

CBW227909

View Extensions for C# Beginners

Andreas Dieckmann
gmp International GmbH

John Pierson
Parallax Team

Learning Objectives

- Learn how to build an extension from scratch
- Work with other Dynamo developers
- ~~Learn how to build Dynamo from source~~
- Build community

Description

This breakout session is specifically targeted at C#-beginners (with some prior programming experience in another language, e.g. Python) and taught by a (relative) C#-beginner who is still very much aware of all the things he had to learn when making his first view extension. It will teach participants extension basics and help participants understand how they can learn how to make simple view extensions. We will look at some code examples to familiarize ourselves with key concepts of view extensions and the Dynamo API and learn how we can create a simple user interface and use it to interact with Dynamo. The DynaMonito view extension will serve as the main learning resource since it contains many smaller, simple tools well suited for learning extension basics.

Speaker(s)

Andreas Dieckmann is a Berlin-based licensed architect, who works as a BIM & Computation Technologist for gmp Architekten (von Gerkan Marg & Partner), Germany's largest architectural office. He is a fairly regular speaker at user groups and conferences like AU and RTC EUR (now BILT EUR). An early adopter of Dynamo, he is the author and co-author of several popular Dynamo packages - first and foremost of Clockwork, currently one of the three most frequently downloaded Dynamo packages. Andreas is a Revit Gunslinger, a member of the Bad Monkeys and participates in the Autodesk Expert Elite program.

John Pierson is a Design Technology Specialist at Parallax Team, and a Revit certified professional for all disciplines. At Parallax, he is engaged in the creation of automated workflows and the exploration of computational design solutions for the AEC industry. John is a frequent presenter at user groups and conferences and is a top-rated speaker at BILT North America. John is also an active member of the Dynamo community and currently manages the Dynamo package Rhythm; which is among the top 5 most downloaded.

Preface

- Since most of the core concepts of view extension are explained during the general session of the Computational BIM Workshop, this is essentially more of a step-by-step walkthrough of the workshop.
- We will not do a lot of typing as that is a very errorprone endeavour, but will rather copy and paste code snippets and talk about what they do.
- Since the code itself is heavily annotated, I am not going to comment much on the code itself in this handout.
- Any necessary actions are marked **bold and red**.
- This tool will be released as part of the DynaMonito view extension shortly after AU.

Data Sets

- The data sets are available on the Developer Workshop repo (<https://github.com/DynamoDS/DeveloperWorkshop>) on GitHub in the subfolder "CBW227909 - View Extensions for C# Beginners"
- If you are git-savvy you can just fork the repo
- Otherwise download it as a ZIP file (<https://github.com/DynamoDS/DeveloperWorkshop/archive/master.zip>)

Folder Structure

- The final state of the project is situated in the "Source" folder. The VS solution and editorconfig are located in the top level folder.
- The "sample_graphs" folder contains a number of sample graphs that can be used for testing throughout the workshop.
- There's a separate code example for each unit in the folder "_Workshop_Units". Each unit has its own VS solution and it makes sense to start each unit from its respective solution rather than using the solution file from unit 01 throughout the workshop.
- Everything that gets added in each step can be found in a "_Snippets" subfolder in the respective unit folder.
- Files that have an "AddTo"-prefix contain code that needs to be added to existing files, other files need to be added to the VS solution.

Building & Debugging

- For the sake of convenience and in order to speed up testing during the workshop the VS solution has been pre-configured to facilitate rapid testing.
- Choosing "Debug" > "Start Debugging" in VS will automatically copy the freshly built view extension as a package (JSON, XML and DLL files) to the appropriate Dynamo directory and launch DynamoSandbox.
- Before you start debugging, make sure that DynamoSandbox is not running, otherwise the build doesn't complete.

NuGet Packages

- You may need to install the Dynamo NuGet packages.
- In Visual Studio go to Tools > NuGet Package Manager > Manage NuGet Packages for solution
- Go to "Browse" and search for "dynamo wpf"
- Install DynamoVisualProgramming.WpfUILibrary
- This should also install all of the necessary dependencies like Dynamo Core etc.

