* 1. 

Hands-On Lab

Microsoft Office 2010 UI Customization

* 1. Lab version: 1.0.0
  2. Last updated: 1/4/2011
  3. 

Contents

[Overview 3](#_Toc281230116)

[Exercise 1: Customizing Office with Ribbons and Backstage 4](#_Toc281230117)

[Task 1 – Adding Export Backstage Tab 4](#_Toc281230118)

[Task 2 – Connecting the Backstage Tab to the Document Properties 9](#_Toc281230119)

[Task 3 – Adding Quick Export Buttons to the Home Ribbon Tab 15](#_Toc281230120)

[Task 4 – Adding the Export Functionality 17](#_Toc281230121)

[Exercise 1 Verification 19](#_Toc281230122)

[Exercise 2: Create a Custom Task Pane 20](#_Toc281230123)

[Task 1 – Creating a Custom Task Pane 21](#_Toc281230124)

[Task 2 – Adding Snippets Ribbon Button 22](#_Toc281230125)

[Exercise 2 Verification 23](#_Toc281230126)

[Exercise 3: Outlook Form Regions 24](#_Toc281230127)

[Task 1 – Creating a Form Region 24](#_Toc281230128)

[Task 2 – Implementing the Form Region 27](#_Toc281230129)

[Exercise 3 Verification 34](#_Toc281230130)

[Summary 35](#_Toc281230131)

Overview

* 1. In this lab, you will explore the different ways in which Microsoft Office 2010’s UI can be customized to provide a platform for building rich office applications. You will make use of the Visual Studio support for Microsoft Office 2010.

# Objectives

* 1. In this lab you will:
  + Customize Microsoft Office 2010’s UI using the Ribbon and Backstage
  + Build a custom Ribbon using Visual Studio 2010
  + Build a custom Outlook Form Region using Visual Studio 2010

# System Requirements

* 1. You must have the following items to complete this lab:
  + Microsoft Office Professional Plus 2010 (RC) (32-bit or 64-bit)
  + Microsoft Visual Studio 2010
  + Microsoft .Net Framework 4

# Exercises

* 1. This Hands-On Lab is comprised by the following exercises:
  2. Customizing Office with Ribbons and Backstage
  3. Customizing the Office UI with Add-Ins
  4. Outlook Form Regions
  5. Estimated time to complete this lab: **60 minutes**.

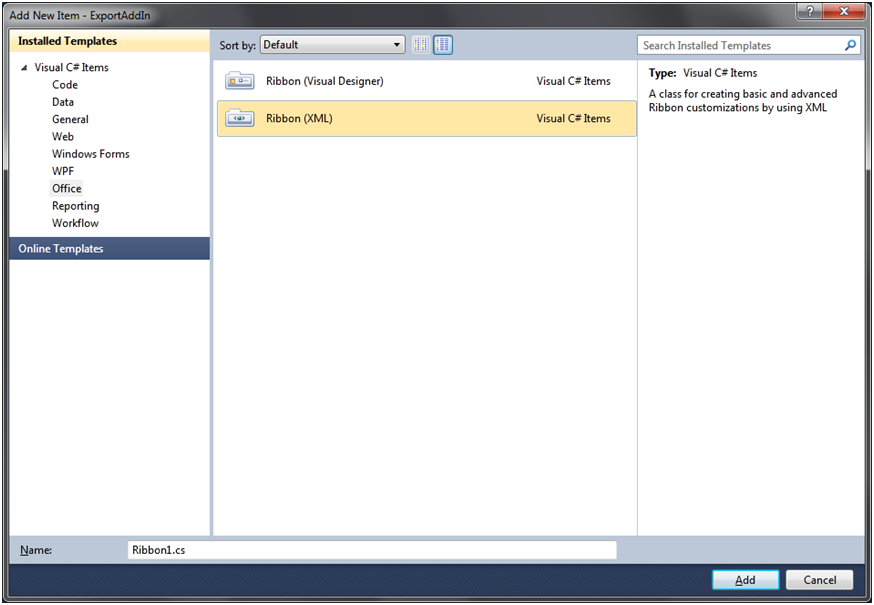
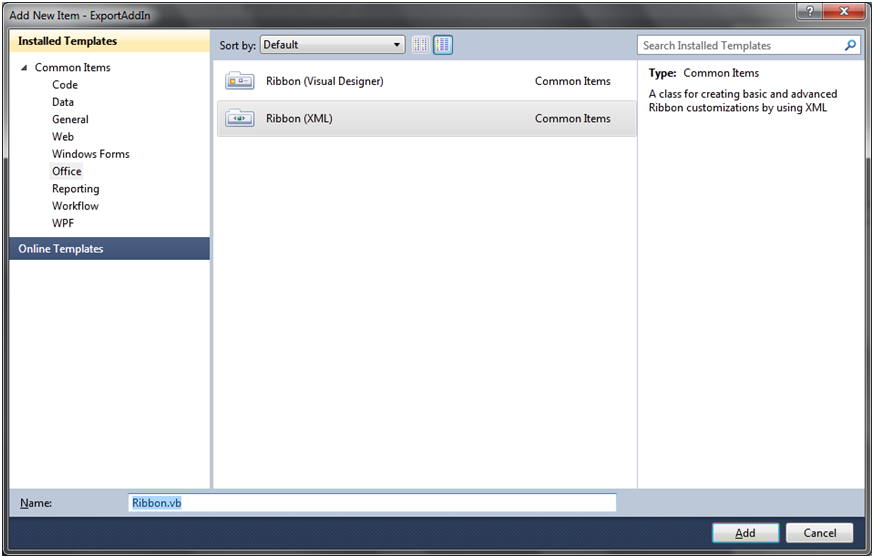
# Setup

* 1. All the requisites for this lab are verified using the **Configuration Wizard**. To make sure that everything is correctly configured, follow these steps.
  2. Run the **Configuration Wizard** for the Training Kit if you have not done it previously. To do this, browse to **Source\Setup** folder of this lab, and double-click the **Dependencies.dep** file. Install any pre-requisites that are missing (rescanning if necessary) and complete the wizard.
     1. **Note:** The Configuration Wizard is used for checking dependencies and setting up the environment. If the Configuration Wizard is not installed on your machine, you must install it running the DependencyChecker.msi file located on the %VS2010TKInstallationFolder%\Assets\DependencyChecker folder (e.g. by default the Training Kit is installed under C:\VS2010TrainingKit).
     2. For convenience, much of the code you will be managing along this lab is available as Visual Studio code snippets. The **Dependencies.dep** file launches the Visual Studio installer file that installs the code snippets.

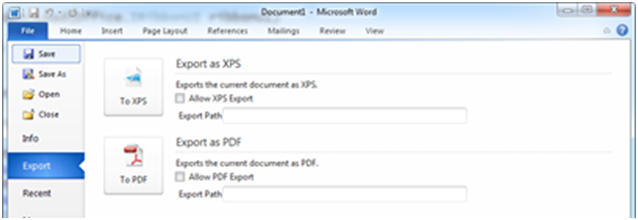
Exercise 1: Customizing Office with Ribbons and Backstage

1. Backstage is a perfect place to put UI elements that take up space but are not necessary when actively editing the document. In this case you will be using Backstage and a Ribbon together to create multiple parts of a complete add-in.

