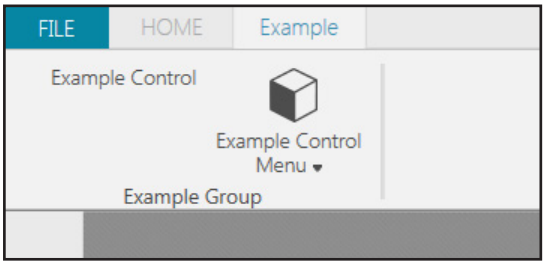
Support
support@visualcomponents.com

Community
community.visualcomponents.net

Adding new ribbon tabs

Next Generation | Version: November 20, 2015 | Example: Available upon request

The ribbon in Essentials can be extended to have additional tabs containing groups of customizable controls.



The setup for adding a ribbon tab consists of three stages: tab, ribbon group, and control.

Tab

In the first stage, you define a subclass of the RibbonTabBase class that is exported as a type of IRibbonTab interface and has a unique Id property that can be referenced by tab groups.

Ribbon Group

In the second stage, you define a subclass of the RibbonGroupBase class that is exported as a type of IRibbonGroup interface. RibbonGroupMetadata is used to associate a ribbon group with a tab. Each ribbon group has a unique Id property that contains a substring of Ribbon. That Id is added to the Essentials application configuration file in order to add both a ribbon group and an associated tab.

Control

In the final stage, you define a control as an action item for executing one or more commands. The control itself is a subclass of the ActionItem class and exported as a type of IActionItem interface. Each control has a unique Id that can be added to any tab group in the Essentials application configuration file.

The topics covered in this tutorial include:

- Minimal setup for adding a tab of controls to the ribbon in Essentials.
- Using the IEventAggregator, ISelectionManager and IRobot to activate controls based on events, selections and types of components.
- Customizing menu control items and adjusting the camera and view of the 3D world.

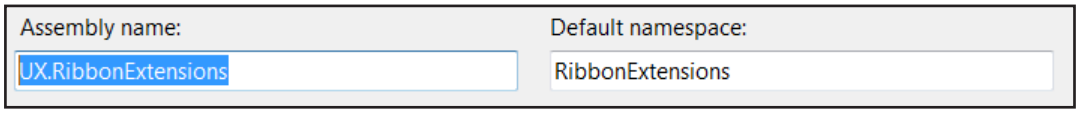
Creating a project in Visual Studio

Visual Studio 2013 and Visual C# are used to create a Class Library project that defines a tab of controls.

1. Run **Visual Studio** as an **administrator**.

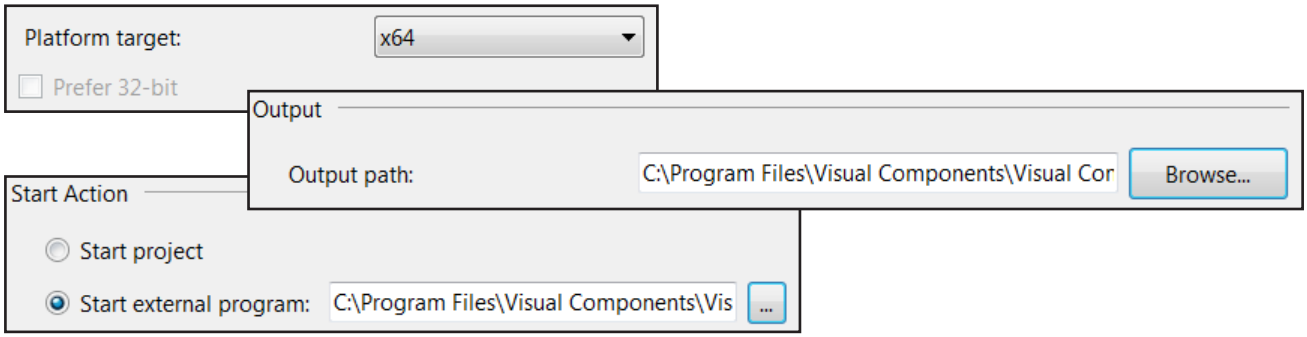
2. Create a new **Class Library** project, and then access the properties of your project.

3. In the **Application** tab, set **Assembly name** to have a prefix of **UX.** in order to id your assembly as a part of the VisualComponents.UX namespace.



