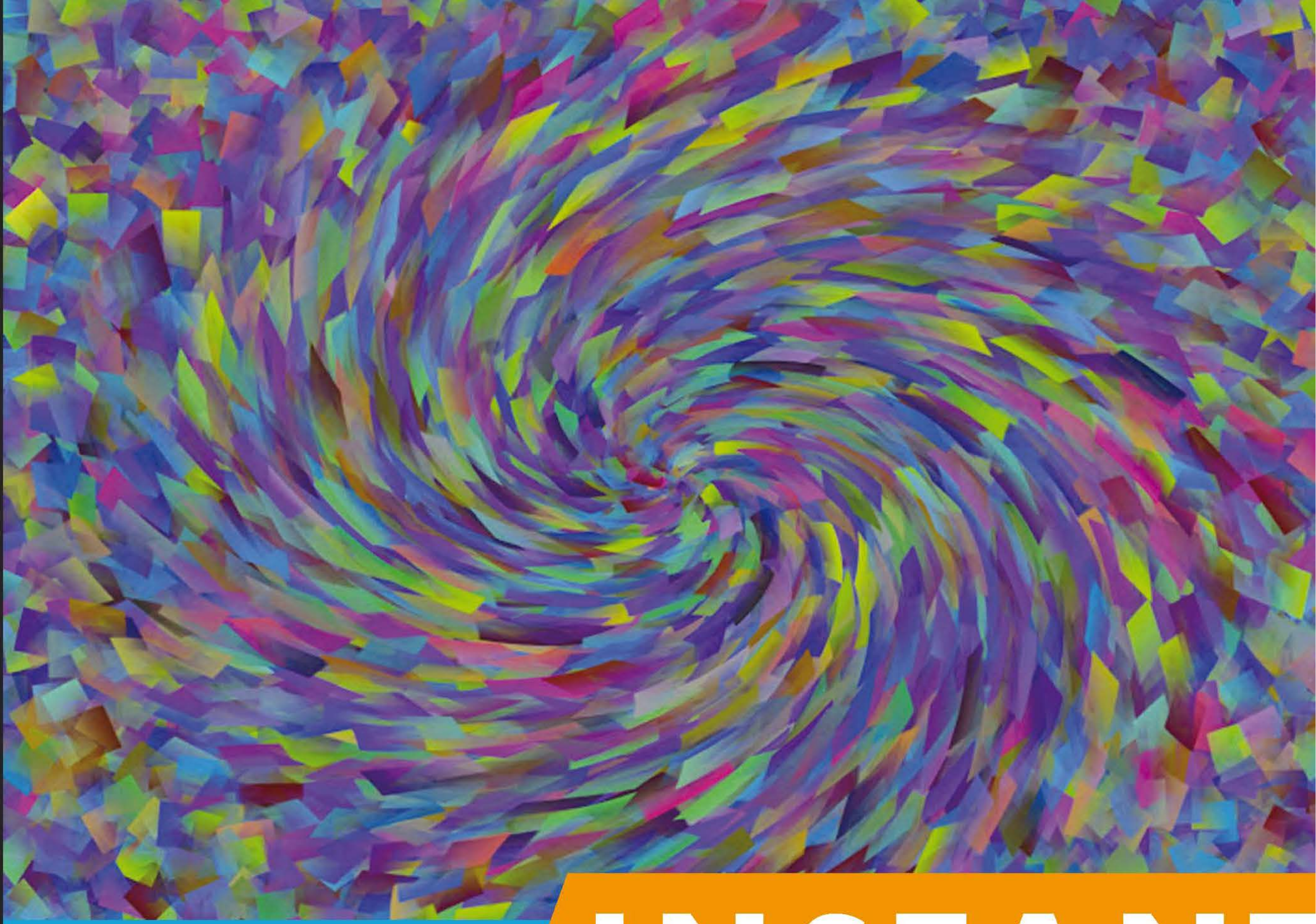


INSTANT

Short | Fast | Focused

Autodesk Revit 2013 Customization with .NET How-to

A supercharged guide to creating your own plugins, add-ons and customizations for Revit with .NET

Don Rudder

[PACKT] PUBLISHING

Instant Autodesk Revit 2013 Customization with .NET How-to

A supercharged guide to creating your own plugins, add-ons, and customizations for Revit with .NET

Don Rudder

[PACKT] PUBLISHING

BIRMINGHAM - MUMBAI

Instant Autodesk Revit 2013 Customization with .NET How-to

First published: January 2013

Production Reference: 1220113

Published by Packt Publishing Ltd.
Livery Place
35 Livery Street
Birmingham B3 2PB, UK.

ISBN 978-1-84968-842-0

www.packtpub.com

Credits

Author
Don Rudder

Reviewer
Harlan R. Brumm

Acquisition Editor
Andrew Duckworth

Commissioning Editor
Priyanka Shah

Technical Editor
Varun Pius Rodrigues

Project Coordinator
Abhishek Kori

Proofreader
Ting Baker

Production Coordinator
Aparna Bhagat

Cover Work
Aparna Bhagat
Prachali Bhiwandkar

Cover Image
Valentina D'silva

About the Author

Don Rudder is the Director of Software Development at CASE and focuses on the creation and management of specialized software and add-ins for various applications developed for client support. With over 16 years of experience in the AEC industry, Don has served well over 10 of those years as an HVAC and electrical designer for various MEP firms. He later began to focus more heavily on software development and related support where he eventually ended up in San Francisco serving as BIM Manager for HOK. He is self-taught in some 14 programming languages and well versed in .NET, web-based AEC tools, and pretty much any kind of automation. Don has also presented at Autodesk University and the Revit Technology Conference of North America.

Don has been the contributing author of the API chapters for *Mastering Autodesk Revit Architecture 2011*, *Mastering Autodesk Revit Architecture 2012*, and *Mastering Autodesk Revit Architecture 2013*.

I would like to thank my rock star friends and coworkers at CASE for just being awesome and bringing me into their mix. Without them, I would probably still be buried too deep in insignificant obligations without any real time to share stuff like what's in this book.

About the Reviewer

Harlan R. Brumm has a wide variety of experience within the architecture, engineering, and the software industries. He's been involved in training development, technical writing, program management, product management, and customer support. He has presented and taught internally to coworkers, at industry conferences, and at Autodesk University. Harlan worked for civil engineers and architects in the midwestern United States as an intern, CAD manager, and a project manager. His passion is the intersection of architectural design and technology.

He is an avid blogger and active in social media discussing all things BIM. Follow him on Twitter: `@HarlanBrumm`.

To my wife, Catie, and daughter, Maloa, for letting me take on many projects and supporting me to finish them.

Table of Contents

Preface 1

Instant Autodesk Revit 2013 Customization with .NET How-to 5

Getting started with the Autodesk Revit 2013 API (Must know) 6
Creating a simple command (Must know) 16
Adding a custom push button (Must know) 17
Element filtering (Must know) 24
Accessing the ProjectInfo data (Must know) 29
Extracting data (Should know) 31
Changing values (Must know) 40
Adding and removing parameters (Become an expert) 43
Creating plan views (Must know) 46
Creating a schedule (Become an expert) 50
Creating sheets and placeholders (Must know) 53
Placing views on sheets (Become an expert) 58
Wall color by length (Become an expert) 62
Subscribing and unsubscribing to events (Should know) 69

Preface

Welcome to *Instant Autodesk Revit 2013 Customization with .NET How-to*. This book will walk you through several of the Autodesk Revit 2013 API features in an easy to follow step-by-step process using powerful code samples. Each of the included code recipes have been designed to help get you right into some of the most common features of the Revit 2013 API.

What this book covers

Getting started with the Autodesk Revit 2013 API (Must know), introduces the fundamentals necessary to understand how add-ins get loaded into Revit and the basics for each of the three supported add-in project types.

Creating a simple command (Must know), describes a "Hello World" style introduction to Revit add-ins.

Adding a custom push button (Must know), describes how to make your own ribbon tab and add a basic push button used to launch a custom command. A brief textbox control sample is also discussed showing how to react to the `TextBoxEnterPressed` event.

Element filtering (Must know), introduces two different ways to filter elements in a Revit model.

Accessing the ProjectInfo data (Must know), describes how to access an element of a specific category and how to read data from properties bound to the element.

Extracting data (Should know), shows how to iterate over a set of elements within a common category and export values for specific parameters to an external CSV file.

Changing values (Must know), describes how to access element parameters and update their values.

Adding and removing parameters (Become an expert), describes how to load new shared parameters into the model and bind them to a category by name.

Creating plan views (Must know), introduces the process of adding new plan views to the model using the new and improved API classes for view generation.

Creating a schedule (Become an expert), describes a new API feature in Revit 2013 that enables the creation of design schedules.

Creating sheets and placeholders (Must know), demonstrates how to create sheets as well as non-graphical placeholder sheets.

Placing views on sheets (Become an expert), introduces the process of placing an existing view on an existing sheet.

Wall color by length (Become an expert), demonstrates how to use the analysis framework to display data graphically on native wall elements by their length.

Subscribing and unsubscribing to events (Should know), explains how to subscribe and unsubscribe from events based on criteria that you provide in this sample.

Dynamically enable a control (Become an expert), describes how to enable or disable a custom ribbon control based on the existence of an element of a required category in the current selection set. This recipe is not present in the book but is available as a free download from the following link: `http://www.packtpub.com/sites/default/files/downloads/8420OT_Bonus_Chapter.pdf`

Creating, loading, and placing a family (Must know), introduces how to create a new family document from an existing family template, generate a basic box extrusion, and load the family into an existing model document. This recipe is not present in the book but is available as a free download from the following link: `http://www.packtpub.com/sites/default/files/downloads/8420OT_Bonus_Chapter.pdf`

What you need for this book

You need to have Autodesk Revit 2013 and Microsoft Visual Studio 2010 Professional installed. You can download a 30-day trial of Autodesk Revit 2013 from the Autodesk website at `usa.autodesk.com/revit/trial/`. It should be noted that none of the sample code provided is supported on Autodesk Revit LT. A 30-day trial of Microsoft Visual Studio Professional can be downloaded from `www.microsoft.com/visualstudio`.

Who this book is for

This book targets design professionals with an intermediate to advanced working knowledge of Autodesk Revit 2013 with some existing .NET programming knowledge. If you are a beginner programmer but are an advanced Revit user, this book may still be suitable for you.

Conventions

In this book, you will find a number of styles of text that distinguish between different kinds of information. Here are some examples of these styles, and an explanation of their meaning.

Code words in the text are shown as follows: "`GetMaterialLINQ` queries the list using a single query command and as a result is quite a bit quicker at gathering materials by name."

A block of code is set as follows:

```
Dim m_app As UIApplication
m_app = commandData.Application
Dim m_doc As Document
m_doc = m_app.ActiveUIDocument.Document
```

When we wish to draw your attention to a particular part of a code block, the relevant lines or items are set in bold:

```
Dim m_app As UIApplication
m_app = commandData.Application
Dim m_doc As Document
m_doc = m_app.ActiveUIDocument.Document
```

Any command-line input or output is written as follows:

```
copy "$(ProjectDir)Revit2013Samples_VB.addin" "$(AppData)\Autodesk\REVIT\
Addins\2013\
```

New terms and **important words** are shown in bold. Words that you see on the screen, in menus or dialog boxes for example, appear in the text like this: "clicking the **Next** button moves you to the next screen".

Warnings or important notes appear in a box like this.

Tips and tricks appear like this.

Reader feedback

Feedback from our readers is always welcome. Let us know what you think about this book—what you liked or may have disliked. Reader feedback is important for us to develop titles that you really get the most out of.

To send us general feedback, simply send an e-mail to `feedback@packtpub.com`, and mention the book title via the subject of your message.

If there is a book that you need and would like to see us publish, please send us a note in the **SUGGEST A TITLE** form on `www.packtpub.com` or e-mail `suggest@packtpub.com`.

If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, see our author guide on `www.packtpub.com/authors`.

Customer support

Now that you are the proud owner of a Packt book, we have a number of things to help you to get the most from your purchase.

Downloading the example code

You can download the example code files for all Packt books you have purchased from your account at `http://www.PacktPub.com`. If you purchased this book elsewhere, you can visit `http://www.PacktPub.com/support` and register to have the files e-mailed directly to you.

Errata

Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you find a mistake in one of our books—maybe a mistake in the text or the code—we would be grateful if you would report this to us. By doing so, you can save other readers from frustration and help us improve subsequent versions of this book. If you find any errata, please report them by visiting `http://www.packtpub.com/support`, selecting your book, clicking on the **errata submission form** link, and entering the details of your errata. Once your errata are verified, your submission will be accepted and the errata will be uploaded on our website, or added to any list of existing errata, under the Errata section of that title. Any existing errata can be viewed by selecting your title from `http://www.packtpub.com/support`.

Piracy

Piracy of copyright material on the Internet is an ongoing problem across all media. At Packt, we take the protection of our copyright and licenses very seriously. If you come across any illegal copies of our works, in any form, on the Internet, please provide us with the location address or website name immediately so that we can pursue a remedy.

Please contact us at `copyright@packtpub.com` with a link to the suspected pirated material.

We appreciate your help in protecting our authors, and our ability to bring you valuable content.

Questions

You can contact us at `questions@packtpub.com` if you are having a problem with any aspect of the book, and we will do our best to address it.

Instant Autodesk Revit 2013 Customization with .NET How-to

Welcome to *Instant Autodesk Revit 2013 Customization with .NET How-to*. We all know how rewarding it can be to develop useful software tools. Becoming efficient at automating and extending the functionality of Revit can result in tremendous cost savings for your firm. The more people in your organization that use the tools you develop will add to these savings resulting in an even more impressive ROI. By the time you finish this book and complete each of the samples you will be fully capable and hopefully inspired to continue developing your own API projects for Autodesk Revit 2013 and beyond.

I've made a few assumptions throughout the book pertaining to your programming and general Revit capabilities. If you should ever find yourself in need of an explanation or definition for any of the Revit terms or concepts used in these writings, consult the online Autodesk Revit 2013 help files. An assumption has also been made that you have at least a basic understanding of programming with the Microsoft .NET Framework. If you're brand new to programming in .NET you should do just fine so long as you follow the instructions and samples closely.

Each of the provided code samples require that you have a Microsoft .NET 4.0 compatible interactive development environment (IDE) installed. Microsoft Visual Studio 2010 Professional was used to develop each of the samples used in this book. You can download a 30-day trial of Microsoft Visual Studio Professional from the Microsoft website at `http://www.microsoft.com/visualstudio/en-us/try`.

Most of the sample code is written in Visual Basic .NET with some being in C#. You can convert the samples between these two languages if you like using the freely accessible website at `http://www.developerfusion.com/tools/convert/vb-to-csharp/`. Converting the samples from one language to another will require further tweaking beyond the automated output of the translator in order to get them to function as intended.