Task 1 – Adding Export Backstage Tab

* 1. In this first task, you will create a new Backstage tab named Export to manage the quick export functionality.
  2. Open **Microsoft Visual Studio 2010** and open the **ExportAddIn.sln** solution located at **Source\Ex1-ExportAddIn\Begin\** (Choosing the folder that matches the language of your preference.)
  3. Add a new **XML Ribbon** item to the existing **ExportAddIn** project. To do this:
     1. Right-click **ExportAddIn** project in the **Solution Explorer** and choose **Add\New Item**.
     2. In the **Add New Item** dialog, select the **Office** category, and choose the **Ribbon (XML)** item.
     3. Type Ribbon as the **Name** and click **Add** to create the new ribbon.
        1. 
        2. Figure 1
        3. Add Ribbon (XML) Item (C#)
        4. 
        5. Figure 2
        6. Add Ribbon (XML) Item (VB)
     4. In the Solution Explorer, double-click the **Ribbon.xml** file to open it.
  4. Add XML Markup to the **Ribbon.xml** file to define the **Export** Backstage tab
     1. Remove the current **<ribbon>** node from the **Ribbon.xml** file.
     2. Add the following markup to define the Backstage tab and the first column of information.
        1. *(Code Snippet – Office UI Customization – Backstage Tab – XML)*
        2. XML
        3. **<backstage>**
        4. **<tab id="tabExport" label="Export" insertAfterMso="TabInfo">**
        5. **<firstColumn>**
        6. **</firstColumn>**
        7. **</tab>**
        8. **</backstage>**
     3. Inside the **<firstColumn>** element, add the following markup to define the group containing the **XPS** export controls.
        1. *(Code Snippet – Office UI Customization – Backstage XPS Button – XML)*
        2. XML
        3. **<group id="grpExportXPS" label="Export as XPS" helperText="Exports the current document as XPS.">**
        4. **<primaryItem>**
        5. **<button id="btnBackStageXPS" label="To XPS" getImage="GetButtonImage" />**
        6. **</primaryItem>**
        7. **<topItems>**
        8. **<checkBox id="chkEnableXPS" label="Allow XPS Export" />**
        9. **<editBox id="txtXPSPath" label="Export Path" sizeString="WWWWWWWWWWWWWWWWWWWWWWWWWWWWWW" />**
        10. **</topItems>**
        11. **</group>**
        12. **Note:** The XPS output controls are made up of a button to perform the export, a check box to enable or disable exporting to XPS and an edit box to define the filename of the exported file.
     4. Immediately following the previous **group**, add the following markup to define the **PDF** export controls
        1. *(Code Snippet – Office UI Customization – Backstage PDF Button – XML)*
        2. XML
        3. **<group id="grpExportPDF" label="Export as PDF" helperText="Exports the current document as PDF.">**
        4. **<primaryItem>**
        5. **<button id="btnBackStagePDF" label="To PDF" getImage="GetButtonImage" />**
        6. **</primaryItem>**
        7. **<topItems>**
        8. **<checkBox id="chkEnablePDF" label="Allow PDF Export" />**
        9. **<editBox id="txtPDFPath" label="Export Path" sizeString="WWWWWWWWWWWWWWWWWWWWWWWWWWWWWW" />**
        10. **</topItems>**
        11. **</group>**
  5. Define the code that will load images for the export buttons.
     1. Open **Ribbon.cs** (C#) or **Ribbon.vb** (VB) by double-clicking it in the **Solution Explorer**.
     2. Add the following **GetButtonImage** method to the **Ribbon** class.
        1. *(Code Snippet – Office UI Customization – GetButtonImage Method – CSharp)*
        2. C#
        3. **public System.Drawing.Bitmap GetButtonImage(Office.IRibbonControl control)**
        4. **{**
        5. **switch (control.Id)**
        6. **{**
        7. **case "btnBackStageXPS":**
        8. **return Properties.Resources.XPS;**
        9. **case "btnBackStagePDF":**
        10. **return Properties.Resources.PDF;**
        11. **default:**
        12. **return null;**
        13. **}**
        14. **}**
        15. *(Code Snippet – Office UI Customization – GetButtonImage Method – VB)*
        16. Visual Basic
        17. **Public Function GetButtonImage(ByVal control As Office.IRibbonControl) As System.Drawing.Bitmap**
        18. **Select Case control.Id**
        19. **Case "btnBackStageXPS"**
        20. **Return My.Resources.XPS**
        21. **Case "btnBackStagePDF"**
        22. **Return My.Resources.PDF**
        23. **Case Else**
        24. **Return Nothing**
        25. **End Select**
        26. **End Function**
        27. **Note:** The **getImage** attribute in the **button** elements define a public method Office will call any time a button loads. This method uses a switch command to differentiate between the different buttons.
  6. Override the **CreateRibbonExtensibilityObject** method in the add-in to define the ribbon to load
     1. In the Solution Explorer, double-click on the **ThisAddIn** class, located under the Word node, to open it.
     2. Add the following namespace statement to it.
        1. C#
        2. **using Microsoft.Office.Core;**
        3. Visual Basic
        4. **Imports Microsoft.Office.Core**
     3. In the **ThisAddIn** class, override the **CreateRibbonExtensibilityObject** method and return a new Ribbon object.
        1. *(Code Snippet – Office UI Customization – CreateRibbonExtensibilityObject Method– CSharp)*
        2. C#

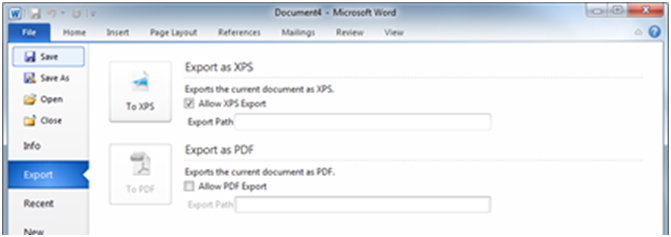
**protected override IRibbonExtensibility CreateRibbonExtensibilityObject()**

* + - 1. **{**
      2. **return new Ribbon();**
      3. **}**
      4. *(Code Snippet – Office UI Customization – CreateRibbonExtensibilityObject Method– VB)*
      5. Visual Basic
      6. **Protected Overrides Function CreateRibbonExtensibilityObject() As Microsoft.Office.Core.IRibbonExtensibility**
      7. **Return New Ribbon()**
      8. **End Function**
  1. Run the add-in and verify the Backstage tab looks correct.
     1. In the **Debug** menu, click **Start Without Debugging**.
     2. Once **Word 2010** loads, click the **Backstage** button (labeled as **File**.)
     3. In **Backstage** switch to the **Export** tab and verify it looks like the image below.
        1. 
        2. Figure 3
        3. Custom Backstage Export tab

