SOLUTIONS FOR REPURPOSING THE DEFAULT ACTIONS AND STATES OF THE OFFICE CONTROLS THROUGH COMPONENT OBJECT MODEL ADD-INS

Alexandru PÎRJAN[1]

ABSTRACT:

In this paper, we have approached a high interest research topic regarding the development of customized Office solutions by designing and implementing a Component Object Model Add-In (COM Add-In) for repurposing the controls' default actions and states and for extending the features offered by the Office suite. In the introduction, we have presented a brief overview of Visual Studio Tools for Office (VSTO) and its evolution. In the second section, we have developed a Word 2013 Add-in using VSTO 2013 and repurposed the default actions of the commands in Word 2013 using VSTO 2013. In the third section, we have developed a solution for extending the default features of the Office 2013 suite by programing in VSTO 2013 an extension to the existing Ribbon bar. The developed extension implements a new customized tab-menu, a customized group containing a customized menu and a toggleButton element. The customized menu contains three checkBoxes for controlling dynamically the state of the Bold, Italic and Underline controls, while the toggleButton dynamically controls the visibility of the existing Tab elements.

Keywords: VSTO 2013, .NET Framework, C#, COM Add-ins, Ribbon (XML).

1. Introduction

The first version of Visual Studio .NET and of the .NET Framework were released in 2002. Before this version of Visual Studio came out, the Office applications had used solely the Visual Basic for Applications (VBA) development environment. The Office applications exposed a complex object model that could be accessed through a Component Object Model (COM) technology. Visual Basic for Applications and the Office Component Object Model were employed by millions of developers to achieve the automation of certain repetitive tasks and the development of complex Office solutions that made use of the user-friendly interface and the extensive features of the Office suite [1], [2], [3], [4].

The developers were able to build office solutions that leveraged the existing features that were already implemented in the Office suite of applications, thus leading to an increased efficiency and reduced costs. The limitations of the VBA development environment and the VBA embedded code in each of the customized documents required a great amount of effort in order to fix the identified bugs and update the developed solutions once the error was propagated across the enterprise's documents. The security vulnerabilities in the VBA model led to a plague of worms and macro viruses that ultimately forced enterprises to drop VBA.

[1] Lecturer, PhD. Faculty of Computer Science for Business Management, Romanian-American University, 1B, Expozitiei Blvd., district 1, code 012101, Bucharest, Romania. Email: alex@pirjan.com

Visual Studio .NET along with the .NET Framework offered an opportunity to solve all these problems by unifying the profuseness of the framework, of the developer tools with the spectacular opportunities offered by the Office platform. This is how VSTO emerged on the technical stage of developing complex Office Solutions for the enterprise. Although being in an incipient stage, the first version of VSTO allowed developers to write code behind Word 2003 and Excel 2003 documents and templates in C# and Visual Basic. Instead of embedding the code directly into a document, VSTO linked the document with a .NET assembly and brought into use for the first time a new security model for Office solutions that exerted a new code access security policy in order to put a stop to macro viruses and malicious code.

The second iteration of VSTO brought even new features, enhancing the developer's technical possibilities by offering support for data biding, data/view separation, design time support for Word and Excel documents inside Visual Studio, enhanced support for Windows Forms controls, custom task panes for Office applications, server-side programming support [1]. VSTO 2005 offered many tools for the advanced developer but in the same time, it offered the possibility to develop and alter complex parts of the Office applications with minimum effort.

VSTO 2007, the third iteration of VSTO, was incorporated as a core feature of Visual Studio 2008, making possible to develop COM Add-ins for all the major applications of the Office suite. VSTO 2007 offered support to build application-level custom task panes, to customize the new Office Ribbon, to alter Outlook's UI by using Forms Regions and many more novel features [1].

The latest release of Visual Studio Tool for Office 2013 offers design-time support for Office 2013 without the need for a separate download. VSTO 2013 offers support for the .NET Framework 4.5.1 thus offering increased performance and versatility along with an improved debugging process. The Visual Studio 2013 Update 3 offers support for the .NET Framework 4.5.2 and the possibility to mix and match different versions of Office with different versions of the .NET Framework. The update also offers better design time support for the process of code signing the certificates using the SHA256 hash algorithm. The latest update offers easier unit testing for VSTO projects. The developer has the possibility to create a Unit Test VSTO project and reference it directly, without having to copy the code in a separate class library project[1].

