Hands-On Labs for Creating Pattern Toolkits

This set of guidance contains two hands-on labs. The first is a simple walkthrough for creating your first pattern toolkit. It intentionally uses a nonsense scenario, Widgets, so that you can focus on working with the NuPattern rather than learning a specific domain to tool up.

The second hands-on lab is more in-depth, and covers a range of likely scenarios you will encounter when creating pattern toolkits. The lab guides you through the process of creating one the beginnings of a larger MVC toolkit that will also be made available.

A Note About Audience & Scope

Building Pattern Toolkits is not something that is exclusively limited to Software Developers or Software Architects. In fact, there is unlimited value in building and using pattern toolkits for solution deployment as well. When you consider that the vast majority of any given solution repeats learnings and patterns from previous built solutions, it is easy to see that pattern toolkits have a place in all kinds of different solution types, and can be defined by all kinds of roles in IT, be those software solutions, infrastructure solutions, or combinations of both. It just so happens that the Pattern Toolkit Manager and Builder extensions currently use the extensibility framework of Visual Studio to provide an integrated tooling experience for pattern toolkit builders and users.

Patterns are patterns, and if expertise can be captured and reapplied using patterns integrated with well-known assets, frameworks, scripting and deployment platforms, etc. then pattern toolkits can be used to capture those, and re-apply them in building and delivering predictable solutions.

By the end of these hands-on-labs, regardless of your specific disciplines and experience, you should see that much of the work that goes into building any kind of IT solution could be captured in patterns toolkits that identify variability in a hierarchy of elements with variable properties. And from that, a pattern toolkit can be created for you that integrates custom automation, reusable assets, and instructional guidance, that is driven by the configuration of that information to drive the creation of many of the components of a custom solution. Freeing up those that deliver the solution to focus only on how it is different in their specific implementation, letting tooling and automation provide the remaining parts of the solution predictably and consistently every time.

Hands-On Lab: Getting Started

This lab will guide you through the process of creating a very simple pattern toolkit. The toolkit you will create will help solution developers create Widget classes in a C# Widget project. Although this lab is focused more towards the generation of custom C# code artifacts, the toolkit you will build can be just as easily adapted to generate other kind of textual artifacts, such as power shell scripts, or XML configuration files for any kind of solution, not necessarily focused at developers.

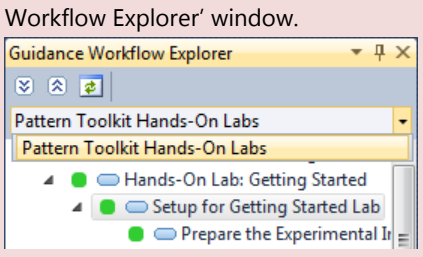
The scenario presented here is intentionally nonsense so that you can focus on the process of building a simple toolkit rather than the specific implementation details of a well-known solution pattern.

Quick Overview

1. You will start by creating a 'Pattern Toolkit' project.
2. You will learn how to test and debug your pattern toolkit in a separate Visual Studio instance.
3. You will learn how to configure a new 'Pattern Model' to simplify, define and represent a type of solution your users may wish to create.
4. Then you will apply some simple 'Automation' to the pattern: a project template, an item template, and a T4 code generation template.
5. You will learn how to enhance the usability of your custom tools that integrate with other windows in Visual Studio.
6. Finally, you will add instructional 'Guidance' to your toolkit, to help your users understand the concepts in your toolkit and how to use it to build their solution.

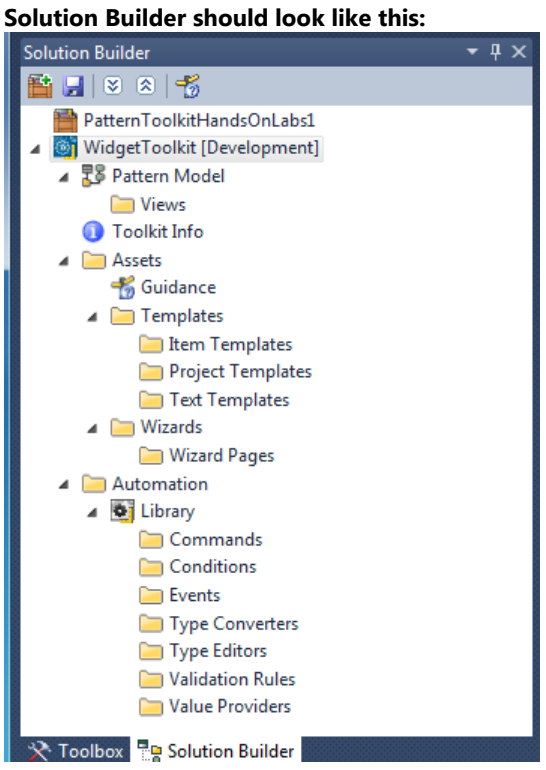
Setup for Getting Started Lab

These steps will help you create a new 'Pattern Toolkit' project, and add it to a solution. If you already have a new pattern toolkit project in your solution that you would like to use, then you do not have to complete these steps.

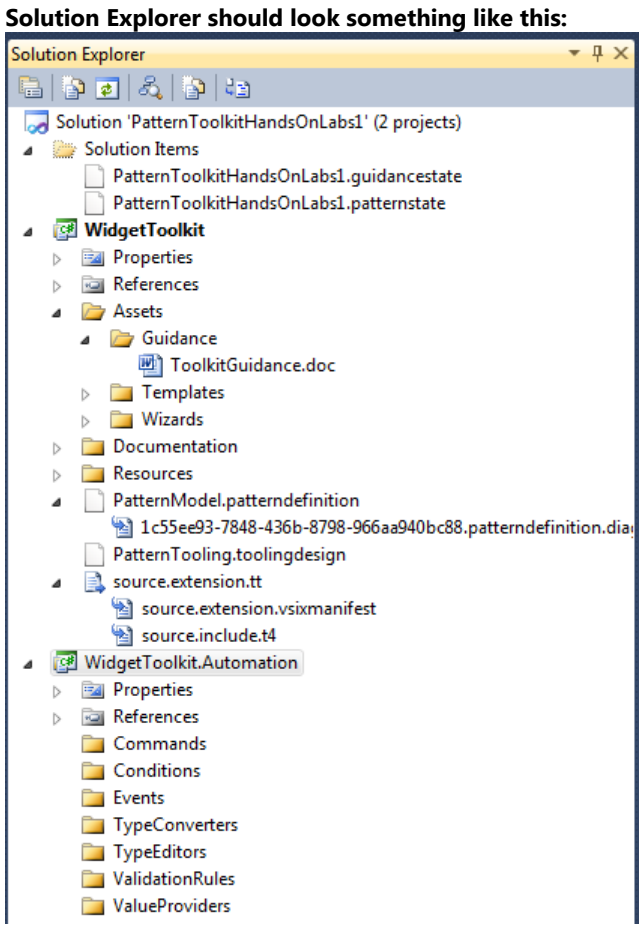
> **Important**: When you are walking through the steps in this hands-on-lab, you will be adding new projects to this solution. These new projects may cause a new guidance topic to be added and selected in the 'Guidance Workflow Explorer' window, which will actually hide the guidance you are reading right now.
> All you have to do to get back to reading this guidance is re-select the "Pattern Toolkit Hand-On-Labs" guidance from the drop down at the top of the 'Guidance Workflow Explorer' window.
>
> 
>

When you are finished with the setup steps that follow, your 'Solution Builder' window should look like the one below. With an element for the pattern toolkit hands-on labs, and an element for the WidgetToolkit project. You may want to refer back to this screenshot later for comparison.

Solution Builder should look like this:



Solution Explorer should look something like this:



Prepare the Experimental Instance of Visual Studio

You have already installed the 'NuPattern Toolkit Builder' extension in your Visual Studio environment, but to test and debug pattern toolkits, you must have the 'NuPattern Manager' extension installed in the 'Experimental Instance' of Visual Studio. (For more information on what the Experimental Instance is, see the Visual Studio SDK documentation.)

> **Recommend:** We recommend that you 'Reset' the Experimental Instance of Visual Studio before developing and testing pattern toolkit projects. However, if you have never developed a Visual Studio extension before, and have never run the 'Experimental Instance' of Visual Studio on your machine, then the following steps are not necessary. Your Pattern Toolkit project will configure the Experimental Instance for you automatically whenever it is built.

Reset the Experimental Instance

The Visual Studio SDK comes with a console program that will allow you to 'Reset the Experimental Instance', that is, it will delete all the extensions installed in the Experimental Instance and copy all the extensions you have installed in the main instance of Visual Studio to the Experimental Instance of Visual Studio.

Click on the Windows Start menu, and type "Experimental" into the search box.

Choose the program called "**Reset the Microsoft Visual Studio 2010 Experimental Instance**."

> **Note**: When you run the above command, a DOS console window should appear, and remain open, tracing out the actions in resetting the 'Experimental Instance', after which it prompts you to "press any key".
>
> **Warning**: If this console window flashes and disappears or the window does not display and prompt you to "press any key", then the reset failed for some reason. This is uncommon, but sometimes occurs. If it does occur, you need to do the following:
>
> 1. In Windows Explorer, paste the following path, and hit ENTER: **%localappdata%\Microsoft\VisualStudio**
> 2. Delete the '**10.0Exp**' sub folder.
> 3. Then re-run the "**Reset the Microsoft Visual Studio 2010 Experimental Instance**." program from the Start Menu again.

Verify Extensions

At this point, you are ready to proceed to the next step.

If you are curious about what was done when you 'Reset' the Experimental Instance, then you can verify that all the extensions are copied and registered in the 'Extension Manager'.

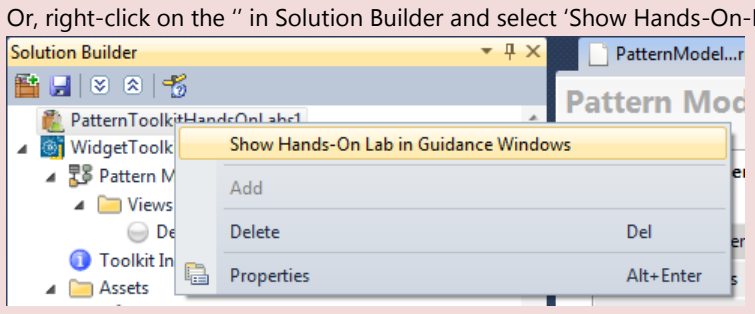
1. Click on the Windows Start menu, and type "Experimental" into the search box. Choose the program called '**Start Experimental Instance of Microsoft Visual Studio 2010**.'
2. Open the 'Extension Manager' window, ('Tools | Extension Manager').
3. Ensure that at the following extensions are present:
    - Pattern Toolkit Manager
4. Close and Save the Experimental Instance of Visual Studio

> **Note**: If while you are developing your pattern toolkit projects, you are having problems testing your toolkits in the Experimental Instance, or problems with the 'Solution Builder' window - it is usually because the above extensions are either not present or are [Disabled].

Part 1: Create a Pattern Toolkit Project

In this section you will create a new Pattern Toolkit project to build a widget solution for the user.

> **Warning**: Before proceeding with this step, be aware that creating or adding a new 'Pattern Toolkit' project to your solution will automatically switch the guidance in the 'Guidance Workflow Explorer' window from that you are following now, and will guide you to specific detailed guidance on building pattern toolkits.
> To restore this guidance, select the 'Pattern Toolkit Hands-On Labs' entry in the 'Guidance Workflow Explorer' window.
> Or, right-click on the '' in Solution Builder and select 'Show Hands-On-Labs Guidance Windows'



> **Tip**: Before proceeding with the following steps in this topic, you may first want to read the whole length of this topic, before stepping through it, as the modal dialogs that pop up will obscure the remaining steps of this topic.

Open the 'Solution Builder' window (**CTRL + W, H**)

Click the 'Add a New Solution Element' button at the top of the 'Solution Builder' window.

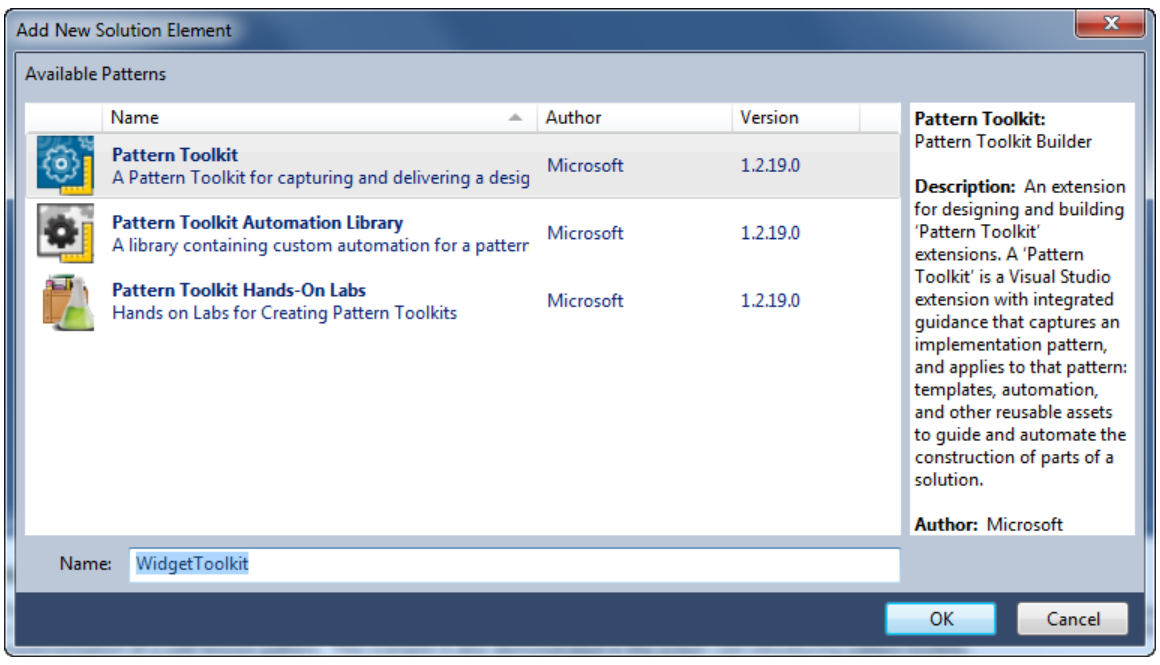
Alternatively, you can click this link to add a 'Pattern Toolkit' project to the solution for you.

> **Advanced Tip**: This link is an example of automation launched from within the guidance. Something you may consider in the future for your own toolkit.

Create the Pattern Toolkit Project

> **Note**: Skip this step if you clicked the link above, go to step 'Configure the Toolkit'.

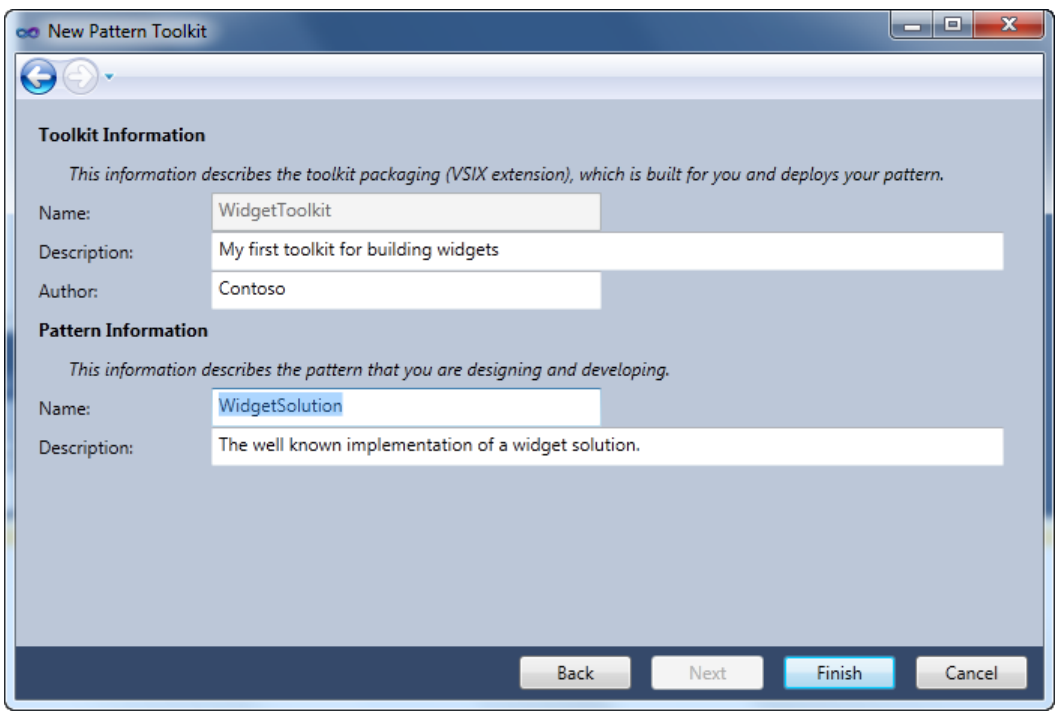
Choose 'Pattern Toolkit' in the 'Add New Solution Element' dialog. Name the toolkit "WidgetToolkit".



Configure the Toolkit

Select 'Create a new toolkit' option, and click 'Next'.

In the 'Toolkit Information', set the Description and Author. In the 'Pattern Information' set the Name to "WidgetSolution", and set the Description.

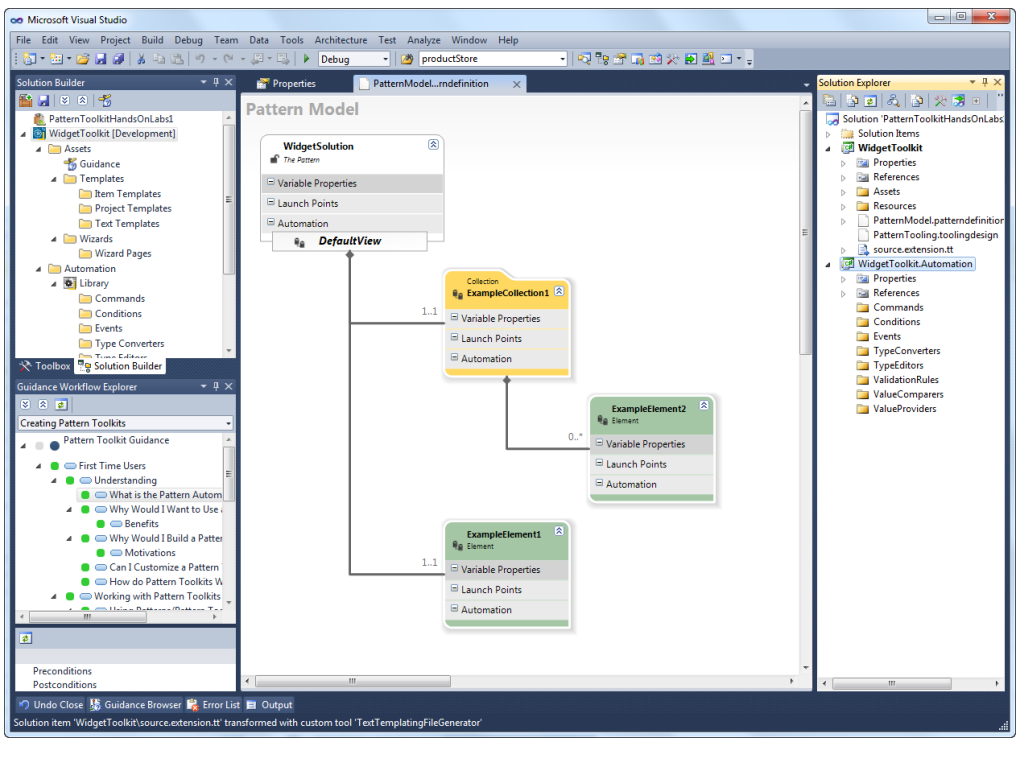


Create the Project

Click 'Finish'.

> **Note**: You will see a new project added to the solution in 'Solution Explorer', as well as new elements created in the 'Solution Builder' window.

The newly created project, and the VS environment should look something like this:



Add the Widget Element to the Pattern Model

The first step of creating any toolkit is configuring the Pattern Model. You will use this model to represent the variability in the pattern that is exposed to the users.

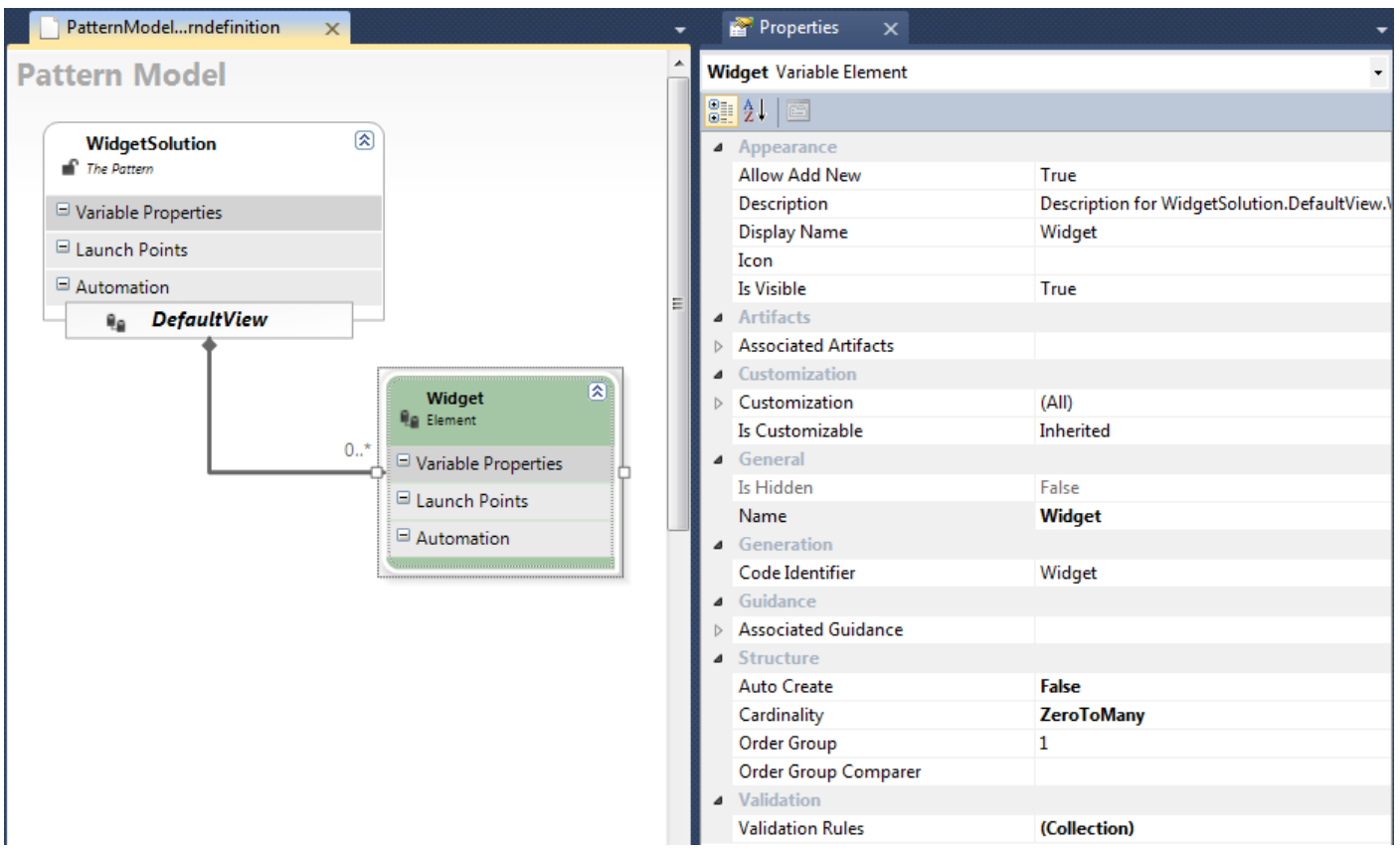
You are not trying to model the pattern itself; you are just exposing the parts of the pattern that are variable—the parts of the pattern that the user can make choices on.

> **Note**: See the guidance topics included in the pattern toolkit guidance for extensive help on the process and methodology for creating pattern models.

In our case, we want to give the user the ability to add Widgets to their solution, and give each Widget a name.

Configure the Pattern Model

Delete the 'ExampleCollection1' shape, rename of the 'ExampleElement1' shape to be "Widget", and set its Cardinality to '"ZeroToMany", and set its 'Auto Create' to "false".



> **Note**: To change the 'Name' property, you can either click on the name in the green shape and type the name, or you can select the shape and open the Properties window and change the 'Name' property.
> **Note**: To change the 'Cardinality' and 'Auto Create' properties, you can either click on the connector, or click on the green shape and open the Properties window and change the 'Cardinality' and 'Auto Create' properties.

Build and Run the Toolkit

That's all we need to do right now. Now let's test our toolkit.

In Visual Studio, click on the 'Debug' menu and select 'Start without Debugging' (**CTRL + F5**).

Tip: It is OK to start with the debugger attached (**F5**), but Visual Studio takes longer to load when a debugger is attached, and there is no code in your toolkit to actually debug at this point.
Note: You may notice that files are being generated into your solution as the build runs. When the build is complete, the Experimental Instance of Visual Studio will open.
Important: Make sure you followed the 'Prepare the Experimental Instance' setup steps at the start of this lab so that Experimental Instance of Visual Studio has all the necessary extensions installed for testing and debugging your toolkit.

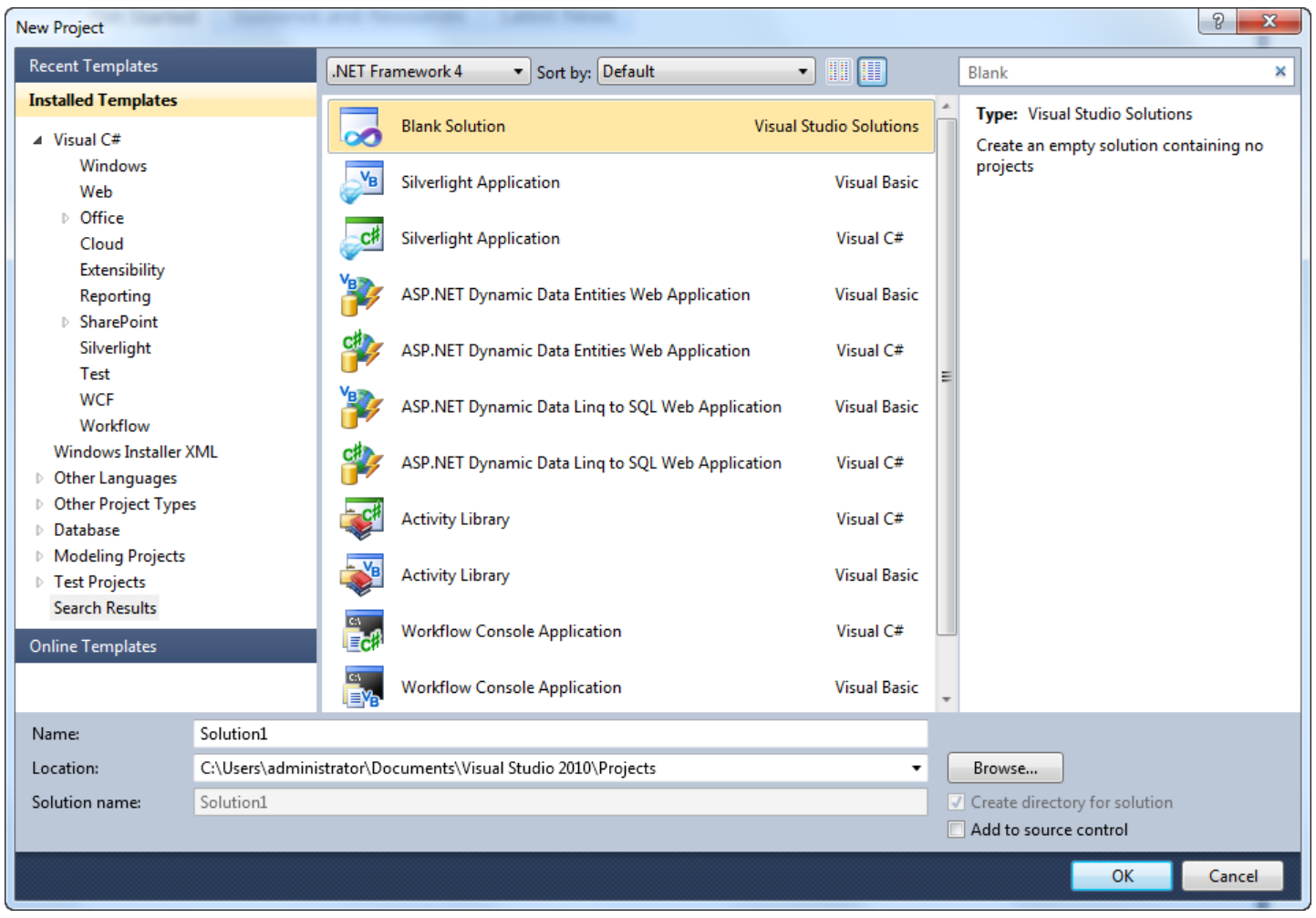
Create a New Solution

> **Note**: When testing your toolkit with **F5**, or **CTRL+F5**, it is loaded into the 'Experimental Instance of Visual Studio'.

In order to test your toolkit at this point, you will have to create an empty solution and then use the 'Solution Builder' window to add an instance of your pattern to the solution.

Start by creating a new, blank solution:

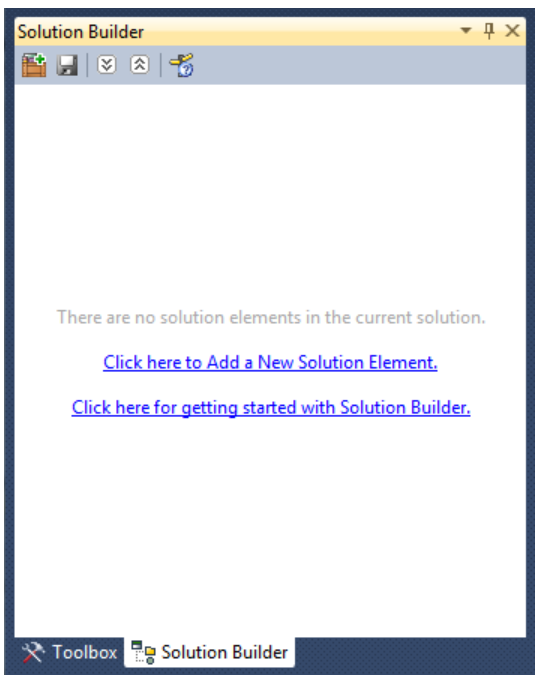
- Click on the 'File | New Project' menu (**CTRL + SHIFT + N**). The 'New Project' dialog opens.
- In the search box at the top right, type "Blank". The list of project types is now filtered to include only templates that have the word 'Blank' in them.
- Name the solution, and click OK to create the solution.



Add a new Widget Solution to Solution Builder

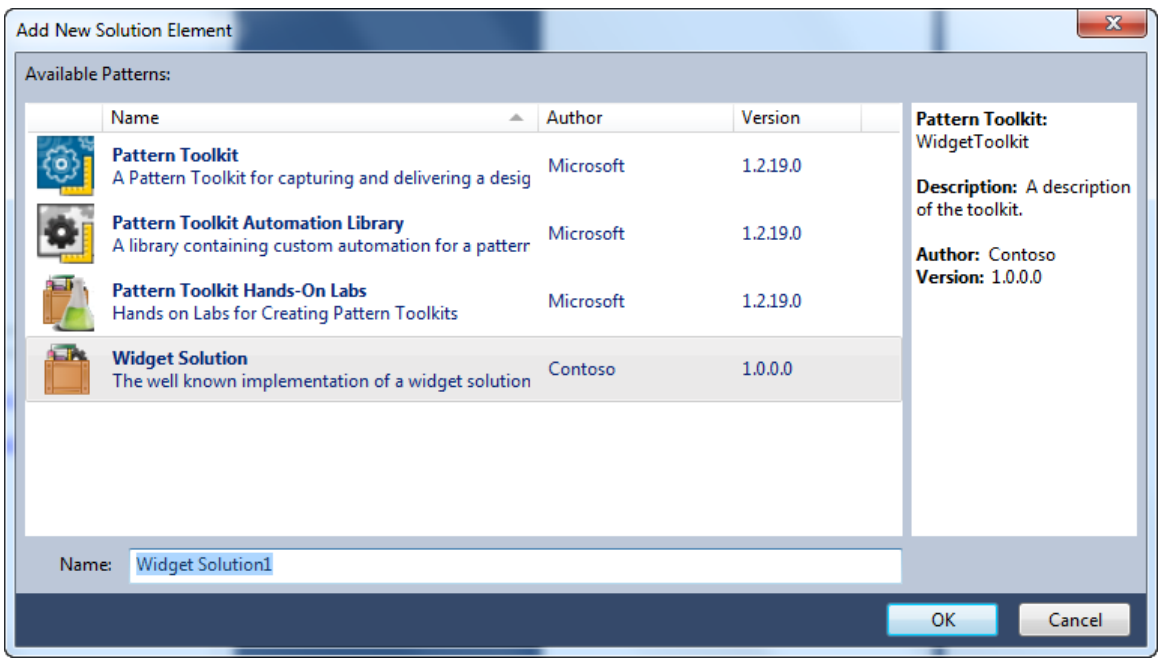
Open the 'Solution Builder' window (**CTRL + W, H**).

> **Note**: If you can't find the 'Solution Builder' window, it is likely that the Experiment Instance of Visual Studio is not correctly configured for toolkit debugging, see the preceding setup steps for 'Preparing the Experimental Instance', or open the 'Extension Manager' (Tools | Extension Manager) window and ensure all extensions are enabled.



Click on the link that says 'Click here to Add a New Solution Element', or click on the 'Add New Solution Element' button in the toolbar.

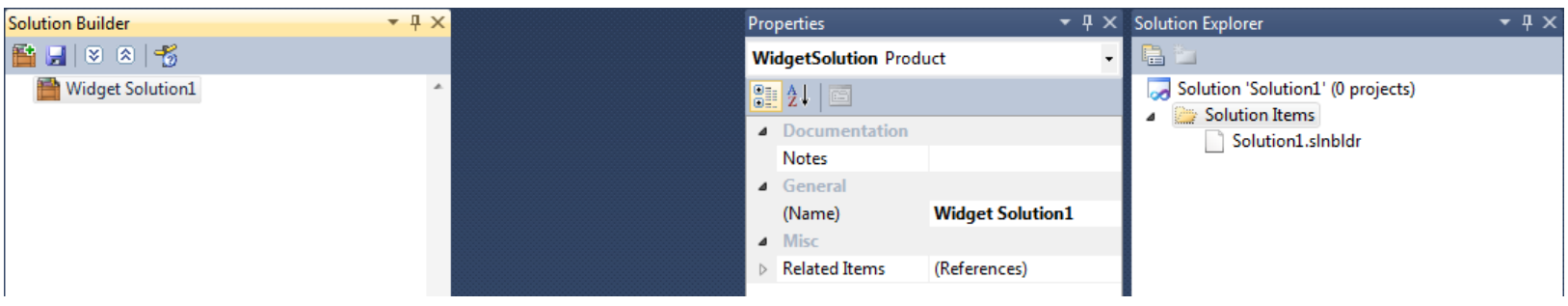
The 'Add New Solution Element' dialog will open, and you can choose your 'Widget Solution', give it a name and click OK.



Widget Solutions are added to Solution Builder

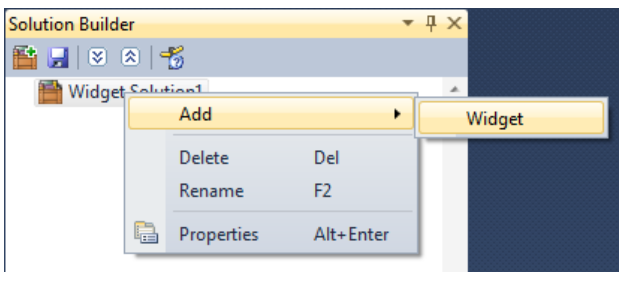
You will see a new element for your solution in Solution Builder, and a new file in Solution Explorer.

> **Note**: You can ignore 'Solution Explorer' for the time being. But leave the 'Properties Window' open.

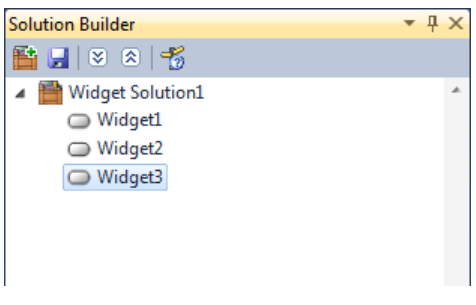


Add More Widgets to Solution Builder

You can now right-click on the 'Widget Solution1' element in Solution Builder, and add new 'Widgets'.



Now add a few, and give them names.

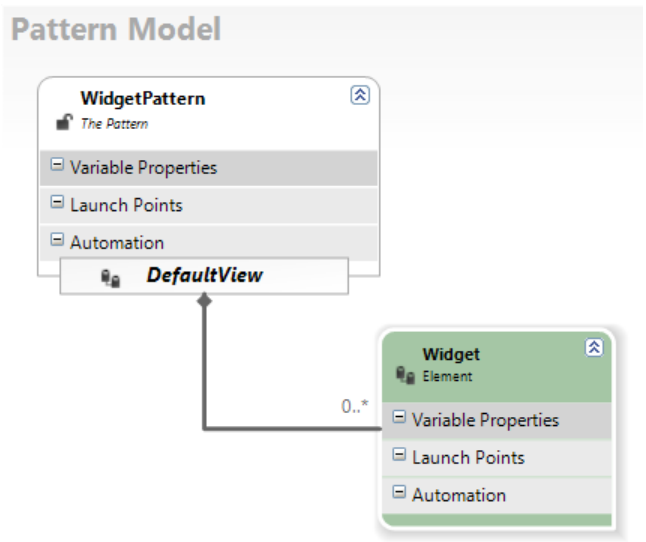


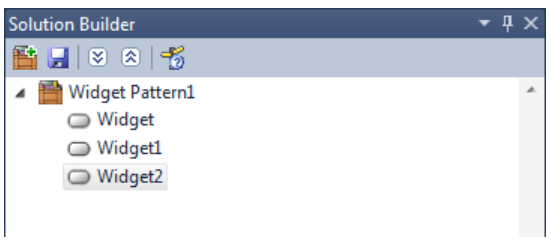
What Have We Done?

You have created your first pattern toolkit, and used it in a solution. Congratulations!

Switch back to your normal instance of Visual Studio, and take a look at the relationship between the pattern model you defined as a toolkit author, and the Widget solution as seen by a toolkit user.

We've defined a model to represent what the user sees and interacts with in the 'Solution Builder' window, and all the tooling has been built for us that installs, and allows a user to add, create and manipulate the properties of these instances in the Solution Builder window.





The next step is to make the toolkit more useful by designing it to create artifacts in a solution for each widget being created.

Part 2: Add Project and Item Templates

In this section we will add project and item templates to the toolkit, so that when the user creates a new instance of the Widget Solutions in 'Solution Builder' a new C# project will be created and added to the solution. And when the user adds a new 'Widget' to the solution, a new Widget code class will appear in the new project.

These Widget classes will be stubs created so that the user will have a place to type widget specific code.

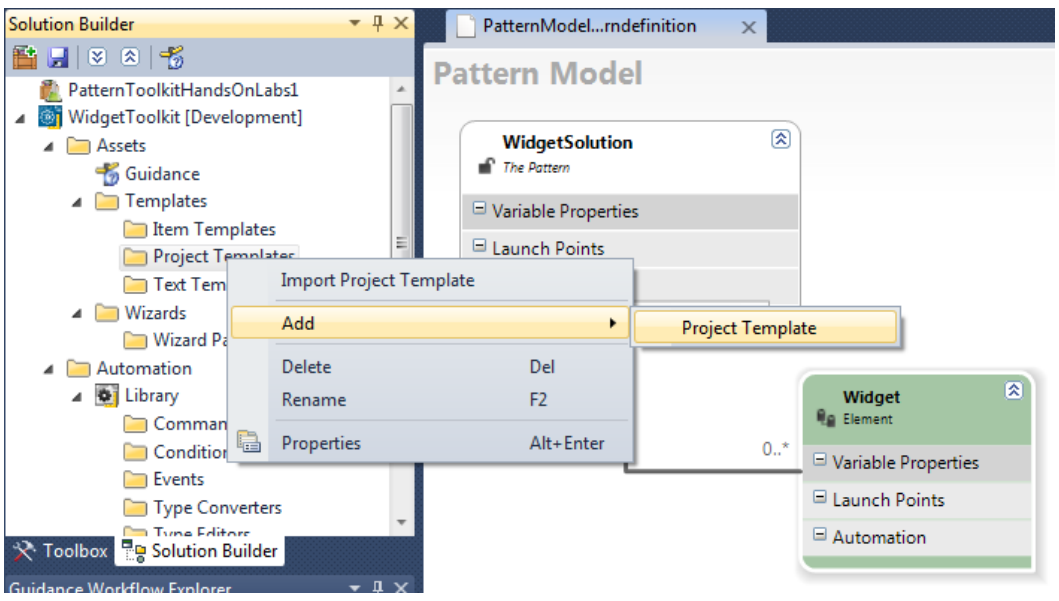
Add a Project Template

Close and Save the Experimental Instance of Visual Studio if it is still open.

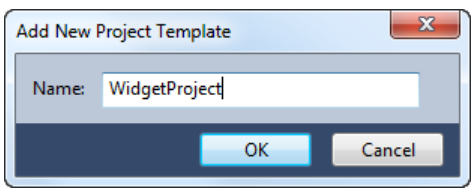
Back in your pattern toolkit project; make sure the 'Solution Builder' window is visible (**CTRL + W, H**).

> **Note**: Believe it or not, you are also using a special toolkit called the "NuPattern Toolkit Builder" toolkit to help you build your widget pattern toolkit project!

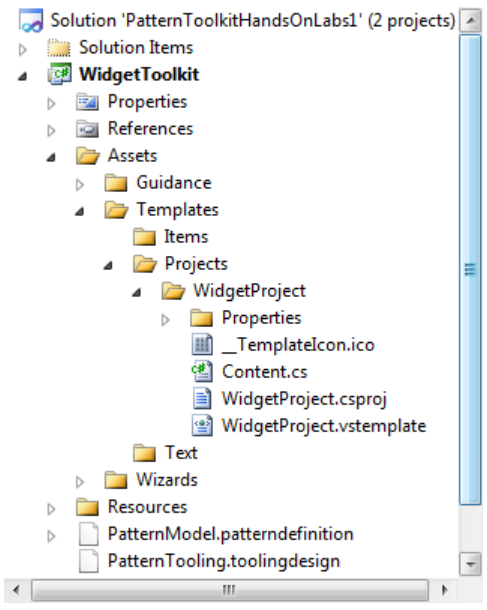
Right-click on the 'Project Templates' element and add a new project template.