4. In the **Build** tab, set **Platform target** to either **x86** or **x64** and then set **Output path** to be the path to your Essentials program files in order for your assembly to be discovered by the MEF and loaded at runtime.

5. In the **Debug** tab, set **Start external program** to be execute your version of Essentials.



6. In your project, add references to the following assemblies and namespaces:

   - **Caliburn.Micro** that is available in your Essentials program files.
   - **Create3D.Shared** that is available in your Essentials program files.
   - **PresentationCore** and **PresentationFramework** that are available in the .NET Framework.
   - **System.ComponentModel.Composition** that is available in the .NET Framework.
   - **UX.Ribbon** that is available in your Essentials program files.
   - **UX.Shared** that is available in your Essentials program files.
   - **WindowsBase** that is available in the .NET Framework.

Creating a ribbon tab

A ribbon tab is a subclass of RibbonTabBase that uses a base class constructor and implements inherited member properties. The ribbon tab itself is exported as a type of IRibbon.

1. In your project, add a new **Class** item and set the class to be **internal**.

2. Add using statements for:

   - **Caliburn.Micro**
   - **System.ComponentModel.Composition**
   - **VisualComponents.Create3D**
   - **VisualComponents.UX.Ribbon**
   - **VisualComponents.UX.Shared**

3. Export your class as a type of **IRibbonTab** and use a **Shared** creation policy to handle requests within a composition container.

4. Inherit **RibbonTabBase** as a base class, and then use an **ImportingConstructor** to define the base class constructor.

```
namespace RibbonExtensions
{
    using Caliburn.Micro;
    using System.ComponentModel.Composition;
    using VisualComponents.Create3D;
    using VisualComponents.UX.Ribbon;
    using VisualComponents.UX.Shared;

    [Export(typeof(IRibbonTab))]
    [PartCreationPolicy(CreationPolicy.Shared)]
    internal class ExampleRibbonTab: RibbonTabBase
    {
        [ImportingConstructor]
        public ExampleRibbonTab([Import] ILocalizationService localizationService,
            [ImportMany] IEnumerable<IRibbonGroup> ribbonGroups)
            : base(localizationService, ribbonGroups)
        {
        }
    }
}
```

5. Implement the member properties in the **RibbonTabBase** class.

```
...
        public override string ContextualTabGroupKey
        {
            get { return String.Empty; }
        }

        public override string Header
        {
            get { return "Example"; }
        }

        //reference this Id value in app config file when adding tab
        public override string Id
        {
            get { return "ExampleTab"; }
        }

        public override string Name
        {
            get { return this.Id; }
        }

        public override double Order
        {
            get { return 0; }
        }
...
```

Creating a ribbon group

A ribbon group is a subclass of RibbonGroupBase that uses a base class constructor and implements inherited member properties. The ribbon group itself is exported as a type of IRibbonGroup, and you can use metadata to define the size of controls associated with the ribbon group.

1. Compose a new **internal class** that defines your ribbon group.

```
[Export(typeof(IRibbonGroup))]
[PartCreationPolicy(CreationPolicy.Shared)]
[RibbonGroupMetadata(MinRows = 3)]
internal class ExampleRibbonGroup : RibbonGroupBase
{
    [ImportingConstructor]
    public ExampleRibbonGroup([Import] ILocalizationService localizationService)
        : base(localizationService)
    {
    }

    public override string Header
    {
        get { return "Example Group"; }
    }

    //refers to item in Icons folder of Essentials program files
    public override string Icon
    {
        get { return ""; }
    }

    //id that is used in application configuration file
    //naming convention in app config file requires 'Ribbon' in Id value
    public override string Id
    {
        get { return "ExampleRibbonGroup"; }
    }
}
```

Creating a control

A control is a subclass of ActionItem that uses a base class constructor and inherited members from BaseActionItem. The control itself is exported as an IActionItem and can execute a System.Action.

Define availability

The availability of a control is defined using inherited members from BaseActionItem. Context filters, events and selections can be used to restrict the availability of a control..

