

Extending Visual Studio 2005: An Overview

This document is a technical writing sample prepared over a 3-day period and submitted to a large software development company on 2-Feb-2007 for a Programmer / Writer position in an SDK tools group. Unfortunately, things did not come together for that but in another area did, and I ended up writing a Linux book.

Anyway, this is my original writing sample, unchanged with exception of this note and the copyright headers, in case someone finds its content useful. Everything presented here is my own original work, including all tables, graphs, and code.

It is also a rough draft as I spent most of my time researching, so be careful. It was a totally new area for me, and I had only three days to do all of this research and writing.

Have fun.

-Mark Qu

This document is formatted in Microsoft Word's *outline view* for ease of perusing.

Researched & written between January 31st – February 2nd, 2007.

CONTENTS

Extending Visual Studio 2005: An Overview	1
Introduction.....	2
Overview of Extension & Customization Options	4
Macros.....	4
Add-ins & Plug-ins	6
Visual Studio Packages.....	13
Additional Resources	15
Online.....	15
Printed.....	15
Appendix A: Visual Studio 2005 Extensibility- Relative Learning and Implementation Times.....	16
Appendix B: Visual Studio 2005 Add-in Environment.....	17
Appendix C: Anatomy of an Add-In-provided Tool Window.....	18
Appendix D: Debugging the Extensibility Automation Chain.....	19

Introduction

Visual Studio 2005 is a customizable Integrated Development Environment (IDE) and robust developer tools host. Its customization and extension options range from developer preferences set during Visual Studio installation to the addition of new programming languages, editors, and debuggers.

Visual Studio can be extended in a number of ways, including:

- Basic customizations & settings (profiles)
- Macros (such as via the keyboard recorder)
- Project Templates & Wizards
- Starter Kits (really just a meatier Project Template)
- Add-ins (via the Extensibility project type)
- VsPackages (also via the Extensibility project type, after Visual Studio SDK installation)

Basic customization begins with Visual Studio installation, during which time the tool installs according to selections made during the installation process. After installation, further configuration refinements may be made using the IDE menu options, such as **tools -> options** settings, help tool optimizations, and so on. These settings may be saved as *profiles* and shared with other Visual Studio users. To work with profiles, select **tools -> import & export settings** from the Visual Studio menu bar.

Visual Studio may be further optimized by automating away repetitive tasks encountered during use. Most commonly this automation is accomplished through *macros* (**VsMacros**) which automate the Visual Studio IDE through the Visual Studio Automation Object Model. In the case of keyboard-recording macros, no coding at all is necessary; Visual Studio will generate the requisite code and produce the macros ready to run.

Project Templates, Wizards, and Starter Kits allow for the creation of customized frameworks from which new projects may be started.

A Wizard is useful for two primary tasks in Visual Studio: creating a new project or adding new items to a project. Wizards are COM DLLs.

To create a Wizard, create a C# Class Library project.

Click this link for full details:

http://msdn.microsoft.com/library/default.asp?url=/library/en-us/extensibility_guided_tour/visual%20studio%20extensibility.asp

Starter Kits embody basic functionality of a concept or technology, usually as a ready-to-run construct complete with documentation. The usual progression is to create a Project Template, and then turn that Project Template into a Starter Kit.

See ‘How to: Create Starter Kits’ in MSDN

The next two extension options, *Add-ins* and *Packages*, are in the domain of the advanced developer and third-party developer tools and language implementers. Add-ins extend the Visual Studio IDE itself, producing features presented in the IDE as if they were built-in to Visual Studio, and are even automatable.

Examples Add-ins (sometimes referred to as Plug-ins) include the popular CodeKeep and XPathmania plug-ins from devx.com. Add-ins most commonly extend Tools -> Options and provide new *tool windows*. You cannot add major Studio features such as a new language, debugger, document type, or project types with Add-ins. Add-ins add in tools which interact with such base Studio functionality. Extending Visual Studio base functionality itself is possible only through Packages.

Packages (also known as *VsPackages*) are base units of Visual Studio functionality, and are the products of the Visual Studio SDK. The Package project type is added to Visual Studio 2005 with the installation of the Visual Studio SDK.

Examples of VsPackages include the VB, Java, and C# languages, debug engines, and program editors.

This overview covers the programmable extensions (Macros, Add-ins, and a brief introduction to Packages) of Visual Studio 2005.