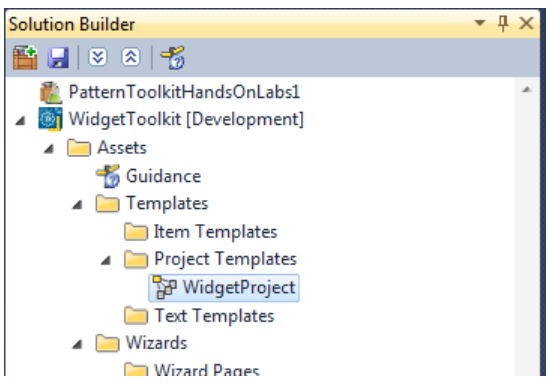
Call it "WidgetProject".



If you look in 'Solution Explorer', you will see that a new project template called 'WidgetProject' has been added to your solution under 'Assets/Templates/Projects'.



> **Tip**: You can automatically navigate to the created template files in Solution Explorer, by double clicking on the new template in Solution Builder.



> **Advanced Tip:** You can also import a previously exported project template from an existing Visual Studio project by clicking the 'File | Export Template' menu. Then you can either 'Import Project Template' (Solution Builder | Project Templates | Import Project Template) or drag and drop the *.zip file onto the 'Project Templates' element in Solution Builder.
> This method of creating project templates allows you to start with an existing project that is an example of the kind of project you want your toolkit to produce, and then templatize it.
> For more information on creating Visual Studio templates, see the MSDN documentation here: http://msdn.microsoft.com/en-us/library/6db0hwky.aspx

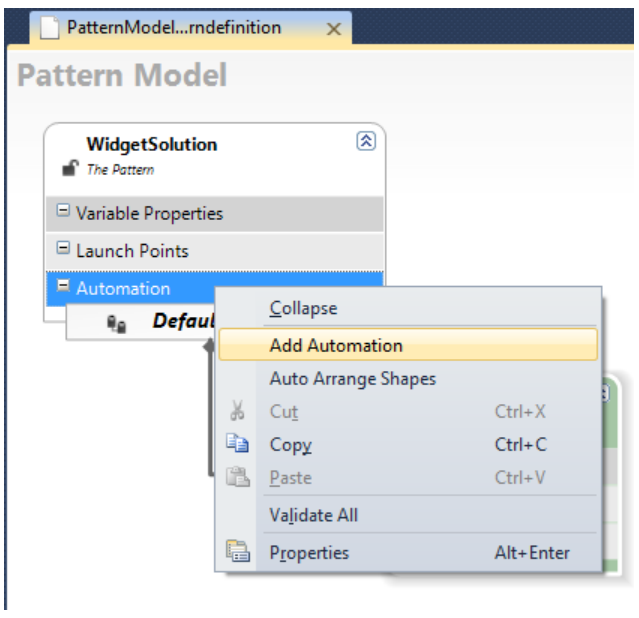
Create a VS Template Launch Point

You have now added a project template to your solution, but we still have to associate it to an element in the Widget pattern model so that when a user creates a new instance of the pattern they get a new project created in their solution.

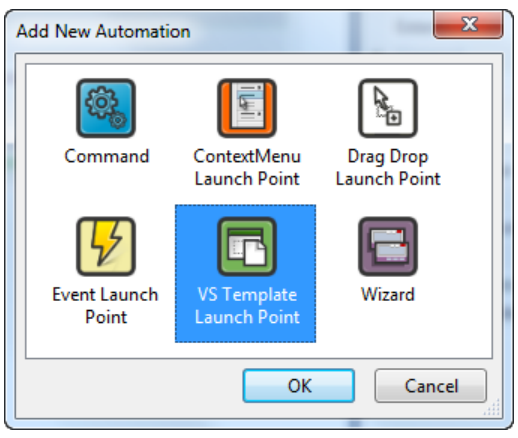
Select the 'WidgetSolution' shape in the Pattern Model.

Right-click on the 'Automation' compartment and choose 'Add Automation'

> **IMPORTANT**: You must right click on either the 'Automation' compartment or the 'Launch Points' compartment in order for the 'Add Automation' context menu to be enabled. Right-clicking anywhere else on the 'WidgetSolution' shape will not allow you at use the 'Add Automation' command.



The 'Add New Automation' dialog appears. Choose 'VS Template Launch Point' and click OK.

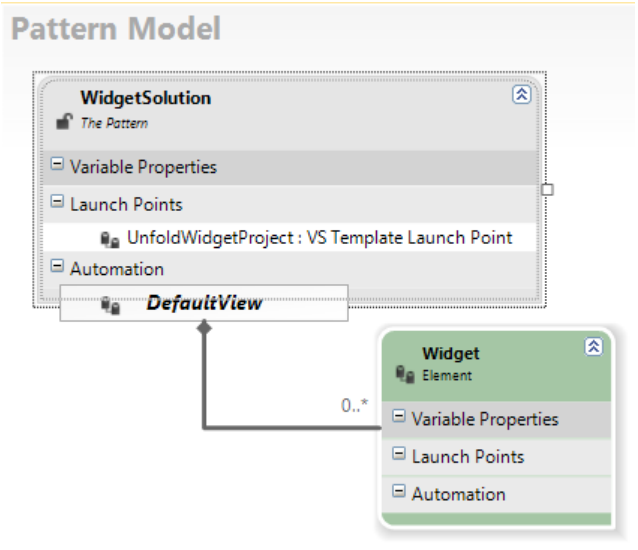


Rename the Template Launch Point

Rename the new launch point to “UnfoldWidgetProject”

> **Tip**: You can either select the launch point and start typing, or find the ‘Name’ property in the Properties window

You may also want to resize the ‘WidgetSolution’ shape so that you can see all the text.

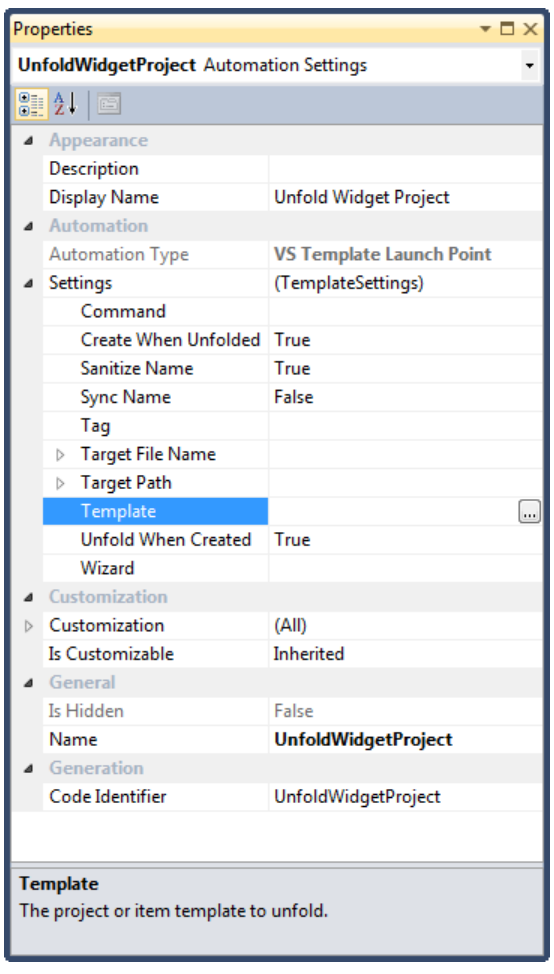


Configure the Template Launch Point

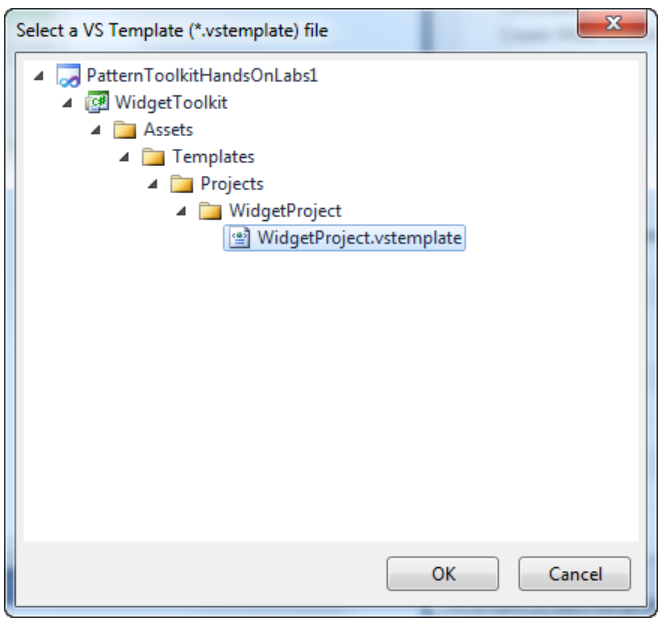
Select the launch point, and look at the properties in the Properties window (**F4**).

Expand the 'Settings' section and you will see the various properties associated with configuring a 'VS Template Launch Point'.

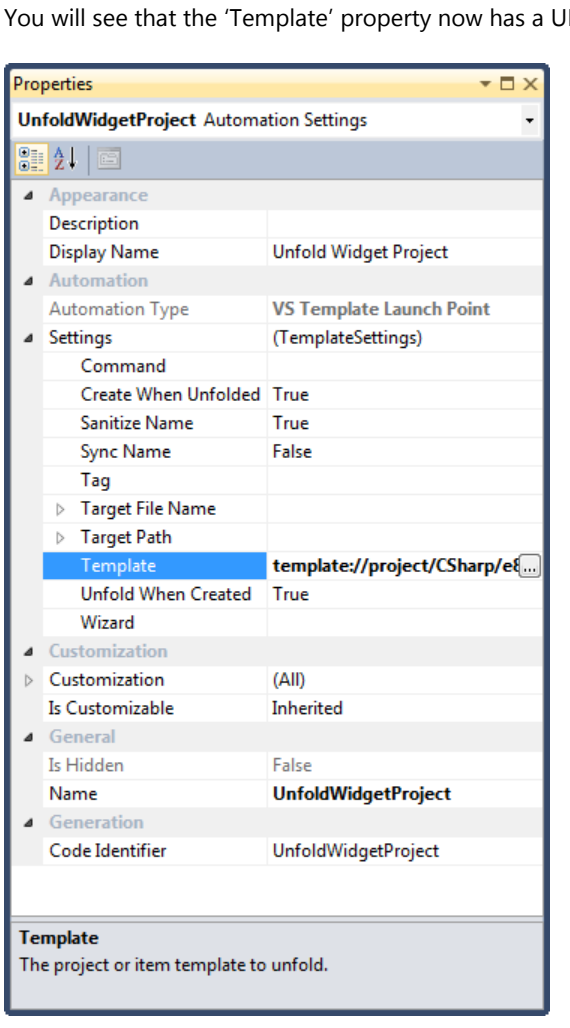
We can accept all of the defaults, but we must define the project template to be unfolded by the launch point, by configuring the 'Template' property.



Click the ellipsis to the right of the 'Template' property and select the *.vstemplate file we added in the step before, and click OK.



You will see that the 'Template' property now has a URI to identify which template will be unfolded by the launch point.



Build and Test

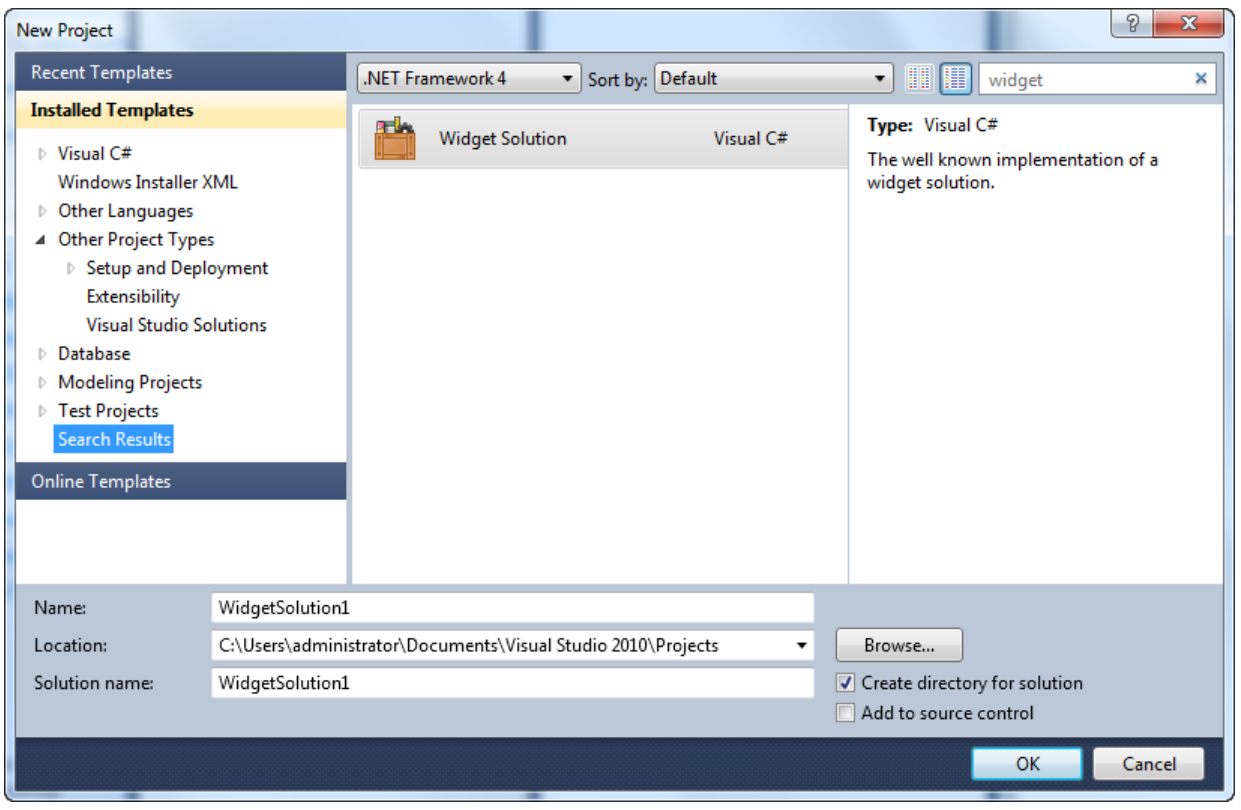
At this point we have created project template, associated it with our pattern model, and configured it to unfold when a new instance of our pattern is unfolded.

Build and Run the toolkit project (**CTRL + F5**).

In the Experimental Instance of Visual Studio that starts up, instead of creating a new blank solution and adding the widget solution in 'Solution Builder', open the 'Add New Project' dialog by clicking on 'File | Add New Project' (**CTRL + SHIFT + N**).

Do a search for "widget" in the search box at the top right corner of the dialog.

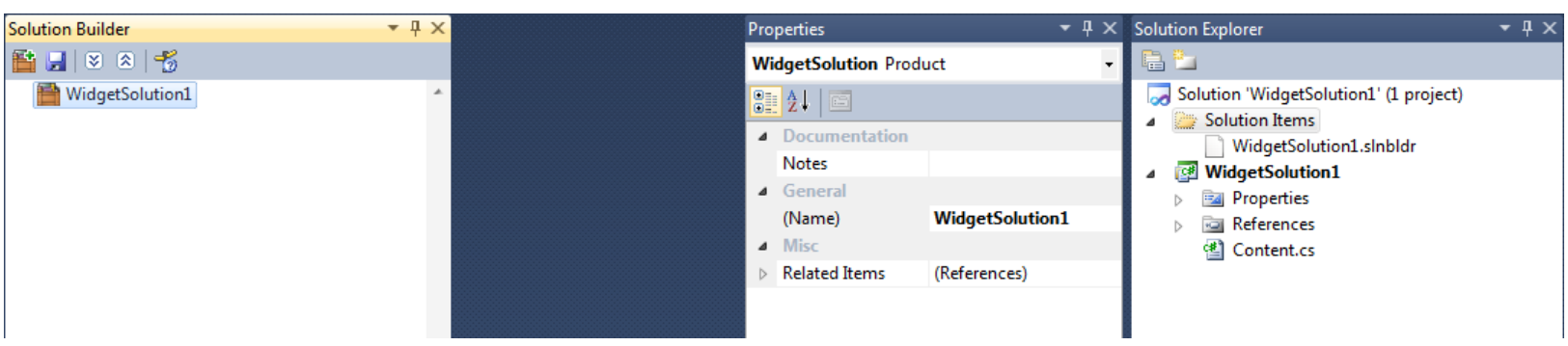
You should see the new project template you just added to your pattern created in the previous steps.



> **Note**: Your new project template, by default, can be found under the 'Visual C#' category.

Click OK to create a new Widget Solution Project.

You will see a new instance of the 'Widget Solution' appear in 'Solution Builder', and a new C# class library project appear in 'Solution Explorer'.



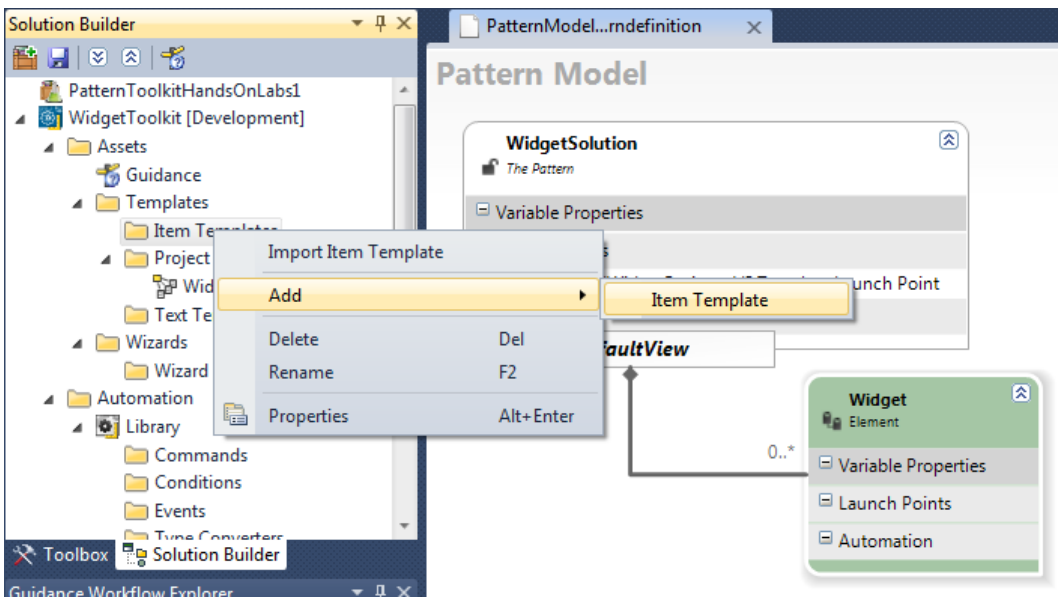
Add an Item Template

Close and Save the Experimental Instance of Visual Studio if it is still open.

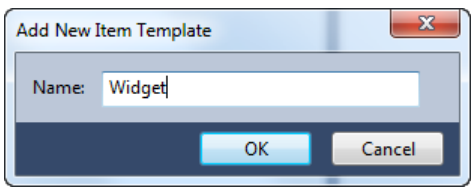
> **Note**: If you leave the Experimental Instance open when you try to build your pattern model project you will get build errors.

Back in your pattern toolkit project; make sure 'Solution Builder' is visible again.

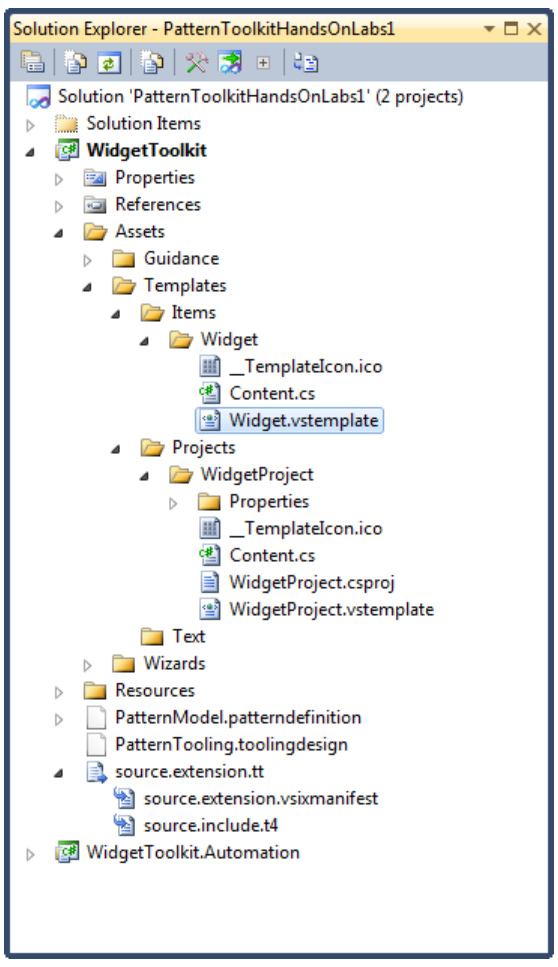
Right-click on the 'Item Templates' element and add a new 'Item Template'.



Call it "Widget"



Again, if you look in 'Solution Explorer', you will see that a new item template called "Widget" has been added to your solution under 'Assets/Templates/Items'.



> **Note**: You will also see that the Content.cs file is open in the editor. It contains special substitution tokens that will be replaced when an instance of your pattern is unfolded.
> '$rootnamespace$' and '$safeitemname$' are part of the standard Visual Studio item template runtime.
> '$InstanceName$' is injected by your toolkit, and will be replaced by the name of the Widget Solution element created by the user.
>
> **Advanced Tip**: You can substitute any properties of any element in the pattern model here also
> (e.g. $InstanceName$, or $MyVariablePropertyName$, or navigate to other elements in the pattern model e.g. $Parent.InstanceName$)

```
namespace $rootnamespace$
{
    /// <summary>
    /// Generated from the $InstanceName$ element in the pattern.
    /// </summary>
    public class $safeitemname$
    {
        public $safeitemname$()
        {
        }
    }
}
```

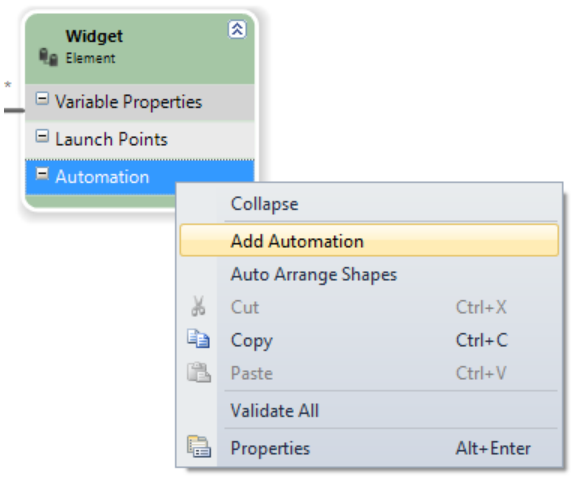
> **Advanced Tip:** You can also import a previously exported project template from an existing Visual Studio project by clicking the 'File | Export Template' menu. Then you can either 'Import Item Template' (Solution Builder | Item Templates | Import Item Template) or drag and drop the *.zip file onto the 'Item Templates' element in 'Solution Builder'.
> This method of creating item templates allows you to start from a file in 'Solution Explorer' that is an example of the kind of file you want your toolkit to produce, and then templatize it. For more information on creating Visual Studio templates, see the MSDN documentation here: http://msdn.microsoft.com/en-us/library/6db0hwky.aspx

Create a Command for Unfolding the Template

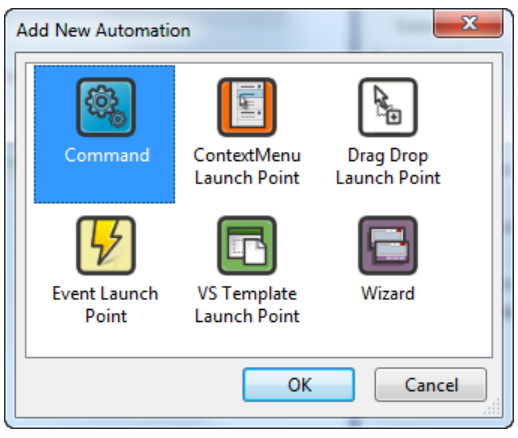
You have now added an item template to your pattern, but we still have to associate it to the 'Widget' element so that when a user creates a new widget in 'Solution Builder' they will get a new C# widget code class in 'Solution Explorer'. We will create a command that will unfold the item template for the user.

Select the 'Widget' shape in the Pattern Model.

Right click on the 'Automation' compartment and choose 'Add Automation'



The 'Add New Automation' dialog appears. Choose 'Command' and click OK.



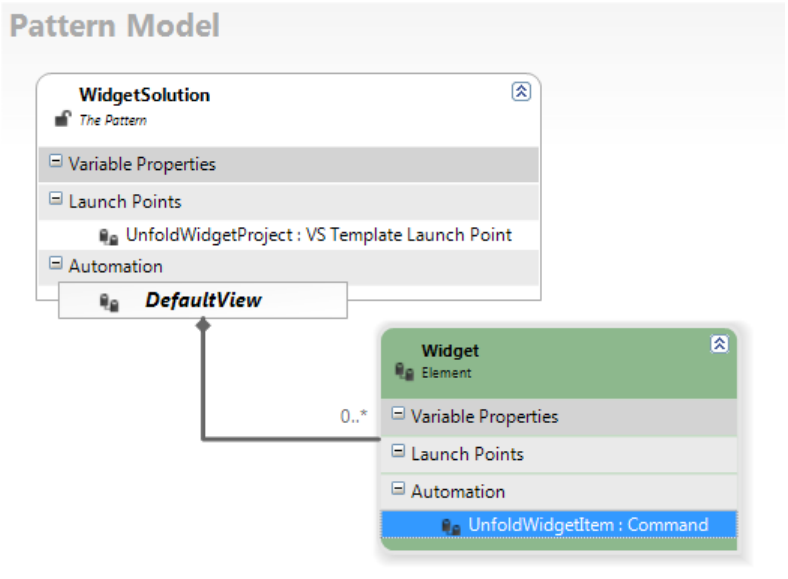
Important: The next few steps are a little different than configuring the project template launch point for the pattern element, where you choose 'VS Template Launch Point'. The 'VS Template Launch Point' is only valid on the pattern element of the pattern. For configuring item and project templates on any other elements of the pattern model you will create a separate 'Command' automation and a separate 'Launch Point' automation.

Rename the Command

The command has now been added to the 'Widget' element, but we are going to give it a more meaningful name.

> **Note**: You can either select the launch point and start typing, or find the 'Name' property in the Properties window.

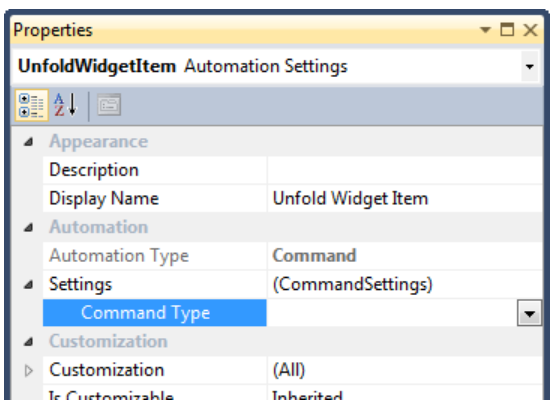
Rename the command to "UnfoldWidgetItem", and you can expand the width of the shape to see all the text.



Configure the Command

Select the command and look at the properties in the Properties window. Expand the 'Settings' section and you will see the various properties associated with a command. The first thing we have to do is choose which type of command we want to run here to unfold the item template.

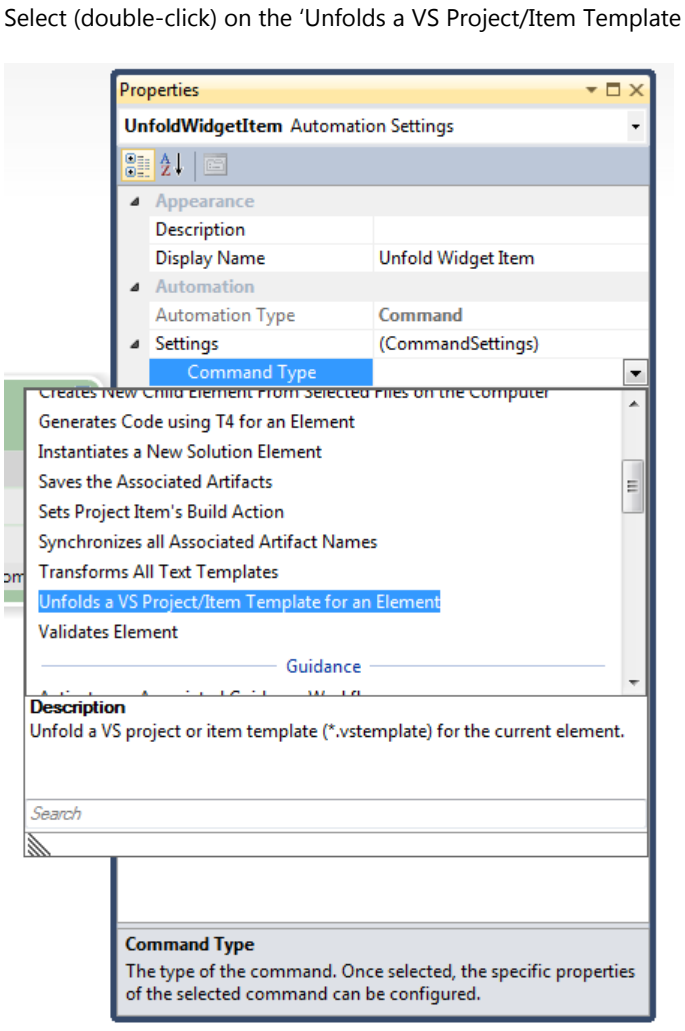
> **Note**: The only property available right now under 'Settings' is 'Command Type'.



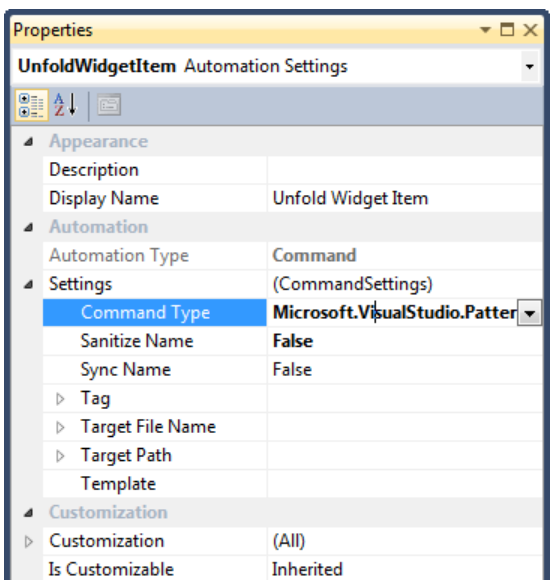
When you click on the drop down in the 'Command Type' property, Visual Studio make take some time, but you will eventually be presented with a list of commands to select from.

> **Note**: The first time you click in the 'Command Type' property Visual Studio interrogates all the assemblies it finds from all projects in the solution for possible commands to include in this list. You only have to wait the first time you open the list.

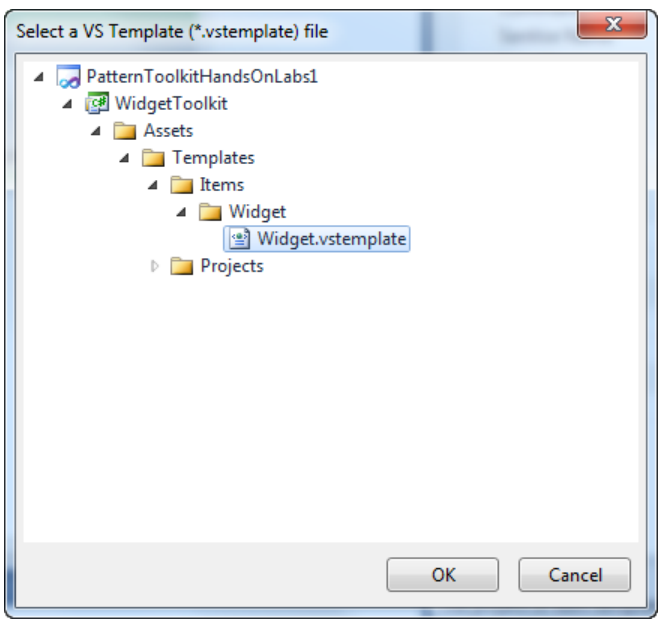
Select (double-click) on the 'Unfolds a VS Project/Item Template for an element' command in this list.



> **Note**: After selecting a command, you will see that a set of more specific properties have been added to the 'Settings' section for the selected 'Command Type'

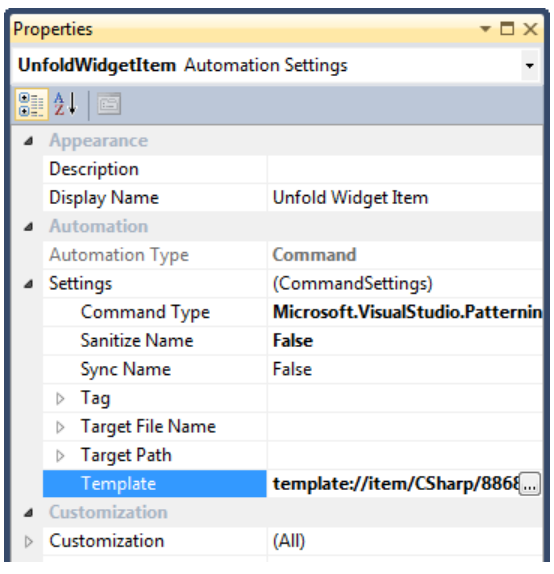


We must now set at least the 'Template' property. Click in the 'Template' property and click on the ellipsis button. That will open the 'Select a VS Template dialog'.



Select the "Widget.vstemplate" file, under 'Assets/Templates/Items/Widget', and click OK.

Your fully-configured command should look something like this:

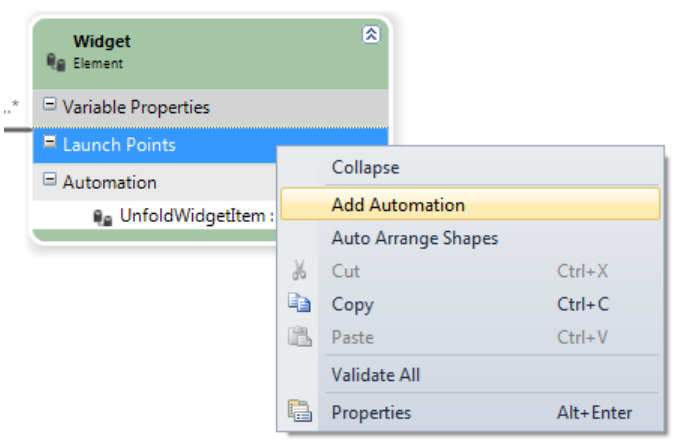


Create a Launch Point to trigger the Unfold Command

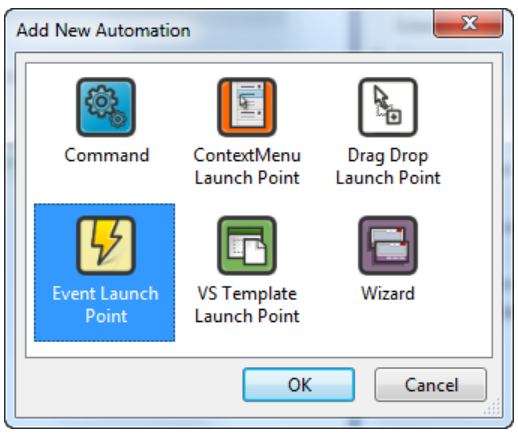
The second half of configuring any command is to create a launch point that triggers the command in response to some event or to some user's gesture.

We want the 'UnfoldWidgetItem' command to execute when a new 'Widget' is created in 'Solution Builder', which occurs when a new instance of a 'Widget' element in instantiated.

Right-click on the 'Launch Points' compartment of the 'Widget' element, and select 'Add Automation'.

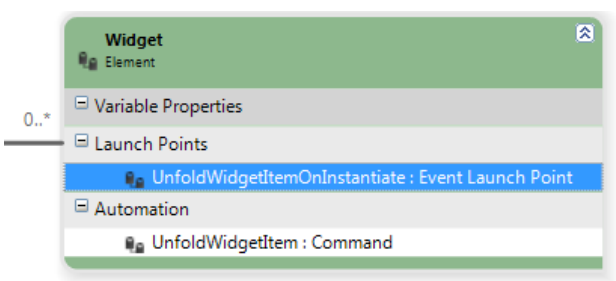


Choose the 'Event Launch Point', because we want the command to run when in response to an event triggered by adding a new 'Widget' item in 'Solution Builder'.



Rename the launch point to "UnfoldWidgetItemOnInstantiate"

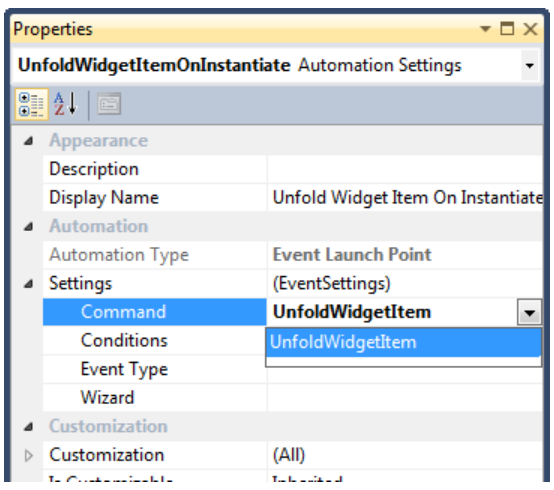
> **Note**: We are implementing a naming convention where the launch point is the name of the command being launched, with a suffix that indicates when the launch point will be run. For example, if we created a context menu launch point we would call it "UnfoldWidgetItemOnMenu".



Click on the launch point, and look at the Properties window.

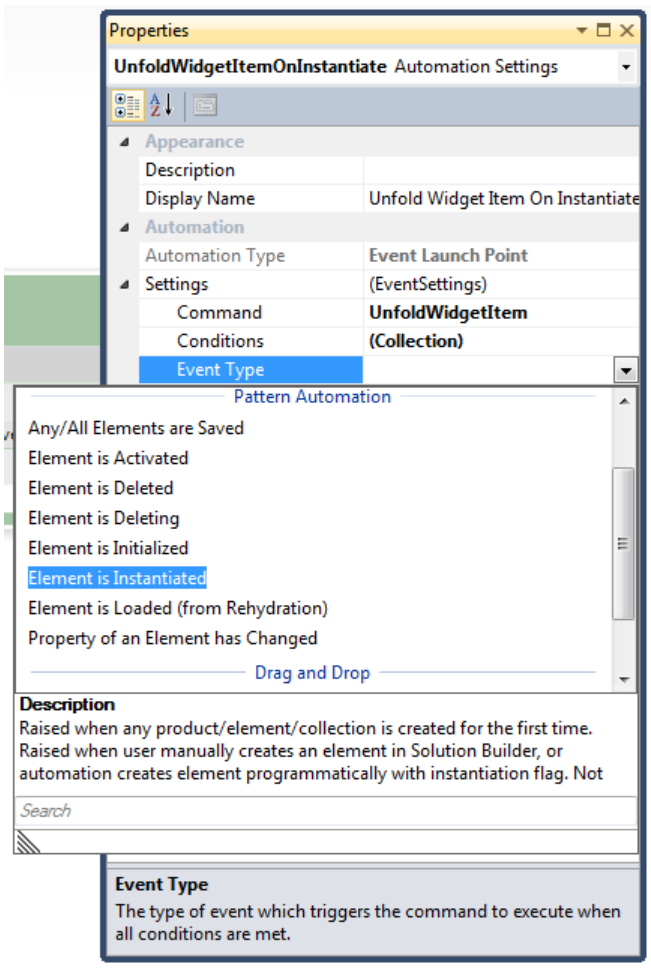
Expand the 'Settings' section. We must tell the launch point which command to run when it is triggered.

In the 'Command' property, select "UnfoldWidgetItem" from the drop-down list.



For this kind of launch point, we must also select the type of event that we want to hook, that will trigger the command.

Set the 'Event Type' property to "Element is Instantiated", which configures the launch point to trigger when a new 'Widget' element (the current element) is created in 'Solution Builder'.



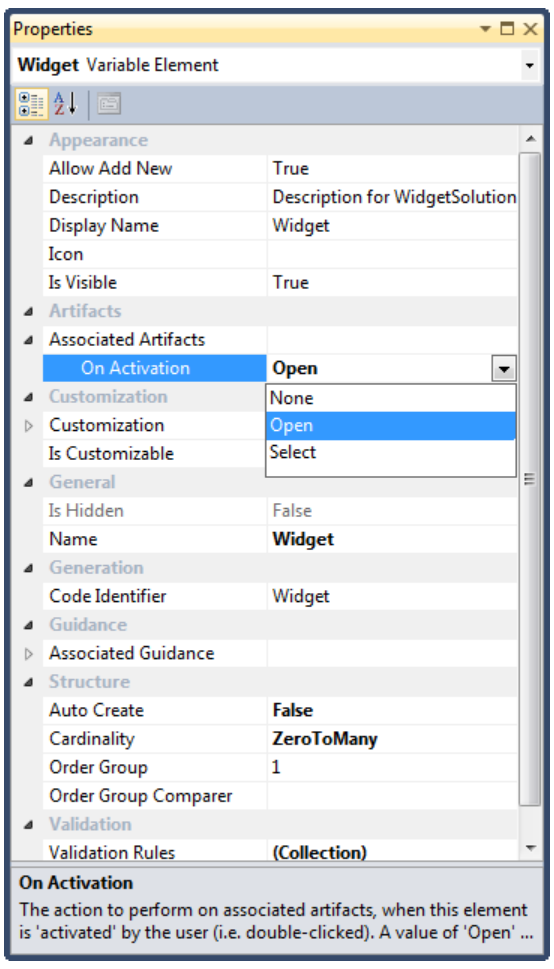
> **Advanced Tip**: It is also possible at this point to configure any 'Conditions' that must be evaluated before the selected command is executed by the launch point.
> You can also set up a 'Wizard' to run before the command is executed if you want to gather data from the user before the command runs.
> See those additional topics in the guidance for Pattern Toolkits, for instructions on how to create and configure conditions and wizards.