1. Compose a new **public class** that defines your control, and make your control available when a robot is selected in the 3D world.

```
[Export(typeof(IActionItem))]
[PartCreationPolicy(CreationPolicy.Shared)]
public class ExampleControl : ActionItem, IHandle<ItemSelectedMessage<ISimComponent>>
{
    //constructor
    public ExampleControl()
        : base("ExampleControl", "Example Control", "", null, null, MenuTools.ButtonTool)
    {
        this.ContextFilter = Contexts.All;

        //create tooltip for control
        this.ToolTipVisibility = System.Windows.Visibility.Visible;
        this.ToolTipHeader = "My Ribbon Control";
        this.ToolTipFooter = "A short description about your control.";

        //get and subscribe to event listener
        this.EventAggregator.Subscribe(this);

        //determine control availability
        this.canExecute = CanBeExecuted;
    }

    //create execution test method
    private bool CanBeExecuted()
    {
        ISelectionManager selection = this.Application.SelectionManager;
        ISimComponent selectedComponent = selection.GetSelected<ISimComponent>();
        if (selectedComponent != null)
        {
            return selectedComponent.GetRobot() != null;
        }
        return false;
    }

    //notify control to test availability
    public void Handle(ItemSelectedMessage<ISimComponent> message)
    {
        this.EvaluateCanExecute();
    }
}
```

Define command

The inherited Execute() method can be used to customize the command executed by a control.

1. In your control class, create a **method** for resetting the joint values of a selected robot to zero, and then override the **Execute()** method to call your robot method.

```
...
        //control command
        public override void Execute()
        {
            SetRobotJointsZero();
        }

        //reset robot to joint zero position
        private void SetRobotJointsZero()
        {
            ISelectionManager selection = this.Application.SelectionManager;

            //get first selected robot
            ISimComponent comp = selection.GetSelected<ISimComponent>();

            //get robot controller and set joints to zero
            if (comp.GetRobot() != null)
            {
                IRobotController rc = comp.GetRobot().RobotController;
                foreach (IJoint joint in rc.Joints)
                {
                    joint.Value = 0;
                }

                this.Application.ActiveWindow.Render();
            }
...
```

Adding ribbon extension

A ribbon extension is added in Essentials by editing the application configuration file.

1. In Visual Studio, open the **application configuration file** located in your Essentials program files.

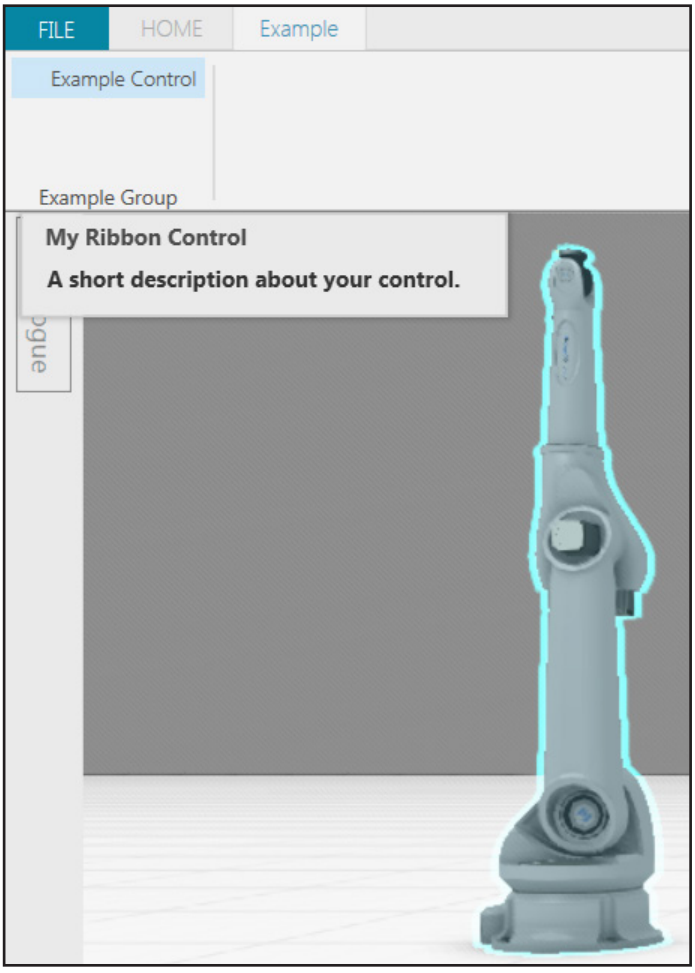
2. In **<UXSitesSection><uxSites>**, add the following code to define your ribbon tab, group and control.

```
<UXSitesSection SectionVersion="1">
  <uxSites>

  <!--ExampleTab-->
    <Site ItemId="ExampleTab">
      <uxSites>
        <Site ItemId="ExampleRibbonGroup">
          <uxEntries>
            <UxItem ItemId="ExampleControl"></UxItem>
          </uxEntries>
        </Site>
      </uxSites>
    </Site>
...
```

3. On the **Standard** toolbar, click **Save All**, and then start debugging the applicaiton.

4. In **Essentials**, verify your ribbon extension has been added, and then test your control.

5. Stop debugging the application.



Adding icons and menu items

You can customize your controls to have icons and additional menu items.

