Pattern Toolkit Guidance

Contents

First Time Users

These topics are aimed at the first time users, and first time authors, of Pattern Toolkits.

Understanding

This section helps you understand what Pattern Toolkits are, why they are useful, who should use them, and how to create them.

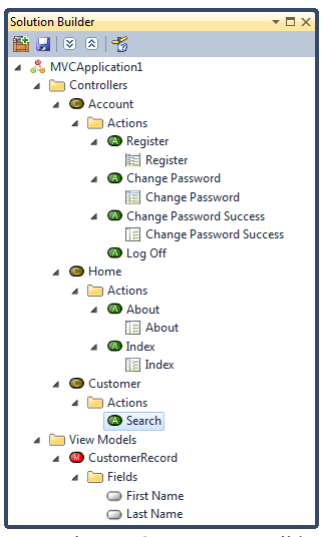
What is NuPattern?

NuPattern is a set of Visual Studio extensions that enable you to build and use custom 'Pattern Toolkits'. A ‘Pattern Toolkit’ is a set of custom tools with integrated guidance that guide and automate the creation and integration of best practices for building custom IT solutions, giving users the ability to build custom solutions significantly more rapidly, more reliably, more consistently and more predictably. This leads to higher quality and more maintainable solutions.

The NuPattern project site contains more up to date information and can be found at: http://nupattern.codeplex.com

A 'Pattern Toolkit' is a Visual Studio extension that is built by solution implementation experts who define how their solutions should be built correctly and consistently with established architectural or technology patterns, that an organization or community have pre-determined to be most effective, superior and efficient from experiences and knowledge of having built and refined those solutions in the past. The primary motivation for the experts to build a ‘Pattern Toolkit’ is to create a set of integrated tools that can be used by users to create repeatable solutions for other projects with similar customer requirements.

Instead of waiting for platform and technology vendors to release starter kits and templates for their technologies to meet general development or deployment scenarios; with NuPattern, any organization can now create their own custom tools, and templates that provide others in their organization specialized tools and guidance for building solution ‘their’ specific way.

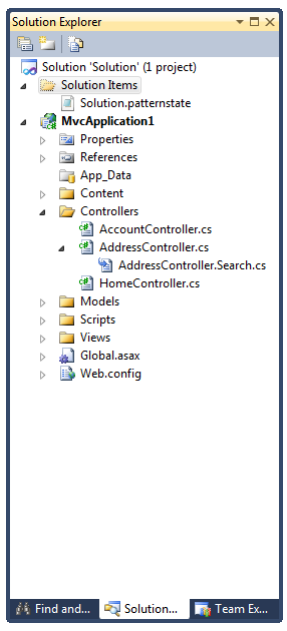


Example MVC Pattern Toolkit

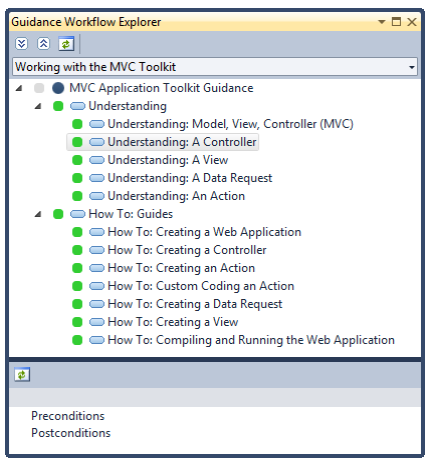
When a ‘Pattern Toolkit’ is installed, a ‘user’ of the toolkit works in a new tool window called 'Solution Builder', which allows them to create, navigate and configure parts of their solution in terms of the components of the working solution instead of in terms of the low-level technological artifacts of the solution as seen for example in ‘Solution Explorer’. Users work at much higher levels of abstraction than that of source code or platform API’s, which guide them to make simpler decisions that are more closely aligned with customer requirements and constraints. The ‘Pattern Toolkit’ has the institutionalized knowledge and intelligence to transform those simple choices into the most optimal implementation of a working solution, based upon these choices, using proven implementation patterns encoded into the toolkit.

Example A: a web developer might use a Model-View-Controller (MVC) pattern toolkit to apply the MVC pattern to the front end of their web application. The model they would use in 'Solution Builder' contains various 'Solution Elements' that describe various elements of the MVC pattern such as the Controller, Actions and Forms that they must name and configure. Each of these elements of the pattern will usually result in the generation of one or more solution artifacts (i.e. source files, configuration files, projects, etc.) that implement, the pattern, and integrate it into the current solution, as displayed in 'Solution Explorer'.

Example B: an infrastructure architect deploying a Lync solution in an enterprise, might use a Lync Deployment pattern toolkit to help them analyze and configure a server environment for many hundreds of users in the organization following a set of best practices based upon existing infrastructure systems and technologies already existing in the data center.



The overwhelming benefit for an organization to build and use Pattern Toolkits is that they can begin to scale their knowledge and expertise across multiple projects simultaneously using a consistent set of guidelines and best practices. And because these are applied reliably and correctly with automation, they dramatically improve productivity, predictability and supportability of their deployed solutions.

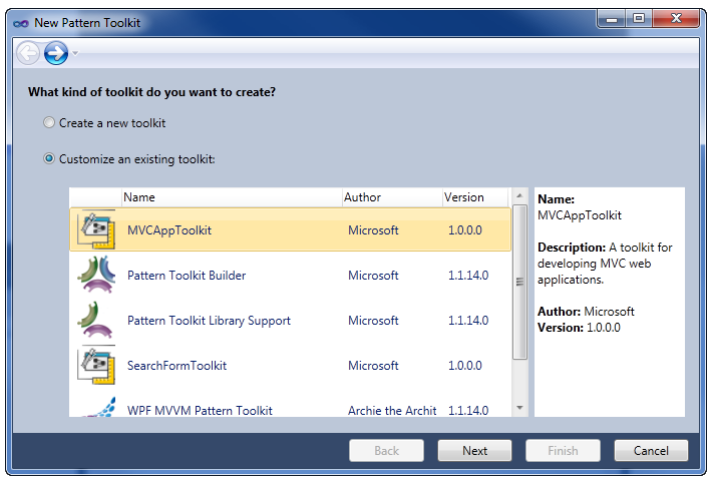


The essential ingredients of a Pattern Toolkit are:

- A Pattern Model, which describes the solution to be built, in terms of the structure of the solution and its variable parts.
- Reusable Assets (i.e. templates, scripts, code generators etc.), each related to elements and properties in the pattern model, which are transformed into a solution implementation.
- Automation, which drives the validation, generation and integration of a solution implementation into the current solution.
- Guidance, which guides the user through understanding the solution, its variance, and how to use the toolkit to build their solution.

Pattern Toolkits can be designed to work together, and to extend each other, so that larger parts of a solution can be composed together by toolkits collaborating and integrating together. Pattern Toolkits can define ‘Extension Points’ that can be implemented by other Pattern Toolkits, to offer extensibility and more extensive choice for users.

For example, an MVC Web Application Toolkit could offer an extension point for its ‘Views’ component, so that other Pattern Toolkits can be defined now, or even later that implement specific types of Views (i.e. Logon Views, Master Details Views, Search Views etc.) as they emerge during a project.



Finally, Pattern Toolkits are natively customizable, and extensible. For example, an organization can install an existing Pattern Toolkit obtained from a community and tailor any part of its: pattern model, reusable assets, automation or guidance, for the specific needs, constraints or requirements of the organization. Perhaps refining, or adding their own and best practices, and producing a new variant of that toolkit that it gives its own organization a competitive distinction for building solutions for its’ specific customers.

Why Would I Want to Use a Pattern Toolkit?

"Answer: I want specialized tools for automating the predictable parts of my custom solutions, so that I don't have to know how to hand-craft each part of my solution from scratch"

A Pattern Toolkit is the result of codifying one or more development or deployment patterns for publishing to other users to maximize reuse and consistency. Benefits such as significantly higher productivity, high consistency and lower development costs are realized only when solution developers reuse pattern-based tooling, that is: easy to discover, easy to install, easy to learn and use, easy to adapt, that provides simplified configuration, and that implements proven and maintainable technologies. A fact today with software reuse is that unless you can easily discover reusable solution assets, be able to easily install them and learn and trust them, and be able to wield them quickly and effectively - you are very likely to discard them; favoring the more trusted path of rewriting what you already do know or have at your disposal. Suffice to say that discovery, learning and innovation are key drivers for solution engineers. Given half a chance, an engineer will always favor re-discovery, re-engineering and re-learning over re-use. The economics of solution procuring, development and delivery on the other hand favor quite the opposite activities. In that world, where budgets and timelines are not infinite, re-use is always favored. The challenge of any reuse technology today is how to make it *easy to reuse* or adapt, versus re-write.

Pattern toolkits have specific been designed to address the above issues, and maximize economic reuse with these features:

- A pattern toolkit is generated, compiled into and deployed, as a single packaged distributable that installs into Visual Studio (A VSIX). As of Visual Studio 2010, Visual Studio now comes with an 'Extension Manager' which is the single place to download and manage any extension to Visual Studio. And this ease of discovery and installation has put extensions right in forefront of the development environment. No longer do users have to go searching online for extensions.
- A pattern toolkit contains all reusable assets, automation, knowledge and expertise to produce one of many variants of a solution consistently and predictably based upon a set of decisions that focus on custom requirements and constraints the user may face. Users, no longer need to interpret those requirements and determine and distill suitable best practices for implementing them. The toolkit does that for them automatically, dramatically increasing implementation time, quality and consistency.
- A pattern toolkit contains integrated guidance that is available within the Visual Studio environment, so that users can learn about how the toolkit works and how to use it for building components in their solution. This integrated guidance typically contains information about the toolkit, the pattern it implements, and tutorials on how to work with its elements and automation. No longer do users have to scour the internet for blogs, reference documentation, tutorials, webcasts. That information is presented to with the development environment, always contextualized with what they are working on.
- Pattern Toolkits are natively customizable. That is to say, that when an existing pattern toolkit has most of what is needed, or is deemed a good baseline for what is needed; it is more cost effective to take that pattern toolkit and customize it, tailoring it to with specific or new requirements, rather build a new toolkit from scratch. The process of creating a new toolkit and customizing an existing toolkit are the same.

Benefits

The benefits for a user using a pattern toolkit are many:

- A user has all the necessary tooling and guidance to implement and integrate a specific pattern into their solution, all within the Visual Studio environment.
- A user is educated and guided in configuring their solution, rather than having to focus on understanding deeply how specific implementations should be realized in their solution.
- A user never has to browse, filter, understand, and copy/paste sample implementations, risking human error in integrating those samples into their solutions.
- A user can very quickly become competent at applying a new technology or pattern with only a conceptual understanding of it, leaving the fine implementation details to the toolkit.
- A user can change their configuration and requirements as the project evolves, and never have to worry how and where their solution needs to change to adapt to that.
- A user can compose related patterns together, tackling larger pieces of their solution.
- A user can obtain and install pattern toolkits very easily from any organizational repository, or online gallery.

Why Would I Build a Pattern Toolkit?

"Answer: I want my team/organization/community to apply proven implementation patterns in their solutions quickly, cheaply and consistently."

With the advent of any new technology/platform, or when faced with a new requirement for the development of an IT solution, there always follows a period of analysis, discovery and experimentation to find a way to craft a set of given assets into a robust, reliable and maintainable solution. For some who have the opportunity and motivation to invest the time to do this, it can be a very rewarding voyage of discovery as the requirements, constraints, patterns and practices are learned. For those who can't afford this initial investment or risk, this exercise can be very frustrating and expensive, and often directly competes or interferes with the delivery of the larger solution that needs to be delivered.

The practices and skills required to understand and apply new technologies to custom solutions that meet a set of known requirements takes experience and time. In some cases, efficient and robust solutions that meet a set of pre-defined or engineered requirements evolve into proven implementations over the course of several refinements. Patterns and standards emerge. This is always the desired outcome for supportability and repeatability.

Oftentimes solutions that are proven to be successful generate their own need to repeat them again and perhaps for a different set of similar requirements, and constraints. The challenge has always been how to harvest the essence of the proven solution for re-application where the requirements are similar (but may not be exactly the same)?

Historically, for various factors and constraints, the least path of resistance has been to 'clone and own' successful existing proven implementations and then invest in working to understand them well enough (but rarely ever fully enough) to adapt them to a new set of slightly different requirements and constraints. This approach has enjoyed limited success over the years, primarily because the individuals who understood the process of engineering the original proven system (for its unique set of requirement and constraints), are more often than not, *not the same individuals* who have to reapply them again to a different set of requirements and constraints. As a result, the learnings and experience is not transferred, much of it is lost, and a great deal of rework is done at great cost that may or may not yield as high-quality a solution as the original work. Consistency and repeatability are not practically transferable by these common methods in practice. However, in the cases where the individual(s) are the same, a process of generalization will naturally occur, and as a result repeatable patterns will emerge as the solution is reapplied numerous times to differing implementations, requirements and constraints.

But something more is needed in order to be able to repeat the solution, with any level of consistency multiple times, that also has the flexibility to adapt to differing requirements and constraints that are inevitable in any custom implementation. That missing something needs to be efficient at laying down the parts that are *invariant* between solution implementations, and needs to be efficient at capturing choices that are *variant* between solution implementations. It needs to transform those choices into a suitable implementation that integrates with the invariant implementation that results in a total working solution.

Motivations

Who would be motivated to define a pattern toolkit for repeatability?

Today, in the IT industry there are many individuals and organizations that specialize in the design, architecture and application of each and every development tool, technology or platform, and many desire to share their knowledge or specific technology solutions between their peers, communities or partners. That need is evident by a vast abundance of technical blogs, books and other publications that typically include downloadable code, script or descriptive samples.

For many users consuming that 'knowledge base', reuse is easily achieved (if perhaps irresponsibly) by briefly reading the context-less description of a highly specific technical solution, followed by a copy and paste activity, and then integration of that sample into their specific solution. Generally the goal of the contributor that provides the samples, is to provide information that the reader can use to resolve similar issues they may have.

The challenges for the consuming user have always included:

- Is this solving the same (or a similar enough) problem to what I face right now?
    - The answer often lays buried deep in the textual description accompanying the sample (if any).
- Do I trust the originating contributor enough to take their word for it?
    - The answer is typically Yes! If not, then you find another one.
- Do I have time to understand this technology or sample sufficiently?
    - The answer is typically No!
- Does the sample assume the same technology constraints as I have? (i.e. platform, programming language, frameworks etc.)
    - The answer may lie in the descriptive text, and sometimes can be derived from the sample itself. Assuming the user is familiar enough with the technology.
- Does the sample address all the requirements I have, and does it honor the assumptions being made in my solution?
    - The answer may lie in the descriptive text, and sometimes can be derived from the sample itself. Assuming the user is familiar enough with the technology.
- Will this sample integrate well enough into my specific solution?
    - The answer there cannot often be pre-determined, and must be discovered by the act of copy pasting, integrating and testing it to try and find out.
    - This typically takes a lot of time, and typically generates further issues down the track.
- Will this sample work or break? or have other side-effects in my solution when applied to all my scenarios?
- Is this sample repeated anywhere else in my solution?
    - If so, I should reuse that instead.
- Are there alternative or a better ways, better best practices for implementing this in my solution?

For the contributors that are providing their solution samples, they face a whole set of other problems:

- Is my solution/sample correctly searchable for what it is?
- Am I credible enough for people to trust my solution/sample?
- Is my sample in a form that is readable and understandable enough to help someone solve the same problem?
- Is my sample generic enough to resolve problems similar to mine?
- Will people abuse my sample's reuse, and apply it blindly to solve problems that it's not designed to solve?
- Have I stated my assumptions and constraints about it correct reuse clearly enough so people don't apply it incorrectly?
- How do I communicate all the possible variables or contexts that need to be changed if used in different scenarios?
- How do I state alternative implementations given different requirements?
- How do I ensure it integrates into everyone's solution, and that they have all the pre-requisites at hand?

The challenge the contributor has is balancing the time and investment they put into documenting the context of the sample, with providing the content of the sample. Very often, very little context is given, because text and pictures alone in HTML format is a very poor means to convey variability and transformation from concepts to variable implementation detail. That is to say, it is very hard and tedious to write down all the permutations of all the solution implementations for any given set of variable requirements, constraints or contexts. Typically, the author will only document a single context, and document their assumptions and constraints for that single context. The reader needs to beware using the sample in any other contexts. Unfortunately, the contributor probably has a lot more knowledge and experience to share about all the other possible variants of the solution.

Instead of investing in writing and publishing sample solutions in the hope that everyone will reuse your expertise and experience in solving the same kind of problems correctly, proficiently, consistently, correctly and responsibly. It would be more efficient to create a tool that implements the transformations that handle the permutation of variation automatically instead. The documentation instead becomes the informational guidance that users can read and follow in-context with the tools they are applying it with. And this fosters accelerated learning of the technology or solution from having applied it correctly to their specific situation.

A pattern toolkit doesn't have to become out of date as its implementation technologies change, or as new requirements are identified. As those changes occur, the toolkit can be updated, forked or customized. You can even define multiple toolkits that have the different implementations, or simply make the choice of implementation a parameter of the pattern model, and allow user to make it a choice.

As you start to refine and standardize your pattern implementations, you may desire to decompose them into smaller toolkits, and have them integrate with each other, rather than encapsulating all functionality in one monolithic toolkit that will need regularly updating. In this way, you would define well-known extension points that other toolkits can plug into and offer different implementations or provide to different requirements. And users can compose the toolkits together to start to tackle larger components of their solutions. You can even let other toolkit builders integrate their toolkits with yours at these extension points.

Can I Customize a Pattern Toolkit?

"Answer: Yes, all pattern toolkits natively support customization and tailoring once they are installed."

A critical success factor in achieving reuse with pattern toolkits is being able to reuse the toolkit itself, and its contents, and applying them to similar problems. This may sound like a nonsensical statement at first, but the point here is that if you (or someone else has) has already invested time (money and resources) into parameterizing and patterning a solution, the chances are that someone else is likely to find that solution very useful, but perhaps in a slightly different scenario or context which may or may not have been originally considered. Or, more likely, they will have slightly different requirements for it to be 're-useful' in their specific scenario.

Past Difficulties with Reuse

It has been demonstrated over and again in the past with other attempts at tooling assets for reuse, that if the effort or cost in understanding and modifying an existing asset for re-application to a slightly but different scenario, outweighs the effort in creating the new asset for the specific scenario, then reuse does not occur. A number of factors influence this decision to **'rebuild versus reuse'**. But, in order for reuse to even be a goal, it needs to be understood that there has to be an objective for reuse, or reuse has to be a stated explicit goal. Typically, and necessarily, in IT these objectives and goals are driven 'by the business'. Sadly, if left to 'engineering' alone, the act of reuse often conflicts with the more enjoyable urges to craft a new design and evolve better engineered solutions from what they already know (quite understandable for creative architects and engineers).

Given that reuse is one of the stated goals of many IT projects, the typical decision process (made by engineers) for determining whether to 'reuse versus rebuild' goes something like this:

- Can it be reused? Does it, Would it, or Could it be manipulated to satisfy our specific requirements? ✓
- Is it easy enough to understand, reverse engineer and to modify it to be reused in our scenario successfully? ✓
- Can we trust its quality? ✓
- Do we have the skills and know-how at hand (or readily available) to modify it? ✓
- Is it cost effective to reuse it? Are we going to recognize benefit and ROI from reusing it? ✓

Given a negative answer (✘) or any uncertainly to any answer above (?) tends to yield a 'rebuild' decision rather than a 'reuse' decision across the board.

The business may not see it that way and may contest that evaluation, but it is universally recognized that it is an engineering-based decision as to how hard and expensive something is to understand, re-engineer, and deliver working correctly to a high enough quality bar. Therefore, if the sum effort of manipulating a reusable asset is perceived to be easy enough to do by engineering – then reuse is a 'candidate' possible outcome. It's not really until you couple that determination with a motivation through an intentional objective or goal for reuse, that reuse will actually occur, in general.

Given these broad challenges in the IT industry, NuPattern was designed specifically to make the process and activities of customizing existing assets like toolkits far easier and more economical to do than creating new ones.

In reality, there is no perceptible difference in the actual tooling and process for creating a new toolkit versus customizing an existing one. Customizing the assets contained within the original toolkit still requires manual effort, but given that the pattern is already implemented with the existing assets in some already templatized, modifying those assets from their current state to satisfy slightly different requirements should be far easier than rewriting a new template from scratch. Changing or extending an existing pattern is as easier and more cost effective than creating a new one.

Now, with pattern toolkit development there is an early decision point that needs to be made, where you need to evaluate whether to reuse an existing toolkit, or create a new toolkit from scratch. In either case, the economies of doing either (new or customized) should far outweigh the cost not reusing the pattern at all.

How do Pattern Toolkits Work?

Pattern Toolkits are specialized Visual Studio extensions (VSIXes) that provide all the tooling and guidance for configuring and applying an implementation of a single development or design pattern. Once installed, like any other Visual Studio extension, they can be managed using the 'Extension Manager' in Visual Studio. You can even browse, download and install them directly from the 'Extension Manager' in Visual Studio. Pattern toolkits can also be found and downloaded from the Visual Studio Gallery.

They Contain a Single Pattern

A Pattern toolkit contains a single pattern that can at any time be 'instantiated' and added to a solution. These instances are viewed and managed in a new window in Visual Studio called the 'Solution Builder' window. Once instantiated, a 'pattern instance', known as a 'solution element', provides an abstraction/model/schema of a pattern that is parameterized to allow a user to configure the instance to apply to their specific use in their solution. The more parameters a pattern requires the more configuration a user has to work with. For many non-trivial patterns that configuration may require the user to create related hierarchical child elements and configure their properties. Many of these elements have default values or default choices for their properties. It is the abstract representation of the pattern instance in the 'Solution Builder' window that visualizes the parameters of the pattern for the user, and provides the physical 'elements' and context in the development environment for a user to interact with.