Unit 01: First Things First

When starting a new view extension from a template, there are a couple of chores we need to do before starting to code:

Task	ToDo
Changing file names	<p><i>(This has already been done.)</i></p> <p>In general, this is optional. All your view extension projects could use the same file names. However, if you want to bundle up multiple tools in a single view extension like DynaMonito have a look at how I've organized the various tools into subfolders and named them accordingly: https://github.com/andydandy74/Monito/tree/master/src</p> <p>One file that definitely needs to have a different name for each view extension is the extension definition file that tells Dynamo where to find the view extension DLL. In our project, it has already been correctly named as "Unfancify_ViewExtensionDefinition.xml".</p>
Changing the application name	<p><i>(This has already been done.)</i></p> <p>This needs to be done in a number of places and will affect the name of the DLL that is generated.</p> <ul style="list-style-type: none"> • Project > Properties > Application > Assembly Name • AssemblyInfo.cs > AssemblyTitle and AssemblyProduct • pkg.json > "name" property • XXX_ViewExtensionDefinition.xml > AssemblyPath • XXXViewExtension.cs > "Name" property, probably also for MenuItems below • XXXWindow.xaml > Usually for the Window title
Changing object names	<p><i>(This has already been done.)</i></p> <p>As with file names, your naming schema might vary depending on the scope of your project. Here's what I would rename:</p> <ul style="list-style-type: none"> • XXXViewExtension.cs > The view extension class (definitely) and the MenuItem variable (if needed) • XXXViewModel.cs > The view model class and its constructor • XXXWindow.xaml.cs > The window class • XXXWindow.xaml.cs > Usually the name of the StackPanel • XXX_ViewExtensionDefinition.xml > Typename needs to use the same class name that is defined in XXXViewExtension.cs
Changing the namespace	<p>First Exercise</p> <p>When renaming stuff in VS, use the Rename tool to help you rename all occurrences of the object you're renaming</p>

	<p>– this is generally much safer than going through the code manually.</p> <ul style="list-style-type: none"> • In UnfancifyViewExtension.cs right-click on the name of the namespace (“MyFirstViewExtension”) • Choose Rename • Change the name to “Unfancify” • Check “Preview Changes” and click on “Apply” • Observe which files are going to be changed and click “Apply” <p>We need to change the namespace in some other locations as well:</p> <ul style="list-style-type: none"> • Project > Properties > Application > Default namespace • XXX_ViewExtensionDefinition.xml > Typename
Changing the GUID	<p>Second Exercise</p> <p>Each view extension needs to have a unique ID.</p> <ul style="list-style-type: none"> • Generate a new GUID using e.g. https://www.guidgenerator.com • Assign it to the “Uniqueld” property in UnfancifyViewExtension.cs

Now let’s build our view extension for the first time.

Go to **Debug > Start Debugging**. Dynamo Sandbox should now launch with our freshly built view extension loaded.

(We’ll build a new version at the end of each unit using this method.)

Unit 02: NodeToCode UI – View Extension

Our template already contains all the basic ingredients for a view extension but now we need to fill it with life.

The `IViewExtension` class is the entry point of our little tool so that's where we'll start. In our case we just need to define what our menu item should do, but depending on what a view extension should do you might have a lot more code in this class. Right now it will only display a message box (line 58 in `UnfancifyViewExtension`).

Replace that line with the contents of `AddTo_UnfancifyViewExtension.cs`

The next time we build the view extension, a click on our menu item will now:

- Instantiate our (very rudimentary) view model
- Create a window for our UI
- Set some properties for our window
- Open our window

Unit 03: NodeToCode UI – View

Our current user interface is just a placeholder so let's add some controls and make it look pretty. Windows based applications use WPF (Windows Presentation Foundation) for rendering user interfaces. WPF in turn uses XAML (Extensible Application Markup Language) to define user interfaces. Anyone familiar with HTML or XML should have no problems designing interfaces in XAML.

Our interface (UnfancyWindow.xaml) already contains a few elements. We have a ScrollViewer that'll enable the user to scroll vertically in case the content of our window exceeds its height. Nested inside that is a StackPanel which is going to be the container for all of our controls. And inside the StackPanel there's a TextBlock. Let's add a checkbox and a button to our interface.

Above the TextBlock, paste the first part of the XAML code from AddTo_UnfancyWindow.xaml

Now we have some controls in our interface, but it still needs some styling. Similar to HTML where you can control the visual style with CSS (Cascading Stylesheets), XAML allows us to use a ResourceDictionary to define visual styles for the controls we use in our interfaces.