Getting started with the Autodesk Revit 2013 API (Must know)

In this recipe you will learn about the fundamentals required to begin building your own API projects. Since the Revit API is so vast and way beyond the scope of this book, we'll just focus on the key points to get you started. Understanding the topics in this recipe is essential to your understanding of some of the more complex topics in later recipes.

The Revit 2013 API consists of two Microsoft .NET 4.0 compatible libraries that will need to be referenced into your API projects. Both of these libraries are found in the `Program` folder of your Revit installation. When referencing these libraries into your projects it is important to always set their `Copy Local` property to `False`.

`RevitAPI.dll` contains all of the database level namespaces required to interact with model documents.

`RevitAPIUI.dll` contains namespaces to the user interface features such as ribbon controls and native Revit styled dialog boxes.

The three API project types

The Revit 2013 API only supports the in-process DLLs which can only run inside the process of its host client, which in this case is Revit. It is not yet and may never be possible to create API utilities for Revit that run in their own process or as their own executable.

Three distinctly different Revit 2013 API project interface types are possible, each different in scope and capability. We will build at least one of each of these project types in this book to demonstrate their use.

- **IExternalCommand interfaces**: These interfaces execute upon user initiated command clicks. Data held in memory throughout the execution of the command does not persist and is destroyed upon the completion of the command execution.
- **IExternalApplication interfaces**: These interfaces support the addition of custom user controls to the ribbon as well as document-level events. Data held in memory by applications will persist throughout the entire session of Revit.
- **IExternalDbApplication interfaces**: These interfaces are used to subscribe to document-level events only and do not support any interactions to the `RevitAPIUI.dll` namespaces. Database-level applications do not support the addition of user interface controls or dialogs. Data held in memory by database-level applications will persist throughout the entire session of Revit.

Transactions

Transactions are a common feature of data interchange technologies and serve as a kind of watchdog preventing failed or unwanted changes to write to a data source. Transactions are mandatory every time an attempt is being made to change data in a Revit document. Read-only queries do not require a transaction.

Returning results

We will always be required to return one of the three supported return types in each of the three Revit 2013 API implementation types.

- **Result.Succeeded**: This will allow changes to be made to the model upon successful execution of the code. This return value will always be the last line of code between the main implementation's `Try` and `Catch` statements.
- **Result.Canceled**: This will not allow changes to be committed to the model and is typically used when the user cancels a command.
- **Result.Failed**: This will not allow any changes to the model and will display a Revit-styled message box containing the text set to the message parameter of an `IExternalCommand` implementation. This return value is always placed between a `Try` block's `Catch` and `End Try` statements.

The manifest file

Manifest files are XML-formatted ASCII files with a `.addin` file extension used to register API projects into Revit. The complete documentation for manifest files can be found in the document entitled `Getting Started with the Revit API.doc` in the Revit 2013 SDK.

Manifest files are only read into Revit during the launch of a new session. Revit will scan two directories for the `.addin` files to load at application start-up. Placing the manifest file in the `C:\ProgramData \Autodesk\Revit\Addins\2013\` directory in Windows 7 will make the add-in available for all users on the local machine and will require administrative privileges to modify. Placing the manifest file in the `%USERPROFILE%\Application Data\Autodesk\Revit\Addins\2013\` directory in Windows 7 will make the add-in available for the current user only and will not require administrative privileges to modify.

A sample manifest file named `Revit2013Samples_VB.addin` is shown next containing the minimum required parameters for each of the three API project types. Notice how each `Addin` tag is embedded within a single `RevitAddins` tag. This is essential when using a single manifest file to load multiple commands, applications, or database-level applications. The first `AddIn` is a command, then an application, and lastly a database-level application:

```
<?xml version="1.0" encoding="utf-8"?>
<RevitAddIns>
  <AddIn Type="Command">
    <Text>Command Revit 2013 Samples VB</Text>
    <Assembly>Revit2013Samples_VB.dll</Assembly>
```

```
        <FullClassName>
           Revit2013Samples_VB.CommandBoilerPlate
        </FullClassName>
        <ClientId>17a6cb83-8563-4dc7-a026-97fe519211a0
        </ClientId>
        <VendorId>XXXX</VendorId>
        <VendorDescription>
           Packt Publishing Samples for
           'Autodesk Revit 2013 Customization with .NET'
        </VendorDescription>
      </AddIn>
      <AddIn Type="Application">
        <Name>Application Revit2013Samples_VB</Name>
        <Assembly>Revit2013Samples_VB.dll</Assembly>
        <ClientId>0c78ba8a-8b96-4d31-b81b-0b77766ff38c
        </ClientId>
        <FullClassName>
           Revit2013Samples_VB.ApplicationBoilerPlate
        </FullClassName>
        <VendorId>XXXX</VendorId>
        <VendorDescription>
           Packt Publishing Samples for
           'Autodesk Revit 2013 Customization with .NET'
        </VendorDescription>
      </AddIn>
      <AddIn Type="DBApplication">
        <Name>Application Revit2013Samples_VB</Name>
        <Assembly>Revit2013Samples_VB.dll</Assembly>
        <ClientId>0c75ba8a-8b9c-4d31-b81b-0b77766f138c
        </ClientId>
        <FullClassName>
           Revit2013Samples_VB.DBApplicationBoilerPlate
        </FullClassName>
        <VendorId>XXXX</VendorId>
        <VendorDescription>
           Packt Publishing Samples for
           'Autodesk Revit 2013 Customization with .NET'
        </VendorDescription>
      </AddIn>
    </RevitAddIns>
```

Downloading the example code

You can download the example code files for all Packt books you have purchased from your account at `http://www.PacktPub.com`. If you purchased this book elsewhere, you can visit `http://www.PacktPub.com/support` and register to have the files e-mailed directly to you.

The Autodesk Revit 2013 SDK

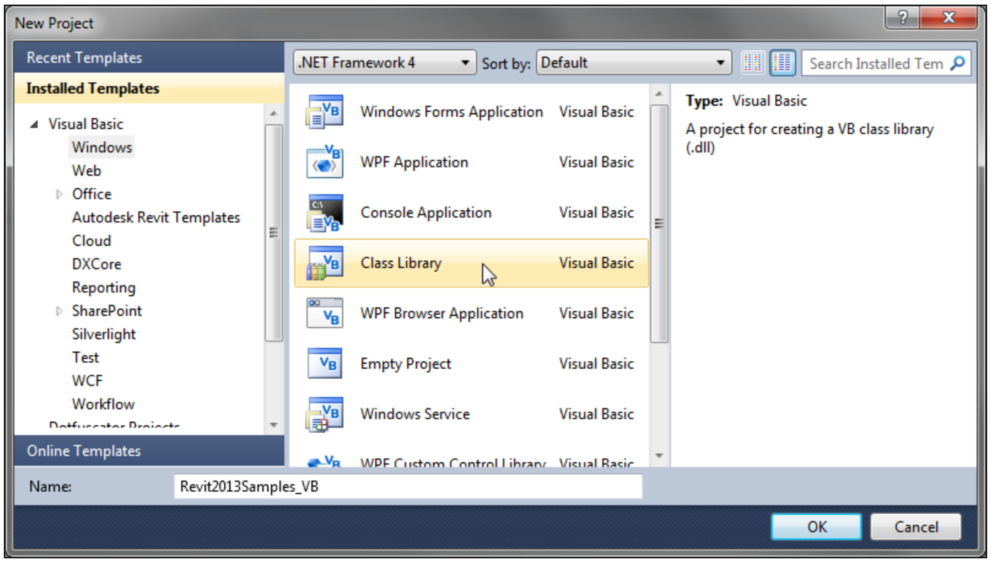
The official Revit 2013 Software Development Kit (SDK) is included in the installation media in each of the Revit product flavors. It can also be downloaded from the Autodesk website at `http://images.autodesk.com/adsk/files/revit2013sdk0.exe`. The SDK contains several code samples in both VB and C#.

Getting ready

It is common to build a single API project containing several commands and applications, but Revit API commands and applications each require their entry point be isolated into their own classes. In this recipe, we will create boilerplate versions of each of the three supported API project types for Revit 2013.

How to do it...

1. Open Visual Studio 2010 and create a new **Class Library** project named `Revit2013Samples_VB` and save it to your local hard drive.

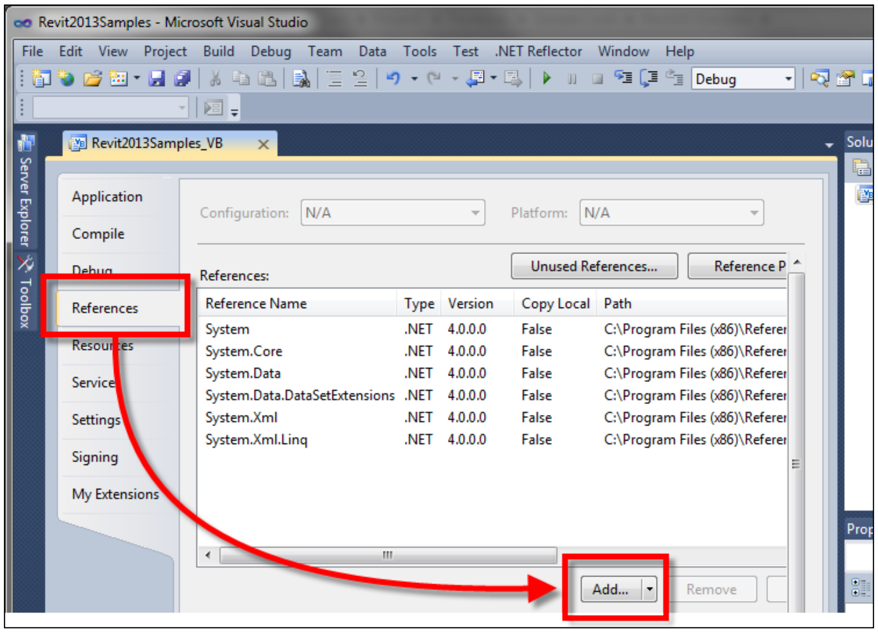

New Project
Recent Templates
Installed Templates
Visual Basic
Windows
Web
Office
Autodesk Revit Templates
Cloud
DXCore
Reporting
SharePoint
Silverlight
Test
WCF
Workflow
Online Templates
.NET Framework 4
Sort by: Default
Search Installed Tem
Windows Forms Application Visual Basic
WPF Application Visual Basic
Console Application Visual Basic
Class Library Visual Basic
WPF Browser Application Visual Basic
Empty Project Visual Basic
Windows Service Visual Basic
WPF Custom Control Library Visual Basic
OK
Cancel

Type: Visual Basic

A project for creating a VB class library (.dll)

Name: Revit2013Samples_VB

2. Open the **Property Pages** window by right-clicking on the project name in **Solution Explorer** and selecting **Properties**.
3. Click on the **References** tab in the properties view and click on the **Add...** button.

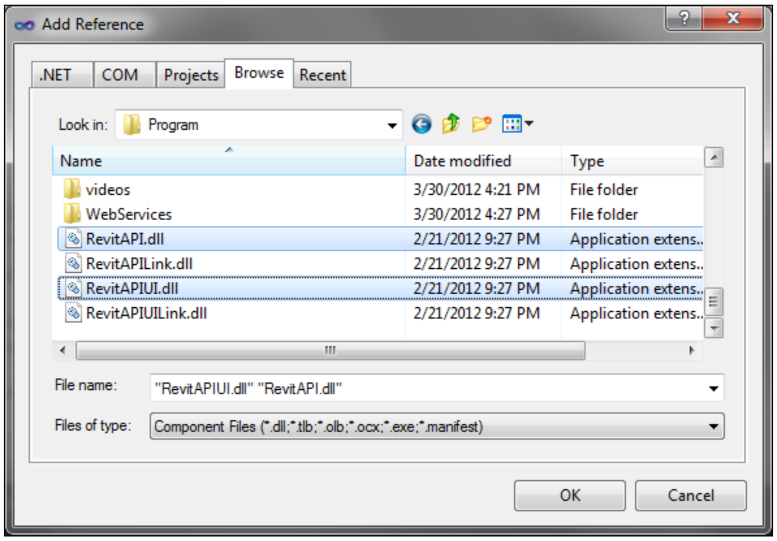


Configuration: N/A Platform: N/A

References:

Reference Name	Type	Version	Copy Local	Path
System	.NET	4.0.0.0	False	C:\Program Files (x86)\Referer
System.Core	.NET	4.0.0.0	False	C:\Program Files (x86)\Referer
System.Data	.NET	4.0.0.0	False	C:\Program Files (x86)\Referer
System.Data.DataSetExtensions	.NET	4.0.0.0	False	C:\Program Files (x86)\Referer
System.Xml	.NET	4.0.0.0	False	C:\Program Files (x86)\Referer
System.Xml.Linq	.NET	4.0.0.0	False	C:\Program Files (x86)\Referer

4. Select the **Browse** tab and add the two Revit namespace libraries `RevitAPI.dll` and `RevitUIAPI.dll` from the `Program` directory of your Revit 2013 installation.



Look in: Program

Name	Date modified	Type
videos	3/30/2012 4:21 PM	File folder
WebServices	3/30/2012 4:27 PM	File folder
RevitAPI.dll	2/21/2012 9:27 PM	Application extens..
RevitAPILink.dll	2/21/2012 9:27 PM	Application extens..
RevitAPIUI.dll	2/21/2012 9:27 PM	Application extens..
RevitAPIUILink.dll	2/21/2012 9:27 PM	Application extens..