Task 2 – Connecting the Backstage Tab to the Document Properties

* 1. In this task, you will connect the controls in the **Export** tab to the **CustomDocumentProperties** object in the active document. This will allow the state of the controls to be persisted in the document and loaded when the document is loaded later.
  2. First, you will define the code required to determine when the button and editBox control are enabled.
  3. Open **Ribbon.xml** by double clicking it in the **Solution** **Explorer**.
  4. Add a **getEnabled** attribute to the two “**button”** and two “**editBox”** controls. The markup should look as the following:
     1. XML
     2. ...
     3. <button id="btnBackStageXPS" label="To XPS" getImage="GetButtonImage" **getEnabled="GetEnable"** />
     4. </primaryItem>
     5. <topItems>
     6. <checkBox id="chkEnableXPS" label="Allow XPS Export" />
     7. <editBox id="txtXPSPath" label="Export Path" sizeString="WWWWWWWWWWWWWWWWWWWWWWWWWWWWWW" **getEnabled="GetEnable"**/>
     9. ...
     11. <button id="btnBackStagePDF" label="To PDF" getImage="GetButtonImage" **getEnabled="GetEnable"**/>
     12. </primaryItem>
     13. <topItems>
     14. <checkBox id="chkEnablePDF" label="Allow PDF Export" />
     15. <editBox id="txtPDFPath" label="Export Path" sizeString="WWWWWWWWWWWWWWWWWWWWWWWWWWWWWW" **getEnabled="GetEnable"**/>
     16. …
     17. **Note:** You should find 2 **button** controls and 2 **editBox** controls.
  5. In the Solution Explorer, double-click the **Ribbon** class to open it.
  6. Add the following private field to it.
     1. C#
     2. **private ExportProperties m\_properties = new ExportProperties();**
     3. Visual Basic
     4. **Private m\_properties = New ExportProperties()**
     5. **Note:** The **ExportProperties** class is a wrapper provided for you that wraps the **CustomDocumentProperties** collection of the current **ActiveDocument**.
  7. Create a **GetEnable** method to determine the enable state of other controls.
     1. *(Code Snippet – Office UI Customization – GetEnable Method– CSharp)*
     2. C#
     3. **public bool GetEnable(Office.IRibbonControl control)**
     4. **{**
     5. **switch (control.Id)**
     6. **{**
     7. **case "btnBackStageXPS":**
     8. **case "txtXPSPath":**
     9. **return m\_properties.XpsEnabled;**
     10. **case "btnBackStagePDF":**
     11. **case "txtPDFPath":**
     12. **return m\_properties.PdfEnabled;**
     13. **default:**
     14. **return false;**
     15. **}**
     16. **}**
     17. *(Code Snippet – Office UI Customization – GetEnable Function– VB)*
     18. Visual Basic
     19. **Public Function GetEnable(ByVal control As Office.IRibbonControl) As Boolean**
     20. **Select Case control.Id**
     21. **Case "btnBackStageXPS", "txtXPSPath"**
     22. **Return m\_properties.XpsEnabled**
     23. **Case "btnBackStagePDF", "txtPDFPath"**
     24. **Return m\_properties.PdfEnabled**
     25. **Case Else**
     26. **Return False**
     27. **End Select**
     28. **End Function**
  8. Next, you will add the code required to populate and respond to events from the enabled checkboxes.
  9. Open **Ribbon.xml** by double clicking it in the **Solution** **Explorer**.
  10. Add a **getPressed** and **onAction** attribute to the two “**checkbox”** elements in the Backstage markup. The code should look as follows for each checkbox:
      1. XML
      2. <checkBox id="chkEnableXPS" label="Allow XPS Export" **getPressed="IsEnableChecked" onAction="EnableChecked"** />
      3. **Note:** The Id of the checkbox might change to match the group (either XPS or PDF).
  11. Open **Ribbon.cs** (C#)or **Ribbon.vb** (VB) file by double clicking it in the **Solution Explorer**.
  12. Create an **IsEnableChecked** method that will respond to requests from check box controls to get their current state.
      1. *(Code Snippet – Office UI Customization – IsEnabledChecked Method– CSharp)*
      2. C#
      3. **public bool IsEnableChecked(Office.IRibbonControl control)**
      4. **{**
      5. **if (control.Id == "chkEnableXPS")**
      6. **return m\_properties.XpsEnabled;**
      7. **else if (control.Id == "chkEnablePDF")**
      8. **return m\_properties.PdfEnabled;**
      9. **else**
      10. **return false;**
      11. **}**
      12. *(Code Snippet – Office UI Customization – IsEnabledChecked Function– VB)*
      13. Visual Basic
      14. **Public Function IsEnableChecked(ByVal control As Office.IRibbonControl) As Boolean**
      15. **If control.Id = "chkEnableXPS" Then**
      16. **Return m\_properties.XpsEnabled**
      17. **ElseIf control.Id = "chkEnablePDF" Then**
      18. **Return m\_properties.PdfEnabled**
      19. **Else**
      20. **Return False**
      21. **End If**
      22. **End Function**
  13. Create an **EnableChecked** method that will respond to the checking of a check box.
      1. *(Code Snippet – Office UI Customization – EnableChecked Method– CSharp)*
      2. C#
      3. **public void EnableChecked(Office.IRibbonControl control, bool value)**
      4. **{**
      5. **if (control.Id == "chkEnableXPS")**
      6. **{**
      7. **m\_properties.XpsEnabled = value;**
      8. **ribbon.InvalidateControl("btnBackStageXPS");**
      9. **ribbon.InvalidateControl("txtXPSPath");**
      10. **}**
      11. **else if (control.Id == "chkEnablePDF")**
      12. **{**
      13. **m\_properties.PdfEnabled = value;**
      14. **ribbon.InvalidateControl("btnBackStagePDF");**
      15. **ribbon.InvalidateControl("txtPDFPath");**
      16. **}**
      17. **}**
      18. *(Code Snippet – Office UI Customization – EnableChecked Sub– VB)*
      19. Visual Basic
      20. **Public Sub EnableChecked(ByVal control As Office.IRibbonControl, ByVal value As Boolean)**
      21. **If control.Id = "chkEnableXPS" Then**
      22. **m\_properties.XpsEnabled = value**
      23. **ribbon.InvalidateControl("btnBackStageXPS")**
      24. **ribbon.InvalidateControl("txtXPSPath")**
      25. **ElseIf control.Id = "chkEnablePDF" Then**
      26. **m\_properties.PdfEnabled = value**
      27. **ribbon.InvalidateControl("btnBackStagePDF")**
      28. **ribbon.InvalidateControl("txtPDFPath")**
      29. **End If**
      30. **End Sub**
      31. **Note:** The calls to **ribbon.InvalidateControl** force those controls to reevaluate their state. This will lead to another call to **GetEnabled** that will now have a different result.
  14. Finally, you will add the code required to populate and respond to events from the path edit boxes.
  15. Open **Ribbon.xml** by double clicking it in the **Solution** **Explorer**.
  16. Add a **getText** and **onChange** attribute to the two “**editBox”** elements, the markup should look as follows:
      1. XML

<editBox id="txtPDFPath" **getText="GetExportPath" onChange="SetExportPath"** ... />