Overview of Extension & Customization Options

Macros

Macros are VB scripts which may be edited, saved, and even exported & shared among users. Such sharing implies code sharing as the macros are treated and maintained only in source code form by Visual Studio 2005.

Create macros from the **Tools -> Macros** menu from within Visual Studio. There you can record a keyboard macro, create a macro from scratch via the macro IDE, import a macro, or a combination of these actions.

Macros can be launched from the macro explorer, executed from the Visual Studio command window, or triggered to execute upon breakpoint or other Visual Studio event.

The Visual Studio macro facility is designed to be a quick, light-weight, and easy to use set of tools to automate repetitive tasks in Visual Studio. As such, only a subset of the Visual Studio Automation model is exposed to macros. Also, while Macros can pop-up a dialog box for user interaction, they cannot modify the Visual Studio 2005 IDE UI itself.

Tools -> Macros -> Macros IDE launches the Macros IDE. The IDE is similar to the Visual Studio IDE, with a Project, instead of Solution, Explorer. There References may be added & deleted, modules (this is VB, after all) defined & edited, debugging conducted, etc.

The Macros IDE is separate from Visual Studio's, to facilitate debugging. However, use of the IDE is not required to actually produce a macro, as they may also be created by keyboard recording or off the Macros Explorer.

Recording Macros

A special module, known as the **RecordingModule**, will be added to a macro if macro recording commences (that is, when the macro developer clicks **Tools -> Macros -> Record Temporary Macro**).

Recording starts immediately, so its best to ensure that your cursor is properly positioned and ready for recording.

All typing is recorded in the Sub TemporaryMacro() method of the RecordingModule, and is overwritten if another 'Record Temporary Macro' operation is initiated. Mouse movements are not recorded.

Writing Macros

Launch the Macros IDE (**Tools -> Macros -> Macros IDE**) and, from the Project Explorer, select 'Module1' (a Visual Basic code module). Here is some example code, most of which was written by the tool:

```
Imports System
Imports EnvDTE                `Visual Studio Object Model
Imports EnvDTE80             `Visual Studio 2005 Object Model
Imports System.Diagnostics

Public Module Module1
    Sub MyMacro()              `Example user-written macro code
        MsgBox(DTE.Version)
    End Sub
End Module
```

This code example gathers the Visual Studio version number via the Visual Studio automation model and displays it in a message box. The automation model is represented within the EnvDTE and EnvDTE80 namespaces, and must be referenced by the project (this referencing is done for you by the Macros IDE).

DTE (Developer Tools Extensibility) is a global reference to the top level Visual Studio automation object, and is available to all modules. The DTE object contains various methods and pathways to other Visual Studio objects and collections, and is the root of the entire Visual Studio automation model. Search '**Visual Studio Automation Object Model**' in MSDN for more information and a color-coded object model chart that lists the DTE and new DTE2 object model objects and collections in an easy to interpret hierarchy.

Place your cursor in the macro module code block (Sub MyMacro() in the example shown) and press the F5 key to run macro code in the Macros IDE. For the example code shown here, a dialog containing the Visual Studio 2005 version (8.0) will display, along with an OK button.

Event-Driven Macros

From the Macro's IDE Project Explorer, double click **EnvironmentEvents**. The relevant code displays.

From the top of the code window, click the left drop-down arrow to display the available events to code against. The top right of the code window sports a drop-down arrow also, which drills down into whatever event is selected in the left drop-down list.

Here is an example, where 'BuildEvents' was selected in the left drop-down and 'OnBuildBegin' from the right drop-down list:

```
Private Sub BuildEvents_OnBuildBegin(ByVal Scope As  
EnvDTE.vsBuildScope, ByVal Action As EnvDTE.vsBuildAction) Handles  
BuildEvents.OnBuildBegin  
    MsgBox("The build begins!") 'Example user-written macro code  
    MyMacro() 'User method call  
End Sub
```

In this example, a message box pops up whenever a build is invoked from within Visual Studio. A method call then follows, which is a call to the version code developed as an example earlier. It is added here to make things slightly more interesting.

Macros Summary

Macros are interpreted units of Visual Studio functionality that are written or keystroke recorded in Visual Basic form. They may be launched by the Macro Explorer, command window, or a Visual Studio event.