The developer has the possibility to customize and extend the functionality of Office applications by developing solutions that address their complex object model. A major advantage when developing a solution targeting the Office system consists in the possibility of being able to reuse components of the most widely known and feature offering applications. For example, an Office solution that handles documents has the possibility to reuse a wide class of accessible popular applications, offering a variety of features: generating, formatting and printing documents by using the Word 2013 features; analyzing or displaying data by using the formatting, charting, computation and data analysis features offered by Excel 2013. VSTO brings a great advantage to the developer by offering him the possibility to reuse parts of the applications that he already knows, instead of developing new ones from the ground up.

[1] *http://blogs.msdn.com/b/vsto/archive/2014/08/04/visual-studio-2013-update-3-released.aspx, accessed on 13.12.2014, at 23:00.*

Many users address the Office suite environment daily. A custom developed Office solution extends the suite's default functionality, by complementing and becoming a component integrant part in the existing Office application. Users regularly need to copy data from another application to an Office application, like a Word document or Excel workbook. VSTO 2013 makes it possible to fetch the desired information without having to copy manually data from another application.

Office Business Applications (OBAs) are custom developed applications that drive business data into Office and Share-Point. The evolving of the Office platform and of the VSTO 2013 facilitates the integration of business data into the Office environment. Regarding this matter, the Business Data Catalog in SharePoint (BDC) can drive custom business data into SharePoint. VSTO 2013 allows to use the data binding concept for driving business data in the most important applications of Office 2013.

Before VSTO, Visual Basic for Applications (VBA) and the macro recording features available in most of the Office applications was the sole approach for developing Office solutions. One of the most common practices for automating certain repetitive actions within an Office application consists in recording macro commands. In contrast with this approach, VSTO 2013 offers professional developer tools, making it possible to scale Office solutions to the whole enterprise, to update an Office solution once it has been deployed, to use advanced programming languages like C#, to have support for team development (source code control).

The technical advantages of the .NET Framework for Office solve most of the problems that professional programmers faced in the process of developing Office solutions. Visual Studio 2013 makes it more efficiently to develop and implement complex Office solutions, tackling complex matters like cryptography, data mining, mobile communications, etc. as it provides in this matter an ample programming environment to the developers [5]. They can make use of the .NET languages, C# or Visual Basic when developing Office solutions. The .NET code can call the unmanaged object models exposed by the Office applications through the Office Primary Interop Assemblies (PIAs). Technologies like Windows Presentation Foundation (WPF) that facilitates the designing and modeling the Graphical User Interface (GUI) and the Windows Communication Foundation (WCF) that allows the transferring of data from the corporates' servers can be used in developing Office solutions.

Taking into account all of the above undisputable advantages, we have decided to approach and present in this paper an interesting topic, related to the VSTO features presented above: the development of an innovative, customized Office solution that implements a Component Object Model Add-In (COM Add-In), designed for repurposing the default commands in Word 2013 and for extending the default features offered by the Office suite.

The paper has the following structure: in the second section, we have developed a Word 2013 Add-in using VSTO 2013 and repurposed the default actions of the commands in Word 2013 using VSTO 2013. In the third section, we have developed a solution for extending the default features of the Office 2013 suite by programing in VSTO 2013 an extension to the existing *Ribbon* bar. The developed extension contains a new customized tab-menu, a customized group containing a customized menu and a *toggleButton* element. The customized menu contains three *checkBoxes* for controlling dynamically the state of the *Bold, Italic* and *Underline* controls. Thus, we have developed and presented a solution for repurposing the *toggleButton* and *gallery* type elements, aspects regarding the dynamical control of the visibility for the existing *Tab* elements, the advantages and limitations of the repurposing techniques in Office Solutions.

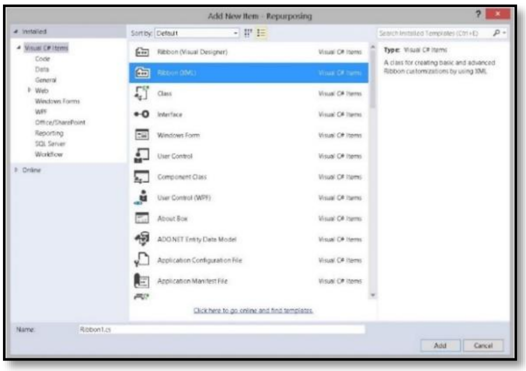
2. Repurposing the default actions of the commands in Word 2013 using Visual Studio Tools for Office 2013

VSTO 2013 offers the possibility to extend the default features of the Office suite by developing customized solutions suited for each activity. VSTO 2013 offers the following template types for developing Office solutions: Component Object Model (COM Add-ins) at the application level; Programmatic control and customization at the document level.