* + 1. **Note:** The Id of the editBox might change to match the group (either XPS or PDF).
  1. In the Solution Explorer, double-click the **Ribbon.cs** (C#) or **Ribbon.vb** (VB) to open it.
  2. Create a **GetExportPath** method that will retrieve the current export path for the **editBox** controls.
     1. *(Code Snippet – Office UI Customization – GetExportPath Method– CSharp)*
     2. C#
     3. **public string GetExportPath(Office.IRibbonControl control)**
     4. **{**
     5. **if (control.Id == "txtXPSPath")**
     6. **return m\_properties.XpsExportPath;**
     7. **else if (control.Id == "txtPDFPath")**
     8. **return m\_properties.PdfExportPath;**
     9. **else**
     10. **return string.Empty;**
     11. **}**
     12. *(Code Snippet – Office UI Customization – GetExportPath Function– VB)*
     13. Visual Basic
     14. **Public Function GetExportPath(ByVal control As Office.IRibbonControl) As String**
     15. **If control.Id = "txtXPSPath" Then**
     16. **Return m\_properties.XpsExportPath**
     17. **ElseIf control.Id = "txtPDFPath" Then**
     18. **Return m\_properties.PdfExportPath**
     19. **Else**
     20. **Return String.Empty**
     21. **End If**
     22. **End Function**
  3. Create a **SetExportPath** method that will respond to the change event of the editBox control.
     1. *(Code Snippet – Office UI Customization – SetExportPath Method– CSharp)*
     2. C#
     3. **public void SetExportPath(Office.IRibbonControl control, string value)**
     4. **{**
     5. **if (control.Id == "txtXPSPath")**
     6. **m\_properties.XpsExportPath = value;**
     7. **else if (control.Id == "txtPDFPath")**
     8. **m\_properties.PdfExportPath = value;**
     9. **}**
     10. *(Code Snippet – Office UI Customization – SetExportPath Sub– VB)*
     11. Visual Basic
     12. **Public Sub SetExportPath(ByVal control As Office.IRibbonControl, ByVal value As String)**
     13. **If control.Id = "txtXPSPath" Then**
     14. **m\_properties.XpsExportPath = value**
     15. **ElseIf control.Id = "txtPDFPath" Then**
     16. **m\_properties.PdfExportPath = value**
     17. **End If**
     18. **End Sub**
  4. Run the add-in and verify the enable checkboxes work.
     1. In the **Debug** menu, click **Start Without Debugging**.
     2. Once **Word 2010** loads, click the **Backstage** button (labeled as **File**.)
     3. In **Backstage** switch to the **Export** tab.
     4. Check and uncheck the enabled checkboxes and verify the related controls are enabled and disabled.
        1. 
        2. Figure 4
        3. Enable check box in Backstage tab

Task 3 – Adding Quick Export Buttons to the Home Ribbon Tab

* 1. In this task, you will add a **Quick Export** ribbon button to the Home ribbon tab. It will tie into the same functionality as the buttons on Backstage.
  2. Double-click the **Ribbon.xml** file in the **Solution** **Explorer** to open it.
  3. Add the markup that will create a new **Quick Export** group in the **Home** tab. Make sure the **<ribbon>** xml element is added before the **<backstage>** xml element.
     1. *(Code Snippet – Office UI Customization – Ribbon XML Element– XML)*
     2. XML
     3. **<ribbon>**
     4. **<tabs>**
     5. **<tab idMso="TabHome">**
     6. **<group id="grpQuickExport" autoScale="true">**
     7. **</group>**
     8. **</tab>**
     9. **</tabs>**
     10. **</ribbon>**
     11. **Note:** The use of the **idMso** attribute tells Office to put the following group in the existing tab.
  4. Inside the **grpQuickExport** group element, add a **menu** control using an existing image within Office.
     1. XML
     2. **<menu id="btnQuickExport" size="large" itemSize="large" label="Quick Export" imageMso="PageMenu" >**
     3. **</menu>**
     4. **Note:** The use of **imageMso** tells Office to use an built in Office image.
  5. Inside the **<menu>** element, add the two **button** controls to perform the export as **XPS** or **PDF**.
     1. XML
     2. **<button id="btnRibbonXPS" label="XPS" getEnabled="GetEnable" getImage="GetButtonImage" />**
     3. **<button id="btnRibbonPDF" label="PDF" getEnabled="GetEnable" getImage="GetButtonImage" />**
  6. Update the code behind to update the **Ribbon** buttons as well as the **Backstage** buttons.
     1. In the Solution Explorer, double-click the **Ribbon** class to open it.
     2. Locate the **GetButtonImage** method and add cases for the ribbon buttons.
        1. C#
        2. **case "btnRibbonXPS":**
        3. case "btnBackStageXPS":
        4. return Properties.Resources.XPS;
        5. **case "btnRibbonPDF":**
        6. case "btnBackStagePDF":
        7. return Properties.Resources.PDF;
        8. Visual Basic
        9. Case "btnBackStageXPS", **"btnRibbonXPS"**
        10. Return My.Resources.XPS
        11. Case "btnBackStagePDF", **"btnRibbonPDF"**
        12. Return My.Resources.PDF
     3. Locate the **GetEnable** method and add cases for the ribbon buttons.
        1. C#
        2. **case "btnRibbonXPS":**
        3. case "btnBackStageXPS":
        4. case "txtXPSPath":
        5. return m\_properties.XpsEnabled;
        6. **case "btnRibbonPDF":**
        7. case "btnBackStagePDF":
        8. case "txtPDFPath":
        9. return m\_properties.PdfEnabled;
        10. Visual Basic
        11. Case "btnBackStageXPS", "txtXPSPath"**, "btnRibbonXPS"**
        12. Return m\_properties.XpsEnabled
        13. Case "btnBackStagePDF", "txtPDFPath"**, "btnRibbonPDF"**
        14. Return m\_properties.PdfEnabled
     4. Locate the **EnableChecked** method and add **InvalidateControl** calls for the ribbon buttons.
        1. C#
        2. if (control.Id == "chkEnableXPS")
        3. {
        4. m\_properties.XpsEnabled = value;
        5. **ribbon.InvalidateControl("btnRibbonXPS");**
        6. ribbon.InvalidateControl("btnBackStageXPS");
        7. ribbon.InvalidateControl("txtXPSPath");
        8. }
        9. else if (control.Id == "chkEnablePDF")
        10. {
        11. m\_properties.PdfEnabled = value;
        12. **ribbon.InvalidateControl("btnRibbonPDF");**
        13. ribbon.InvalidateControl("btnBackStagePDF");
        14. ribbon.InvalidateControl("txtPDFPath");
        15. }
        16. Visual Basic
        17. If control.Id = "chkEnableXPS" Then
        18. m\_properties.XpsEnabled = value
        19. **ribbon.InvalidateControl("btnRibbonXPS")**
        20. ribbon.InvalidateControl("btnBackStageXPS")
        21. ribbon.InvalidateControl("txtXPSPath")
        22. ElseIf control.Id = "chkEnablePDF" Then
        23. m\_properties.PdfEnabled = value
        24. **ribbon.InvalidateControl("btnRibbonPDF")**
        25. ribbon.InvalidateControl("btnBackStagePDF")
        26. ribbon.InvalidateControl("txtPDFPath")
        27. End If