Macros typically wrap and automate available Visual Studio functionality. To go further with Visual Studio customization, such as modifying the IDE itself or protecting the intellectual rights enabling a given customization, Add-ins are the requisite technology. Macros may be exported and imported, and thus shared among developers. Such sharing implies source code sharing as macros are treated and managed in source code form by Visual Studio 2005.

Add-ins & Plug-ins

Add-ins, sometimes referred to as Plug-ins, are compiled units of extensibility functionality. This is in contrast to macros, whose source code is always available to users. Add-ins then, by their compiled nature, provide some measure of intellectual property protection that macros cannot. Add-ins also have a greater reach into the Visual Studio Automation Object Model, and can create new IDE components such as property pages and tool windows. These persistent functionality are not possible with macros. In addition, Add-in provided IDE functionality may be automated, making Add-ins and macros complementary technologies.

Creating Add-ins

From the Visual Studio IDE menu bar, navigate to:

File -> New -> Project -> Other Project Types Extensibility: Visual Studio Add-in

...and create a project. The examples and discussion in this paper assume a C# Add-in project.

Add-in Application Hosts

The Visual Studio Extensibility Wizard will ask for the application host for your Add-in. This is simply the environment you wish to extend with the Add-in, and presently there are two choices:

Microsoft Visual Studio 2005

Microsoft Visual Studio 2005 Macros (this is the Macros IDE)

Add-in Execution Options

Also in the Visual Studio Extensibility Wizard will be settings for arranging for toolbar support for the Add-in, load options (load on Visual Studio startup or load on user command), and whether or not a 'Help -> About' should be provided with the Add-in and, if so, the dialog's message content.

Discussion

Conceptually, Add-ins implement the methods of the **IDTExtensibility2** and **IDTCommandTarget** interfaces and make calls to the Visual Studio Automation Object Model in response to Visual Studio calls to these interfaces. Thus, Add-ins are event-driven. The automation object model, which is described in the EnvDTE (DTE) and EnvDTE80 (DTE2) namespaces, must be referenced by the Add-in project in order to be called by the Add-in. This referencing task is taken care of by the Add-in Wizard, which among other things also produces a ready-to-edit source file (Connect.cs) which stubs these two interfaces. The following wizard-generated code snip (minus my comments) illustrates the inclusion of these requisite namespaces into the Add-in project.

Visual Studio 2005 is Studio version 8.0, hence the 'EnvDTE80' namespace. There are about 300 objects in the automation model, and you can add more. Source is from the Add-in wizard-generated Connect.cs file:

```
using Extensibility; // use the namespace defining IDTExtensibility2
```

```
using EnvDTE;           // Visual Studio Object Model
using EnvDTE80;        // Visual Studio 2005 Object Model
```

The IDTExtensibility2 and IDTCommandTarget Interfaces

The IDTExtensibility2 and IDTCommandTarget interfaces describe the lifecycle of an Add-in. An Add-in must be launched for these events to be responded to. Add-in launch options are set in the Add-in Manager, located at **Tools -> Add-in Manager**. Presented here are some of the interface methods to be fleshed out as required, in the Wizard-generated **Connect.cs** file. All methods are from IDTExtensibility2, except where noted:

- public void **OnStartupComplete** (ref Array custom): this is called when Visual Studio completes its own initialization and is available to the Add-in.
- public void **OnAddInsUpdate** (ref Array custom): this is called when an Add-in is added or deleted from the Visual Studio Add-in Manager.
- public void **OnBeginShutdown** (ref Array custom): this is called when the user clicks CLOSE, ending the Visual Studio session.
- public void **OnDisconnection** (ref Array custom): this is called when the Add-in is being deallocated.
- public void **QueryStatus** (string commandName, vsCommandStatusTextWanted neededText, ref vsCommandStatus status, ref object commandText): this is called when the command's availability is questioned by Visual Studio. For example, it allows Visual Studio to determine if the Add-in is currently accepting commands. If all is well, control passes to the Exec method. This method is defined in **IDTCommandTarget**.
- public void **Exec** (string commandName, vsCommandExecOption executeOption, ref object varIn, ref object varOut, ref bool handled): this is called when the Add-in's command is invoked. Defined in **IDTCommandTarget**.
- public void **OnConnection** (object application, ext_ConnectMode connectMode, object addInInst, ref Array custom): this is called when the Add-in is being loaded. The method is also the means by which your Add-in receives access to the Visual Studio Automation Object Model and reference to itself (highlighted yellow in the argument list). Consequentially, this is the most implemented and perhaps the most important of the IDTExtensibility methods.