File name: "RevitAPIUI.dll" "RevitAPI.dll"

Files of type: Component Files (*.dll;*.tlb;*.olb;*.ocx;*.exe;*.manifest)

5. Select the two Revit namespace libraries that you just imported in the **References** tab of the properties view and set their property for `Copy Local` to `False`.
6. Click on the **Debug** tab of the properties view and under **Start Action** set the **Start external program:** option **active** and enter the path to your `Revit.exe`. This file is typically located at `C:\Program Files\Autodesk\Revit Architecture 2013\Program\Revit.exe` depending on the flavor of Revit that you have installed.
7. Select the `Class1.vb` file in the solution explorer and change the property for **File Name** in the properties palette to `CommandBoilerPlate.vb`.
8. Enter the following boilerplate code into the `CommandBoilerPlate.vb` file:

```
Imports Autodesk.Revit.DB
Imports Autodesk.Revit.UI
Imports Autodesk.Revit.Attributes

''' <summary>
''' Sample Command
''' </summary>
<Transaction(TransactionMode.Manual)>
Public Class CommandBoilerPlate

    ' Required Implementation
    Implements IExternalCommand

    ''' <summary>
    ''' Revit UI Entry Point for a Command
    ''' </summary>
    Public Function Execute(commandData As _
                    ExternalCommandData,
                    ByRef message As String,
                    elements As ElementSet) _
                        As Result Implements _
           IExternalCommand.Execute

        Try
            ' Begin Code Here
            ' Return Success
            Return Result.Succeeded
        Catch ex As Exception
            ' In Case of a Failure
            message = ex.Message
            Return Result.Failed
        End Try
    End Function
End Class
```

9. Add a new class named `ApplicationBoilerPlate.vb` and enter the following boilerplate code:

```
Imports Autodesk.Revit.DB
Imports Autodesk.Revit.UI
Imports Autodesk.Revit.DB.Events
Imports Autodesk.Revit.ApplicationServices
Imports Autodesk.Revit.Attributes

''' <summary>
''' Sample Application
''' </summary>
<Transaction(TransactionMode.Manual)>
Public Class ApplicationBoilerPlate

    ' Required Implementation
    Implements IExternalApplication

    ''' <summary>
    ''' Runs when Revit Shuts Down
    ''' </summary>
    Public Function OnShutdown(application As _
           UIControlledApplication) _
                    As Result Implements _
           IExternalApplication.OnShutdown

        Try
            ' Begin Code Here
            ' Return Success
            Return Result.Succeeded
        Catch
            ' In Case of Failure
            Return Result.Failed
        End Try
    End Function

    ''' <summary>
    ''' Runs when Revit Starts Up
    ''' </summary>
    Public Function OnStartup(application As _
           UIControlledApplication) _
                    As Result Implements _
           IExternalApplication.OnStartup

        Try
```

```
                ' Begin Code Here

                ' Return Success
                Return Result.Succeeded

            Catch

                ' In Case of Failure
                Return Result.Failed

            End Try
        End Function
    End Class
```

10. Add a new class named `DBApplicationBoilerPlate.vb` and enter the following boilerplate code:

```
Imports Autodesk.Revit.DB
Imports Autodesk.Revit.UI
Imports Autodesk.Revit.DB.Events
Imports Autodesk.Revit.ApplicationServices
Imports Autodesk.Revit.Attributes

''' <summary>
''' Sample Database Level Application
''' </summary>
<Transaction(TransactionMode.Manual)>
Public Class DBApplicationBoilerPlate

    ' Required Implementation
    Implements IExternalDBApplication

    ''' <summary>
    ''' Runs when Revit Shuts Down
    ''' </summary>
    Public Function OnShutdown(application As _
           ControlledApplication) _
            As ExternalDBApplicationResult _
            Implements IExternalDBApplication.OnShutdown

        Try
            ' Begin Code Here
            ' Return Success
            Return Result.Succeeded
        Catch
            ' In Case of Failure
            Return Result.Failed
        End Try
    End Function

    ''' <summary>
    ''' Runs when Revit Starts Up
    ''' </summary>
    Public Function OnStartup(application As _
           ControlledApplication) _
            As ExternalDBApplicationResult _
            Implements IExternalDBApplication.OnStartup

        Try
            ' Begin Code Here
            ' Return Success
            Return Result.Succeeded
        Catch
            ' In Case of Failure
            Return Result.Failed
        End Try
    End Function
End Class
```

How it works...

Did you notice the similarities between each of the three sample classes before?

The two namespace imports for `Autodesk.Revit.DB` and `Autodesk.Revit.UI` should always be listed at the very top of any code file that makes a reference to these namespaces. This prevents you from having to include the full namespaces for classes each time you call them in your projects.

`TransactionMode` is a required attribute for each Revit 2013 API project type and should always be placed directly above the class declaration. The `Manual` mode has been set for this attribute on each class since this will be the only supported option in future releases of the Revit API.

Each class begins with an implementation. These implementations contain the functions necessary to initiate and conclude connectivity to the Revit API for the command or application type.

The `Try` blocks should always be used to encapsulate the main functionality of your project. `Return Result.Succeeded` should always be the last line prior to `Catch` while `Return Result.Failed` should always be between the `Catch` and `End Try` statements. Following this strategy will help prevent model corruption when code should fail for any reason.

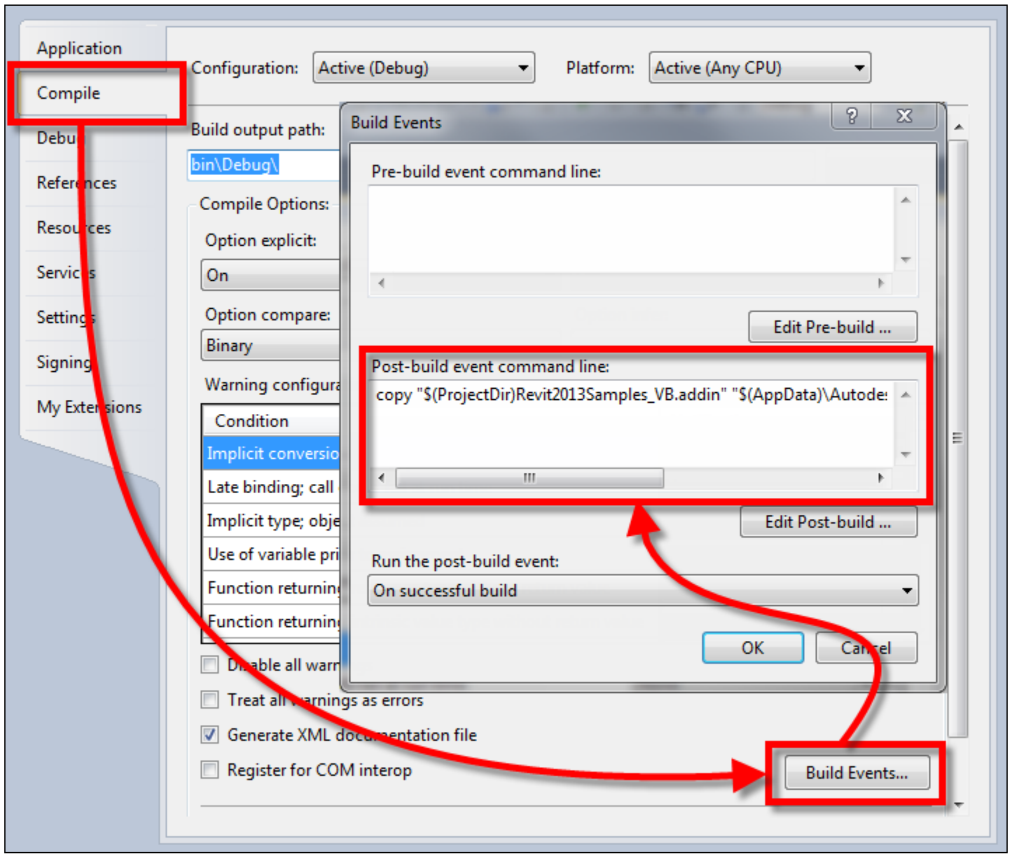
There's more...

Did you know that you can automatically set your manifest file(s) to copy to their installation directory on each successful build and debug event? Open up the properties of your solution and in the **Compile** tab click on the **Build Events...** button in the lower right corner.

Enter the following command into the **Post-build event command line:** area:

```
copy "$(ProjectDir)Revit2013Samples_VB.addin" "$(AppData)\Autodesk\REVIT\
Addins\2013\Revit2013Samples_VB.addin"
```

Now each time you debug your solution, the manifest file will get copied into the `Addins` directory so your project DLL will load into the Revit session.



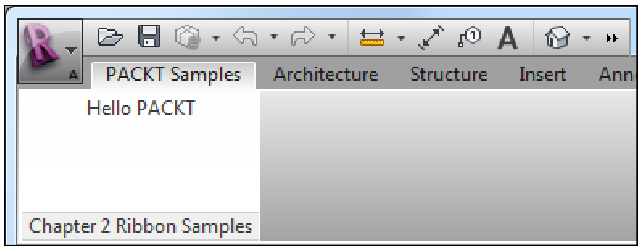
copy "$(ProjectDir)Revit2013Samples_VB.addin" "$(AppData)\Autode

You may have used third party tools for Revit that utilize their own custom ribbon tabs with custom buttons and controls for accessing the tools within the utility. There are a number of options you have for customizing ribbon tabs and adding your own user controls for accessing commands within your applications. This section will focus on some of the more common controls available to you for interacting with commands.

In this section we will build a custom `IExternalApplication` that adds a pair of custom user controls to a custom ribbon tab where we can access a sample hello world style `IExternalCommand` class that we will build in this section as well.

Understanding a few important ground rules will help you understand custom Revit ribbon user controls. Ribbon controls can only be added from the `IExternalApplication` classes and the only controls possible to use are made available from within the `RevitAPIUI.dll` library. A full list of supported ribbon controls can be found in the `RevitAPI.chm` document of the SDK under `Autodesk.Revit.UI`.

You will hear the terms tab and panel used throughout the samples in this chapter. Tabs contain panels. The following screenshot shows a custom tab named **PACKT Samples** containing a panel named **Chapter 2 Ribbon Samples**. I cheated a little and held down the *Ctrl* key and dragged the tab to the far left for the sake of image clarity. Custom tabs will typically be at the far right-hand side of the tab listing.



Creating a simple command (Must know)

We will create a very boring and perfectly useless sample command that will display a basic **Hello PACKT** message box to indicate success when the user initiates the command from the custom ribbon controls.

Getting ready

Now that our sample code files are beginning to stack up in our solution, it is a good time to organize them into directories within our solution making them easier to find. Create a directory in your solution named `Ch1 Intro to Revit API` and place all of the boilerplate code classes that we created in the previous recipe into this directory. Create another directory named `Ch2 Ribbon` and place each of the code class samples you create in this recipe into this directory.

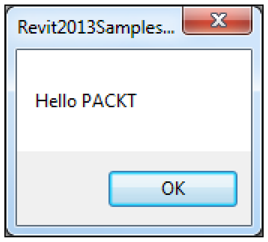
How to do it...

1. Create a new class in the `Ch2 Ribbon` directory named `CommandHelloPACKT.vb`.
2. Copy and paste the boilerplate code from the `IExternalCommand` example that we created in `Ch1 Intro to Revit API` into this file but be sure that our new class name remains `CommandHelloPACKT`.
3. The only difference between the boilerplate code and our new class other than the class name will be the addition of one single line of code just beneath the `' Begin Code Here` line within the `Execute` function:

```
' Begin Code Here
MsgBox("Hello PACKT")
```

How it works...

When the user initiates the command, a standard .NET message box class is displayed making it easy to notice when the command has been executed. The following screenshot shows just how this message box will look:



Adding a custom push button (Must know)

Now that we have a fundamental grasp of ribbon controls, let's create a push button to access the command we just created. In this recipe, we will build a reusable function that we will use to add a push button control to a custom tab in the Revit ribbon.

Getting ready

Remember the manifest file and how to add new applications to it? We need to edit our `Revit2013Samples_VB.addin` manifest file so that our new application class will load into the Revit session. Add the following block of code inside the `RevitAddIns` attribute being careful not to nest it inside any other `Command`, `Application`, or `DBApplication` tags:

```
<AddIn Type="Application">
  <Name>Application Ribbon</Name>
  <Assembly>Revit2013Samples_VB.dll</Assembly>
```

```
    <ClientId>
      0c78ba8b-8b96-4d31-b81b-0b77766ff38d
    </ClientId>
    <FullClassName>
      Revit2013Samples_VB.ApplicationRibbon
    </FullClassName>
    <VendorId>XXXX</VendorId>
    <VendorDescription>
      Packt Publishing Samples for 'Autodesk Revit 2013 Customization
with .NET'
    </VendorDescription>
  </AddIn>
```

Now with the manifest file edited to load our new application into Revit when we debug the project, we can create the application class and assemble the code that will load our custom controls into the ribbon.

How to do it...

1. Adding custom ribbon controls requires a few additional namespace references to your project in order to accommodate button images. Open up your project's properties page and open the **References** tab.
2. Click on the **Add...** button and set the **.NET** tab active.
3. Add **PresentationCore**, **System.Xaml**, and **WindowsBase** then click on **OK**.
4. Add a new class in the **Ch2 Ribbon** folder named **ApplicationRibbon.vb**.
5. Copy and paste the boilerplate code for `IExternalApplication` and be sure that our new class name remains `ApplicationRibbon`.
6. A couple of new namespaces are required by this new ribbon sample that we did not have in our boilerplate code samples. Add the following two namespaces just beneath the ones that are already there:

   ```
   Imports System.Reflection
   Imports System.Windows.Media.Imaging
   ```

7. A reusable function for adding push buttons comes in handy in case we ever decide later that we want to add more than one button. Add the function shown next inside our new class beneath the `OnStartup` function:

   ```
     ''' <summary>
     ''' Add a PushButton to the Ribbon
     ''' </summary>
     Private Function AddPushButton(Panel As RibbonPanel,
                                   ButtonName As String,
                                       ButtonText As String,
                                       ImagePath16 As String,
   ```

```
                                  ImagePath32 As String,
                                  dllPath As String,
                                  dllClass As String,
                                  Tooltip As String) _
                  As Boolean

    Try

      ' PushButtonData Object
      Dim m_pbData As New PushButtonData _
        (ButtonName, ButtonText, dllPath, dllClass)

      ' Small Image
      If ImagePath16 <> "" Then
        Try
          m_pbData.Image = _
            New BitmapImage(New Uri(ImagePath16))
        Catch
          ' Couldn't Find the Image
        End Try
      End If

      ' Large Image
      If ImagePath32 <> "" Then
        Try
          m_pbData.LargeImage = _
            New BitmapImage(New Uri(ImagePath32))
        Catch
          ' Couldn't Find the Image
        End Try
      End If

      ' Set the Tooltip
      m_pbData.ToolTip = Tooltip

      ' Add it to the Panel
      Dim m_pb As PushButton = Panel.AddItem(m_pbData)

      ' Success
      Return True

    Catch ex As Exception

      ' Failure
      Return False

    End Try

  End Function
```

8. The next thing to do is update the `OnStartup` code with the necessary information to call the function. The completed `OnStartup` code is shown as follows:

```
''' <summary>
''' Runs when Revit Starts Up
''' </summary>
Public Function OnStartup(app As UIControlledApplication) _
        As Result Implements IExternalApplication.OnStartup

  Try

    ' Our Target Tab Name
    Dim m_tabName As String
    m_tabName = "PACKT Samples"

    ' First Create the Tab
    Try
      app.CreateRibbonTab(m_tabName)
    Catch ex As Exception
      ' Might already exist...
      '  Common when multiple applications target
      '  the same ribbon tab
    End Try

    ' Ribbon Panel Name
    Dim m_rpName As String
    m_rpName = "Chapter 2 Ribbon Samples"

    ' The Ribbon Panel
    Dim m_RibbonPanel As RibbonPanel = Nothing

    ' Get the Panel Within the Tab by Name
    Dim m_RP As New List(Of RibbonPanel)
    m_RP = app.GetRibbonPanels(m_tabName)
    For Each x As RibbonPanel In m_RP
      If x.Name.ToUpper = m_rpName.ToUpper Then
        m_RibbonPanel = x
        Exit For
      End If
    Next

    ' Add the Panel if it doesn't Exist
    If m_RibbonPanel Is Nothing Then
      m_RibbonPanel = app.CreateRibbonPanel(m_tabName,
                                            m_rpName)
    End If
```

```
            ' PushButtonData for Hellow World Sample
            If AddPushButton(m_RibbonPanel,
               "Ch2HelloPACKT",
               "Hello PACKT",
               "",
               "",
               "Revit2013Samples_VB.dll",
               "Revit2013Samples_VB.CommandHelloPACKT",
               "Show a simple message box 'Hello PACKT!'") _
               = False Then

              ' Tell the User it Failed
              MsgBox("Failed to Add PushButton Control!",
                     MsgBoxStyle.Critical,
                     "Error")

            End If

            ' Return Success
            Return Result.Succeeded

          Catch

            ' In Case of Failure
            Return Result.Failed

          End Try

        End Function
```

How it works...

