes to the outermost context while it reaches the global context that is the core Visual Studio IDE.

From this point of view contexts are nested into each other. Here is a very simple example how contexts are nested. The real life is a bit more complex, but this helps you imagine nesting.

Assume you create a package with a tool window and register it with Visual Studio IDE. Because both the package itself and the tool window owned by the package are command targets, you have the following contexts from the top to the bottom:

- The top level (global context) is Visual Studio IDE.
- Your package is sited in the IDE. Before siting it has its own logical context. After you site the package its context logically will be a context nested into the IDE.
- The tool window of the package has its own contexts. When an instance is created its context becomes a nested context within the package.

Visual Studio IDE, packages and package-owned objects form a tree of nested command contexts. From this perspective the routing algorithm starts from a leaf of that tree and bubbles up while it reaches the root of this tree that is called the global context.

The Routing Algorithm

The algorithm that routes commands has many fine details which add small twists. Here you will not learn all those subtle details; instead you'll be given a high level overview.

In the route the current level is called the active command context. This context has the chance to handle the command or to say "I do not know what to do with this command" and the bubble goes on its way.

The routing algorithm defines the following levels from the leaves to the root:

- **Present Add-ins and special packages.** Commands first are offered to the registered and loaded Add-ins or specially registered packages.
- **Context (shortcut) menus.** If the user initiates a command from a context menu, the command target object belonging to this menu has the first chance to handle the command. If it does not, then the normal route (starting from Present Add-Ins) is applied.
- **Focus window.** The window having the focus is the next entity that can undertake command handling. This can be either a tool window or a document window, for instance, a window related to an editor. The management of the command is different depending on what kind of window is focused.

    - **Document window.** There are windows in Visual Studio like editors and designers that have physical or virtual documents behind them. Document windows are composed logically from two separate parts: a document view that is responsible to display the UI representing the document and a document data object that is responsible to handle the information set behind the document. Both the document view and the document data can be command targets. The command first goes to the document view and goes on to the document data if the view does not support the command.
    - **Tool window.** The tool window can handle the command by its own logic. There are tool windows that route the commands within themselves to nested command targets. The Solution Explorer window is an example of them. Within Solution Explorer a command is routed according to the hierarchy composed from the elements of the Solution Explorer where each node type (file, folder, project, solution, etc.) has the ability to handle the command. This internal route also goes from the lower hierarchy levels to the upper ones.
- **Current project.** The current project gets the opportunity to process the command. If it does not handle, the command goes up in the hierarchy of projects till the level of solution. All nodes on this route can manage the command just like other command target objects. (Visual Studio allows creating nested projects. When subprojects are used, the upper level of a project is not necessary the solution.)
- **Environment.** Each package should be able to handle commands owned (defined) by it. Theoretically it is not mandatory, but why to define a custom command which is not handled by the only entity that knows it?
- **Global level.** If a command is not handled in the previous levels, the environment attempts to route it to the appropriate package (the package defining the command). If necessary, Visual Studio loads the appropriate package into the memory.

Any command target objects along the route can decide how to process a command. They must answer status queries and execute requested commands.

Summary

Visual Studio clearly separates the concept of menu items and commands.

A command is responsible for determining its state (name, visibility, enabled, disabled, etc.) and executing the command triggered. A menu item is responsible for presenting the visual properties of a command and providing a way the user can trigger the execution of the command. A command object can be bound to zero, one or more menu items.

A command itself it a logical entity that can be forwarded to command targets which know how to handle the semantics of a specific command. There is a command routing model in the IDE that forwards a command to a command target. The target either can do something with the request related to a command (for example, set the command state disabled, execute the command, etc.) or can pass back the command as not supported (the target does not know what to do with that). The target even can pass the command to other command targets.

VS 2010 Package Development

Each command has a state in the current Visual Studio context. This state is built up from two orthogonal factors: availability (the command is enabled or disabled) and visuals (the command is visible or hidden, the label it has, and so on).

The Visual Studio Command Table describes the commands and related UI elements that should be merged with Visual Studio menus. The command table is compiled to a binary resource which is embedded into the package infrastructure resources during the build process. When deploying the package the command table is merged into the menus and toolbars of the IDE.

Visual Studio 2010 uses an XML format with .vsct extension to describe the command table. Older versions used a simple textual format called .ctc that has been deprecated.

In this chapter you have seen the structure of the .vsct file and solved simple tasks like building a new main menu item or a toolbar, putting commands visually into command groups or submenus, working with menu controllers, etc. You have also examined a sample illustrating how visibility constraints work.