Using events to update controls

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

Support
support@visualcomponents.com

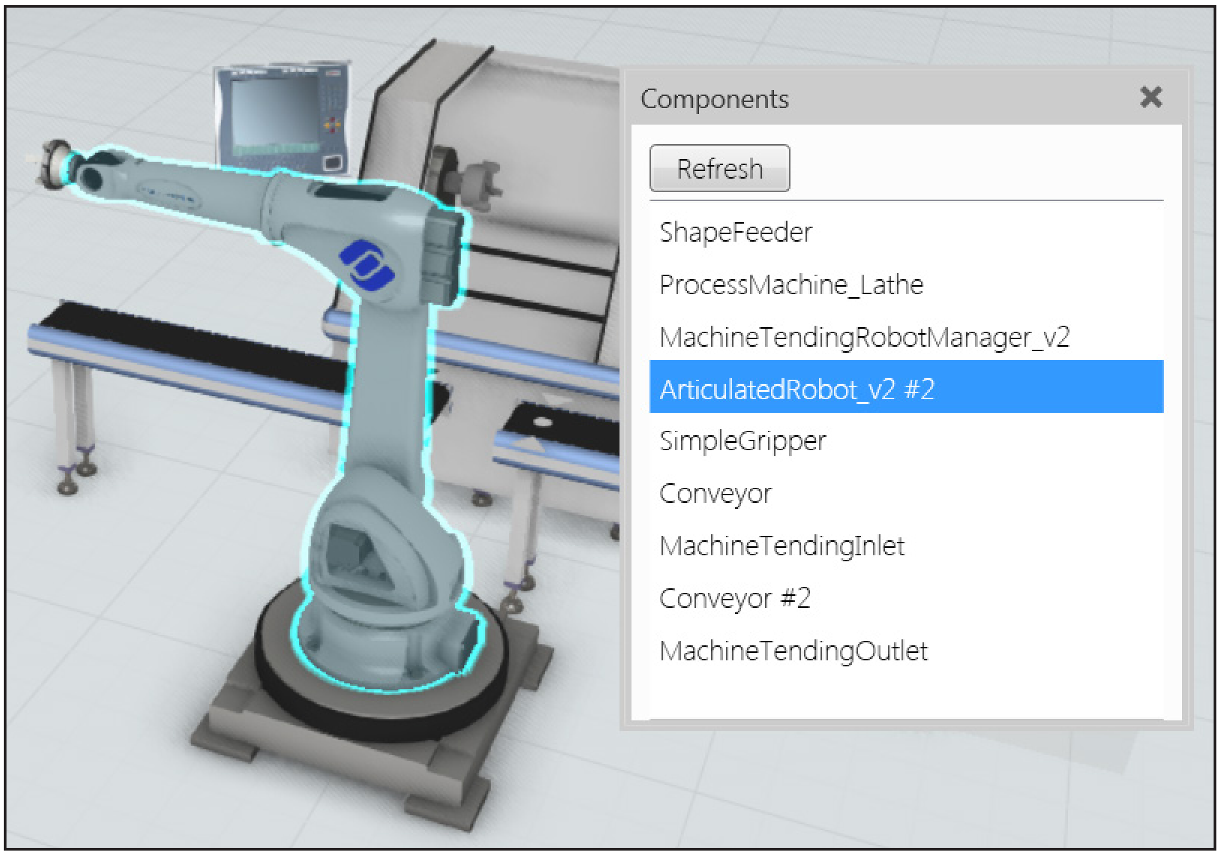
Community
community.visualcomponents.net

Controls in an extension can be updated automatically based on an event that occurs within Essentials, for example when a component is added or removed from the 3D world.

The setup involves assigning a method to an event handler defined in available type, for example ISimWorld. Depending on the context, methods in the Screen class can be used to add and remove methods from event handlers.

The topics covered in this tutorial include:

- Using event handlers to update data binded to controls.
- Tracking components that have been created during a simulation.
- Executing actions by using the properties of a control.