Add a ResourceDictionary to the project:

- Right-click on the UnfancyViewExtension_AU2018 project in the Solution Explorer
- Choose Add > Existing Item
- In the file dialog set the file type to XAML
- Add the file Shared.xaml from the _Snippets directory

Now we need to make our window use that ResourceDictionary. So let's go back to our UnfancyWindow.xaml file.

- **Above the ScrollViewer, paste the second part of the XAML code from AddTo_UnfancyWindow.xaml**
- **Let's also delete the Text property of the TextBlock tag for now.**

Once we use other controls, we can add additional styles to our ResourceDictionary. Notice the CheckBox and its label still look a bit odd. That's because our ResourceDictionary doesn't contain a general style for CheckBoxes, only a named one. This way we can define several styles for the same element type (e.g. for different interfaces).

Back in UnfancyWindow.xaml add the last line from AddTo_UnfancyWindow.xaml to the CheckBox tag.

For now, we are done with our UI.

Unit 04: NodeToCode UI – ViewModel

The ViewModel is the part of our tool that does all the interaction with Dynamo. Right now, it's just an empty shell. We'll need to tell it what to do. Our first very simple tool is only going to do perform node-to-code on everything in the current workspace, so we don't need to add a lot.

Paste the contents of AddTo_UnfancifyViewModel.cs into UnfancifyViewModel.cs below the Dispose method.

We've now added a property and two new methods. We'll use the UnfancifyMsg property later in the UI to give the user feedback on what the tool did. The OnUnfancifyCurrentClicked method is later going to be triggered by a click on the button in the UI. It updates the value of the UnfancifyMsg property and calls another method (UnfancifyGraph) that will do the actual work: select all nodes and call the node-to-code command.

Note, though, that the code doesn't do anything yet since it isn't triggered anywhere.

Unit 05: NodeToCode UI – Bindings

We already have a working menu item that opens a window and we have some code that can perform an operation on the model. Now we will make the view talk to the view model. For that we need bindings. Bindings are a powerful feature since they enable interaction of the view with the view model.

Let's start with creating a command binding for our button. In our view model we will need to define a DelegateCommand.

After line 16 in UnfancifyViewModel.cs paste the first line from AddTo_UnfancifyViewModel.cs

Notice that the ICommand type in the statement we just pasted shows up as unresolved. That's because we're missing a "using" statement. There are two ways to fix this:

- **Option 1: Add the next line from AddTo_UnfancifyViewModel.cs to the using statements at the start of UnfancifyViewModel.cs**
- **Option 2: Let Visual Studio fix it for us:**
 - Hover over the ICommand type
 - Select "Show potential fixes"
 - Select "using System.Windows.Input"

In our constructor, we now need to instantiate the DelegateCommand.

After line 31 in UnfancifyViewModel.cs paste the third snippet from AddTo_UnfancifyViewModel.cs

We're seeing an unresolved type again (DelegateCommand).

We can solve this the same way as above, this time choosing "using Dynamo.UI.Command".

We've now defined a command that we can bind our button to. In order to do that we need to switch to UnfancifyWindow.xaml

Add the first line from AddTo_UnfancifyWindow.xaml to the Button tag in UnfancifyWindow.xaml

Now the button is bound to the view model command and clicking on the button will trigger the command.

We can also bind view model properties to UI elements. We'll now do that for the message the tool is supposed to display to the user. Let's switch back to our view model.

- **After line 18 in UnfancifyViewModel.cs paste the next line from AddTo_UnfancifyViewModel.cs**
- **Replace the definition for UnfancifyMsg (starting at line 43) in UnfancifyViewModel.cs with the last block from AddTo_UnfancifyViewModel.cs**

Notice that the set definition of the property includes a RaisePropertyChanged statement. This is going to inform the UI when our property has been changed and will cause the UI to update (once the property has been bound to a UI element).

The only thing left to do now is to bind our property to the TextBlock in our XAML code.

Add the last line from AddTo_UnfancifyWindow.xaml to the TextBlock tag in UnfancifyWindow.xaml

Great! We've now built a working view extension that allows us to run node-to-code on all nodes in a graph without selecting them manually. We've now covered all the basics of building a view

extension. In the next units we will refine our tool by adding additional functionality that will help us learn more about other aspects of the Dynamo API.