VSTO offers the possibility to execute source code only when a certain document is opened or every time an application runs. In order to extend the functionality of the existing *Ribbon* bar or to repurpose the built-in commands we must take into account the controls that the Office Ribbon provides, along with their object and event model. All the *Ribbon* controls have a set of common properties that is complemented by specific properties corresponding to the particularities of each control. The set of common properties for all the *Ribbon* controls consists in [1]:

- *Enabled* property that is of type *bool* and allows to enable or disable a control
- *Id* property that is of type *string* and represents the unique identifier of a control
- *Name* property that is of type *string* and that represents the name of the control
- *Tag* property that is of type *object* and offers the possibility to specify custom data associated with the control at runtime.

In order to develop our solution we have launched Microsoft Visual Studio Ultimate 2013 and we have created a new Word 2013 Add-in project, entitled *Repurposing*. Within this project we have added a *Ribbon (XML)* item to our existing project, entitled *Ribbon1* (**Fig. 1**). This item contains two elements, *Ribbon1.xml* (containing the xml code that defines the elements of the *Ribbon* bar extension) and *Ribbon1.cs* (containing the C# code corresponding to the elements defined within *Ribbon1.xml*).



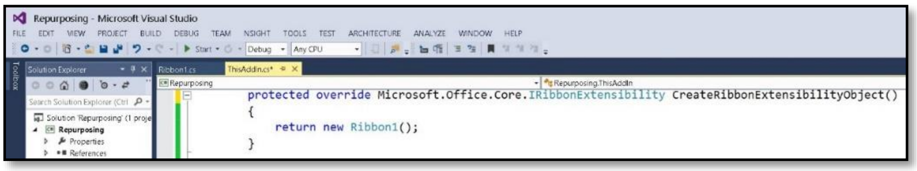
Type: Visual C# Items

A class for creating basic and advanced Ribbon customizations by using XML

Name: Ribbon1.cs

Fig. 1. Adding a new Ribbon (XML) element

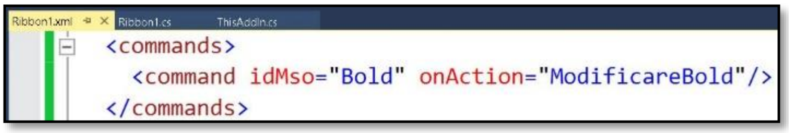
In order to incorporate the new *Ribbon* extension that we have just created into the existing Word application *Ribbon* bar, we have accessed the file *ThisAddIn.cs* and we have overridden the *CreateRibbonExtensibilityObject()* function, that returns a new instance of an *IRibbonExtensibility* object (**Fig. 2**).


Repurposing - Microsoft Visual Studio
FILE EDIT VIEW PROJECT BUILD DEBUG TEAM NSIGHT TOOLS TEST ARCHITECTURE ANALYZE WINDOW HELP
Solution Explorer
Ribbon1.cs
ThisAddIn.cs*
Repurposing
Repurposing.ThisAddIn
Solution 'Repurposing' (1 proje
Repurposing
Properties
References

```
protected override Microsoft.Office.Core.IRibbonExtensibility CreateRibbonExtensibilityObject()
{
    return new Ribbon1();
}
```

Fig. 2. Overriding the CreateRibbonExtensibilityObject() function

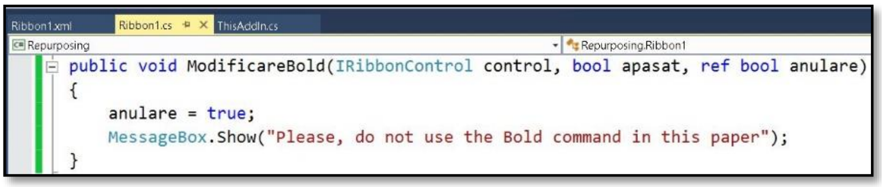
From this moment on, the control elements that we define in the *Ribbon1.xml* file and their corresponding programmatic actions created within the *Ribbon1.cs* file become available to the Office application that loads the corresponding Component Object Model Add-In. In order to repurpose one of Word 2013 build-in commands, we have accessed within *Ribbon1.xml* the commands collection and we have referred our desired command (in this case, the *Bold* control) using the built-in identifiers of the Word application provided in the Official VSTO 2013 documentation. Before the <*ribbon*> xml element, we have accessed the commands collection using the xml element <*commands*>. Within this element, we have defined an xml element command, we have used the *idMso* attribute to specify the built-in id of the control we want to repurpose. Using the on-action attribute, we have delegated the name of the function that will be triggered when the button is accessed (**Fig. 3**).



```
<commands>
  <command idMso="Bold" onAction="ModificareBold"/>
</commands>
```