They Have State

Instances of patterns (solution elements) are created and source-controlled in the solution. When the properties or hierarchies of the solution elements are modified the changes are automatically saved in the solution. The file that persists the state of all solution elements is automatically checked out for editing.

They Create an Implementation

A configured solution element has the inherent ability to automate and manage its implementation in the solution. This is typically achieved initially by 'unfolding' or 'generating' projects, scripts, source code, or configuration files into the solution, that *are* the physical implementation of the pattern. The pattern toolkits job is to ensure that the physical implementation of the pattern is representative of the current state of the solution elements and that they are correctly configured at all times. This is achieved by a set of validations applied by the toolkit at keys times. Validating the solution element's current state and notifying the user when the state is invalid and how to fix it. In some toolkits, generating solution artifacts may only occur when that valid state has been reached, or solution artifacts are removed until a valid or different state has been reached.

They Teach and Guide Solution Development

Any solution element may have guidance associated with it that provides information on what the element's semantics are, and how the user can manipulate it and its configuration to address the specific requirements of the solution. The guidance is usually launched from the solution element, and is always contextualized to that element. Guidance may have state also. That is, if the guidance contains prescribed steps, those steps may have state that controls the flow of the steps. A workflow is initiated that the user can follow to complete a given process with the solution element.

They Apply Automation

A solution element uses automation extensively. Automation is used for example to validate the solution elements. Automation is used to manage the linkage and traversal between the solution element and the physical artifacts in the solution that it may represents or relate to. Automation is used both in generating those artifacts and managing their state, naming and content. Automation is used to present wizards to the users at certain times to help correctly configure properties of the solution element so that other automation can correctly perform additional actions in the Visual Studio environment. Automation is used to provide menu items on solution elements, to execute other automation that interacts with the development environment or other solution elements. Automation is used to evaluate when menus, guidance, generation, validation and other capabilities are permitted. Suffice to say, that all automation is controlled by conditional automation. It's the automation of a pattern that provides the most value in accelerating the development of a pattern's implementation. For the most part, a user using a pattern may be unaware that it is the automation that helps them get their work done with this pattern much faster.

They Compose Together

Solution Elements support dynamic extensibility. That is, if any given toolkit permits, other toolkits can plug-in their solution elements into the solution elements of another toolkits. For example, let's say we have a pattern toolkit that requires certain configuration information that has been standardized by the toolkit creator. That toolkit will 'advertise' that standardized configuration in the form of a special kind of solution element - one with that specific configuration. Now, other toolkits can provide specific variants of that kind of special solution element, which may have a specific meaning to a user or a specific architecture or solution. Now, if there are one or more toolkits providing those different kinds of special solution elements, then the user can choose which one of them to plug into the original pattern. And this is one way that toolkit can collaborate to extend each other. It is this extensibility mechanism that makes pattern toolkits extensible and enables users to compose them dynamically.

Working with Pattern Toolkits

Find out how to use and create Pattern Toolkits.

Using Patterns/Pattern Toolkits

The tools and user interface for using pattern toolkits enables you to: browse the installed patterns, view and create instances of those patterns called ‘Solution Elements’, and configure the solution elements to create projects in your solution.

Pre-requisites for Using

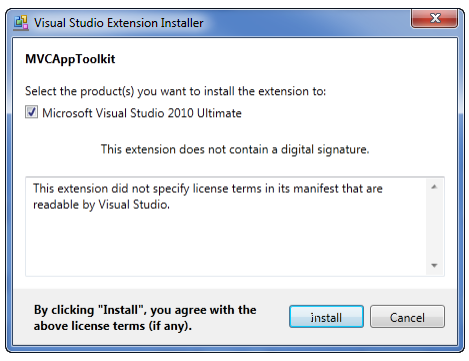
All you need to install and use a pattern toolkit is Visual Studio 2010/2012 Professional, Premium or Ultimate.

All pattern toolkits, when installed, deploy all their own pre-requisites and dependencies.

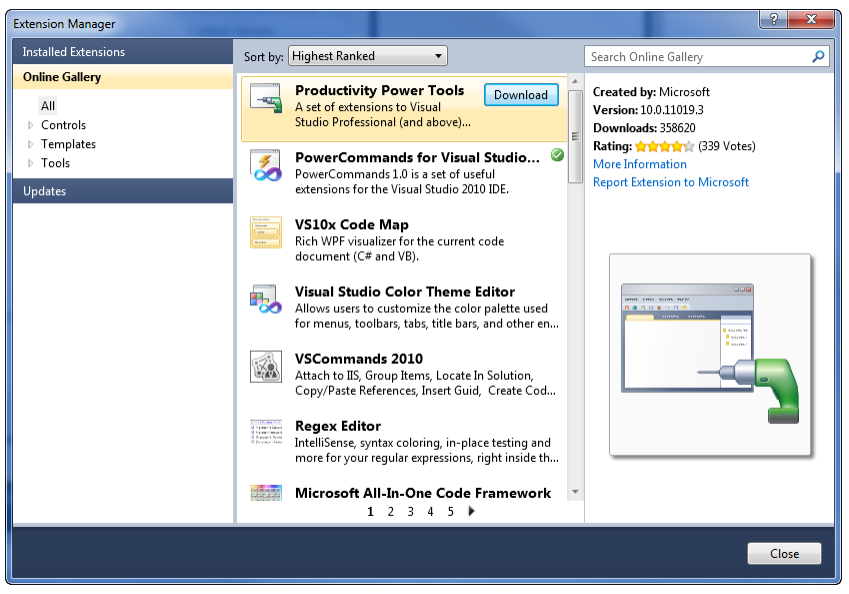
Installing a Pattern Toolkit

To install a pattern toolkit you have obtained as a file with a *.vsix file extension for the toolkit author, is simply to open that file and click though the installer.

> **Note**: Like any other Visual Studio extension, you can search the Visual Studio Gallery for pattern toolkits.

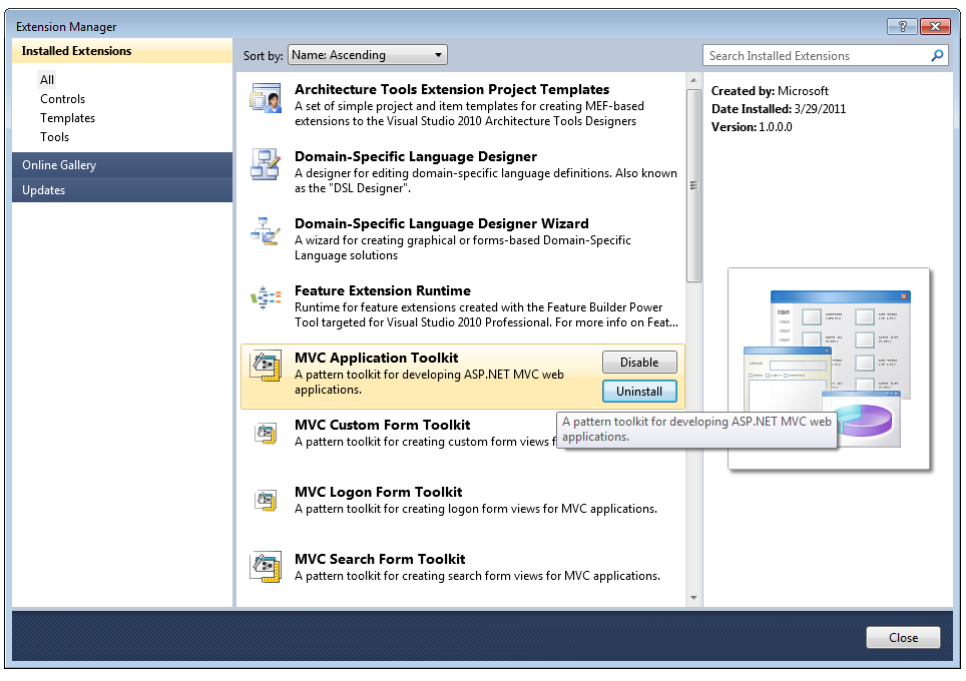


To install a pattern toolkit obtainable from the 'Visual Studio Gallery', you either download the *.vsix file to your machine and open it (as described above), or you use 'Extension Manager' in Visual Studio to 'Download' and 'Install' it in Visual Studio.



Browsing & Managing Installed Pattern Toolkits

You can browse the installed patterns toolkits, and uninstall or disable them using the Visual Studio 'Extension Manager'.



Note: After 'Disable' or 'Uninstall' buttons are clicked, a restart of Visual Studio is required to remove the extension from Visual Studio.

Using Pattern Toolkits

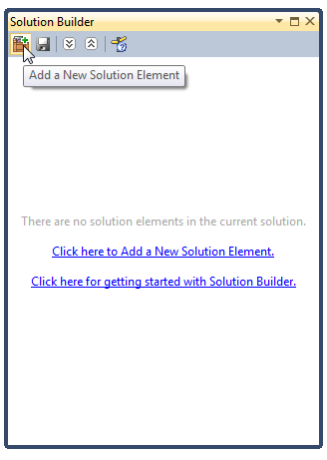
Once a pattern toolkit is installed, the contained pattern within it is available to be used.

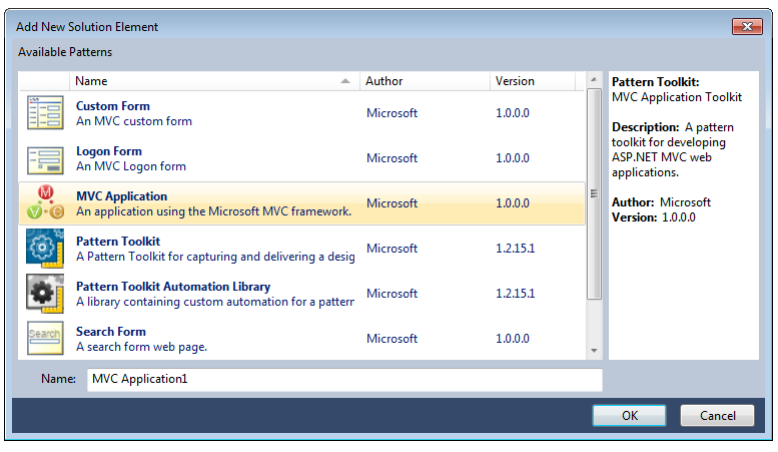
To the user using a pattern toolkit, they are really more focused on applying useful patterns to their solution, and therefore once installed; they really just want to use graphical elements that help them build their solution. For this reason, the terminology used from a user's perspective switches to 'building solutions' with 'Solution Elements', rather than creating instances of patterns.

Creating new 'Solution Elements' is performed either directly from within the 'Solution Builder' tool window, or automatically, from the more traditional method of creating new projects or items in Visual Studio using the 'Add New Project/Item' dialog, (As long as the particular Pattern Toolkit provides a project or item template for the pattern).

To create a new solution element in the **'Solution Builder'** tool window, click the 'Add New Solution Element' button.

This displays the 'Add New Solution Element' dialog, where you can choose to create from one of the patterns provided by the currently installed Pattern Toolkits.



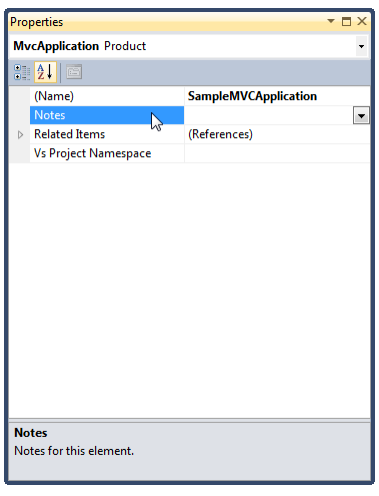


Viewing & Configuring Solution Elements

You configure and compose the 'Solution Elements' in your solution in the 'Solution Builder' window.

Depending on how the specific pattern toolkit is implemented for a specific solution element, various standard features may be available to you.

Editing General Properties of a Solution Element



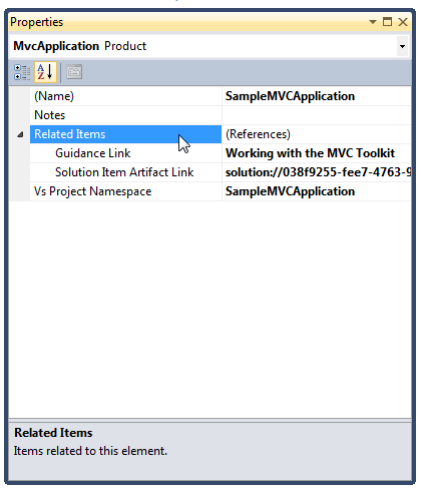
Selecting a solution element allows you to edit its properties in the 'Properties Window':

Editing Properties:

- Each property may have its own editor for its value, a dropdown of values or allow you to type a value.
- Some properties are read only.

All properties have a 'Description' pane below explaining their meaning.

Related Item Properties



All solution elements have a 'Related Items' property, that when populated, relate the solution element other items (e.g. solution artifacts, guidance workflows, model elements, source code elements etc.) within the Visual Studio environment or externally to Visual Studio. See Related Items for details.

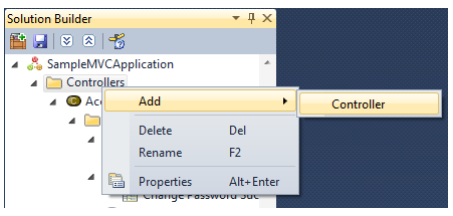
> **Note**: Depending on the type of related item, they can usually be modified (or fixed) to relate to other existing items, should the related item ever become un-related for whatever reason (i.e. deleted).

Notes Properties

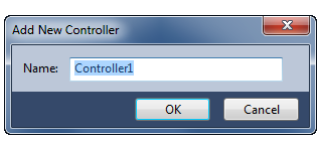
Every solution element has a 'Notes' property that allows you to leave design notes or annotations describing the current solution element.

Adding Child Solution Elements

If the solution element has child elements defined in the pattern, and allows more than one instance of that child element, then you can add more instances by right-clicking on the element and selecting the type of child to add from the 'Add' menu.

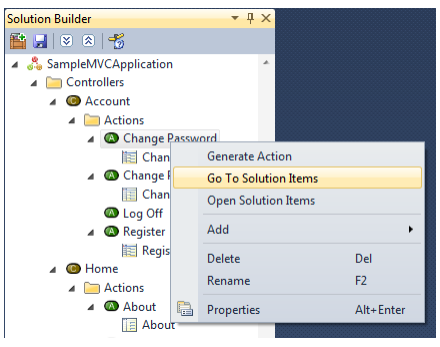


Once the menu is selected, the child element is named in the 'Add New' dialog:



Navigating to Related Solution Artifacts

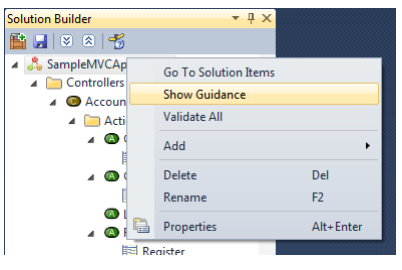
If the solution element has a 'Related Item' (one or more) that refer to solution artifact which are navigable in 'Solution Explorer' (called a 'Solution Item Artifact Link'), then you can navigate to them using the 'Go to Solution Item' menu.



If the solution element has a 'Related Item' that refers to a solution artifact which be edited in the IDE, then you can open it with the 'Open Solution Item' menu, or you can double-click on them in Solution Builder.

Viewing Related Guidance

If the solution element has a 'Related Item' that refers to guidance (called a 'Guidance Link'), then you can view the guidance by selecting the 'Open Guidance' menu.

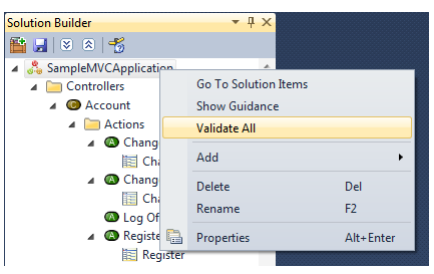


Validating Solution Elements

If the solution element has properties that are required by the pattern, or the values that must have certain values, then validation rules will notify you if these properties and solution elements are invalid, when validated.

When Validation occurs is controlled by the pattern toolkit. Typically, it may occur on any one, or all, of these events occur:

- When any property of any solution element is modified.
- When the solution is built.
- From the 'Validate All' menu on a solution element.
- Other custom events that the toolkit implements.



Running Automation

Solution Elements are the physical launch points, and define the scope, for automation that could be provided by pattern toolkit. Typically, automation is contextualized to a specific solution element, and/or to a set of its descendant solution elements. But some automation may also traverse up and down other solution elements in the pattern model.

The real power of the automation capability in pattern toolkits is that it can be used to do automate just about any task inside of (or even outside of) the Visual Studio environment. Basically, whatever the pattern toolkit needs to do to make development of the pattern quick, easy, and not tedious.

Automation can be triggered by any number of events (e.g. some provided ones include: On Build, On Save, On Creation, On Property Changed etc.) or under any number of conditions (e.g. a Solution Element's Properties are all Valid, or All Related Items in the Solution are Saved, etc.). These conditions are again all defined and controlled by the pattern toolkit.

Most automation uses the properties and the state of the solution elements as its input (or context). Some automation utilizes the state of the other artifacts and services outside the development environment.

It is very difficult to quantify or summarize all the possible types of automation that can be implemented by a pattern toolkit. The scope, depth and quality of the provided automation from a toolkit are entirely dependent on the pattern being modeled, and the other tools available to the pattern toolkit to automate.

There is a provided set of automation types and an automation framework that comes standard with a toolkit, making the application of automation very easy for a pattern toolkit author. See What is Automation for more details.

Some general examples of typical kinds of automation in a toolkit would include:

- Generating projects or source code files into 'Solution Explorer' (either from a text transformation or from VS project or item templates).
- Configuring the properties of a related project in the solution.
- Running validation rules when the solution element reaches a correctly configured state.
- Automatically generate reports for traceability.

Creating and Customizing Pattern Toolkits

Creating a pattern toolkit (Visual Studio project) is similar to the experience of using pattern toolkits. A Pattern Toolkit is after all a specific design pattern!

You will need to have some special tooling installed in order to create a new or customize an existing pattern toolkit. See Pre-requisites for Authoring.

Pre-Requisites for Authoring

> **Note**: If you are reading this guidance from the 'Guidance Browser' tool window in 'Visual Studio' right now, then you already have the 'NuPattern Toolkit Builder' extension installed!
> That's' all you need to create your own pattern toolkit, skip to 'Creating a Pattern Toolkit'.

Authoring of Pattern Toolkits requires Visual Studio 2010/2012 Professional, Premium or Ultimate., and requires installation of the 'Visual Studio SDK' (VSSDK), which can be downloaded from here.

> **Warning**: If you have already installed Visual Studio 2010 Service Pack 1, then you must install the Visual Studio 2010 SDK SP1.

Installing NuPattern

> **Note**: These steps are unnecessary for simply using an existing a custom Pattern Toolkit. Instead, see Installing a Pattern Toolkit for details.

To create a new Pattern Toolkit or customize an existing Pattern Toolkit, you must have first prepared and installed the Pre-requisites for Authoring.

Once the pre-requisites have been met, perform these steps:

1. Save your work, and close any and all running instances of 'Visual Studio'
2. Install the 'NuPattern Toolkit Builder VS2010' or 'NuPattern Toolkit Builder VS2012'extension from the Visual Studio Gallery.
3. Start "Visual Studio"

Developing and Debugging

You debug and test your pattern toolkit in a special instance of Visual Studio called the 'Experimental Instance of Visual Studio'. This is a separate version of Visual Studio that can be reset back to a known state, used for debugging and testing extensions. This instance of Visual Studio won't interfere with the version that you use to development your toolkit.

In order to test or debug your pattern toolkit, you need to build your toolkit project and hit **CTRL+F5** to run your toolkit in the Experimental Instance. There you can create new instances of your pattern and try it out, as the users of your toolkit would.

Your toolkit is automatically updated and installed, with all extensions necessary to run your toolkit, whenever you build your toolkit project.

> **Tip**: If you are having trouble using your toolkit in the Experimental Instance, please follow steps to 'Reset' the Experimental Instance.

Creating a Pattern Toolkit

Creating a deliverable pattern toolkit is a process that requires preparation and pre-work before using this tooling to deliver a pattern toolkit.

Pre-Requisites to Reuse

"You have to have something to reuse!"

In order to create an *effective* pattern toolkit that deploys a proven pattern implementation, you need to already have access to assets that represent that proven implementation. These assets typically come in the form of things like:

- Code snippets, Code files, Configuration files, Templates, UML models, and/or other existing artifacts for implementation of the pattern.
- Libraries (class libraries), Frameworks, and/or other redistributable dependencies required for the implementation of the pattern.
- Documentation, Articles, Publications, Design notes, and/or other forms for creating guidance for the pattern.

You **must** have access to these assets before and during the development of your pattern toolkit to be effective.

> **Note**: We do not recommend designing a pattern toolkit for an implementation that has not been already implemented and proven to some degree. This could result in a pattern toolkit that is not implementable by users who try to use it, or is unnecessarily complex to use!

Processing these obtained assets into a pattern toolkit is the activity of '**harvesting'** them from existing solutions/designs/etc. along with **'templatizing'**, are two key fundamental steps in creating a reusable 'asset' such as a 'Pattern Toolkit'.

The Methodology

Pattern Toolkit development is consistent with the methodology of developing **'Software Factories'**, and in this context, a Pattern Toolkit *is* a Software Factory.

A Pattern Toolkit is a specialized custom made and standardized development tool that implements a product line for a specific development domain, and combines: model driven development, architectural frameworks, patterns, frameworks guidance and tools.