Unit 06: Ungrouping

The next thing we're going to do is to add an ungroup feature to our tool. First we're going to add the relevant logic and a new property to our view model.

- **After line 41 in `UnfancifyViewModel.cs` paste the first snippet from `AddTo_UnfancifyViewModel.cs`**
- **Insert the second snippet from `AddTo_UnfancifyViewModel.cs` at the beginning of the `UnfancifyGraph` method**

The `UngroupAll` property is going to be used to store a user's input in the next step. The addition to our main method only takes effect if the user has chosen to ungroup all of the groups in the graph. In that case it is going to select all groups before calling the `UngroupAnnotation` command (groups are called annotations in the Dynamo API).

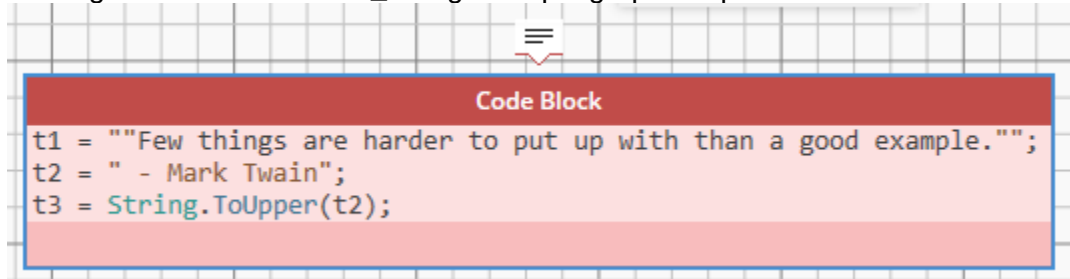
Now we still need to create a UI control that is bound to the `UngroupAll` property:

Insert the contents of `AddTo_UnfancifyWindow.xaml` after the first `CheckBox` tag in `UnfancifyWindow.xaml`

Notice that the `IsChecked` property contains a binding that will update the `UngroupAll` property of the `ViewModel` when the user interacts with the control. Also note that this `CheckBox` has been named. We will need that name later in step 10.

Unit 07: Modifying Node Values

If you've tested the current state of the tool with the sample graphs you might have noticed that running the tool on the Core_Strings sample graph will produce an error:



```
Code Block
t1 = ""Few things are harder to put up with than a good example."";
t2 = " - Mark Twain";
t3 = String.ToUpper(t2);
```

This is due to one of the Dynamo bugs I discovered while preparing this class. String Input nodes that contain double quotes are not correctly converted to code blocks because the double quotes are not escaped on conversion (<https://github.com/DynamoDS/Dynamo/issues/9120>). The same thing happens with backslashes in String Input nodes (<https://github.com/DynamoDS/Dynamo/issues/9117>). While this is generally unfortunate and will hopefully get fixed soon by Team Dynamo, it provides us with the opportunity to learn about modifying node values. Also, I think it's a good example of how we as users can enhance existing Dynamo functionality without having to touch Dynamo Core. We now need to process all the nodes in the graph before we add them to the selection and run node-to-code.

Replace lines 94 & 95 in UnfancifyViewModel.cs with the first snippet from AddTo_UnfancifyViewModel.cs

If any of the examined nodes are of the StringInput type, our tool is now going to escape any double quotes or backslashes it may contain prior to running node-to-code. You'll notice that we're missing some using statements again.

We can solve this the same way as in unit 05, this time choosing "using CoreNodeModels.Input" and "using Dynamo.Graph" respectively.

Unit 08: Auto-Layout

Our next feature is going to be cleaning up the graph after we have called node-to-code by using the AutoLayout command. This is where things got tricky for me when I was preparing the tool for the workshop. In order for auto-layout to work I first needed to make sure that nothing was selected in the graph. Dynamo has a very handy ClearSelection method for that, but unfortunately it is part of an internal class (DynamoSelection) and therefore not available via Dynamo's API. Here's hoping that more parts of Dynamo will be exposed in the API in future Dynamo versions. However, this is an excellent opportunity to introduce the concept of reflection. With reflection we can access non-public methods of loaded assemblies (as long as we know how they work).