Task 4 – Adding the Export Functionality

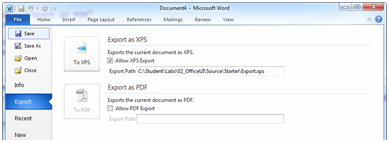
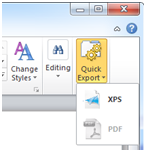
* 1. In this task, you will add the Export Functionality to the Add-In to actually export the file to the selected format.
  2. Open **Ribbon.cs** (C#) or **Ribbon.vb** (VB) by double clicking it in the **Solution Explorer**.
  3. Add the following namespace to the file.
     1. C#
     2. **using Microsoft.Office.Interop.Word;**
     3. Visual Basic
     4. **Imports Microsoft.Office.Interop.Word**
  4. Create a new method named **ExportDocument** that will use the **ExportAsFixedFormat** document method to export the current document.
     1. **Note:** For the sake of simplicity, you will not add validation. Note that if you enter an invalid path, Word will crush, throwing an exception.
     2. *(Code Snippet – Office UI Customization – ExportDocument Method– CSharp)*
     3. C#
     4. **public void ExportDocument(Office.IRibbonControl control)**
     5. **{**
     6. **switch (control.Id)**
     7. **{**
     8. **case "btnRibbonXPS":**
     9. **case "btnBackStageXPS":**
     10. **Globals.ThisAddIn.Application.ActiveDocument.**
     11. **ExportAsFixedFormat(**
     12. **m\_properties.XpsExportPath,**
     13. **WdExportFormat.wdExportFormatXPS);**
     14. **break;**
     15. **case "btnRibbonPDF":**
     16. **case "btnBackStagePDF":**
     17. **Globals.ThisAddIn.Application.ActiveDocument.**
     18. **ExportAsFixedFormat(**
     19. **m\_properties.PdfExportPath,**
     20. **WdExportFormat.wdExportFormatPDF);**
     21. **break;**
     22. **}**
     23. **}**
     24. *(Code Snippet – Office UI Customization – ExportDocument Sub– VB)*
     25. Visual Basic
     26. **Public Sub ExportDocument(ByVal control As Office.IRibbonControl)**
     27. **Select Case control.Id**
     28. **Case "btnRibbonXPS", "btnBackStageXPS"**
     29. **Globals.ThisAddIn.Application.ActiveDocument.ExportAsFixedFormat(m\_properties.XpsExportPath, WdExportFormat.wdExportFormatXPS)**
     30. **Exit Select**
     31. **Case "btnRibbonPDF", "btnBackStagePDF"**
     32. **Globals.ThisAddIn.Application.ActiveDocument.ExportAsFixedFormat(m\_properties.PdfExportPath, WdExportFormat.wdExportFormatPDF)**
     33. **Exit Select**
     34. **End Select**
     35. **End Sub**
  5. Open **Ribbon.xml** by double clicking it in the **Solution Explorer**.
  6. Add the **onAction** attribute to the four “**button”** elements. Your markup should look as follows:
     1. XML
     2. <button id="btnRibbonXPS" label="XPS" getEnabled="GetEnable" getImage="GetButtonImage" **onAction="ExportDocument"**/>
     3. **Note:** The Id of the editBox might change to match the group (either XPS or PDF).

Exercise 1 Verification

* 1. In order to verify that you have correctly performed all steps in the above exercise, proceed as follows:

#### Test the Add-in

Test your add-in to confirm that the export ribbon and backstage buttons work as expected.

* 1. Run the add-in and verify the enable checkboxes work.
     1. In the **Debug** menu, click **Start Without Debugging**.
     2. Once **Word 2010** loads, type a phrase in the document and click the **Backstage** button (labeled as **File**.)
     3. In **Backstage** switch to the **Export** tab.
     4. Check the **Allow XPS Export** checkbox.
     5. Enter an Export Path, for example: **Export.xps**.
        1. **Note:** Remember you did not provide any validation for the export path. So make sure you enter a valid path.
        2. 
        3. Figure 5
        4. Export as XPS
  2. Click the **To XPS** button and using **Windows Explorer** navigate to the folder and open the **Export.xps** document. Close the .xps file.
  3. Verify the **Home** ribbon contains a new **Quick Format** button.
     1. Switch to the **Home** ribbon tab and type an additional phrase into the document.
     2. Click the **Quick Export** button and click the **XPS** button. Navigate to the .xps document and view it again to see your changes.
        1. 
        2. Figure 6
        3. Quick Export Ribbon Button
  4. When you are done cleanup and remove the add-in.
     1. Close **Word 2010**.
     2. In the **Solution Explorer**, right-click **ExportAddIn** and click **Clean**.

Exercise 2: Create a Custom Task Pane

1. In this exercise, you will create a custom task pane that can be used to insert snippets of text into the active document. A ribbon button will toggle visibility of a custom task pane that hosts a WPF User Control.

Task 1 – Creating a Custom Task Pane

* 1. In this first task, you will use a pre-created WPF user control as a task pane.
  2. Open **Microsoft Visual Studio 2010** and open the **WPFTaskPane.sln** solution located at **\Source\Ex2-WPFTaskPane\Begin\** (Choosing the folder that matches the language of your preference.)
  3. Open **ThisAddIn.cs** (C#) or **ThisAddIn.vb** (VB) by double clicking it in the **Solution Explorer**
  4. Locate the **ThisAddIn\_Startup** method and add the following code to create a new **UserControl**
     1. C#
     2. **UserControl taskPane = new UserControl();**
     3. Visual Basic
     4. **Dim taskPane = new UserControl()**
  5. Below, in the same method, create a new **ElementHost** control to contain the WPF user control and add it to the Windows Forms **UserControl**.
     1. C#
     2. **taskPane.Controls.Add(**
     3. **new ElementHost {**
     4. **Child = new TaskPane(),**
     5. **Dock = DockStyle.Fill });**
     6. Visual Basic
     7. **taskPane.Controls.Add(**
     8. **New ElementHost With {**
     9. **.Child = New TaskPane(),**
     10. **.Dock = DockStyle.Fill**
     11. **})**
  6. Next to the previous code, add the Windows forms user control to the add-in as a task pane, to do this:
     1. Create a new CustomTaskPane object wrapping the user control from the previous step.
        1. C#
        2. **Microsoft.Office.Tools.CustomTaskPane formulaTaskPane = CustomTaskPanes.Add(taskPane, "Custom Snippets");**
        3. Visual Basic
        4. **Dim formulaTaskPane = CustomTaskPanes.Add(taskPane, "Custom Snippets")**
     2. Set the visibility and size of the new task pane.
        1. C#
        2. **formulaTaskPane.Visible = true;**
        3. **formulaTaskPane.DockPosition = Office.MsoCTPDockPosition.msoCTPDockPositionRight;**
        4. **formulaTaskPane.Width = 300;**
        5. Visual Basic
        6. **formulaTaskPane.Visible = True**
        7. **formulaTaskPane.DockPosition = Office.MsoCTPDockPosition.msoCTPDockPositionRight**
        8. **formulaTaskPane.Width = 300**

Task 2 – Adding Snippets Ribbon Button

In this second task, you will add a ribbon button that will toggle the visibility of the snippets task pane.