The Software Factories Methodology (SFM) in a nutshell, when applied in-general for building any factory, describes the following theoretical development process:

1. **Analyze** existing (sample or production) solutions and identify reuse opportunities.
2. Analyze and **Identify** the artifacts and assets that make up these solutions for reusable: architectures, patterns and assets.
3. **Harvest** those patterns and assets for reuse from the existing solutions.
4. **Generalize** those patterns and identify commonality and sets of variance among them.
5. Construct **Reference Implementations**, libraries and frameworks that demonstrate the reuse of that pattern variance and uses those assets plus and others created from the commonality. Create test scenarios that prove the reuse.
6. Construct a schema or **Domain Model** of the pattern and identify how the variance will be represented in a model of the domain as configuration.
7. **Templatize** the reusable assets for applying variance from configuration in the model.
8. Create **Automation** that reads the domain model and executes an implementation of the software using the templatized assets and fixed common assets.
9. Create **Guidance** that provides instructions on how to use and configure that domain model.
10. Create a **Software Factory** tool that, when deployed allows a user to configure an instance of the domain model and execute automation to generate parts of the software.

Activities 1-5 involves all the pre-analysis work required, at least to some degree, to establish the need or justification for a pattern toolkit, and confirm the ability to deliver a implementable pattern with that toolkit. Without these critical steps, you will have few usable reusable assets to build upon.

Activities 6-10 are around constructing a schema or domain model (called a 'Pattern Model'), applying Automation to it, and creating the Guidance for it. All have very rich tooling experiences in the Pattern Toolkit creation process.

> **Note**: Variability Analysis + Modeling = Variability Modeling

As you will see next in the Development Cycle (Overview), the process for building a successful Pattern Toolkit and modeling the variability in a pattern model is in full compliance with the theoretical process above.

The Development Cycle (Overview)

Building a pattern toolkit follows a simple cycle in every iteration of the of the toolkit's design-development-test lifecycle.

Overall, there is no expectation that a toolkit can be fully designed up front and then implemented in one development iteration. Experience teaches us that a more successful approach is to start with a very simple pattern design that has very little or minimal variance in what it builds and then iteratively add more and more variance to it. As you add more variance to it, so then you add more automation, more assets and more guidance to it. In this way you can manage its complexity and adapt it to provide the best usability experience possible.

We recommend a tight, iterative development cycle that:

- As an author:
  - **Modify** the Toolkit
  - **Build & Deploy** the Toolkit
- As a User:
  - **Modify & Test the Pattern** (from the toolkit)
  - **Verify the Solution** (implementation)

See the Development Cycle (In Practice) for how to execute this process.

The Development Cycle (In Practice)

These are the steps in the cycle you should follow when developing a pattern toolkit.

Modify the Toolkit

Generally the cycle begins with a planned improvement in expanding the variability of the pattern. In some iterations you may also decide just to improve or add automation in the existing pattern, where it previously didn't exist.

In either case, keep the improvement increments small and manageable.

For any change you make to the automation, pattern or assets, you should make an associated change in the guidance (if any) to reflect that improvements. This ensures that the guidance is always up to date with the toolkit.

Synchronize the Guidance

It is critically important to keep the guidance up to date with the pattern. Treat this as a refactoring step.

> **Note**: Although not really appreciated well enough by the toolkit development team - until such time as they have to use their own toolkit for real work!, having guidance that is out of date is like having no guidance, but far more damaging. Wrong guidance, as opposed to no guidance, tells your users that your quality bar is low, and *their* perceived integrity of your toolkit diminishes quickly. Remember, they are using your toolkit to improve their productivity – the goal of all pattern toolkits. You don't want your users to think that your toolkit is no good at what it does.

Build & Deploy the Toolkit

Simply build the toolkit project/solution so that everything in it and everything that it depends upon is up to date.

Typically, you won't need to debug your pattern to deploy and test it. **CTRL+F5** ('Start Without Debugging') in Visual Studio compiles your toolkit, installs it into the 'Experimental Instance', and then starts the 'Visual Studio Experimental Instance' automatically for you with your toolkit ready to use.

Modify & Test the Pattern

Open the 'Solution Builder' window and either create a new instance of the pattern, or use an existing test solution from the last iteration.

Test out your incremental improvement, and note any usability issues.

Usability Testing

It is critically important for usability at this point that you transition your perspective from toolkit author to solution developer with the toolkit. If you are pairing, get your pair to evaluate your change and give you critical feedback.

> **Note**: You need to remain objective here, because your development tool (pattern toolkit) needs to be highly usable. If a user of it cannot figure out how to use it effectively, or does not understand how the pattern is represented or should be manipulated – they quickly lose interest in using your toolkit.

Don't overlook all the visual cues provided for the pattern structure and the properties of each solution element. These cues are the primary source of guidance that your pattern users have to stay on track before they go to the guidance for more information.

Automated Testing

Automated testing of a pattern toolkit is limited for this kind of verification. It is possible to create UI Tests that verify the UI in Solution Builder is working as you expect, but the UI is likely to evolve rapidly with each of these iterations. However, the more tests to verify working software you have the better. We recommend you invest time wisely in UI testing at this stage.

Automated testing of your automation is essential and strongly recommended for any custom built automation in your toolkit. These tests are unlikely to evolve rapidly as automation is well isolated, decoupled and encapsulated from the toolkit operation. We strongly recommend automated unit and integration testing for your toolkit automation.

Troubleshooting

If problems occur whilst testing your pattern or toolkit for whatever reason, make a note of any errors that occur if they are reported directly to you. If not directly reported the patterning toolkit keeps traces and logs all diagnostic information and issues encountered in the 'Output Window' of Visual Studio. Here you can see varying levels of trace information and all errors are also recorded when they occur. This trace log provides vital information for troubleshooting difficult problems. See more details in Troubleshooting Toolkits.

Verify the Solution

The final verification is to ensure that, if your incremental change effected the implementation of the pattern (i.e. the changes made to the users solution, environment or other systems by your toolkit), then you verify those changes also.

Again, automated integration testing of these kinds of end-to-end changes may have limited value for the effort invested.

↓ Iterate ↑

Now that your incremental improvement has been tested out, it's time to re-iterate from the top again.

Be sure that the improvements you plan to make next iteration are kept reasonably manageable. Pattern toolkits can evolve to become fairly complex very quickly, as it is very easy to add more new variability fairly quickly with the tooling.

> **Note**: Whenever things start looking too complex, you will find better success (and more confidence in your improvements) by breaking the improvements into smaller increments.

The Development Process

This section summarizes the key activities, and some implied sequencing in building a new pattern toolkit.

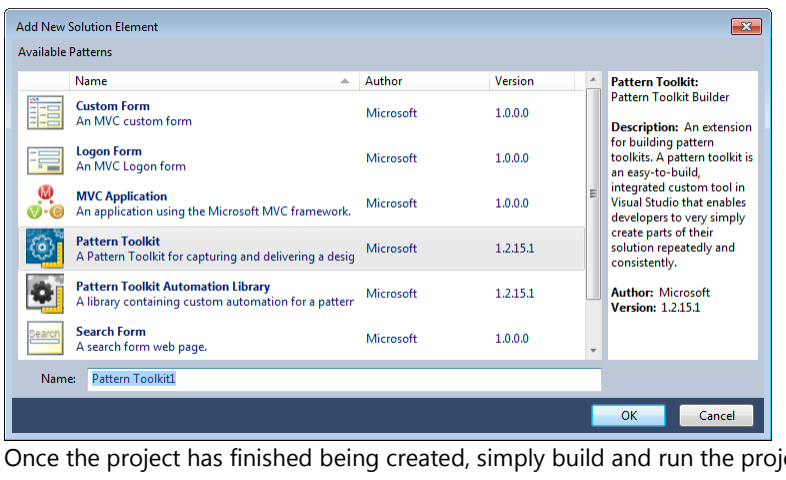
All or some of these steps may occur in any given iteration cycle. The last two steps for Deployment and Customization occur after development of the toolkit in releasing the toolkit, and modifying it or other toolkits later.

We recommend these activities be performed in roughly the given sequence, although there is no hard and fast rule about which activities you do first or next.

First-Steps: Creating New Pattern Toolkit Project

> **Note**: This activity needs only to be performed once for each pattern toolkit you build.

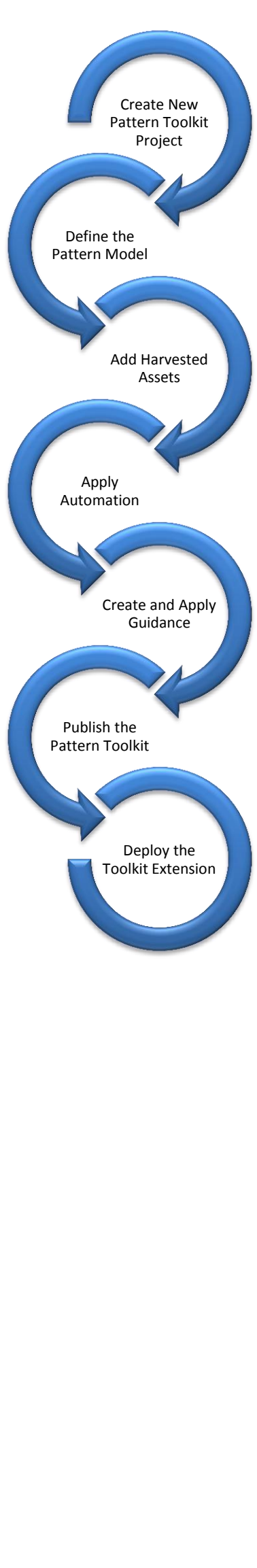
Create a new toolkit project by creating a new **'Pattern Toolkit'** from the Add New Solution Element button in the Solution Builder window.
Complete the project creation wizard and select the 'New Toolkit' option.



Once the project has finished being created, simply build and run the project..
In the Experimental Instance of Visual Studio (VSExp), create a blank solution, then open the Solution Builder window and create a new instance of your pattern that should now be in the list of available patterns.

> **Note**: You have now created a simple named and branded pattern toolkit, but it has no variance, no automation, no guidance and no realized implementation. This is the recommended starting point for developing any new toolkit.

From this point forward, given the new pattern toolkit project, you can now focus on the following development activities.



Define the Pattern Model

The pattern model is the single most important artifact for working with when building a pattern toolkit. The pattern model is the blueprint or schema of your pattern's variance. See more details in What is a Pattern Model.
You develop a pattern model in the 'Pattern Model Designer'. The general idea here is to represent the variability you have in this model. You decompose the pattern into different Variable Views (or aspects), each view is decomposed into Collections and Elements, each with multiplicity of either 1..1 (one-to-one) or 1...* (one-to-many). For each collection or element you define one or more Variable Properties.
Once you have established a definition of the pattern in this model, you can start to add assets that make up its implementation, guidance for using it and its behavior with automation.

Add Harvested Assets

Assets are represented in many forms: from code templates, through libraries and frameworks, configuration files, to documentation. See more details in Assets.
Assets are introduced into the toolkit once they have been harvested in some form, and in some cases 'templatized' to some degree ready for reuse in the toolkit.
The pattern toolkit project allows you to place these assets in the 'Assets' folder, which is further subdivided by Assets type.

Assets are applied to the pattern in most cases through configuring the elements in the model with automation.
For example: Code generation templates (*.tt or *.t4 files) and Visual Studio templates (*.vstemplate files) are configured individually with specific automation commands. Guidance is configured with properties for 'Associated Guidance' on each element. Other assets such as libraries and frameworks are deployed by the toolkit project using standard VSIX mechanisms.

Create and Apply Guidance

Guidance is typically represented in textual form in what are called 'Content Documents', one for each guidance topic.
These documents contain information about the various elements in your pattern and have text, images and links to other topics and web sites to help describe them and how to work with them.
These documents are orchestrated into 'Guidance Workflows' to help organize the information in a structure that can be browsed.
Individual guidance workflows are associated to specific elements in the pattern model. Or you can associate a guidance workflow to the whole pattern.
Guidance is technically just a special form of asset, and can be found also in the 'Assets' folder of the toolkit project.

Apply Automation

Automation is the key ingredient in providing significant productivity benefits from a Pattern Toolkit in use. Automation is the mechanism that is responsible for verifying and creating an implementation of your pattern in the solution, based upon the configured state of your pattern model at any time. See more details in Automation.
Automation is configured on each of the elements in the pattern model. Here you select the type of automation and configure the parameters of the automation. Some automation requires specific assets, such as the code generation and template commands. You configure these Commands to reference the assets in your toolkit, and then the assets are initialized and prepared by the toolkit to be automated in your pattern.
Automation is a very powerful mechanism in toolkits that controls many aspects of how your pattern is represented: how it behaves and interacts with users, whether the pattern is configured correctly, what the implementation will be for the pattern, and when and how the implementation is realized by the toolkit.

Publishing the Pattern Toolkit

Once a pattern model is ready for releasing to users, a process loosely called 'publishing'. See more details in Publishing.
In this process you set the basic information about the toolkit, such as: its name, description, version number, EULA.
Once you have completed the publishing information for the toolkit, it's time to deploy the toolkit to your users.

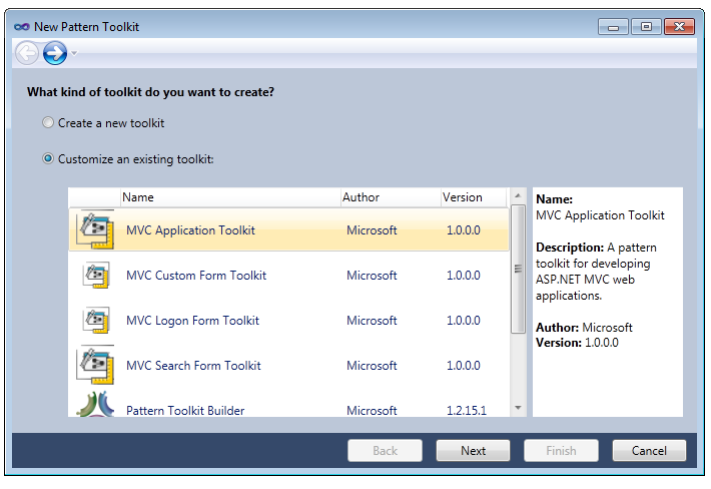
Deploy the Toolkit Extension

A Pattern Toolkit is deployed as a single VSIX file to a user to be installed into their Visual Studio environment. This file contains all the assets, the pattern model and any automation and guidance for the toolkit. See more details in Deploying a Toolkit.
As such, a pattern toolkit is just another Visual Studio Extension, and is deployed, versioned, installed and lifecycle managed like any other Visual Studio extension.
As a single VSIX file, a pattern toolkit can be delivered to users in a number of ways. E.g. downloaded from a site, emailed, etc. Installing them simply requires the user to run the file, and the VSIX installer application installs them directly into Visual Studio.

The Customization Process

Customizing an existing toolkit is as easy as developing a new one, and is strongly recommended when a toolkit provides the pattern you need, but requires extending or modifications to its assets and automation.

The only difference between creating a customized toolkit and creating a new toolkit is that when you go to create a toolkit project, you select the "Existing Toolkit' option and select the toolkit from the list to customize. That toolkit's pattern model is then copied to the new project as a starting point.



See What is Customization? For details on why customization is so easy and so much cheaper than building new from scratch.

Concepts

This section identifies and explains the main concepts for: using, creating and customizing ‘Pattern Toolkits’.

What are Patterns?

Answer. "A Pattern in the context of Pattern Toolkits is simply design or an implementation pattern for implementing best practices in IT development and deployment. A pattern is not without some conceptual structure, an implementation, and some parameters."

When implementing complex IT solutions, irrespective of domain, technology or platform selections, it is natural to engineer 'patterns' that abstract implementation detail and focus solely on the concerns being addressed by the creation of the pattern. These patterns are often parameterized so the users of them (those who actually implement and deploy the solutions) can focus on programming them to suit their needs in use. Typically, these kinds of patterns manifest themselves in a set of API's that work together to solve a particular problem. Often, these patterns are formalized and documented. Often they are generalized to adapt to multiple contexts in use.

Patterns are most useful when they can be applied to a well-defined set of use cases or scenarios, and reused more than once.

Problem with reusing patterns include:

- Being able to constrain when they are used, and the context in which they are used.
- Being able to ensure that the parameters of the patterns are configured correctly in use.
- Packaging them with all their dependencies for reuse.

Unless you enjoy writing (and presumably can afford to re-write) every component of your software from scratch, patterns are the most common reuse mechanism in software development today. They tend to be platform and language agnostic, but in that, tend also to have multiple possible implementations. Hundreds of hits on the web for any software development topic are evidence of this. One thing that tends to transcends patterns are best practices, but in order to be effective and efficient and practical all best practice implementation require some form of specificity of platform, language and technology. The end result is an implementation pattern that declares its platforms, languages and technologies. If you can specify those aspects of a best practice pattern, and a consumer can live within them, then the benefits of reuse become significant.

Patterns can have unlimited scope. That is to say you can define a pattern from the smallest thing, like a line of code, to whole systems of software. But in general, larger patterns for complex software solution will almost certainly require narrower cases of reuse and smaller sets of parameters to be effective. As implementation scope increases, practical reuse tends to decrease. Keeping scope and parameters to a practical minimum aids in wider and more frequent reuse. One way to help manage larger scope is to break the pattern into smaller patterns that can be composed into larger software components and systems.

What are Pattern Toolkits?

Answer. "A Toolkit in the context of Pattern Toolkits is the delivered package of tools that is deployed and installed into the Visual Studio development environment. The tools work together to create, modify, manage and implement instances of the pattern defined within the toolkit."

In Visual Studio, in general a toolkit is compiled into and deployed as a 'Visual Studio Extension'. A 'Visual Studio Extension' integrates with Visual Studio using with the extensibility model of Visual Studio. These extensions are packaged and deployed as *.vsix files that install things like: editors, designers, commands, windows, etc. into the development environment for developing general or specific types of software. Visual Studio Extensions are managed in Visual Studio using the 'Extension Manager'. For more details about Visual Studio Extensions, see Customizing, Automating and Extending the Development Environment.

A 'Pattern Toolkit', is a special type of 'Visual Studio Extension' that is automatically compiled for you, and that provides all the information, guidance, tools and automation for creating instances of specific kinds of development patterns that you define.

A 'Pattern Toolkit' can be defined as being composed of the following 4 essential ingredients, which together deliver all the value of using a toolkit:

1. Assets, harvested for reuse from existing solutions where the pattern was derived.
2. A 'Pattern Model', describing how the pattern is configured when applied.
3. Automation, to expedite the implementation of the pattern in a solution correctly and consistently.
4. Guidance, to understand to pattern, and be highly productive with using the Pattern Toolkit.

To build a 'Pattern Toolkit', you follow this simple process:

- Identify a reusable pattern from existing solution implementations. Optionally, create a reference implementation (RI).
- Harvest assets from those existing solutions or RI. (i.e. code samples, libraries, frameworks, configuration etc.)
- Define your 'Pattern Model', and how the variability of your future implementation will be represented and configured in that model
- Apply Automation, to the Pattern Model, to expedite the application, configuration and refactoring of the assets as the configuration of the model changes.
- Apply Guidance to the Pattern Model, to educate and instruct users on how to use it to complete the pattern.
- Build, Package and Deploy the resulting 'Pattern Toolkit'.

> **Note**: It is unreasonable and unrealistic to expect that this process is executed sequentially from the first step through the last step in a single iteration. In reality, the sequence is completed in very short iterations. Applying effort at each stage as the variability of the pattern and assets is explored.
> It is highly recommended to build pattern toolkits using a highly iterative process for best results.

What is Customization?

The main purpose of customizing (or tailoring) a pattern toolkit is to reap whole or part of the value offered by the original toolkit. By reusing what has already been provided and only modifying the parts that need to be different, as long as the cost of customization is lower than the cost of creating new, then customization will provide a return on that investment. The process and tooling to customize a toolkit is the same as creating a new toolkit from scratch. It is a standardized process. Therefore, customization becomes an economically viable consideration for toolkit development and evolution.

Any pattern toolkit can be customized, and new toolkits can be created to derive from them. Customized pattern toolkits can be further customized, and further customized as requirements become more refined or custom. Whole families of pattern toolkits and whole value chains of pattern toolkits can emerge.

Customization also brings others benefits:

- Obtaining or creating broader purpose pattern toolkits and then tailoring them for either a specific organization or program, where there are certain compliance requirements at each level in the organization. Naming standards, coding conventions, technology choices etc.
- Customized toolkits can be successively customized, (or versioned) so that the toolkits can evolve as the patterns, requirements or technology choices evolve, fork or change.

When and Why Customize?

In general, you should consider customizing an existing toolkit when the pattern in the original toolkit is representative of the pattern and variability you want to present to your users. You can then customize the pattern model to extend or constrain it, and modify the assets, automation and or guidance to suit your specific solution implementation requirements. Most things in a pattern toolkit can be customized to change the way they 'appear' given a certain audience, certain requirements or a certain usage in a solution. There are exceptions to how much customization is acceptable of course, and if the pattern cannot itself be suitably applied to a set of requirements then customization is perhaps not the correct course of action.

Understanding the trade-off between specificity and reusability is a key consideration in this decision process. The more specific something is, the less more general reusability it will have. The ability to customize pattern toolkits waters this equation down somewhat because the specific parts of a toolkit can be replaced with more general parts, due the flexibility of customization in pattern toolkits.

The real cost to weigh in determining whether to customize versus build from new, is in the cost to develop or customize the reusable assets and custom automation classes needed in any toolkit. The cost of developing a new toolkit and designing and configuring pattern models pales into insignificance to the cost of developing reusable assets and custom automation.

Common Scenarios

The most common reason to customize a toolkit is to make that toolkit comply with organizational compliance requirements.

These kinds of requirements commonly include, things like:

- Changing the implemented solution artifacts to meet compliance to: static analysis policies, coding standards etc.
- Changing the vocabulary of a pattern to comply with an organizations established vernacular.
- Introducing traceability in either the generated output of the tools, on in the process of using the tools.

All these things, and many others, can be achieved with simply customizing an existing toolkit.

What can be customized?