Obtaining Access to the Visual Studio Automation Object Model

The Visual Studio Add-in Wizard takes care of receiving and storing a reference to the Visual Studio object model in the code it generates for the Connect.cs file. Specifically, references to the object model, and the Add-in itself, are private data members initialized when Visual Studio calls the OnConnection() method of the Add-in's IDTExtensibility implementation. The following code, from Connect.cs, illustrates this initialization:

```
_applicationObject = (DTE2)application; // Top-level DTE object  
_addInInstance = (AddIn)addInInst; // Reference to our (this) Add-In
```

Modifying the Visual Studio IDE UI with Add-ins

Creating Tool Windows

Overview

Unlike macros, Add-ins can directly modify the Visual Studio IDE UI. Macros, on the other hand, can only automate it. *Add-ins have their own limitations: they do not have the power to create new base Visual Studio functionality such as implementation of a programming language, program editor, or debugger. In other words, Add-ins do not allow for the creation of new document types. For these tasks you need to use the Visual Studio SDK and write VsPackages.* Add-ins are used to extend Visual Studio around an already installed base functionality.

A common modification to the Visual Studio IDE by Add-Ins is the addition of Tool Windows to display built-in or custom user interfaces.

There are many tool windows available from within the Visual Studio automation model, such as **task list** and **output windows**, etc. These are of course objects onto themselves, and are thus automatable via macros and Add-ins. But sometimes these may not be exactly what you want. In such cases use the ‘**CreateToolWindow**’ (ActiveX) or ‘**CreateToolWindow[2]**’ (.NET) method to create a custom tool window. Such windows allow you to define the placement of controls and content much as you would when laying out a Windows form.

Tool windows are created using either ActiveX or .NET user controls, and there is really no difference between the two, but there are more ActiveX user controls simply because the technology has been around longer. Its all COM under the covers, however it is recommended that further development be along .NET rather than COM lines. Either way, the resulting custom functionality is then accessible for automation by macros or Add-ins just like the stock, built-in functionality of Visual Studio.

Coding

Right click on your Add-in project in Solution Explorer and choose ‘**Add -> User control**’. The ‘Add New Item’ dialog box appears. Select ‘User Control’ and then press OK to dismiss the dialog.

For this example, drag a command button over to the control box.
Add the following code to the ‘OnConnection’ method of IDTExtensibility2 (most will have already been added by the Wizard):

```
public void OnConnection(object application, ext_ConnectMode
connectMode, object addInInst, ref Array custom)
{
    _applicationObject = (DTE2)application;
    _addInInstance = (AddIn)addInInst;
    object obj = null;
    Window win =
((Windows2)_applicationObject.Windows).CreateToolWindow2
    (_addInInstance,

System.Reflection.Assembly.GetExecutingAssembly().Location,
    "FileHeader.UserControll1",
    "My Tool Window",
    "{27ECB278-EC5D-40ff-9ACD-8D3ADAC230D2}",
    ref obj);
    win.Visible = true;
}
```

Note that Visual Studio’s call to our OnConnection() implementation provides us with a reference to our Add-In instance (argument: object addInInst) and a reference to the Visual Studio automation model (argument: object application).

Press F5 to execute. The “My Tool Window” tool window appears as designed.

Adding Method Functionality

In the User Control that we added (UserControll1) lets implement an interface, IMyToolObj, and a method on that interface, MyMethod(). The listing below is from UserControll1.cs).