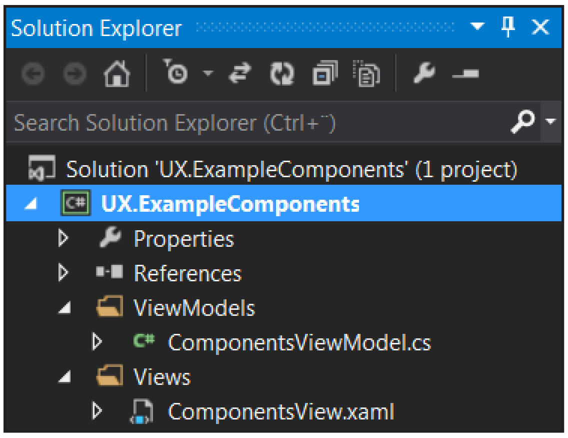


ShapeFeeder
ProcessMachine_Lathe
MachineTendingRobotManager_v2
ArticulatedRobot_v2 #2
SimpleGripper
Conveyor
MachineTendingInlet
Conveyor #2
MachineTendingOutlet

Creating a project in Visual Studio

Visual Studio 2013 and Visual C# are used to create an extension that tracks components in the 3D world.

1. Run **Visual Studio** as an administrator.
2. Create a new **Class Library** project and name that project **UX.ExampleComponents**.
3. In your project, add references to the following assemblies and namespaces:
   - **Caliburn.Micro**
   - **System.ComponentModel.Composition**
   - **System.Windows.Interactivity**
   - **System.Xaml**
   - **VisualComponents.Create3D.Shared**
   - **VisualComponents.UX.Shared**
4. Access the properties of your project.
5. In the **Build** tab, set **Platform target** to either **x86** or **x64** depending on your version of Essentials.
6. Set **Output path** to be the path to your **Visual Components program files**.
7. In the **Debug** tab, set **Start external program** to be the **.exe** file for Essentials.
8. In **Solution Explorer**, delete **Class1**, and then add two **Folder** items: one named **ViewModels** and the other named **Views**.
9. In the **ViewModels** folder, add a new **Class** item and name that item **ComponentsViewModel**.
10. In the **Views** folder, add a new **WPF User Control** item and name that item **ComponentsView**.



Solution 'UX.ExampleComponents' (1 project)
UX.ExampleComponents
Properties
References
ViewModels
ComponentsViewModel.cs
Views
ComponentsView.xaml

Creating code-behind

Your View Model can be exported as a type of IDockableScreen and contain members for collecting and updating data.

Define constructor and export attribute

1. In **ComponentsViewModel**, type the following code to create a **class constructor** and **export attribute**.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace UX.ExampleComponents.ViewModels
{
    using Caliburn.Micro;
    using System.ComponentModel.Composition;
    using VisualComponents.Create3D;
    using VisualComponents.UX.Shared;

    [Export(typeof(IDockableScreen))]
    class ComponentsViewModel: DockableScreen
    {
        public ComponentsViewModel()
        {
            this.DisplayName = "Components";
            IEventAggregator eventAggregator = IoC.Get<IEventAggregator>();
            eventAggregator.Subscribe(this);
        }
    }
}
```

Create bindable collection

A collection of objects can be created to contain data about components in the 3D world.

1. In the **ComponentsViewModel** class, type the following code to create members for collecting and storing component data.

```
#region Properties
private BindableCollection<ISimComponent> _components { get; set; }

//store component objects in property then bind to control
public BindableCollection<ISimComponent> Components
{
    get { return _components; }
    set
    {
        _components = value;
        NotifyOfPropertyChange(() => Components);
    }
}
#endregion

[Import]
private Lazy<IApplication> _app { get; set; }

#region Methods
public void getComponents()
{
    //clear collection and then add components
    _components.Clear();
    foreach (ISimComponent comp in _app.Value.World.Components)
    {
        _components.Add(comp);
    }
}
#endregion
```

2. In the **class constructor**, add the following code to assign a new **BindableCollection** of type **ISimComponent** to a member property.

```
public ComponentsViewModel()
{
    this.DisplayName = "Components";
    IEventAggregator eventAggregator = IoC.Get<IEventAggregator>();
    eventAggregator.Subscribe(this);
    this.Components = new BindableCollection<ISimComponent>();
}
```

Create and assign methods to update collection

A method may be added to an event handler if there is a match between the method and event handler signatures. For example, you can create methods that can be called when a ComponentAdded or ComponentRemoving event occurs in an ISimWorld instance.

1. In the **ComponentsViewModel** class, type the following code to define two methods for updating the data in a member property.