There are several areas of a toolkit that can be customized depending on needs:

- The Pattern Model:
  - You can change the appearance and description of any of the elements and their properties.
  - You can hide or disable existing elements and properties, and add new ones.
  - You can modify the default values and value providers of existing properties.
  - You can constrain existing extension points, and new ones to open up the pattern.
  - You can hide or disable existing views, and add new ones.
- Assets:
  - You can modify or replace existing assets with your own versions
  - You can add new or different assets
- Automation:
  - You can disable existing automation.
  - You can add new automation.
- Guidance:
  - You can modify or replace existing guidance with your own versions.

What cannot be customized?

In general, the integrity of the original toolkit's pattern cannot be compromised with customization. You can therefore, in general, not delete or change the identity, type or behavior of any existing pattern elements in the original toolkit. This would potentially break any existing automation in the original toolkit (or any ancestor toolkits built upon it) that depend on the integrity of the original schema of the pattern model.

> **Note**: You can however, in most cases, hide, disable and replace all these things to achieve new or different structure and behavior, but the original integrity must remain intact.

- Pattern Model:
  - You cannot delete existing elements, properties or change the cardinality of their relationships
  - You cannot change the name (identity) of any element or property.
  - You cannot change the type of any property
- Automation:
  - You cannot delete any automation.
  - You cannot change the settings of existing automation.

Controlling Customization

In addition to the rules about what cannot be customized above, a pattern toolkit author can further define additional 'customization rules' that are applied to the things that can be normally customized in their toolkit, which prevent further customization of them in customized toolkits.

For Example, although (by default) the appearance attributes (i.e. display name, description, is visible etc.) of elements and properties in a pattern model can be customized, the author of the toolkit can decide to apply an exception, and define specific customization rules to specific elements or properties, to prevent customization of them.

The customization rules that can be applied are very granular. They can apply to very specific attributes of appearance and values, and to only very select elements in the pattern model. The customization rules also inheritable, so that whole hierarchies of elements in the pattern model can be applied with the same rules. The mechanism is very flexible.

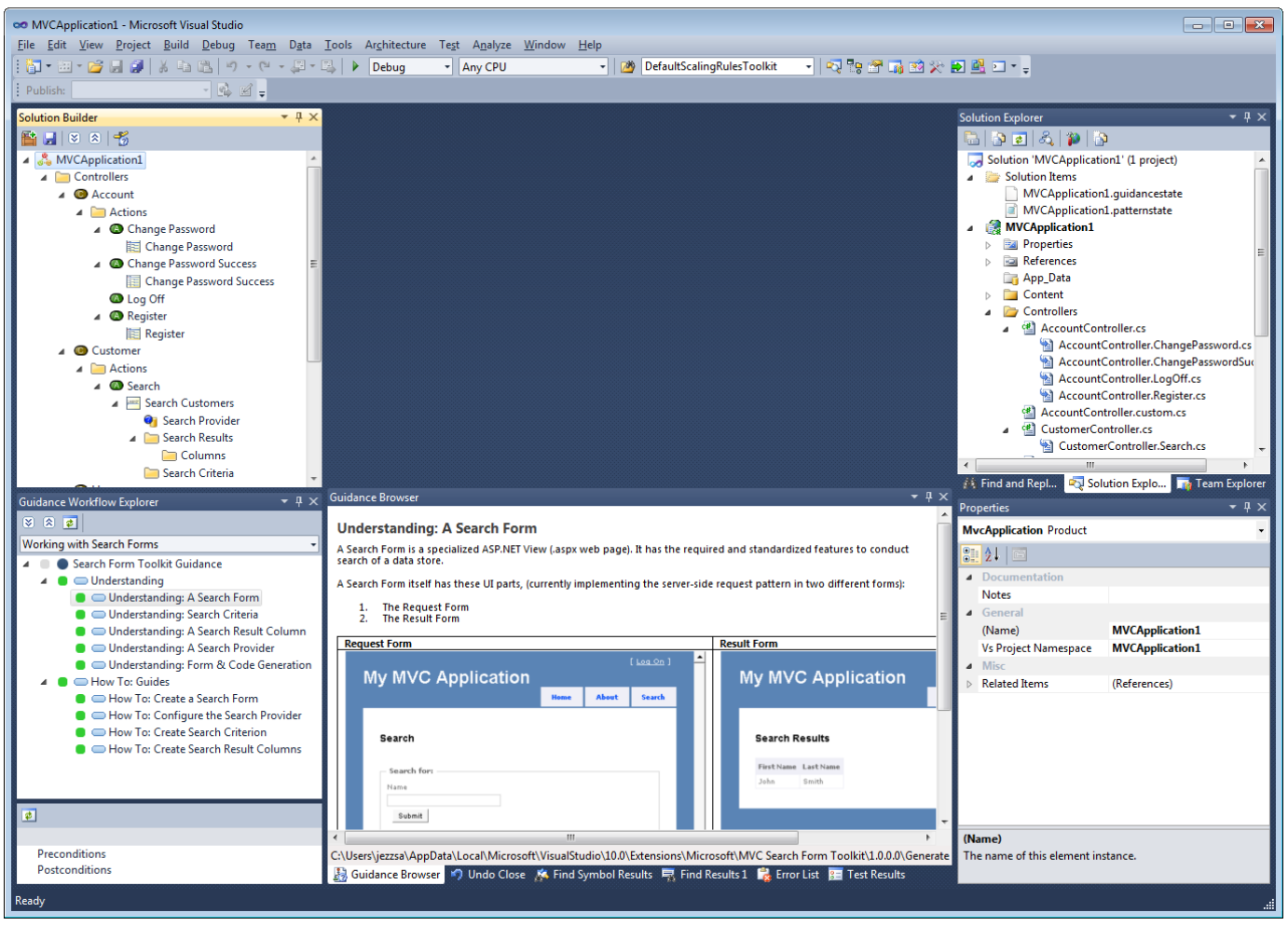
However, in general, applying these customization rules should be an exception to the rule, and should be used sparingly. Constraining customization of the pattern model can in some cases prevent appropriate customization, forcing a re-write of the pattern. For that reason, these rules are not applied by default and are left up to the toolkit author to apply at their discretion.

Solution Builder

What is Solution Builder?

Answer. 'Solution Builder' is a new tool window in Visual Studio where you manage, create and configure elements of your solution.

The 'Solution Builder' window is the place you go to view, manage and configure the elements of your solution that are currently managed by installed 'Pattern Toolkits'.



> **Note:** 'Solution Elements' are what are being displayed in the 'Solution Builder' tool window, they are the abstract representations of one or more artifacts in the current solution scope, be those projects, files or other artifacts visible from the development environment.

In the 'Solution Builder' window, you perform the following common activities:

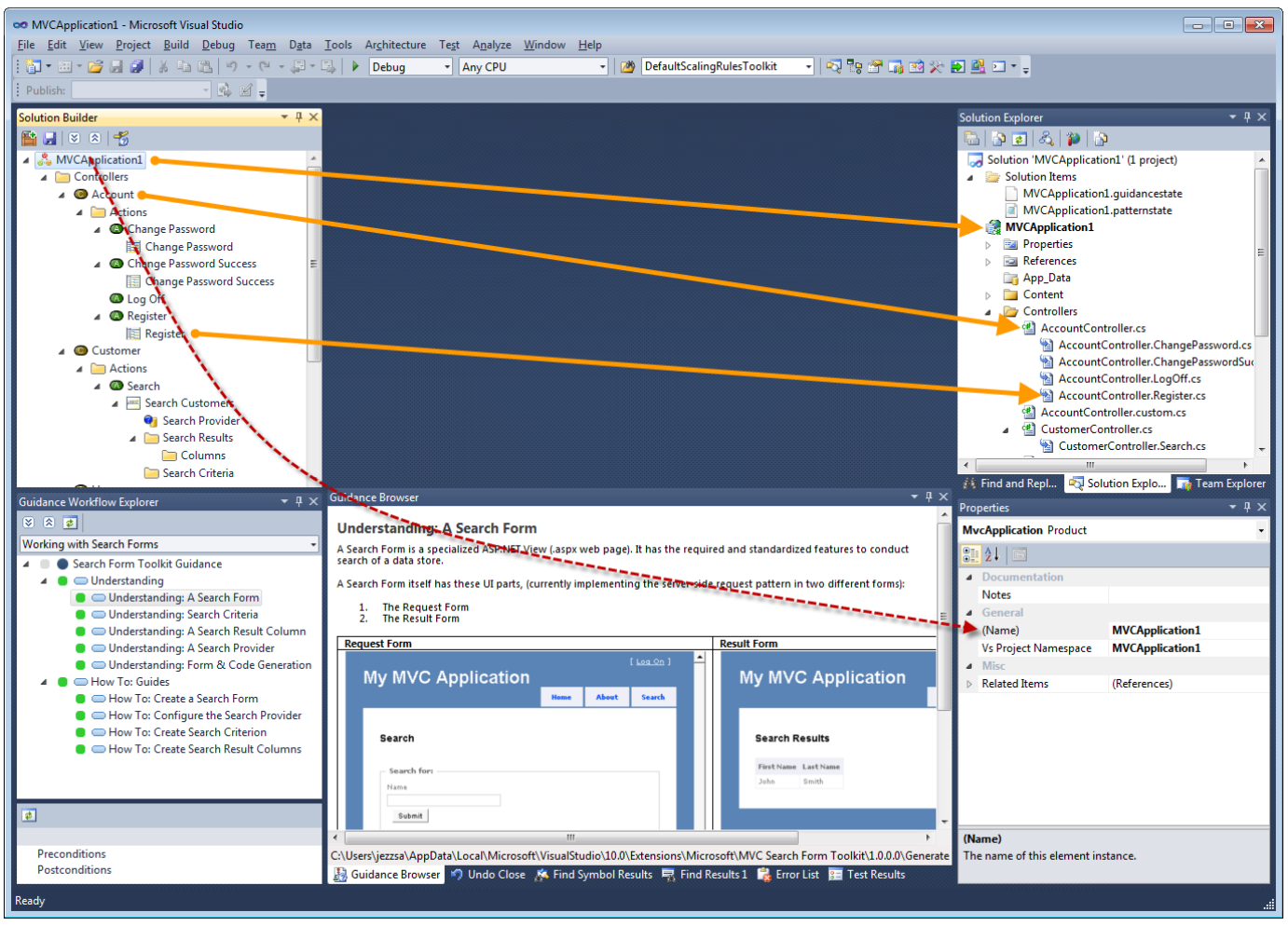
- View, and Browse the existing solution elements for the current solution.
- Configure the solution elements.
- Create new solution elements.
- Compose new or existing solution elements together.
- Invoke Automation.
- Navigate to other Related Artifacts in the development environment.
- Navigate to the Guidance that describes how to use them.

Note: In addition there is guidance for helping

What is a Solution Element?

Answer. "A 'Solution Element' is the representation/abstraction of one or more artifacts in the scope of the current solution."

A 'Solution Element' can represent any architectural concept or physical implementation and in some cases any composition or arrangement of additional artifacts that can be reached within or outside of the development environment. For example, in any given pattern toolkit, a solution element could represent one or more projects, files, database records, web service responses, configuration data, etc. A 'Solution Element' can have no physical representation, and just be notional. Its presence in the pattern could be simply structural or conceptual such as for containment or grouping of other solution elements. There are no bounds to what the solution element represents, large or small, abstract or concrete.



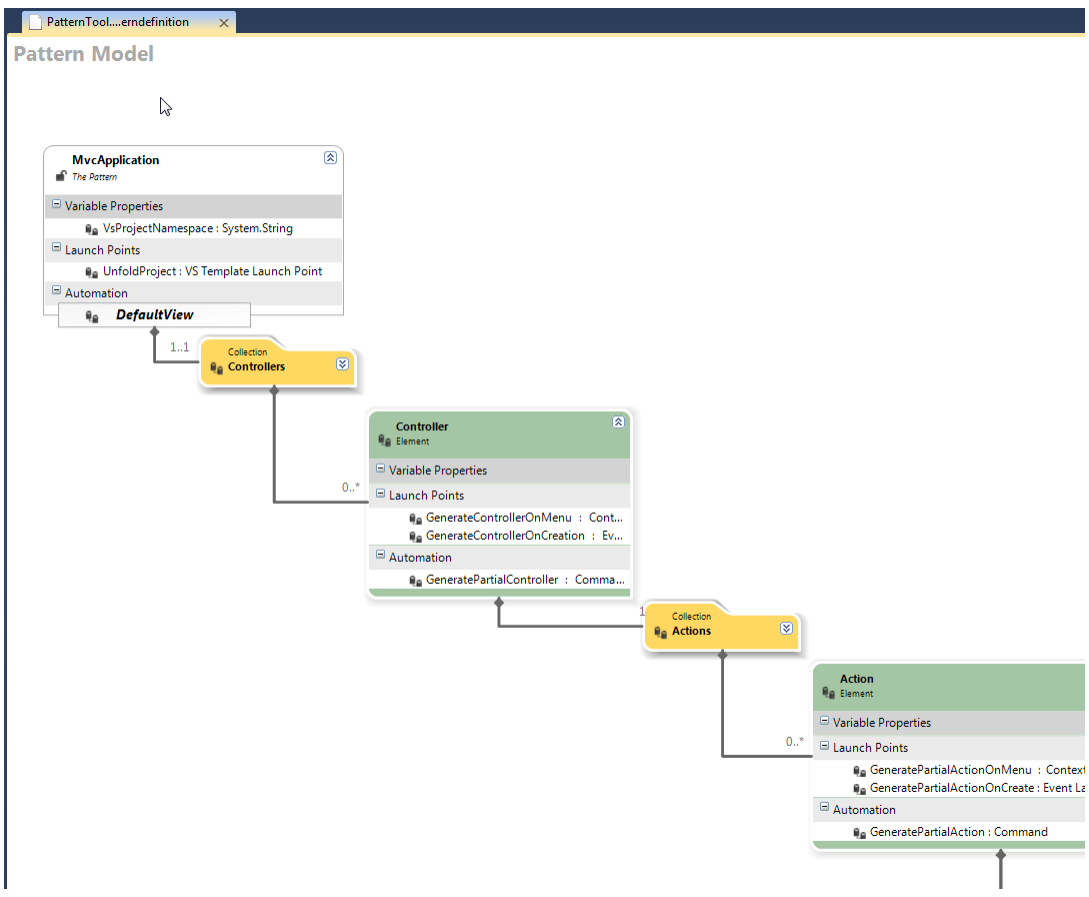
A user of a Pattern Toolkit, and the toolkit's built-in automation, work with defined properties of the solution elements to establish their present configuration and state. When verified as valid, the configured state is then used typically to automate and transform that state into a representative implementation in the solution. The sum of the state of all solution elements from a specific pattern toolkit represents the configuration of that instance of the pattern in the current solution.

> **Note**: The state is managed by 'Solution Builder', and persisted in a source controlled file in the solution (*.slnbldr).

Pattern Model

What is a Pattern Model?

The pattern model defines the schema (structure) of the pattern, it provides the user interface framework (model) and it provides a simplified abstraction of the pattern, to which automation and guidance can be applied. It is defined in terms of the language of the domain of the pattern, and adds behavior to the pattern with automation.

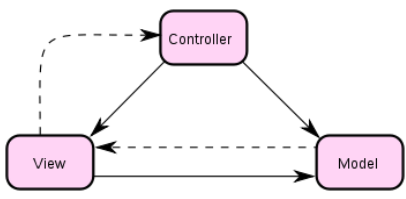


With this model you define all the descriptive elements of your model with names that reflect the pattern being described. Each property on each element (or collection) in the model represents a configuration point or variability point in the pattern, that someone using your model (or automation) can configure. See Commonality and Variability for more details.

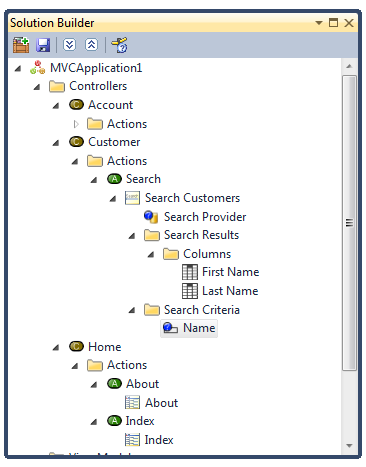
What it is and What it Isn't

A pattern model does not describe the entire pattern in all its implementation detail. There is nothing constraining that level of description, you can describe each and every detail if it is important, but the intent is to provide a simplified (abstracted) view of the pattern that focuses only what is variant within the pattern. That pattern variance is the focal point of configuration for the user of the pattern, and should be presented in a form that is simple to learn and understand.

Generally, a pattern focuses only on what varies in the implementation, and omits to represent the parts of the implementation that are fixed or given for that specific implementation.



For example: the pattern model to represent the classic Model-View-Controller pattern may well not include any representation of the Model concept at all if the model is implemented for the user in generated code. It may not even represent the relationships between Controller and Model, or View and Model explicitly. Those can be hidden from the user, and inferred, to allow them to focus on just providing the Views and controller Actions that drive the views. The pattern toolkit, only needs to know that given a set of configured views, configured actions for a controller, that it can implement the MVC pattern in a specific, predefined and consistent way. What it represents, and how it represents that to the user can be vastly different than the model on the right.



The image at the top of his page shows a pattern model that implements the MVC pattern, and generates an implementation of the Model-View-Controller pattern with a specific .NET C# implementation.

This image shows how the pattern is applied by a user in their solution.

The point is that a pattern toolkit does not need to model the theoretical MVC pattern as is defined and represented in software engineering academia. It needs only model the entities that are configurable by someone applying the pattern with a specific implementation, and it can apply automation to the pattern model to create a specific implementation of that pattern configured with the data provided by the user applying the pattern.

What is Commonality & Variability?

Commonality

Commonality is the collection of concepts, features, capabilities which are common or shared across all instances of use. This collection does not vary and is not configured in any way by those using the common capability. In a sense this collection is fixed, and can be assumed to be applied to an implementation automatically.

In the context of a pattern model, commonality is not explicitly represented, it is found embedded in the assets being reused that support the automation and implementation of the pattern.

Examples of assets that contain commonality would be the libraries and frameworks upon which code is generated to utilize or integrate with, or the parameters and configuration files that are generated in the implementation.

Variability

Variability is the collection of concepts, features, capabilities that vary across the applied instances of the pattern. This collection of variability requires direct configuration from a toolkit user, or provided by defaulted configuration in new instances of the pattern that can be refined by a user or by automation.

In the context of a pattern model, the variability of a pattern is represented in the hierarchical structure of the elements and their properties in the pattern model.

Examples of variability would include: names and values of parameters within various solution artifacts generated into an implementation of the pattern.

In essence, variability is the measure of the number of relationships and number of variable properties of a pattern, since any permutation of any combination of those should result in a slightly different implementation from the pattern. For every point of additional variability you raise the possible configured permutations of the pattern by a factor of 2.

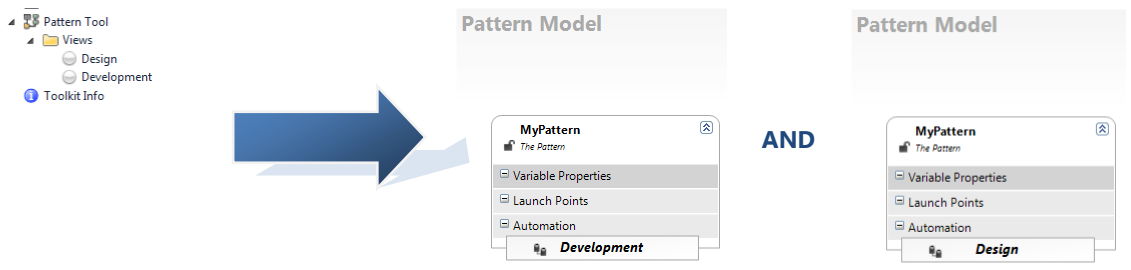
> **Note**: The most minimal variability that can be attributed to a pattern is one with only the pattern element, one view, no elements and no properties. Such a pattern has only one point of variability that can be configured by a user, and that is the name of the instance of the pattern created by the user. There is nothing else to vary.

What are Variability Views?

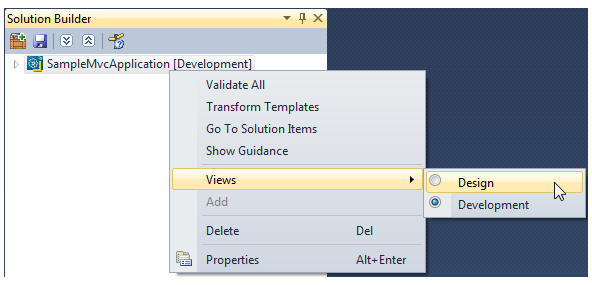
Every Pattern Model has one or more ‘Variability Views’. By default, a Pattern Model has one 'Default' view.

You can more than one view of your pattern to describe different aspects, cross-cutting or orthogonal concerns of your pattern. Such as: Architectural cross-cutting concerns like: logging, instrumentation, or Application Lifecycle or Configuration aspects such as: deployment, security etc.

Each ‘view’ of the pattern model has its own hierarchy of elements and collections describing the separate aspect of the pattern. This is useful in larger patterns where you may need to separate out different development activities of working with the pattern.



The toolkit users can switch between the different views to work with the different elements within them.

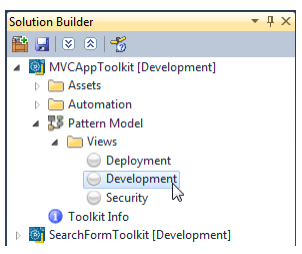


> **Note**: Every pattern has at least one view that is the ‘Default’ view. For smaller or simpler patterns that view is always represented to the user.

For larger more complex patterns that involve many development activities with many aspects, it is generally advisable to try and split the variability into multiple related views.