Using available icons

Essentials has a long list of reusable icons that you can preview and assign to controls.

TIP! You can use any icon listed in the Icons folder located in your Essentials program files.

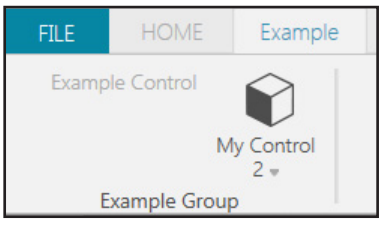
1. Compose a new **control menu** that is always available and uses an Essentials icon.

```
#region MyRibbonControlMenu members
[Export(typeof(IActionItem))]
[PartCreationPolicy(CreationPolicy.Shared)]
public class ExampleControlMenu : ActionItem
{
    //icon name is given in constructor and <path> xml tag for icon is used for display at runtime
    public ExampleControlMenu()
        : base("ExampleControlMenu", "Example Control Menu", "rBox", null, null, MenuTools.ButtonMenuTool, true)
    {
        this.ToolTipVisibility = System.Windows.Visibility.Visible;
        this.ToolTipHeader = "My Ribbon Control Menu";
        this.ToolTipFooter = "A short description about your control.";
        this.EventAggregator.Subscribe(this);
        this.execute = Execute;
        this.canExecute = (() => true);
        EvaluateCanExecute();
    }
}
#endregion
```

2. Add your control menu to the application configuration file, and then save and close the file.

```
...
            <Site ItemId="ExampleRibbonGroup">
              <uxEntries>
                <UxItem ItemId="ExampleControl"></UxItem>
                <UxItem ItemId="ExampleControlMenu"></UxItem>
              </uxEntries>
            </Site>
...
```

3. Start debugging your program to verify your new control with an icon is added to your tab group, and then stop debugging your program.



Create menu items

You can add menu items and availability logic to your control menu by implementing IMenuToolCommand and using the MenuItemBase class.

1. In your control menu class, implement **IMenuToolCommand**, create a **BindableCollection** to contain menu items, and then add a **method** for creating and containing menu items that identify directions.

```
[Export(typeof(IActionItem))]
[PartCreationPolicy(CreationPolicy.Shared)]
public class ExampleControlMenu : ActionItem, IMenuToolCommand
{
    //create menu items using enumeration
    private enum MyCompass
    {
        FaceNorth = 0,
        FaceSouth = 1,
        FaceEast = 2,
        FaceWest = 3,
        FaceUp = 4,
        FaceDown = 5
    }
    private BindableCollection<MenuItemBase> _items = new BindableCollection<MenuItemBase>();
...
    public System.Collections.ObjectModel.ObservableCollection<MenuItemBase> MenuItems
    {
        get { return _items; }
    }
    public ButtonType MenuToolButtonType
    {
        get { return ButtonType.DropDown; }
    }
    public ToolSizingMode MenuToolSize
    {
        get { return ToolSizingMode.ImageAndTextNormal; }
    }

    private bool _isChecked;
    public bool IsChecked { get { return _isChecked; } set { _isChecked = value; } }

    void CreateMenuItems()
    {
        foreach (MyCompass direction in Enum.GetValues(typeof(MyCompass)))
        {
            MenuItemBase menu_item = new MenuItemBase(direction.ToString(), direction.ToString(), "", this);
            menu_item.AttachTo = "[Event Click] = [Action InvokeExecute]";
            menu_item.Execute = () => Execute(menu_item);
            _items.Add(menu_item);
        }
    }
```

NOTE! The Execute member property in BaseActionItem may highlight an error, which is resolved in the next section by defining an Execute() method for menu items.

Executing menu items

You can define a separate type of execution for each menu item in a control menu.

1. In your control menu class, create a **method** that adjusts the view of the 3D world differently for each menu item, and then override the **Execute()** method to call your view method and publish event messages.