Fig. 3. Repurposing the Bold command in Ribbon1.xml

We have taken into account the fact that the *Bold* command is associated to a *toggleButton* type control. When repurposing a *toggleButton* control in VSTO 2013, one must take into account the signature of the repurposing function. Thus, our *ModificareBold* function accepts three arguments: the first is a *IRibbonControl* object that represents the control that triggered the command, the second is a boolean variable representing the state of the *toggleButton* (if the button is pressed or not) and the third one is also a boolean variable passed by reference that specifies whether to cancel or not the default action of the respective command (**Fig. 4**).



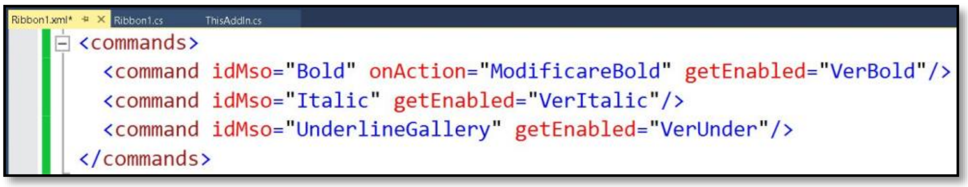
```
public void ModificareBold(IRibbonControl control, bool apasat, ref bool anulare)
{
    anulare = true;
    MessageBox.Show("Please, do not use the Bold command in this paper");
}
```

Fig. 4. The ModificareBold function that repurposes the Bold command

We have repurposed the *Bold* command so that when the Bold *toggleButton* is pressed, the default action is no longer executed and the user receives a message box with the following message: *Please, do not use the Bold command in this paper*.
In the following section, we create a new tab, entitled *Repurpose*, containing a group entitled *Optiuni de Dezactivare si Formatare*. This group contains a control of type menu entitled *Dezactivare* and within it, three *checkBox* controls entitled *Bold, Italic, Underline.* Each of these *checkBox* controls dynamically disables the corresponding control when checked and enables it back when unchecked. The group contains also a *toggleButton* control entitled *Dezactivare tab-ul HOME.* When the *toggleButton* is pressed, the Home Tab is no longer visible and when the *toggleButton* is depressed, the Home Tab becomes visible.

3. Repurposing the default state of the controls in Word 2013 using Visual Studio Tools for Office 2013

In order to achieve the repurposing the controls' default state in Word 2013 using VSTO 2013, we have accessed the *Ribbon1.xml* file and we have added to our previously repurposed *Bold* command the xml attribute *getEnabled="VerBold"* through which we have specified the function *VerBold* that will determine if the control is enabled or not by returning a boolean variable that will be used by the Office environment to set the state of the control.
We have added another command xml element with the attribute *idMso="Italic"* in order to access the built in *Italic* command. We have used the xml attribute *getEnabled="VerItalic"* in order to delegate the function *VerItalic* to return the state of the control (enabled or disabled) through a boolean variable.
In the same manner, we have addressed the *Underline* command, by using the built in identifier *idMso="UnderlineGallery"* and the xml attribute *getEnabled="VerUnder"*, in order to designate the function that will return the state of the control.
We have taken into account the fact that *Bold* and *Italic* are *toggleButton* controls while the *Underline* is implemented as a *gallery* control. One should note that VSTO 2013 does not allow the repurposing of the built in actions of *gallery* controls. Thus, the previously analyzed example regarding the *Bold* command can be applied also to the *Italic* command, but not to the *Underline* command. However, VSTO 2013 allows us to enable or disable these controls even if they are *toggleButton* controls or *gallery* controls (**Fig. 5**).

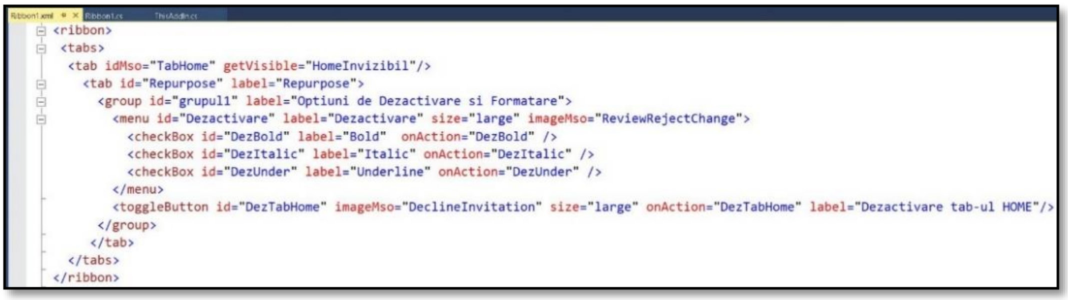


```
<commands>
  <command idMso="Bold" onAction="ModificareBold" getEnabled="VerBold"/>
  <command idMso="Italic" getEnabled="VerItalic"/>
  <command idMso="UnderlineGallery" getEnabled="VerUnder"/>
</commands>
```