Enable Navigation

Since each instance of a ‘Widget’ element will unfold (at least) one code file in the solution, it is very handy for users of your toolkit to be able to navigate easily from the element instance in ‘Solution Builder’ to the code file wherever it may be in ‘Solution Explorer’. This is particularly important in large or complex solutions.

Select the ‘Widget’ element, and set the built-in artifact navigation options for it.

Expand the ‘Associated Artifacts’ property, and set the ‘On Activation’ property to “Open”.



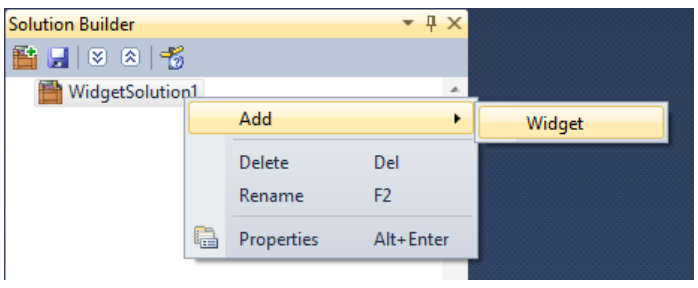
Note: As you will see later, this selection adds several gestures to elements in ‘Solution Builder’ to navigate to artifacts associated to this element.

Build, Run and Create Widget Classes

Build and Run the toolkit project (CTRL + F5).

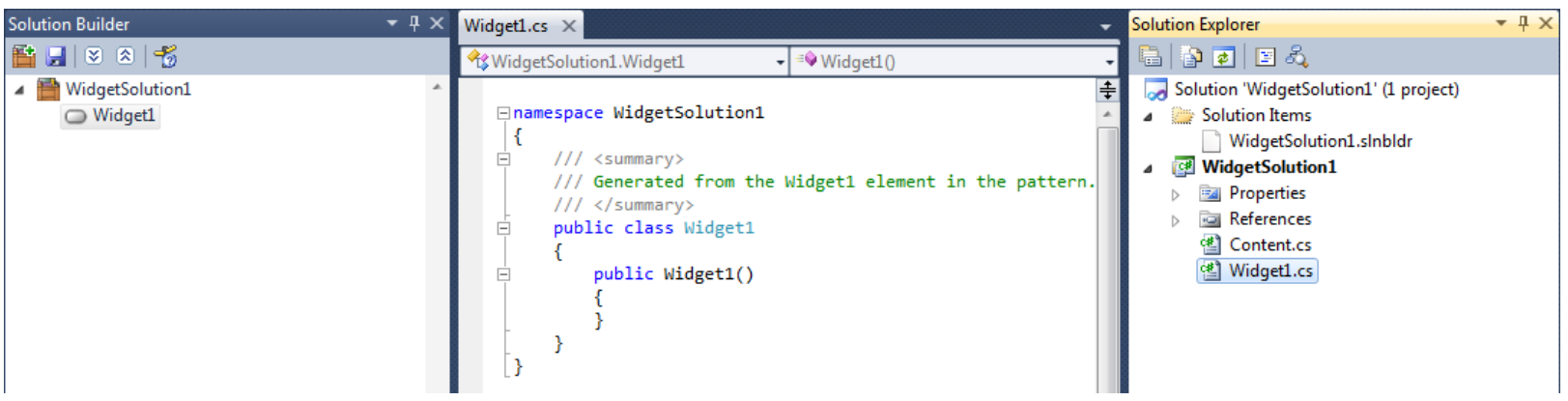
In the Experimental Instance of Visual Studio that starts up, this time you can create a new 'Widget Project' in the 'Add New Project' dialog (File | Add New Project) or you can use the previous solution from the last test cycle.

Either way, start adding new instances of the 'Widget' element in 'Solution Builder'.

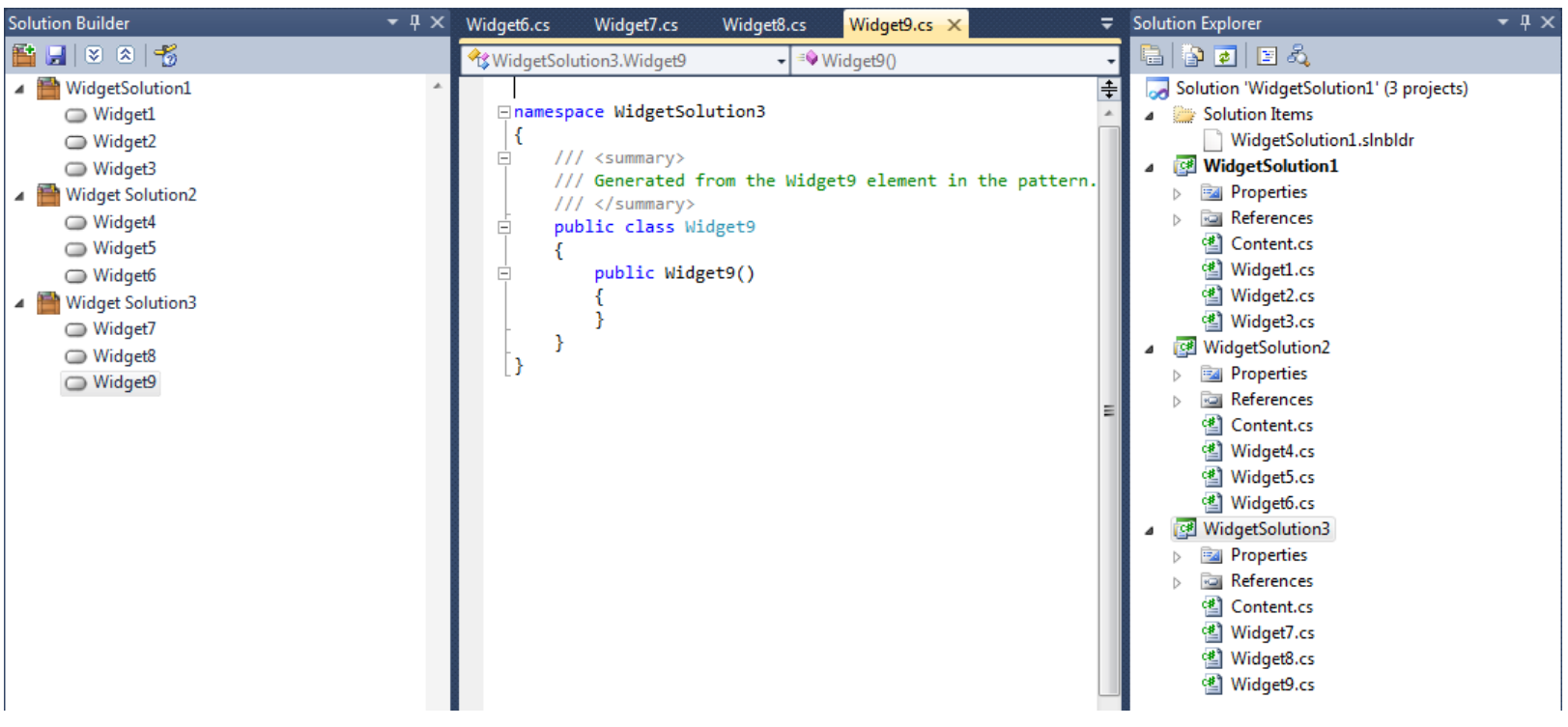


Now, you will notice that whenever you add a new instance of a 'Widget' element in 'Solution Builder', it automatically creates a new C# code file in the Widget project in 'Solution Explorer'.

Each file is named the same as the new element, and you can see the substitutions (from properties of the new element) are automatically made in the content of the file.

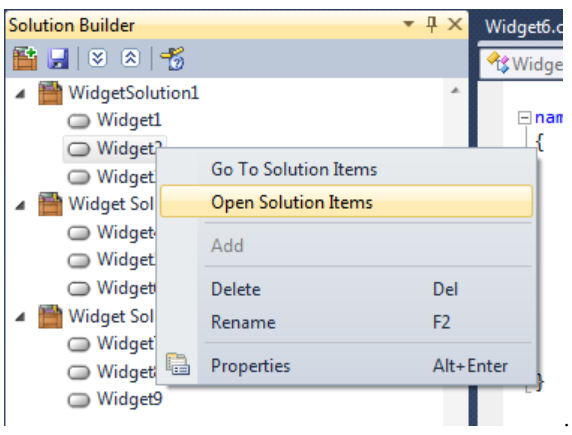


Note: Each element that is created now adds a file to the owning project that is associated to the Widget Solution element.
Your toolkit is smart enough to know which project in Solution Explorer to add the file to.
You can demonstrate this 'smart binding' yourself, by adding multiple instances of the 'Widget Solution', and show that instances of 'Widget' elements within each 'Widget Solution' correctly add their C# code files to the correct projects in the solution.



You can also navigate or traverse from elements in 'Solution Builder' to projects or files in 'Solution Explorer' in two ways:

1. Double-clicking the element in 'Solution Builder'.
2. Right-clicking on the element in 'Solution Builder', and selecting 'Open Solution Items'



Where Are We Now?

We have now used a project template to add a new project in the solution associated with our Widget Solution. We have also used an item template to add a code file to each Widget element.

Users can now discover our pattern through the 'Add New Project' dialog in Visual Studio, and users get a specialized project template unfolded when they instantiate Widget Solutions. Users also get a specialized code class every time they create a new Widget in 'Solution Builder' and we can inject properties from elements in the pattern model into it.

At this point, this is the beginnings of a pattern toolkit that can provide starting templates and guidance for implementing specific kinds of projects and coding patterns, much like other kinds of project templates in Visual Studio do.

As you build out your toolkit in this lab, even though the pattern is intentionally nonsense, you should see that as well as project and item templates for starting points, pattern toolkits can offer a lot more intelligence and functionality throughout the whole lifecycle of the solutions' development. Pattern toolkits give users a far simpler view and interface to work with that affords them the opportunity to focus on the solution being built as a whole, rather than tracking, organizing, managing and modifying projects, source files and configuration files in their solution. The toolkit will be doing that that kind of work automatically more consistently and more reliably for the users.

Next we are going to look at how things can change in the solution as elements are configured in the 'Solution Builder'.

Part 3: Generating Code with T4 Templates

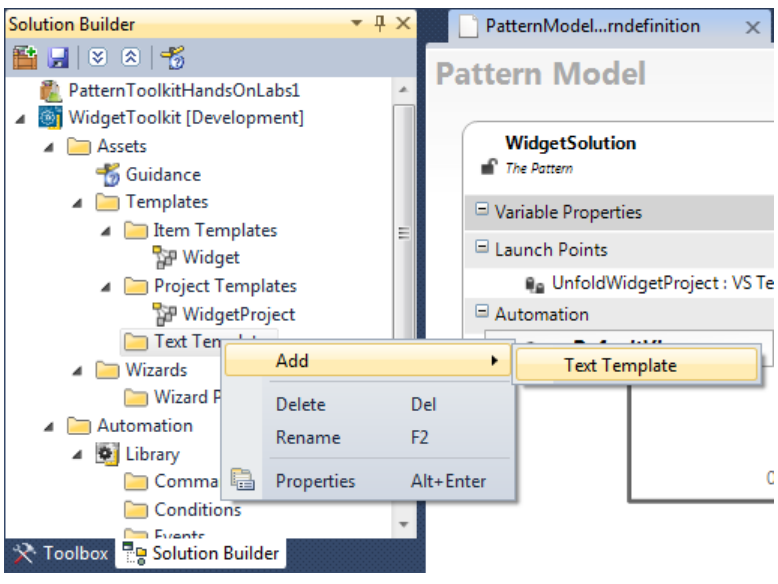
In this section you will add a T4 Text Template to your pattern toolkit in order to generate code that is calculated from properties in your pattern model. We will implement a common code generation pattern which allows for generated code mixed with custom hand written code.

Already in this lab, we have demonstrated unfolding of item templates for each new Widget element added to the Widget Solution in Solution Builder. In this step, we will show you how to create a T4 template for generating the generated code.

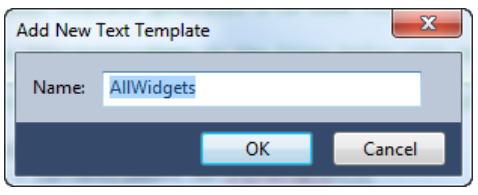
> **Note**: Project and Item templates are a great means to lay down fixed parts of a solution, or to provide starting points in projects that can be evolved manually. However pattern toolkits can take a further step and automate the generation or modification of artifacts in the solution from elements in the pattern model using either data in the pattern model, or data from elsewhere in the environment, so that the solution can be built out as requirements change.

Add a New T4 Text Template

In the Widget Toolkit project, right click on the 'Text Templates' element and choose 'Add | Text Template'.



Name the template 'AllWidgets'.



> **Important**: The name of this text template name is not a naming convention of any kind for code generation classes. It is named like this purely for the sake of illustrating how generated code and custom code can be integrated in a pattern model.

A new T4 template called "AllWidgets.t4" is created, added to your project, and displayed in the Visual Studio document window.

> **Note**: If you need help getting started with T4, see the MSDN documentation at: http://msdn.microsoft.com/en-us/library/bb126445.aspx

Copy the code below, and replace all the text in 'AllWidgets.t4' with it:

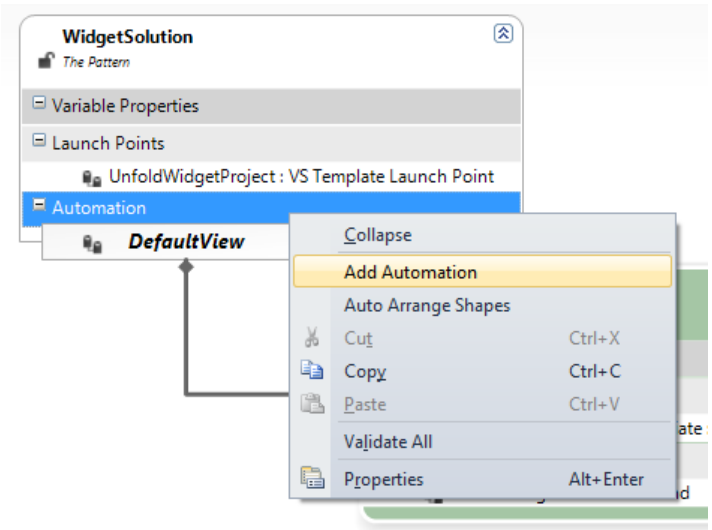
```
<#@ Template Inherits="NuPattern.Library.ModelElementTextTransformation" HostSpecific="True" Debug="True" #>
<#@ ModelElement Type="NuPattern.Runtime.IProductElement" Processor="ModelElementProcessor" #>
<#@ Assembly Name="NuPattern.Runtime.Interfaces.dll" #>
<#@ Import Namespace="NuPattern.Runtime" #>
<#@ Assembly Name="WidgetToolkit.Automation.dll" #>
<#@ Import Namespace="WidgetToolkit" #>
<#@ Import Namespace="System.Linq" #>
<#@ Output extension=".cs" #>
//------------------------------------------------------------------------------
// <auto-generated>
// This code was generated by a tool.
//
// Changes to this file may cause incorrect behavior and will be lost if
// the code is regenerated.
// </auto-generated>
//------------------------------------------------------------------------------
<#
       IDefaultView defaultView = this.Element.As<IWidgetSolution>().DefaultView;
       foreach (IWidget widget in defaultView.Widgets)
       {
              RenderWidgetClass(widget.InstanceName);
       }
#>
<#+
void RenderWidgetClass(string className)
{
#>
public partial class <#=className#>
{
       // Some generated implementation goes here.
}
<#+
}
#>
```

Save and close the 'AllWidgets.t4' file.

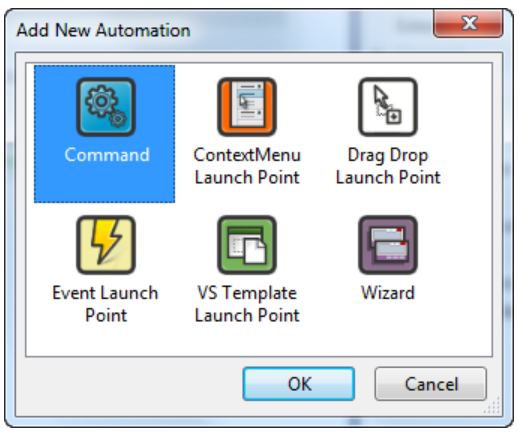
Add a Command for Running the T4 Template

We are going to add a T4 code generation command to the 'WidgetSolution' element of the pattern model, then two launch points to execute the same command: one to execute code generation on build, and the other to execute code generation when the user right-clicks on a context menu.

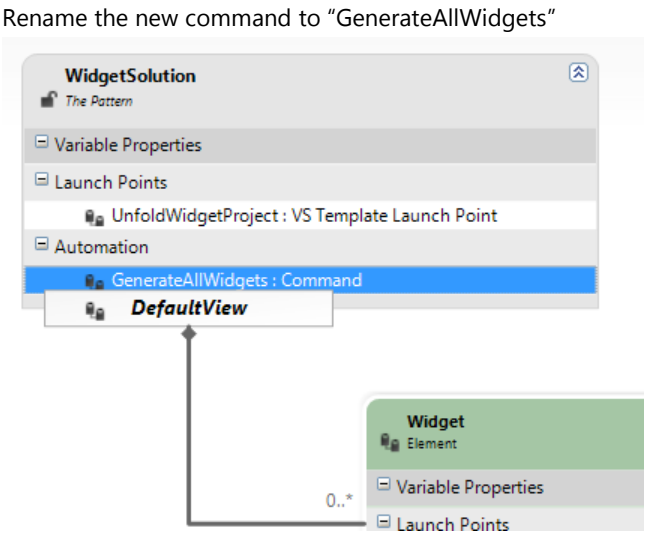
Right-click on the 'Automation' compartment of the 'WidgetSolution' shape and choose 'Add Automation'.



In the 'Add New Automation' dialog, choose 'Command' and click OK.



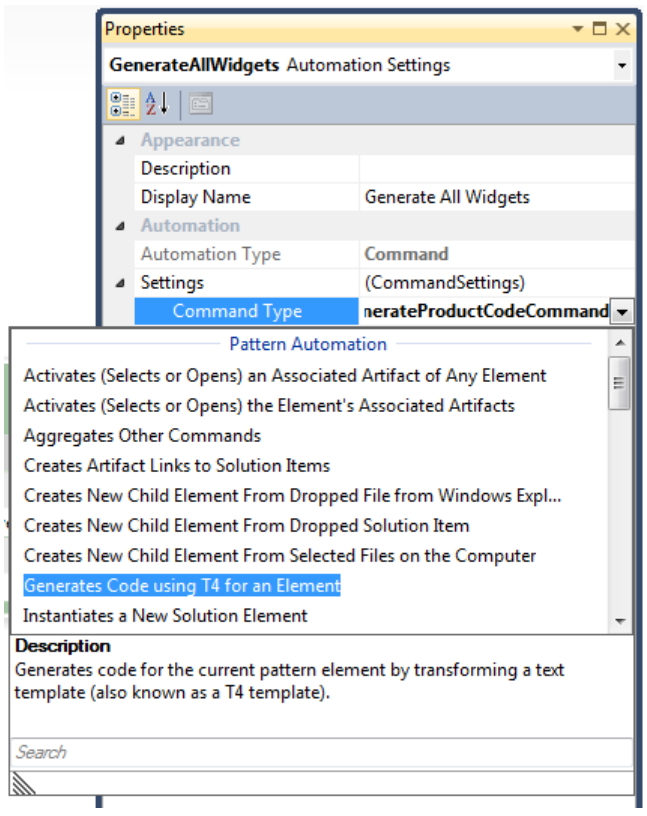
Rename the new command to "GenerateAllWidgets"



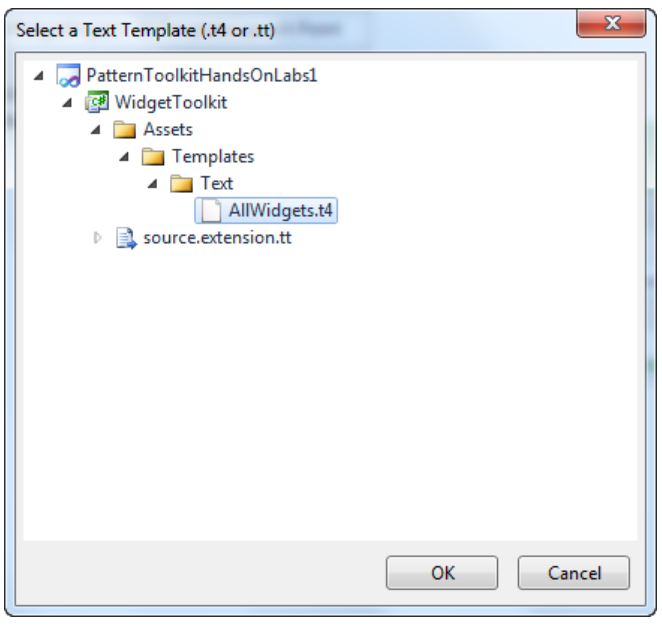
Configure the Command for Running the T4 Template

With the 'GenerateAllWidgets' command selected, open the Properties window (**F4**), and expand the 'Settings' property.

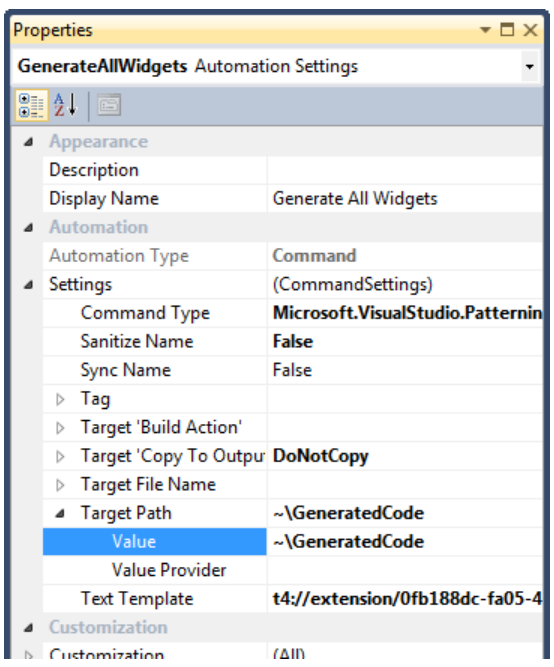
In the 'Command Type' property click the drop down and choose "Generates Code using T4 for an Element".



In the 'Text Template' property, click the ellipsis button and choose the "WidgetPartialClass.t4" template we just created.



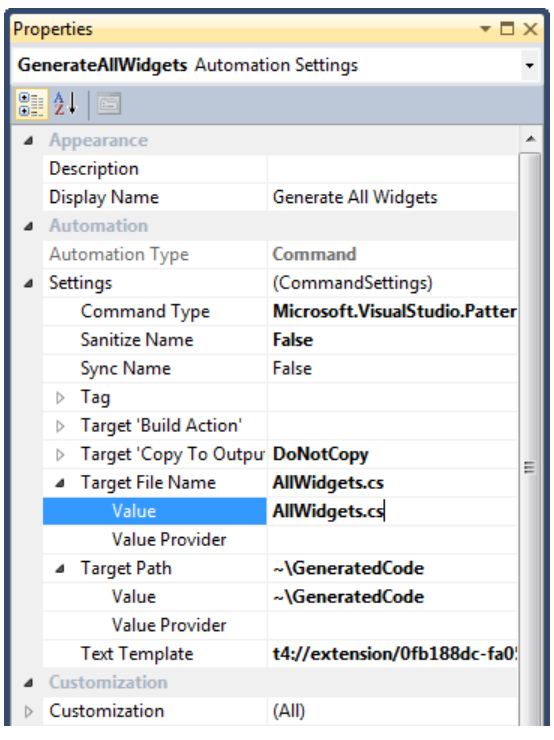
Expand the 'Target Path' property, and enter "~\GeneratedCode" in the 'Value' property.



> **Note**: The 'Target Path' property determines where the generated file goes in the user's solution (i.e. which project or folder). We want the file to be generated in a sub folder of the project called "GeneratedCode", within the same project created for the same Widget Solution. The '~\' character we put in front of 'GeneratedCode' path instructs the command to traverse across from the current element in 'Solution Builder' to the first associated artifact for it in 'Solution Explorer'. In the case of this particular pattern model as it stands right now, that should be to the Widget project that was unfolded when the 'Widget Solution' element was created. The remaining part of the 'Target Path' creates whatever folder structure is specified. In this case it creates a sub folder called 'GeneratedCode' in that project.

Expand the 'Target File Name' property, and enter "AllWidgets.cs" in the value property.

> **Note**: This property determines the name of the file that will be generated.



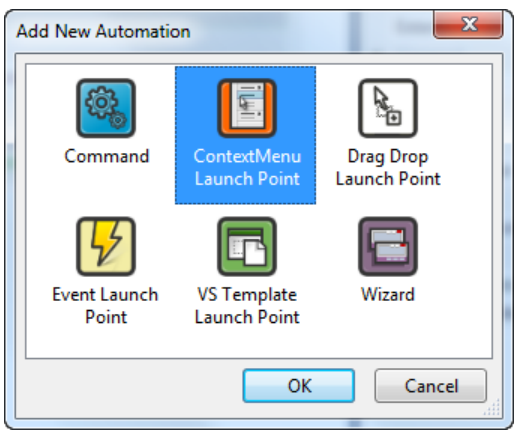
> **Note**: The previous two properties required you to expand them in the properties window before entering a value in the 'Value' property. Many values in a pattern model allow you to either: type a fixed value or expression, or select a 'ValueProvider'.
> A 'ValueProvider' is a custom class provided by a toolkit (Automation Library) that calculates the value of the property dynamically at the time the user is using the toolkit. With ValueProviders you can access any of the properties of any element in the pattern model, and even reach out of Visual Studio into other services on the desktop using data in the pattern model. It is a very powerful mechanism to drive automation for the user.
>
> **Advanced Tip**: For example, if you want to get the current project's root namespace, or generate a GUID. You can write your own value providers, or use ones already provided.

Create a Launch Point to Trigger Code Generation With a Menu

The next step is to create launch points for executing the T4 template command.

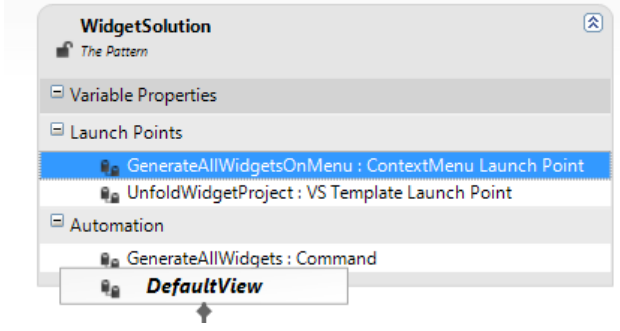
Right-click on the 'Launch Points' compartment of the 'WidgetSolution' element in the Pattern Model.

Click on 'Add Automation', and choose 'Context Menu Launch Point' this time.



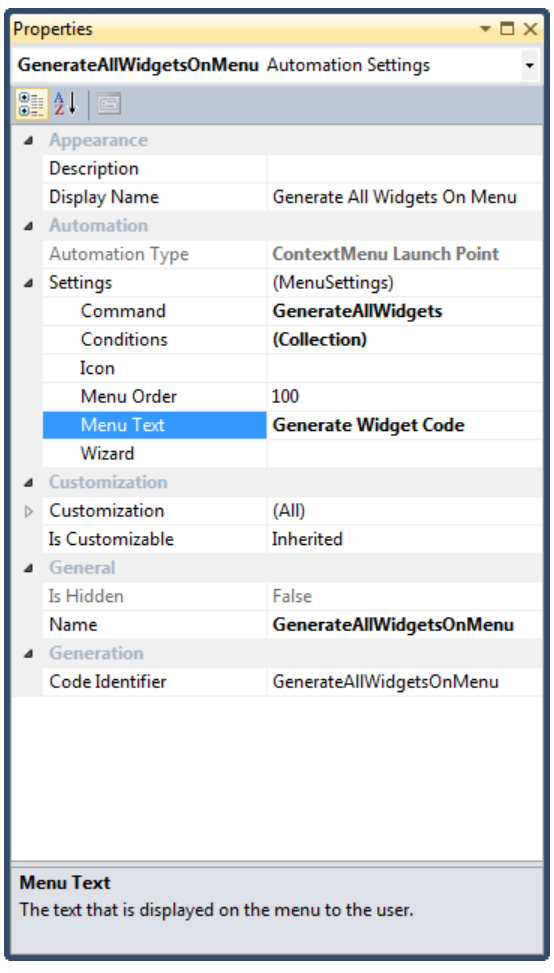
Rename the launch point to "GenerateAllWidgetsOnMenu"

> **Note**: This name follows the naming convention of <CommandName><LaunchEvent> because we are creating a launch point for the 'GenerateAllWidgets' command when the user clicks on a context menu, which is triggered by the 'OnMenu' event.



Select the launch point, and open the Properties window (**F4**).

Expand the 'Settings' property, and set the 'Command' property to "GenerateAllWidgets", and set the 'Menu Text' property to "Generate Widget Code"



> **Advanced Tip**: It is also possible to configure any 'Conditions' that must be evaluated before any menu is displayed and the selected command is executed by the launch point.
> You can also set up a Wizard to run before the command is executed if you want to gather data from the user before the command runs.
> See the topics in the guidance for Pattern Toolkits for instructions on how to create and configure conditions and wizards.

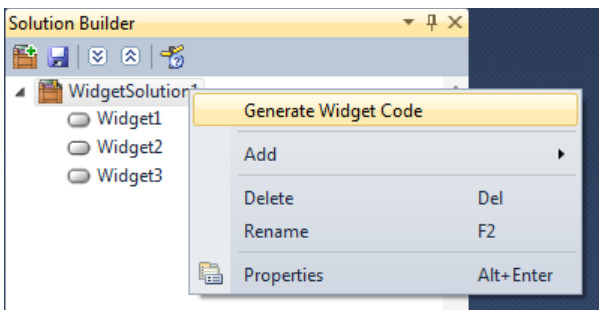
Build and Test the Context Menu Launch Point

Build and Run the toolkit project (**CTRL + F5**).

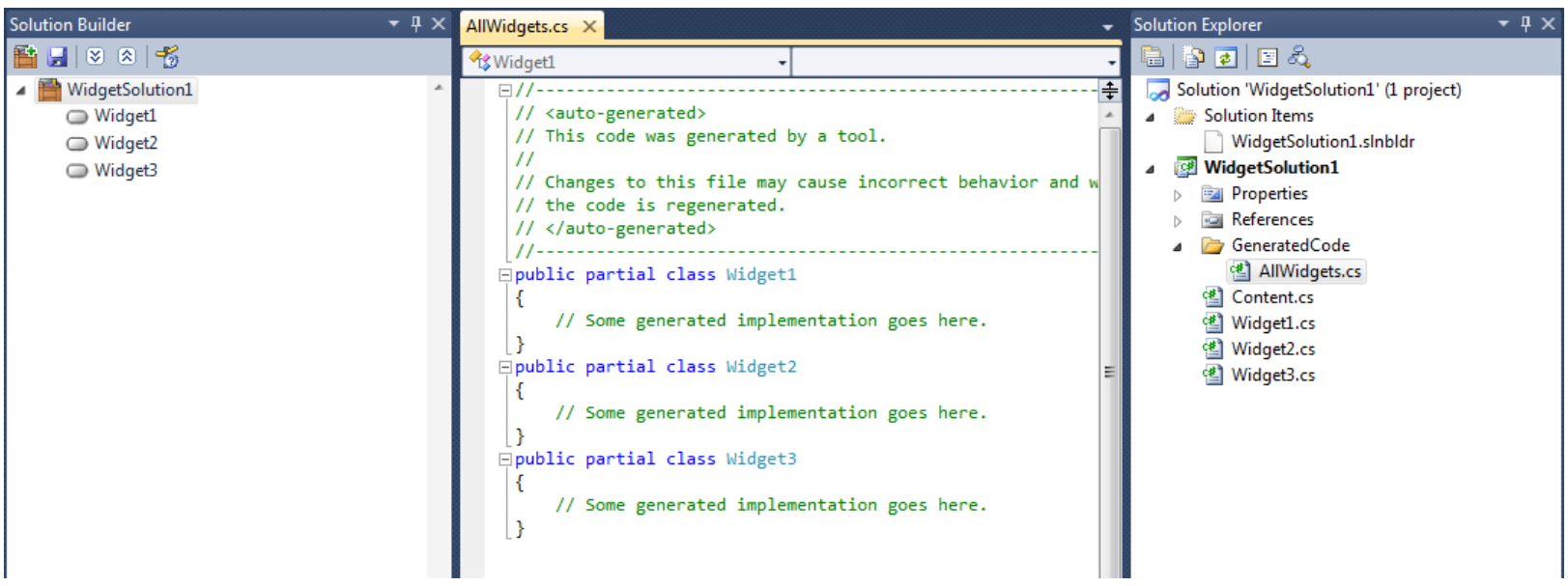
In the Experimental Instance of Visual Studio that starts up, you can create a new 'Widget Solution' in the 'Add New Project' dialog ('File | Add New Project') or you can use the previous solution from the last test cycle.

Either way, add some new instances of the 'Widget' element in 'Solution Builder'.

Right-click on a 'Widget Solution' in 'Solution Builder' and you will now see the new context menu.



Click the 'Generate Widget Code' menu. You will see a new folder created in your project called 'GeneratedCode', with a class file in that folder called "AllWidgets.cs". If you open that class file, you will see the code below:



These generated partial classes do not have any implementation yet. When you determine what you want widgets to do, you can add that code to the T4 template and it will be generated here.

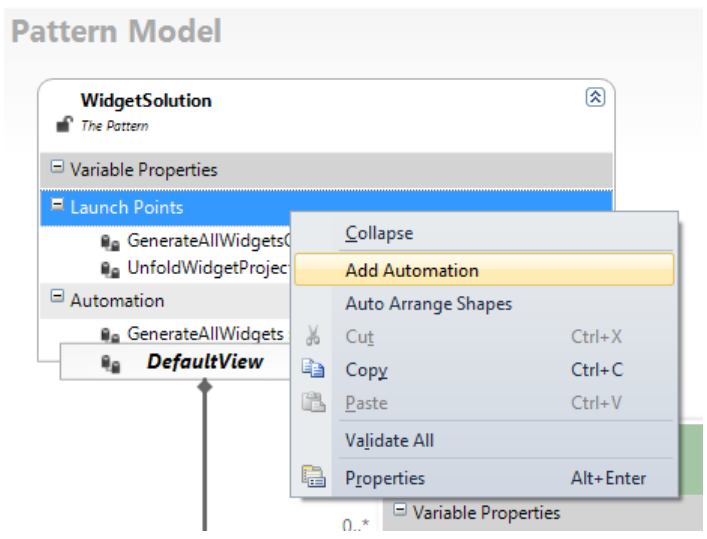
> **Advanced Tip**: You can create additional properties on each Widget element in the pattern model, and then have the T4 template read those properties to calculate new generated code, either with conditional statements or use the values of the properties in content of the generated text. See the T4 Text Template Reference for more details on that syntax.

Create a Launch Point to Trigger Code Generation On Project Build

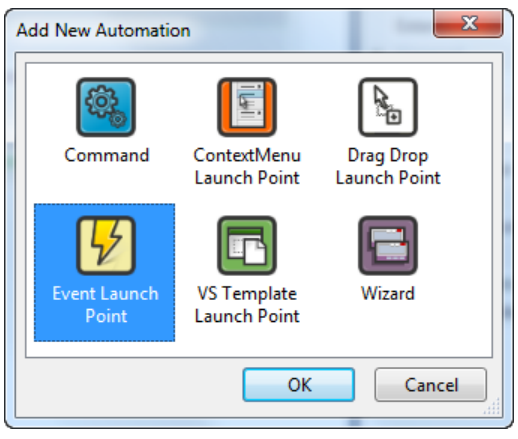
We are now going to trigger the code generation to run when the user triggers a build in the solution. This code generation strategy ensures that whenever the user builds their solution, all the generated code is up to date. This is particularly important when that generated code uses properties of the pattern model to determine what code is generated.

Close and Save the Visual Studio Experimental Instance.

Back in your Widget Toolkit project, right-click on the 'Launch Points' compartment of the 'WidgetSolution', and click on 'Add Automation'.



In the 'Add New Automation' dialog, choose 'Event Launch Point'.

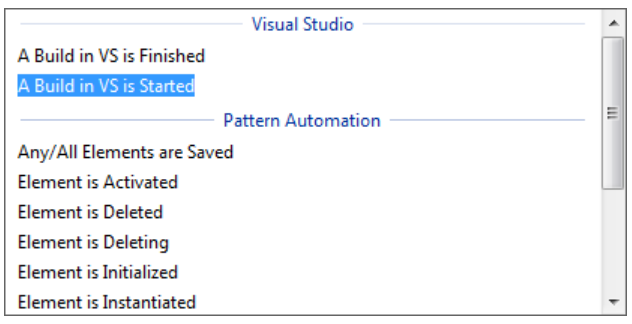


Rename the launch point to "GenerateAllWidgetsOnBuild"

> **Tip**: Remember that you can just start typing the new name if the launch point is already selected. So after you click OK you can immediately start typing to rename the launch point.

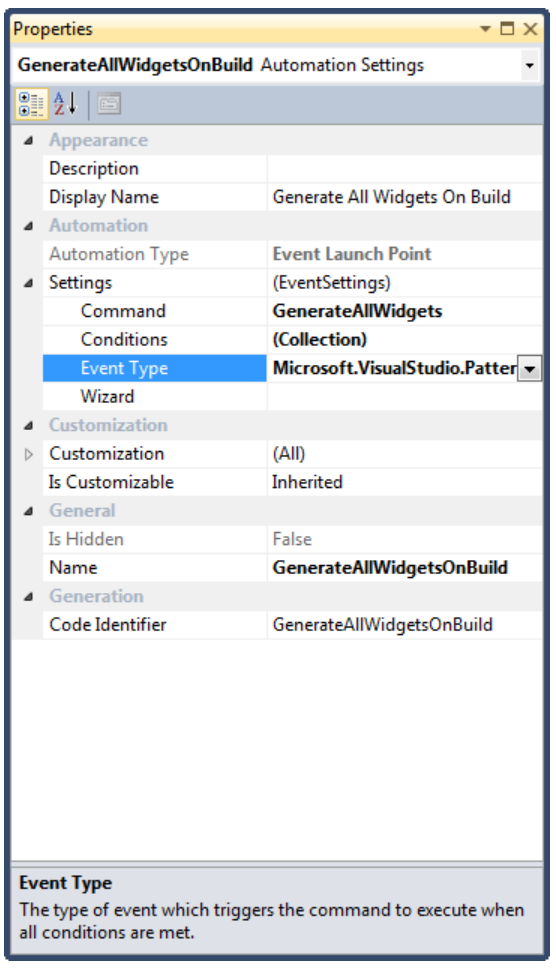
Expand the 'Settings' property, and in the 'Command' property, use the drop-down to select the 'GenerateAllWidgets' command.

Click the drop-down in the 'Event Type' property, and select the "A Build in VS is Started" event.



> **Note**: This event will execute the command when the user starts a build in Visual Studio.

The properties for the Event Launch Point should now look like this:



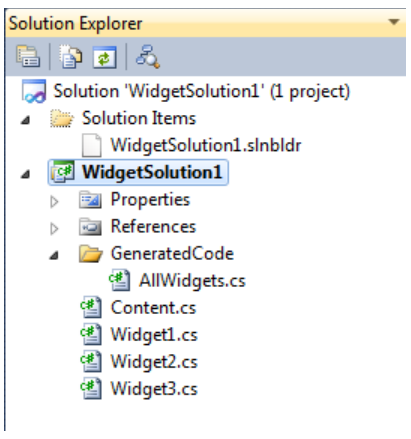
Build and Test the Build Event Launch Point

Build and Run the toolkit project (**CTRL + F5**).

In the Experimental Instance of Visual Studio that starts up, this time create a new 'Widget Solution' in the 'Add New Project' dialog ('File | Add New Project') or you can use the previous solution from the last test cycle, but delete the 'GeneratedCode' folder in the project.

Execute a build (**F6**).

When the build is complete you should see the 'GeneratedCode' folder added to the project, along with the 'AllWidgets.cs' file.



What Have We Got Now?

At this point we now have a toolkit that unfolds a project with files that allow a user to custom code their widget solution. Whether these are C# code files, XML configuration files, powershell scripts, database scripts, to name a few artifacts that are written to implement any given solution. Each of these files is located in the right place, and is named accordingly. In many solutions the names of these kinds of files will be defined by some naming convention that the toolkit can now apply, and enforce for the user automatically.

As well as that, the toolkit is now generating additional code for the user that could adapt itself based upon the properties and state of the Widgets that the user configures in Solution Builder. This is the kind of predictable code that developers in particular find very tedious to write, and is often very prone to manual copy & paste errors. Now the toolkit takes care of that for them automatically, so it is always well written, always correct and integrated with any hand crafted code.

This combination of unfolding custom code placeholders (using Item Templates), and generated code (using T4 Text Templates), tied together with partial classes is a very powerful code generation pattern that many pattern toolkits take advantage of. The users of the toolkit no longer need to worry about code/scripts/configuration files that can be predetermined from a simple selection of property values. They just configure the elements in Solution Builder and build their solution! If they need some custom behavior in their particular solution then they simply open the custom code files and add or extend the behavior. They never need to see or care about maintaining the generated code. But it is always there in the project if they need to.

By executing code generation on build, the generated code is always up to date and it ensures that if the text template code generator is using data from the widget elements in Solution Builder, or other services inside or outside Visual Studio, that the code is always up to date at build time.

Providing a menu as well that generates the code manually at any time, often gives users a level of comfort that they are still in control of code generated into their solution.

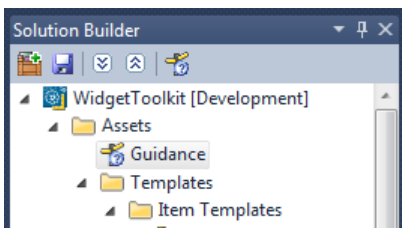
Part 4: Create Guidance

When you distribute a pattern toolkit you will likely want to provide with it some guidance to help users understand the concepts behind the toolkit as well as how to use it in Solution Builder.