Note that our public partial class implements our new interface `IMyToolObj`:

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Drawing;
using System.Data;
using System.Text;
using System.Windows.Forms;

namespace FileHeader
{
    //
    // Here we create an interface on the tool window...
    public interface IMyToolObject
    {
        void MyMethod();
    }
    //
    // Here we inherit the interface into the UserControl1 tool
    window...
    public partial class UserControl1 : UserControl, IMyToolObject
    {
        public UserControl1()
        {
            InitializeComponent();
        }

        #region IMyToolObject Members
        //
        // and here we implement our method, which pops up a
    message box...
        public void MyMethod()
        {
            System.Windows.Forms.MessageBox.Show("Some method!");
        }
        #endregion
    }
}
```

See also Appendix C: Anatomy of an Add-In-provided ToolWindow, for where we are in the Add-In scheme of things.

Creating a Property Page

Select **Tools -> Options** in the Visual Studio IDE to view property pages. The scroll box of the displayed Options dialog presents information as stored in an Add-In's .addin XML file. The top-level elements are Category-level, under which exist SubCategory-level elements. These are represented in the .addin file by <Category> and <SubCategory> XML tags bounded by a <ToolsOptionsPage> XML tag.

Right click your Add-In project in the Solution Explorer and choose Add -> UserControl to add another .NET user control to your project. Select controls from the tool box for inclusion in the new form. Then, add the following XML to your project's .addin file:

```
...
<ToolsOptionsPage>
  <Category Name="Environment">
    <SubCategory Name="My Sub category">
      <Assembly>put the full path to your Add-In's dll
here</Assembly>
      <FullClassName>YourAddin.UserControl2</FullClassName>
    </SubCategory>
  </Category>
</ToolsOptionsPage>
</Extensibility>
```

Open the UserControl's source file. Note that the UserControl2 partial class inherits UserControl and EnvDTE.IDTToolsOptionsPage. Implement this interface as follows:

```
#region IDTToolsOptionsPage Members
//
// Called when properties viewed or changed
public void GetProperties(ref object PropertiesObject)
{
    System.Windows.Forms.MessageBox.Show("OnGetProperties");
}
//
// Called when the tool window is created
public void OnAfterCreated(EnvDTE.DTE DTEObject)
{
    System.Windows.Forms.MessageBox.Show("OnAfterCreated");
}
//
// Called if the user clicks CANCEL
public void OnCancel()
{
    System.Windows.Forms.MessageBox.Show("OnCancel");
}
```