The `OnStartup` code is fairly straightforward here in that it only adds a single control. First a tab named `PACKT Samples` is created with a panel named `Chapter 2 Ribbon Samples`. If the code is successful in returning a `RibbonPanel` object, then a call is made to add the `PushButton` control.

Only after a `RibbonPanel` object has successfully been created or retrieved by the code, is a `PushButton` object is added to the panel using the `AddPushButton` function. The `AddPushbutton` function returns `False` on failure giving us an opportunity to display an error message to the user if we want. Take a look at the arguments that this function consumes and notice that we did not pass a value for `ImagePath16` or `ImagePath32`. If we were to provide a full path to the image files accessible by our application to either of these two arguments, they would display as icons in the ribbon above the command text.

- **Panel**: It is the `RibbonPanel` object that we want to add the control into.
- **ButtonName**: It is the unique name for the button.

- **ButtonText**: It will be used as the text displayed in the user interface. If an image is specified, the button text will display beneath the button image.
- **ImagePath16**: It is used to load an image file for the small icon such as when it would be added to the QAT. The image used by this argument should be formatted as 16 x 16 pixels.
- **ImagePath32**: It is used to load an image file for the small icon, such as when it would be added to the QAT. The image used by this argument should be formatted as 32 x 32 pixels.
- **dllPath**: It should contain the full path to the DLL file that contains the command that we want the button to execute.
- **dllClass**: It is the full class name where the `IExternalCommand` interface is implemented.
- **Tooltip**: It will display when the command button is hovered over. It is suggested to always use tooltips to help keep your users out of the dark if the button text is not sufficient to inform first time users as to the button's purpose.

There's more...

The textbox control is another interesting control supported by the Revit API. A textbox control might come in handy if you wanted to make some sort of pseudo command line or to record comments to a log as you work in a model.

Adding a TextBox control

1. Add the `Autodesk.Revit.UI.Events` namespace to the top of the `ApplicationRibbon` class:

```
Imports Autodesk.Revit.UI.Events
```

2. Add the event handler subroutine inside the main class declaration area that we want to run when the *Enter* key is pressed inside the textbox control:

```
''' <summary>
''' TextBox Enter Event Handler
''' </summary>
Public Sub MyTextBoxEnter(ByVal sender As Object, _
                 ByVal args As TextBoxEnterPressedEventArgs)

  ' Message showing contents
  MsgBox(sender.value,
         MsgBoxStyle.Information,
         "PACKT Text Box")

End Sub
```

3. Enter the function for adding a textbox control to the ribbon. This function also includes the required event subscription for handling the *Enter* key in the textbox control:

```
''' <summary>
''' Add a textbox control to the ribbon
''' </summary>
Private Function AddTextBox(Panel As RibbonPanel,
                            tbName As String,
                            tbTooltip As String) As Boolean

  Try

    ' The TextBoxData Object
    Dim m_tbD As New TextBoxData(tbName)

    ' Add the Control
    Dim m_tb As TextBox
    m_tb = Panel.AddItem(m_tbD)

    ' Sample Text
    m_tb.PromptText = "I'm a Text Box!"

    ' Button
    m_tb.ShowImageAsButton = True

    ' Tooltip
    m_tb.ToolTip = tbTooltip

    ' Add the Handler for Enter Key
    AddHandler m_tb.EnterPressed, _
          New EventHandler(Of TextBoxEnterPressedEventArgs) _
          (AddressOf Me.MyTextBoxEnter)

    ' Success
    Return True

  Catch

    ' Failure
    Return False

  End Try

End Function
```

4. Add the call to `AddTextBox` inside the `OnStartup` function immediately above the line that reads `' Return Success`:

```
' Add a textbox
If AddTextBox(m_RibbonPanel,
              "tbPACKT",
              "Enter text and press the Enter key!") _
         = False Then

  ' Tell the User it Failed
  MsgBox("Failed to Add TextBox Control!",
         MsgBoxStyle.Critical,
         "Error")

End If
```

5. Run the add-in in the debugger and you'll see the new textbox control in the ribbon. Enter a text string into the textbox and hit *Enter* and you will get a message box pop up displaying the same text string that you entered into the textbox control.

Element filtering (Must know)

The Revit API offers quite a few options for finding elements in the active document. Some methods for collecting elements are quicker than others and it is important to understand how to gauge this performance between the various methods that you may want to be experimenting with. Filtering for elements by class type or category is an example of a very simple and straightforward filtering type and due to its simplicity can be performed quite efficiently. It is when you have a set of more strict filtering requirements that will typically result in slower element filtering performance, such as across multiple categories matching a specific component naming pattern.

The two filtering methods we will learn in this recipe are `OfCategory` using element iteration and LINQ. The **Language Integrated Query** (**LINQ**) method uses query syntax similar to **Structured Query Language** (**SQL**) used in database queries and can perform quite efficiently for some types of search requirements but poorly in others.

We will learn how to utilize the native .NET `StopWatch` class to time the performance of each filtering method so we can determine and use the method that performs most efficiently. If you are ever unsure as to which method will perform the most efficiently, test each scenario using `StopWatch` and know for sure which method is fastest.

Getting ready

We will follow the same directory organization strategy that we used in the previous recipe by placing all of the samples we create in this recipe into a directory named `Ch3 Elements` in our solution project. This sample will show two distinctly different ways of filtering objects using the same criteria. The first sample command will use the `FilteredElementCollector` method and the second using LINQ. We will utilize the .NET `StopWatch` class to time each of our filters and report the results to the user.

Since this sample implements `IExternalCommand` it requires its own entry into the `.addin` manifest file in order to run. Add the following entry to our `Revit2013Samples_VB.addin` file inside the `RevitAddIns` tag being careful not to nest it inside any other `Command`, `Application`, or `DBApplication`:

```
<AddIn Type="Command">
  <Text>
    Sample - Ch3 Element Filtering
  </Text>
  <Assembly>
    Revit2013Samples_VB.dll
  </Assembly>
  <FullClassName>
    Revit2013Samples_VB.CommandElementFiltering
  </FullClassName>
  <ClientId>14a6cba3-a563-4dc7-a026-47fe5c9a11a0</ClientId>
  <VendorId>XXXX</VendorId>
  <VendorDescription>
    Packt Publishing Samples for
    'Autodesk Revit 2013 Customization with .NET'
  </VendorDescription>
</AddIn>
```

How to do it...

1. Add a new class named `CommandElementFiltering` and copy the boilerplate code from `CommandBoilerPlate` into this class being careful to keep the new class name as `CommandElementFiltering`.
2. Add the following namespace to the top of the class just after the three namespaces already there to enable our `StopWatch` functionality:

   ```
   Imports System.Diagnostics
   ```

3. The first function that we will add to our new class will collect the list of materials using the traditional method of filtering by name. Add the `GetMaterial` code inside our `CommandElementFiltering` class after the `Execute` function:

```
''' <summary>
''' Get a specific materials by name
''' </summary>
Private Function GetMaterial(p_name As String,
                             p_doc As Document) _
                 As List(Of Material)

  ' The List
  Dim m_materials As New List(Of Material)

  ' The Collector
  Dim m_col As New FilteredElementCollector(p_doc)
  m_col.OfCategory(BuiltInCategory.OST_Materials)

  ' Find Matching Names
  For Each x In m_col.ToElements

    ' Does the Name Match
    If x.Name.ToLower.Contains(p_name) Then

      ' Add the Matching Item
      m_materials.Add(x)

    End If

  Next

  ' Return the List
  Return m_materials

End Function
```

4. The second function performs the same action but uses LINQ to find matching materials by name. Add the `GetMaterialLINQ` code after the `GetMaterial` function that you just added:

```
''' <summary>
''' Get a specific materials by name using LINQ
''' </summary>
Private Function GetMaterialLINQ(p_name As String,
                                 p_doc As Document) _
                 As List(Of Material)

  ' Fresh List
  Dim m_materials As New List(Of Material)

  ' The Collector
```

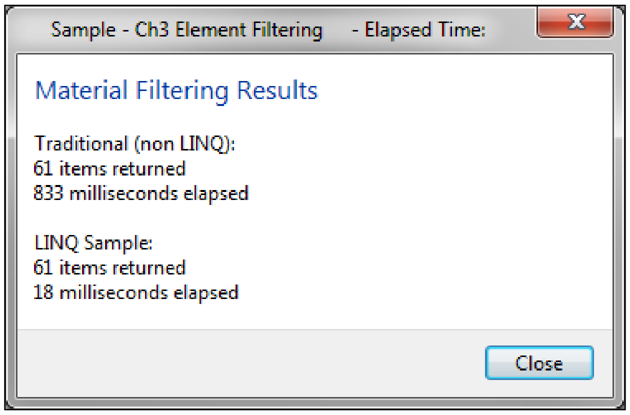
```
      Dim m_col As New FilteredElementCollector(p_doc)
      m_col.OfCategory(BuiltInCategory.OST_Materials)

      ' The non LINQ List
      Dim m_eList As IEnumerable(Of Element)
      m_eList = m_col.ToElements

      ' LINQ Selection by Name
      Dim MaterialElement = _
          From e In m_eList
          Where e.Name.ToLower.Contains(p_name)

      ' Any Matches?
      If MaterialElement.Count > 0 Then

        For Each x In MaterialElement

          ' Add the Item
          m_materials.Add(x)

        Next

      End If

      ' Return the List
      Return m_materials

    End Function
```

5. Now that we have our element filtering functions added to our class, we will need some code to call them from within the `Execute` function. Add the following code between the `' Begin Code Here` and `' Return Success` lines inside the `Execute` function that we pasted previously from the `CommandBoilerPlate` class we built in the previous recipe entitled *Getting started with the Autodesk Revit 2013 API (Must know)*:

```
        Dim m_doc As Document
        m_doc = _
          commandData.Application.ActiveUIDocument.Document

        ' Test #1 - Traditional
        Dim m_sw_1 As New Stopwatch
        m_sw_1.Start()

        ' Return the Values
        Dim m_materials_1 As New List(Of Material)
        m_materials_1 = GetMaterial("a", m_doc)
        m_sw_1.Stop()
```

```
' Test #2 - LINQ
Dim m_sw_2 As New Stopwatch
m_sw_2.Start()

' Return the Values
Dim m_materials_2 As New List(Of Material)
m_materials_2 = GetMaterialLINQ("a", m_doc)
m_sw_2.Stop()


' Display the Results
Dim m_results As String = _
    "Traditional (non LINQ):" & vbCr
m_results += m_materials_1.Count.ToString &
    " items returned" & vbCr &
      m_sw_1.ElapsedMilliseconds.ToString() &
   " milliseconds elapsed" & vbCr & vbCr
m_results += "LINQ Sample:" & vbCr &
   m_materials_2.Count.ToString &
   " items returned" & vbCr &
   m_sw_2.ElapsedMilliseconds.ToString() &
   " milliseconds elapsed"

' Construct and Display a Revit TaskDialog
Dim m_td As New TaskDialog("Elapsed Time:")
m_td.MainInstruction = "Material Filtering Results"
m_td.MainContent = m_results
m_td.Show()
```

6. Run the command and take a look at the results dialog. In this case, the LINQ method is much faster than the method that uses traditional iteration.



Material Filtering Results

Traditional (non LINQ):
61 items returned
833 milliseconds elapsed

LINQ Sample:
61 items returned
18 milliseconds elapsed

How it works...

The `StopWatch` class is very easy to use and provides valuable information helping you to test and quantify performance comparisons as you develop. I use it frequently to understand where my applications are running slow so that I know where to focus to improve the overall performance of my applications. Just remember to comment out any message box code that you use to report `StopWatch` results to the screen prior to releasing your applications to your users.

The `GetMaterial` function collects all the material elements in the model using the `OfCategory` method just the same as the `GetMaterialLINQ` function. The only difference between these two functions is how the name criteria are searched within the resulting collection gathered from the `Materials` category. The traditional method requires that each element be tested individually by iteration gathering matching elements as it goes. The LINQ function does not have to iterate through the list of elements. `GetMaterialLINQ` queries the list using a single query command and as a result is quite a bit quicker at gathering materials by name.

Accessing the ProjectInfo data (Must know)

The `ProjectInfo` data comprises several visible system parameters by default. Some of these parameters are formatted as text and others are special data parameters such as the parameters that store the energy settings. In this recipe, we will learn how to access the project information object and read some if its data. The `ProjectInfo` object is the same object that you access from the user interface under the `Manage` tab by clicking on the `Project Information` command button.

Getting ready

Create a new directory in your solution file named `Ch4 Data` and place each of the samples we create in this recipe here. Since there is only one project information object per model, finding and reading it is fairly straight forward.

We will also need to update our `Revit2013Samples_VB.addin` file with our new command. Add the following to this file inside the `RevitAddIns` tag being careful not to nest inside any other command or application:

```
<AddIn Type="Command">
  <Text>
    Sample - Ch4 Project Info
  </Text>
  <Assembly>
    Revit2013Samples_VB.dll
  </Assembly>
```

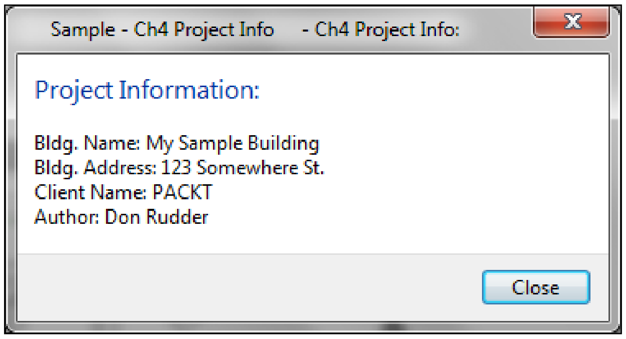
```
    <FullClassName>
      Revit2013Samples_VB.CommandProjectInfo
    </FullClassName>
    <ClientId>14a6cba3-a563-4dc7-a126-47fe5c9a11a1</ClientId>
    <VendorId>XXXX</VendorId>
    <VendorDescription>
      Packt Publishing Samples for
      'Autodesk Revit 2013 Customization with .NET'
    </VendorDescription>
  </AddIn>
```

How to do it...