A Pattern Toolkit project helps you create guidance easily that will appear integrated within Visual Studio when users are working on elements of your pattern in 'Solution Builder'.

Close and Save the Visual Studio Experimental Instance.

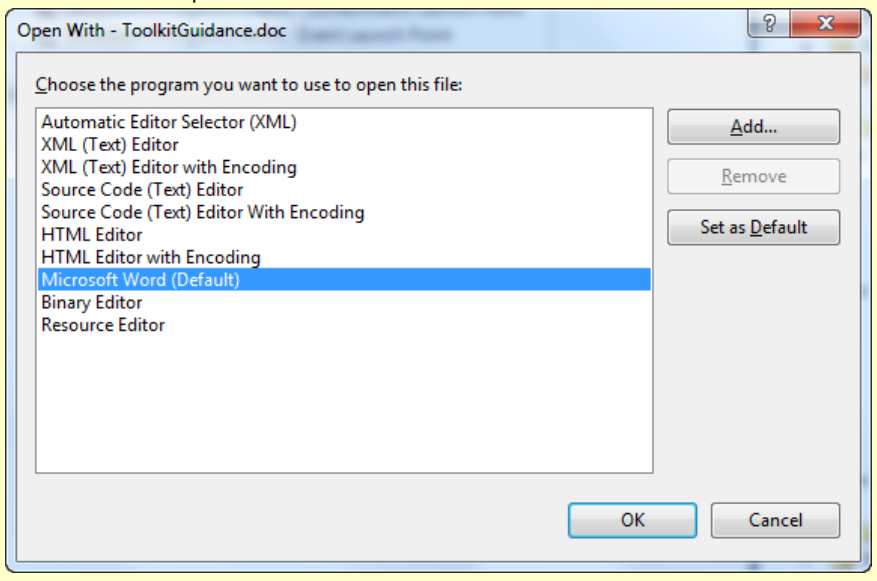
Back in your Widget toolkit project, **double-click** on the 'Guidance' element in 'Solution Builder'.



A Microsoft Word document should open on the desktop.

> **Note**: If an XML document opens in the Visual Studio instead, then follow these instructions to configure Visual Studio to launch Microsoft Word when you open a *.docx file.

- Locate the Word document in the solution. It will be in the Project '(WidgetToolkit)/Assets/Guidance/ToolkitGuidance.docx'.
- Right click and choose 'Open With…'
- Click 'Microsoft Word' in the dialog box, and then click the 'Set as Default' button.
- Click OK to open the document.



Examine the Guidance Document

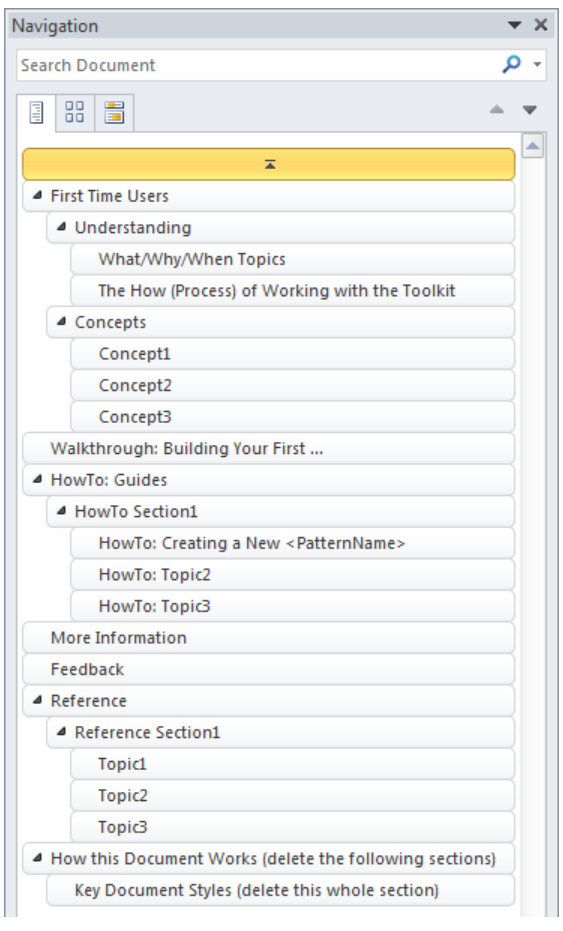
This Word document is a template for creating guidance for your toolkit.

> **Note**: The following instructions presume you have Microsoft Word 2010 installed. If you have another version of Word installed please consult the documentation for equivalent functionality in your version of Word.

Navigating a guidance document is easier if you show the 'Navigation Pane'.

Click on the 'View' tab in the ribbon, and check the 'Navigation Pane' checkbox.

If you look at the Navigation pane, the document illustrates a suggested pattern for creating guidance. There's a section for 'First Time Users', 'Walkthroughs', 'How To' guides, and a 'Reference' section. At the end there is a section explaining how the document template works.



> **Note**: You are free to use any, none, or all of these sections as described in the document, or create your own structure. This is just a guide to structuring guidance for your toolkit in a format that users will expect.

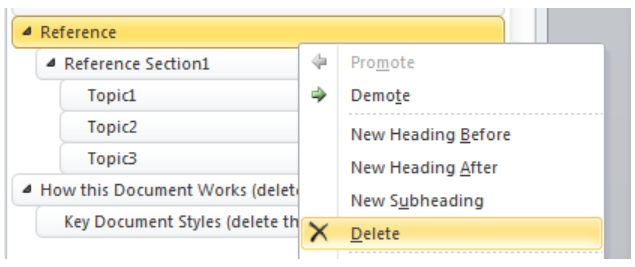
Edit the Guidance Document

For the purpose of this lab we'll reduce the number of sections in the document and add some fake content (these are Widgets, after all.)

Delete all sections except 'First Time Users', 'Concept's, and 'Feedback'.

The simplest way to remove the sections is to right click in the Navigation pane and click 'Delete'.

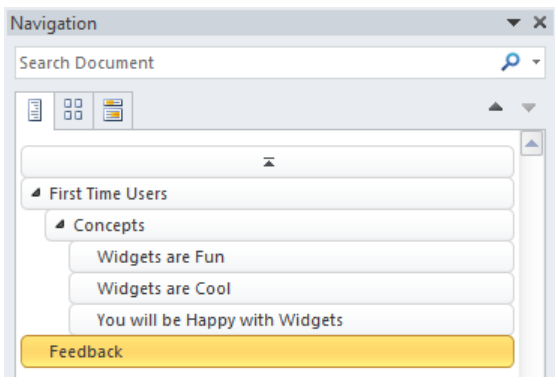
> **Note**: If you have an earlier version of Word you may have to select and delete the text in the document.



Then update the individual concepts with these headings.

- "Widgets are Fun"
- "Widgets are Cool"
- "You will be Happy with Widgets"

When you are finished, the Navigation pane in Word should look like this:



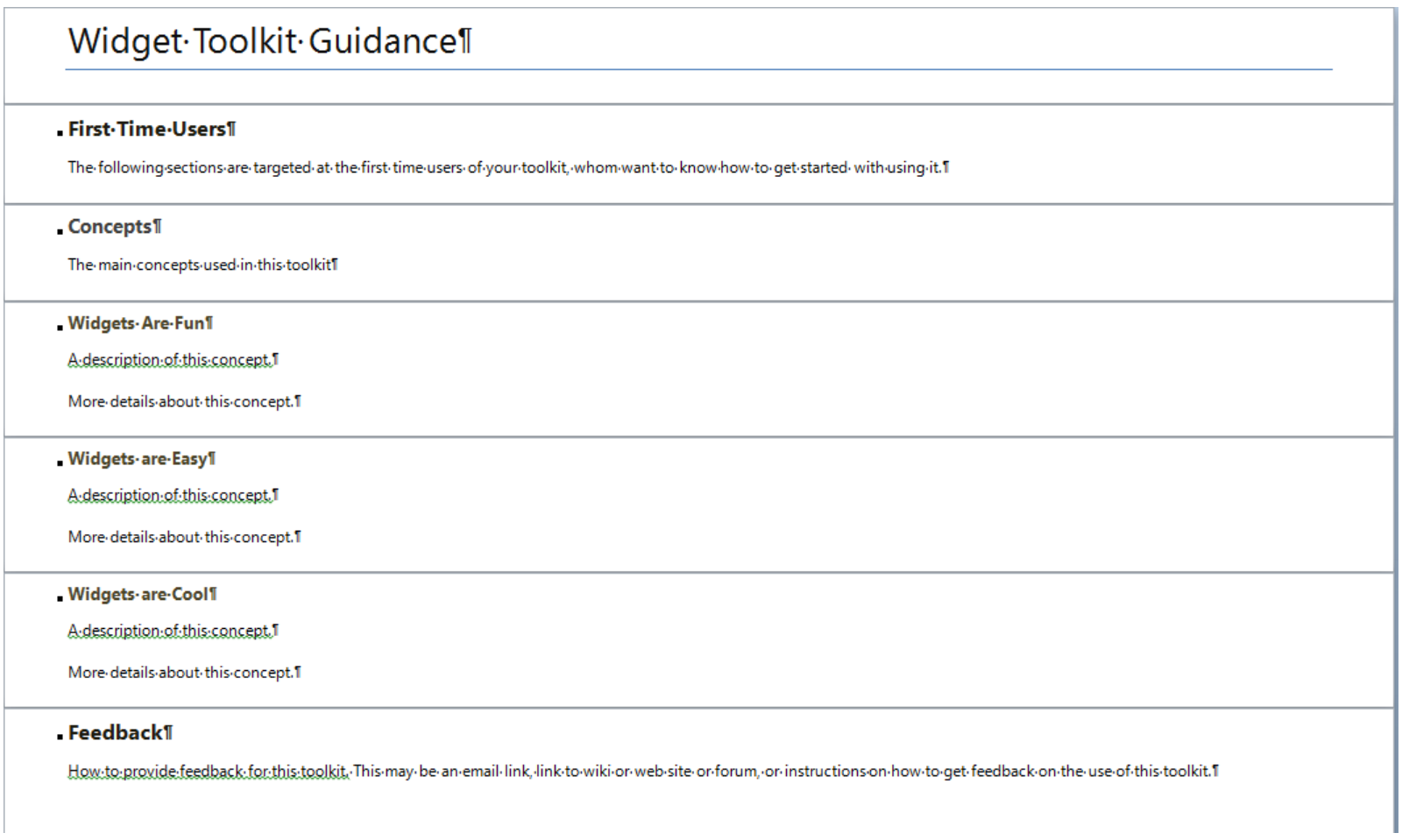
Delete the remaining yellow sections in the text.

Update the title of the document to say "Widget Toolkit Guidance", and delete the additional comments.

Add one additional paragraph to each of the sections under 'Concepts'. It is important to have at least two paragraphs in these sections.

> **Note**: When the guidance build process runs (see next step in the lab), each page in this document will be transformed into a single page guidance document that will appear to the user in the 'Guidance Browser' tool window. Every page in your guidance document that has more than one paragraph will result in a hierarchical guidance topic in the 'Guidance Workflow Explorer' navigation, and its content displayed in the 'Guidance Browser' window. Any page in your guidance document that has only one sentence will result in a hierarchical guidance topic in the 'Guidance Workflow Explorer' navigation only, and no content displayed in the 'Guidance Browser' window.
> Keep these things in mind when creating guidance:
>
> - The heading of each page in this document will be used to name the workflow topic in the 'Guidance Workflow Explorer'.
> - Use the built in heading styles Heading1 - Heading 6.
> - The content of each page (exceeding one paragraph) in this document will result in a page of guidance being displayed to the user in the 'Guidance Browser' window.
> - Any guidance that is more than one page will be truncated at the end of the first page. (Each guidance topic should be short enough to fit on a single page. Longer guidance topics result in poor usability for the reader. If you need more than one page to display the content, consider breaking it up into multiple steps, using headings.

The finished document should look like this (showing paragraph marks ¶ for clarity):



Widget·Toolkit·Guidance¶

First·Time·Users¶

The·following·sections·are·targeted·at·the·first·time·users·of·your·toolkit,·whom·want·to·know·how·to·get·started·with·using·it.¶

Concepts¶

The·main·concepts·used·in·this·toolkit¶

Widgets·Are·Fun¶

A·description·of·this·concept.¶

More·details·about·this·concept.¶

Widgets·are·Easy¶

A·description·of·this·concept.¶

More·details·about·this·concept.¶

Widgets·are·Cool¶

A·description·of·this·concept.¶

More·details·about·this·concept.¶

Feedback¶

How·to·provide·feedback·for·this·toolkit.·This·may·be·an·email·link,·link·to·wiki·or·web·site·or·forum,·or·instructions·on·how·to·get·feedback·on·the·use·of·this·toolkit.¶

Close and save the document.

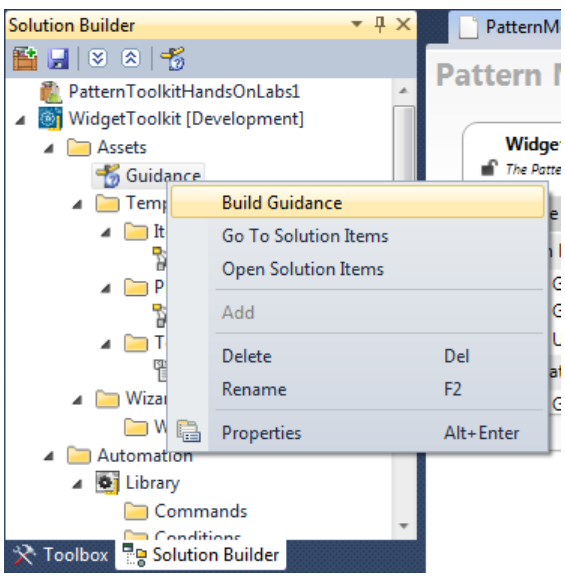
Build the Guidance

Guidance requires a build step that is separate from the regular Visual Studio build.

> **IMPORTANT**: Running the Visual Studio build will not execute a build of the guidance.

Close the guidance document if it is still open in Word.

In the Widget toolkit project, right-click on the 'Guidance' element in 'Solution Builder', and select 'Build Guidance'.



Building the guidance may take some time, and if any errors are found in the guidance document they will be displayed in the 'Error List' window.

> **Note**: In 'Solution Explorer' if you collapse the 'Assets' folder and scroll down to the 'GeneratedCode' folder. You will see that several .mht files have been added to the project, along with a class file called 'GuidanceWorkflow.cs'. The class files controls what the user will see in the 'Guidance Workflow Explorer', and the .mht files are the individual guidance topics the user will see in the 'Guidance Browser' window.
> **Note**: You do not see one .mht file for every section in the guidance document you were just editing, because the build process only creates .mht files for guidance sections that have more than one sentence or paragraph.
>
> **Warning**: There have been several issues reported when trying to build guidance with some installations of the 64 bit version of Office 2010. If you are experiencing these issues when building your guidance, please see the topic entitled "Error, building guidance with 64bit versions of Microsoft Word." in the pattern toolkit guidance of your toolkit project for a resolution.

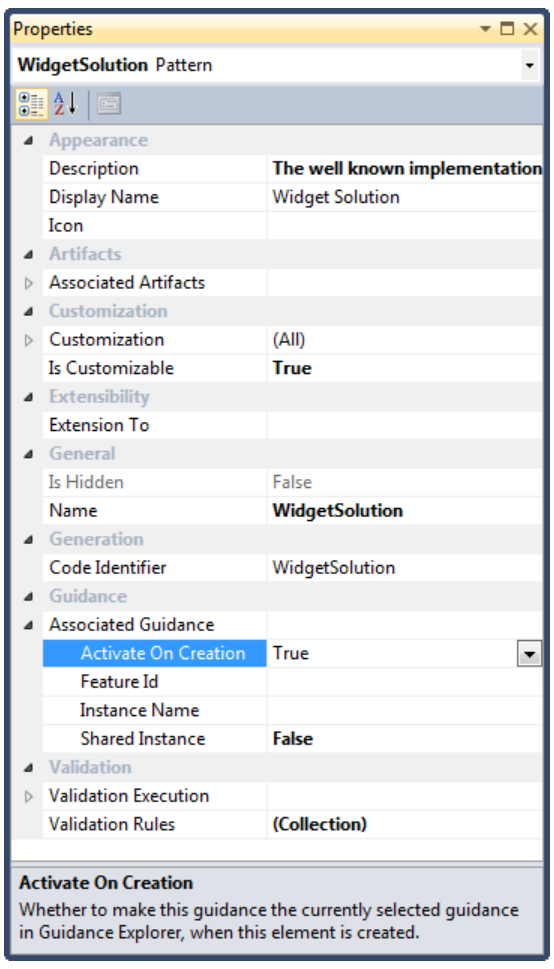
Associate the Guidance with the Pattern

The last step in creating guidance is to associate this guidance with one or more elements in the Pattern Model.

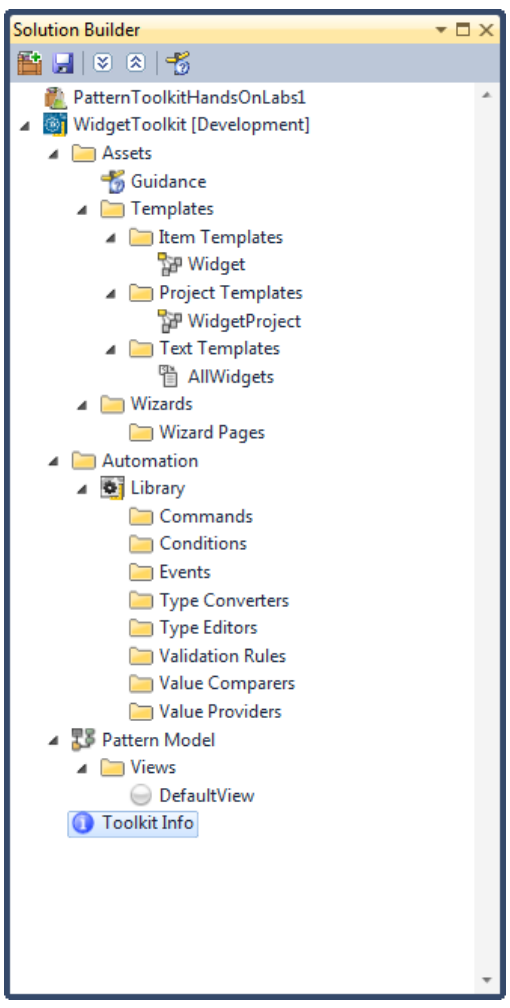
Typically, in many toolkits guidance will usually only be associated with the pattern element so that the guidance is instantiated and displayed when the user first creates an instance of your pattern in 'Solution Builder'. But there may be times when you want guidance to launch for individual elements in the pattern model.

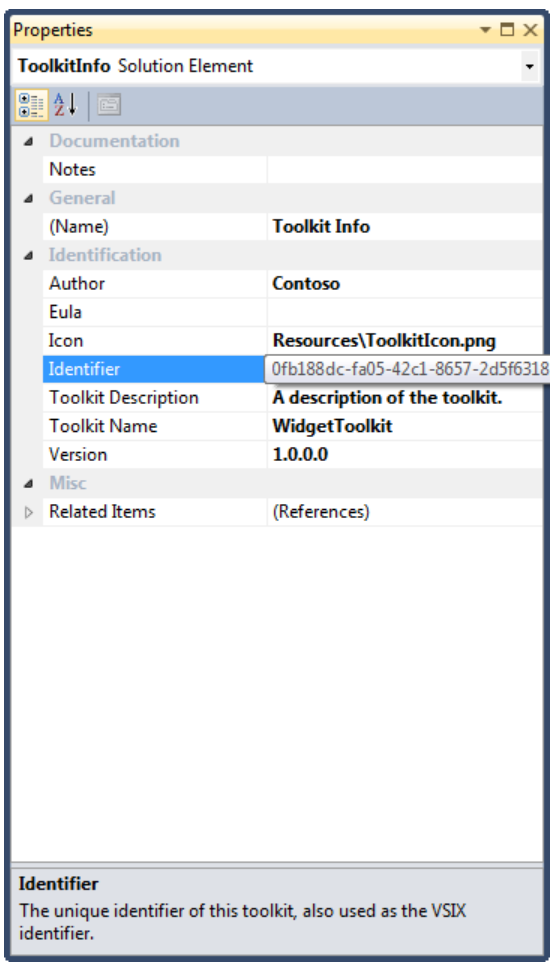
> **Note**: In the current version of the 'NuPattern Toolkit Builder', you can only have one guidance document per toolkit project.

Open the Pattern Model, and click on the 'WidgetSolution' element, in the Properties Window (**F4**), expand the 'Associated Guidance' property.



We must set the 'Feature Id' property. The correct 'Feature Id' for your toolkit can be found by clicking on the 'Toolkit Info' element in 'Solution Builder' and copying the 'Identifier' property.



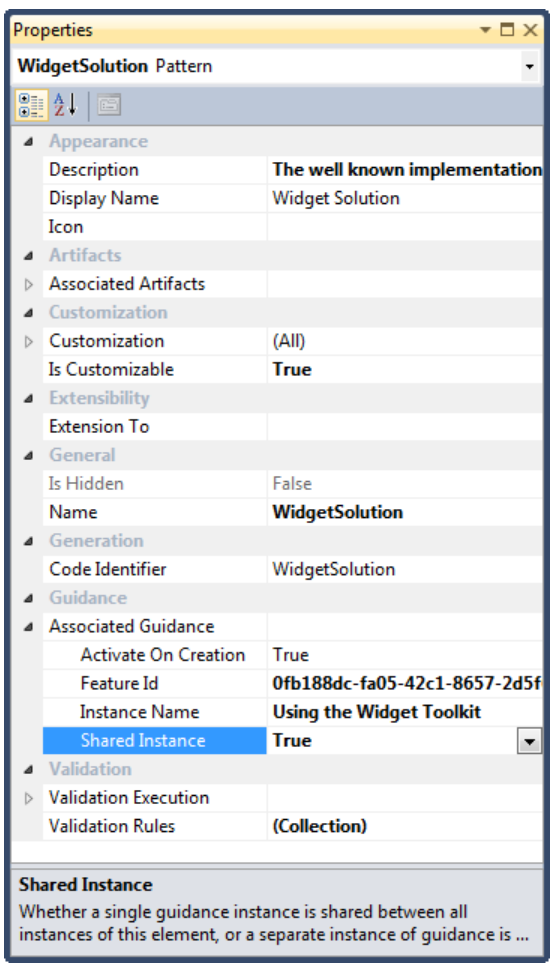


Copy the value of the 'Identifier' above, click on the 'WidgetSolution' element again, and paste the value into the 'Feature Id' property of the 'Associated Guidance' property.

Set the 'Instance Name' property to "Using the Widget Toolkit". That is the name that will appear in the 'Guidance Workflow Explorer' for the user.

Finally, set the 'Shared Instance' property to "true". This will ensure that only one instance of this guidance is created for all instances of 'Widget Solutions'.

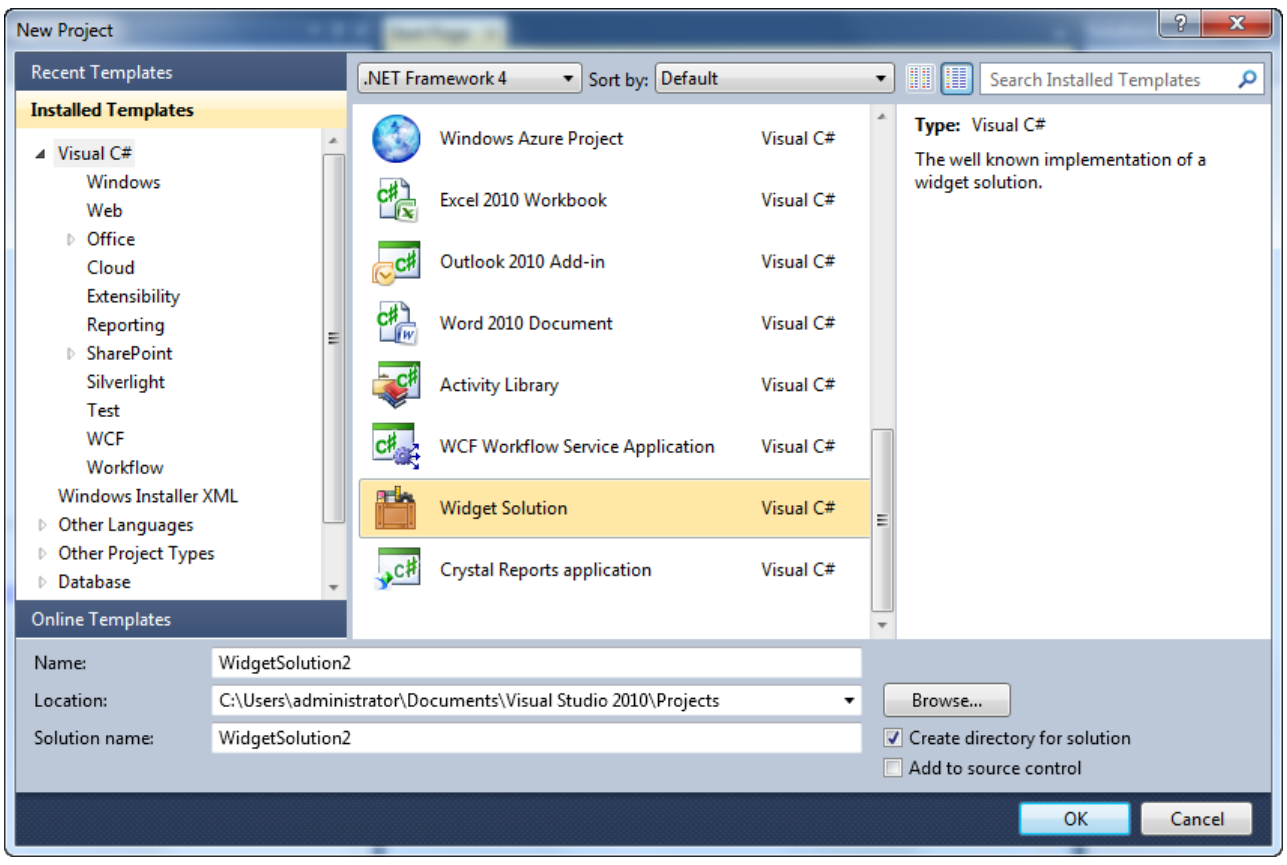
The properties for the pattern element should look like this:



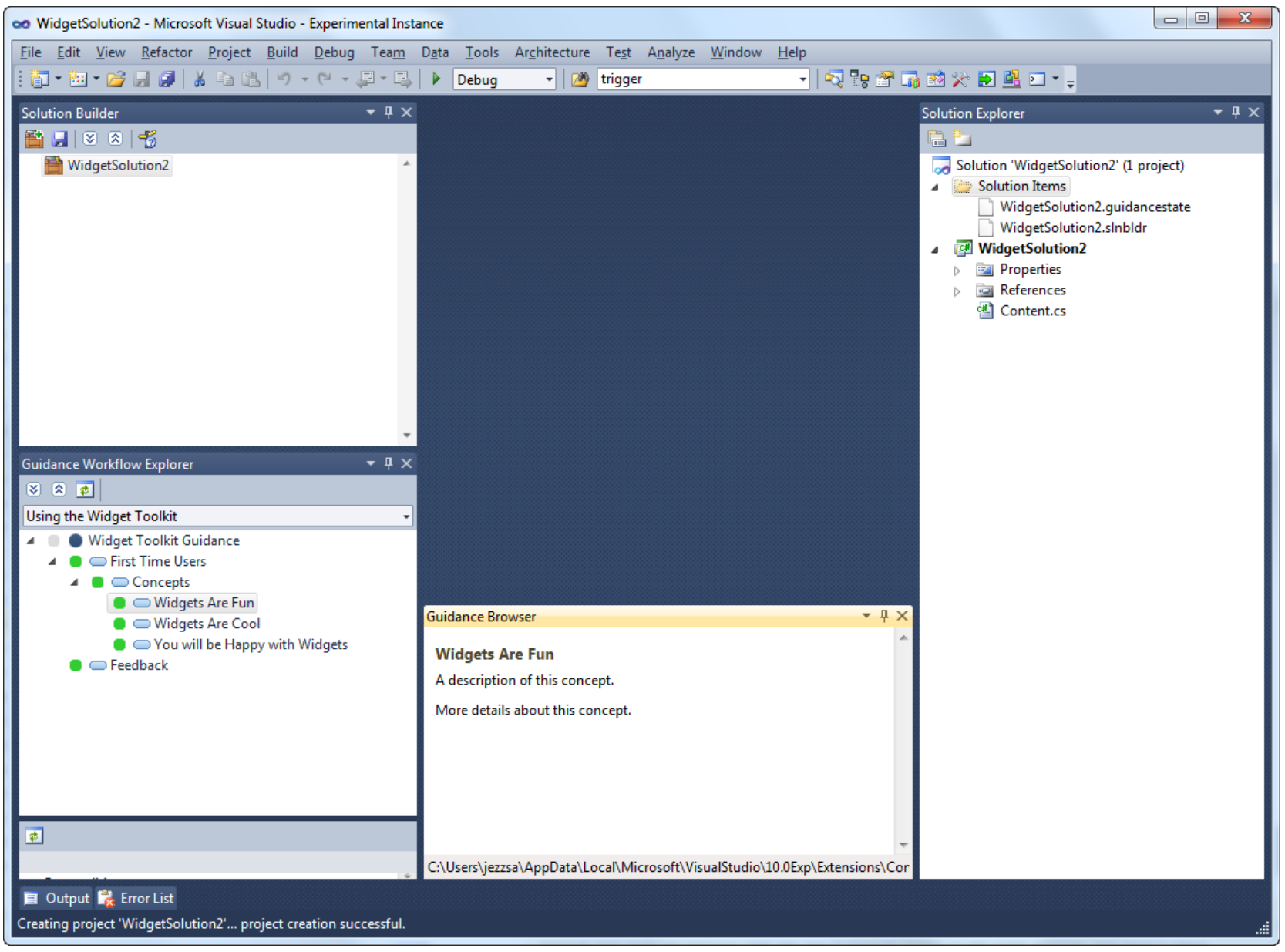
Build and Test the Guidance

Build and Run the toolkit project (**CTRL + F5**).

In the Experimental Instance of Visual Studio that starts up, this time you must create a new 'Widget Solution' in the 'Add New Project' dialog ('File | Add New Project').



Now, you will see the 'Guidance Workflow Explorer' window appear along with the 'Guidance Browser' window, and your first guidance topic for your toolkit displayed in them.



> **Note**: In order to see both the 'Guidance Workflow Explorer' and the 'Solution Builder' tool windows are the same time (as you see above) simply click the tab of either window and drag the window over the top or bottom halves of the other window, and drop it there.

Click through the guidance topics to display the different topics you created in the guidance document.

Close and Save the Visual Studio Experimental Instance.

Hands-On Lab 1 Review

Here are the topics we have covered in this first hands-on-lab:

- Creating a Pattern Toolkit project.
- Building and testing Pattern Toolkits (in the Experimental Instance of Visual Studio).
- Working with the 'Pattern Model' to define variable architectural concepts.
- Adding Automation to speed creation of solution artifacts predictably and consistently.
- Adding and configuring Project Templates, and Item Templates.
- Enabling Navigation between 'Solution Builder' and 'Solution Explorer'
- Creating Code Generators (T4 Text Templates), and triggering them from user gestures.
- Creating and configuring integrated Guidance.

Most of these topics that we have covered in this lab will be useful in building any kind of pattern toolkit.

Any pattern toolkit you build will likely need to include many of the elements of Pattern Modeling, Automation, Guidance, and many of the Assets types that we have covered here.

The hands-on-lab was intended to give you just the basics of building a simple toolkit to illustrate some of the power of pattern toolkits, and providing automation for those building solutions.

You may have noticed that your pattern toolkit can do rather a lot for a developer with only a little configuration in the pattern model beyond what we have covered here. With realistic content in the project, item and text templates you could easily adapt this toolkit into something very useful for use on a project now where there are identified patterns of implementation.

In the next hands-on-lab, we are going to dig a little deeper into some of the other important elements of pattern toolkits, that round out the functionality of a pattern toolkit.

Hands-On Lab: Building Better Pattern Toolkits

This lab contains a number of detailed walkthroughs for creating richer pattern toolkits. Some of the lessons can be completed out of order, but in general, each lesson builds on the previous lesson.

The overall scenario is to build the beginnings of a pattern toolkit for implementing an ASP.NET MVC 2 application using implementation patterns specifically determined by your solution development organization. This is a specific implementation of ASP.NET MVC, and one that your architects and IT Pros have pre-determined is best for delivery from your organization.

> **Note**: This implementation may or may not agree with one you may want to build for one of your real projects. It is not important for the purposes of this lab that we define patterns that everyone in the industry agrees upon. Rather that we focus on demonstrating how you can build a pattern toolkit that implements an arbitrary pattern that you would define yourself. We just need to make this scenario a little more realistic than the last lab.
>
> **Important:** As you will see in the labs, part of the power of pattern toolkits is that anyone can build a pattern toolkit for any specific architecture or solution they want, and refine it for their specific needs, requirements, constraints, technology choices etc.
> As you may see in later labs, as well as building your own pattern toolkit, you can pick up an existing pattern toolkit that you like, and tailor the way it works for you or your organization, without starting from scratch.

As noted in the first topic of this guidance, we need to select *a* specific scenario here for illustrative purposes of this lab. This scenario just happens to be more applicable to Software Developers writing code, than for IT Pros deploying a solution. But we do not intend to exclude any disciplines in IT solution development and deployment. This lab can be just as easily adapted to illustrate deployment patterns just as well as development patterns, where the artifacts may instead be power shell scripts or XML configuration files, and the automation may drive deployment frameworks and services, rather than tooling inside the Visual Studio project system.

Major topics of this lab are:

- Modeling Variability in the Pattern Model
- Extending and composing Pattern Toolkits with Extension Points.
- Adding Wizards for enhancing usability.
- Validating user input with built-in validators and custom validators
- Working with Project templates and Item templates
- Creating code generators
- Configuring built-in commands and launch points
- Creating wizards - TBD
- Using drag and drop events to allow drag and drop from ‘Solution Explorer’ to ‘Solution Builder’ - TBD
- Deleting solution items when elements are deleted - TBD
- Creating custom commands - TBD
- Creating and setting conditions on commands - TBD
- Enhancing property grid input with type converters - TBD
- Customizing existing toolkits and controlling customization of your toolkits - TBD
- Adding multiple views of a pattern - TBD
- Integration with domain specific languages created with DSL Tools - TBD
- Executing commands from hyperlinks in the guidance - TBD

Part 1: Modeling Variability in a Pattern

The first step to creating a pattern toolkit is to create a pattern model that the toolkit will operate on, and use as its *schema*; its metadata description. The pattern model should represent the structure and variable parts of your pattern; the elements of the pattern that the users would configure if they could, that may also include structural elements for arranging and grouping of those elements.

> **Important:** The pattern model is not meant to be used for modeling all the elements of your pattern or solution; it need only represent the variability in the pattern. It is meant to provide an abstraction (simplification) of your implementation pattern focused on providing configuration of the variance of your implementation pattern, in terms of how users understand the solution they are creating.

In this lab we will be working with a pattern model for creating a specific implementation of ASP.NET MVC 2 web applications. Since a pattern model represents the variability in a pattern, the first thing to do is look at a model for the pattern itself.

The Difference between Modeling a Pattern and Modeling Variability

A model of an 'MVC pattern' might look like the UML diagram below. A controller references a model. Action methods on the controller return view instances or data. This model is useful in explaining how the MVC pattern works (if it is accompanied with text or a verbal explanation), but it is not helpful to those implementing it in three key ways:

1. It does not explain how an individual instance of the pattern works.
2. It does not help the pattern implementer (a developer in this case) understand which parts are the same in every instance and which parts are variable.
3. It does not tell them how to implement the pattern in technologies of their choice, and it does not allows for their specific constraints and requirements.

In other words, your users - your toolkit users - still need help understanding how to implement the pattern in their solution. What artifacts do they need to create every time they create a new instance of the pattern? What parts of each artifact are fixed for them (and don't need to implement) and which are variable (and require their configuration)?

For example, in this pattern, users need to know that when they create a new 'Controller' class they must do the following: derive the class from a particular base class, comply with a naming convention in which the class name ends with "Controller", and implement 'Action' methods decorated with specific attributes that return specific types. They must also follow all the best practices for creating controller class instances for their organization, perhaps using the factory patterns or dependency injection to make the code unit testable.

None of that implementation detail is visible in the diagram below.

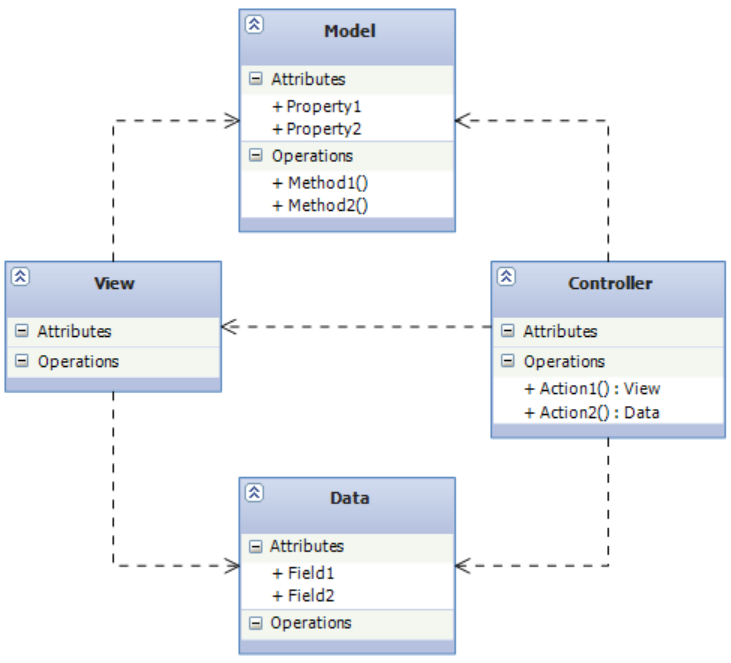
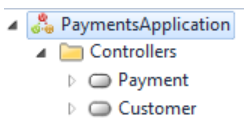


Figure 1: A model representing the MVC pattern. It does not show the variability among different implementations of the pattern, nor detail of implementing the pattern in a specific technology or with specific best development practices.

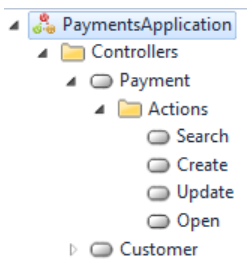
Think About Variability

If solution architects at Contoso were to create a model to help their developers implement MVC web apps, it would look very different from the UML model of the pattern. It would only show the parts of MVC that vary from one implementation to another, and only focus on parts of the MVC pattern that a Contoso developer should configure for their specific web application. It would embody all the best practices and standards for creating web applications at Contoso. The Contoso model would describe particular instances of MVC web application implementations that they build for their customers rather than the industry accepted theoretical pattern itself. The toolkit that operates on the Contoso model would guide developers towards correctly implementing web applications (MVC-style) rather than modeling MVC at all.

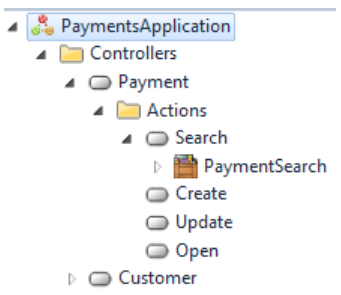
If a developer at Contoso were to interact with a model of MVC that only exposes the variability of their web applications, it might look like the one below. Users can create an application as a root node of the model, and then add 'Controller' elements to it.



Each 'Controller' element may have a set of configurable 'Action' elements.

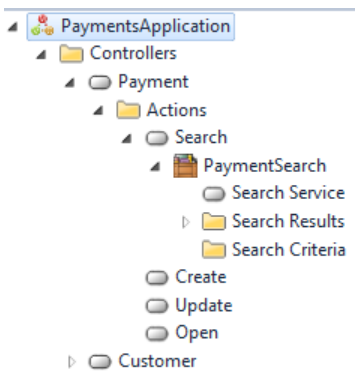


Each 'Action' might have a 'View' associated with it that is returned by the action.



The 'Views' themselves may be individual patterns, and there may be many different types. So we might want different types (archetypes) of views to plug into this model rather than try to define them all up front.

For example, a pattern for creating search kinds of views would be very different from the pattern for creating master-details kinds of views, or other edit views.



As you can see, this looks nothing like the UML model that represents MVC in the general case. That is because this model helps users create new individual MVC implementations rather than trying to explain how the pattern works. It is highly structured, so it guides users to correct implementations (i.e. you can't create ANY kind of application with this model, only ASP.NET MVC applications at this particular organization).

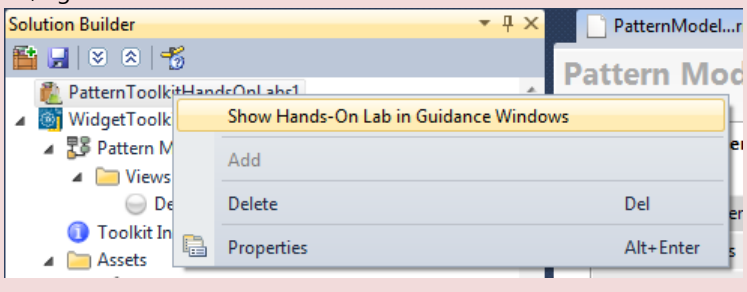
Furthermore, this level of specificity allows a Contoso architect to attach automation and code generators to the model (because it is highly structured) that will create the skeleton of the implementation for the users in accordance with whatever architecture Contoso architects decide upon.

Create a new Pattern Toolkit for ASP.NET MVC

For this lab you should start with a pattern toolkit project called “MVCApplicationToolkit”.

If you are following the first lab you already have a solution created with the Widget Toolkit, in this lab you are going to add another toolkit to the solution.

> **Warning**: Before proceeding to the next step, be aware that creating or adding a new Pattern Toolkit to your solution will automatically switch the guidance in the ‘Guidance Workflow Explorer’ window from that you are now following now, and will guide you to specific detailed guidance on building pattern toolkits.
> To restore this guidance, select the ‘Pattern Toolkit Hands-On Labs’ entry in the ‘Guidance Workflow Explorer’ window.
> Or, right-click on the ‘’ in Solution Builder and select ‘Show Hands-On-Labs Guidance Windows’

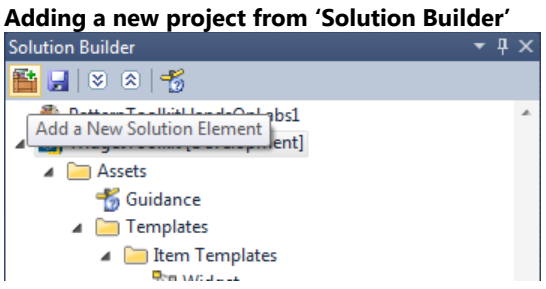


> **Tip**: Before proceeding with the following steps in this topic, you may first want to read the whole length of this topic, before stepping through it, as the modal dialogs that pop up will obscure the remaining steps of this topic.