```
//  
// Called if user clicks (?) (HELP)  
public void OnHelp()  
{  
    System.Windows.Forms.MessageBox.Show("OnHelp");  
}  
//  
// Called if user clicks OK  
public void OnOK()  
{  
    System.Windows.Forms.MessageBox.Show("OnOK");  
}  
#endregion
```

Execute your Add-In, navigate to Tools -> Options, and verify that the sub class 'My Sub Category' appears under the 'Environment' category. Observe that the methods fire as expected. See also Appendix C: Anatomy of an Add-In-Provided Tool Window.

Registering Add-ins

Add-ins, being .NET (or COM) objects, must be registered either with the Windows Registry (in the case of COM objects) or, new for Visual Studio 2005, described with an XML .addin file (.NET objects only). Installation of Add-ins that use the XML .addin approach requires that the .addin file be copied to a well-known Visual Studio directory defined for this purpose. Visual Studio, on startup, reads any such .addin file(s) and presents the Add-in information in the **Tools -> Add-in Manager** dialog box. COM object Add-ins must use the Windows Registry for registration. These include Add-Ins created by Visual Studio Add-In Wizard-generated C++ / ATL code. Add-ins produced in the other supported languages (Visual C#, Visual Basic, Visual J#, C++ / CLR) may register using the new .addin approach.

Visual Studio Packages

VsPackages, also known simply as Packages, are compiled units of Extensibility.

VsPackages, like Plug-ins, are also COM objects.