1. Add a new class named `CommandProjectInfo` in the `Ch4 Data` directory and paste the `CommandBoilerPlate` code into this new class being careful to keep the name for this new class `CommandProjectInfo`.
2. Add the `ProjectInfo` filtering and reporting code just beneath the line that reads `' Begin Code Here`:

```
Dim m_doc As Document
m_doc = _
  commandData.Application.ActiveUIDocument.Document

' Collect the ProjectInfo Object
Dim m_col As New FilteredElementCollector(m_doc)
m_col.OfClass(GetType(ProjectInfo))

' Get the First
Dim m_pi As ProjectInfo
m_pi = m_col.First

' Read a Few Properties
Dim m_msg As String = ""
m_msg += "Bldg. Name: " & m_pi.BuildingName & vbCr
m_msg += "Bldg. Address: " & m_pi.Address & vbCr
m_msg += "Client Name: " & m_pi.ClientName & vbCr
m_msg += "Author: " & m_pi.Author

' Dispay the Results in a Revit Taskdialog
Using m_td As New TaskDialog("Ch4 Project Info:")
  m_td.MainInstruction = "Project Information:"
  m_td.MainContent = m_msg
  m_td.Show()
End Using
```

3. Open an `.rvt` file and run the sample in the debugger. You will get a dialog like the one shown in the following screenshot displaying the `ProjectInfo` data for the active model.



Project Information:

Bldg. Name: My Sample Building
Bldg. Address: 123 Somewhere St.
Client Name: PACKT
Author: Don Rudder

How it works...

Did you notice that we did not initiate a transaction for this sample? This is because everything we did was read only and no changes were being made to the model. This was about the simplest data access sample that you will see using the Revit API. Each of the data values that we reported were all text and accessible as properties rather than `Parameter` class objects. We first got a reference to the current document so that we could build a `FilteredElementCollector` for gathering up the `ProjectInfo` object. Once we had the element we wanted to report data from we concatenated a few data strings of interest and displayed the results in a `TaskDialog` form.

Extracting data (Should know)

You will undoubtedly find yourself in a situation where you need to get data out of the model. The format in which you export the data is purely up to you, but the means for reading the data basically remain the same. We will build a command in this recipe that exports all room data to a **Comma Separated Values** (**CSV**) file.

There are a couple of things to consider when exporting data from parameters other than text. The `ElementID` parameters can export either the seven digit numerical representation of `ElementID` or with a little trickery can export the name of the element that it points to.

Numerical parameters have a similar capability where they can be exported as either the decimal representation for their values or the human readable string representation as seen in the user interface. An example of this would be exporting an area parameter as either `4.0` or as `4.0 SF`. Our sample will export both values for these kinds of situations.

Getting ready

This sample will get a little trickier, so we will be creating a couple of helper classes to provide all of our CSV and parameter functionality. We will save all of the files that we create in this recipe in our `Ch4 Data` directory within our solution.

We will also edit our `Revit2013Samples_VB.addin` file again so that our new command will load into Revit. Add the following code to this file inside the `RevitAddIns` tag being careful not to nest inside any other command or application:

```
<AddIn Type="Command">
  <Text>
    Sample - Ch4 Export Data
  </Text>
  <Assembly>
    Revit2013Samples_VB.dll
  </Assembly>
  <FullClassName>
    Revit2013Samples_VB.CommandExportData
  </FullClassName>
  <ClientId>14a6cba3-a563-4dc7-a026-47fe5c9a11a1</ClientId>
  <VendorId>XXXX</VendorId>
  <VendorDescription>
    Packt Publishing Samples for
    'Autodesk Revit 2013 Customization with .NET'
  </VendorDescription>
</AddIn>
```

How to do it...

1. Add a new class named `clsExport`. This class is just a simple `StreamWriter` class that allows various separator formats such as comma, semicolon, and tab:

```
Imports System.IO

Public Class clsExport

  Private _separator As String

  Public Property FilePath As String

  ''' <summary>
  ''' Constructor
  ''' </summary>
  Public Sub New(p_path As String,
                 p_model As String,
                 p_headers As List(Of String),
                 Optional p_sep As String = ",")

    ' Widen Scope
    FilePath = p_path
    _separator = p_sep

    ' Create the File
    Using sw As New StreamWriter(FilePath, False)

      ' Write Date and Title
      sw.WriteLine("Rooms from Model: " & p_model)
      sw.WriteLine("Exported: " & Now().ToLongTimeString)
      sw.WriteLine("")

    End Using

    ' Write the Headers
    WriteLine(p_headers)

  End Sub

  ''' <summary>
  ''' Append a Line to the Log
  ''' </summary>
  Public Sub WriteLine(p_params As List(Of String))

    ' Append the Line
    Using sw As New StreamWriter(FilePath, True)

      ' Comma Separated Values
      Dim m_row As String = ""

      For Each x As String In p_params

        ' Append the Separator
        m_row += x & _separator

      Next

      ' Write Line
      sw.WriteLine(m_row)

    End Using

  End Sub

End Class
```

2. Add a new class named `clsParameter` and add the functions that can read the parameter data in string and data formats as shown here. We will leave the `SetValue` function empty for now:

```
Imports Autodesk.Revit.DB

''' <summary>
''' Helper class used to work with parameters
''' </summary>
Public Class clsParameter

  Private _parameter As Parameter

  ''' <summary>
  ''' Constructor
  ''' </summary>
  Public Sub New(ByVal p As Parameter)

    ' Widen Scope
    _parameter = p

  End Sub

  ''' <summary>
  ''' The parameter reference
  ''' </summary>
  Public ReadOnly Property ParameterObject() As Parameter
    Get
      Return _parameter
    End Get
  End Property

  ''' <summary>
  ''' Returns value of the parameter
  ''' </summary>
  Public Property Value() As String
    Get
      Try
        Dim v As String = GetValue(False)
        If Not String.IsNullOrEmpty(v) Then
          Return v
        Else
          Return ""
        End If
      Catch
        Return Nothing
      End Try
  End Get
  Set(ByVal v As String)
    Try
      SetValue(v, False)
    Catch
    End Try
  End Set
End Property

''' <summary>
''' Returns value of the parameter
''' as a string
''' </summary>
Public Property ValueString As String
  Get
    Try
      Dim v As String = GetValue(True)
      If Not String.IsNullOrEmpty(v) Then
        Return v
      Else
        Return ""
      End If
    Catch
      Return Nothing
    End Try
  End Get
  Set(ByVal v As String)
    Try
      SetValue(v, True)
    Catch
    End Try
  End Set
End Property

''' <summary>
''' Set a value to a parameter
''' </summary>
Private Sub SetValue(ByVal value As Object,
                     asString As Boolean)
    ' Cannot edit readonly
  If _parameter.IsReadOnly Then Exit Sub

  Try
    ' Storage Type
```

```
        Select Case _parameter.StorageType
          Case StorageType.Double
            If asString = True Then
              _parameter.SetValueString _
                (TryCast(value, String))
            Else
              _parameter.Set(value)
            End If

          Case StorageType.ElementId
            Dim m_eid As ElementId
            m_eid = DirectCast((value), ElementId)
            _parameter.Set(m_eid)

          Case StorageType.Integer
            _parameter.SetValueString _
              (TryCast(value, String))

          Case StorageType.None
            _parameter.SetValueString _
              (TryCast(value, String))

          Case StorageType.String
            _parameter.Set(TryCast(value, String))
            Exit Select

        End Select
      Catch
      End Try

    End Sub

    ''' <summary>
    ''' Get the value of a parameter
    ''' </summary>
    Private Function GetValue(asString As Boolean) As String

      ' Return the Value
      Select Case _parameter.StorageType
        Case StorageType.Double
          If asString = True Then
            Return _parameter.AsValueString
          Else
            Return _parameter.AsDouble.ToString
          End If
```

```
            Case StorageType.ElementId
              If asString = True Then
                ' Get the Element's Name
                Dim m_eid As New ElementId _
                  (_parameter.AsElementId.IntegerValue)
                Dim m_obj As Element
                m_obj =
                 _parameter.Element.Document.GetElement(m_eid)
                Return m_obj.Name
              Else
                Return _parameter.AsElementId.ToString
              End If

            Case StorageType.Integer
              Return _parameter.AsInteger.ToString

            Case StorageType.None
              Return _parameter.AsValueString

            Case StorageType.String
              Return _parameter.AsString

            Case Else
              Return ""

          End Select

        End Function

      End Class
```

3. Add yet another command class copying the `CommandBoilerPlate` code naming this new class `CommandExportData`.
4. Add the following code beneath the line `' Begin Code Here` to collect all rooms and cast them into a list of `Architecture.Room` elements:

```
            Dim m_doc As Document
            m_doc =
              commandData.Application.ActiveUIDocument.Document

            ' Get the Collection of Rooms
            Dim m_col As New FilteredElementCollector(m_doc)
            m_col.OfCategory(BuiltInCategory.OST_Rooms)
```

5. Next construct the `clsExport` object that we will use to export the data to the file. We will use a comma as a column separator in this case:

```
' Get First Room, as a Room
Dim rm As Architecture.Room
rm = TryCast(m_col.FirstElement, Architecture.Room)

' List of Parameter Names
Dim params As New List(Of String)
For Each p As Parameter In rm.Parameters
  If p.StorageType = StorageType.ElementId Or
    p.StorageType = StorageType.Double Then
    params.Add("text_" & p.Definition.Name)
    params.Add(p.Definition.Name)
  Else
    params.Add(p.Definition.Name)
  End If
Next

' The Data File
Dim m_fname As String
m_fname = GetFolderPath _
  (Environment.SpecialFolder.MyDocuments)
m_fname += "\PACKTdataExport.csv"
Dim m_csv As New clsExport(m_fname,
                           m_doc.PathName,
                           params,
                           ",")
```

6. Now we need to iterate over each room and process each parameter:

```
' Iterate Each Room
For Each elem In m_col.ToElements

  ' Cast as a room
  Dim r As Architecture.Room = _
    TryCast(elem, Architecture.Room)

  If Not r Is Nothing Then

    ' Sore the Values
    Dim m_values As New List(Of String)

    ' Maintain the Order of Values and Header
    For Each x As String In params
```

```
          ' Parameter
          Dim m_p As Parameter = Nothing

          ' Parameter Helper
          Dim m_parameter As clsParameter = Nothing

          ' Parameter Name
          If x.StartsWith("text_") Then
            m_p = elem.Parameter(Mid(x, 6))
            m_parameter = New clsParameter(m_p)
            m_values.Add(m_parameter.ValueString)
          Else
            m_p = elem.Parameter(x)
            m_parameter = New clsParameter(m_p)
            m_values.Add(m_parameter.Value)
          End If

        Next

        ' Write the Line
        m_csv.WriteLine(m_values)

      End If

    Next
```

7. The last part we'll add is a means to report to the user what we just did and where the file was saved:

```
    ' Tell the user what happened
    Using td As New TaskDialog("Room Data Export")
      td.MainInstruction = m_col.ToElements.Count.ToString &
        " Rooms Processed..."
      td.MainContent = m_csv.FilePath
      td.Show()
    End Using
```

8. Open the resulting CSV file and take notice of the parameters prefixed with `text_` and their raw values and notice how they represent basically the same data but in different formats.

How it works...

The first thing the command does is collect all of the rooms as the `Element` objects and cast them into a list of the `Room` classes. We then iterate over each `Room` and read each parameter and store the values into our `clsExport` class that ultimately write them out to the `.csv` file. `Double` and `ElementID` formatted parameters are exported in both their numerical representations and their string representations with a header prefix of `string_` for the string representation so you can see the difference between their absolute values and their string representations.

The `clsExport` class is quite simple in that it only initiates a `StreamWriter` class and makes calls to append new lines to this file. The header is written first containing the names of each parameter being exported. Each row sent to this class later using the `WriteLine` routine adds the data for a room until all rooms have been processed.

The `clsParameter` class is a little more complicated yet still fairly straight forward. The two public properties for `Value` and `ValueString` do what they sound like they should in that `Value` returns the absolute value for a parameter while the `ValuString` property returns the user interface string representation of the parameter's value.

Changing values (Must know)

Revit parameters store data in one of four basic formats each requiring a slightly different means of modifying its data. The code sample that we will build in this recipe simplifies the modification of parameter data regardless of the parameter's data format.

Getting ready

Update the `Revit2013Samples_VB.addin` file to enable the command in the Revit environment as shown in the following code:

```
<AddIn Type="Command">
  <Text>
    Sample - Ch4 Modify Data
  </Text>
  <Assembly>
    Revit2013Samples_VB.dll
  </Assembly>
  <FullClassName>
    Revit2013Samples_VB.CommandModifyData
  </FullClassName>
  <ClientId>14a6cba3-a563-4dc7-a026-47fe5c9a11a2</ClientId>
  <VendorId>XXXX</VendorId>
```

```
    <VendorDescription>
      Packt Publishing Samples for
      'Autodesk Revit 2013 Customization with .NET'
    </VendorDescription>
  </AddIn>
```

How to do it...

1. Open the `clsParameter` class and add the following code to the empty `SetValue` function:

```
''' <summary>
''' Set a value to a parameter
''' </summary>
Private Sub SetValue(ByVal value As Object,
                     asString As Boolean)

  ' Cannot edit readonly
  If _parameter.IsReadOnly Then Exit Sub

  Try
    ' Storage Type
    Select Case _parameter.StorageType
      Case StorageType.Double
        If asString = True Then
          _parameter.SetValueString _
            (TryCast(value, String))
        Else
          _parameter.Set(value)
        End If

      Case StorageType.ElementId
        Dim m_eid As ElementId
        m_eid = DirectCast((value), ElementId)
        _parameter.Set(m_eid)

      Case StorageType.Integer
        _parameter.SetValueString _
          (TryCast(value, String))

      Case StorageType.None
        _parameter.SetValueString _
          (TryCast(value, String))

      Case StorageType.String
        _parameter.Set(TryCast(value, String))
        Exit Select
    End Select
  Catch

  End Try

End Sub
```

2. Create a new class named `CommandModifyData` and paste the boilerplate code from our `CommandBoilerPlate` class being careful to keep our new class named `CommandModifyData`.
3. Add the following collector code to get all rooms and iterate over each of the comments parameters changing its value to the room's `UniqueId`:

```
Dim m_doc As Document
m_doc = _
  commandData.Application.ActiveUIDocument.Document

Dim m_col As New FilteredElementCollector(m_doc)
m_col.OfCategory(BuiltInCategory.OST_Rooms)

' Transaction
Using t As New Transaction(m_doc,
                           "Modify Parameter")

  ' Start the Transaction
  If t.Start Then

    ' Iterate Each Element
    For Each elem As Element In m_col.ToElements

      ' Get the Comments Parameter
      Dim m_p As Parameter
      m_p = elem.Parameter("Comments")

      ' Helper
      Dim m_param As New clsParameter(m_p)

      ' Change the Value
      m_param.Value = elem.UniqueId.ToString

    Next

  End If

  ' Commit
  t.Commit()

End Using

' Inform the User
```

```
            Using td As New TaskDialog("Modify Parameters")
              td.MainInstruction = "Changed Values"
              td.MainContent = m_col.ToElements.Count.ToString &
                " room comments edited..."
              td.Show()
            End Using
```

4. Select a room element after you run this sample and inspect the contents of the comments parameter. The value that you see is the `UniqueId` for the room element.