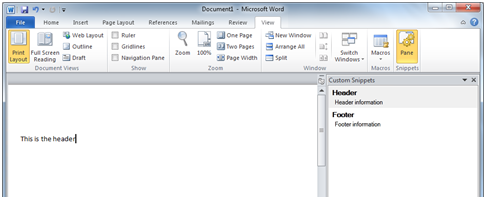
* 1. Open **Ribbon.xml** by double clicking it in the **Solution Explorer**.
  2. Add the tab element referencing the standard **View** ribbon tab.
     1. XML
     2. <ribbon>
     3. <tabs>
     4. **<tab idMso="TabView">**
     5. **</tab>**
     6. </tabs>
     7. </ribbon>
  3. Add a snippets group and toggle button to the **<tab>** element.
     1. XML
     2. **<group id="GroupSnippets" label="Snippets">**
     3. **<toggleButton id="Snippets" label="Pane" size="large"**
     4. **imageMso="PageMenu" getPressed="IsSnippetsPressed"**
     5. **onAction="SnippetsToggle" />**
     6. **</group>**
  4. Add the code behind to process the button toggle changes, to do this:
     1. Open **Ribbon.cs** (C#) or **Ribbon.vb** (VB) by double clicking it in the **Solution Explorer**
     2. Add an **IsSnippetPressed** method to the **Ribbon** class
        1. C#
        2. **public bool IsSnippetsPressed(Office.IRibbonControl control)**
        3. **{**
        4. **return Globals.ThisAddIn.CustomTaskPanes[0].Visible;**
        5. **}**
        6. Visual Basic
        7. **Public Function IsSnippetsPressed(ByVal control As Office.IRibbonControl) As Boolean**
        8. **Return Globals.ThisAddIn.CustomTaskPanes(0).Visible**
        9. **End Function**
        10. **Note:** The status is set using the Visible property of the CustomTaskPane object.
     3. Below, in the same class, add a **SnippetsToggle** method to the **Ribbon** class.
        1. C#
        2. **public void SnippetsToggle(Office.IRibbonControl control, bool value)**
        3. **{**
        4. **Globals.ThisAddIn.CustomTaskPanes[0].Visible = value;**
        5. **}**
        6. Visual Basic
        7. **Public Sub SnippetsToggle(ByVal control As Office.IRibbonControl, ByVal value As Boolean)**
        8. **Globals.ThisAddIn.CustomTaskPanes(0).Visible = value**
        9. **End Sub**
        10. **Note:** The value of the button is stored directly in the CustomTaskPane object.

Exercise 2 Verification

* 1. In order to verify that you have correctly performed all steps in the above exercise, proceed as follows:

#### Test the Add-in

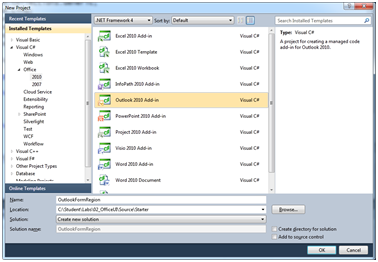
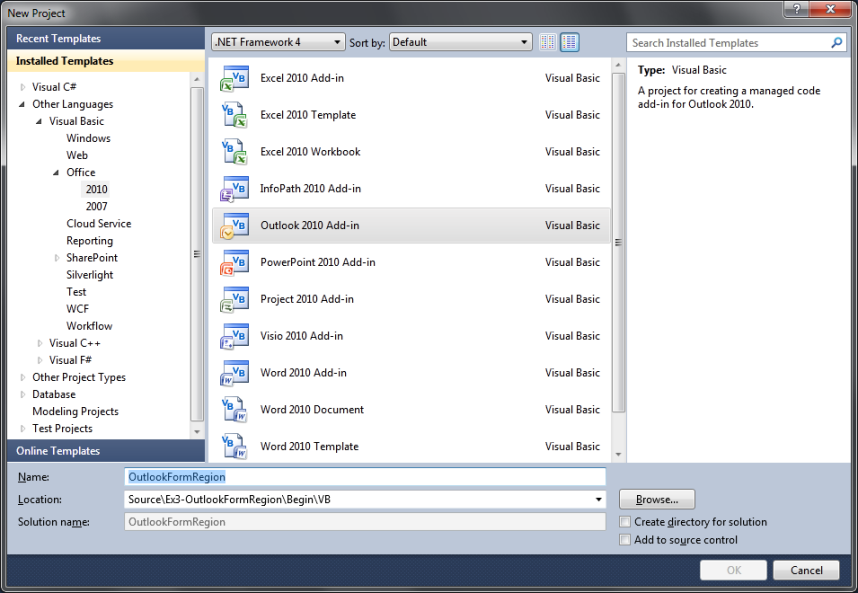
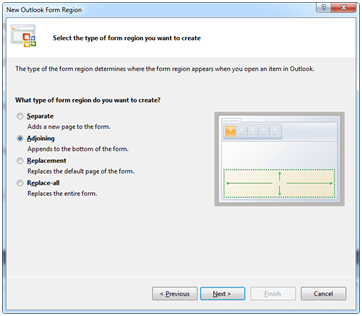
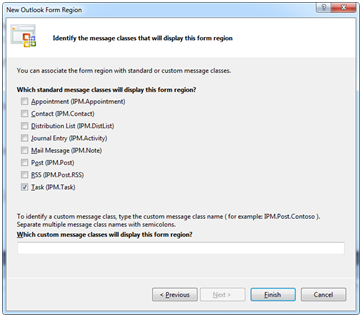
Test your add-in to confirm that the export ribbon and backstage buttons work as expected.

* 1. Run the add-in and verify the enable checkboxes work.
     1. In the **Debug** menu, click **Start Without Debugging**.
     2. Once **Word 2010** loads, switch to the Views ribbon tab.
     3. On the **View** tab, click the **Pane** button in the **Snippets** group and verify the task pane is shown and hidden.
     4. Double click one of the items in the task pane and verify the content is added to the document.
        1. 
        2. Figure 7
        3. Custom Task Pane
  2. When you are done cleanup and remove the add-in.
     1. Close **Word 2010**.
     2. In the **Solution Explorer**, right-click **WPFTaskPane** and click **Clean**.

Exercise 3: Outlook Form Regions

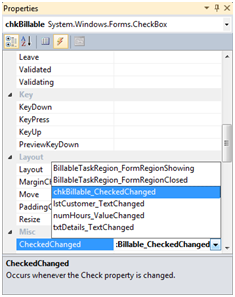
1. In this exercise, you will use an Outlook Form Region to extend the Task form in Outlook. The extensions will allow tracking of billable tasks within Outlook’s storage location.