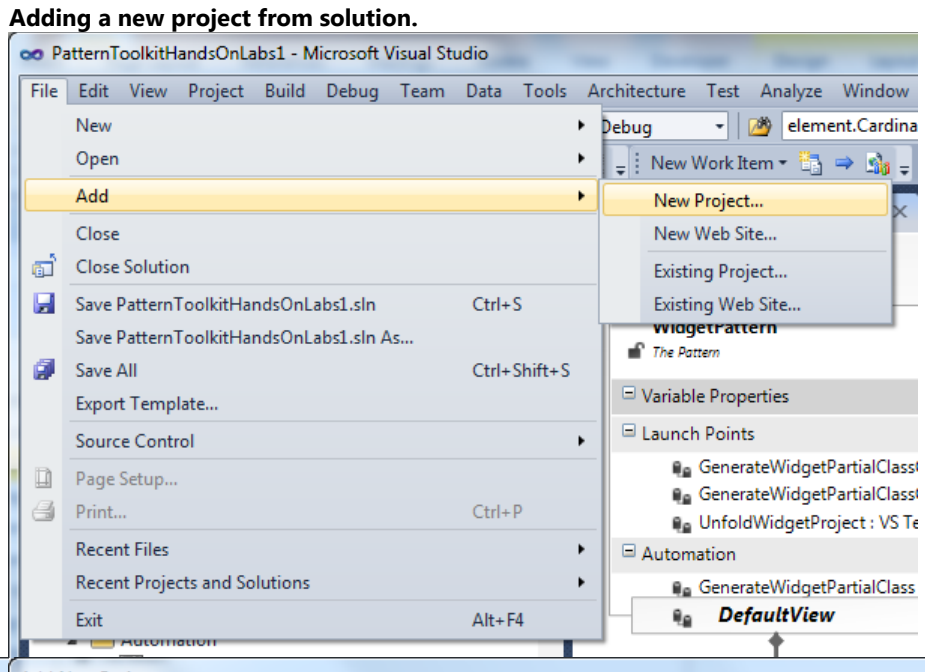
You can add a new pattern toolkit project to your solution in two ways:

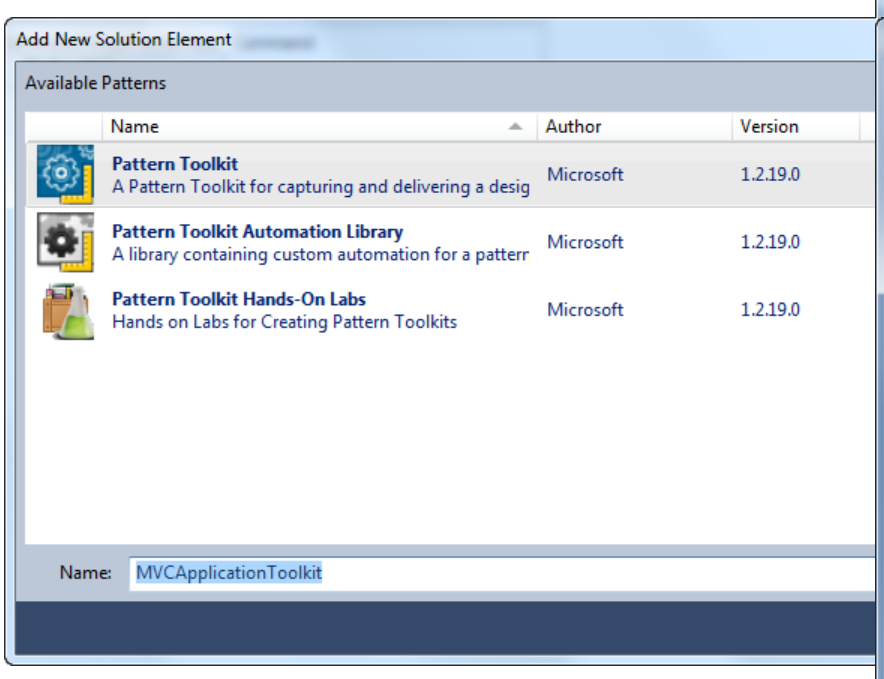
1. Add a new pattern toolkit by clicking the ‘Add New Solution Element’ button in ‘Solution Builder’
2. Add a new pattern toolkit project by clicking on the ‘File | Add | New Project’ menu.

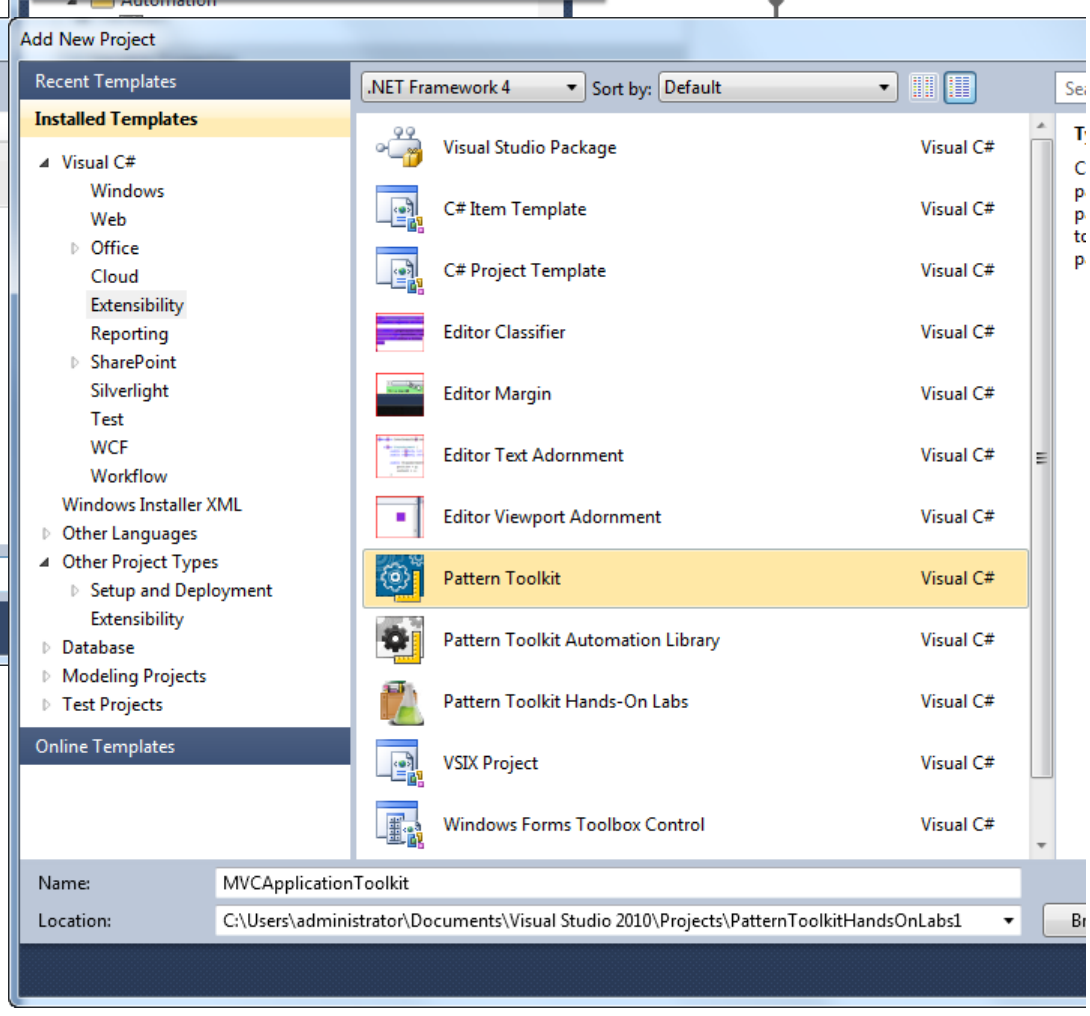
Adding a new project from ‘Solution Builder’



Adding a new project from solution.







In either case, create and configure a new Pattern Toolkit project as follows:

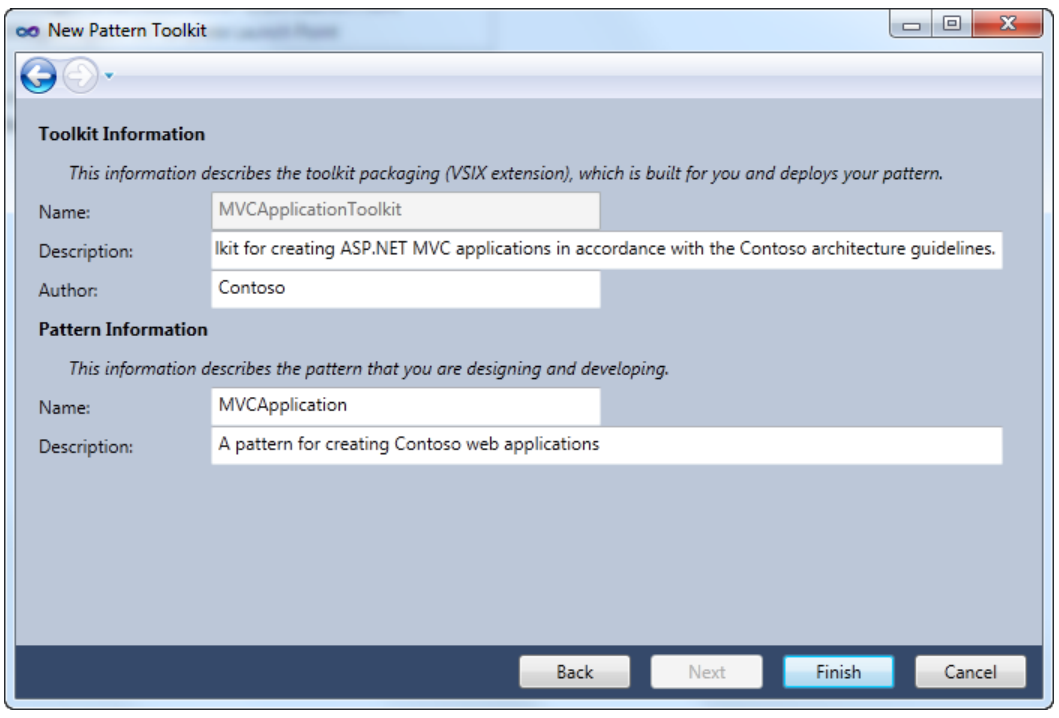
Toolkit Information

Name: MVCApplicationToolkit

Description: A toolkit for creating ASP.NET MVC applications in accordance with the Contoso architecture guidelines.
Author: Contoso

Pattern Information:

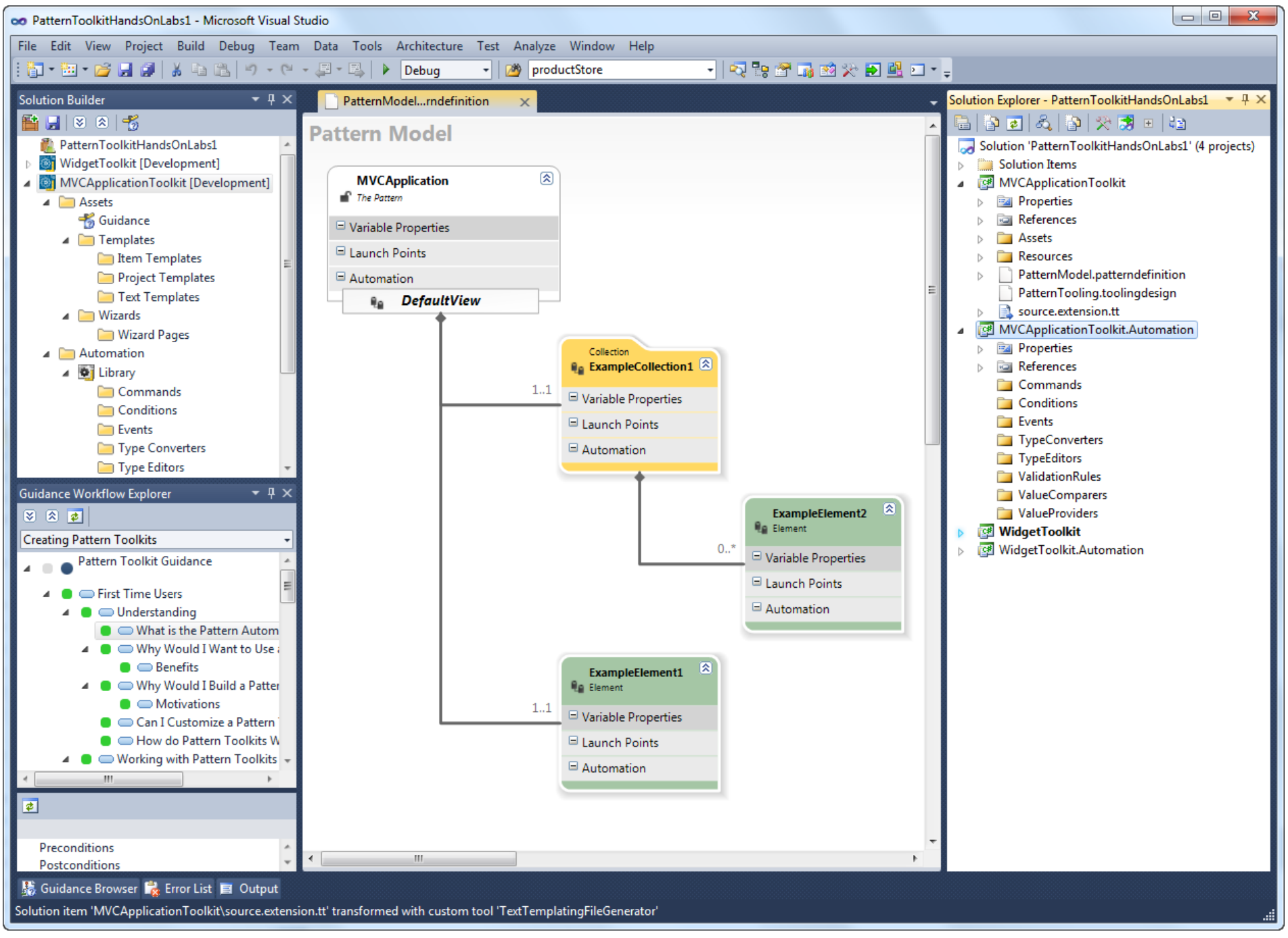
Name: MVCApplication
Description: A pattern for creating Contoso web applications



Verify the Development Environment

The environment should now look something like the image below.

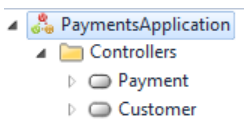
> **Note**: You can have several pattern toolkits present in the same solution, if you were following the first lab you will now have both the 'WidgetToolkit' and this 'MVCApplicationToolkit' in the solution. But this hands-on-lab does not require the 'WidgetToolkit' and you can remove it from the solution if you want.



Add Controllers to the Pattern Model

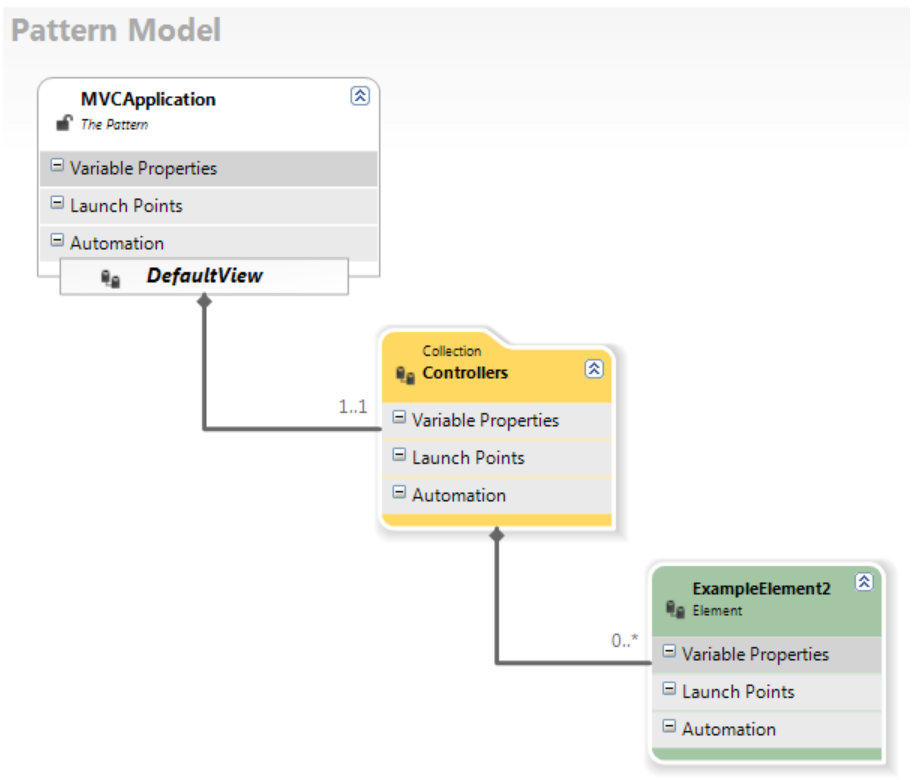
As designers of this toolkit, we want the user's experience of it to look like the one below, so we are going to add a 'Controllers' collection to the pattern model and add a 'Controller' element to that.

> **Tip**: Read the Think About Variability topic in this section for more information on why we want the user experience to look like this.



In the 'Pattern Model' delete the shape 'ExampleElement1', and rename the 'ExampleCollection1' shape to "Controllers".

> **Tip**: You can also open the 'Toolbox' window (**CTRL + W, X**), and drag a 'Collection' tool onto the design surface and rename the shape to "Controllers".

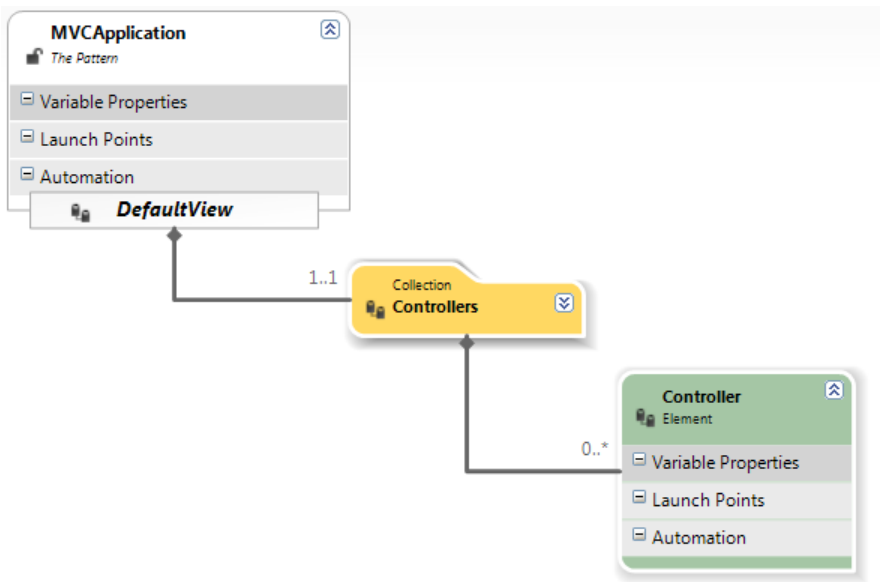


> **Note**: We are going to leave the 'Cardinality' at '1..1' (OneToOne) and 'Auto Create' "true" so that a single 'Controllers' collection is created every time a new 'MVCApplication' pattern is created.

Collapse the 'Collection' shape, and right-click on the diagram and select the 'Auto Arrange Shapes' menu.

Next, rename the 'ExampleElement2' shape to "Controller", and change 'Auto Create' to "false" so that a user must explicitly create instances of the 'Controller' element.

> **Note**: We are going to leave the 'Cardinality' at '0..*' (ZeroToMany).



Verify the User Experience

Build and Run the toolkit project (**CTRL + F5**).

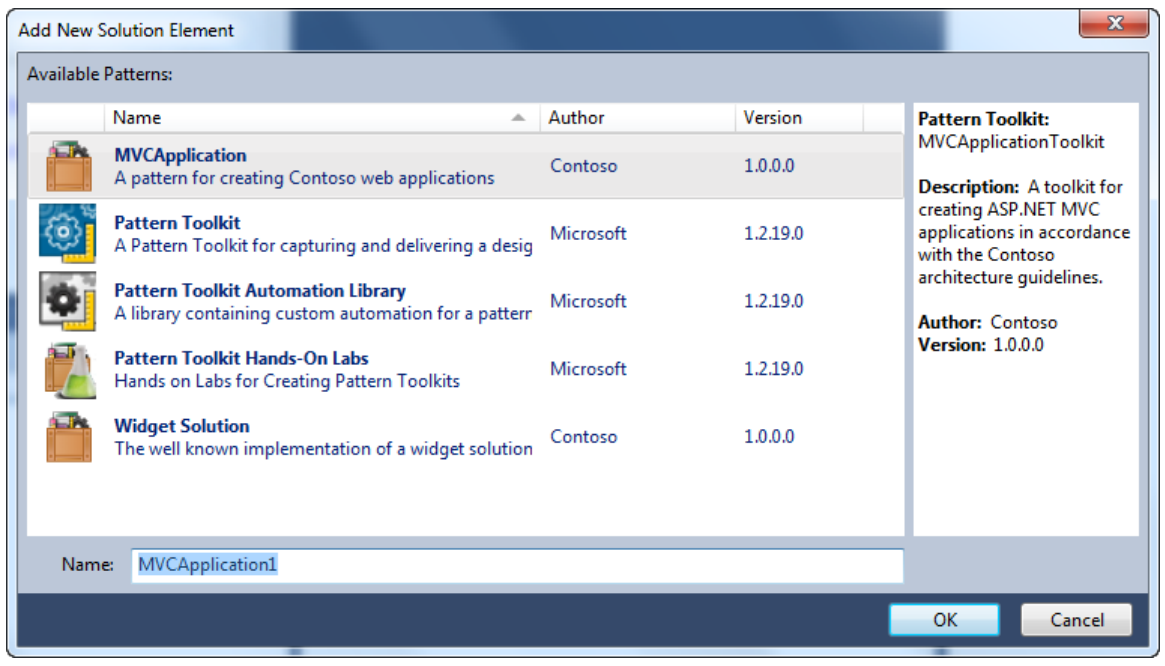
In the Experimental Instance of Visual Studio that starts up, in order to test your toolkit at this point, you will have to create an empty solution and then use the 'Solution Builder' window to add an instance of your pattern to the solution.

Start by creating a new, blank solution:

- Click on the 'File | New | Project' menu (**CTRL + SHIFT + N**). The 'Add New Project' dialog opens.
- In the search box at the top right, type "Blank". The list of project types is now filtered to include only the 'Blank Solution' project template.
- Click OK to create the solution.

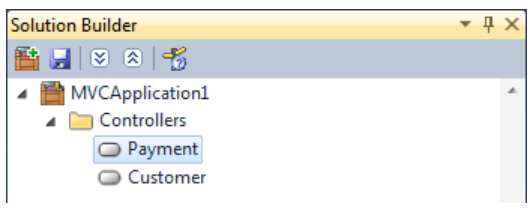
Open the 'Solution Builder' window, and 'Add New Solution Element'

The 'Add New Solution Element' dialog will open, and you can choose your 'MVCApplication', give it a name and click OK.



Users of your toolkit should now be able to create a model for an MVC application and add controllers to it.

Below is an example of a payments web application with two controllers.



At this point nothing has happened in Solution Explorer to implement the application. That will be done in subsequent parts of this hands-on lab.

Add Actions to the Pattern Model

Close and Save the Visual Studio Experimental Instance.

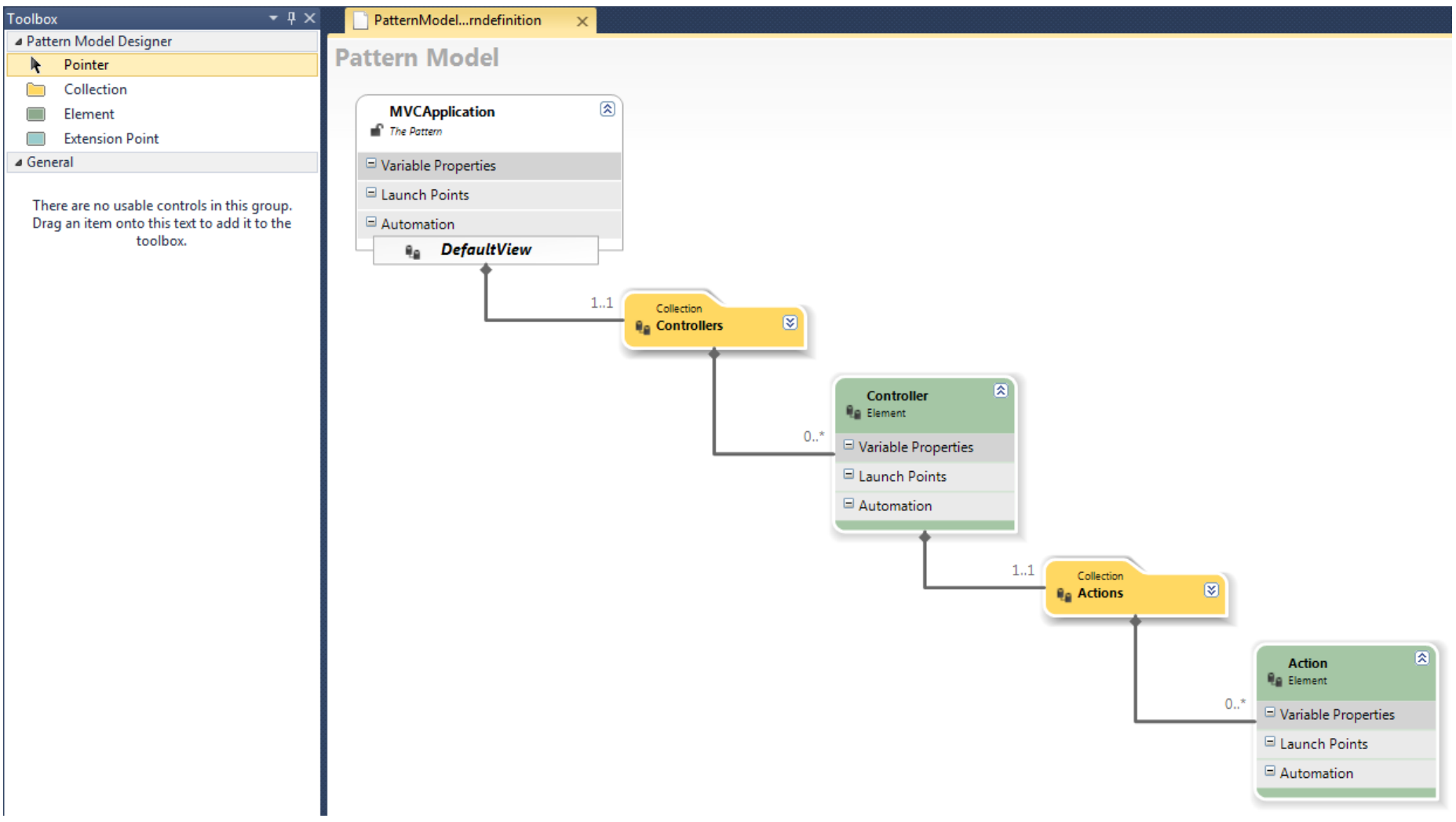
The next step is to add the concept of 'Actions' to the Pattern Model.

In the 'Toolbox' (**CTRL + W, X**), drag the 'Collection' tool from the toolbox and drop it on the 'Controller' shape.

Name it "Actions".

Then drag the 'Element' tool from the toolbox and drop it on the 'Actions' collection.

Name it "Action".

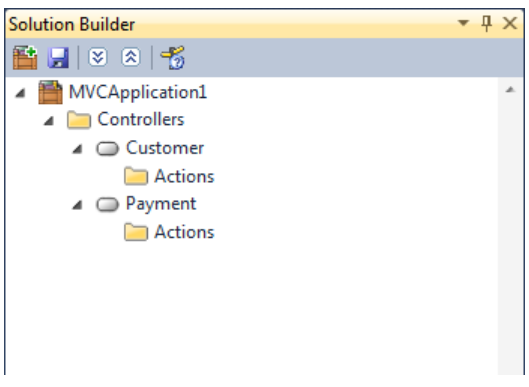


Tip: You can right-click on the diagram and select 'Auto Arrange Shapes' menu to arrange the shapes as shown.

Test the Action

Build and Run the toolkit project (**CTRL + F5**).

In the Experimental Instance of Visual Studio that starts up, you may open the solution from the previous test cycle. You may also create a new 'Blank' solution and add a new 'MVCApplication' to it from 'Solution Builder'.



> **Note**: If you opened the previous solution, you will notice that each controller now has automatically added to it an 'Actions; collection ready to add 'Action' elements.

You can now add one or more actions to each controller.

Close and Save the Visual Studio Experimental Instance.

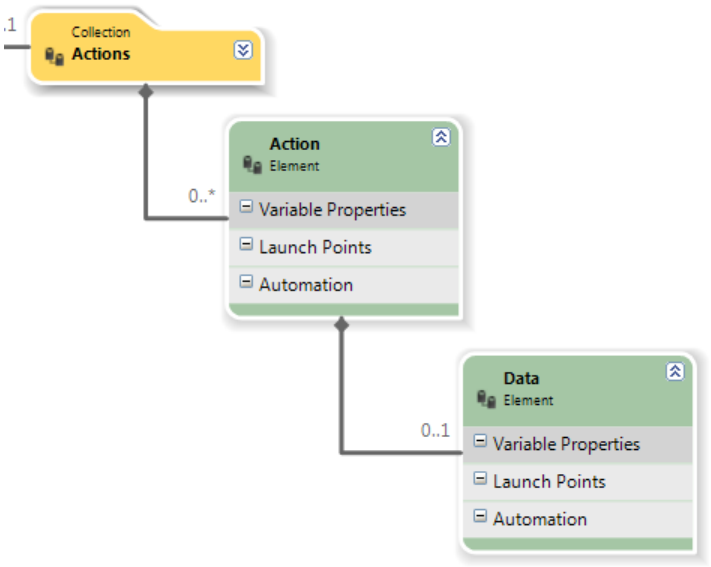
Add Data to an Action

The next step is to add the concept of 'Data' to an 'Action', as an 'Action' in MVC may just respond with some data for a web page to consume, such as data for a client-side XJAX request.

In the 'Toolbox' (**CTRL + W, X**), drag the 'Element' tool from the toolbox and drop it on the 'Action' shape.

Name it "Data".

Change the 'Cardinality' property of the 'Data' shape to 'ZeroToOne'.



Part 2: Extension Points

Extension Points allow a toolkit author to define a place in their pattern where other pattern toolkits can plug in and extend its functionality. They are useful when the author does not know every type of variability that the user may need, and wants to make that variability pluggable, or to allow additional patterns to integrate with this pattern to enhance its functionality.

In this lab, knowing MVC, we know that users will have to create 'Views' associated with 'Actions'. We may know some of the types of views that may be created, like: login, search, update, master/details and display for example. However, we don't know about every different type of view a user may want to create in our organization. We expect that at some point a new type of view will be defined and will then need to be standardized across all applications. And sometimes users of the toolkit will need to create something completely custom for their specific web application. Perhaps other parties will want to provide them. We know that when any of these cases arises we don't want to ship a new version of the MVC toolkit, but instead just ship toolkits containing those new views.

The concept of an 'Extension Point' is similar to creating a .NET interface for a plug-in where the host application defines an interface that plug-ins must implement. An 'Extension Point' defines a contract that must be implemented by the plug-in pattern that implements the extension point provided by the host toolkit. The contract is defined by the host toolkit and by the properties of the 'Extension Point'.

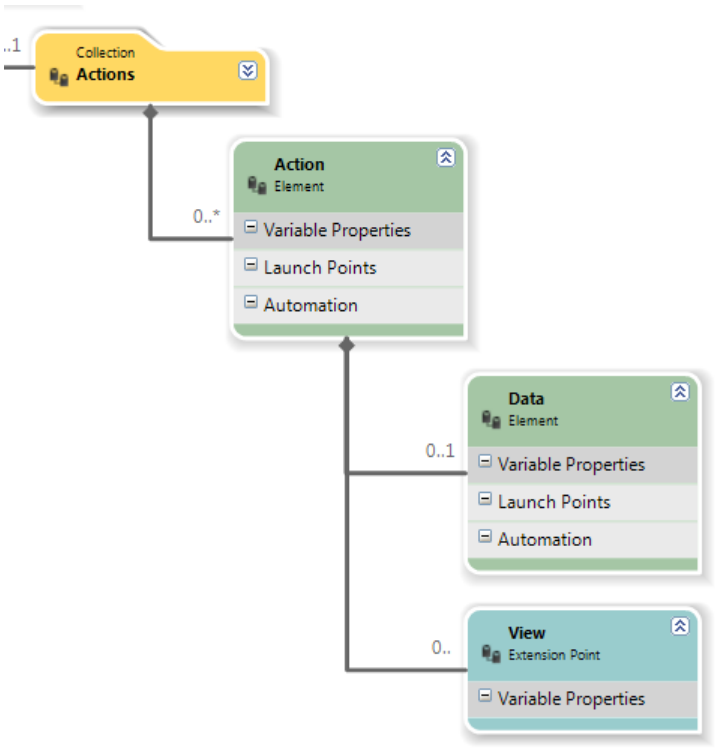
Create an Extension Point for Views

In the MVCApplication toolkit project, open the toolbox (**CTRL + W, X**)

Drag the 'Extension Point' tool from the toolbox and drop it onto the 'Action' shape.

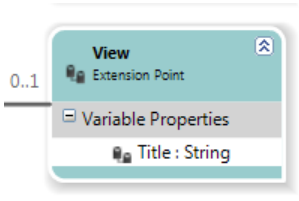
Name it "View".

Change the 'Cardinality' property of the 'View' shape to 'ZeroToOne'.



Add a new 'Variable Property' to the 'View' shape called "Title".

> **Tip**: You can add new 'Variable Properties' by right clicking on the 'Variable Properties' compartment of the shape.

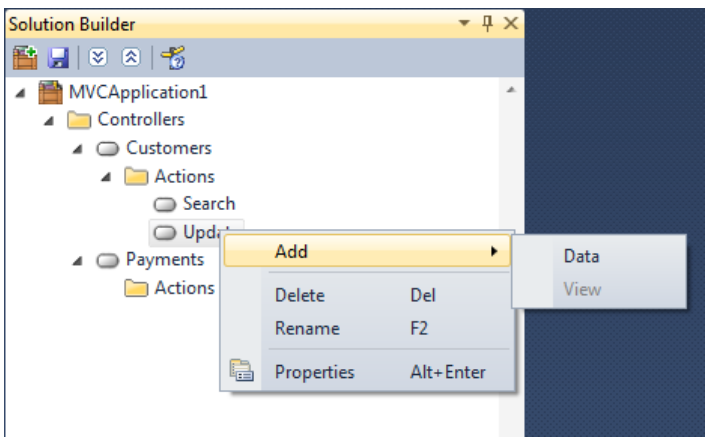


Test the Extension Point

Build and Run the toolkit project (**CTRL + F5**).

In the Experimental Instance of Visual Studio that starts up, open the solution from the previous test cycle.

Right-click on any 'Action' and you will see that you cannot add a new 'View'. This is because no installed pattern toolkit yet implements that extension point.



Close and Save the Visual Studio Experimental Instance.

Register the Extension Point in Visual Studio

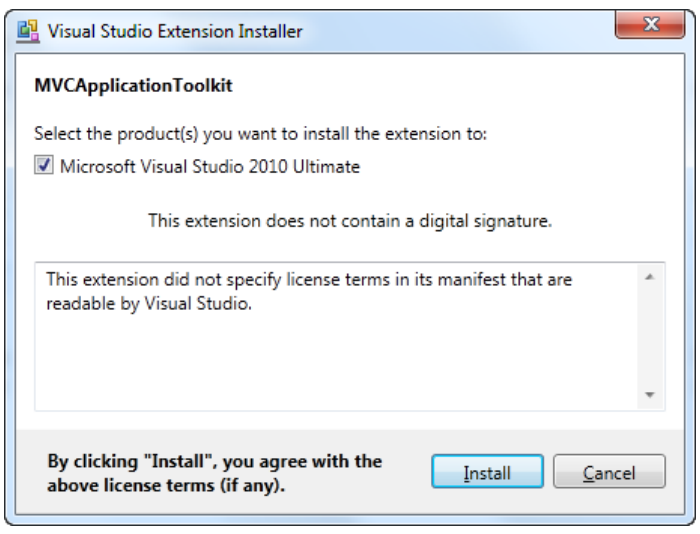
In order to implement an 'Extension Point' of another pattern toolkit, the pattern toolkit containing the Extension Point must be installed in Visual Studio (not just installed in the Experimental Instance, which is where pattern toolkits are automatically installed when you build them.) You must install the current version of your MVCApplication toolkit into Visual Studio so that you can reference its extension points from another pattern toolkit.

Open the folder on disk that contains your current toolkit solution, and find the 'Binaries' subfolder, e.g. <SolutionFolder>\Binaries.

> **Warning**: You are about to be instructed to close Visual Studio in the step below, which will prevent you from reading the instructions completing this step in the guidance. Please ensure you read this topic entirely before proceeding.

Close all instances of Visual Studio.

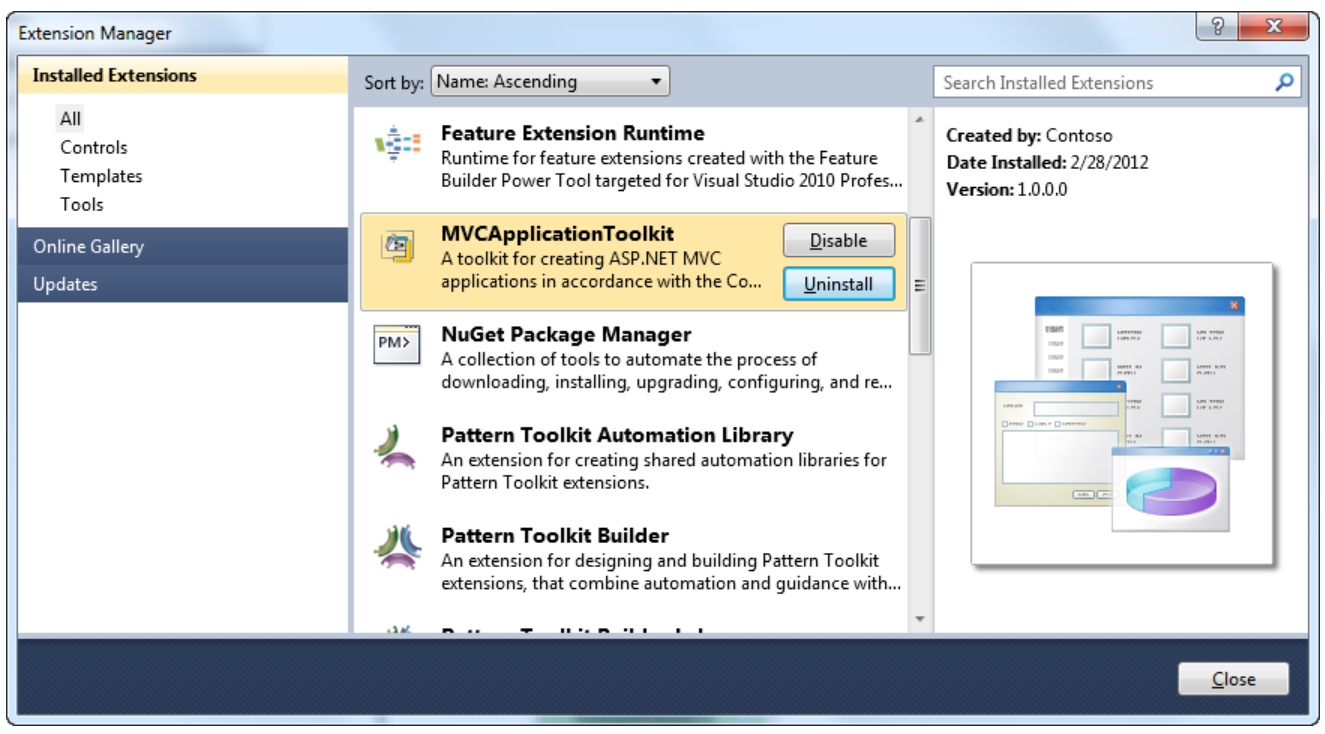
Double-click on the 'MVCApplicationToolkit.vsix' file to install the toolkit into Visual Studio.



Restart Visual Studio, re-open your pattern toolkit solution, and navigate back to this guidance step.

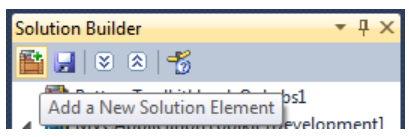
Add a New Pattern Toolkit to the Solution

Now that we have registered the 'MVCApplicationToolkit' in Visual Studio, we will be able to reference the extension points exposed by it, and build other pattern toolkits that extend it. To confirm that the MVCApplicationToolkit' has been registered successfully in Visual Studio, open the 'Extension Manager' (Tools | Extension Manager) and ensure it is present in the list of extensions there.