Technically, we could add our own ClearSelection method to our view model but since it could be used by various tools within the same view extension I've chosen instead to put it in a separate utilities class.

Add our Utilities class to the project:

- Right-click on the UnfancyViewExtension_AU2018 project in the Solution Explorer
- Choose Add > Existing Item
- Add the file Utilities.cs from the _Snippets directory

Now, let's turn to our view model. There are a couple of things we'll want to do here.

First we're going to create another property for another CheckBox.

After line 48 in UnfancyViewModel.cs paste the first snippet from AddTo_UnfancyViewModel.cs

Now that we have a method to clear the current selection, we can play things a bit safer and use that method before we start selecting stuff anywhere.

- After line 91 in UnfancyViewModel.cs paste the second snippet from AddTo_UnfancyViewModel.cs
- After line 103 in UnfancyViewModel.cs paste the third snippet from AddTo_UnfancyViewModel.cs

Next, we'll add the part that calls the AutoLayout command.

After line 127 in UnfancyViewModel.cs paste the fourth snippet from AddTo_UnfancyViewModel.cs

And, of course, we're missing a using statement again.

Solve this the same way as in previous units by choosing "using System.Windows.Threading".

So why can't we call the AutoLayout command the same way we're calling node-to-code? Well, apparently we first need to wait for the graph to update. Otherwise AutoLayout will not know the correct sizes of the code blocks generated by node to code. By calling it via the Dispatcher and setting its Priority to Background, we can make sure that AutoLayout is not executed prematurely.

Now we only need to add an additional CheckBox in our XAML code so users can choose whether they want AutoLayout or not.

Insert the contents of AddTo_UnfancyWindow.xaml after the first CheckBox tag in UnfancyWindow.xaml

Unit 09: Batch-Processing

Wouldn't it be exciting if we could use our tool on a whole directory of graphs at the same time? Well, we can! And, better yet, we can use the same design pattern for other tools we build on top of Dynamo since it's really quite simple. This way we can build tools that work for a single graph as well as for multiple graphs.

First, we'll add a new method to our view model that does the batch-processing:

After line 82 in `UnfancifyViewModel.cs` paste the first snippet from `AddTo_UnfancifyViewModel.cs`

Again, we're missing a using statement.

Solve this the same way as in previous units by choosing "using `Dynamo.Models`".

We now have a method that opens every graph in a directory, calls our `BatchUnfancify` method, saves the graph and closes it.

Next, we'll need to do some work on our UI. We now need a second button. An easy way to have two buttons next to each other and have them resize with the window is to use a Grid.

Replace the entire `Button` tag in our XAML file with the contents of `AddTo_UnfancifyWindow.xaml`

But where does the user pick a directory? Notice that our second button has a "Click" property? This is pointing to an (as yet undefined) method in the code behind (the `UnfancifyWindow.xaml.cs` file). Let's go there next.

Add the `SelectSource_Click` method after the constructor of our `UnfancifyWindow` class

Again, we're missing a using statement.

Solve this the same way as in previous units by choosing "using `System.Windows.Forms`".

This new method will bring up a directory browser dialog. If the user selects a directory it will call the batch processing method we added to the view model (and pass the directory path).

Just a couple of things to note about batch-processing:

- Batch-processing of DYN files is a bit different to batch-processing of DYF files since there doesn't seem to be a publicly available command to close a custom node workspace so all those custom node workspaces are going to clutter up your Dynamo UI.
- Batch-processing a large number of files may lead to a memory leak. You may experience a significant slow-down of the operation after a while which will eventually grind to a halt completely.
- Have a look at the Dynamo Workspace Upgrade Extension as another example for batch processing graphs.
- UI rendering in batch mode is different. You'll notice that auto layout (the way it's implemented here at least) will not work. Unfortunately, I haven't had the time yet to investigate this further.

Unit 10: Ignore List for Groups

There may be certain groups you'd like to keep, e.g. a group that contains all the inputs or a group that contains a description of what the graph does. So let's add an option to exclude certain groups from ungrouping. In our case we're going to create a TextBox where the user can enter an ignore list. Any group that has a title that starts with any of the terms on the ignore list is not going to be ungrouped.