Note that Visual Studio right out of the box cannot produce VsPackages. Install the **Visual Studio SDK** to extend the types of Extensibility projects that may be created with Visual Studio to include Packages.

The following Extensibility project types are added to Visual Studio upon installation of the **Visual Studio SDK**:

- Visual Studio Integration Package
- Visual Studio Language Package
- Domain-specific Language Designer
- Domain-specific Language Setup
- Help Integration Wizard

Examples of Packages include the VB, VJ, and C# languages.

About the Visual Studio SDK

The Visual Studio SDK, previously known as the VSIP SDK, is a free download from Microsoft which adds additional extensibility functionality to Visual Studio, primarily in that it enables Package projects to be created. The SDK also installs tools in **Start -> Programs -> Visual Studio 2005 SDK -> (revision number)**.

The SDK allows developers to integrate tools, editors, designers, languages, and much more into Visual Studio 2005.

The various package types are added to **Files -> New -> Project -> Extensibility** by installing the SDK.

Power Toys and **Innovasys Help Authoring Tools** are optionally available during SDK install.

Download Sources:

See *Additional Resources* at the end of this document.

Using the SDK

VsPackage development requires Visual Studio SDK installation. VsPackages typically maintain state and configuration information in the Windows registry. To prevent instability of the system registry during development, an 'experimental hive' is used instead of the system registry.