Fig. 5. Accessing the built in Bold, Italic, Underline commands using xml code

In order to have access to the built in Home Tab and to further extend the ribbon with our customized *Repurpose* tab along with the customized group *Optiuni de Dezactivare si*

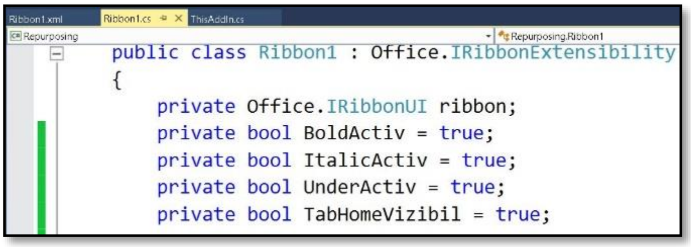
Formatare containing the *Dezactivare* menu and the *toggleButton* for modifying the visibility of the built-in Home Tab we have accessed the file *Ribbon1.xml*.
We have added a <*tab*> xml element and used the *idMso="TabHome"* identifier to refer the built in Home Tab. We have used the xml attribute *getVisible="HomeInvizibil"* in order to specify the function *HomeInvizibil* that will set the visibility of the built-in tab by returning a boolean variable.
We have used a <*tab*> xml element to create our own custom tab, in this situation we had to use the identifier *id="Repurpose"* to set a unique name for our tab's identifier. One must note that in this situation we have used solely the attribute *id* instead of *idMso* as this tab is newly created and does not exist in the built-in Office tab collection. By using the xml code *label=" Repurpose"* we have specified the title of the tab, title that will be visible to the user in the *Ribbon*. Within this tab we have used the xml element <*group*> with the identifier *id="grupul1"* and associated label *Optiuni de Dezactivare si Formatare* in order to create our custom group.
Inside this group, we have created a menu by using the xml element <*menu*> having the identifier *id="Dezactivare"*. We wanted this menu to have a large size as to accommodate three *checkBoxes* so we have used the xml attribute *size="large"*. VSTO 2013 offers the possibility to set the desired icon for controls either by using a built-in icon identifier or by creating a customized icon.
For the *Dezactivare* menu we have decided to use a built-in icon, so we added the xml attribute *imageMso="ReviewRejectChange*". Within the xml element <*menu*>, we have developed three *checkBoxes* by using the xml element <*checkBox*> along with unique identifiers for each of the *checkBoxes*: *id="DezBold", id="DezItalic", id="DezUnder"*. We have used suggestive labels for each of the *checkBoxes* (*label="Bold", label="Italic", label="Underline"*) and by using the *onAction* xml attribute we have designated three functions that will be called when the *checkBoxes* are checked or unchecked.
Besides the *Dezactivare* menu, we have developed a *toggleButton* and assigned it the unique identifier *id="DezTabHome"* along with a built-in icon by using the attribute *imageMso="DeclineInvitation"*. We have set the size of the *toggleButton* large by using the attribute size and we have designated the function *DezTabHome* to be triggered when the button is pressed or depressed using the *onAction* xml attribute. The label of the *toggleButton* that will be displayed to the user is *Dezactivare tab-ul HOME* (**Fig. 6**).



```
<ribbon>
 <tabs>
  <tab idMso="TabHome" getVisible="HomeInvizibil"/>
    <tab id="Repurpose" label="Repurpose">
      <group id="grupul1" label="Optiuni de Dezactivare si Formatare">
        <menu id="Dezactivare" label="Dezactivare" size="large" imageMso="ReviewRejectChange">
          <checkBox id="DezBold" label="Bold"  onAction="DezBold" />
          <checkBox id="DezItalic" label="Italic" onAction="DezItalic" />
          <checkBox id="DezUnder" label="Underline" onAction="DezUnder" />
        </menu>
        <toggleButton id="DezTabHome" imageMso="DeclineInvitation" size="large" onAction="DezTabHome" label="Dezactivare tab-ul HOME"/>
      </group>
     </tab>
  </tabs>
</ribbon>
```