Task 1 – Creating a Form Region

* 1. In this first task, you will create a new Outlook Form Region attached to the existing Task forms in Outlook.
  2. Open **Microsoft Visual Studio 2010** and create a new Outlook 2010 Add-in project. To do this, Select **File\New Project**.
  3. In the New Project dialog choose an **Outlook 2010 Add-in** template, located under **Office\ 2010**.
  4. Set the **Name** of the project to **OutlookFormRegion** and set the location to **\Source\Ex3-OutlookFormRegion\Begin\** (choosing the folder that matches the language of your preference.)
  5. Verify Create directory for solution is unchecked and click **OK** to create the new project.
     1. 
     2. Figure 8
     3. New Outlook 2010 Addin Project (C#)
     4. 
     5. Figure 9
     6. New Outlook 2010 Addin Project (VB)
  6. Add a new form region named **BillableTaskRegion** to the Task form, to do this:
     1. Right click on **OutlookFormRegion** in the Solution Explorer and click **Add\ New Item**.
     2. In the **Add New Item** dialog choose **OutlookFormRegion** from the **Office** templates.
     3. Type **BillableTaskRegion** as name, and click **Add** to create the new item.
     4. In the first page of the **New Outlook Form Region** wizard, choose **Design a new form region** and click **Next**.
     5. In the next page, select **Adjoining** and click **Next**.
        1. 
        2. Figure 10
        3. Choose Form Region Type
     6. Change the name to **Billable Task** and click **Next**.
     7. Choose the **Task** option and clear all other check boxes then click **Finish**.
        1. 
        2. Figure 11
        3. Choose Form Region Class
        4. **Note:** The message classes chosen determine which types of items receive the new form region. In this case, you will be extending the **IPM.Task** message class only.

Task 2 – Implementing the Form Region

In this second task, you will add controls to the form and implement the code behind necessary to make the form region functional.