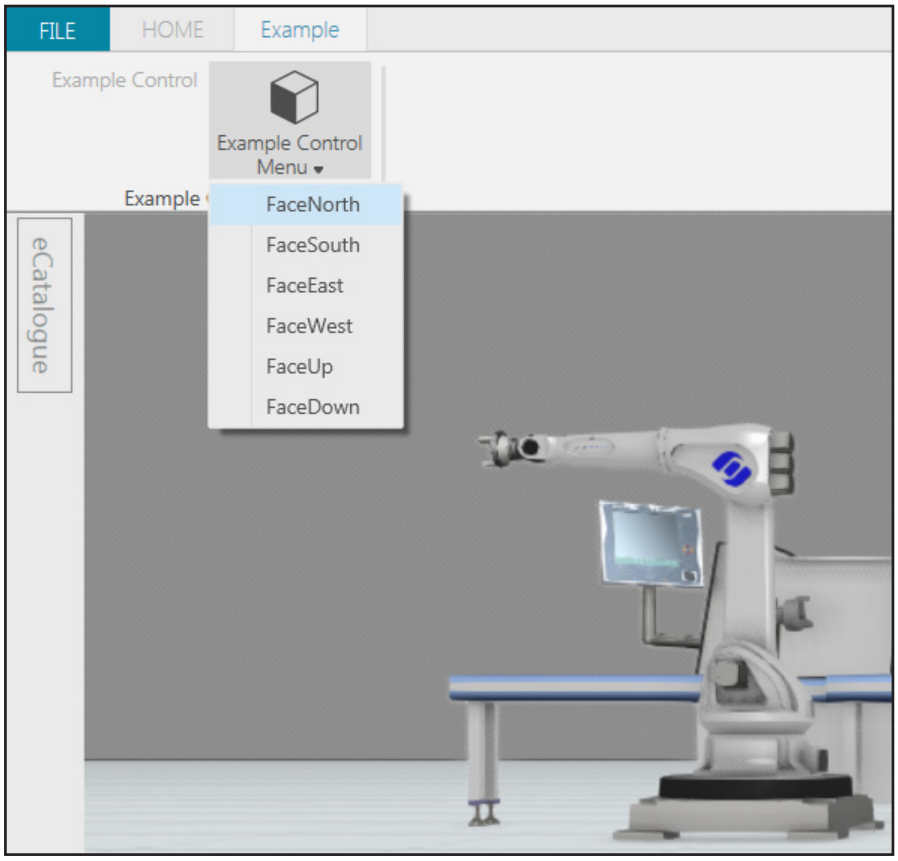
```
...
        public void Execute(MenuItemBase menu_item)
        {
            this.EventAggregator.Publish(new ActionItemExecutingMessage(this));
            SetMyDirection(menu_item);
            this.EventAggregator.Publish(new ActionItemExecutedMessage(this));
        }

        private void SetMyDirection(MenuItemBase menu_item)
        {
            ICamera camera = this.Application.ActiveWindow.Camera;
            camera.FillWorld();
            switch (menu_item.Content)
            {
                case "FaceNorth":
                    camera.ViewFront();
                    break;
                case "FaceSouth":
                    camera.ViewBack();
                    break;
                case "FaceEast":
                    camera.ViewRight();
                    break;
                case "FaceWest":
                    camera.ViewLeft();
                    break;
                case "FaceUp":
                    camera.ViewBottom();
                    break;
                case "FaceDown":
                    camera.ViewTop();
                    break;
                default:
                    break;
            }

            this.Application.ActiveWindow.Render();
        }
...
```

2. In your control menu class constructor, add your method for creating and containing menu items.

```
public ExampleControlMenu()
    : base("ExampleControlMenu", "Example Control Menu", "rBox", null, null, MenuTools.ButtonMenuTool, true)
{
    this.ToolTipVisibility = System.Windows.Visibility.Visible;
    this.ToolTipHeader = "My Ribbon Control Menu";
    this.ToolTipFooter = "A short description about your control.";
    this.EventAggregator.Subscribe(this);
    this.execute = Execute;
    this.canExecute = (() => true);
    EvaluateCanExecute();
    CreateMenuItems();
}
```

3. Start debugging the application.

4. In **Essentials**, add any component to the 3D world, and then test your custom menu items.



This concludes the tutorial.