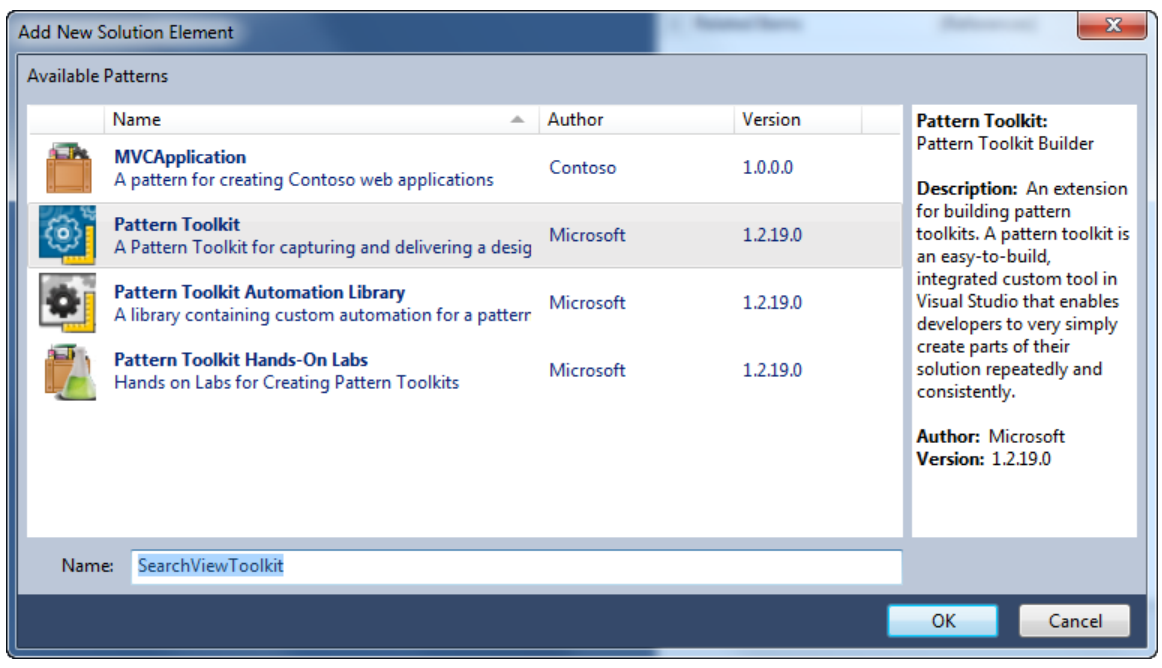


In this step we will add a new Pattern Toolkit project to the solution that will implement the extension point we defined in the 'MVCApplication' pattern.

In 'Solution Builder', click on the 'Add a New Solution Element' button at the top of Solution Builder.



Select the 'Pattern Toolkit' pattern and give it the name "SearchViewToolkit".



Create a new toolkit, and configure as follows:

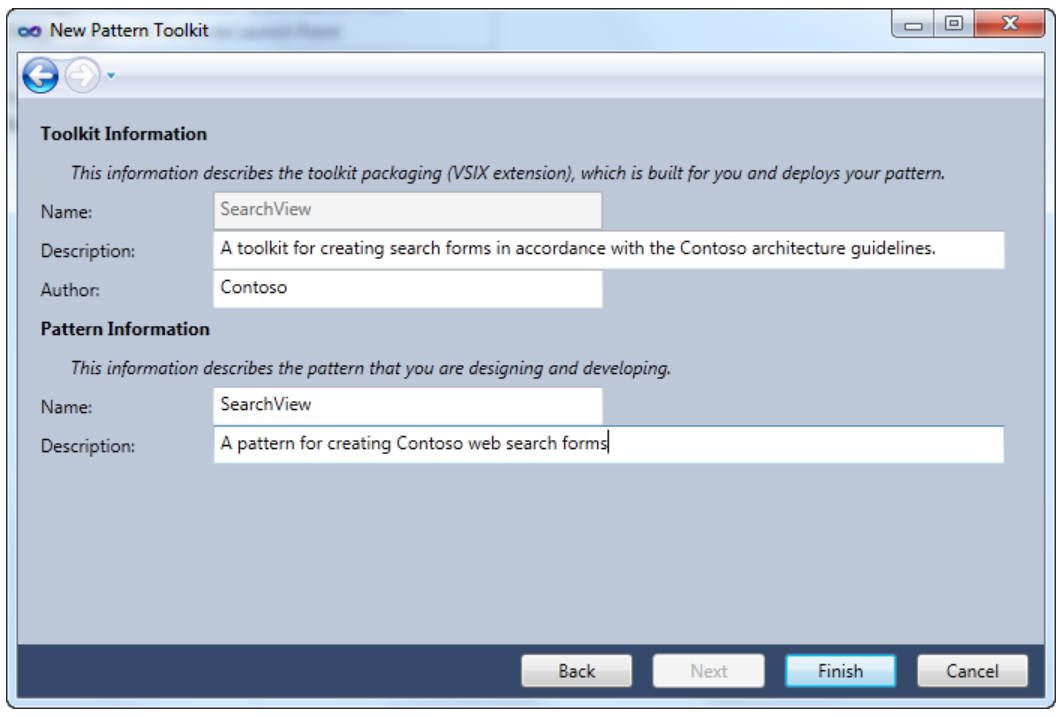
Toolkit Information

Name: SearchViewToolkit

Description: A toolkit for creating search forms in accordance with the Contoso architecture guidelines.
Author: Contoso

Pattern Information:

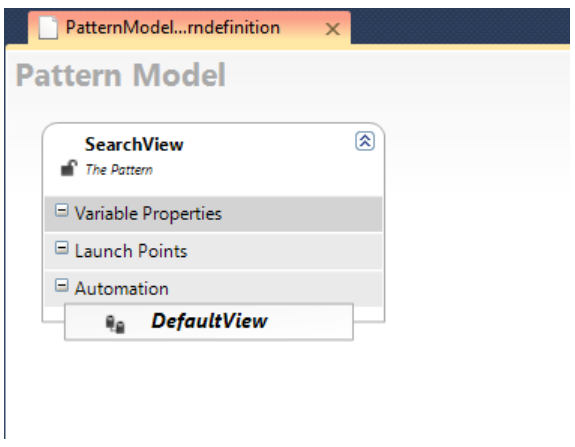
Name: SearchView
Description: A pattern for creating Contoso web search forms



A new pattern toolkit will be added to your solution, alongside your MVCApplication toolkit project.

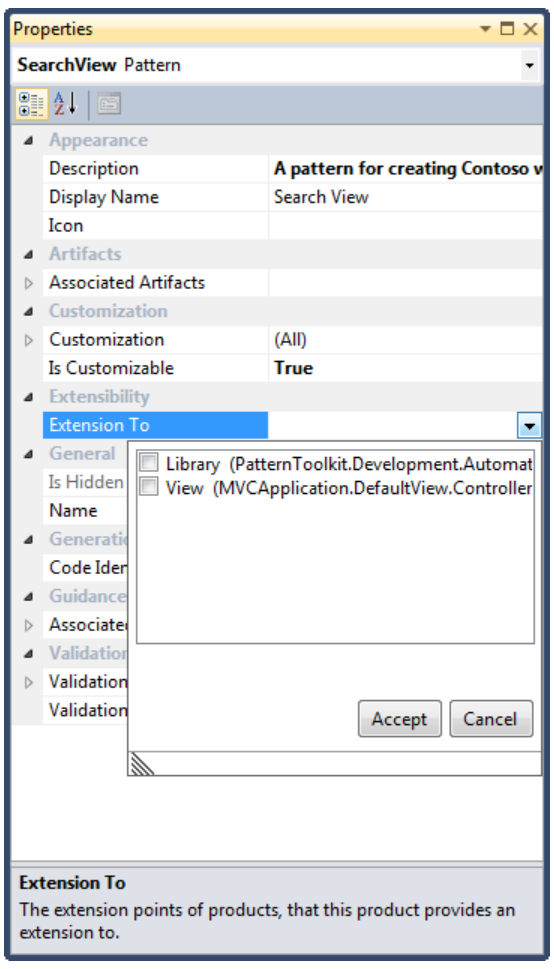
Designate the SearchView Pattern as Extending View

After you have added the new 'SearchView' pattern toolkit to the solution, delete all the 'Collection' and 'Element' shapes in the pattern model, leaving just the 'SearchView' element



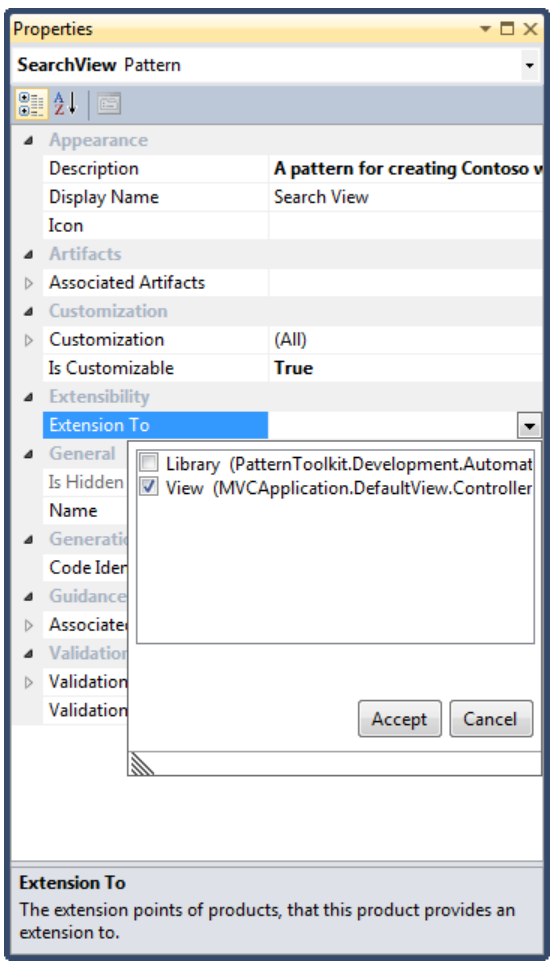
Select the 'SearchView' shape and examine the Properties Window (**F4**) for its properties.

Find the 'Extension To' property, and click the drop down control to display a list of extension points from toolkits installed in Visual Studio.

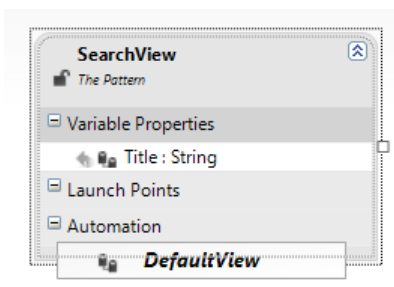


Select the checkbox next to 'View' to indicate that this pattern is an extension to the 'View' extension point of the 'MVCApplication' toolkit.

Click the 'Accept' button.



Notice that the 'SearchView' shape now has a 'Variable Property' called "Title" that has been inherited from the 'View' extension point of the 'MVCApplication' toolkit.

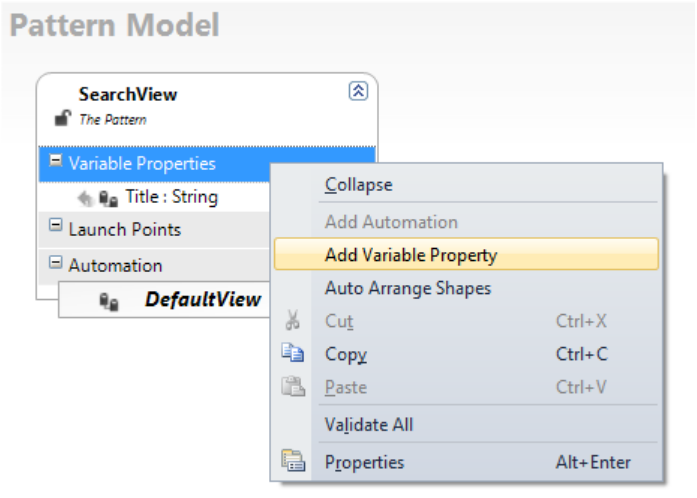


Add Extension Properties to Search View

Part of the benefit of using extension points in your pattern toolkits is that the extensions themselves may need to define additional variability that was not considered at the time the extension point was defined. The extension point only defines the properties which it needs to function in the host toolkit. However, a extending toolkit can define additional properties for its own uses.

We are going to add two properties that are pertinent to search views, and can later be used to generate the code for he view in the application.

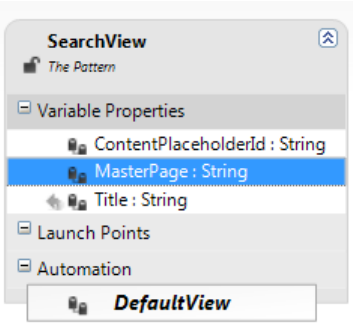
Right-click on the 'Variable Properties' compartment of the 'SearchView' shape, and add two new properties.



Name the first property: "MasterPage", and set its 'Default Value' property to "~/Views/Shared/Site.Master".

Name the second property: "ContentPlaceholderId", and set its 'Default Value' property to "MainContent".

> **Note**: These properties and default values have specific and familiar values to ASP.NET MVC 2 web applications, but are only used here for illustration purposes only, and will revisited in a further lesson on Wizards.

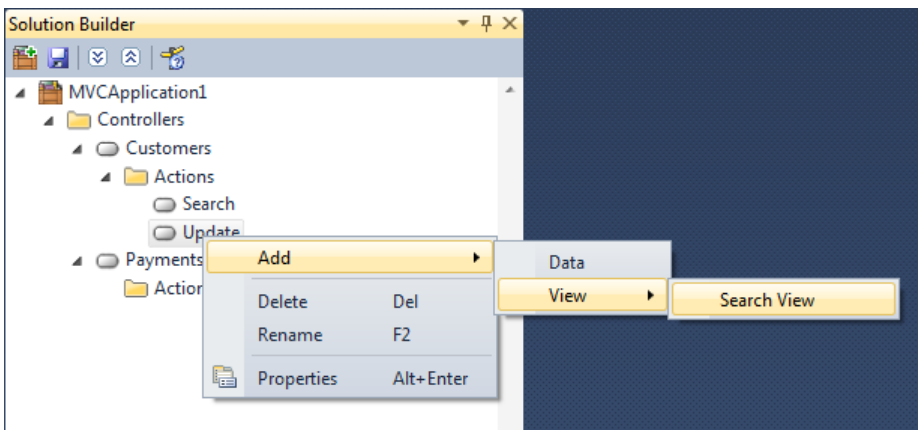


Test the Implemented Extension Point

Build and Run the toolkit project (**CTRL + F5**).

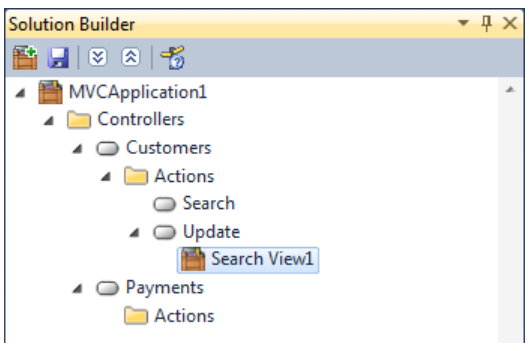
In the Experimental Instance of Visual Studio that starts up, open the solution from the previous test cycle.

Right-click on any 'Action' and you will see that you can now add a new 'Search View'.

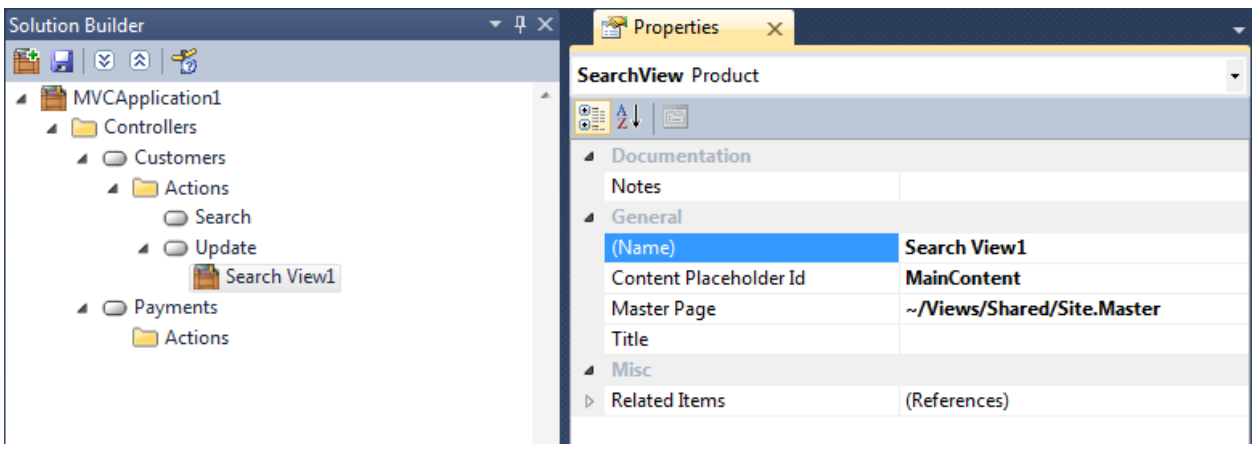


> **Note**: Solution Builder has automatically detected that there is a new pattern toolkit installed that implements the 'View' extension point, and adds a menu for the user to add an instance of it.
>
> **Tip:** You can see which toolkits are installed in this instance of Visual studio by looking in the Extension manager (Tools | Extension Manager)

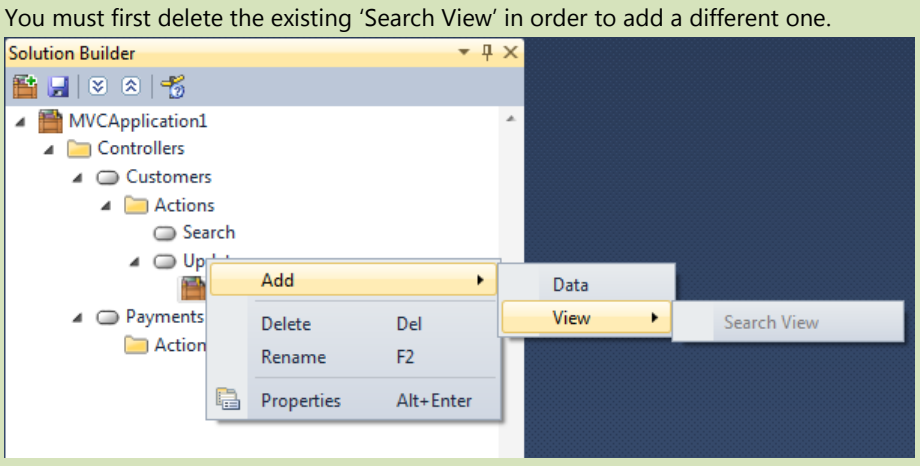
If you add a 'Search View' to an action element, it will look something like this:



Notice the default values of the properties of a new 'Search View'.



Try to add another instance of a 'Search View' to the same action.

> **Tip**: Notice, that once you have added one instance of a 'View' to an 'Action' element, you can't add others. This is because the Cardinality of the 'View' extension point in the 'MVCApplicationToolkit' was 'ZeroToOne'.
> You must first delete the existing 'Search View' in order to add a different one.
>
> 
>

Close and Save the Visual Studio Experimental Instance.

Part 3: Validation

Pattern toolkits help users implement their solutions correctly and consistently, but often valid configuration is vital for a correct implementation of the pattern. The pattern toolkit should never create an invalid solution for the user. All input provided by the user should not only be checked for correctness, but should also be verified in order to create a valid permutation of the solution.

Users may pass through various invalid states when configuring the patterns in their solutions in 'Solution Builder', and may move around the pattern completing various parts of the solution at any one time. The pattern toolkit not only needs to facilitate partial configuration, but must also provide helpful pointers to the user to unfinished or incomplete work.

For example, a user of the 'MVCApplication' toolkit can create an 'Action' that does not return anything (i.e. no 'Data' and no 'View'), but architecturally that is not correct. An action must either return a 'Data' or a 'View'. Users of the pattern toolkit can also create an 'Action' that returns both 'Data' and a 'View', and that is not correct either. Neither the cardinality rules of either 'Data' nor 'View' are able to resolve this assumed rule of the pattern.

This part of the lab will walk you through adding validation to the model to ensure the users of your pattern toolkit are providing meaningful and valid configuration for their solution specific implementation of an MVC application.

There are two steps to adding validation: (1) enabling validation on the model by choosing how and when it will execute, and (2) adding specific validation rules to specific elements in the Pattern Model to verify their specific configuration.

When you add validation rules you can choose to invoke existing validation rules that are provided in your toolkit, or you can write your own custom validation rule classes.

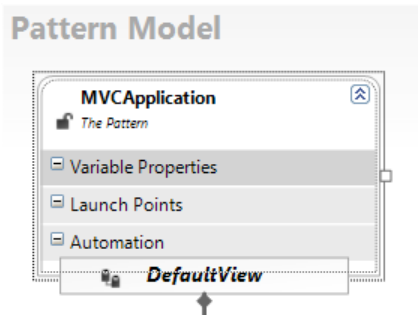
For this part of the lab you can choose to use the 'MVCApplication' toolkit we created in the previous section, or you can create a new toolkit.

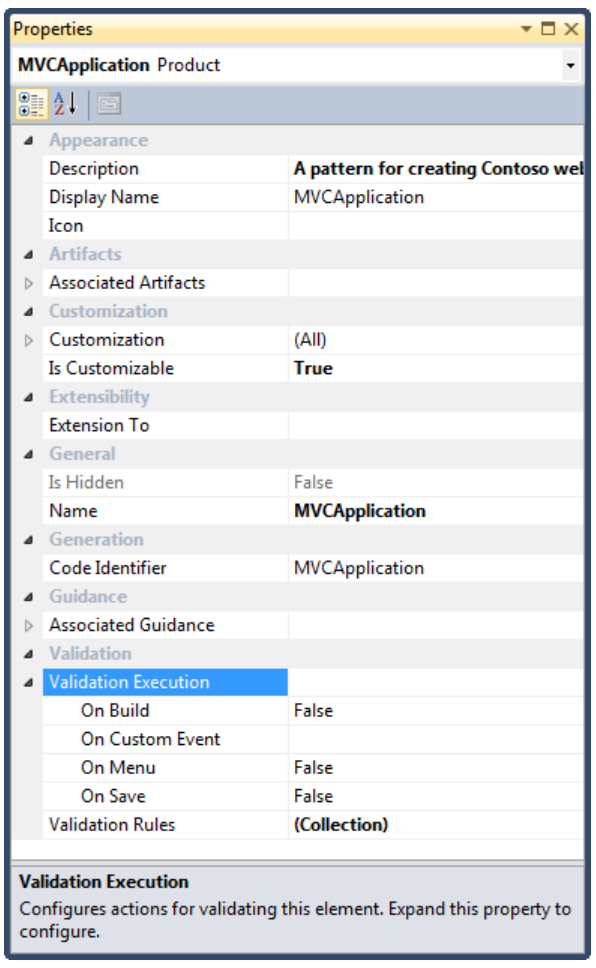
Enable Validation of the Whole Pattern

Validation is not executed by default, but is easy to configure. To enable validation, do the following:

Click on the pattern element and in the Properties window (**F4**), and expand the 'Validation Execution' property.

You will see that the 'On Build', 'On Menu', and 'On Save' properties are all set to False.





Change the 'On Menu', and 'On Build' properties to true.

> **Note**: These selections will provide a context menu on the pattern element in 'Solution Builder' that allows users to validate the model manually, as well as performing validation automatically when they build their solution.

Test Validation

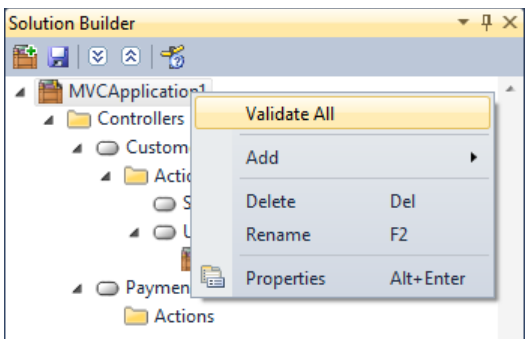
Build and Run the toolkit project (**CTRL + F5**).

In the Experimental Instance of Visual Studio that starts up, open the solution from the previous test cycle.

Right-click on any 'MVCApplication' element and you will see that you now have a 'Validate All' menu.

Clicking it will execute validation across whole pattern.

> **Note**: At this point no validation errors should appear.



Close and Save the Visual Studio Experimental Instance.

Using Built-In Cardinality Validation

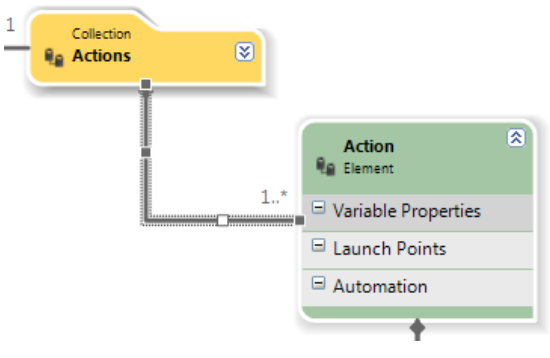
Now that validation is enabled in your pattern model, you can take advantage of the built-in cardinality validation.

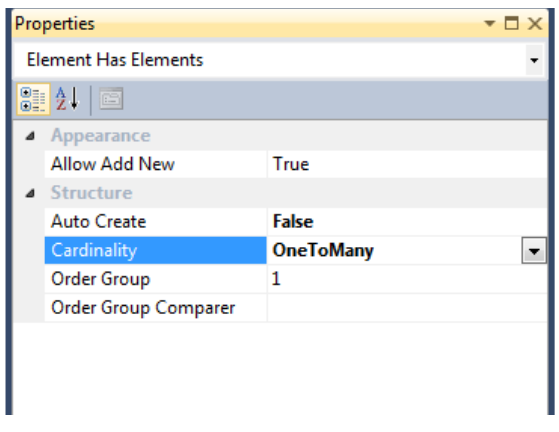
In this step we will change one of the relationships in your model to “OneToMany”, which requires that the parent element have at least one instance of that child element.

Click on one of the relationship lines connecting two of the elements in your pattern model (e.g. between ‘Actions’ and ‘Action’ elements in the MVCApplication toolkit).

In the Properties window, change the ‘Cardinality’ property to ‘OneToMany’.

> **Note:** You can also set the ‘Auto Create’ property to “true” if you want an instance automatically added each time a parent instance is added.



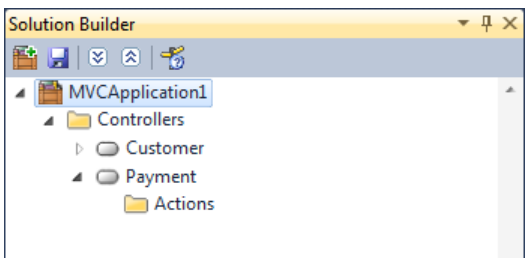


Test Cardinality Validation

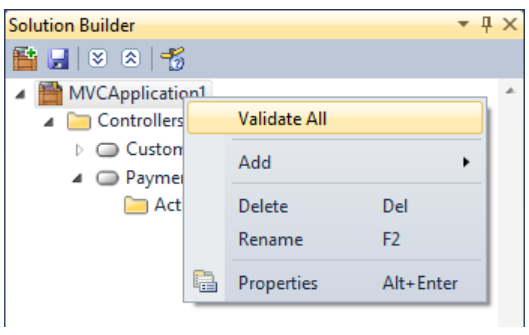
Build and Run the toolkit project (**CTRL + F5**).

In the Experimental Instance of Visual Studio that starts up, open the solution from the previous test cycle.

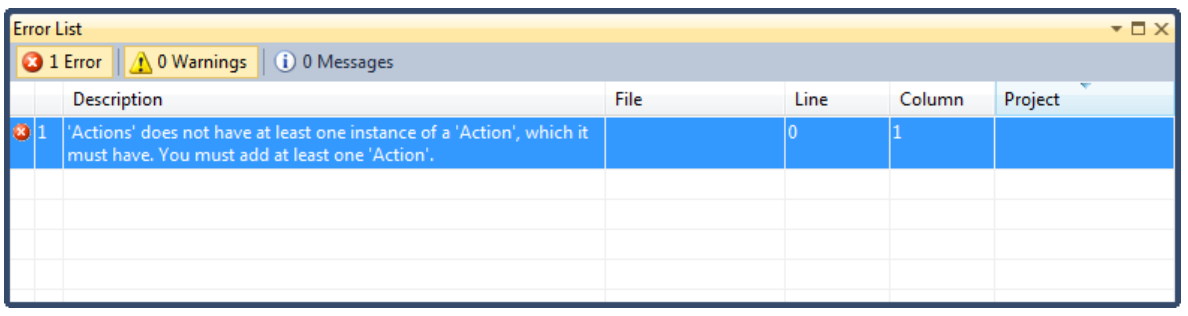
Now delete all ‘Action’ elements from a ‘Controller’ element, if you are using the MVCApplication toolkit.



Right-click on the ‘MVCApplication’ element and click the ‘Validate All’ menu.



Notice, that you now see a validation error in the ‘Error List’ window appear.



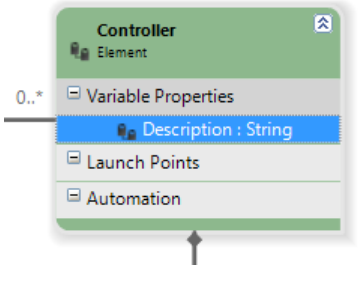
Close and Save the Visual Studio Experimental Instance.

Add Validation to a Node in the Pattern Model

Validation rules can be applied to any element and any property in the pattern model.

In this step, we will add a property to the 'Controller' element and validate that the property has a value, and that the value is more than 4 characters.

Add a new 'Variable Property' to the 'Controller' element called "Description".

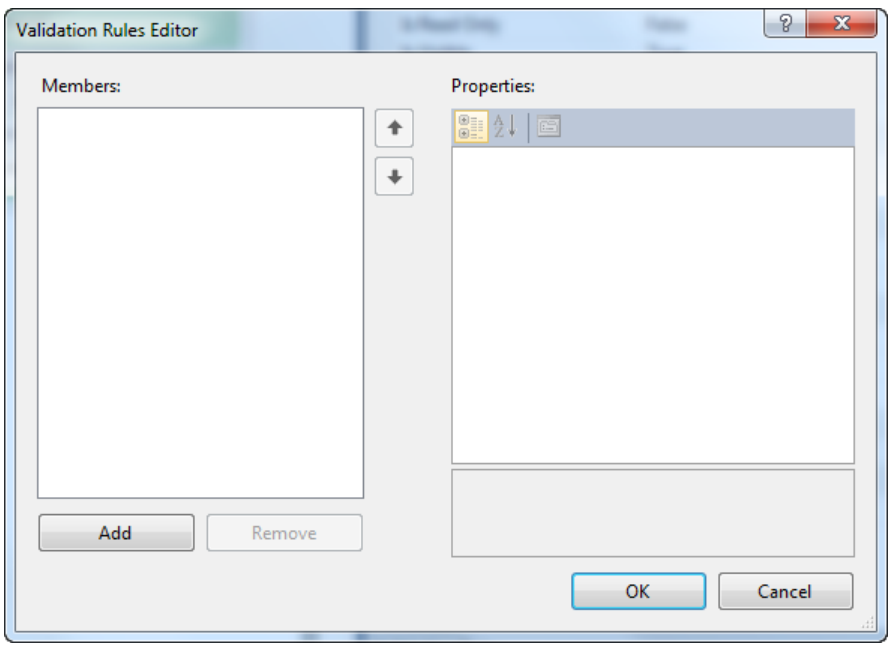


> **Note**: This property is intended to allow users to give a description of what the controller is supposed to do.

Select the property by clicking on it in the Controller element, and in the Properties window (**F4**), find the 'Validation Rules' property.

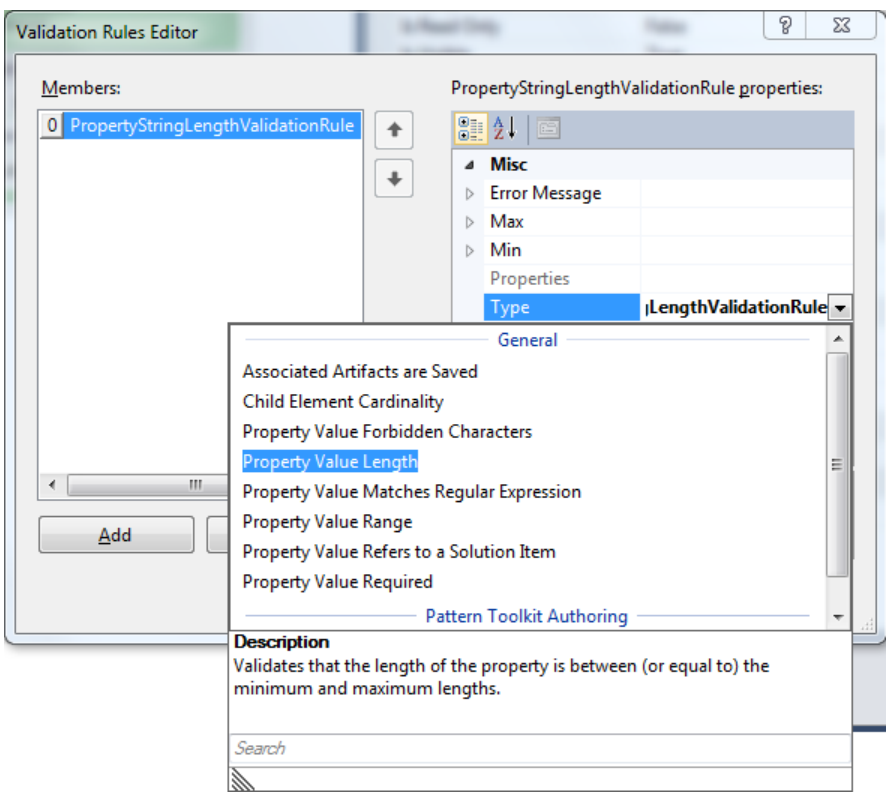
Click the ellipsis button which will open the 'Validation Rules Editor' window.

> **Note**: You can also set validation rules on any element in the pattern model, but in this case we are adding validation rules to an individual property of an element.



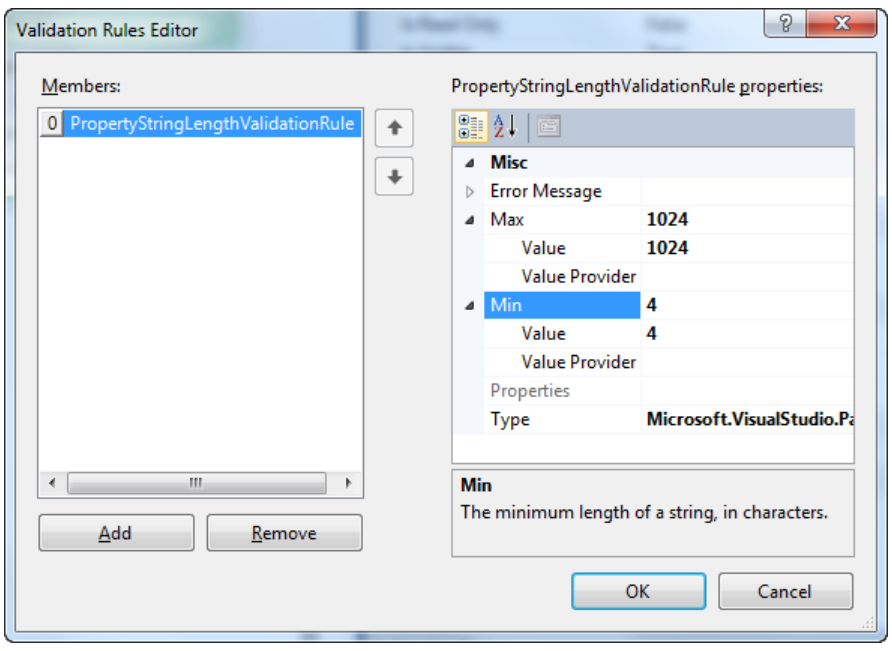
Click 'Add' to add a new validation rule.

In the 'Type' property, choose the 'Property Value Length' rule.



After choosing the validation rule type, additional properties appear in the properties window, allowing you to configure the validation.

Set the 'Min' property to "4", to require that the property must have at least four characters, and a reasonable Max, like "1024".



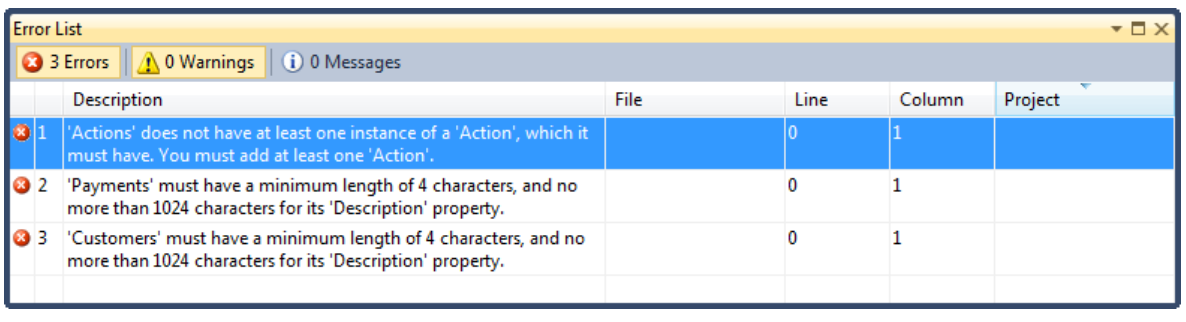
Press OK to close the Validation Rules Editor.

Test Property Rule Validation

Build and Run the toolkit project (**CTRL + F5**).

In the Experimental Instance of Visual Studio that starts up, open the solution from the previous test cycle.

Right-click on the 'MVCApplication' element and click the 'Validate All' menu.



> **Advanced Tip**: The generic error message for this validation rule can be customized for your pattern by providing your own custom 'Error Message' in the configuration of the validation rule.

Close and Save the Visual Studio Experimental Instance.

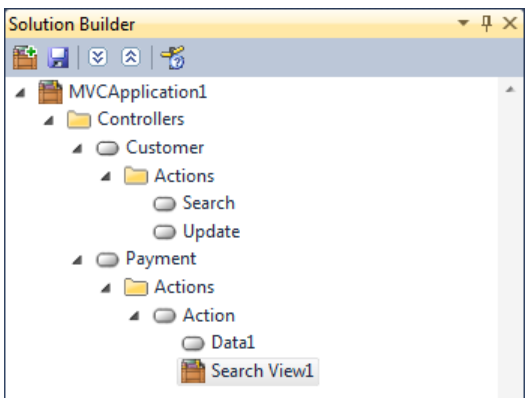
Part 4: Custom Validation Commands

If you have validation rules that cannot be accomplished using any of the built-in validator rules, you can create your own custom validation rules.

In this part, we will create a custom validation rule to prevent users from creating controller actions that have both a 'View' and a 'Data' elements to enhance the built-in cardinality rules.

The rule we will enforce in this toolkit will be: that each 'Action' can have either one 'View' or one 'Data', but not both.

What you see below is an invalid state that we want to prevent through validation.

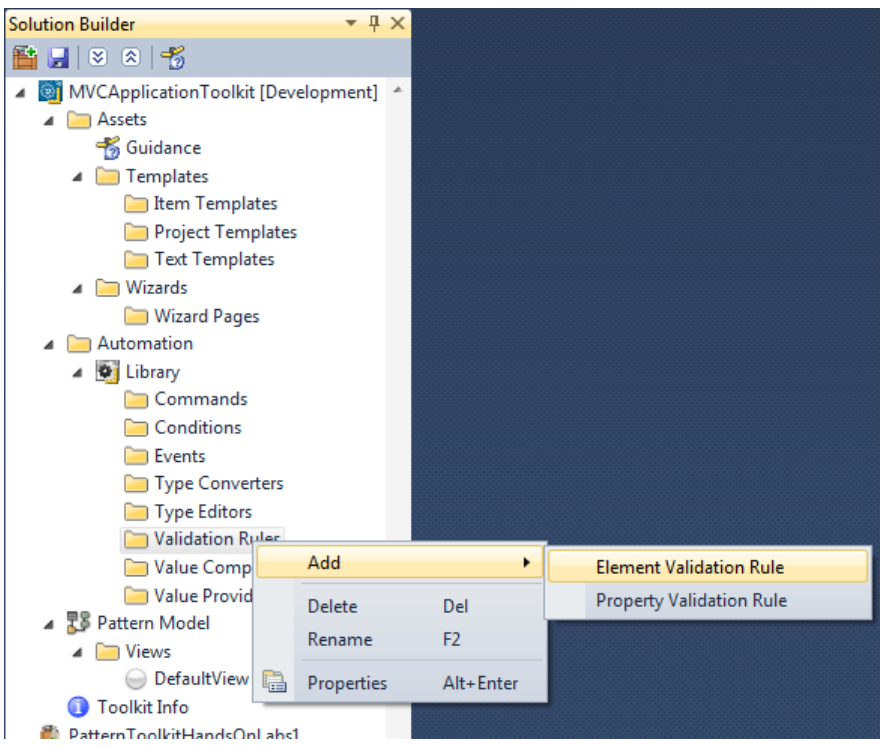


Add a New Custom-Coded Validation Rule

Close and Save the Visual Studio Experimental Instance.

In the 'MVCApplication' toolkit project, open the 'Solution Builder' window (**CTRL + W, H**) and locate the 'Validation Rules' element.

Right-click on 'Validation Rules' element and add a new 'Element Validation Rule' named "ActionReturnType".



A new custom validation class will be added to your solution and the class will be opened in the editor window.

> **Note**: Take a look at the code that was added. It contains comments to help you determine what you need to do to implement the validation rule.

After you are familiar with the code, delete all of it and replace it with the code below.

> **Important**: Adjust the code for any variations in namespace or pattern model names you have in your project.