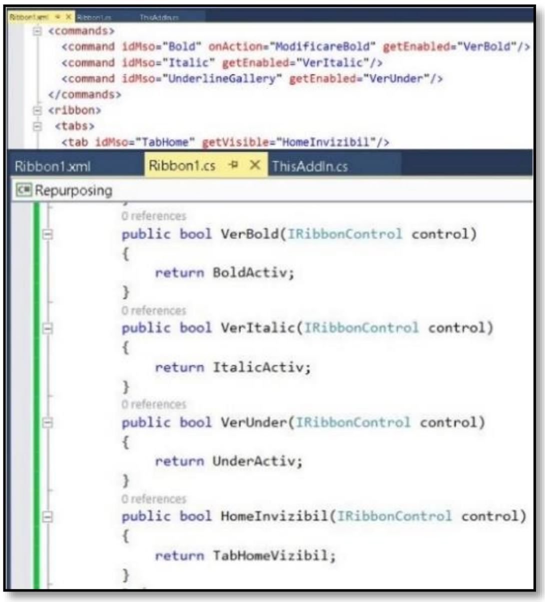
Fig. 6. The xml code for defining the user interface and for designating the functions

After having developed the interface and defined the functions that will be called when the newly developed controls are used, we have accessed the *Ribbon1.cs* file and have created the functions along with the programmatic actions that they must perform. We have created at the global level of the *Ribbon1* class four variables that represent the state of the *Bold, Italic, Underline* and *Home Tab* controls. We want these elements to be enabled by default, when the Component Module Add-in is loaded, so we set them with the *true* value (**Fig. 7**).



```
public class Ribbon1 : Office.IRibbonExtensibility
{
    private Office.IRibbonUI ribbon;
    private bool BoldActiv = true;
    private bool ItalicActiv = true;
    private bool UnderActiv = true;
    private bool TabHomeVizibil = true;
```

Fig. 7. The four boolean variables that represent the state of the controls



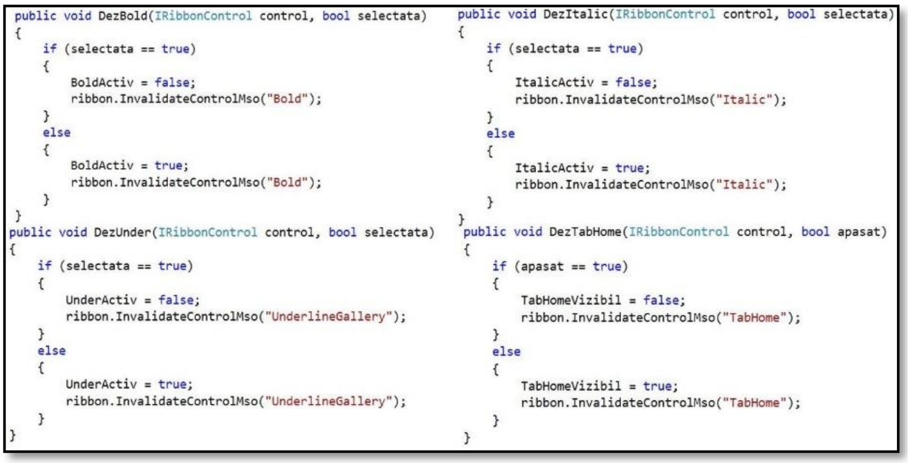
```
<commands>
  <command idMso="Bold" onAction="ModificareBold" getEnabled="VerBold"/>
  <command idMso="Italic" getEnabled="VerItalic"/>
  <command idMso="UnderlineGallery" getEnabled="VerUnder"/>
</commands>
<ribbon>
 <tabs>
  <tab idMso="TabHome" getVisible="HomeInvizibil"/>
```

```
public bool VerBold(IRibbonControl control)
{
    return BoldActiv;
}
public bool VerItalic(IRibbonControl control)
{
    return ItalicActiv;
}
public bool VerUnder(IRibbonControl control)
{
    return UnderActiv;
}
public bool HomeInvizibil(IRibbonControl control)
{
    return TabHomeVizibil;
}
```