Example

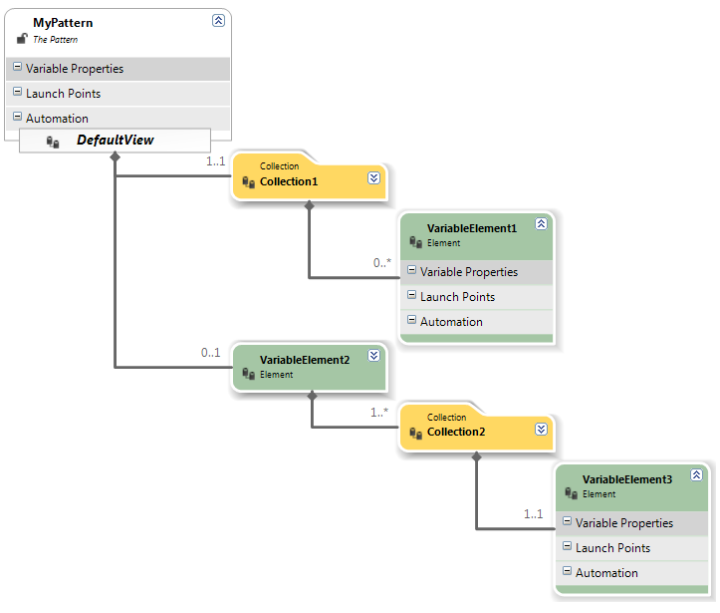
A Pattern Model for building Web Applications could utilize separate views: a view for ‘Development’ of the view navigation (i.e. MVC concepts), a second view for defining ‘Security’ concepts of the web application (i.e. authentication/authorization etc.), and a third view for defining ‘Deployment’ of the web application (i.e. hosting and configuration options).



All these views display elements that are directly related to the pattern, but grouped separately for focusing on different aspects of a web applications development.

What are Variability Elements, Collections and Properties?

Variability 'Elements' and 'Collections' are used to represent abstractions or concepts in a Pattern Model. They are the elements which the user creates, configures and interacts with. They are the state-full persisted atomic units of structure which display the variable aspects of the pattern to the user, and are the objects which automation and guidance operates directly on.



You use 'Collection' and 'Element' shapes to break up a pattern into arbitrary entities or concepts, each of which help group together variable properties of pattern. This is the primary means of describing the variability in your pattern.

> **Note**: The 'Pattern' shape is just a specialized 'Element' shape that also has its own set of properties.

You can use 'Collection' and 'Element' shapes in any combination, and cardinalities to represent the relationships between them (i.e. One-to-One, One-to-Many, Zero-to-One and Zero-to-Many).

> **Note**: 'Collection' and 'Element' shapes are configured identically and behave the same. The only tangible difference between them is that 'Collection' shape has a different visual appearance (a folder shape) to the author, and a different default visual appearance to the user (a folder icon). An author can tailor the visual appearance of a 'Collection' by setting a different icon to it.

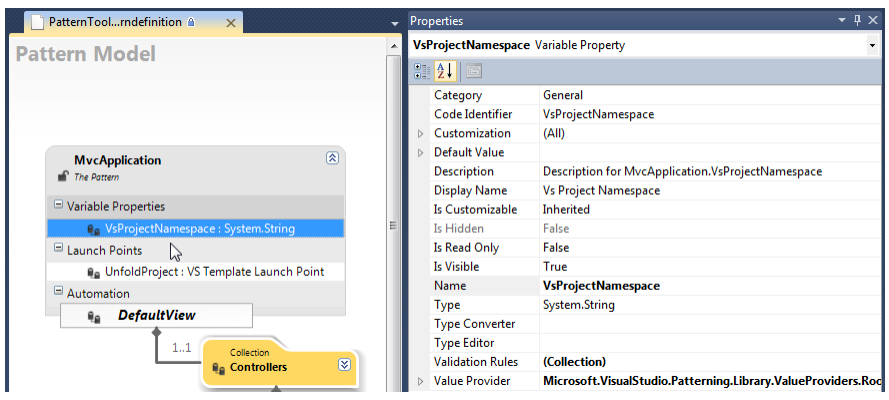
A 'Collection' shape is intended to be used by an author to represent either hierarchy or grouping of elements, even though the 'Element' shape can be used in exactly the same way. The choice is entirely the author's as to how to use collections and elements on the design surface of the Pattern Model.

Variable 'Properties' are used to configure the variance of 'Elements' and 'Collections'. They provide the parameters and persisted state of the elements of the pattern. A 'Property' has a name and primitive data type (i.e. string, int, bool etc.). They have many attributes that control their visual appearance to a user including: a display name, a description and visibility and modifiable attributes. And several attributes for controlling and verifying their value.

> **Note**: Properties also support dynamic values. That is, you can configure automation to fetch the value of a property from the environment, and create a 'calculated' property. This kind of variability is not per-se represented in the structure of your pattern model in its elements, but nonetheless provides variability to the implementation of the pattern.

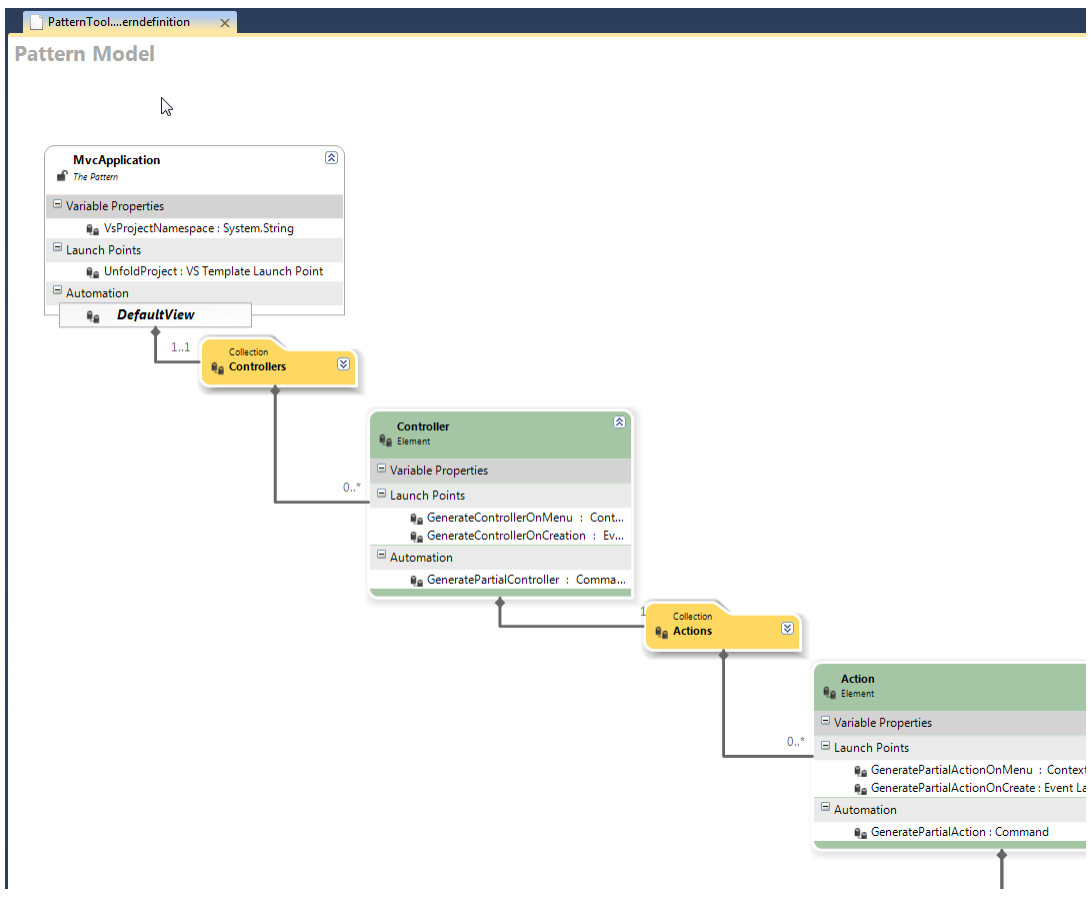
Not all 'Properties' must be visible to or be modifiable by a user; they can be hidden or made read-only, and access restricted to automation only.

Validation rules also can be applied to variable properties to ensure they always contain valid data at all times.



For Example

A Pattern Model for building Web Applications could define 'Collection', 'Element' and 'Properties' similar to this.



What are Extension Points?

'Extension Points' are the means to break patterns into smaller toolkits that can be pieced together by users. This ability allows authors to keep toolkits small and focused, but able to participate with other toolkits to make up larger pieces of solutions. 'Extension Points' are also very useful for delegating pieces of a pattern to other toolkits when that 'piece' of the pattern is itself variable. This allows the user to vary their application of a pattern from one toolkit by choosing a piece of it to be implemented by the pattern within another pattern toolkit.

An 'Extension Point' defines a 'data contract' between the toolkit 'Declaring' the extension, and the toolkit 'Implementing' the extension, in a standard 'Requires'/'Provides' relationship.

There are three main roles played with Extension Points:

1. Declaring for Extensibility
2. Implementing for Composability
3. Integrate for Reusability

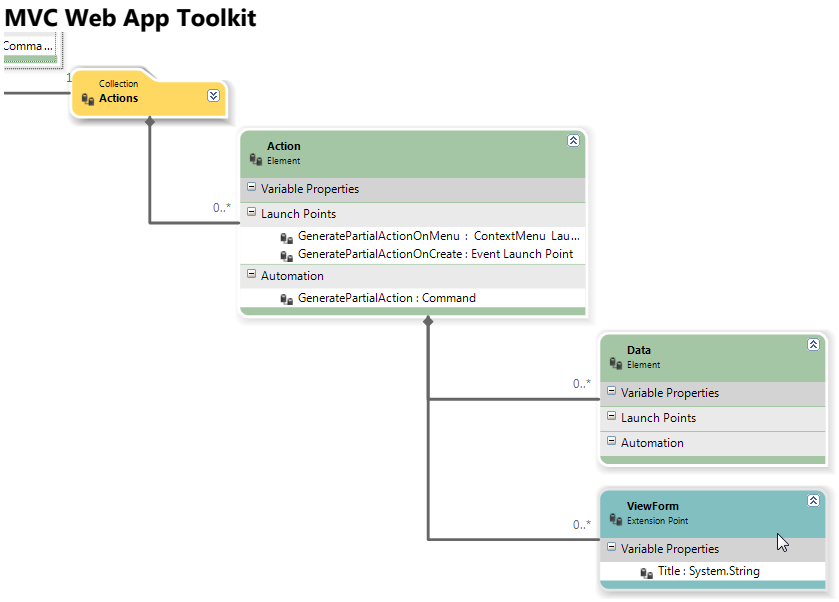
Declaring For Extensibility

You *declare* an 'Extension Point' in your pattern when you want that element to be implemented by another pattern toolkit.

You may want to do this to delegate the implementation of that element of the pattern to another pattern toolkit, because that part of the pattern is itself variable. This leads to the pattern being composable. Or, you may want to do this reduce the size of the pattern you are implementing, but allow it to participate as part of larger patterns provided by other toolkits. This leads to the pattern being far more reusable in a solution.

For Example, in the MVC Application pattern toolkit below, an 'Extension Point' is used for the view concept, the 'ViewForm' element. The toolkit author decides not to try and define and represent all possible web forms of all possible web applications in this toolkit. This toolkit is focused only on defining web applications that implement MVC that have views which are nonetheless still an integral part of implementing this pattern. Recognizing that there may be many different types or archetypes of web form possible, she decides to make 'ViewForm' extensible, and allow other pattern toolkits provide the implementation them. The only thing that this toolkit requires of a 'View Form' is that it has a 'Title' property, as this is used by this toolkit in some part of the implementation of the web application.

MVC Web App Toolkit



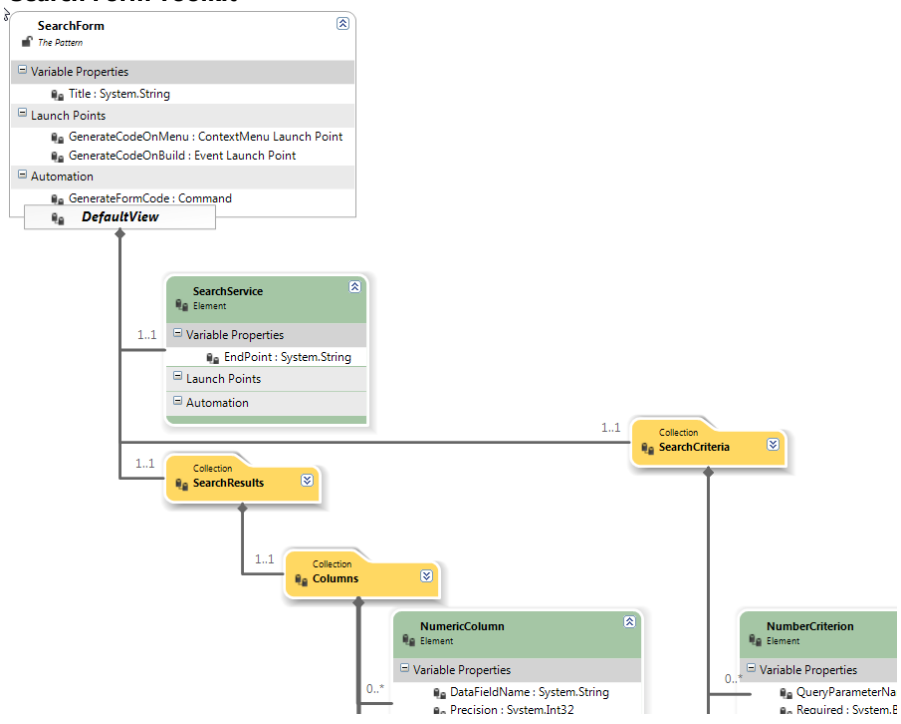
Implementing for Composability

You *implement* an 'Extension Point' when you want to extend another pattern toolkit.

You do this so that your toolkit can be 'composed' by a user with other patterns provided by other toolkits.

For Example, (following from previous example) the 'Search Form' pattern toolkit provides an implementation of the 'ViewForm' extension from the MVC Application pattern toolkit. This toolkit defines an implementation of a commonly developed 'Search Form' where there is a defined set of search criteria to execute a search (using a RESTful service), and a standard form to present the search results, which each link to other 'details' web pages for that search result.

Search Form Toolkit



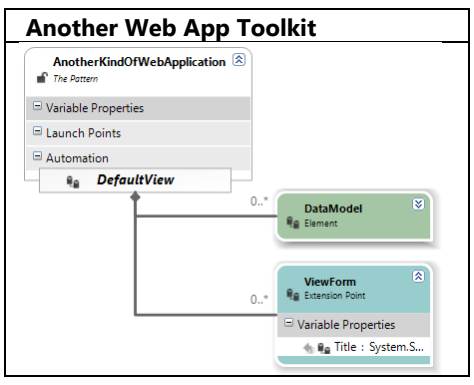
Integrate for Reusability

You *integrate* an existing 'Extension Point' into your pattern model from another pattern toolkit when you want to reuse the pattern toolkits that already *implement* the specific 'Extension Point'.

You do this so that your toolkit can benefit from reusing the extensions provided by other toolkits, so that this toolkit is not re-defining the extension point, and makes use of the extended toolkits already available.

For Example, (following from the previous example) a new type of Web Application pattern toolkit is built that could benefit from using the same notion of 'web views' as the MVC Web Application toolkit. One of the goals being to integrate with all the other toolkits that already implement the 'ViewForm' extension point (such as 'Search Form' toolkit form previous example). The new toolkit simply integrates the existing 'ViewForm' extension point from the MVC Web Application toolkit as part of its pattern model, and now gets all the benefits from the other toolkits implementing this extension point.

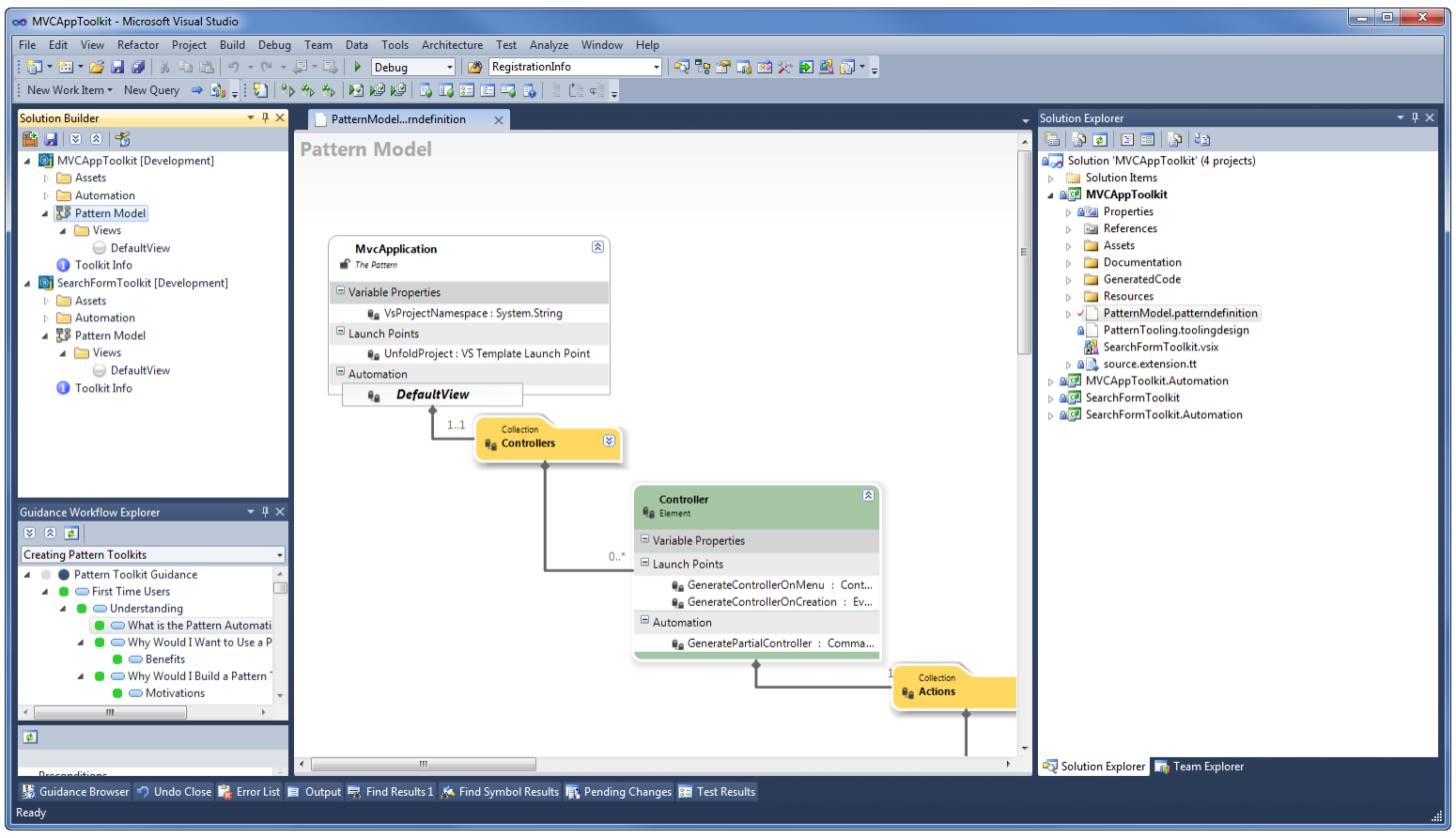
Another Web App Toolkit



What is the Pattern Model Designer?

The Pattern Model Designer is the visual designer in Visual Studio used to create and configure the 'Pattern Model'.

Every Pattern Toolkit project includes a 'PatternModel.patterndefinition' file that when opened, opens in the 'Pattern Model Designer'.



What is Cardinality?

Cardinality is the measure of the number of instances that a 'parent' element is permitted to have of a given 'child' element.

In pattern models, each 'View', and each 'Collection' or 'Element' supports zero or more instances of other child 'Collection' or 'Elements'. This Cardinality is used to control how instances of 'Collections' and 'Elements' are created by the user of the pattern.

The following cardinalities are supported:

Cardinality	Notation	Description
OneToOne	1…1	The parent element has one and only one instance of the child element.
OneToMany	1…*	The parent element has one or more number of instances of the child element.
ZeroToOne	0…1	The parent element has either one instance of the child element or none.
ZeroToMany	0…*	The parent element has any number of instances of the child element (including none).

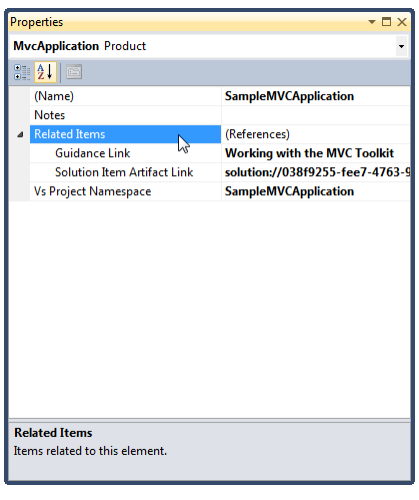
Note: the 'Pattern' element always supports 'OneToMany' Views. This cardinality is not configurable.

What are Related Items?

Related Items are references to items or data that do not exist within the pattern model. These items can be are related to, or represented by, the elements of the pattern model. Typically for tracking purposes and for automation. (i.e. the physical source files that are generated from a solution element or source data database records containing data represented by the pattern, etc.).

> **Note**: Related item may exist outside of the Visual Studio environment, such as messages from remote web services, or records in a remote database.

Every solution element has a configurable collection of 'Related Items' and each of these items can be of a different type.



Related Items are a very flexible mechanism to allow the pattern toolkits to integrate with other systems or other tools in the Visual Studio environment to expedite automated development of the solution.

> **Note**: Related items which exist in the 'Solution Explorer' window are commonly referred to as 'artifact links'.

Assets

What are Assets?