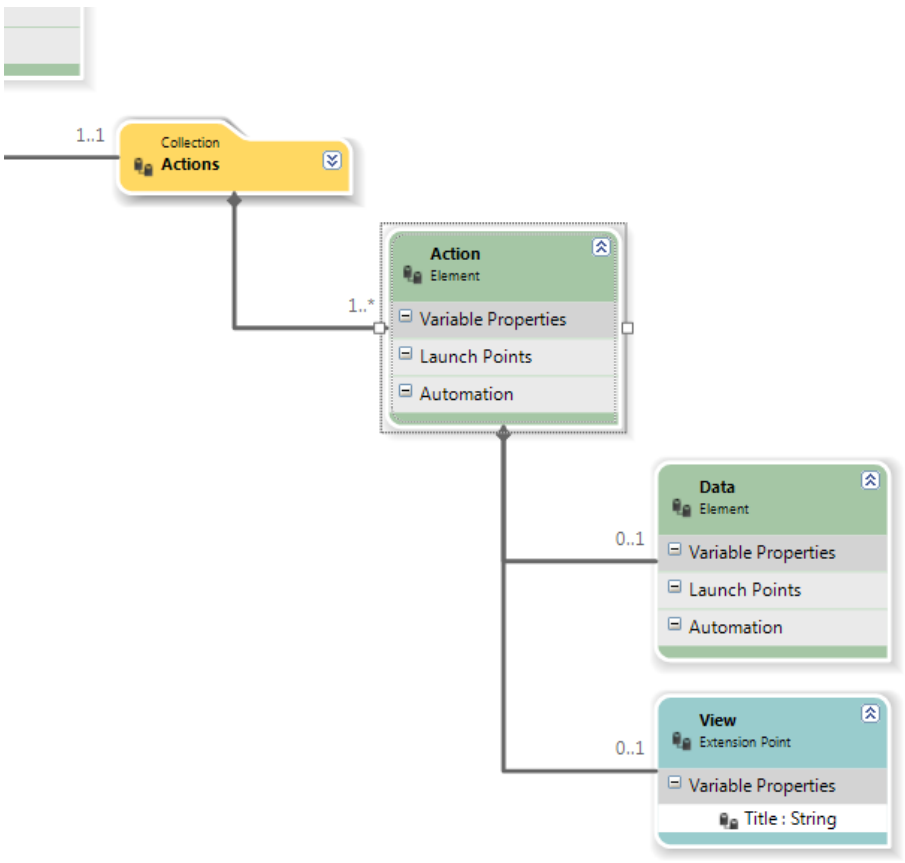
```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.ComponentModel.Composition;
using System.ComponentModel.DataAnnotations;
using System.Globalization;
using System.Linq;
using NuPattern.Extensibility;
using Microsoft.VisualStudio.TeamArchitect.PowerTools.Features.Diagnostics;

namespace MVCApplicationToolkit.Automation.ValidationRules
{
    /// <summary>
    /// Validates that the controller action returns either a view or data, but not both.
    /// </summary>
    [DisplayName("Controller Action Return Type is Valid")]
    [Category("Contoso Web Applications")]
    [Description("Validates that the controller action returns either a view or data, but not both.")]
    [CLSCompliant(false)]
    public class ActionReturnType : ValidationRule
    {
        private static readonly ITraceSource tracer = Tracer.GetSourceFor<ActionReturnType>();

        /// <summary>
        /// Gets or sets the current element to validate.
        /// </summary>
        [Required]
        [Import(AllowDefault = true)]
        public IAction CurrentElement { get; set; }

        /// <summary>
        /// Evaluates the violations for the rule.
        /// </summary>
        public override IEnumerable<ValidationResult> Validate()
        {
            List<ValidationResult> errors = new List<ValidationResult>();

            // Verify all [Required] and [Import]ed properties have valid values.
            this.ValidateObject();

            tracer.TraceInformation("Validating ActionReturnType on current element '{0}'", this.CurrentElement.InstanceName);

            // Action elements must have either a Data or a View child element.
            if ((this.CurrentElement.Data != null && this.CurrentElement.View != null)
                || (this.CurrentElement.Data == null && this.CurrentElement.View == null))
            {
                errors.Add(new ValidationResult(
                    string.Format(CultureInfo.CurrentCulture,
                    "The Action '{0}' must have either one 'Data' element or one 'View' element, and not both.",
                    this.CurrentElement.InstanceName)));
            }

            tracer.TraceInformation("Validated ActionReturnType on current element '{0}' as '{1}'",
                this.CurrentElement.InstanceName, !errors.Any());

            return errors;
        }
    }
}
```

Configure the Custom-Coded Validation Rule

Build the project (**F6**), but do not run or debug it just yet.

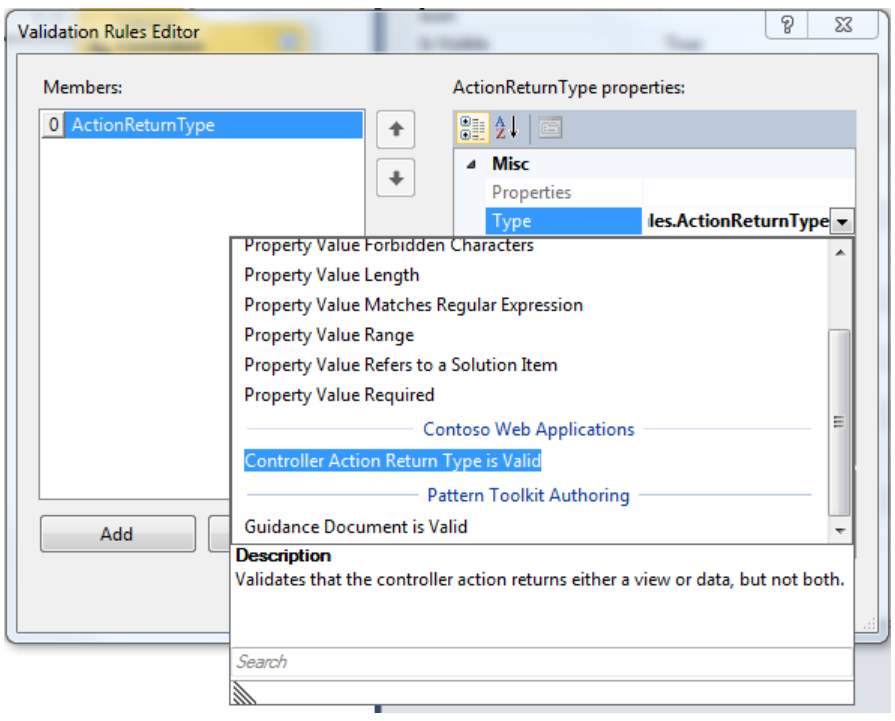
> **Important**: The validation rule must be compiled successfully before it will be recognized by the pattern model, and before you can configure your pattern model to use it.

Select the 'Action' shape in the 'MVCApplication' pattern model, and in the Properties window (**F4**), set the 'Validation Rules' property to include your custom validation rule.



Use the same steps we used before to configure validation for the element in the previous steps of this lab.

When you are done, your 'Validation Rules' property editor for the 'Action' shape should look something like the following.



Press OK to close the 'Validation Rules Editor' dialog.

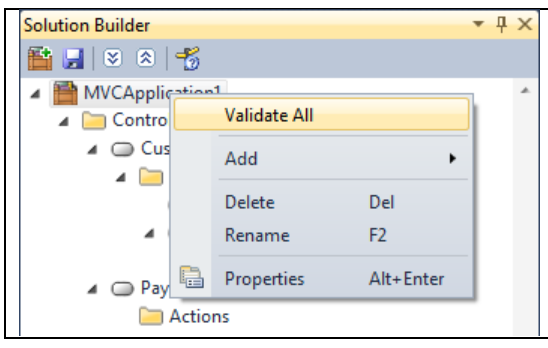
Test the Custom Validation

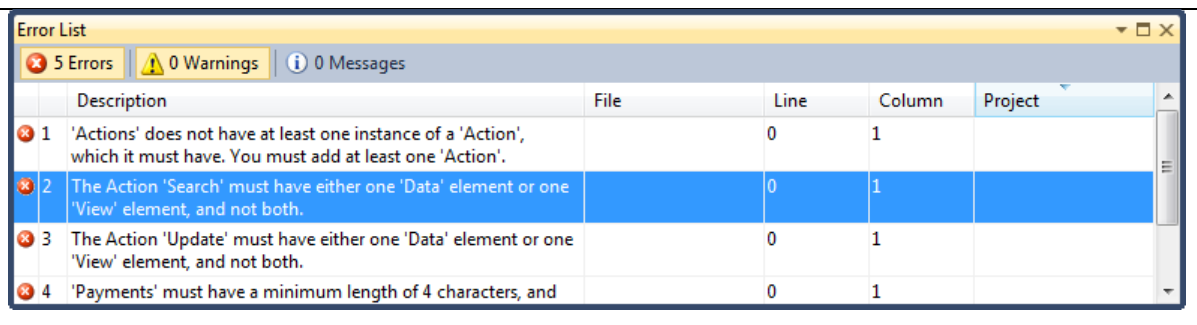
Build and Run the toolkit project (**CTRL + F5**).

In the Experimental Instance of Visual Studio that starts up, open the solution from the previous test cycle.

Ensure that you have an ‘Action’ element that has neither a ‘Data’ element nor a ‘View’ element.

Right-click on the ‘MVCApplication’ element and click the ‘Validate All’ menu.





Now add an instance of either a ‘Data’ element or a ‘View’ and validate the pattern again.

You will not see the validation error for the element because it is now configured correctly.

Close and Save the Visual Studio Experimental Instance.

Part 5: Wizards

Once elements in your pattern model have one or more variable properties defined in the pattern model, a user of your toolkit must be very diligent in updating the property values correctly in order to avoid validation errors. This is usually necessary straight after creating a new instance of an element in Solution Builder. For this reason, providing default values for variable properties is always helpful for a user using your toolkit. However, there are many cases where going an extra step and providing a better initial configuration experience for a user is often very well appreciated and, sometimes necessary to get these properties configured correctly, to avoid stacks of validation errors.

To help provide better guided experiences for your users, you can provide wizards in your toolkit that guide users in configuring the elements in your pattern model more quickly and reliably, improving the accuracy of data entered and minimizing validation errors.

In this part we are going to provide a wizard which combines both default values and validation rules to help users of your toolkit configure new instances of elements in Solution Builder.

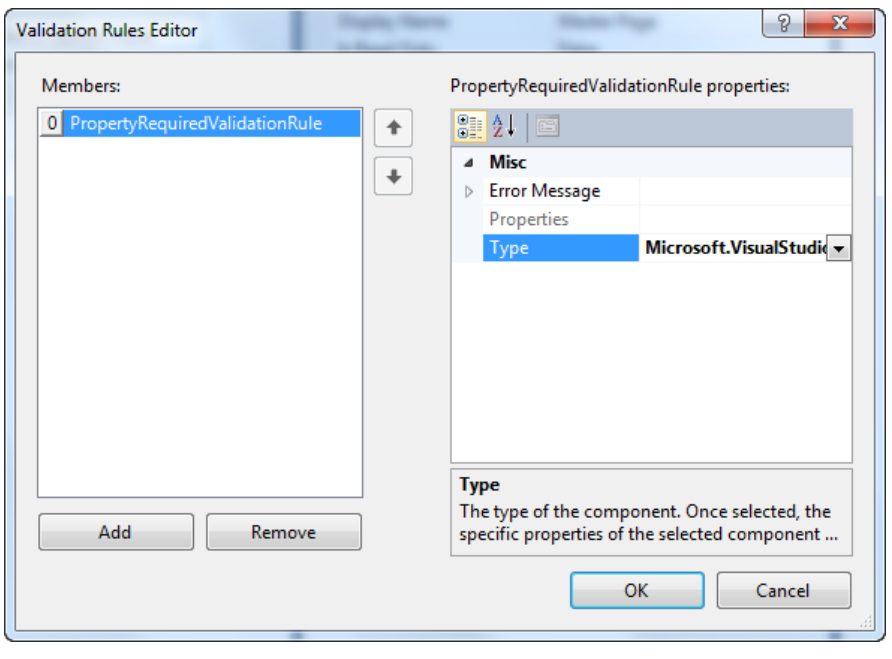
Configure Validation Rules

Close and Save the Visual Studio Experimental Instance.

In the 'SearchView' toolkit project, open the pattern model.

Select the 'MasterPage' variable property in the pattern element, and in the Properties Window (F4), click the ellipsis for the 'Validation Rules' property.

Add a new validation rule using the "Property is Required" type.



Ensure that the 'Default Value' property is set to "~/Views/Shared/Site.Master"

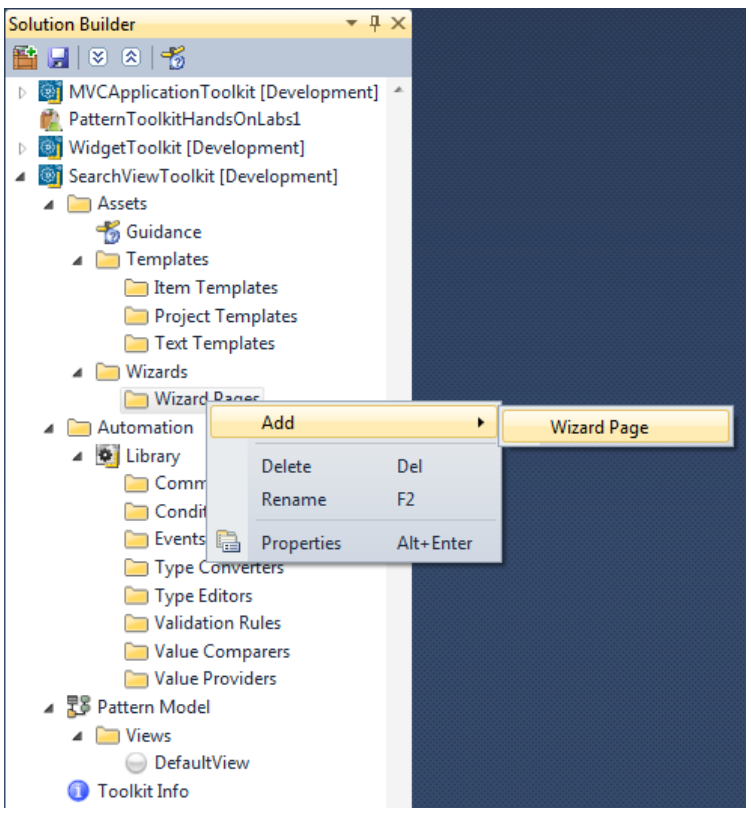
> **Note**: The 'Default Value' for this property was defined at the end of the lab on Extension Points.

Create a new Properties Page

Close and Save the Visual Studio Experimental Instance.

In the 'SearchView' toolkit project, open the 'Solution Builder' window (**CTRL + W, H**) and locate the 'Wizard Pages' element.

Right-click on 'Wizard Pages' element and add a new 'Wizard Page' named "ViewProperties".



A new custom wizard page XAML document will be added to your solution and the XAML will be opened in the editor window.

> **Note**: Take a look at the XAML that was added. It contains comments to help you determine what you need to do to implement the wizard page.

After you are familiar with the code, delete all of it and replace it with the code below.

> **Important**: Adjust the code for any variations in namespace or property names you have in your pattern model.

```
<Page
    x:Class="SearchViewToolkit.Assets.Wizards.Pages.ViewProperties"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:p="http://schemas.microsoft.com/nupattern/2012/xaml/"
    xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
    xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
    mc:Ignorable="d" d:DesignHeight="400" d:DesignWidth="600"
    Title="View Properties" >
    <Grid>
        <Grid.RowDefinitions>
            <RowDefinition Height="Auto" />
            <RowDefinition Height="Auto" />
            <RowDefinition Height="Auto" />
        </Grid.RowDefinitions>
        <Grid.ColumnDefinitions>
            <ColumnDefinition Width="120"/>
            <ColumnDefinition Width="400"/>
        </Grid.ColumnDefinitions>

            <Label Grid.Column="0" Grid.Row="0">Name:</Label>
            <p:ValueEditor Name="InstanceNameEditor" Value="{Binding InstanceName, ValidatesOnDataErrors=True}" Grid.Column="1" Grid.Row="0"/>
        <Label Grid.Column="0" Grid.Row="1">Master Page:</Label>
        <p:ValueEditor Name="MasterPageEditor" Value="{Binding MasterPage, ValidatesOnDataErrors=True}" Grid.Column="1" Grid.Row="1"/>
        <Label Grid.Column="0" Grid.Row="2">Content Placeholder Id:</Label>
        <p:ValueEditor Name="ContentPlaceholderIdEditor" Value="{Binding ContentPlaceholderId, ValidatesOnDataErrors=True}" Grid.Column="1" Grid.
Row="2"/>
       </Grid>
</Page>
```

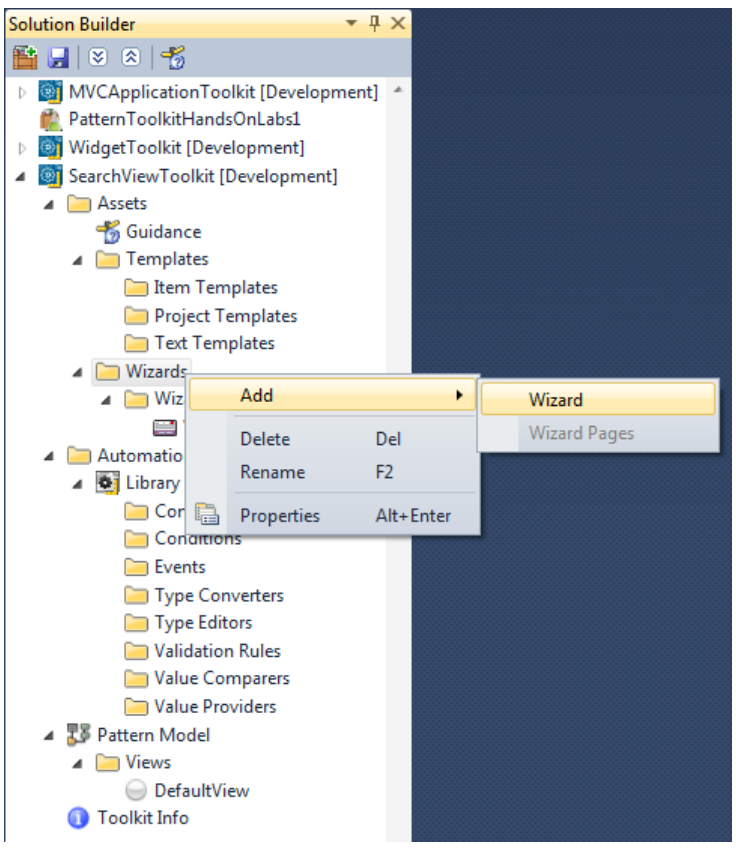
Create a New Configuration Wizard

Wizards in pattern toolkits are defined as a collection of Wizard Pages. Once you have created one or more Wizard Pages, you must then combine them together into one or more Wizards.

> **Note**: A single Wizard Page can be reused in one or more Wizards.

In the 'SearchView' toolkit project, open the 'Solution Builder' window (**CTRL + W, H**) and locate the 'Wizards' element.

Right-click on 'Wizards' element and add a new 'Wizard' named "ConfigureSearchView".



A new custom wizard XAML document will be added to your solution and the XAML will be opened in the editor window.

> **Note**: Take a look at the XAML that was added. It contains comments to help you determine what you need to do to implement the wizard.

After you are familiar with the code, delete all of it and replace it with the code below.

> **Important**: Adjust the code for any variations in namespace or property names you have in your pattern model.

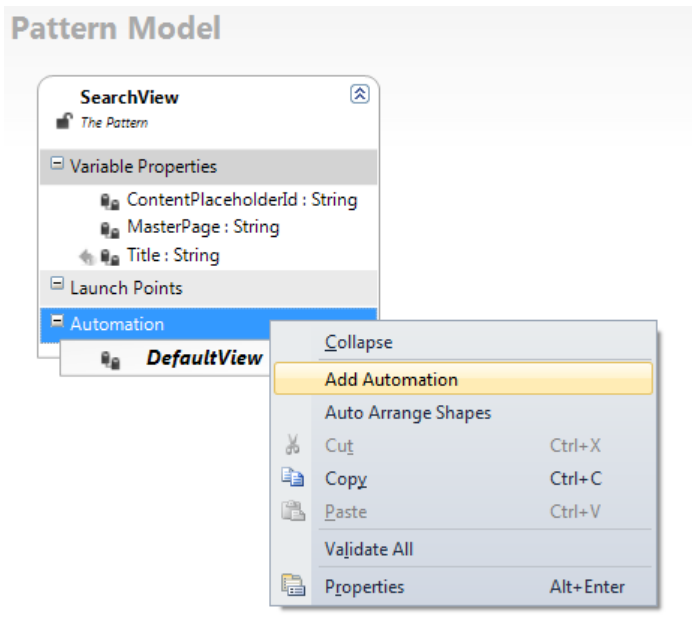
```
<p:WizardWindow
    x:Class="SearchViewToolkit.Assets.Wizards.ConfigureSearchView"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:p="http://schemas.microsoft.com/nupattern/2012/xaml/"
    xmlns:l="clr-namespace:SearchViewToolkit.Assets.Wizards.Pages"
    xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
    xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
    mc:Ignorable="d" Height="400" Width="600"
    Title="Configure Search View">
    <p:WizardWindow.Pages>
        <l:ViewProperties />
    </p:WizardWindow.Pages>
</p:WizardWindow>
```

Configure the Wizard

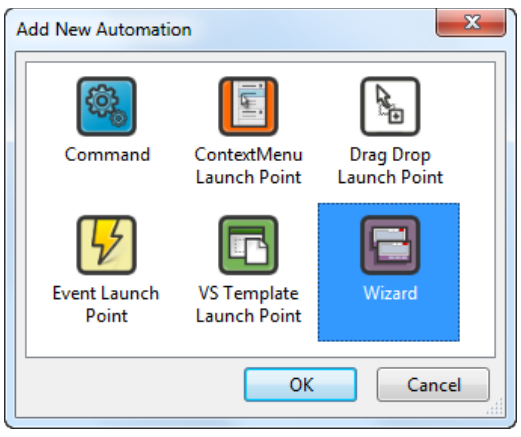
Build the project (**F6**), but do not run or debug it just yet.

> **Important**: The wizard must be compiled successfully before it will be recognized by the pattern model, and before you can configure your pattern model to use it.

In the 'SearchView' pattern model, right-click on the 'Automation' compartment and choose 'Add Automation'.



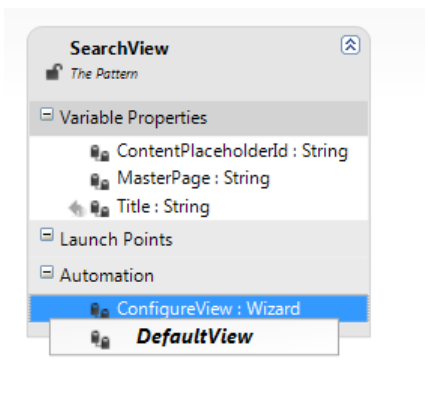
The 'Add New Automation' dialog appears. Choose 'Wizard' and click OK.



Rename the new wizard to "ConfigureView".

> **Tip**: You can either select the wizard and start typing, or find the 'Name' property in the Properties window

You may also want to resize the 'SearchView' shape so that you can see all the text.

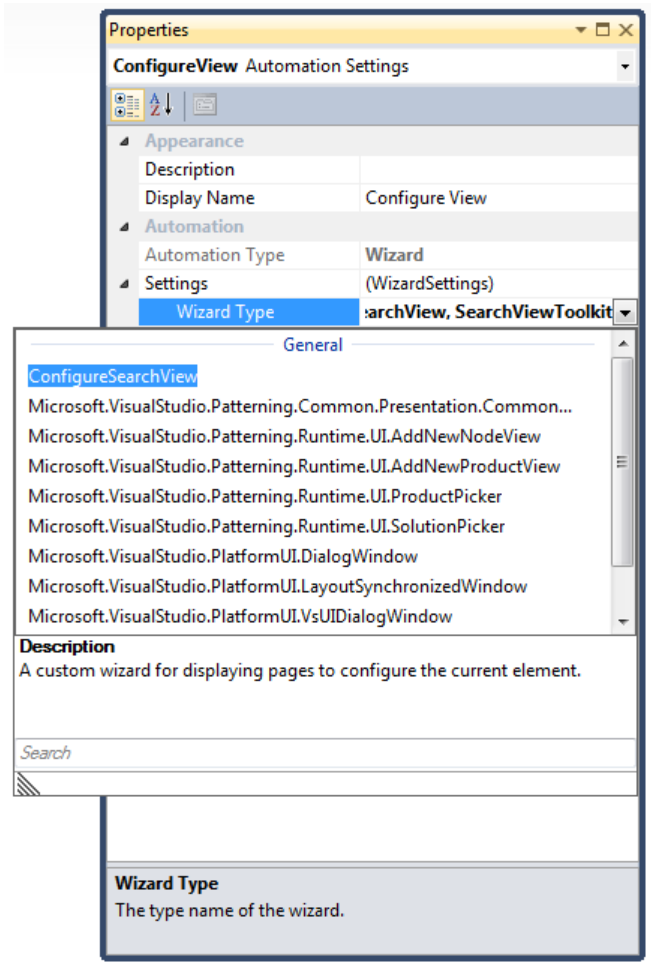


Select the wizard, and look at the properties in the Properties window (**F4**).

Expand the 'Settings' section and you will see the various properties associated with configuring a 'Wizard.

You must select the 'Wizard Type' property.

Select "ConfigureSearchView" from the list.



Configure the Launch Points

We are going to add two launch points to run this wizard. One that displays the wizard when a 'Search View' element is first created, and another that provides a menu to configure the element manually at any time.

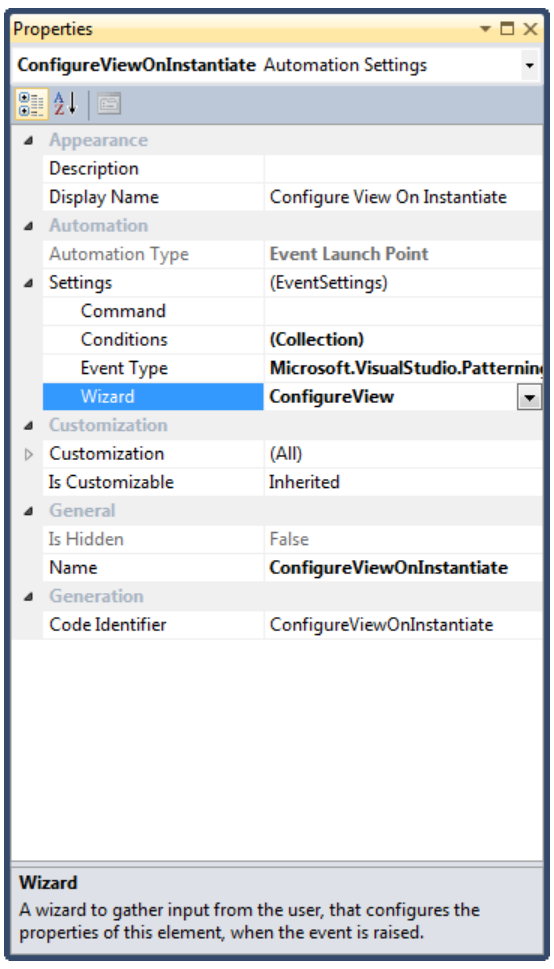
Configure the Instantiate Event

Right-click on the 'Launch Points' compartment of the 'SearchView' element in the Pattern Model.

Click on 'Add Automation', and choose 'Event Launch Point' this time.

Rename the launch point "ConfigureViewOnInstantiate"

Set the 'Event Type' property to "Element is Instantiated", and select the "ConfigureView" for the 'Wizard' property.



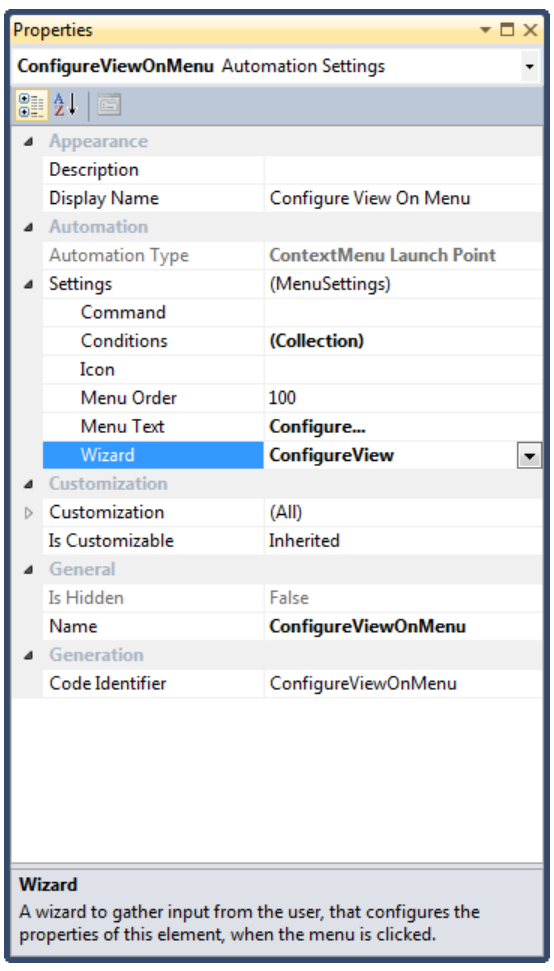
Configure the Menu

Right-click on the 'Launch Points' compartment of the 'SearchView' element in the Pattern Model.

Click on 'Add Automation', and choose 'Context Menu Launch Point' this time.

Rename the launch point "ConfigureViewOnMenu".

Set the 'Menu Text' property to "Configure...", and select the "ConfigureView" for the 'Wizard' property.

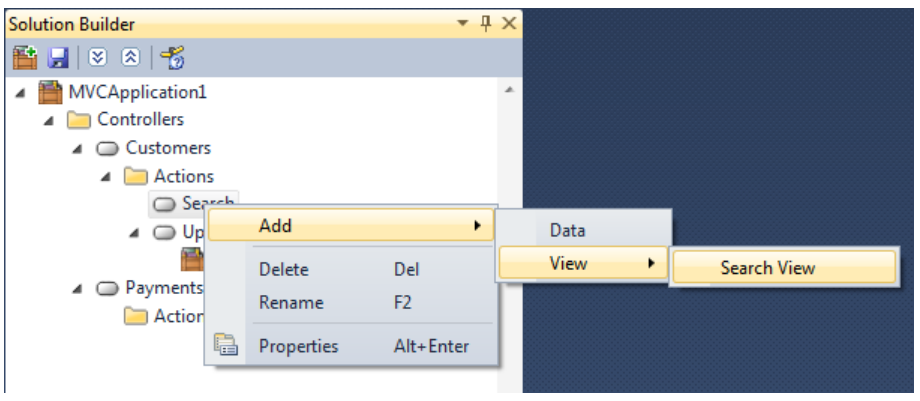


Test the Wizard

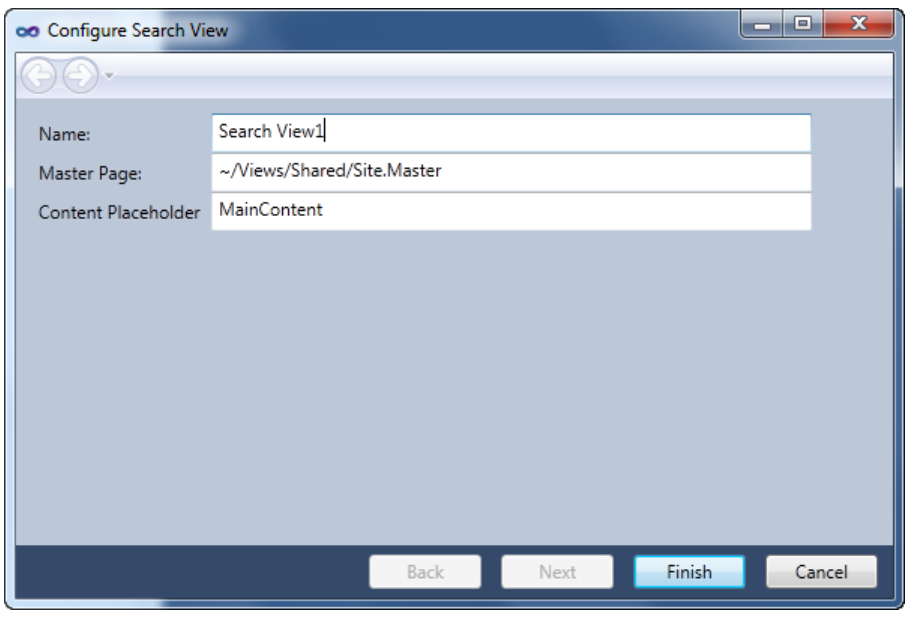
Build and Run the toolkit project (**CTRL + F5**).

In the Experimental Instance of Visual Studio that starts up, open the solution from the previous test cycle.

Ensure that you have an 'Action' element that has neither a 'Data' element nor a 'View' element, then right-click on the 'Action' element and add a new 'Search View' element.

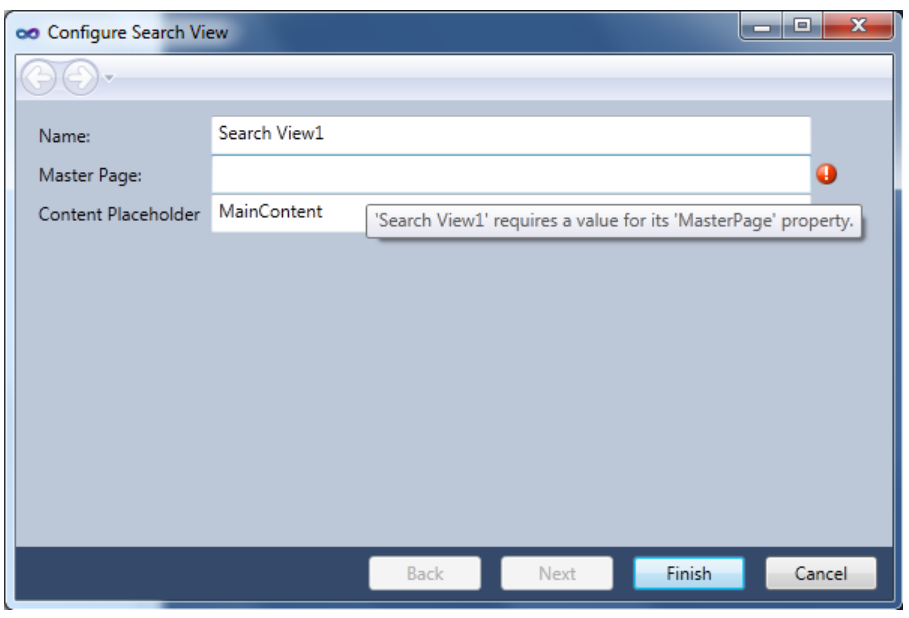


Notice that when a new instance of a 'Search View' is now created, the user is provided a handy wizard to configure the view.

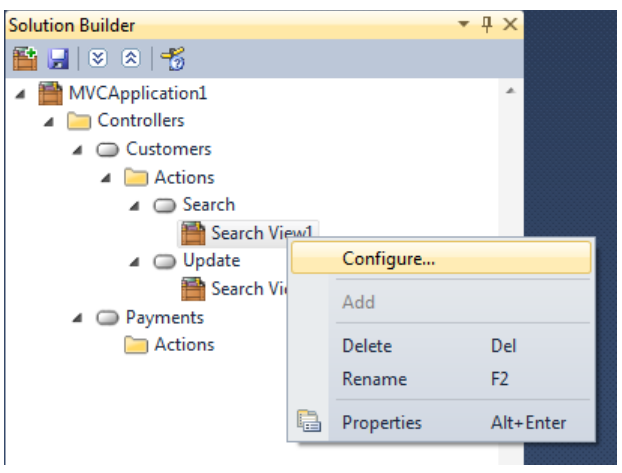


> **Note**: This wizard is about as basic as it gets, and for a real toolkit you would spend a little more time with the presentation of the data in the dialog.

Now, blank out the value for the 'Master Page' property and notice that the wizard honors any validation rules defined on the element.



Now, close the Wizard, and right-click on the 'Action' element in Solution Builder.

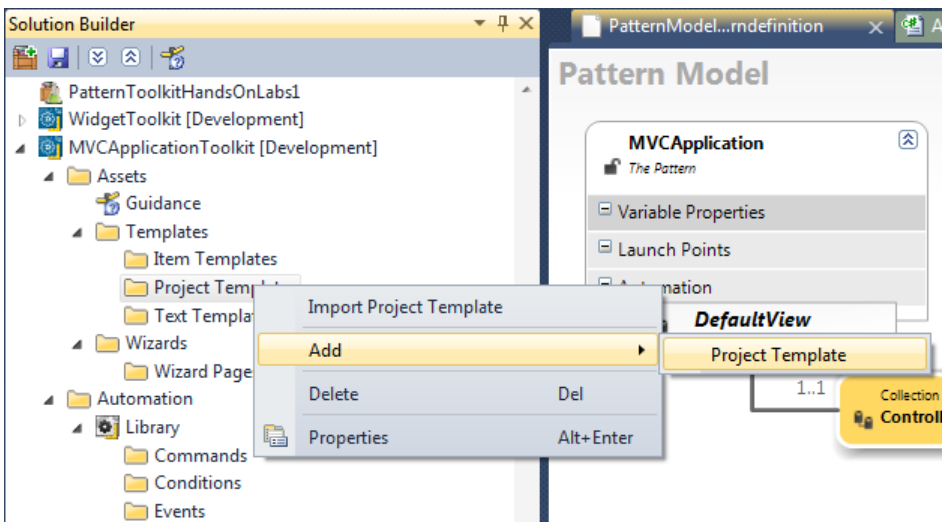


Notice the 'Configure...' menu, which when clicked shows the wizard again.

Close and Save the Visual Studio Experimental Instance.

Part 6: Working with Project Templates and Item Templates

In the 'Getting Started' lab we added project and item templates by adding a new template from 'Solution Builder'.



In this part of the lab, we will go through the process of creating our own project templates, or reusing existing ones, and using the 'Import Project Template' command to import it.

Visual Studio actually does a very good job of helping you create your own Project and Item templates, which is very useful for capturing and harvesting existing solutions and turning them into Pattern Toolkits.

Export a New MVC Project Template from Visual Studio

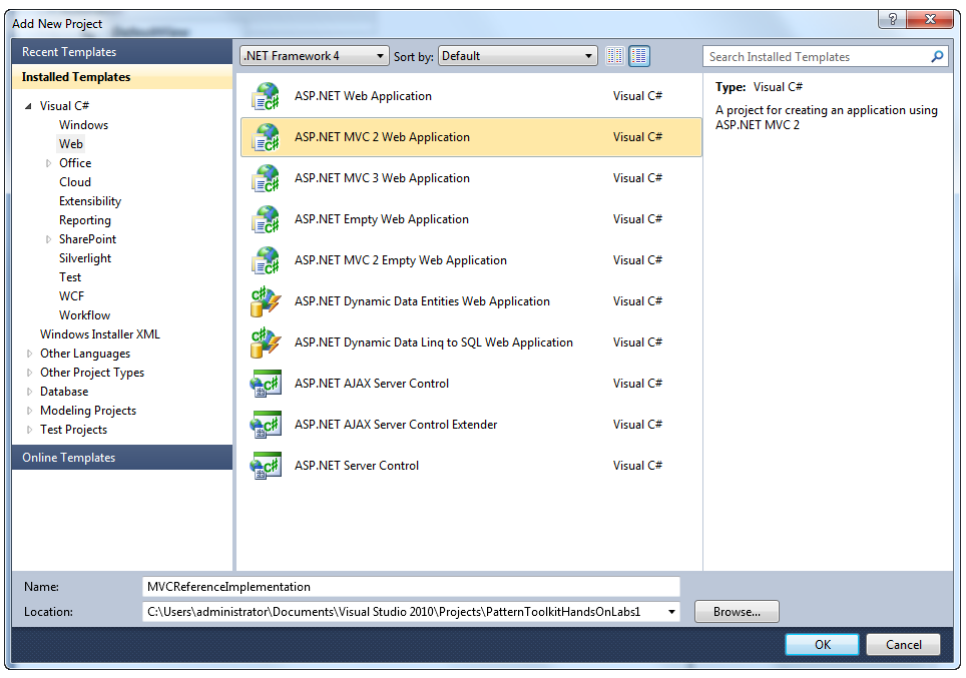
The ‘Add | Project Template’ command works well if you only need a simple C# class library, but in most cases you will want your toolkit to unfold a reference project with additional assembly references, folder structures, specific files, and even possibly a different project type.

The first step is to create a project that is a good example of where you want your users to start.

In this case we are going to create a template starting with an ASP.NET MVC 2 Web Application.

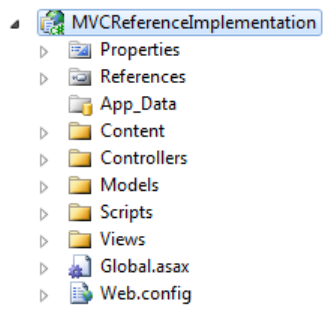
Create the reference project in the current solution using the ‘File | Add | New Project’ menu, and chose the C# ‘ASP.NET MVC 2 Web Application’ project type.

Call the new project “MVCReferenceImplementation”.

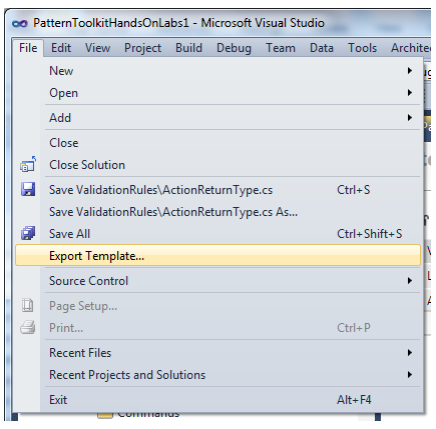


> **Note**: You may choose to use a more recent version of ASP.NET MVC, but you will need to install that new SDK first.

You will notice that this project template includes a wizard that asks you to include a ‘Test Project’, and it includes a set of existing files, and folder structure.

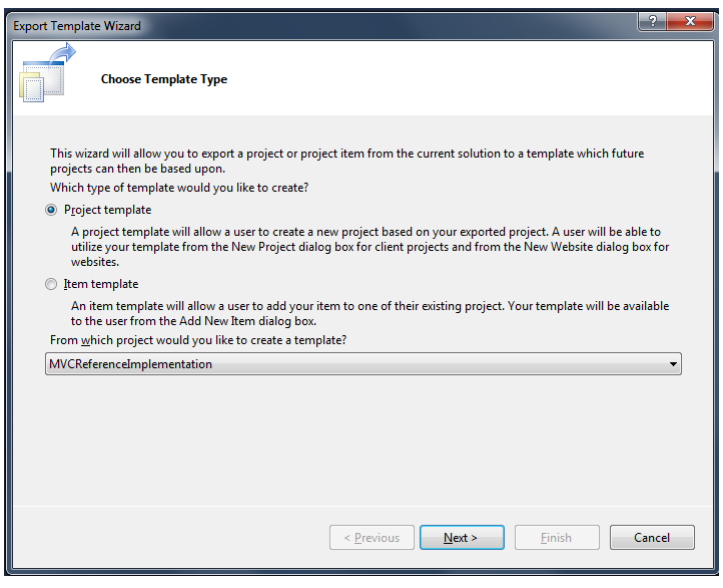


Now let’s export this template click the ‘File | Export Template’ menu.