How it works...

We had to initiate a new transaction in this sample because we were making changes to the model. It is always recommended to trap your transactions in a `Using` statement and starting it with an `If` statement to avoid the rare yet possible occasions where a transaction may not start prior to attempting to make changes to the model. The `SetValue` function in our `clsParameter` function accepts an object that in this case was just a simple text value and passes it into the comments parameter of each room.

Adding and removing parameters (Become an expert)

You will occasionally run into situations where you need to add parameters to a model. There's not yet a way to add general parameters to a project model, but you can definitely add parameters from a shared parameter file and bind them to any category you like. We will build a quick command in this recipe demonstrating the concept of adding parameters from a shared parameter file and binding them to a category by name.

Getting ready

We need to edit our `Revit2013Samples_VB.addin` file again so that our new command will load into Revit. Add the following inside the `RevitAddIns` tag being careful not to nest inside any other command or application:

```
<AddIn Type="Command">
  <Text>
    Sample - Ch4 Add Parameters
  </Text>
  <Assembly>
    Revit2013Samples_VB.dll
  </Assembly>
  <FullClassName>
    Revit2013Samples_VB.CommandAddParameter
  </FullClassName>
  <ClientId>14a6cba3-a563-4dc7-a026-47fe5c9a11a3</ClientId>
  <VendorId>XXXX</VendorId>
  <VendorDescription>
    Packt Publishing Samples for
    'Autodesk Revit 2013 Customization with .NET'
  </VendorDescription>
</AddIn>
```

How to do it...

1. Add another command class named `CommandAddParameter` and copy the `CommandBoilerPlate` code in being careful to maintain the new class name as `CommandAddParameter`.
2. Next we create a reference to the shared parameter file as a `DefinitionFile` and bind each parameter in the file to the `Rooms` category. Since we are making changes to elements in the model we must trap our code within a transaction block:

```
Dim m_app As UIApplication
m_app = commandData.Application
Dim m_doc As Document
m_doc = m_app.ActiveUIDocument.Document

' Active Shared Parameter
Dim m_defFile As DefinitionFile
m_defFile = _
  m_app.Application.OpenSharedParameterFile()

' Active Shared Parameter File?
If Not m_defFile Is Nothing Then

  ' Transaction
  Using t As New Transaction(m_doc,
                             "Add Parameters")
    If t.Start Then

      ' Create the Rooms Category Set
      Dim m_cSet As CategorySet
      m_cSet = m_app.Application.Create.NewCategorySet
      Dim m_rm_c As Category
      m_rm_c = m_doc.Settings.Categories.Item("Rooms")
      m_cSet.Insert(m_rm_c)

      ' Add all shared parameters to
      For Each g As DefinitionGroup In m_defFile.Groups
        For Each d As Definition In g.Definitions
```

```
                    ' Bind to Category
                    Dim m_ibind As InstanceBinding
                    m_ibind = _
             m_app.Application.Create.NewInstanceBinding(m_cSet)
                    m_doc.ParameterBindings.Insert(d, m_ibind)

                  Next

                Next

                ' Commit
                t.Commit()

              End If
            End Using

          Else

            ' No Active Shared Parameter File
            MsgBox("No Active Shared Parameter File!",
                   MsgBoxStyle.Exclamation,
                   ":(")
            Return Result.Cancelled

          End If
```

3. Run the command and then select a room element in the model and each parameter in you're the shared parameter file has been bound to the rooms category.

How it works...

As with most of the commands we've done previously, we first get a reference to the active document, but in this one we also get a reference to the application object. The next thing we do is check to see if there is an active shared parameter file loaded and if there is we start iterating over each group in the file and load each parameter we find into the category set that ultimately gets bound to the category.

A complex design model can often comprise hundreds if not thousands of views.
The good news is that the views required from one project to another are fairly consistent. Sheet and view creation is definitely an area of automation that can gain you a serious ROI if planned properly.

In the following few recipes, we will build a few sample commands demonstrating how to use the new `ViewPlan` methods, and create schedules, as well as how to create sheets. After we have the concepts for creating views down, we will learn how to place views on sheets.

Creating plan views (Must know)

The method for generating new plan views has changed quite a bit in the 2013 API. The new way is far cleaner and from what I can tell is actually faster as well. We are about to create a simple command sample that will create a floor plan, ceiling plan, and an area plan for each view type available in the active model for each level.

Getting ready

We just started a new recipe, so let's keep the magic going by creating a new directory in our solution named `Ch5 Document` and place each of this recipe's code samples in it.

As always when adding a new command class to the project we need to update the `Revit2013Samples_VB.addin` file so it can be accessed from the Revit session. Add the following inside the `RevitAddIns` tag being careful not to nest it inside any other `Command`, `Application`, or `DBApplication`:

```
<AddIn Type="Command">
  <Text>
    Sample - Ch5 Views
  </Text>
  <Assembly>
    Revit2013Samples_VB.dll
  </Assembly>
  <FullClassName>
    Revit2013Samples_VB.CommandViews
  </FullClassName>
  <ClientId>14a6cba3-a563-4dc7-a026-47fe5c9a11a4</ClientId>
  <VendorId>XXXX</VendorId>
  <VendorDescription>
    Packt Publishing Samples for
    'Autodesk Revit 2013 Customization with .NET'
  </VendorDescription>
</AddIn>
```

How to do it...

1. Add a new class named `CommandViews` and copy in the code from our `CommandBoilerPlate` sample from the *Getting started with the Autodesk Revit 2013 API (Must know)* recipe. Be careful to keep the name of this new class as `CommandViews`.

2. We start by adding the code that gains us access to the active `Application` and `Document` object. Add the following just beneath the line that reads `' Begin Code Here`:

```
Dim m_app As UIApplication
m_app = commandData.Application
Dim m_doc As Document
m_doc = m_app.ActiveUIDocument.Document
```

3. Next we must collect a list of each of the `ViewFamilyType` elements in our model by using `FilteredElementCollector`. The `ViewFamilyType` argument is a new required argument for Revit 2013 when creating plan views:

```
' Get List of View Types
Dim m_colVT As New FilteredElementCollector(m_doc)
m_colVT.OfClass(GetType(ViewFamilyType))
Dim m_vt As New List(Of ViewFamilyType)
For Each x As ViewFamilyType In m_colVT.ToElements
  m_vt.Add(x)
Next
```

4. The next thing we need to do is gather each of the levels so that we can create a view for each level in the model:

```
' Get List of Levels
Dim m_colL As New FilteredElementCollector(m_doc)
m_colL.OfCategory(BuiltInCategory.OST_Levels)
Dim m_levels As New List(Of Level)
For Each x In m_colL.ToElements
  If TypeOf x Is Level Then
    m_levels.Add(x)
  End If
Next
```

5. We are making changes to the model so we will need to start a new transaction:

```
' New Transaction
Using t As New Transaction(m_doc, "Create Views")

  ' Start the Transaction
  If t.Start Then

    ' Success and Error List
    Dim m_s As New List(Of String)
    Dim m_e As New List(Of String)
```

6. Now that we have all of the `Level` and `ViewFamilyType` elements, we can iterate over the levels and then create the views that we are interested in:

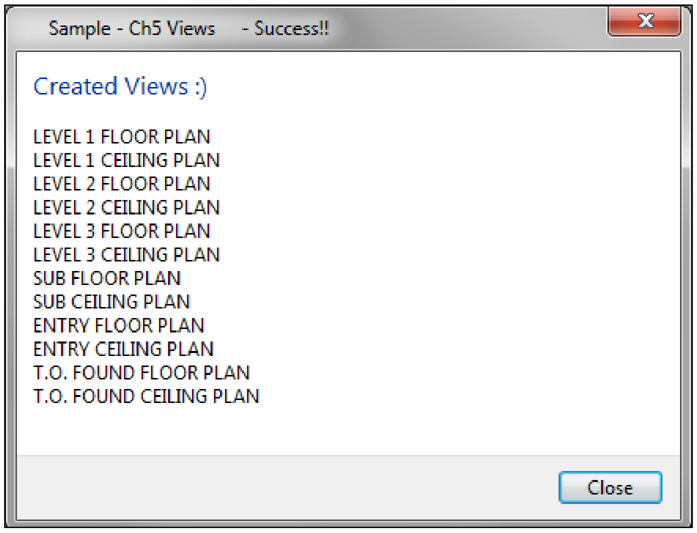
```
' Create a Plan of each type per level
For Each l As Level In m_levels

  ' Views of Plan or Ceiling Only
  For Each vt As ViewFamilyType In m_vt

    ' FloorPlan, CeilingPlan, or AreaPlan Only
    If vt.ViewFamily = ViewFamily.CeilingPlan Or
       vt.ViewFamily = ViewFamily.FloorPlan Or
       vt.ViewFamily = ViewFamily.AreaPlan Then

      Try

        ' Create the Plan View
        Dim m_fp As ViewPlan
        m_fp = ViewPlan.Create(m_doc, vt.Id, l.Id)

        ' Rename the View
        m_fp.Name = l.Name.ToUpper & " " &
                    vt.Name.ToUpper

        m_s.Add(l.Name.ToUpper & " " &
                vt.Name.ToUpper)

      Catch ex As Exception

        ' Record Errors
        m_e.Add(ex.Message & ": " &
                l.Name.ToUpper & " " &
                vt.Name.ToUpper)

      End Try

    End If

  Next

Next
```

7. Display the results to the user for what we just did and commit the transaction:

```
' Report Views Created
If m_s.Count > 0 Then
  Using m_td As New TaskDialog("Success!!")
    m_td.MainInstruction = "Created Views :)"
    For Each x In m_s
      m_td.MainContent += x & vbCr
    Next
```

```
          m_td.Show()
        End Using
      End If

      ' Report Errors if Any
      If m_e.Count > 0 Then
        Using m_td As New TaskDialog("Some Errors")
          m_td.MainInstruction = "Issues with Views:"
          For Each x In m_e
            m_td.MainContent += x & vbCr
          Next
          m_td.Show()
        End Using
      End If

      ' Commit
      t.Commit()

    End If

  End Using
```

8. Open the `rac_advanced_sample_project.rvt` from the sample code bundle and run the command. You will get a dialog showing you the views that it generated similar to this.



Sample - Ch5 Views - Success!!

Created Views :)

```
LEVEL 1 FLOOR PLAN
LEVEL 1 CEILING PLAN
LEVEL 2 FLOOR PLAN
LEVEL 2 CEILING PLAN
LEVEL 3 FLOOR PLAN
LEVEL 3 CEILING PLAN
SUB FLOOR PLAN
SUB CEILING PLAN
ENTRY FLOOR PLAN
ENTRY CEILING PLAN
T.O. FOUND FLOOR PLAN
T.O. FOUND CEILING PLAN
```

How it works...

The first two groups of elements that we collect in the preceding sample are critical for adding new plan views to the model. A `Level` argument is required since it tells Revit where in the model you want the view to appear. The `ViewFamilyType` is a new argument required for plan views and is a nice addition because it allows you control over which view type to use for the view.

The sample then iterates over each `Level` in the model and creates a plan view for each `FloorPlan`, `CeilingPlan`, and `AreaPlan` type that is loaded in the model. The new `Create` method for generating plan views does not allow for a naming argument, so you need to maintain a reference to the view while creating it so that you can rename it later.

Creating a schedule (Become an expert)

Creating a schedule with the API is a new feature in 2013. We will build a schedule for rooms in this recipe showing each of the available room parameters.

Getting ready

We need to update the `Revit2013Samples_VB.addin` file so it can be accessed from the Revit session. Add the following inside the `RevitAddIns` tag being careful not to nest it inside any other `Command`, `Application`, or `DBApplication`:

```
<AddIn Type="Command">
  <Text>
    Sample - Ch5 Schedule
  </Text>
  <Assembly>
    Revit2013Samples_VB.dll
  </Assembly>
  <FullClassName>
    Revit2013Samples_VB.CommandSchedule
  </FullClassName>
  <ClientId>14a6cba3-a563-4dc7-a026-47fe5c9a11a5</ClientId>
  <VendorId>XXXX</VendorId>
  <VendorDescription>
    Packt Publishing Samples for
    'Autodesk Revit 2013 Customization with .NET'
  </VendorDescription>
</AddIn>
```

How to do it...