In the context of pattern toolkits, an asset is any predetermined harvested and/or developed artifact that is reused to be part of the implementation created by a toolkit (called 'Solution Implementation Assets'), or developed, configured and applied in the process of constructing and applying of the pattern defined in the toolkit (called 'Automation Assets'). In either case, both kinds of assets are contained within, and deployed by a pattern toolkit in a solution.

Assets are not necessarily pre-determined before creating a toolkit, although one or more are expected to be identified in order to establish that a pattern toolkit should be built. Assets can be discovered in the development process as variability of the pattern is teased out.

See 'Solution Implementation Assets' and 'Automation Assets' for more details.

What are Solution Implementation Assets?

This kind of asset is either delivered in or applied to the solution implementation by a toolkit.

These assets are usually harvested from one or more reference implementations where the pattern has already been applied or can be derived from. In some cases, these assets will need individual development effort to get them in a shape for reuse. See the process of harvesting for more details.

There are typically two classes of solution implementation assets: Fixed and Variable.

Fixed Assets:
Fixed solution implementation assets are supplied by the toolkit (as is) and will not expected to vary between different implementations by the toolkit. The come already configured and ready for inclusion in to the solution implementation.

In most cases, they establish a base architecture and/or technology platform for the solution implementation.

Examples of common fixed solution implementation assets include:

- Class libraries, pre-compiled libraries
- Frameworks (application frameworks or platform frameworks)
- Graphic resources, executables, build dependencies etc.
- etc.

> **Note:** All these assets are deployed by the toolkit into the solution, and typically require no additional pre-configuration by the toolkit.

Variable Assets:
Variable solution implementation assets are also supplied by the toolkit in some form, to be modified either by the toolkit with automation, or by the user of the toolkit to complete the solution implementation.
In most cases, they establish what is physically variable within a specific solution implementation.

Examples of common variable solution implementation assets include:

- Partial code.
- Configuration files.
- Document templates.
- Project/Code template outputs.
- etc.

> **Note:** The output of a project template can be in fact a variable asset. (i.e. perhaps intended to be changed by the toolkit or user in the solution implementation), but the project template itself is actually fixed asset, and is an example of an automation asset when unfolded by a toolkit.

What are Automation Assets?

This kind of asset is pre-configured by an author of a toolkit, and is used by the toolkit to create or configure solution implementations.

In general, automation assets are designed to be reusable across many toolkits, and libraries of these assets can be built to share between toolkits.

A library of general automation assets is provided for some common needs in toolkits, but in many cases of more complex toolkit development, automation assets will need to be developed, and will be potentially added to other libraries of these assets.

Automation assets almost always require some configuration from a toolkit author to be configured for use in a particular toolkit.

Examples of common automation assets include:

- Commands, Conditions, Value Providers, Validation Rules, Wizards etc.
- Text Templates and code generators
- Visual Studio project/item templates
- Domain Specific Languages (DSL's)
- etc.

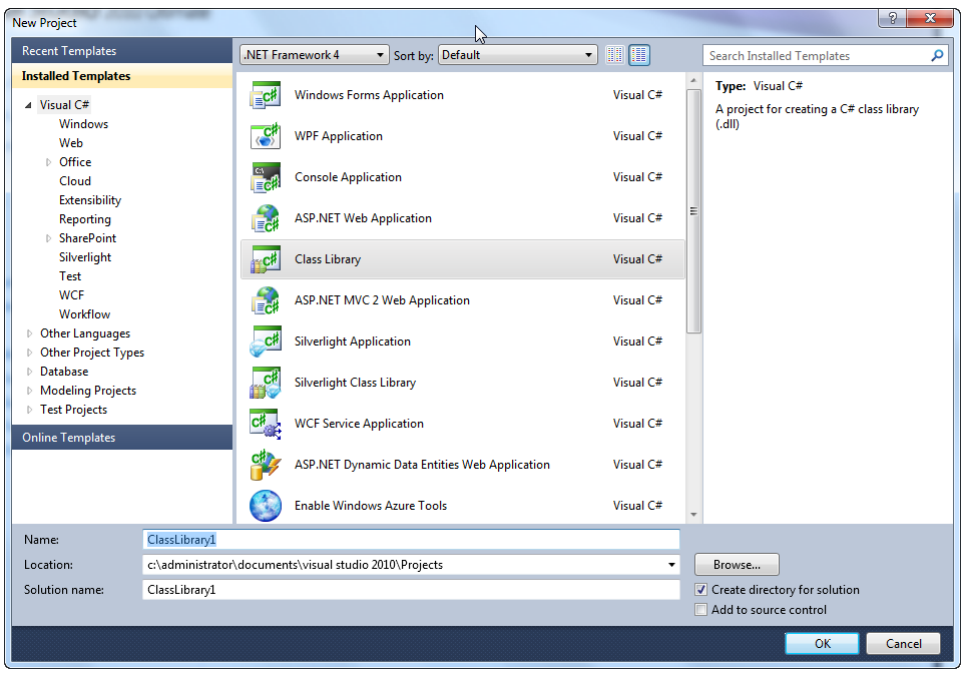
> **Note:** All Automation assets and their dependencies are installed and deployed by the toolkit.

What are VS Templates?

Visual Studio Project and Item templates are the means to lay down projects and solution structure and starting artifacts in Visual Studio solutions. Typically, a Project template contains a project file (of a given type) with files and folders within it. An Item template is typically a single file. This template mechanism has been around for a long time and there is a lot of support in Visual Studio for creating and working with these types of templates. See Visual Studio Templates for more details.

In Visual Studio you use project and item templates whenever you create a new project, or whenever you add a new item to an existing project using the Add New Project/Item dialog box. With these templates you can 'unfold' a new solution with one or more projects, and/or one or more related items (files).

> **Note**: Project templates are made distinct from item templates.



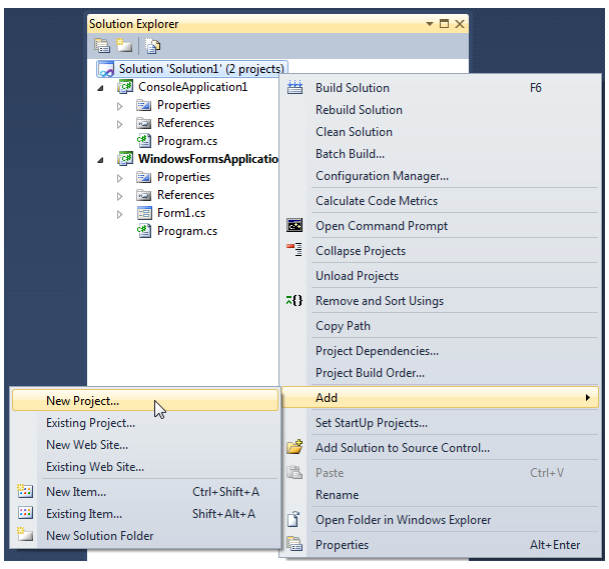
The 'New Project' dialog box displays all the project templates installed in Visual Studio, categorized usually by development language first, and then technology.

> **Note**: There are a few exceptions to this categorization pattern, usually where the project type is agnostic to the development language. (i.e. New Blank Solutions, setup projects or database and modeling projects)

You can select the template, name the project or item, and determine where to 'unfold' the template on your hard drive. If a solution exists already that template is unfolded into that solution. If no solution is open at present, and you create a new project using a project template, a solution is created for you. If you create a new project, then the existing solution is closed and a new solution is created for you.

Contextualizing Templates

When using Visual Studio to 'add' new items to a solution, you would select the 'parent container' (i.e. the solution, solution folder, project or project folder) in 'Solution Explorer' where you want the new item to be created.



By default, if no explicit selection exists then the solution is assumed. That sets the parent context for where the template is unfolded. The template is 'unfolded' in that location and the artifacts added to the selected parent container in the solution.

Text Substitutions

All VS Templates support text substitutions, which is the mechanism by which you can feed data gathered from either the environment or from a Template Wizard (displayed prior to the unfolding) into the template. These substitutions are performed during the unfolding process so that the templates are setup correctly for use in the environment.

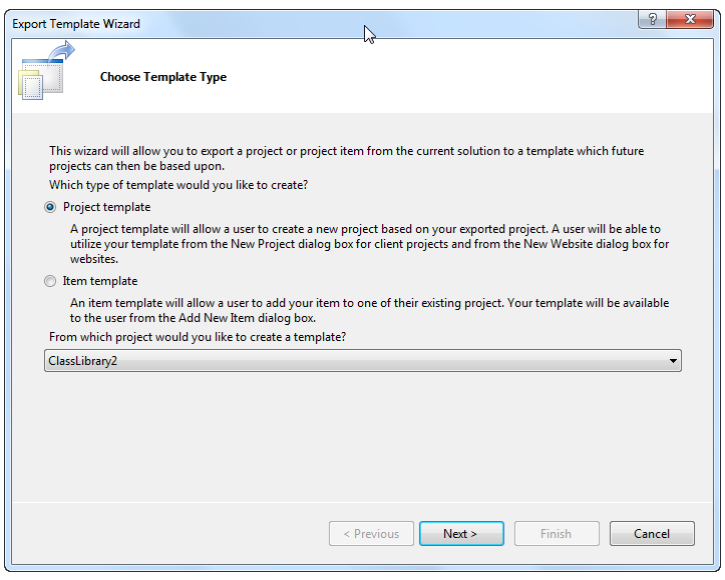
Template Makeup

VS Templates are built into *.zip files and installed to known directories on the machine where Visual Studio can access them. The zip file contains all the physical files and to be unfolded, and a manifest file (*.vstemplate) that describes the physical layout of the files to be unfolded on disk. It also defines the information about the template that is seen in the Add/New Project/Item dialog.

Creating Templates

Creating a VS Template from scratch, involves creating a *.zip file containing the files and folders, and writing the manifest. Although the manifest is just an XML file it is still a tedious manual process and very error prone, and relatively difficult to troubleshoot.

Visual Studio does however provide an 'Export Template Wizard' (File | Export Template...) that eradicates the need to create a VS template from scratch. Once an initial template is exported, it is far easier to refine it manually. And in pattern toolkit projects, the VS templates are automatically zipped up and packaged in the toolkit.



The basic process for using the Export Template Wizard is that you build up your project or item in a solution in Visual Studio, and then run the 'Export Template Wizard' to capture the project/item template. The 'Export Template Wizard' takes care of creating and naming the *.zip file, creating the manifest with basic descriptive information, and applies some default text substitutions to the project and files in the template. Exported templates are then copied to known locations on your disk to be installed or reused across solutions.

When to Use

VS Templates are most appropriate to use when creating an initial project structure for the toolkit to work with, that has some structure on disk, contains multiple static ('fixed') artifacts that may or may not be textual, and requires only little contextualization to integrate into the solution being built.

VS Templates are one of the primary building blocks when it comes to creating or configuring solutions using Pattern Toolkits. There is a lot of support for importing, creating new, and using project and item templates as a means of capturing, harvesting and reusing existing solution based artifacts as assets in a toolkit.

What are Text Templates?

Text Templates (*.t4 or *.tt files) are the templated input used for creating text code generators in Visual Studio. See Code Generation and Text Templates for more details.

A Text Template is used to output a text file (i.e. C# source code file, XML file, RTF document, text file, HTML webpage etc.) containing content that can be variable depending on the data source that the template operates on. The text template is a mixture of text blocks and control logic that generate the text file.

You use C# or Visual Basic to program the text template to get the desired outputted text, and this code navigates over a data source upon which the code makes determinations of what text should be output. A text template looks similar to a classic ASP web page with mixed HTML and server side code statements that make the substitutions.

For example:

```
<#@ 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="SamplePatternToolkit.Automation.dll" #>
<#@ Import Namespace="SamplePatternToolkit" #>
<#@ Import Namespace="System.Linq" #>
<#@ Output extension=".cs" #>
<#
        var pattern = ((IProductElement)this.Element).As<IMyPattern>();
        var defaultView = pattern.DefaultView;
        var anElement = defaultView.VariableElement1;
        var namespace = anElement.Namespace;
#>
//------------------------------------------------------------------------------
// <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>
//------------------------------------------------------------------------------
using System;

namespace <#=namespace#>
{
        public class <#=anElement.InstanceName #>
        {
        }
}
```

This T4 template yields the following code:

```
//------------------------------------------------------------------------------
// <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>
//------------------------------------------------------------------------------
using System;

namespace MyCompany.MyNamespace
{
        public class AClass
        {
        }
}
```

Similar to unfolding a VS Template, generating with a Text Template typically yields development artifacts with variable content. A Text Template yields only a single textual file, whereas a project or item template may yield one or more related files and populated folders, but there are several key advantages of a Text Template over VS Project/Item Templates:

- The content of a code generated file can vary far more than a VS Template because a code generator has full access to a data source object model upon which it can navigate and make complex determinations using C# code as to what to output. A VS Template has no data source except what is already in the current solution, and even though a VS Template Wizard can gain access to such a data source, the template can only accept primitive textual substitutions.
- The file generated from the Text Template can be varied in its filename and/or location in a solution. A VS Template has a predetermined location to be unfolded in the solution, and the files within the template will have pre-determined names (and extensions).
- Code generators (text templates), in general, also have another powerful advantage over one-shot mechanisms like VS Templates, and that is that they can regenerate the contents of their files at any time, overwriting out-dated content. VS Templates cannot achieve this capability by their nature. In this way, text templates can evolve and refactor an implementation as a pattern is built-up or configured.

In summary, a code generator is more appropriate to use when the content of the file is textual in nature, and its content varies as the solution implementation develops.

Text Templates are therefore another very important primary building block when it comes to creating or configuring solution using Pattern Toolkits. There is a lot of support for creating new, and using text templates as a means of capturing and reusing existing code based artifacts as assets in a pattern toolkit.

Guidance

What is Guidance?

Guidance is a new capability in general development that provides contextual information about what it is you are developing and how to develop it in a form that includes text, images and other media, to give assistance in developing software in Visual Studio.

As distinct from general Help reference topics on general development or technology topics, guidance is intended to impart expert domain knowledge with recommended practices and processes on how to be productive for the specific solution being built.

Typical guidance, like the guidance you are reading, includes:

- Overview information on the architecture of the solution being built and technologies used.
- Explanations of why and where that architecture is used.
- How and when to use that architecture for the specific solution being built
- Instructions on how to use the purpose built tooling to be productive with this architecture.
- References on how to troubleshoot, test and integrate the software produced as a result.

Although, much of this kind of information can be found from other sources either externally from the public domain, or internally within institutionalized knowledge in a parent organization, guidance is meant to be the condensed collection of that knowledge applied to specific development domain and further contextualized to the current solution being built. Guidance, is not just a high level description of what is to be built, it also contains how to build, test, deploy and troubleshoot it.

In the context of using a Pattern Toolkit, guidance is associated with individual solution elements being developed in 'Solution Builder'.

Guidance is viewed in the 'Guidance Explorer' tool window in Visual Studio, where the user can browse topics, and complete instruction lists that help them be most productive with understanding and using the toolkits in the current solution. That guidance can have state in the current solution, so that users on the same team can pick up work from where other user s have left off. Guidance can also track and maintain the state of the solution being built, so that instead of just instructing a user in text what to do next, the guidance can contain hyperlinks that actually execute the tools to do it for them. In this way, working through prescriptive instructional guidance can be very productive for the solution development team.

Each Pattern Toolkit being built delivers its own set of guidance to help users use it patterns and automation to get their work done more productively.

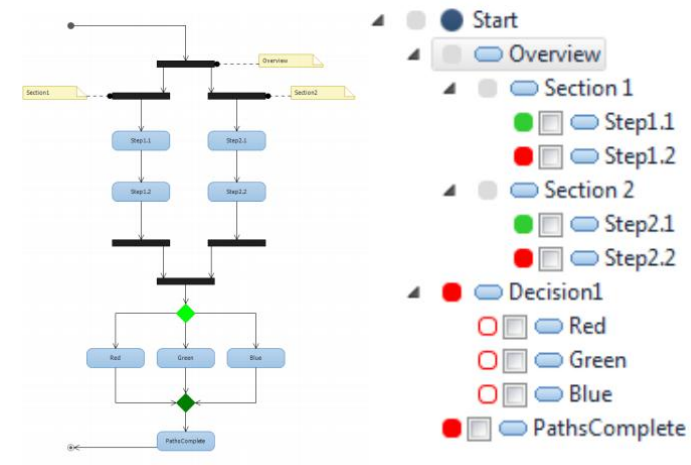
That guidance is composed of one or more a 'Guidance Workflows', each of which is a multi-pathed workflow of individual 'Guidance Documents' (topics), each of which has some browsable content, and each of which may or may not have state to indicate progress through it, and also contain links that automate some part of the described process.

What is a Guidance Workflow?

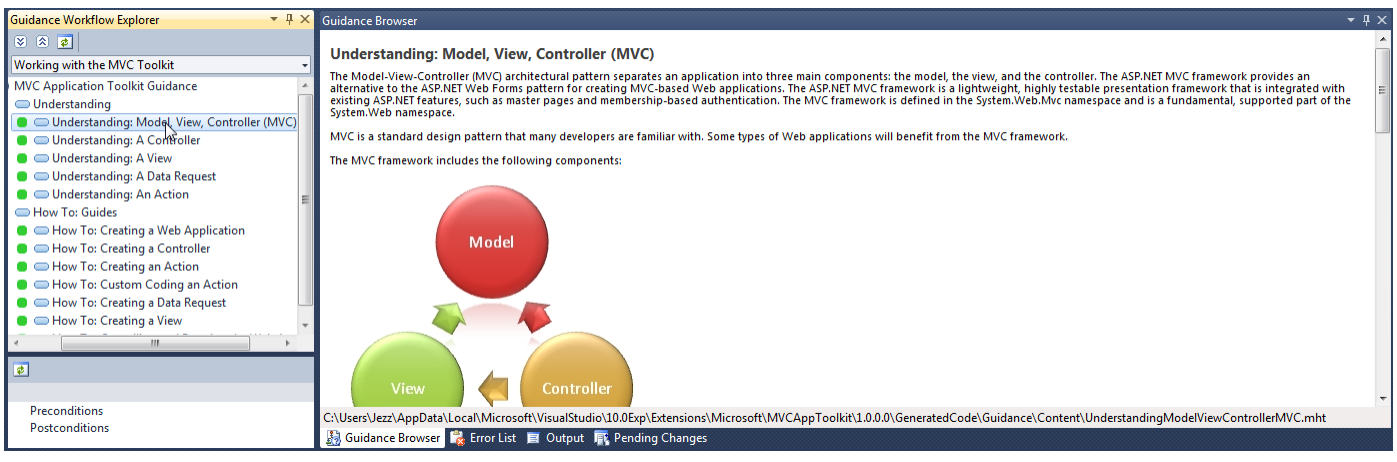
A Guidance Workflow describes the structure (layout) of a set of guidance, that may include both informative and instruction guidance steps.

A guidance workflow operates in much the same way as a classic state-full workflow, in that each step or 'activity' may have an individual state of either: ready, blocked or complete. When applied to guidance, this state directs readers on what they can do now, and what they can't do yet, resulting in a guided set of instructions.

The state of each activity in the workflow is governed by pre and post conditions on the individual activity that must be satisfied to move the state from blocked to ready to complete. The reader can either manually complete the step (ticking a checkbox), or automation can calculate the completion of the step using these pre/post conditions (or some combination of manual and automation). Each activity has a set of name/value pair properties that can also be used to hold state that can be used in the calculations as well.



Each activity may also have associated to it browsable content that can be viewed in the 'Guidance Browser' window when the activity is selected. This content can contain links that are active based upon the state of the activity, and can execute automation using the state of the workflow, and gathered state of the development environment.



What is a Guidance Document?

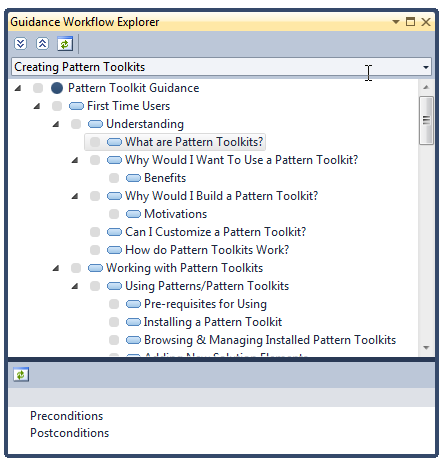
A Guidance Document is simply the content document which can represent a topic of any node in the guidance workflow.

This document is viewed in the 'Guidance Browser' window in Visual Studio. That content can be static content as with MHT page or other textual page, or can be dynamic as with an online web, wiki page or community site page. Ultimately, the guidance document boils down to a URI to some accessible source.

The content of this document is typically some form of HTML that can contain links to other documents within the guidance workflow, or link to external documentation and sites on the web. The content can further contain special automation links (content links) which execute automation commands. These commands can use the state of the current activity in the workflow to configure the command also, so that when a command executes it is contextualized to the work being done in the workflow.

What is the Guidance Explorer?

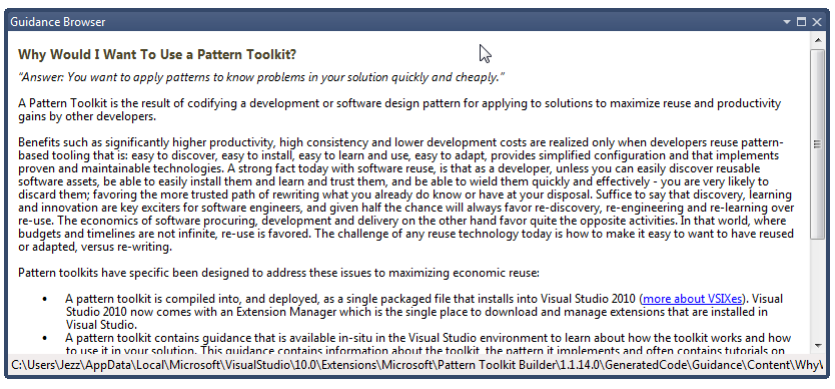
The 'Guidance Explorer' window is the tool window in Visual Studio where you can select, view and interact with the currently active 'Guidance Workflow'.