```
public void OnComponentAdded(object sender, ComponentAddedEventArgs e)
 {
     _components.Add(e.Component);
 }

 public void OnComponentRemoved(object sender, ComponentRemovingEventArgs e)
 {
     _components.Remove(e.Component);
 }
```

At the time of execution, some instances of a type may be unavailable until the initialization of the application, main window and 3D world is completed in Essentials. In that case, you can use screen logic to assign functionality to event handlers.

1. In the **ComponentsViewModel** class, type the following code to override the **OnInitialized()** method inherited from the **Screen** class.

```
override protected void OnInitialize()
{
    ISimWorld world = _app.Value.World;
    world.ComponentAdded += this.OnComponentAdded;
    world.ComponentRemoving += this.OnComponentRemoved;
    getComponents();
}
```

NOTE! The DockableScreen class is derived from the Screen class.

Binding data to controls

A BindableCollection object can be a source of data for a control that you can style and customize as needed.

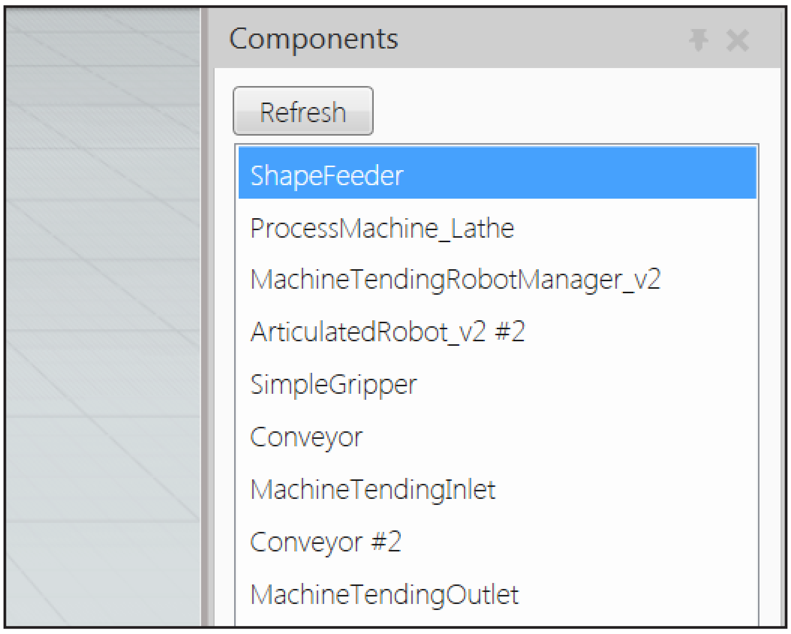
1. Access the **Designer** view for **ComponentsView**.
2. Add **Button** and **ListBox** controls to the **Grid** element, and then readjust the margins of the controls.
3. In the **XAML** editor, type the following code to bind the **Button** and **ListBox** elements to members in the **ComponentsViewModel** class.

```
<UserControl x:Class="UX.ExampleComponents.Views.ComponentsView"
             xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
             xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
             xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
             xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
             mc:Ignorable="d"
             d:DesignHeight="300" d:DesignWidth="300">
    <Grid>
        <Button x:Name="getComponents"
          Content="Refresh" Margin="10,10,0,0" HorizontalAlignment="Left" Width="75" VerticalAlignment="Top"/>
        <ListBox x:Name="Components" Margin="10,40,10,0" HorizontalAlignment="Center" Width="280">
            <ListBox.ItemTemplate>
                <DataTemplate>
                    <Grid Margin="4">
                        <Grid.ColumnDefinitions>
                            <ColumnDefinition Width="Auto" SharedSizeGroup="Key" />
                            <ColumnDefinition Width="*" />
                        </Grid.ColumnDefinitions>
                        <TextBlock Text="{Binding Name}" FontWeight="Bold"  />
                    </Grid>
                </DataTemplate>
            </ListBox.ItemTemplate>
        </ListBox>
    </Grid>
</UserControl>
```

Testing event-driven updates

1. On the **Standard** toolbar, click **Start** to debug your program and run Essentials.

2. In **Essentials**, add the **Machine Tending basic demo layout** to the 3D world to verify the names of components in the layout are automatically listed in the Components panel.



ShapeFeeder
ProcessMachine_Lathe
MachineTendingRobotManager_v2
ArticulatedRobot_v2 #2
SimpleGripper
Conveyor
MachineTendingInlet
Conveyor #2
MachineTendingOutlet

3. Run the simulation, and then click **Refresh** to verify the names of dynamic components are listed in the Components panel.

4. Reset the simulation, and then click **Refresh** to update the list of names in the Components panel.

5. In **Visual Studio**, stop debugging to exit Essentials.

Tracking dynamic components

You can track dynamic components in a collection by using different event handlers.

1. In the **ComponentViewModel** class, type the following code to add methods for updating the collection during a simulation.