1. Create a new class named `CommandSchedule` and copy the code from our `CommandBoilerPlate` sample.
2. Just beneath the line `' Begin Code Here`, we begin by adding the code that gains us access to the `Application` and `Document` objects:

```
Dim m_app As UIApplication
m_app = commandData.Application
Dim m_doc As Document
m_doc = m_app.ActiveUIDocument.Document
```

3. Then since we are making changes to the model, we need to trap the rest inside a new transaction:

```
' New Transaction
Using t As New Transaction(m_doc, "Create Schedule")

  ' Start the Transaction
  If t.Start Then
```

4. With a transaction started, we first create the `Schedule` object using the room category:

```
    ' Add Room Schedule
    Dim m_vs As ViewSchedule
    Dim m_cat As Category = Nothing
    For Each c As Category In m_doc.Settings.Categories
      If c.Name.ToLower = "rooms" Then
        m_cat = c
        Exit For
      End If
    Next
    m_vs = ViewSchedule.CreateSchedule(m_doc, m_cat.Id)
    m_vs.Name = "PACKT Room Schedule"
```

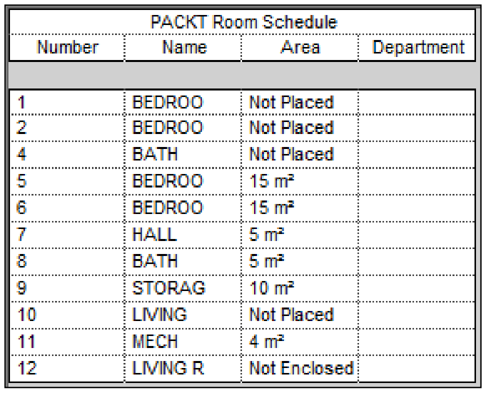
5. So far we've only built an empty schedule. We need to get the schedule's definition object next so we can use it to add fields and sorting. We use the definition object to add fields and sorting:

```
    ' Get the Definition
    Dim m_sd As ScheduleDefinition
    m_sd = m_vs.Definition

    ' Append Four Fields to Definition
    Dim m_areaField As ScheduleField
    m_areaField = m_sd.AddField( _
      New SchedulableField(ScheduleFieldType.Instance,
          New ElementId(BuiltInParameter.ROOM_NUMBER)))
    m_sd.AddField(New  _
          SchedulableField(ScheduleFieldType.Instance,
          New ElementId(BuiltInParameter.ROOM_NAME)))
    m_sd.AddField(New  _
          SchedulableField(ScheduleFieldType.ViewBased,
          New ElementId(BuiltInParameter.ROOM_AREA)))
    m_sd.AddField(New  _
          SchedulableField(ScheduleFieldType.Instance,
      New ElementId(BuiltInParameter.ROOM_DEPARTMENT)))

    ' Sort Ascending by Number
    Dim m_sort As New List(Of ScheduleSortGroupField)
    Dim m_numberSort As New ScheduleSortGroupField()
    m_numberSort.FieldId = m_areaField.FieldId
    m_numberSort.SortOrder = ScheduleSortOrder.Ascending
    m_sort.Add(m_numberSort)
    m_sd.SetSortGroupFields(m_sort)
```

6. If all goes well, we can report to the user that we've created a schedule and commit the transaction:

```
                ' Report to User
                Using m_td As New TaskDialog("Success!!')
                  m_td.MainInstruction = "Created Schedule :)"
                  m_td.MainContent = "Sorted by Room Number"
                  m_td.Show()
                End Using

                ' Commit
                t.Commit()

              End If

            End Using
```

7. Open the `rac_basic_sample_project.rvt` from the sample code bundle and run the command. Take a look at the room schedule named **PACKT Room Schedule** that was just created.



PACKT Room Schedule

Number	Name	Area	Department
1	BEDROO	Not Placed	
2	BEDROO	Not Placed	
4	BATH	Not Placed	
5	BEDROO	15 m²	
6	BEDROO	15 m²	
7	HALL	5 m²	
8	BATH	5 m²	
9	STORAG	10 m²	
10	LIVING	Not Placed	
11	MECH	4 m²	
12	LIVING R	Not Enclosed	

How it works...

Creating a schedule is the easy part but requires that you add fields afterwards using its `Definition` object. The `AddField` method of the `Definition` object allows you to add various kinds of schedulable data just like you can in the user interface.

Did you notice that the `ROOM_AREA` parameter's `ScheduleFieldType` was set to `ViewBased` while all the others were set to `Instance`? This is because a room's area is not a data object that is entered into the room element but is rather a condition of the room that is calculated spatially in the view.

Sorting is fairly straightforward and supports multiple fields of sorting. We only added one in this sample to keep it simple while demonstrating how to use this feature. The only field we added to the sort list was the `ROOM_NUMBER` parameter so that all of the records in the schedule would sort by their room number.

Creating sheets and placeholders (Must know)

Sheets are unfortunately still a necessary element for most BIM projects and can be a very tedious step in your production cycle. This recipe will cover sheet and placeholder sheet creation.

Getting ready

Hopefully by now you are an expert at this step. We need to first update the `Revit2013Samples_VB.addin` file so it can be accessed from the Revit session. Add the following inside the `RevitAddIns` tag being careful not to nest it inside any other `Command`, `Application`, or `DBApplication`:

```
<AddIn Type="Command">
  <Text>
    Sample - Ch5 Sheets
  </Text>
  <Assembly>
    Revit2013Samples_VB.dll
  </Assembly>
  <FullClassName>
    Revit2013Samples_VB.CommandSheets
  </FullClassName>
  <ClientId>14a6cba3-a563-4dc7-a026-47fe5c9a11a6</ClientId>
  <VendorId>XXXX</VendorId>
  <VendorDescription>
    Packt Publishing Samples for
    'Autodesk Revit 2013 Customization with .NET'
  </VendorDescription>
</AddIn>
```

How to do it...

1. Create a new class named `CommandSheets` and copy the code from our `CommandBoilerPlate` sample.
2. We start by adding the code that gains us access to the active document object. Add the following just beneath the `' Begin Code Here` line:

```
Dim m_app As UIApplication
m_app = commandData.Application
Dim m_doc As Document
m_doc = m_app.ActiveUIDocument.Document
```

3. Next we need to get the available title block family types. Inform the user if we do not find any because we will not be able to create real sheets without one:

```
' Get the Titleblocks
Dim m_col As New FilteredElementCollector(m_doc)
m_col.WhereElementIsElementType()
m_col.OfCategory(BuiltInCategory.OST_TitleBlocks)
Dim m_tb As New List(Of Element)
```

```
m_tb = m_col.ToElements

' Prompt if no Title Blocks
If m_tb.Count = 0 Then
  MsgBox("No Title Block Families Found in Model!" &
         vbCr &
         "Load a Title Block Family to Add Sheets",
         MsgBoxStyle.Information,
         "Cannot Create Sheets")
End If
```

4. I sometimes like to use error and success lists to report to the user what my code does. Add a couple of simple string lists for tracking our success and failures as we churn through the rest of the command:

```
' Success and Failures
Dim m_s As New List(Of String)
Dim m_e As New List(Of String)
```

5. We're making changes to the model so we need to start a transaction:

```
' Transaction
Using t As New Transaction(m_doc, "Create Sheets")

  If t.Start Then
```

6. We will first build a single placeholder sheet. After that we will build one real sheet for each of the loaded title block types in the active model:

```
    ' The Sheet Object
    Dim m_vs As ViewSheet

    Try

      ' Create a Placeholder Sheet
      m_vs = ViewSheet.CreatePlaceholder(m_doc)
      m_vs.Name = "PACKT Placeholder!"
      m_vs.SheetNumber = "X.X"

      ' Record Success
      m_s.Add("Created Placeholder: " &
              m_vs.SheetNumber)

    Catch ex As Exception

      ' Record Failure
      m_e.Add("Placeholder Error: " &
              ex.Message)
```

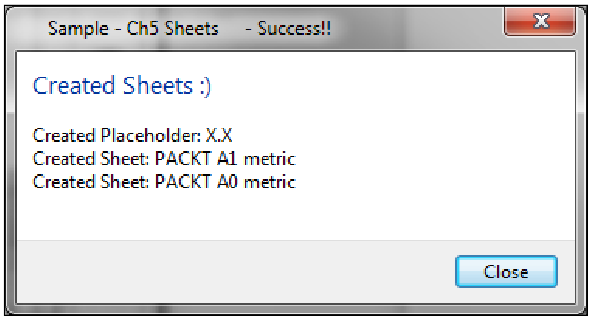
```
End Try

' Each Titleblock
For Each tb In m_tb

  Try

    ' Create Sheet
    m_vs = ViewSheet.Create(m_doc, tb.Id)
    m_vs.Name = "PACKT Sheet from Titleblock! " &
                 tb.Name
    m_vs.SheetNumber = "PACKT " &
                        tb.Name

    ' Record Success
    m_s.Add("Created Sheet: " &
            m_vs.SheetNumber)

  Catch ex As Exception

    ' Record Failure
    m_e.Add("Sheet Error: " &
            ex.Message)

  End Try

Next
```

7. The last part commits the transaction and reports the results to the user:

```
' Commit
t.Commit()

' Report Sheets Created
If m_s.Count > 0 Then
  Using m_td As New TaskDialog("Success!!")
    m_td.MainInstruction = "Created Sheets :)"
    For Each x In m_s
      m_td.MainContent += x & vbCr
    Next
    m_td.Show()
  End Using
End If

' Report Errors if Any
If m_e.Count > 0 Then
  Using m_td As New TaskDialog("Some Errors")
    m_td.MainInstruction = "Issues with Sheets:"
    For Each x In m_e
```

```
                    m_td.MainContent += x & vbCr
                  Next
                  m_td.Show()
                End Using
              End If

            End If

          End Using
```

8. Open the `rac_advanced_sample_project.rvt` from the sample code bundle and run the command:



Created Sheets :)

Created Placeholder: X.X
Created Sheet: PACKT A1 metric
Created Sheet: PACKT A0 metric

How it works...

The methods for creating both real and placeholder sheets are accessible from the `ViewSheet` class. Both methods result in a `ViewSheet` object that can then be used to change the name, number, or any parameter of the object that you choose.

We created one placeholder sheet and assigned it a number of `X.X`. The process for creating real sheets is a little more interesting. You can create a sheet with any title block in the model you want but you must have a valid title block element to create a real sheet. We iterated over each of the tile block families available in the model and created a sheet for each one.

There's more...

The list of sheets that we created in this sample was quite basic but there's really nothing stopping us from making hundreds or even thousands of sheets using this method. Imagine the possibilities if you were to build a tool using these concepts that reads data from an external source such as a database or spreadsheet. The time savings possible are virtually limitless.

Placing views on sheets (Become an expert)

Now that we know how to create views as well as sheets we can start to build some meaningful documentation. We will build a sample command in this recipe that places the first available floor plan view onto the active sheet. You will need to have a sheet view active when running this command.

Getting ready

We need to first update the `Revit2013Samples_VB.addin` file so it can be accessed from the Revit session. Add the following inside the `RevitAddIns` tag being careful not to nest it inside any other `Command`, `Application`, or `DBApplication`:

```
<AddIn Type="Command">
  <Text>
    Sample - Ch5 Views on Sheets
  </Text>
  <Assembly>
    Revit2013Samples_VB.dll
  </Assembly>
  <FullClassName>
    Revit2013Samples_VB.CommandViewsOnSheets
  </FullClassName>
  <ClientId>14a6cba3-a563-4dc7-a026-47fe5c9a11a7</ClientId>
  <VendorId>XXXX</VendorId>
  <VendorDescription>
    Packt Publishing Samples for
    'Autodesk Revit 2013 Customization with .NET'
  </VendorDescription>
</AddIn>
```

How to do it...

1. Add a new class named `CommandViewsOnSheets` and copy in the code from our `CommandBoilerPlate` sample from the *Getting started with the Autodesk Revit 2013 API (Must know)* recipe. Be careful to keep the name of this new class as `CommandViewsOnSheets`.
2. We start again by adding the code that gains us access to the active document object. Add the following code just beneath the line that reads `' Begin Code Here`:

   ```
   Dim m_app As UIApplication
   m_app = commandData.Application
   Dim m_doc As Document
   m_doc = m_app.ActiveUIDocument.Document
   ```

3. First we need to collect the list of views in the active model:

```
' Get All Views
Dim m_colV As New FilteredElementCollector(m_doc)
m_colV.OfCategory(BuiltInCategory.OST_Views)
Dim m_views As New List(Of View)
For Each x As View In m_colV.ToElements
  m_views.Add(x)
Next
```

4. We're making changes to the model again, so start a new transaction:

```
' New Transaction
Using t As New Transaction(m_doc, "Create Views")

  If t.Start Then
```

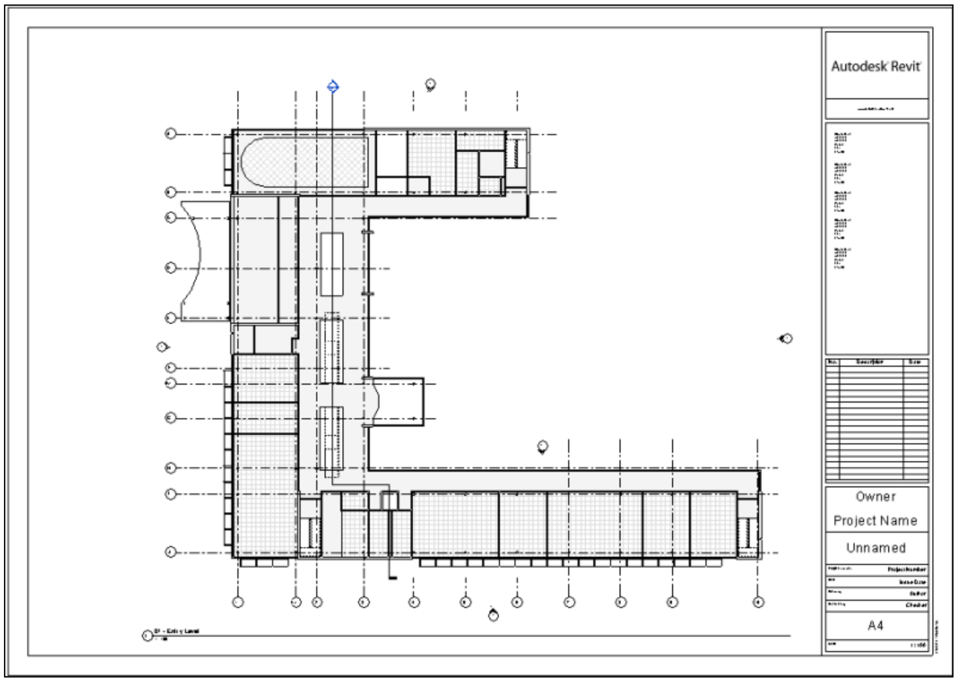
5. Now for the main sequence of code, first check if the active view is a sheet. If we are in an active sheet we find the first plan view that is not yet on a sheet and place it into the sheet:

```
    ' User Reporting
    Dim m_msg As String
    m_msg = "Available Floorplan Not Found to Add"

    ' Is the Current View a Sheet?
    Dim m_v As View = m_doc.ActiveView
    If TypeOf m_v Is ViewSheet Then

      ' Find an Availabe Plan
      For Each v As View In m_views

        ' Add the First Available Plan
        If TypeOf v Is ViewPlan Then

          Try

            ' Can it be added?
            If Viewport.CanAddViewToSheet(m_doc,
                        m_v.Id,
                        v.Id) Then

              ' Add it
              Viewport.Create(m_doc,
                       m_v.Id,
                       v.Id,
                       New XYZ(1.25, 1.25, 0))
              m_msg = "Added Floorplan " & v.Name
              Exit For

            End If

          Catch ex As Exception

            m_msg = "Floorplan " & v.Name & " failed!"

          End Try

        End If

      Next

    Else

      ' Failure Message
      message = "You must be in an active sheet " &
                "view for this command"
      Return Result.Failed

    End If
```

6. The last part is committing the transaction and reporting to the user what happened:

```
    ' Commit
    t.Commit()

    ' Report Views Created
    Using m_td As New TaskDialog("Views on Sheets")
      m_td.MainInstruction = "Results"
      m_td.MainContent = m_msg
      m_td.Show()
    End Using

  End If

End Using
```

7. Open the `rac_advanced_sample_project.rvt` from the sample code bundle and run the command. Hopefully your sheet now looks like the one shown in the following figure:



How it works...