Fig. 8. The VerBold, VerItalic, VerUnder and HomeInvizibil functions

We have developed the *VerBold, VerItalic, VerUnder* and *HomeInvizibil* functions. These functions take as an argument an object of *IRibbonControl* type and return a boolean variable thus determining the state of the control that triggered them. In our project, *VerBold* determines the state of the *Bold* control, *VerItalic* sets the state of the *Italic* control, *VerUnder* specifies the state of the *Underline* control and *TabHomeVizibil* determines the state of the built-in Home *Tab*. The functions return the values of the previously created boolean variables: *BoldActiv, ItalicActiv, UnderActiv* and *TabHomeVizibil*. By default when the Component Object Module Add-in is loaded all these Office controls will be active as the functions that determine their state return the *true* value (**Fig. 8**).
In this point, we had to come up with a solution as to be able to change dynamically the state of these controls, not to be restricted only to the moment when the module loads. We

have developed the previously defined functions *DezBold*, *DezItalic*, *DezUnder* and *DezTabHome* in the *Ribbon1.cs* file.
The functions take two arguments: an object of *IRibbonControl* type and a boolean variable. We have developed the functions as to set the corresponding value of *true* or *false* to the state variables according to the state of the *checkBoxes* and of the *toggleButton*. For example, in the case of the *DezBold* function, if the *checkBox* that triggered the function is checked, we set the state variable *BoldActiv* to *false* and otherwise, if the *checkBox* is not checked, we set the variable *BoldActiv* to *true*. After the controls have been loaded, Office stores the state of the controls within the cache. This is the reason why we had to invalidate the controls using the built in identifiers *Bold*, *Italic*, *UnderlineGallery* and *TabHome*. By invalidating these controls, we instructed Office to delete their state from the cache and to recheck their state. Thus, the functions *VerBold*, *VerItalic*, *VerUnder* and *HomeInvizibil* have been retriggered and returned now the new state of the controls (**Fig. 9**).



```
public void DezBold(IRibbonControl control, bool selectata)
{
    if (selectata == true)
    {
        BoldActiv = false;
        ribbon.InvalidateControlMso("Bold");
    }
    else
    {
        BoldActiv = true;
        ribbon.InvalidateControlMso("Bold");
    }
}
```

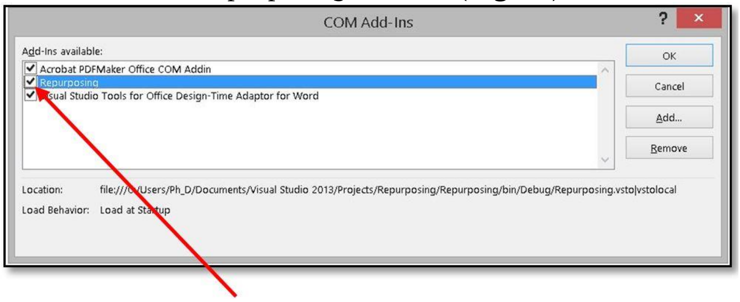
```
public void DezItalic(IRibbonControl control, bool selectata)
{
    if (selectata == true)
    {
        ItalicActiv = false;
        ribbon.InvalidateControlMso("Italic");
    }
    else
    {
        ItalicActiv = true;
        ribbon.InvalidateControlMso("Italic");
    }
}
```

```
public void DezUnder(IRibbonControl control, bool selectata)
{
    if (selectata == true)
    {
        UnderActiv = false;
        ribbon.InvalidateControlMso("UnderlineGallery");
    }
    else
    {
        UnderActiv = true;
        ribbon.InvalidateControlMso("UnderlineGallery");
    }
}
```

```
public void DezTabHome(IRibbonControl control, bool apasat)
{
    if (apasat == true)
    {
        TabHomeVizibil = false;
        ribbon.InvalidateControlMso("TabHome");
    }
    else
    {
        TabHomeVizibil = true;
        ribbon.InvalidateControlMso("TabHome");
    }
}
```

Fig. 9. The DezBold, DezItalic, DezUnder and DezTabHome functions that invalidate the necessary controls

The user has the possibility to load or unload our *Repurposing* Component Object Model Add-in in Word 2013 by accessing the COM Add-Ins menu from the Add-Ins group within the Developer tab. In the Add-Ins available list within the COM Add-Ins menu the user can check or uncheck our *Repurposing* solution (**Fig. 10**).



COM Add-Ins

Add-Ins available:

- [✓] Acrobat PDFMaker Office COM Addin
- [✓] Repurposing
- [✓] Visual Studio Tools for Office Design-Time Adaptor for Word

Location: file:///C:/Users/Ph_D/Documents/Visual Studio 2013/Projects/Repurposing/Repurposing/bin/Debug/Repurposing.vsto|vstolocal

Load Behavior: Load at Startup

Fig. 10. The Repurposing solution loaded within the Add-Ins list

After the repurposing COM Add-In has been loaded, the Office ribbon bar is extended with a new customized tab-menu entitled *Repurpose*, containing a customized group entitled *Optiuni de Dezactivare si Formatare* that includes:

1. a *menu* control entitled *Dezactivare* comprising of three *checkBoxes* entitled *Bold, Italic* and *Underline* that dynamically enable or disable the corresponding *Bold*, *Italic* and *Underline* controls from the Font group within the Home Tab (**Fig. 11**).