```
public void OnDynamicComponentAdded(object sender, ComponentAddedEventArgs e)
{
    _components.Add(e.Component);
}

public void OnDynamicComponentRemoving(object sender, ComponentRemovingEventArgs e)
{
    _components.Remove(e.Component);
}
```

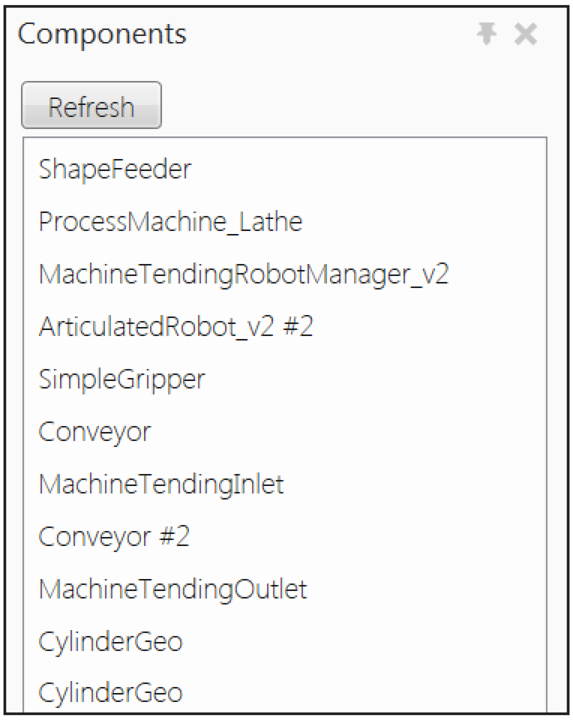
2. Modify the **OnInitialize()** method to assign functionality to the **DynamicComponentAdded** and **DynamicComponentRemoving** event handlers.

```
override protected void OnInitialize()
{
    ISimWorld world = _app.Value.World;
    world.ComponentAdded += this.OnComponentAdded;
    world.ComponentRemoving += this.OnComponentRemoved;
    world.DynamicComponentAdded += this.OnDynamicComponentAdded;
    world.DynamicComponentRemoving += this.OnDynamicComponentRemoving;
    getComponents();
}
```

3. Start debugging your program.

4. In **Essentials**, test data collection during a simulation and when you reset a simulation.

5. Stop debugging your program.



ShapeFeeder
ProcessMachine_Lathe
MachineTendingRobotManager_v2
ArticulatedRobot_v2 #2
SimpleGripper
Conveyor
MachineTendingInlet
Conveyor #2
MachineTendingOutlet
CylinderGeo
CylinderGeo

Dynamic components added to end of list during simulation

Executing actions for selected items

You can execute methods when setting the value of a property that is binded to a control.

TECHNICAL! By using the Caliburn.Micro framework, a selected item can be stored and retrieved automatically by following a naming convention.

1. In the **ComponentsViewModel** class, type the following code to create a property for handling a selected item in the **Components** property.

```
private ISimComponent _selectedComponent { get; set; }

//store selected item using prefix "Active", "Current" or "Selected"
public ISimComponent SelectedComponent
{
    get { return _selectedComponent; }
    set
    {
        _selectedComponent = value;
        NotifyOfPropertyChange(() => SelectedComponent);
    }
}
```

2. Type the following code to import an instance of the **ISelectionManager** interface and define a method for selecting a component in the 3D world.

```
[Import]
private Lazy<ISelectionManager> _selectManager { get; set; }
...
public void selectComponent()
{
    //select component  in 3D world based on selected item in collection
    if (_selectedComponent != null)
    {
        _selectManager.Value.Clear();
        _selectManager.Value.SetSelection(_selectedComponent);
    }
}
```

3. In the member property for a selected item, add the following code to call your method for selecting a component.