The Guidance Explorer window works with the 'Guidance Browser' window to show the content of selected topics.

What is the Guidance Browser?

The 'Guidance Browser' tool window in Visual Studio is where you browse and interact with the content of each 'Guidance Document'.



Automation

What is Automation?

Automation is the aspect of a Pattern Toolkit that makes development with the toolkit significantly more productive than hand crafting the software from scratch in a traditional way.

In general, it is the automation that speeds the development with the pattern toolkit by performing activities on behalf of the user that ensure the software being implemented is expeditiously, accurately and consistently implemented.

Without automation, and with guidance, a pattern toolkit could only instruct a user on what to do with its solution-based assets (such as frameworks, libraries and code samples). With automation however, a pattern toolkit can for example: automate the unfolding of project templates, generation of source code, validation of pattern models, support drag and drop, provide contextual menus, run custom tools, refactor code and configuration, synchronize the solution artifacts with the design, etc. There are many areas of a toolkit that leverage automation to make building a solution with them orders of magnitudes faster and cheaper than handcrafting solutions from scratch.

In a Pattern Toolkit, automation is provided by one or more collaborating 'Automation Extensions', which themselves are extensible and configurable.

What are Automation Extensions?

The automation mechanism for Pattern Toolkits include constructs called 'Automation Extensions' such as: Launch Points, Commands, Conditions, Value Providers, Validation Rules, Events, Wizards, etc.

The automation framework underpinning pattern toolkits is in fact highly extensible and pluggable using MEF. Meaning that you can build additional automation extensions, for example other Launch Points or automation extensions, to suit your additional automation needs.

The provided automation extensions from the NuPattern Toolkit Library (i.e. Launch Points, Commands, and Conditions etc.) have been designed to work together to provide most of the basic automation needs for most toolkits.

But how do these automation extensions work together?

Commands, Conditions, Validation Rules, Value Providers, Conditions, Events and Wizards are primitive automation extension types which supplement almost all other existing (and future) automation extensions to provide the hooks, triggers and configuration options for most of the integration needs of Visual Studio. They are general purpose, and in of themselves provide little of any value.

- Commands execute a piece of programmed automation.
- Conditions evaluate a given set of data.
- Validation Rules, evaluate some business rule on an element in the Pattern Model.
- Value Providers simply supply a value.
- Wizards data bind to an element in the pattern model and display a user interface.
- Events are a mechanism to handle the raising and catching of events inside or outside of the Visual Studio environment.

These are the types of automation extension. There are many derived instances of each of these types which further extend their usefulness. For example, there are commands that implement code generators, and commands that validate models, and several Conditions that evaluate different data sources, many different Validation Rules to evaluate a pattern model, and Value Providers that access various different services in Visual Studio etc.

Automation Configuration

Furthermore, each one of these automation extension types can support both design-time and runtime configuration that together provide the context and data with which to execute.

The author of the toolkit decides to apply a specific automation extension, let's use a Command as an example here, to a specific element on the Pattern Model. Each automation extension type can declare (in its class) a set of design time configuration properties, and a set of MEF imports from its execution context that it needs to perform its specific task.

Design-time configuration is provided by the author configuring the declared properties. Many of these properties will accept static defined data form the author, or the author can choose to configure a specific Value Provider to that property which will fetch the value dynamically when the user is using the toolkit.

Runtime configuration is provided by importing (using MEF) other services and available data sources in the Visual Studio environment.

Together the combination of design time and run time data forms a context for the automation to execute with.

This capability leads to not only a very powerful and extensible automation framework, but one where the individual parts are very highly reusable and composable.

Automation Triggering

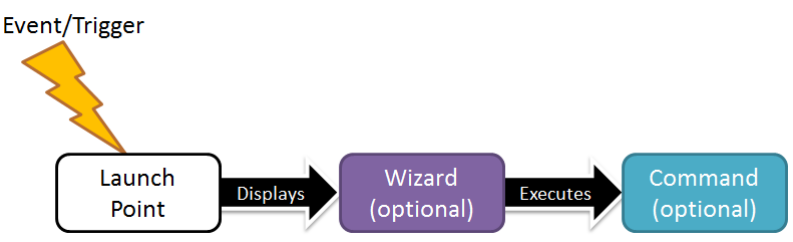
It is really the 'Launch Point' automation extension type that brings together all the other automation extensions to deliver the overall value of automation to a pattern toolkit.

A Launch Point coordinates a number of other automation extensions together to respond to some stimulus in the development environment, typically either a manual gesture by a user using a toolkit, or an event raised by another source such as Visual Studio.

For example, a 'Context Menu Launch Point' responds to a user right-clicking on a solution element in Solution Builder, by showing a pre-configured menu item. A 'Template Launch Point' responds to the user selecting a project or item template from the VS Add New Dialog, and unfolds that template into the solution. A 'Drag Drop Launch Point' responds to the user dragging and dropping data over solution elements in Solution Builder. And an 'Event Launch Point' responds to any other custom event the author configures.

The Automation Pattern

By convention of the Launch Point Automation Extension, all the Launch Points sub-types provided, are designed to both display an optional Wizard, and execute an optional Command. Both the Wizard and Command are in themselves just other automation extensions.



All Launch Points use design time and runtime configuration to setup themselves correctly for executing on their configured pattern model element.

Some of the Launch Points such as the 'Context Menu Launch Point' and 'Drag Drop Launch Point' also use Conditions to determine when they should respond to their respective events. This gives them the ability to 'filter' their events in order to contextualize the automation to run at only certain times, or only under certain conditions.

What are Launch Points?

A launch point responds to user gestures and triggers automation.

A Launch Point is typically configured to optionally show a 'Wizard', and optionally execute one or more 'Commands'.

The 'NuPattern Toolkit Builder' extension provides several types of launch points:

 **Template Launch Point**	configured to unfold an item or project template into the solution represented by an element in the pattern. This LP displays an optional wizard first to capture information from the user to contextualize the unfolded artifacts, and then runs an optional command to automate a post-unfolding process.
 **Event Launch Point**	configured to respond to any event in the development environment. When the event is raised, this LP displays an optional wizard first to capture information from the user to contextualize the command which automates a process.
 **Context Menu Launch Point**	configured to provide a context menu on instances of a specific element in the pattern model, that appear when the user right-clicks on a node in the Solution Builder. When the menu is active, and the user clicks on it, this LP displays an optional wizard first to capture information from the user to contextualize the command which automates a process.
 **Drag & Drop Launch Point**	configured to allow dragging and dropping of items from any source, to be dropped on elements in Solution Builder. When items are dropped, this LP displays an optional wizard first to capture information from the user to contextualize the command which typically automates the creation or configuration of elements in Solution Builder with items being dropped.

> **Note**: Custom launch points can be created that integrated with a Pattern Toolkit that respond to different gestures in the development environment.

In general, a Launch Point follows this lifecycle:

- **At design time**, a LP is configured by an author. The author typically configures parameters of the launch point that are specific to the gestures it responds to, and typically configures an optional Command, and optional Wizard and in some cases Conditions that configure the behavior of the LP.
- **At runtime**, the LP responds to user gestures, and given the right configured conditions, displays the optional wizard to gather information from a user, and then if the wizard is completed, executes the optional command to apply automation to the gesture.

> **Note:** Launch Points have their own configuration, which can be provided at design-time by the author, or can be evaluated at runtime using 'Value Providers'.

What are Events?

An Event is raised from any source in response to some activity, or user gesture. When these events are raised, listeners to these events are triggered, and automation can occur as a result.

Many of the launch points respond to events of one type or another. As an example, an 'Event Launch Point' responds to a configured event such as 'OnBeforeBuild' to perform automation before a solution is built, or 'OnElementInstantiated' to perform automation, such as unfolding a template, after a solution element is created. Other launch points such as the 'Drag Drop Launch Point' respond to the 'OnDrag' and 'OnDrop' events, to respond to a user dragging a dropping data in the Solution Builder.

Events are extensible, so as a toolkit author you can implement your own events from within or without the development IDE to trigger additional automation.

What are Commands?

 A Command is the means to execute any automation in a toolkit. Commands are executed by other triggers such as Launch Points in response to events or other user gestures. Commands can be as varied as: validating a model, generating a file, building a solution, importing some data, calling a web service, etc.

A command is configured by the author with configuration pertaining to its function. For example, a 'T4 Text Transformation' command must be configured with a text template file to transform, and a file name to generate.

Commands are developed to be the atomic unit of automation reuse for toolkits. That is, it is by combining commands with different triggers and different configuration that makes the basis for all the automation in a toolkit.

> **Note:** Commands have their own configuration, which can be provided at design-time by the author, or can be evaluated at runtime using 'Value Providers'.

What are Wizards?

 A Wizard is primarily used to gather data from a user before other automation executes using that data.

The provided Launch Points in the toolkit all allow an author to configure an optional 'Wizard' to run prior to executing their optional 'Commands'.

An author typically creates a wizard with the purpose to direct the user to configure a set of properties of a solution element together as a set of related configuration.

The wizard can wholesale validate that data before proceeding to apply the automation from the following commands.

Wizards can also provide richer user interfaces for collections of related data that would otherwise be harder to edit as individual properties.

> **Note:** Wizards can also be used to edit properties that are hidden form the user, so that they only can be edited together with other properties in a wizard.

What are Conditions?

Conditions are used to control or 'gate' what and when configured automation executes.

For example, an 'Event Launch Point' uses a collection of configured conditions to determine whether to handle a specific event for a specific solution element. A 'Drag Drop Launch Point' uses a condition to determine if dragged data is valid for the current element the data is dragged over. A 'Context Menu Launch Point' uses a collection of condition to determine when to show/enable a context menu.

In these kinds of ways conditions are used to 'contextualize' automation to ensure it only occurs at predefined times.

> **Note:** Conditions have their own configuration, which can be provided at design-time by the author, or can be evaluated at runtime using 'Value Providers'.

What are Value Providers?

A Value Provider is a means to provide a value to configuration of other automation dynamically, given a specific context and configuration from which it evaluates and returns a value.

Value Providers are typically used in the configuration of Commands, Conditions, Validation Rules, Launch Points etc. so that authors can fetch configuration values from the environment dynamically during the use of the toolkit.

For example, the 'ProjectRootNamespaceProvider' fetches the current value of the root namespace of a given Visual Studio project. That value will be retrieved from the project at the time the automation is executed sometime during the use of the toolkit.

> **Note:** Value Providers have their own configuration, which can be provided at design-time by the author, or can be evaluated at runtime using other Value Providers!

Value Providers are also used by 'Variable Properties' of elements in the pattern model to optionally provide their default values, and to make a calculated property.

What are Validation Rules?

A Validation Rule is a means to validate instances of solution elements and their properties when using a toolkit.

A Validation Rule is used typically to ensure that the solution elements in Solution Builder are correctly configured before executing any automation, such as code generation.

For example, the ‘PropertyRequiredValidationRule’ verifies that the configured property of the current element has a value (according to its data type).

Validation Rules are executed when solution elements are explicitly validated during toolkit use. Triggering of that validation is invoked by a validation command, which in turn is executed by a Launch Point.

Commonly, validation is triggered in any one, or more, of these cases:

- When the Solution is Built
- When a Solution Element Property is Changed
- With a Content Menu on the Root Element
- When another event is raised.

What are Artifact Links?

The term ‘Artifact link’ is commonly used to refer to a ‘Related Item’ of an element that resolves to a solution artifact displayed in the ‘Solution Explorer’ window.

These artifact links provide the ability to track solution items, and are typically used to navigate between the elements in ‘Solution Builder’ window and their associated artifacts in the ‘Solution Explorer’ window. Artifact links are resilient to changes in location in the solution and renaming, since in the development of any solution users constantly move and rename solution artifacts to suit their specific physical structural and naming conventions.

Publishing

What is Toolkit Info?

The Toolkit Info of a Pattern Toolkit is the information used to identify, version, and display information about the toolkit to its users. It is the metadata information used to describe the package within which the pattern toolkit is deployed and distributed.

A Pattern Toolkit is ultimately compiled into a Visual Studio Extension package called a 'VSIX', and the toolkit info is used to populate part of that VSIX information.

Visual Studio uses the information in the VSIX package to:

- Ensure the VSIX extension is not installed to an edition of Visual Studio it was not designed for.
- Ensure the dependencies of the VSIX extension are already installed.
- Provide the chance to read the license agreement before installation of the VSIX.
- Manage versioning and updates to installed VSIXes.
- Display important information about the VSIX to its users, in the Extension Manager in Visual Studio.

Every pattern toolkit must have valid and unique Toolkit Information before being compiled and deployed.

Deployment

How is a Pattern Toolkit Deployed?

Since a Pattern Toolkit is a custom development tool, it needs to be deployed into the development environments of the development team wishing to use it to create software.

Deployment of a Pattern Toolkit is straightforward VSIX deployment. That is, a Pattern Toolkit is compiled into a single VSIX file which is then distributed and installed to a development machine, by double clicking on the file. See VSIX Deployment for details on this mechanism, and Visual Studio Extension Deployment for more options on how to deploy a VSIX.

See Pre-requisites for Installation for details on using a deployed pattern toolkit.

Hands-On Labs

Hands on labs that walk you through various exercises to help you learn from completing guided examples.

Creating Pattern Toolkits

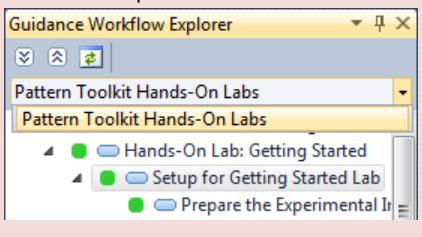
Two hands-on labs (HOL) are available that walk you through the experience of creating a pattern toolkit.

> **IMPORTANT**: You must install the 'NuPattern Toolkit Builder Labs VS2010' or 'NuPattern Toolkit Builder Labs VS2012' extension from the Visual Studio Gallery first before you can create the labs.

The first HOL 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 Pattern Toolkits rather than having to understand a specific pattern or set of tools you may want to implement.

The second HOL is more in-depth and covers a range of common scenarios, and features of toolkits that you may encounter when creating your own pattern toolkits.

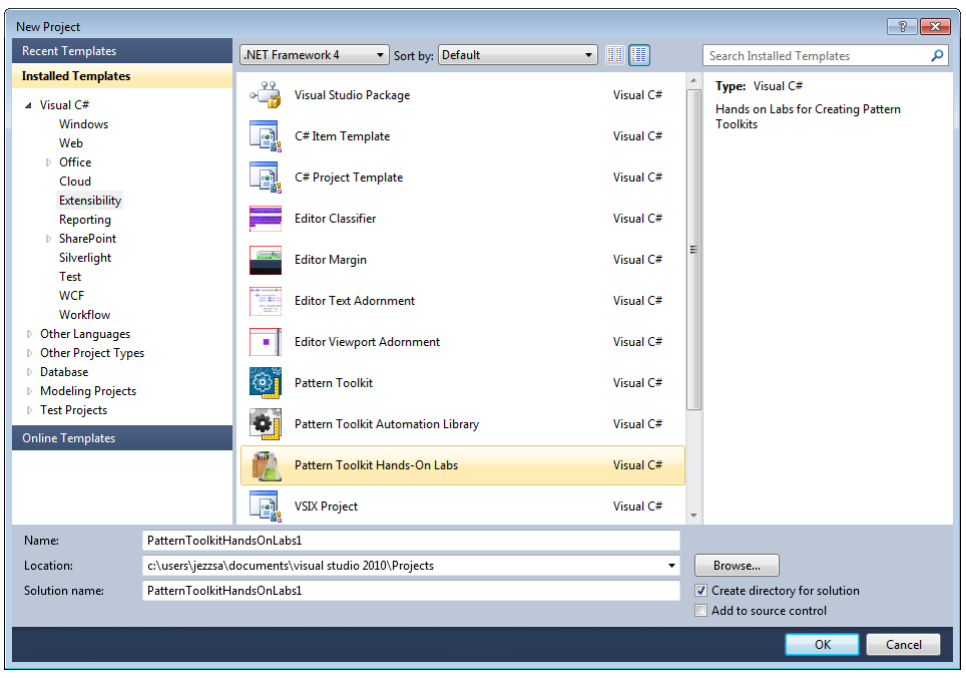
There are two ways you can get started with the labs:

> **Important**: When you switch to the hands-on-labs, the guidance that you are reading now will switch in the 'Guidance Workflow Explorer' window.
> 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.
>
> 
>

- If you want to start the Hands-On Labs within this Visual Studio solution, click this link (VS2010) or click this link (VS2012) which will add the labs project to your solution in the 'Solution Builder' window.

> **Note:** The link above will not work if you haven't already installed the 'Pattern Toolkit Builder Labs VS2010' or 'Pattern Toolkit Builder Labs VS2012' extension first.

- If you want to start with a clean Visual Studio solution (recommended for first-time users), then follow these steps
  - Close the current solution (which will close this guidance document, so you may want to make a note of these steps first).
  - Click on the File | Add | New Project menu, which will open the Add New Project dialog.
  - In the left pane of the New Project dialog, click on "Visual C#", then "Extensibility", and find the project type called "Pattern Toolkit Hands-On Labs".



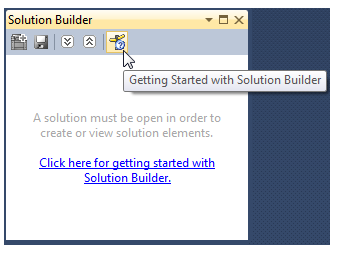
  - You can keep the default name (or provide your own) and then click OK.
  - Follow the lab instructions that appear in the 'Guidance Workflow Explorer' window.

How To: Guides

The ‘How To’ guides provide background information, tips and instructions for performing the most common activities with the toolkit.

Using

The guidance for ‘Using’ pattern toolkits is separate from this guidance, and is found from the guidance links in the ‘Solution Builder’ window.



> **Recommend:** Toolkit authors should make themselves familiar with the guidance that toolkit users are reading, and gain a clearer understanding of how the users are expecting to interact with toolkits in general.

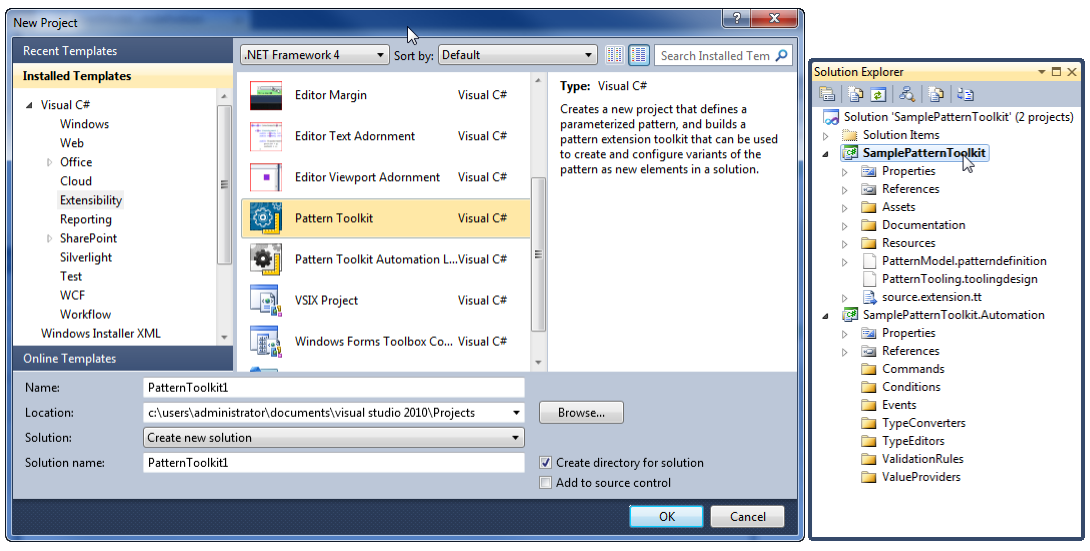
You can go straight to that guidance, by clicking this link, but note that you will need to change the selected guidance in the dropdown at the top of the ‘Guidance Workflow Explorer’ window, to return back here.

Authoring

Designing, creating and deploying Pattern Toolkits.

Understanding: What is a Toolkit Project?

A Pattern Toolkit project is a Visual Studio project template for creating a new or customizing an existing Pattern Toolkit.



The project provides all the necessary folders and files for creating a new Pattern Toolkit, such as:

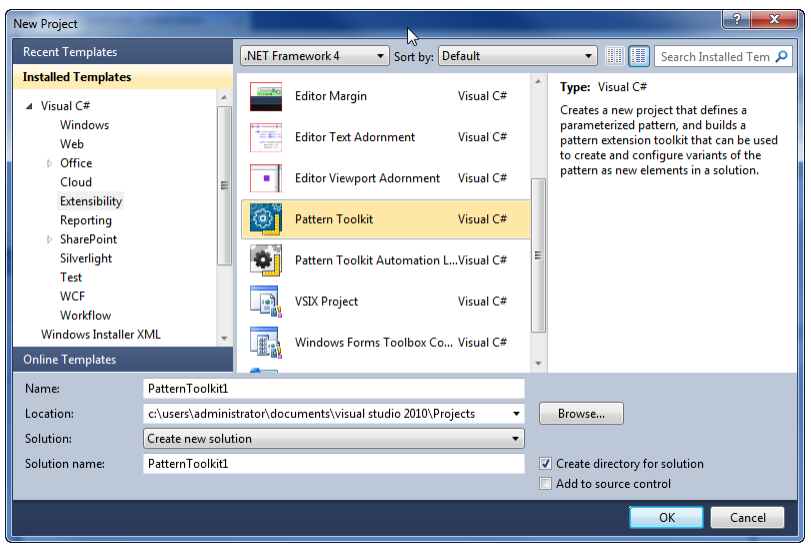
- Modeling the pattern with the Pattern Model Designer
- Managing reusable Assets
- Applying Guidance and Automation to the pattern.