A DLK (Developers License Key) is provided for local development, but a PLK (Package Load Key) is required for deployment.

Additional Resources

Online

Visual Studio 2005 Extensibility Center

<http://msdn2.microsoft.com/en-us/vstudio/aa700819.aspx>

Extensibility Guided Tour

http://msdn.microsoft.com/library/default.asp?url=/library/en-us/extensibility_guided_tour/visual%20studio%20extensibility.asp

Visual Studio Industry Partner Affiliate (VSIP)

<http://affiliate.vsimembers.com/affiliate/default.aspx>

Visual Studio SDK

September 2006 **Community Technology Preview (CTP)**

<http://www.microsoft.com/downloads/details.aspx?FamilyId=7E0FDD66-698A-4E6A-B373-BD0642847AB7&displaylang=en>

Printed

The following two books are mentioned in the online literature. I personally do not own these books and so cannot comment more about them:

Visual Studio Hacks (O'Reilly)

James Avery (Paperback, 478 pages - Mar 24, 2005)

ISBN-10: 0596008473

ISBN-13: 978-0-596-00847-5

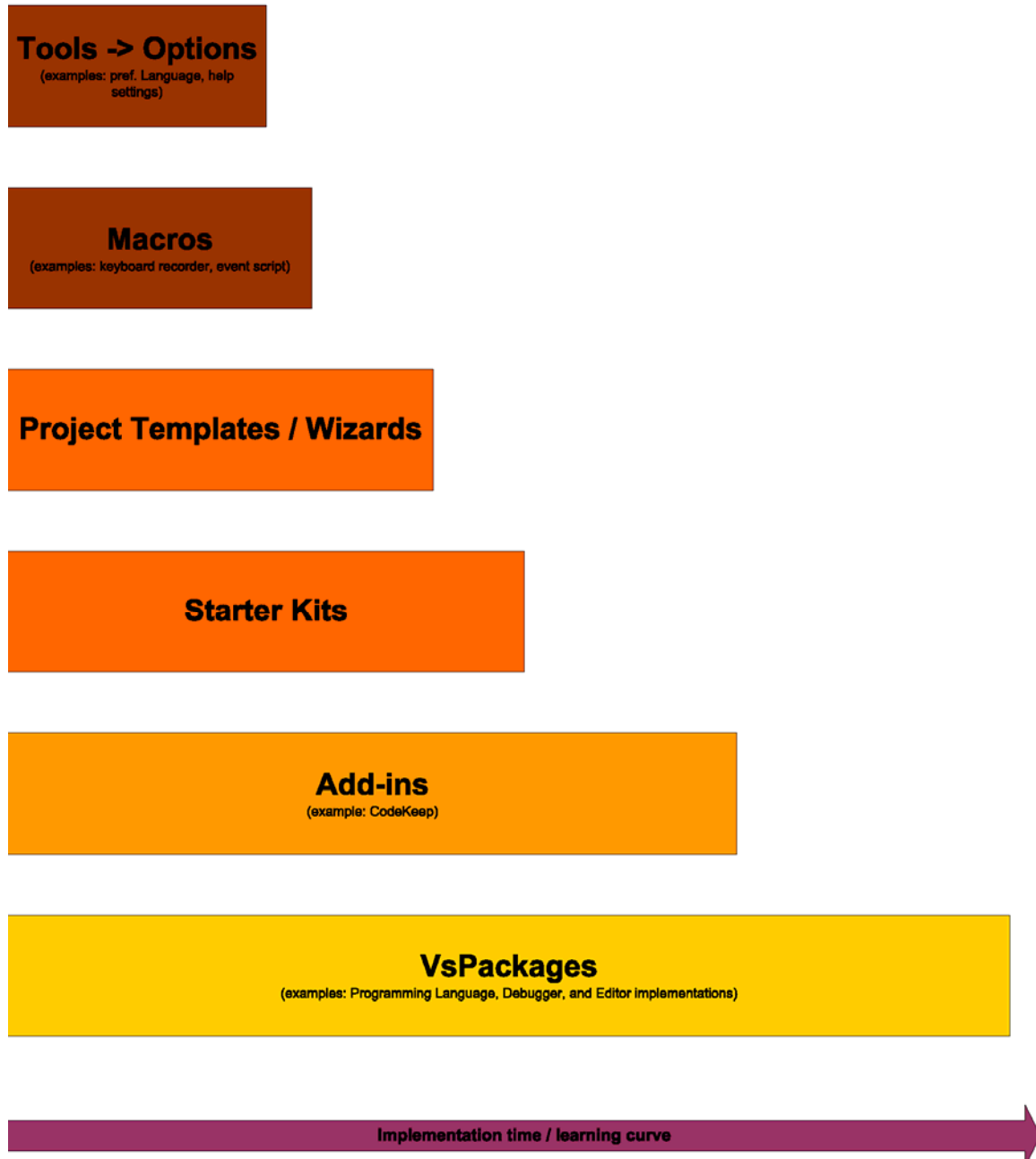
Windows Developer Power Tools (O'Reilly)

James Avery, Jim Holmes (Paperback, 1263 pages - December 21, 2006)

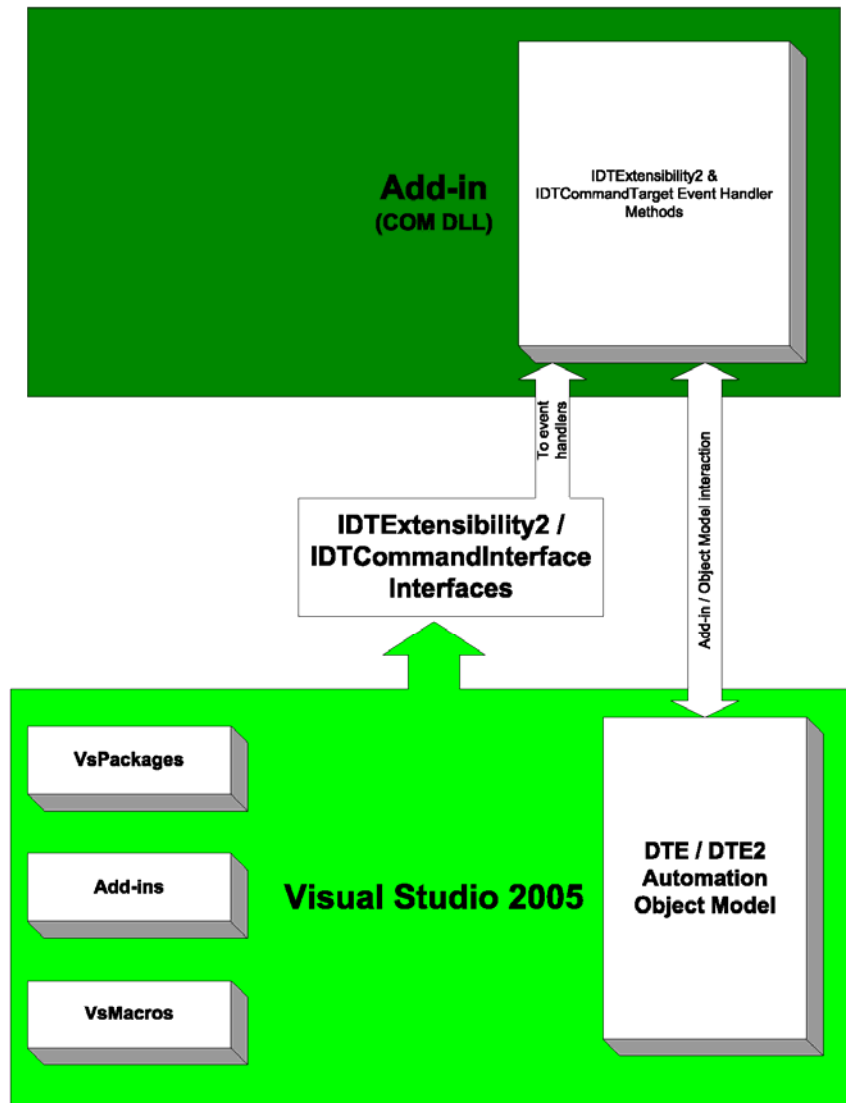
ISBN-10: 0596527543

ISBN-13: 978-0-596-52754-9

Appendix A: Visual Studio 2005 Extensibility- Relative Learning and Implementation Times

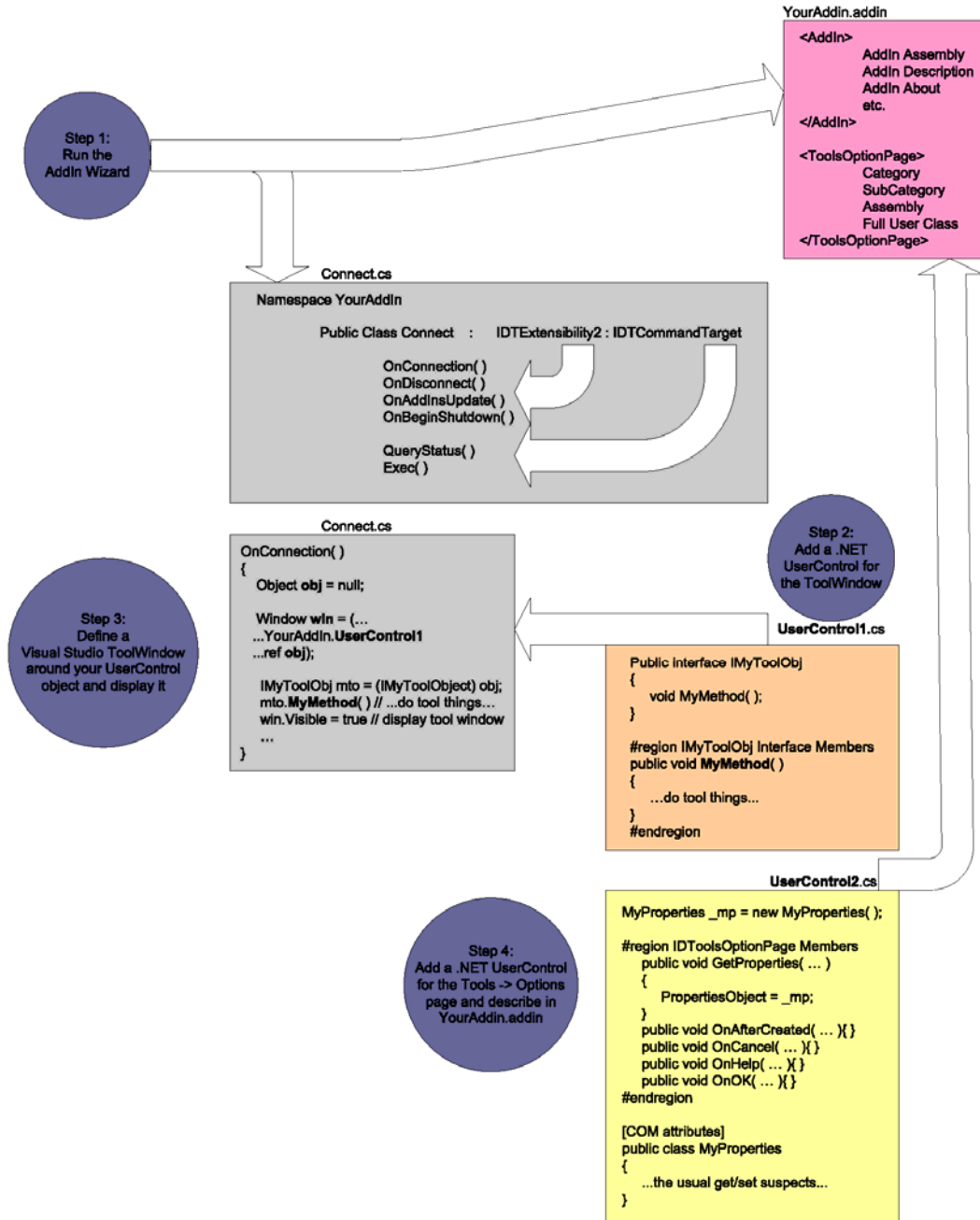


Appendix B: Visual Studio 2005 Add-in Environment



Visual Studio 2005 / Add-in Environment

Appendix C: Anatomy of an Add-In-provided Tool Window



Appendix D: Debugging the Extensibility Automation Chain