```
public ISimComponent SelectedComponent
{
    get { return _selectedComponent; }
    set
    {
        _selectedComponent = value;
        NotifyOfPropertyChange(() => SelectedComponent);
        selectComponent();
    }
}
```

4. In **ComponentsView**, access the **XAML** editor.

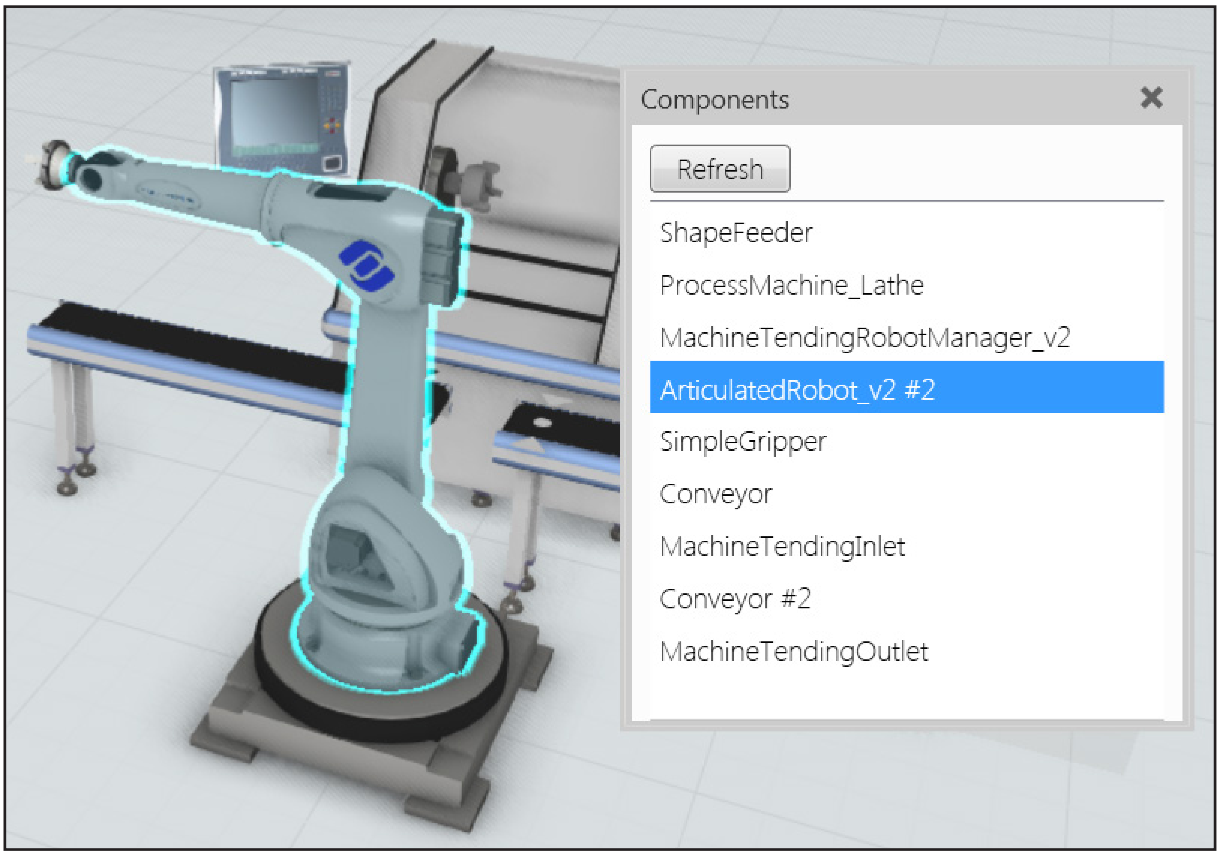
5. Add the following markup in the **ListBox** element to bind the **SelectedItem** property to the member property for a selected item in your View Model.

```
<ListBox x:Name="Components" SelectedItem="{Binding SelectedComponent, Mode=TwoWay}"
        Margin="10,40,10,0" HorizontalAlignment="Center" Width="280">
```

6. On the **Standard** toolbar, click S**ave All**, and then click **Start** to test your project in Essentials.

7. In **Essentials**, add the **Machine Tending basic demo layout** to the 3D world.

8. In **Components**, click an item in the list box to select a component in the 3D world.



ShapeFeeder
ProcessMachine_Lathe
MachineTendingRobotManager_v2
ArticulatedRobot_v2 #2
SimpleGripper
Conveyor
MachineTendingInlet
Conveyor #2
MachineTendingOutlet

NOTE! There are different ways to execute actions for events in a control. One approach involves the use of messages to attach events to actions and use events to invoke actions.

This concludes the tutorial.