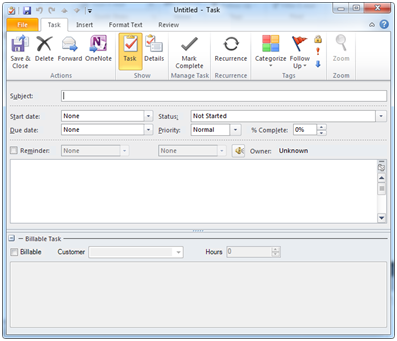
* 1. The **Billable Task** extensions pane will end-up looking as the following:
  2. 
  3. Figure 12
  4. Completed Custom Form Region
  5. Open the designer for **BillableTaskRegion.cs** (C#) or **BillableTaskRegion.vb** (VB) by right clicking it in the **Solution Explorer** and clicking **View Designer**.
  6. Right click the design surface and click **Properties**.
  7. In the **Properties** window, set the size of the canvas to **486, 150**.
  8. Using the toolbox, drag the following controls onto the canvas in the layout showed in Figure 10.
     1. 1 CheckBox
     2. 1 ComboBox
     3. 1 NumericUpDown
     4. 1 TextBox
     5. 2 Labels
  9. Set the following properties on the **CheckBox** controls.
     1. **(Name) & Text** – chkBillable & Billable
     2. **Location & Size** – 6, 6 & 59, 17
  10. Set the following properties on one of the **Label** controls.
      1. **Text** –Customer
      2. **Location & Size** – 87, 7 & 51, 13
  11. Set the following properties on the **ComboBox** control.
      1. **(Name)** – lstCustomer
      2. **Location & Size** – 143, 4 & 167, 21
      3. **Items** – Fabrikam Inc., Adventure Works
         1. **Note:** Enter each company on its own line
  12. Set the following properties on the remaining **Label** control.
      1. **Text** –Hours
      2. **Location & Size** – 347, 7 & 35, 13
  13. Set the following properties on the **NumericUpDown** control.
      1. **(Name)** – numHours
      2. **Location & Size** – 386, 4 & 95, 20
  14. Set the following properties on the **TextBox** control.
      1. **(Name)** - txtDetails
      2. **Multiline** - true
      3. **Location & Size** – 5, 31 & 476, 116
      4. **Anchor** – Top, Bottom, Left, Right
  15. Add the code behind that will implement the form’s functionality.
      1. Open the code behind for **BillableTaskRegion.cs** (C#) or **BillableTaskRegion.vb** (VB) by right clicking it in the **Solution Explorer** and clicking **View Code**.
      2. Add the following properties to the **BillableTaskRegion** class.
         1. *(Code Snippet – Office UI Customization – Outlook Properties – CSharp)*
         2. C#
         3. **private Outlook.TaskItem m\_taskItem;**
         4. **private Outlook.ItemProperty m\_isBillable;**
         5. **private Outlook.ItemProperty m\_customer;**
         6. **private Outlook.ItemProperty m\_hours;**
         7. **private Outlook.ItemProperty m\_details;**
         8. *(Code Snippet – Office UI Customization – Outlook Properties – VB)*
         9. Visual Basic
         10. **Private m\_taskItem As Outlook.TaskItem**
         11. **Private m\_isBillable As Outlook.ItemProperty**
         12. **Private m\_customer As Outlook.ItemProperty**
         13. **Private m\_hours As Outlook.ItemProperty**
         14. **Private m\_details As Outlook.ItemProperty**
      3. In the same class, add the **EnsureItemProperty** method to aid in creating custom properties on the task.
         1. *(Code Snippet – Office UI Customization – EnsureItemProperty Method– CSharp)*
         2. C#
         3. **private void EnsureItemProperty(ref Outlook.ItemProperty property, string name, Outlook.OlUserPropertyType propertyType)**
         4. **{**
         5. **if (property == null)**
         6. **{**
         7. **property = m\_taskItem.ItemProperties[name];**
         8. **if (property == null)**
         9. **property =**
         10. **m\_taskItem.ItemProperties.Add(name, propertyType);**
         11. **}**
         12. **}**
         13. *(Code Snippet – Office UI Customization – EnsureItemProperty Sub– VB)*
         14. Visual Basic
         15. **Private Sub EnsureItemProperty(ByRef prop As Outlook.ItemProperty, ByVal name As String, ByVal propertyType As Outlook.OlUserPropertyType)**
         16. **If prop Is Nothing Then**
         17. **prop = m\_taskItem.ItemProperties(name)**
         18. **If prop Is Nothing Then**
         19. **prop = m\_taskItem.ItemProperties.Add(name, propertyType)**
         20. **End If**
         21. **End If**
         22. **End Sub**
      4. Next to the previous code, add the **EnsureProperties** method to verify all of the custom task properties exists.
         1. *(Code Snippet – Office UI Customization – EnsureProperties Method– CSharp)*
         2. C#
         3. **private void EnsureProperties()**
         4. **{**
         5. **EnsureItemProperty(ref m\_isBillable, "Billable", Outlook.OlUserPropertyType.olYesNo);**
         6. **EnsureItemProperty(ref m\_customer, "Billable Customer", Outlook.OlUserPropertyType.olText);**
         7. **EnsureItemProperty(ref m\_hours, "Billable Hours", Outlook.OlUserPropertyType.olNumber);**
         8. **EnsureItemProperty(ref m\_details, "Billing Details", Outlook.OlUserPropertyType.olText);**
         9. **}**
         10. *(Code Snippet – Office UI Customization – EnsureProperties Sub– VB)*
         11. Visual Basic
         12. **Private Sub EnsureProperties()**
         13. **EnsureItemProperty(m\_isBillable, "Billable", Outlook.OlUserPropertyType.olYesNo)**
         14. **EnsureItemProperty(m\_customer, "Billable Customer", Outlook.OlUserPropertyType.olText)**
         15. **EnsureItemProperty(m\_hours, "Billable Hours", Outlook.OlUserPropertyType.olNumber)**
         16. **EnsureItemProperty(m\_details, "Billing Details", Outlook.OlUserPropertyType.olText)**
         17. **End Sub**
      5. Add the **UpdateEnableState** method, to update the state of the controls based on the state of the **Billable** checkbox.
         1. *(Code Snippet – Office UI Customization – UpdateEnableState Method– CSharp)*
         2. C#
         3. **public void UpdateEnableState()**
         4. **{**
         5. **lstCustomer.Enabled = chkBillable.Checked;**
         6. **numHours.Enabled = chkBillable.Checked;**
         7. **txtDetails.Enabled = chkBillable.Checked;**
         8. **}**
         9. *(Code Snippet – Office UI Customization – UpdateEnableState Method– VB)*
         10. Visual Basic
         11. **Public Sub UpdateEnableState()**
         12. **lstCustomer.Enabled = chkBillable.Checked**
         13. **numHours.Enabled = chkBillable.Checked**
         14. **txtDetails.Enabled = chkBillable.Checked**
         15. **End Sub**
      6. In the same class, implement **FormRegionShowing** method to load the current property values into the controls and setup the initial state of the form region.
         1. *(Code Snippet – Office UI Customization – FormRegionShowing Implementation – CSharp)*
         2. C#
         3. **m\_taskItem = this.OutlookItem as Outlook.TaskItem;**
         4. **EnsureProperties();**
         5. **chkBillable.Checked = m\_isBillable.Value;**
         6. **UpdateEnableState();**
         7. **lstCustomer.SelectedText = m\_customer.Value;**
         8. **numHours.Value = (decimal)m\_hours.Value;**
         9. **txtDetails.Text = m\_details.Value;**
         10. *(Code Snippet – Office UI Customization – FormRegionShowing Implementation – VB)*
         11. Visual Basic
         12. **m\_taskItem = TryCast(Me.OutlookItem, Outlook.TaskItem)**
         13. **EnsureProperties()**
         14. **chkBillable.Checked = m\_isBillable.Value**
         15. **UpdateEnableState()**
         16. **lstCustomer.SelectedText = m\_customer.Value**
         17. **numHours.Value = CDec(m\_hours.Value)**
         18. **txtDetails.Text = m\_details.Value**
  16. In the same **BillableTaskRegion** class, add code behind that will write the contents of the controls as they change.
      1. Create a method named **chkBillable\_CheckedChanged** to handle the **Billable** check box’s checked event.
         1. C#
         2. **private void chkBillable\_CheckedChanged(object sender, EventArgs e)**
         3. **{**
         4. **m\_isBillable.Value = chkBillable.Checked;**
         5. **UpdateEnableState();**
         6. **}**
         7. Visual Basic
         8. **Private Sub chkBillable\_CheckedChanged(ByVal sender As Object, ByVal e As EventArgs) Handles chkBillable.CheckedChanged**
         9. **m\_isBillable.Value = chkBillable.Checked**
         10. **UpdateEnableState()**
         11. **End Sub**
      2. Create a method named **lstCustomer\_TextChanged** to handle the **Customer** dropdown control’s changed event
         1. C#
         2. **private void lstCustomer\_TextChanged(object sender, EventArgs e)**
         3. **{**
         4. **m\_customer.Value = lstCustomer.Text;**
         5. **}**
         6. Visual Basic
         7. **Private Sub lstCustomer\_TextChanged(ByVal sender As Object, ByVal e As EventArgs) Handles lstCustomer.TextChanged**
         8. **m\_customer.Value = lstCustomer.Text**
         9. **End Sub**
      3. Create a method named **numHours\_ValueChanged** to handle changes in the **Hours** control.
         1. C#
         2. **private void numHours\_ValueChanged(object sender, EventArgs e)**
         3. **{**
         4. **m\_hours.Value = (double)numHours.Value;**
         5. **}**
         6. Visual Basic
         7. **Private Sub numHours\_ValueChanged(ByVal sender As Object, ByVal e As EventArgs) Handles numHours.ValueChanged**
         8. **m\_hours.Value = CDec(numHours.Value)**
         9. **End Sub**
      4. Create a method named **txtDetails\_TextChanged** to handle changes to the **Details** control
         1. C#
         2. **private void txtDetails\_TextChanged(object sender, EventArgs e)**
         3. **{**
         4. **m\_details.Value = txtDetails.Text;**
         5. **}**
         6. Visual Basic
         7. **Private Sub txtDetails\_TextChanged(ByVal sender As Object, ByVal e As EventArgs) Handles txtDetails.TextChanged**
         8. **m\_details.Value = txtDetails.Text**
         9. **End Sub**
  17. Connect the code behind to the controls using the designer.
      1. **Note:** This step is only needed in C#.
      2. Open the designer for **BillableTaskRegion.cs** (C#) by right clicking it in the **Solution Explorer** and clicking **View Designer**.
      3. Right click the **Billable** check box control and click **Properties**.
      4. In the **Properties** window, click the **Event** icon (the lightning bolt.)
      5. Set the **CheckedChanged** event to **chkBillable\_CheckedChanged** using the dropdown list.
         1. 
         2. Figure 13
         3. Control Events
      6. Repeat the above process to attach the **Customer** control’s **TextChanged** event to the **lstCustomer\_TextChanged** method.
      7. Repeat the above process to attach the **Hours** control’s **ValueChanged** event to the **numHours\_ValueChanged** method.
      8. Repeat the above process to attach the **Details** control’s **TextChanged** event to the **txtDetails\_TextChanged** method.

Exercise 3 Verification

* 1. In order to verify that you have correctly performed all steps in the above exercise, proceed as follows:

#### Test the Add-in

Test your add-in to confirm that the export ribbon and backstage buttons work as expected.

* 1. Run the add-in and verify the enable checkboxes work.
     1. In the **Debug** menu, click **Start Without Debugging**.
     2. Once **Outlook 2010** loads, switch to the **Home** ribbon tab.
     3. Click the **New Items -> Task** option to create a new task.
     4. Verify the **Billable Task** form region is visible.
        1. 
        2. Figure 14
        3. Custom Task Form Region
     5. Enter some data into the new task and save it.
     6. Switch to the **Tasks** section using the buttons in the lower left and open the task you just created.
     7. Verify the information in the **Billable Task** region is loading correctly.
  2. When you are done cleanup and remove the add-in.
     1. Close **Word 2010**.
     2. In the **Solution Explorer**, right-click **OutlookFormRegion** and click **Clean**.

Summary

* 1. In this exercise, you learned how to extend Office application using custom ribbons, backstage tabs, task panes, and Outlook form regions. Each of these tools are used to solve a different problem. Ribbons provide functionality while editing while Backstage provides tools that work on the state of the document, no necessarily the content.