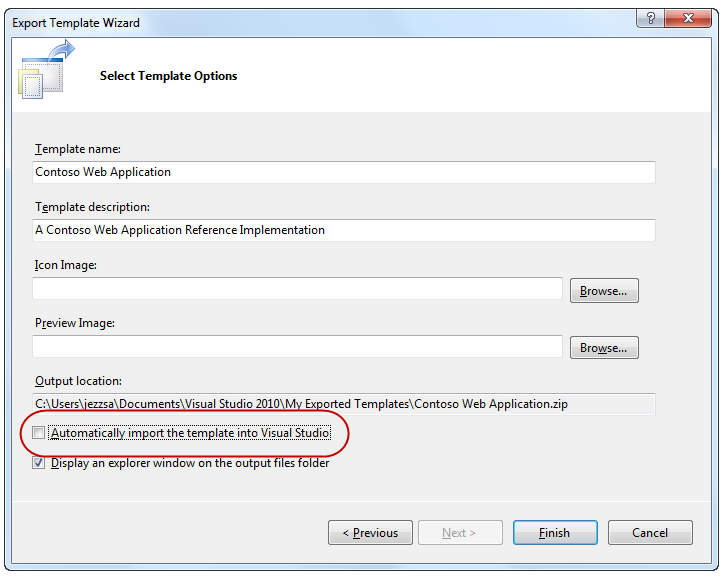


The ‘Export Template Wizard’ will appear.

Accept the default “Project Template” selection (since we are exporting a project template), and choose the project in your solution that you want to use for creating the template. Click Next



> **IMPORTANT**: Uncheck the “Automatically import the template into Visual Studio” check box.
> Our toolkit project will register this template in Visual Studio. Leaving this checked will cause a duplicate project template to be registered with Visual Studio on our development machine.



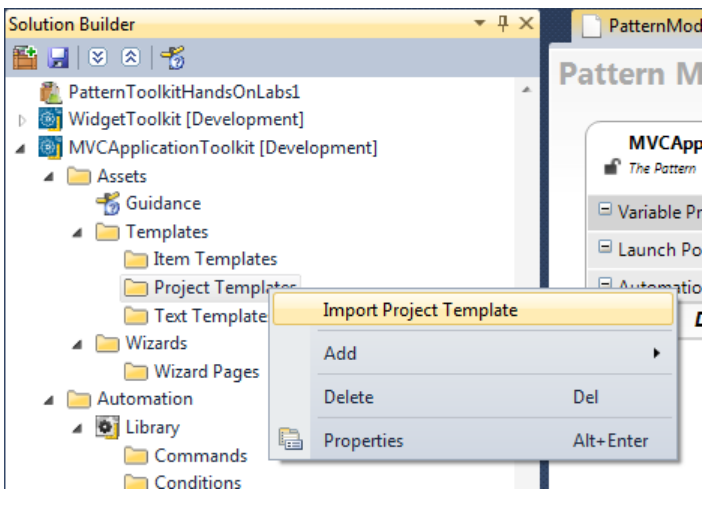
Click Finish. A Windows Explorer window will open to the folder where the template was exported.

You can close this window.

Import the Project Template into the Toolkit Project

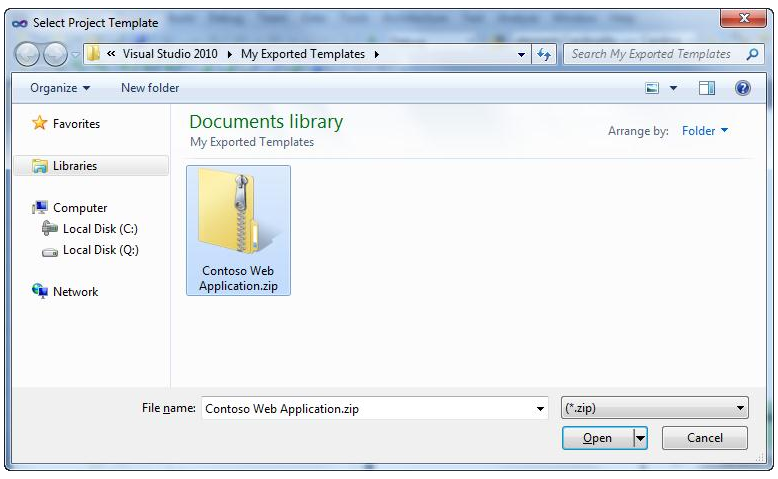
In the 'MVCApplication' toolkit project, open the 'Solution Builder' window.

Right-click on the 'Project Templates' element and click 'Import Project Template'.



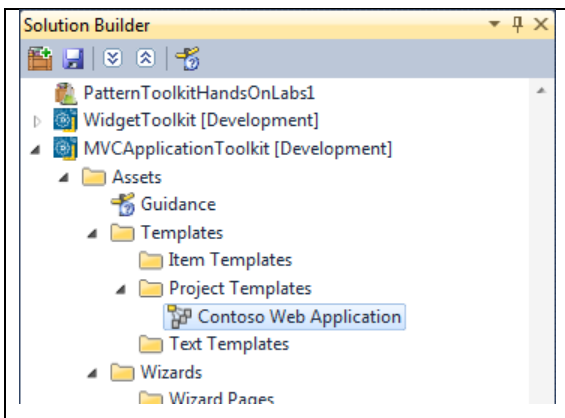
The 'Select Project Template' window will open to the default folder for storing exported Visual Studio templates.

> **Note**: If you exported your template to another folder, navigate to that folder now.

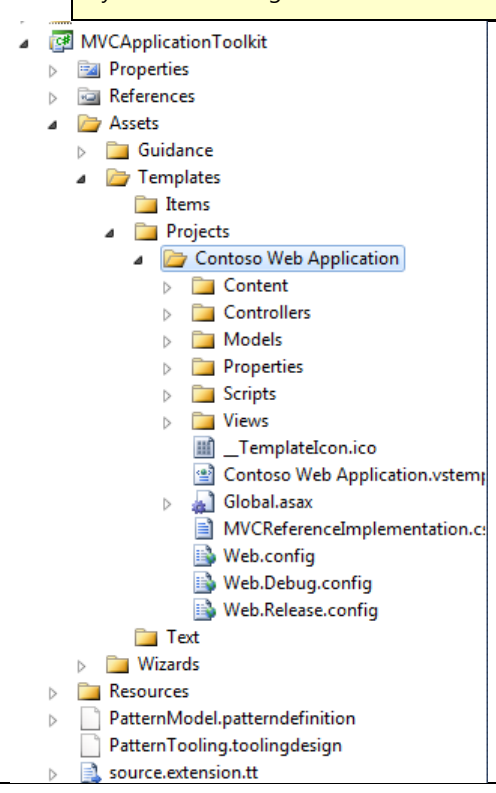


Select the 'Contoso Web Application.zip' template you exported, and click Open.

You will now see your project template under the 'Project Templates' folder.



> **Note**: You can also see the contents of the template in the toolkit project, by double-clicking on the element in 'Solution Builder'



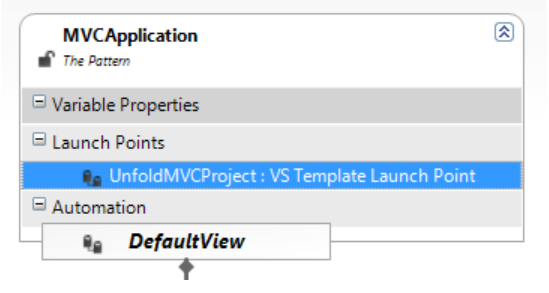
> **Tip:** You can also import one of the built-in Visual Studio project templates instead of creating your own. Many of them are located in the folder: %programfiles(x86)%\Microsoft Visual Studio 10.0\Common7\IDE\ProjectTemplates. (Use %programfiles% if you are on a 32 bit machine.)

Configure the Project Template to Unfold with the Toolkit

At this point we have added the project template to the toolkit, but we still need to configure it to unfold into the user's solution when the user adds a toolkit.

Follow the steps in the 'Getting Started' hands-on lab, in Part 2: Add Project and Item Templates.

The result will be a new 'VS Template Launch Point' on the pattern element that is configured to unfold the project template when a user adds the toolkit to their solution.

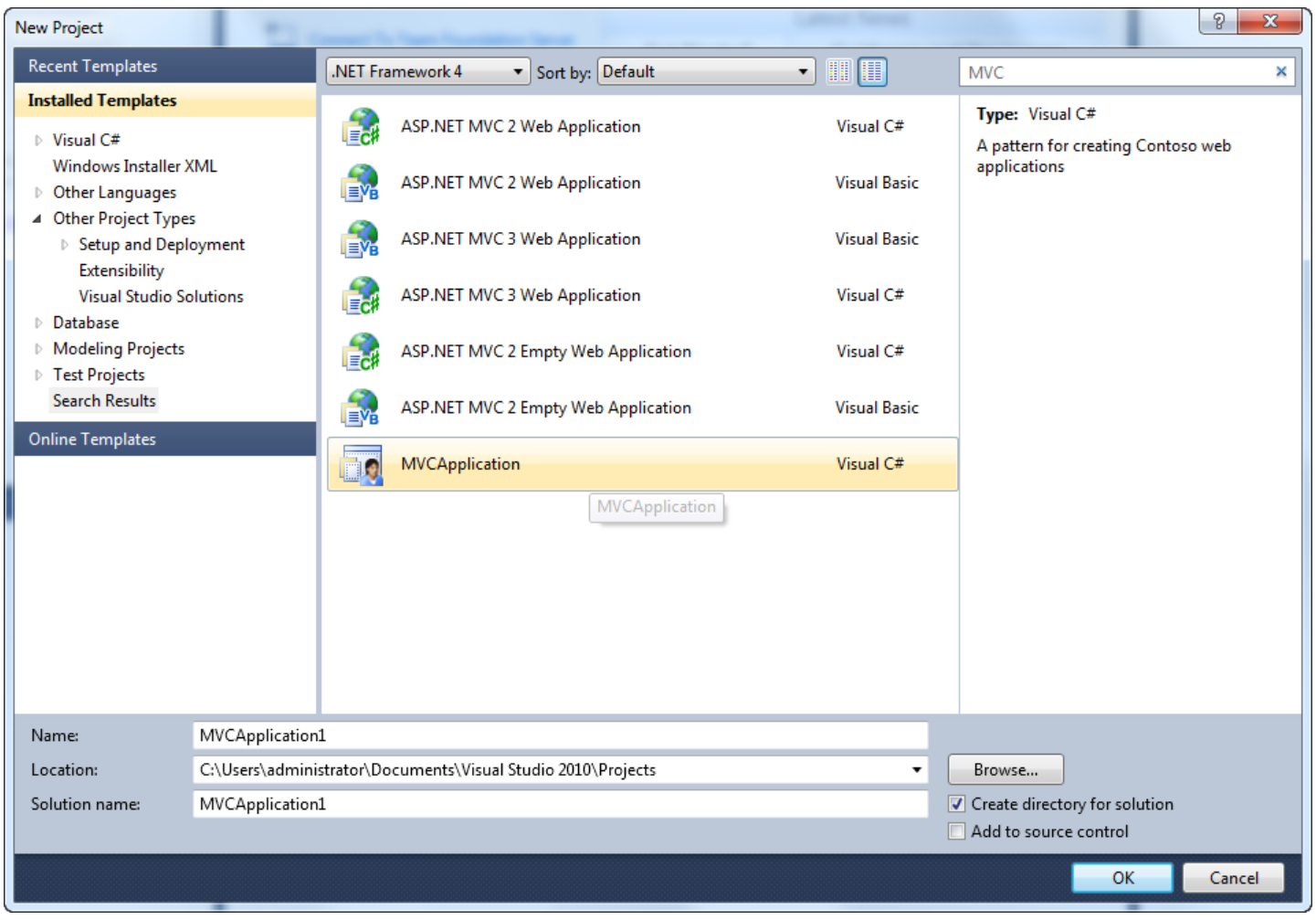


Test Custom Project Template

Build and Run the toolkit project (**CTRL + F5**).

In the Experimental Instance of Visual Studio that starts up, open the 'File | Add new Project' dialog (**CTRL + SHIFT + N**) and search for the term "MVC".

You should see your new project template.



Clicking OK will create a new solution containing a new project using the project template.

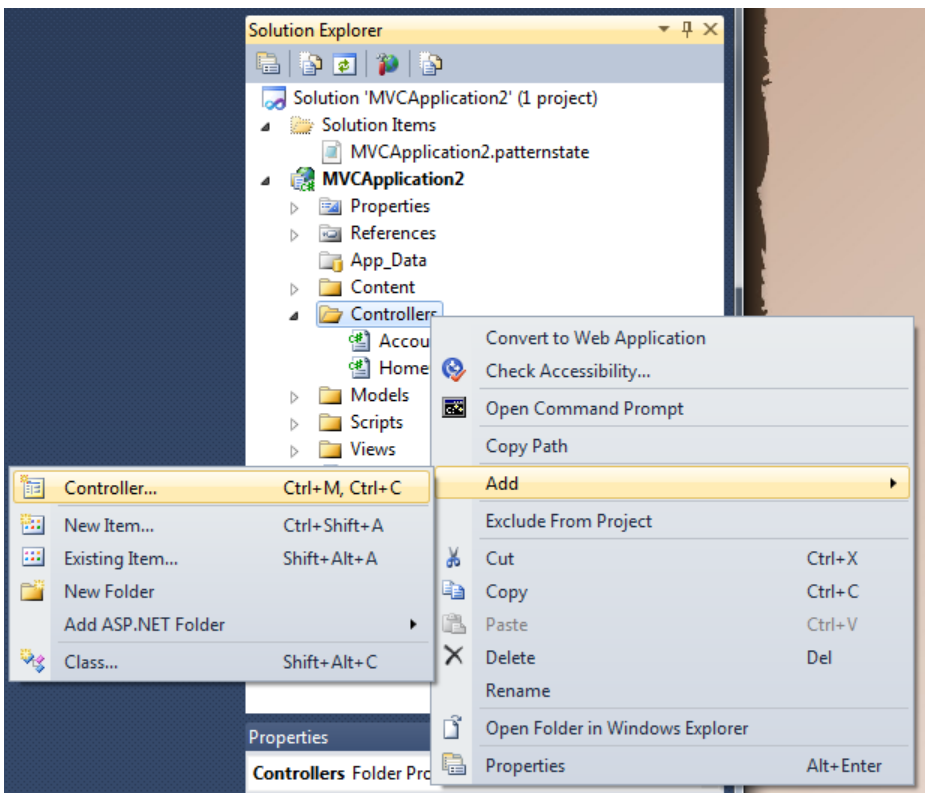
Close and Save the Visual Studio Experimental Instance.

Create a New Controller Item Template

Take a look at the project that is now unfolded with the toolkit. It contains a folder for 'Controllers'. Every time the user adds a 'Controller' element in 'Solution Builder', we want to add a new 'Controller' class in the 'Controllers' folder in 'Solution Explorer'. We will first create a new controller.cs item template.

Open the Visual Studio Experimental instance, and open the project created in previous step.

Right-click on the 'Controllers' folder in 'Solution Explorer' and add a new 'Controller' class called "ExampleController"



Delete all code except for the class declaration.

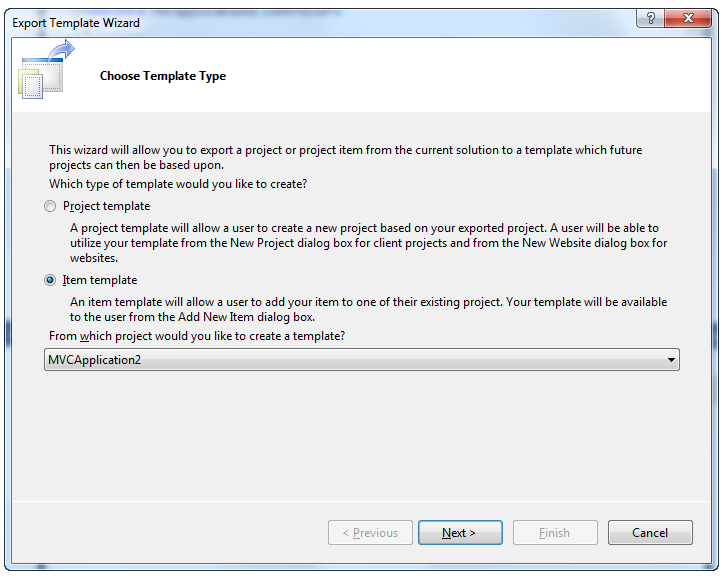
Add the 'partial' key word to the class declaration. (This allows you to generate partial implementations for controller classes.)

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Web.Mvc;

namespace MVCApplication2.Controllers
{
    public partial class ExampleController : Controller
    {
    }
}
```

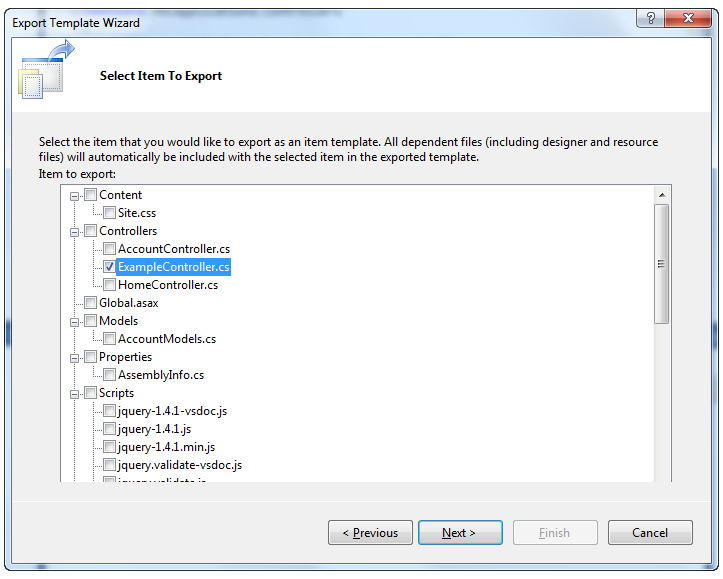
Export the Template

Click the 'File | Export Template'.

In the 'Export Template Wizard' that appears, change the selection to 'Item Template', and make sure the project that contains the controller you just added is selected below.

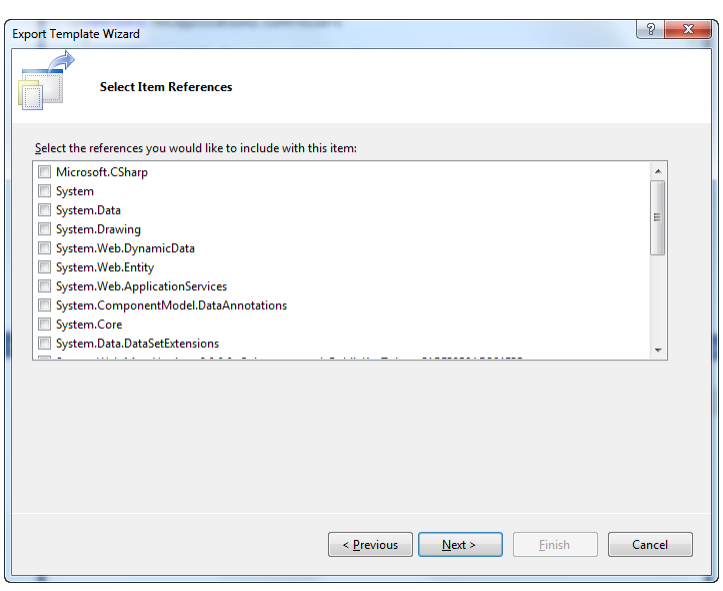


Click Next. In the next wizard page find the controller class you just created and select the check box next to it.



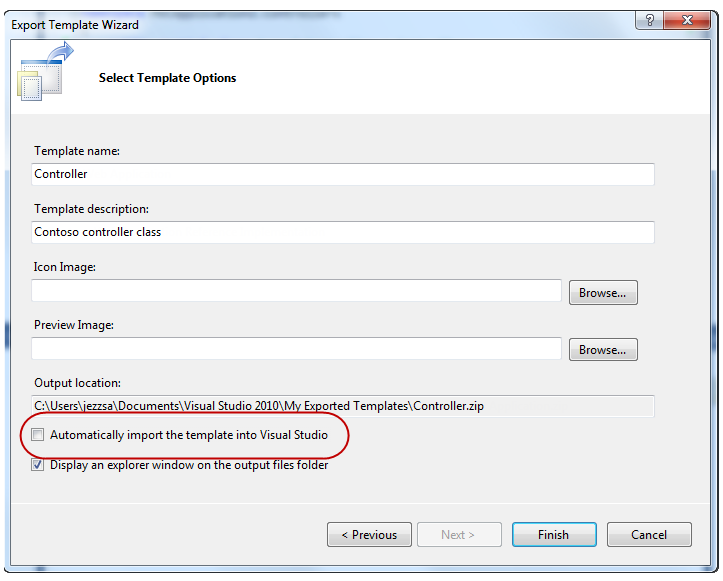
Click Next. In the 'Select Item References' wizard page, you do not need to add any references because the references we need are already part of the project template.

> **Note**: Visual Studio is allowing you to select any references that must be added to the project in order for the class unfolded by your item template to compile.



Click Next. In the next wizard page give the template a friendly name and description.

> **Note**: This name and description will not appear in Visual Studio because this template will not be available from the 'Add New Item' dialog.
> The only way for users to invoke the template will be by adding a new item to the 'Controllers' folder in 'Solution Builder'.

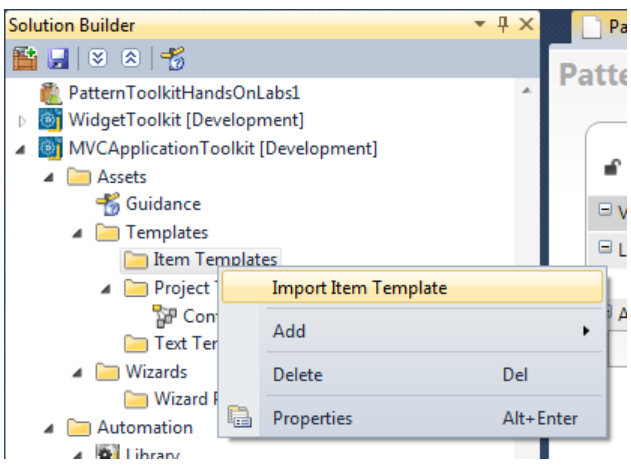


Click Finish.

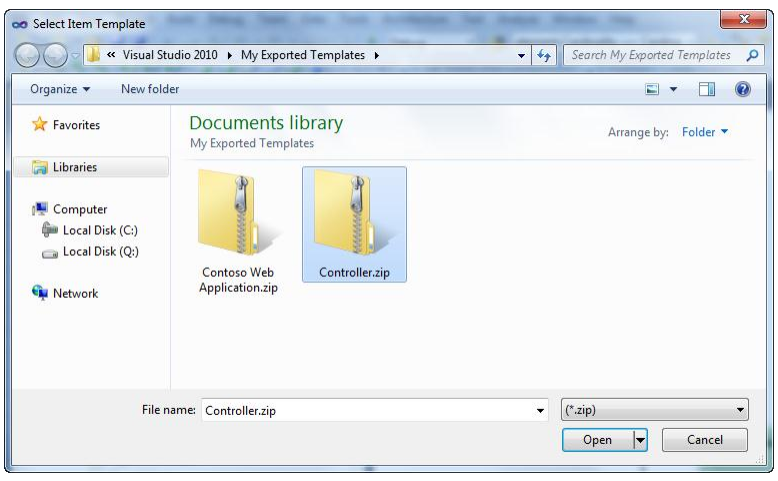
Close and Save the Visual Studio Experimental Instance.

Import the Item Template into the Toolkit Project

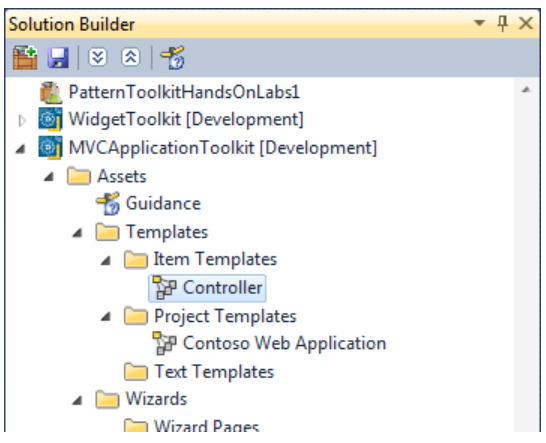
In the 'MVCApplication' toolkit project, right-click on the 'Item Templates' element in 'Solution Builder' and click 'Import Item Template'.

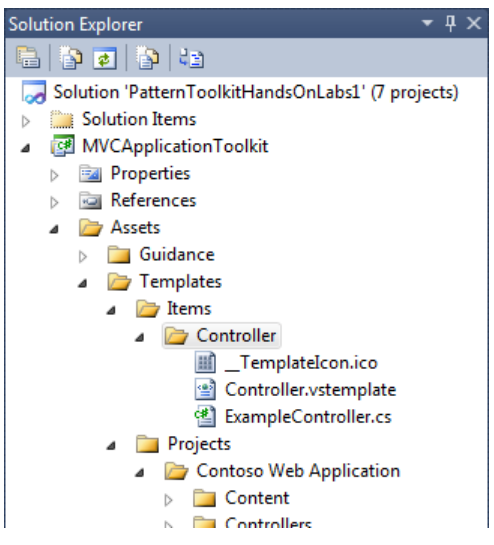


In the 'Select Item Template' dialog that appears, find the template we just exported and click Open.



A new template named 'Controller' will appear under 'Item Templates' in 'Solution Builder', and the template is unpacked in 'Solution Explorer'





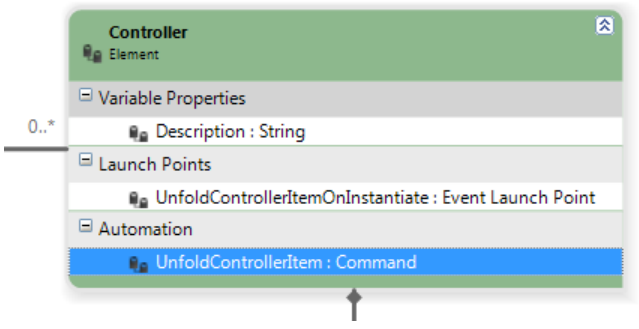
Configure the Item Template to Unfold with the Controller Node

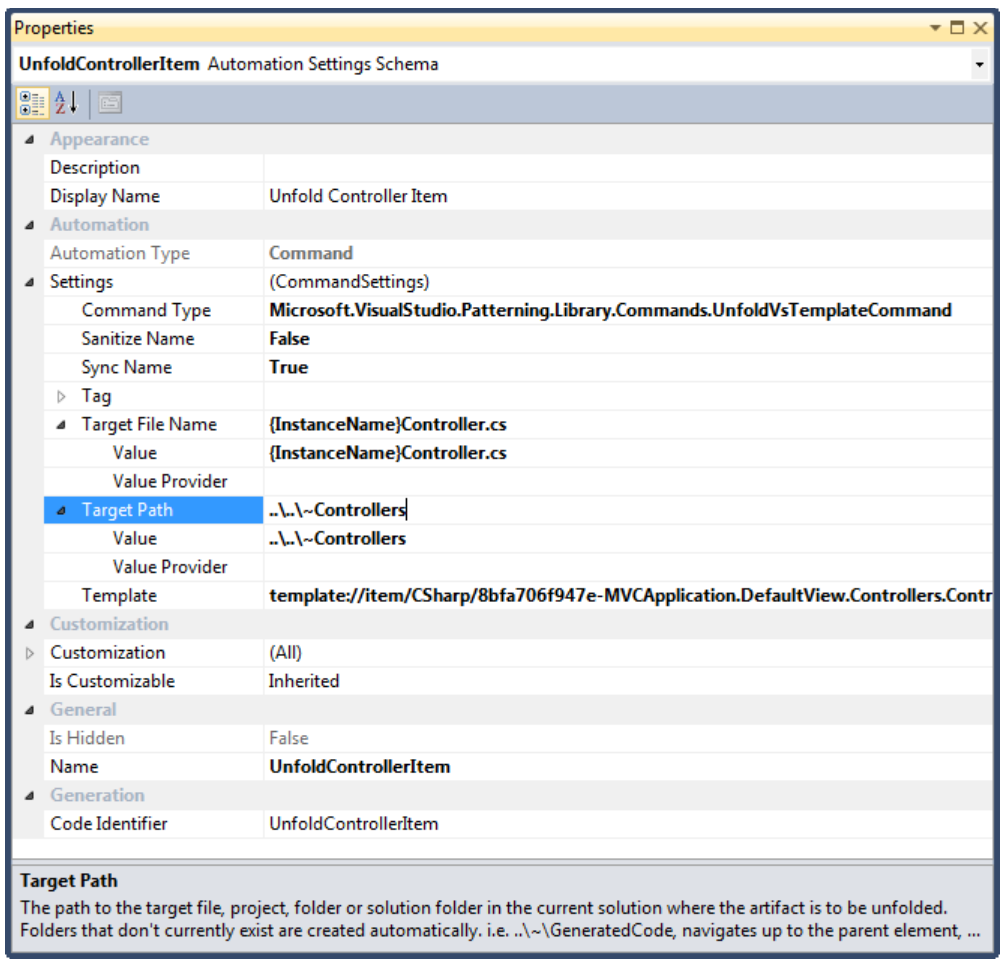
At this point we have added the item template to the toolkit, but we still need to configure it to unfold into the user's solution when the user adds a 'Controller' element.

Follow the steps in the 'Getting Started' hands-on lab, in Part 2: Add Project and Item Templates.

The result will be a new 'Command' and 'Event Launch Point' pair on the 'Controller' element that is configured to unfold the project template when a user adds the element to their solution.

The configured 'Controller' element will look like this, and properties settings for the 'UnfoldControllerItem' Command are also shown below.





Take a look at the command settings above.

- The 'Sync Name' property is set to "true" so that if the user changes the element name in 'Solution Builder', the class name will be changed in 'Solution Explorer'.
- The 'Target File Name' property is set to "{InstanceName}Controller.cs" so that the file name will be the 'Controller' element's name property added by the user, plus the fixed text "Controller.cs." (This is an ASP.NET MVC naming convention).
- The 'Target Path' property is set to "..\..\~\Controllers" so that all controller classes will be added to 'Controllers' sub folder of the project in 'Solution Explorer' which is created by the grandparent element of the 'Controller' element. (In this case, the grandparent element is the pattern element).

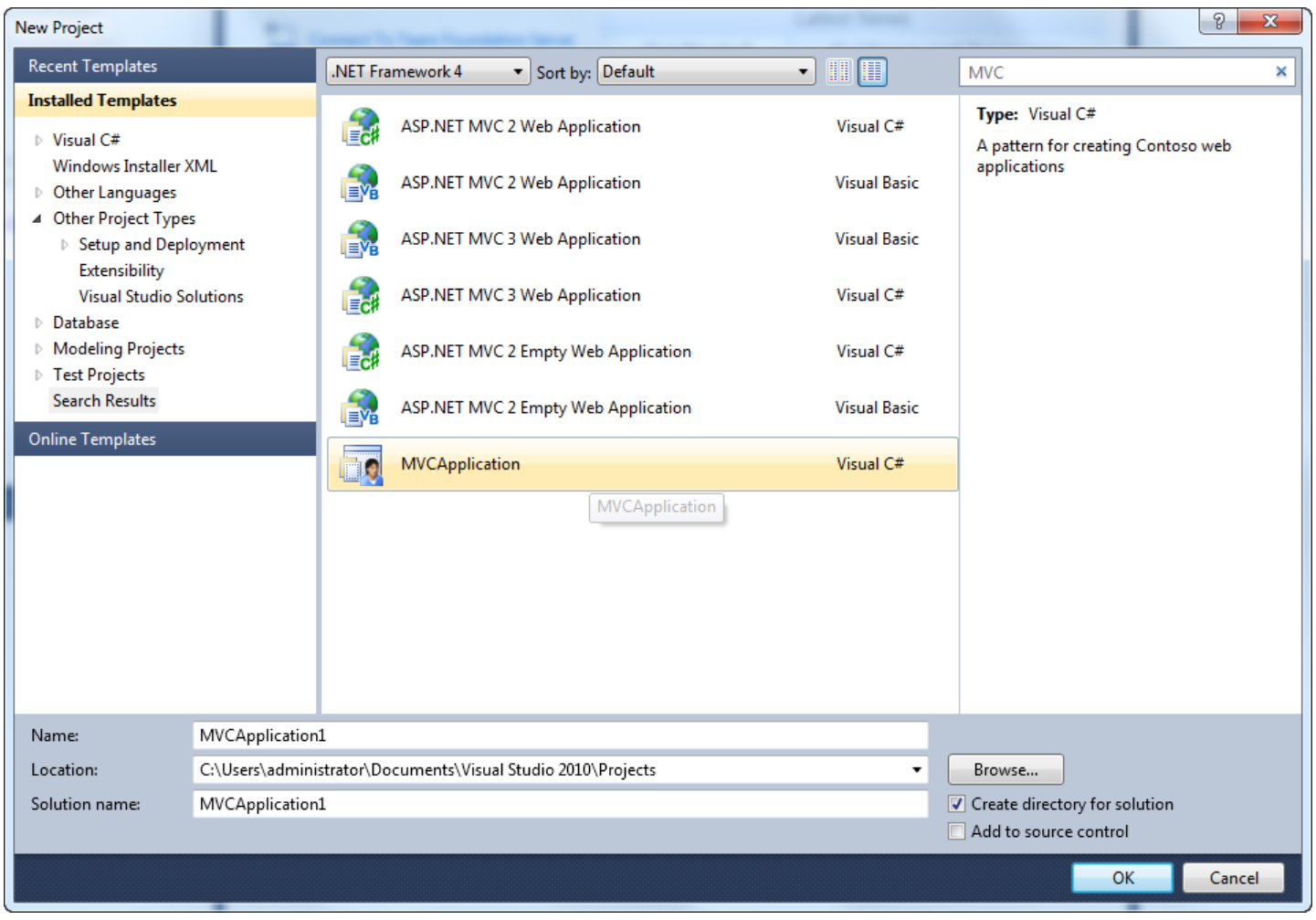
> **Note**: See the guidance for pattern toolkits for details on using the tilde (~) character for more information on how the toolkit finds the owner of the 'Controllers' folder.

Don't forget to create a launch point that will call the command when the 'Controller' element is created, and use the 'Element is Instantiated' event type.

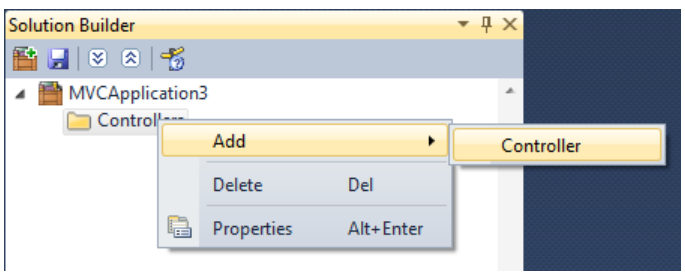
Test Custom Item Template

Build and Run the toolkit project (**CTRL + F5**).

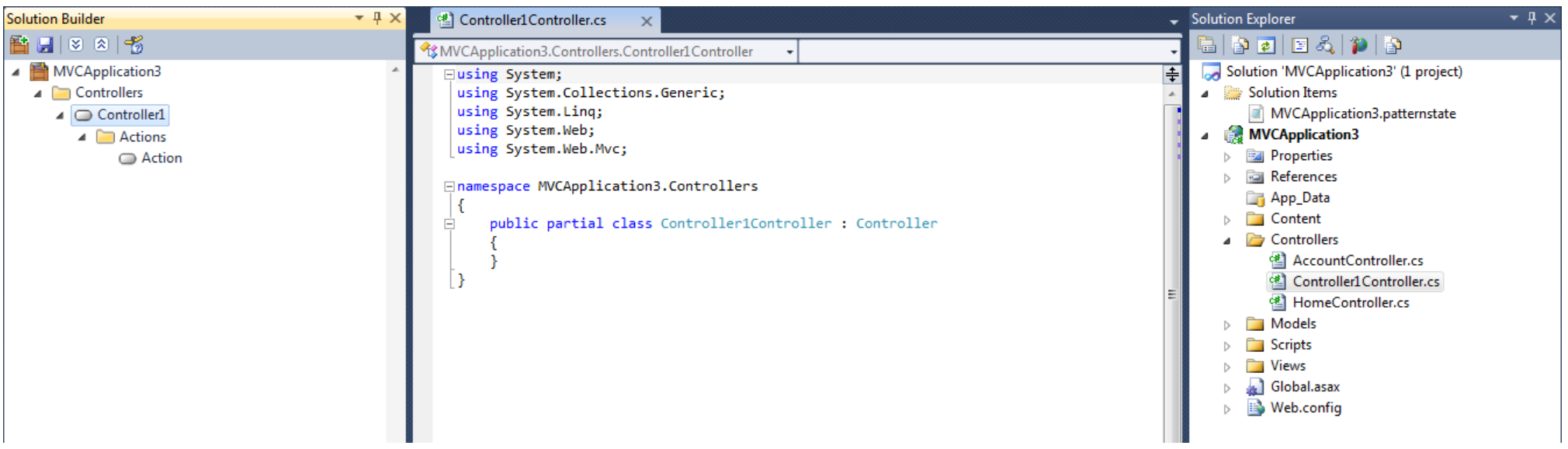
In the Experimental Instance of Visual Studio that starts up, open the 'File | Add new Project' dialog and search for the term "MVC". And create a new project.



Add a new 'Controller' element in 'Solution Builder'



You will now see the new controller class be unfolded into the 'Controllers' folder in 'Solution Explorer'

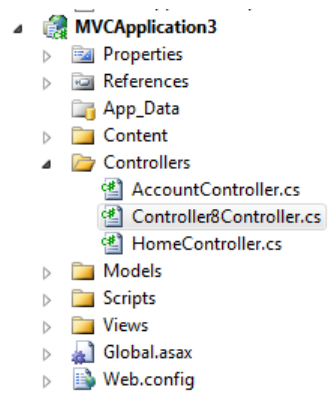


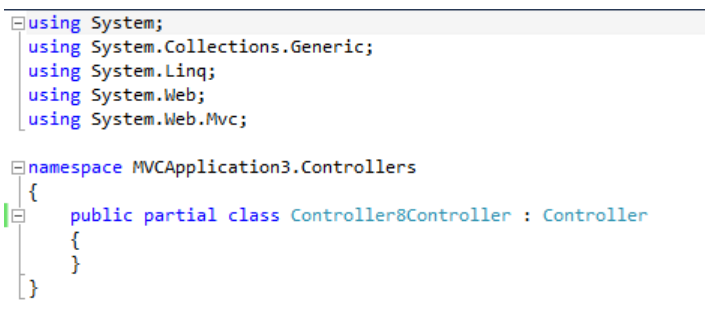
> **Note**: The name of the code file (Controller1Controller.cs) takes the form of the element name. The class name within the content of the file is the same as the file name. The namespace is correctly assigned for the project.

Now change the name of the 'Controller1' element in 'Solution Builder' to be "Controller8".

> **Note**: To rename an element in 'Solution Builder', either select the element and hit **F2**, or right click and select the 'Rename' menu, or hit **F4** and change the name in the Properties Window.

You will notice that the class file name in 'Solution Explorer' updated, and the class name is updated.





Close and Save the Visual Studio Experimental Instance.

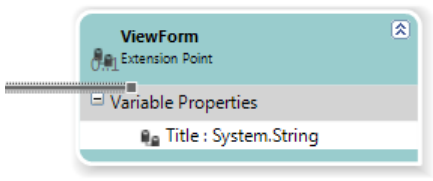
Feedback

All feedback, bugs, suggestions, questions etc. for ‘NuPattern’ are very welcome at the NuPattern project site: http://nupattern.codeplex.com.

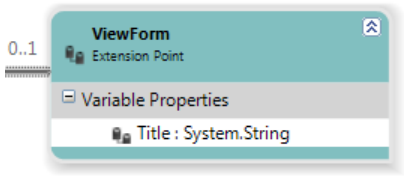
Notes

Tips on building and debugging pattern toolkits:

- Do not press F5 or "Start Debugging" to debug your toolkit because Visual Studio loads much more slowly with a debugger attached. When you need to debug some code that is running in Visual Studio, you can attach the debugger by clicking on the 'Debug | Attach to Process' menu (Ctrl + Alt + P), then selecting the Visual Studio (devenv) process you want to debug from the list.
- If you need to get access to the folders where Visual Studio extensions are installed, you can find them in these locations:
    - Extensions installed in the main instance of Visual Studio: %localappdata%\Microsoft\VisualStudio\10.0\Extensions\
    - Extensions installed in the experimental instance of Visual Studio: %localappdata%\Microsoft\VisualStudio\10.0Exp\Extensions\
- You can delete extensions just by deleting their folders in the Extensions directory. You can install them by copying their folders there as well. For example, you can copy an extension from the experimental instance of Visual Studio to the main instance by copying the folder in the Extensions directory. However, you may have to enable the extension in the Extension Manager in Visual Studio if you use this method.
- You must close the Experimental Instance every time you build a toolkit or else you will get build errors because the toolkit binaries are locked by the Experiment Instance process.
- If you change the name of a VSIX package you will have to manually delete it from the Visual Studio Experimental Instance. You will find it here: %localappdata%\Microsoft\VisualStudio\10.0Exp\Extensions\
- If you are using Windows XP you will not have the %localappdata% environment variable defined. You will have to add it manually.
- If you have problems in the Experimental Instance that you can't explain, you can try deleting the experimental instance data folder at %localappdata%\Microsoft\VisualStudio\10.0Exp, then 'Reset the Visual Studio 2010 Experimental Instance' form the Windows Start menu.
- The default layout of connector lines in Pattern Models is often not what you would want. You can move the connection point by hovering the mouse over the exact spot in which the connector meets the shape. You will see the mouse icon change to a larger cross shape. When the shape has changed you can click and drag the connector to a different spot on the shape. You can right click anywhere on the diagram and select 'Auto Arrange Shapes' to layout your shapes hierarchically.
- Sometimes the connectors on the shapes will be drawn inside the shape, like in the picture below. That can happen if you resize the shape be dragging the right border to the right.



To correct it, hover the mouse pointer over the end of the connector until you see the pointer turn into four black arrows. (If you see four white arrows you need to move the pointer until it turns into black arrows.) Then click and drag the pointer up or down to reposition the connector. When you let go of the mouse button the connector should be in the correct position.



- If you find that a command is not executing or an item template is not unfolding when you expect it to, check to make sure you have a launch point configured to run the command.
- If you want to find out what goes wrong (or right) with your toolkit when being used for troubleshooting, use the 'Output Window' (CTRL+W,O) and select the 'NuPattern Toolkit Extensions' category from the dropdown list at top of the pane. Also, see the options for the detail of tracing in 'Tools | Options | NuPattern Toolkit Extensions'
- Use the guidance provided in the 'Guidance Explorer' window for 'Creating Pattern Toolkits', which has far more detail on how to build and run pattern toolkits, this hands-on-lab has only scratched the surface of what you can do with a pattern toolkit.