Let's start with our view model and add a new property that we can later bind to a TextBox in our UI:

After line 55 in UnfancifyViewModel.cs paste the first snippet from AddTo_UnfancifyViewModel.cs

Next, we'll need to create a list of strings to keep track of items in our graph that we want to keep.

At the very beginning of our UnfancifyGraph method paste the second snippet from AddTo_UnfancifyViewModel.cs

Again, we're missing a using statement.

Solve this the same way as in previous units by choosing "using System.Collections.Generic".

Now we're ready to refine the part of our method that does the ungrouping.

Replace lines 138-150 with the fourth snippet from AddTo_UnfancifyViewModel.cs

We now examine every group title and either add the respective group (and its contents) to our list of keepers or ungroup it.

This change also affects the way we should be checking our String Input nodes since we only want to modify the node values of nodes that will later be processed with node-to-code.

Replace lines 171-190 with the last snippet from AddTo_UnfancifyViewModel.cs

Now we'll still need to add a new multiline text input to our UI.

Insert the contents of AddTo_UnfancifyWindow.xaml after the first "Ungroup All" CheckBox tag in UnfancifyWindow.xaml

Note that the IsEnabled property of the TextBox is bound to the CheckBox. That way the TextBox only becomes editable if the user has the ungroup option checked.

Our TextBox still looks a bit funny, though. Let's add another style to our ResourceDictionary:

Paste the contents of AddTo_Shared.xaml into Shared.xaml

Unit 11: Deleting Text Notes (with Ignore List)

We can recycle most of the logic we've already created for groups for this next feature. You might want to unclutter your graph further and get rid of those pesky text notes.

Let's first add two new properties to our view model.

After line 61 in UnfancyViewModel.cs paste the first snippet from AddTo_UnfancyViewModel.cs

Next we can add the logic for processing text nodes.

After line 175 in UnfancyViewModel.cs paste the second snippet from AddTo_UnfancyViewModel.cs

The only difference to the whole logic for groups is that we don't ungroup at the end, but rather delete the text notes that were identified as obsolete.

Of course, our interface needs updating as well and here, too, we can recycle stuff from our ungroup feature.

Insert the contents of AddTo_UnfancyWindow.xaml after the TextBox tag in UnfancyWindow.xaml

Unit 12: Disabling Geometry Preview

Another thing that can boost graph performance is disabling geometry preview. So let's do that. It also gives us an opportunity to learn about modifying node properties.

Let's first add another new property to our view model.

After line 71 in UnfancyViewModel.cs paste the first snippet from AddTo_UnfancyViewModel.cs

Next we can add the logic for disabling geometry preview

After line 238 in UnfancyViewModel.cs paste the second snippet from AddTo_UnfancyViewModel.cs

Obviously, we'll want to run that after having executed node-to-code.

And again, our interface needs another CheckBox.

Insert the contents of AddTo_UnfancyWindow.xaml after the Auto Layout CheckBox tag in UnfancyWindow.xaml

Unit 13: Deleting Downstream Watch Nodes

The last feature we're going to add to our tool is to get rid of any Watch nodes at the end of our graph. Watch nodes and other UI nodes are not converted when executing node-to-code. So we're going to run node-to-code first and after that check the type of any remaining nodes and delete any Watch nodes.

Let's first add another new property to our view model.

After line 76 in UnfancifyViewModel.cs paste the first snippet from AddTo_UnfancifyViewModel.cs

Since disabling geometry preview also requires us to cycle through all of the remaining nodes we're only going to do that once.

Replace lines 245-252 in UnfancifyViewModel.cs with the second snippet from AddTo_UnfancifyViewModel.cs

And again, our interface needs adjusting.

Insert the contents of AddTo_UnfancifyWindow.xaml after the Disable Geometry Preview CheckBox tag in UnfancifyWindow.xaml

Helpful Resources – View Extensions on GitHub

If you want to look at some more code examples for view extensions, you can find a couple on GitHub:

- <https://github.com/andydandy74/Monito>
- <https://github.com/alfarok/DynamoViewport>
- <https://github.com/DynamoDS/DynamoWorkspaceUpgradeExtension>
- <https://github.com/MarkThorley/designtechViewExtension>
- <https://github.com/mitevpi/thesaurus>
- <https://github.com/LongNguyenP/DynaShape>