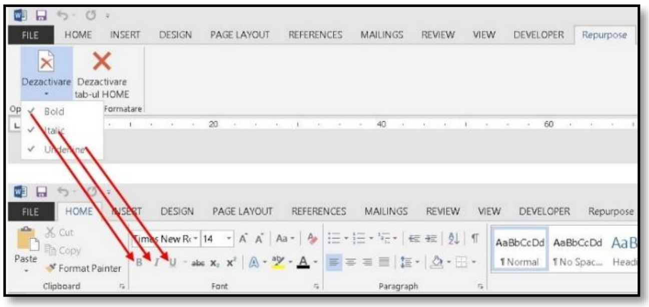


Fig. 11. An example of disabling the Bold, Italic and Underline controls using the developed Repurposing solution

2. a *toggleButton* control entitled *Dezactivare tab-ul HOME* that overrides the visibility of the Home Tab even if the user selects it to be visible in the Word Options menu (**Fig. 12**).

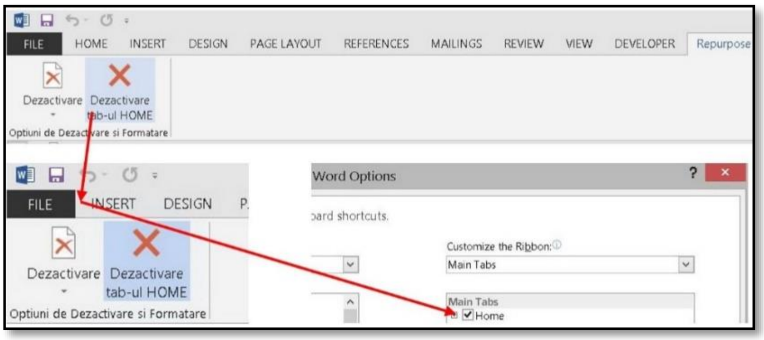


Fig. 12. Overriding the visibility of the Home Tab even if the user selects it to be visible in the Word Options menu

In the same manner, one can modify, using VSTO 2013, the default state and features of the controls within the Office Ribbons or an entire new customized *Ribbon* can even be built from scratch according to the enterprises' Office solutions needs.

4. Conclusions

VSTO 2013 offers the necessary technical means for evolving the Office suite and helps overcoming many of its limitations. From the developer's perspective, VSTO 2013 offers endless possibilities for developing new features and extensions for the Office suite, aspects that seemed to be out of reach until now from the technical and economical point of view.

By analysing the developed solution, repurposing the default actions of the commands and repurposing the default state of the controls in Word 2013 using VSTO 2013, one can conclude that VSTO 2013 has brought significant improvements to the previous versions representing a powerful and useful tool in developing customized Office solutions.

5. Acknowledgement

This paper presents a series of results obtained within the project "Practical School: Innovation in Higher Education and Success on the Labour Market", POSDRU/156/1.2/G/132920, implemented by the Romanian-American University in partnership with the Center for Legal Resources. The project is co-financed from the European Social Fund through the Sectorial Operational Programme Human Resource Development 2007-2013.

REFERENCES

[1] Carter Eric, Lippert Eric, Visual Studio Tools for Office 2007: VSTO for Excel, Word, and Outlook, Publisher: Addison-Wesley Professional, 2009, ISBN-10: 0321533216, ISBN-13: 978-0321533210

[2] Bruney Alvin, Professional VSTO 2005: Visual Studio 2005 Tools for Office, Publisher: Wrox, 2006, ISBN-10: 0471788139, ISBN-13: 978-0471788133

[3] Sempf Bill, Jausovec Peter, VSTO For Dummies, Publisher: For Dummies, 2010, ISBN-10: 0470046473, ISBN-13: 978-0470046470

[4] Anderson Ty, Pro Office 2007 Development with VSTO, Publisher: Apress, 2008, ISBN-10: 1430210729, ISBN-13: 978-1430210726

[5] Tăbușcă A., A new security solution implemented by the use of the multilayered structural data sectors switching algorithm (MSDSSA), Journal of Information Systems & Operations Management, vol.4, no.2, Ed. Universitară, 2010, ISSN 1843-4711