First we check if the active view is a sheet. We then iterate over each view in the model checking to see if it is a `ViewPlan` and if it is we check to see if it has not yet been placed onto a sheet by calling `CanAddViewToSheet` from the `Viewport` object. If the view is available to be added onto our new sheet, we add it. The `XYZ` location that we used for placing the view onto the sheet was from the centre of the `ViewPlan` on the sheet's paper coordinates.

The analysis framework is an interesting feature of the API that can be used to alter the appearance of elements in the model. People have done all kinds of interesting things with this framework. One of the more radical demonstrations that I've seen was by Jeremy Tammik on his popular *Building Coder* blog where he painted the image from a webcam onto a curved wall and had it update every so often if the model was idle `http://thebuildingcoder.typepad.com/blog/2010/06/display-webcam-image-on-building-element-face.html`.

Wall color by length (Become an expert)

We will create a command utilizing the analysis framework to colorize walls based on their length in this recipe. This command will create a new custom 3D view, set it active, and then turn everything off in the model leaving only the analysis elements visible. A legend showing the wall length ranges will be displayed in the lower-left corner of the view with the longer walls shown in blue and the shorter in red.

Getting ready

Create a new directory in your solution file named `Ch6 Analysis` and place this code sample in there. We also need to update the `Revit2013Samples_VB.addin` file so it can be accessed from the Revit session. Add the following inside the `RevitAddIns` tag being careful to not nest it inside any other `Command`, `Application`, or `DBApplication`:

```
<AddIn Type="Command">
  <Text>
    Sample - Ch6 Colour by Wall Length
  </Text>
  <Assembly>
    Revit2013Samples_VB.dll
  </Assembly>
  <FullClassName>
    Revit2013Samples_VB.CommandAnalysis
  </FullClassName>
  <ClientId>14a6cba3-a563-4dc7-a026-47fe5c9a11b1</ClientId>
  <VendorId>XXXX</VendorId>
  <VendorDescription>
    Packt Publishing Samples for
    'Autodesk Revit 2013 Customization with .NET'
  </VendorDescription>
</AddIn>
```

How to do it...

1. Create a new class named `CommandAnalysis` and copy the boilerplate command code into this new class being careful to maintain the name of this new class as `CommandAnalysis`.
2. This command needs an additional namespace reference that was not included in our boilerplate code sample. Add this to the very top of our class file:

   ```
   Imports Autodesk.Revit.DB.Analysis
   ```

3. We will need one class-wide variable named `_schemaId` for our analytical framework setup. Place the following line of code just beneath the line that reads `Implements IExternalCommand`:

```
Dim _schemaId As Integer = -1
```

4. We're going to need a few helper functions and subroutines to get our analytical visualizations going. First place the following function at the very bottom of the class just inside the `End Class` line. This function will return `solid` from the wall for us:

```
''' <summary>
''' Get the Solid From the Geometry
''' </summary>
Public Function GetGeometry(ByVal g As GeometryElement) _
                As Solid

  ' Get the Wall Geometry
  Dim m_geo1 = g.GetEnumerator
  Do While m_geo1.MoveNext
    Dim geomObj As GeometryObject = m_geo1.Current

    ' Return the Solid if it has a Volume
    If TypeOf geomObj Is Solid Then
      Dim solid As Solid = DirectCast(geomObj, Solid)
      If solid.Volume > 0 Then
        Return solid
      End If
    End If

  Loop

  ' Failure
  Return Nothing

End Function
```

5. The next helper routine we need will create an analysis style in the model for us. Place this function just after the `GetGeometry` function we just added:

```
''' <summary>
''' Create an Analysis Display Style
''' </summary>
Private Sub CreateAnalysisStyle(ByVal d As Document,
                               ByVal v As View)

  ' Start a New Transaction
  Using t As New Transaction(d)
    If t.Start("Create AVF Style") Then
```

```
                ' Surface Settings
                Dim m_faces As New  _
                    AnalysisDisplayColoredSurfaceSettings()
                m_faces.ShowGridLines = False

                ' Colors
                Dim m_colors As New  _
                    AnalysisDisplayColorSettings()

                ' Display Legend
                Dim m_legend As New  _
                    AnalysisDisplayLegendSettings()
                m_legend.ShowLegend = True
                m_legend.NumberOfSteps = 10

                ' Create the Style
                Dim m_style As AnalysisDisplayStyle
                m_style = _
                    AnalysisDisplayStyle.CreateAnalysisDisplayStyle( _
                    d, "WallLength", m_faces,
                    m_colors, m_legend)

                ' Set the Style to the View
                v.AnalysisDisplayStyleId = m_style.Id

                ' Commit Transaction
                t.Commit()

              End If

            End Using

          End Sub
```

6. This is the last of the three helper functions that we need and it performs the actual coloring of the elements in the model. Place this routine after the `CreateAnalysisStyle` function:

```
''' <summary>
''' Colorize the Wall
''' </summary>
Private Sub PaintSolid(ByVal doc As Document,
                       ByVal s As Solid,
                       ByVal value As Double)

  ' Active View
  Dim v As View = doc.ActiveView
```

```
' Create the Style if Necessary
If v.AnalysisDisplayStyleId =
     ElementId.InvalidElementId Then
  CreateAnalysisStyle(doc, v)
End If

' Get the Spatial Field Manager
Dim sfm As SpatialFieldManager = _
    SpatialFieldManager.GetSpatialFieldManager(v)

' Create one if missing
If sfm Is Nothing Then
  sfm = _
   SpatialFieldManager.CreateSpatialFieldManager(v, 1)
End If

' Get any Registered Results
If _schemaId = -1 Then
  Dim results As IList(Of Integer) = _
      sfm.GetRegisteredResults()
  If Not results.Contains(_schemaId) Then
    _schemaId = -1
  End If

End If

If _schemaId = -1 Then

  Dim resultSchema1 As New  _
    AnalysisResultSchema("WallLength",
                         "Color Wall by Length")

  _schemaId = sfm.RegisterResult(resultSchema1)
End If

' Get the Faces and Transform
Dim faces As FaceArray = s.Faces
Dim trf As Transform = Transform.Identity

' Process and Colorize each Face
For Each f As Face In faces
  Dim idx As Integer = _
      sfm.AddSpatialFieldPrimitive(f, trf)

  Dim uvPts As IList(Of UV) = New List(Of UV)()
  Dim doubleList As New List(Of Double)()
  Dim valList As IList(Of ValueAtPoint) = _
      New List(Of ValueAtPoint)()
```

```
            Dim bb As BoundingBoxUV = f.GetBoundingBox()
            uvPts.Add(bb.Min)
            doubleList.Add(value)
            valList.Add(New ValueAtPoint(doubleList))

            ' Points and Values
            Dim pnts As New FieldDomainPointsByUV(uvPts)
            Dim vals As New FieldValues(valList)

            ' Update the Primitives
            sfm.UpdateSpatialFieldPrimitive(idx,
                                            pnts,
                                            vals,
                                            _schemaId)

          Next

        End Sub
```

7. Now that we have our helper functions all in place, we can begin to build up our `Execute` function code that brings this all together. The rest of the steps will all include code that needs to be added to the main `Execute` function. First we need to get the `Application`, `Document`, and `ActiveUIDocument` references. Place the following lines of code just after the line that reads `' Begin Code Here`:

```
      Dim m_app As UIApplication
      m_app = commandData.Application
      Dim m_uidoc As UIDocument
      m_uidoc = m_app.ActiveUIDocument
      Dim m_doc As Document
      m_doc = m_uidoc.Document
```

8. The next step is to get all of the wall instances in the model:

```
      ' Get Wall Instances
      Dim m_col As New FilteredElementCollector(m_doc)
      m_col.WhereElementIsNotElementType()
      m_col.OfClass(GetType(Wall))
      Dim m_walls As New List(Of Wall)
      For Each x In m_col.ToElements
        m_walls.Add(TryCast(x, Wall))
      Next
```

9. Next we get the first 3D `ViewFamilyType` and use it to create a new 3D view:

```
      ' Get the First 3D View Type
      Dim m_3dvt As ViewFamilyType = Nothing
      Dim m_colV As New FilteredElementCollector(m_doc)
      m_colV.OfClass(GetType(ViewFamilyType))
```

```
For Each x In m_colV.ToElements
  Dim m_vt As ViewFamilyType = _
      TryCast(x, ViewFamilyType)
  If Not m_vt Is Nothing Then
    If m_vt.ViewFamily = _
            ViewFamily.ThreeDimensional Then
      m_3dvt = m_vt
      Exit For
    End If
  End If
Next

' Start a New Transaction
Dim m_v As View3D = Nothing
Using t As New Transaction(m_doc, "New 3D View")
  If t.Start Then

    ' Create a New 3D View
    m_v = View3D.CreateIsometric(m_doc, m_3dvt.Id)
    m_v.Name = "PACKT Analysis 3D View"

    ' Commit
    t.Commit()

  End If
End Using
```

10. Next we set the new view active:

```
' Set this View Active
m_uidoc.ActiveView = m_v
```

11. Next we hide all categories in the new 3D view:

```
Using t As New Transaction(m_doc, "Hiding Objects")
  If t.Start Then

    ' Turn off all Categories in the New View
    For Each c As Category In m_doc.Settings.Categories
      Try
        c.Visible(m_v) = False
      Catch
      End Try
    Next

    ' Commit
    t.Commit()

  End If
End Using
```

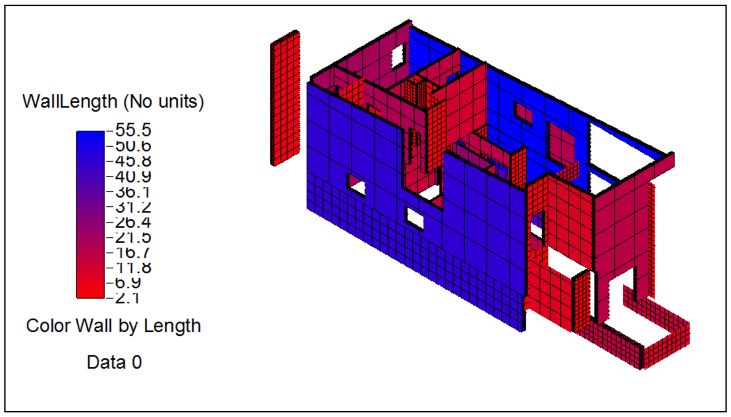
12. We are only concerned about the coarse view level of detail for our wall elements:

```
' Geometry Options
Dim m_opt As New Options
m_opt.IncludeNonVisibleObjects = True
m_opt.DetailLevel = ViewDetailLevel.Coarse
```

13. The last bit of code is where we iterate over each of the wall elements and extract the solid geometry to send it to our coloring routine:

```
' Process the Walls
For Each w As Wall In m_walls

  ' Color by Length Value
  Dim m_length As Double = _
      w.Parameter("Length").AsDouble

  ' Ignore Tiny Walls
  If m_length > 1 Then

    ' Get the Wall's Solid
    Dim geo As GeometryElement = w.Geometry(m_opt)
    Dim m_solid As Solid = GetGeometry(geo)
    If Not m_solid Is Nothing Then

      ' Paint the Results
      PaintSolid(m_doc, m_solid, m_length)

    End If

  End If

Next
```

14. Open the `rac_basic_sample_project.rvt` from the sample code bundle and run the command. You should end up with a view similar to the one shown in the following figure:



How it works...

We first created a clean 3D view named `PACKT Analysis 3D View` and made it the active view. We then turned off all model categories so only the analysis data would stand out.

The next thing we needed to do was get or create an analysis display style and apply it to our new 3D view, which is exactly what our `CreateAnalysisStyle` function does. This function is where we configure how we want our legend to display the resulting information to us as well as any coloring or surface grid settings that we want to use.

We then iterate through the list of walls in our model and use the `Length` parameter as a value to drive our analysis coloring. Longer walls display blue while shorter length walls display in a bright red. We dug into the wall elements with our `GetGeometry` function getting their raw solids and then passing the resulting solids over to our `PaintSolids` function where all the analysis colorization was managed.

Subscribing and unsubscribing to events (Should know)

There are quite a few events supported by the Revit API. You can turn events on and off from the ribbon if you like. Another topic that we will discuss in this recipe is how to override standard Revit commands. This is handy when you might need to disable a feature entirely for your users.

Anytime you want to enable an event in your application, you must first subscribe to the event and tell Revit what you want it to do when that event occurs. In this recipe, we will learn how to subscribe to the `DocumentOpened` event as well as how to unsubscribe to it when the Revit session closes.

Getting ready

Don't forget to edit the `.addin` file with this new `DBApplicationEvents` application so we can launch and debug it.

How to do it...

1. First create a new class named `DBApplicationEvents` and paste the code from our `DBApplicationBoilerPlate` class into this one. We will use `DBApplication` for this sample.

2. Add the following simple subroutine to this class that we will point our event handler to execute when an event occurs in the Revit session:

```
''' <summary>
''' Document Opened Event Handler
''' </summary>
Private Sub DocOpened(ByVal sender As Object,
                      ByVal e As DocumentOpenedEventArgs)

  ' Basic Popup Message
  MsgBox("You Opened a New Doc!" & vbCr &
         e.Document.PathName,
         MsgBoxStyle.Information,
         "PACKT Event Message")

End Sub
```

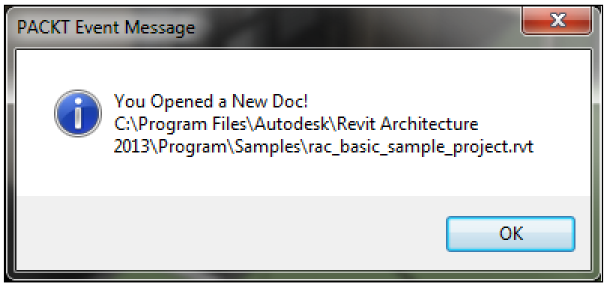
3. Subscribing to an event is done by using an `AddHandler` call inside our `OnStartup` function. Place the following code just beneath the line that reads `' Begin Code Here`:

```
    AddHandler application.DocumentOpened,
        New EventHandler(Of DocumentOpenedEventArgs)(AddressOf
DocOpened)
```

4. Unsubscribing to an event is done by using a `RemoveHandler` call inside our `OnShutdown` function. Place the following code just beneath the line that reads `' Begin Code Here`:

```
 RemoveHandler application.DocumentOpened,
  New EventHandler(Of _
  DocumentOpenedEventArgs)(AddressOf DocOpened)
```

5. Now run the tool and open a document.



You Opened a New Doc!
C:\Program Files\Autodesk\Revit Architecture 2013\Program\Samples\rac_basic_sample_project.rvt

How it works...

The event is added and pointed to the function named `DocOpened`, which is a subroutine that we added that could easily contain any code we want.