See next topic on Creating a Toolkit Project

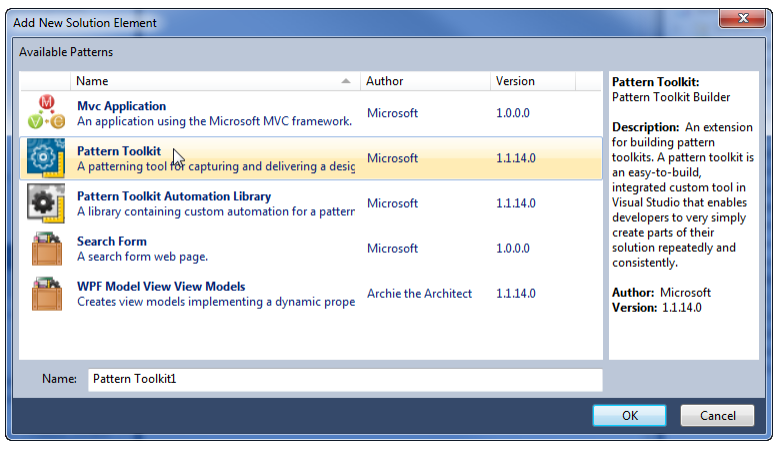
How To: Creating a new Toolkit Project

To create a new Pattern Toolkit project, either:

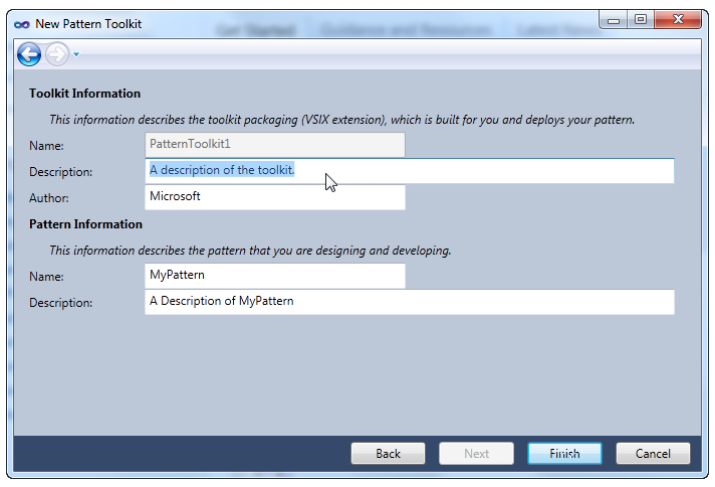
Add or create a new 'Pattern Toolkit' project from the Add/New Project/Item dialog.



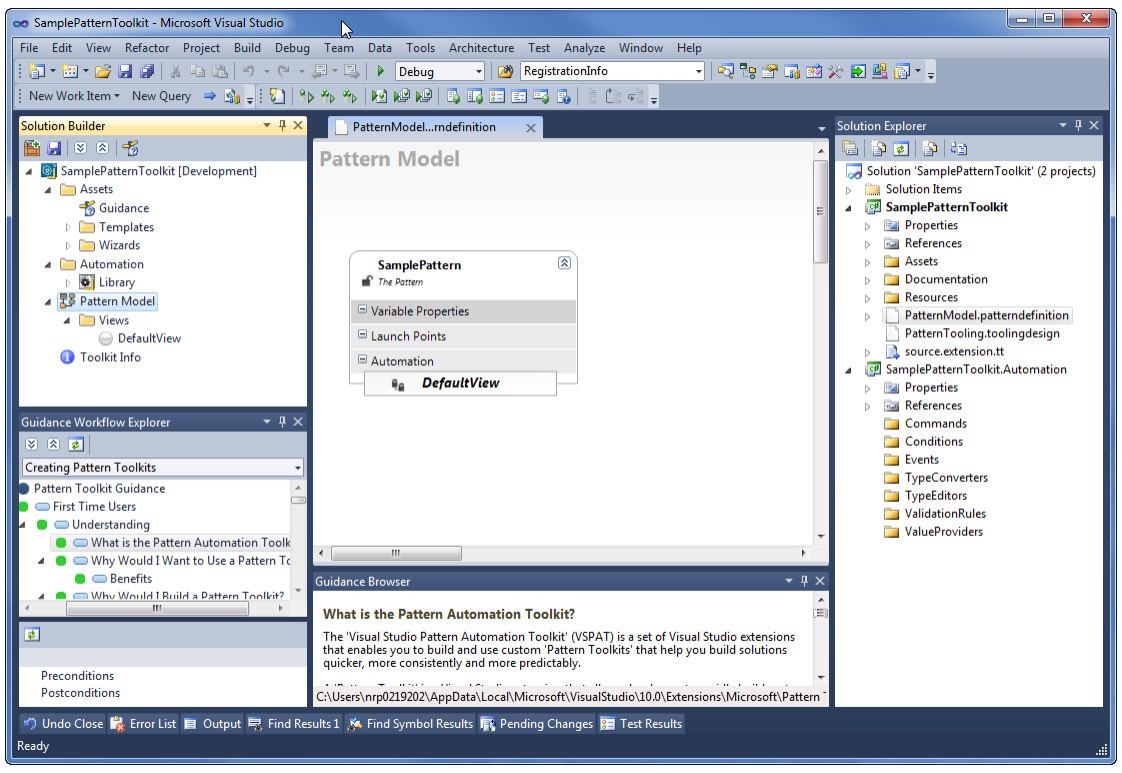
OR, create a new 'Pattern Toolkit' solution element in the 'Add New Solution Element' dialog, in the 'Solution Builder' window.



Then complete the creation wizard, selecting the 'Create a New Toolkit' option, and providing the information about the toolkit and pattern.



In order to create a pattern toolkit faster and more reliably, instead of working with the files directly in Solution Explorer, you can create the toolkit using the 'Solution Builder' tool window instead. And in this experience you benefit from both working at a higher level of abstraction of the individual source files by working with a model of a pattern toolkit in the 'Solution Builder' tool window, and your work is guided by information, tips, and instructions in the Guidance Explorer window.



How To: Start Building a Pattern Toolkit

Refer to Creating a Pattern Toolkit and The Development Process for an overview of the processes involved in building pattern toolkits.

Know What to Build

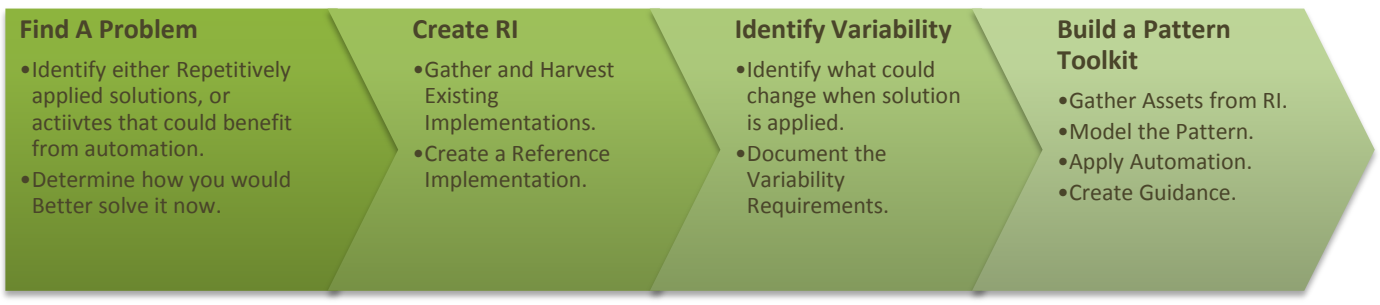
There are many ways and approaches to start thinking about building a toolkit, but before you start on building the toolkit, you need to have a clear goal in mind for what your pattern will be, and what you are going to use to implement it, the potential outcome and audience for it.

The following sets of activities have been proven to be very effective at identifying and developing a pattern toolkit from scratch. Like brewing a great beer, architecting, planning, designing and building a pattern toolkit requires some thought in: applicability, appreciation, success, validity and final implementation of the pattern within the toolkit. As such, the best patterns to capture in a toolkit are those processes and implementations that have already been proven successful and useful in existing software development solutions.

How to Approach It

Below are the steps we recommend to guide you to work towards and implement a successful toolkit.

> **Note:** There are many ways to approach this process, but starting first with just the tooling and skipping these steps is often fruitless.



> **Note:** Below is a straightforward and simplified description of a process that can be used for building any toolkit, and has been kept brief to identify the key steps, providing only the bare minimum guidance on how to accomplish them to keep this guidance concise.

- Find a Problem
- Create an RI
- Identify Variability
- Build The Toolkit

Find a Problem

Find a development problem that you solve over and over again, or one that is hard or tedious to solve manually.

Some examples:

- Applying an established or common architecture or proven pattern to a new software product
- Following guidelines for developing against a given set of technologies, libraries or frameworks
- Bringing a development team up to a productive speed on applying a new technology

Think about how you would like to have solved it, should you have to do it again.

Avoid boiling the ocean.

Since building pattern toolkits is very economical with NuPattern, you no longer need to focus solely on building toolkits that will have only the most widespread reuse. These kinds of toolkits may seem to offer the most return on investment, but that comes at a cost of expensive generalization, which reduces the opportunities of higher productivity in perhaps domains with perhaps less widespread reuse.

Instead, you can now afford to focus on smaller (perhaps less ambitious) and more specialized toolkits, that provide high productivity gains with high automation in smaller or more constrained areas. Armed with one or more of these smaller toolkits, you provide a dramatic impact on productivity for your team and foster a climate of consistency across whole software development projects.

Another attractive kind of toolkit to consider building is one that performs what would otherwise be a very labor intensive development task. Like: generating or mapping XML or configuration files, or setting up a coding project with structure, shared includes, code analysis policies and naming conventions. The value of these kinds of toolkits is that it can dramatically cut the manual development cost and manual errors that these kinds of necessary development activities typically generate.

Essentially, the value of building a pattern toolkit for these kinds of tools, is that there is already a tooling framework upon which this kind of tool easily be built, and run and interact with the user. All you need to do is figure out what to automate, and you have a powerful set of visual tools to present it to the user for use in their solution.

Next, Create an RI

Create an RI

Once you have an idea of what kind of pattern or toolkit to develop, the next step is to gather existing implementations, development assets, tools, etc.

This critical step will be iterative and ongoing as the toolkit is developed, but it's important to get a good certain start in this area, otherwise the risk is that the pattern implementation will be constantly in flux.

Gather: existing implementations, requirements, code samples, libraries, frameworks, design notes, technical information, research best practices and recommended patterns etc.

Harvest and Harden: refactor, simplify, decouple architectures and implementations, and create reusable assets that can be built upon with a solution.

Given the 'notional' idea of the pattern you want, ensure you have the ability to provide/develop an implementation of it in working software. And verify that implementation would add value to a development team.

> **Note:** This step invariably validates and confirms your ability to (a) derive a pattern and (b) to reproduce an implementation of it consistently and repeatably given different requirements for it.

Build reference implementations that compile, execute and can be tested, with a set of given assumptions of how they can be configured.

It is recommended that you have some initial end-to-end integration tests to verify the RI is working with its assumed configuration.

Next, Identify Variability

Identify Variability

Given a more firm idea of the developing pattern, which is now represented in some form in the reference implementation, the next task is to identify how that implementation can be applied to different solutions, and whether it would require some change in the implementation to accommodate different applications of the implementation. These differences are the variability points of the pattern. Or in other words, the parameters that control the implementation, when applied in different solutions.

In most cases, the variability points will be apparent, because very they were fixed in some form in the creation of the reference implementation. New variability points typically arise from the need to apply the implementation to other solutions.

Document these variability points, with suggestions on how they can be presented to the user configuring them. The idea is to create and maintain a list of variability requirements for the pattern that will be constantly re-prioritized as people apply the pattern in more than one use.

> **Tip:** Initially, keep the list of variability requirements to a minimum. Don't conjure up or brainstorm all possible variability requirements, as this leads to designing a highly generalized toolkit, reducing its value, and increasing the cost of maintaining it as a reusable asset to your organization - ultimately reducing it worthiness. Instead focus incrementally only on those variability points that are required by existing users of the toolkit. Every pattern that is applied by a toolkit is expected to require some custom coding of the implemented pattern. Weigh the reuse, cost and complexity of implementing the variability requirement by the toolkit against the cost of one-off custom coding the requirement against the implementation provided by the toolkit. If the cost, frequency or complexity of custom coding the requirement is repeatedly high, consider it for standardization and automation (either partially or fully) in the toolkit.

The final goal of this stage is to refactor out all that is not variable into reusable packaged assets, such as: templates, libraries, frameworks etc. These assets then become the baseline set of assets (or framework if you will) upon which the variability points are applied by the toolkit.

Next, Build the Toolkit

Build the Toolkit

With all the activities above completed to some degree, it's time to capture the pattern, and start to develop the toolkit.

> **Note**: Building the Toolkit is an inherently iterative process, as variability requirements are implemented and new ones are discovered or refined.

Start by modeling the pattern's implementation, teasing out the variability and representing that in elements and properties of those elements.

> **Tip**: Don't model each and every aspect of the pattern for academic accuracy and completeness. Represent the elements of the pattern in the language of the user using the pattern. This keeps the representations in the pattern understandable and concrete. Think about the audience for the toolkit and what they want to configure. On the flip side, don't represent every technical detail in the model either. This makes configuration of the pattern difficult to use.

Next, start to reason about how automation can help with both the configuration of the pattern and how the implementation can be derived from it.

Begin developing generative strategies such as: project templates, and code generation, to get the fixed assets and coding artifacts into the solution.

Develop guidance that helps the users using your pattern understand its use, and how to use the toolkit to apply it.

Iterate, and loop back with the variability requirements and evolution of the reference implementation as variability is uncovered, refactoring the assets of the RI, and keep the iterations short and focused on delivering value to the end-solution.

Develop integration or end-to-end tests that verify the implementation of the pattern.

> **Note**: It is not necessary to deliver tests that verify if the pattern implementation actually works; those kinds of tests should be applied in the reference implementation and not delivered to the users using the toolkit. These tests will always be just assumed to work because the integrity of the pattern implementation is assumed to be good for reuse. However, it is often very useful to provide the users with end-to-end tests and/or testing harnesses generated from the toolkit that help verifying their specific variation of the pattern with their solution.

How To: Toolkit Design Guidelines

Following are a few general design tips and guidelines for authoring Pattern Toolkits that maximize their usability, effectiveness and the use of in-built features intended to help users be most productive with applying your pattern with them.

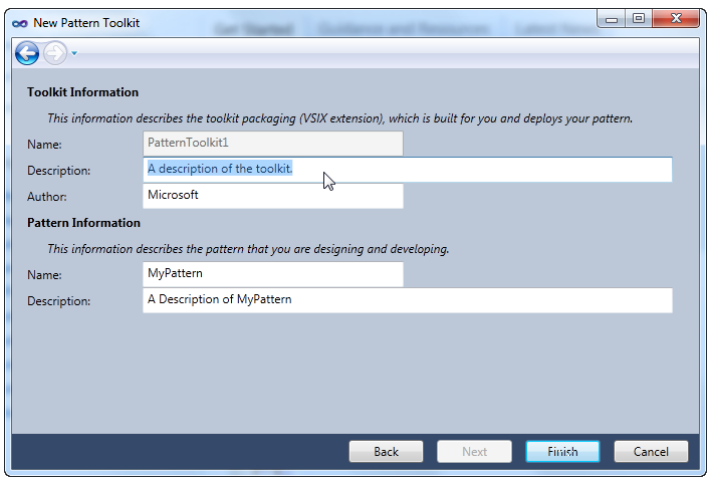
- Modeling Guidelines
- Automation Guidelines

How To: Naming in a Toolkit Project

This is a guide to the naming conventions of the various artifacts in a pattern toolkit.

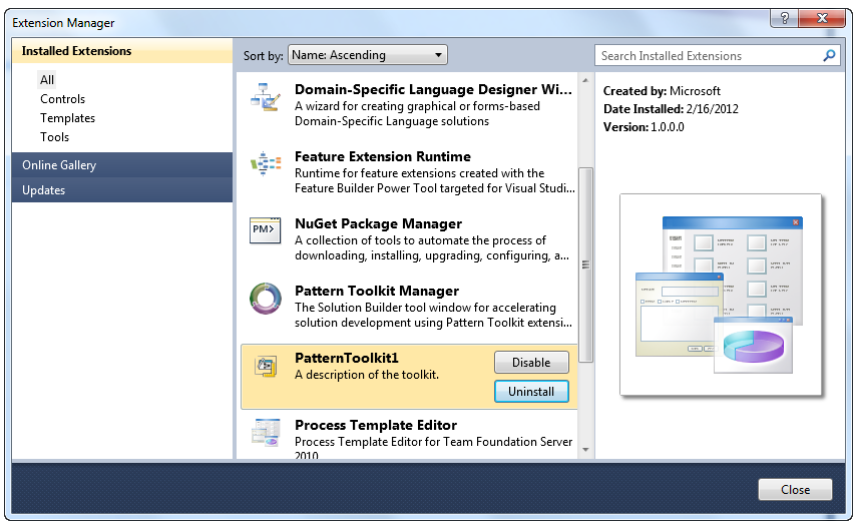
Naming the Toolkit Project

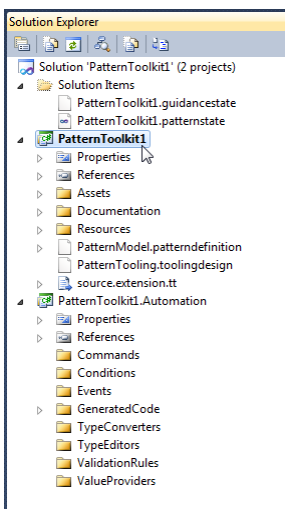
When creating a toolkit project you are asked to provide the name and description of the toolkit, as well as the name and description of the pattern (see later).

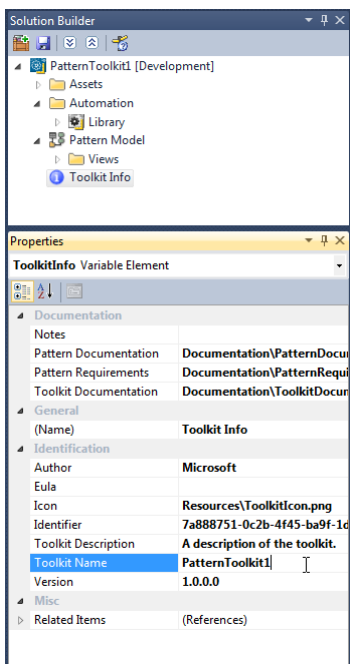


> **Note**: The name of the toolkit is already determined by the name you gave in the previous dialog box and cannot be changed here. Cancel this wizard and try again to alter that name. Or change that name again once the project is created.

The name and description of the toolkit should reflect the name and description you want people to see about your toolkit when installing it in the Visual Studio Extension Manager. And essentially should reflect the marketing message of the toolkit you produce. This will rarely duplicate the name and description of the pattern contained within the toolkit.







The toolkit name and description are initially used to name the projects of the toolkit solution, and the name of the toolkit in Solution Builder. These project names can be changed independently later at any time, to suit your organizations project naming conventions

> **Note**: Changing the name of the toolkit project in Solution Builder will rename the project in Solution Explorer.

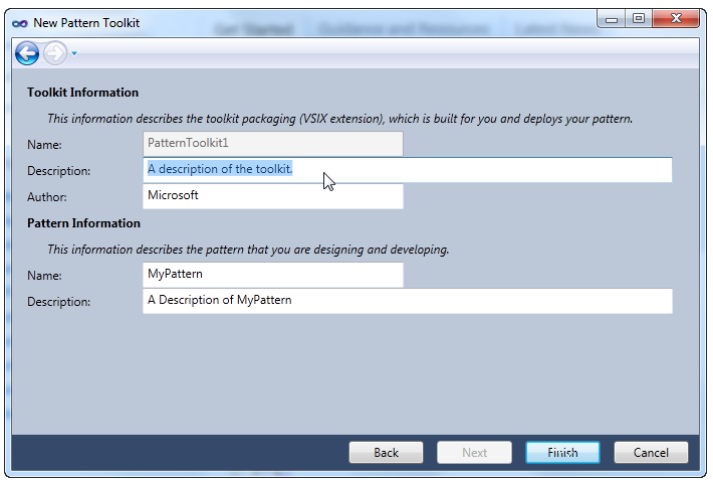
You can view and modify the name and description of the toolkit at any time in the 'Toolkit Info' element of the toolkit project.

> **Note**: Changing the 'Toolkit Name' and 'Toolkit Description' here does not change the names of the projects in the solution.

Naming the Pattern

The name and description of the pattern is the name the user will identify the pattern with.

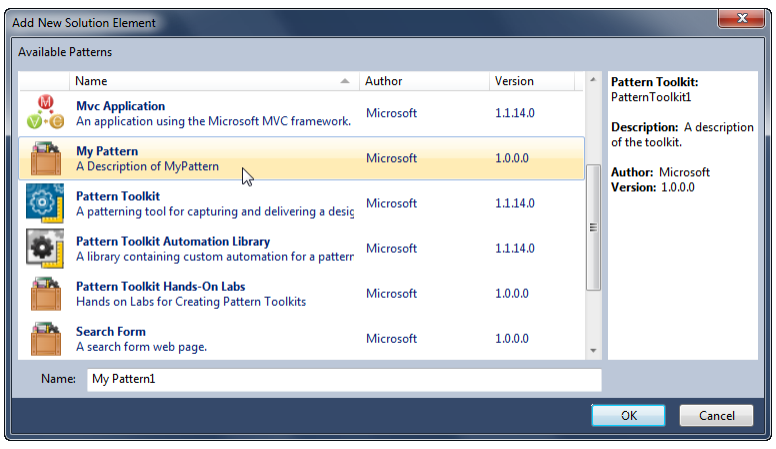
> **Note**: The toolkit name is generally not at displayed to the user, but the pattern name is.



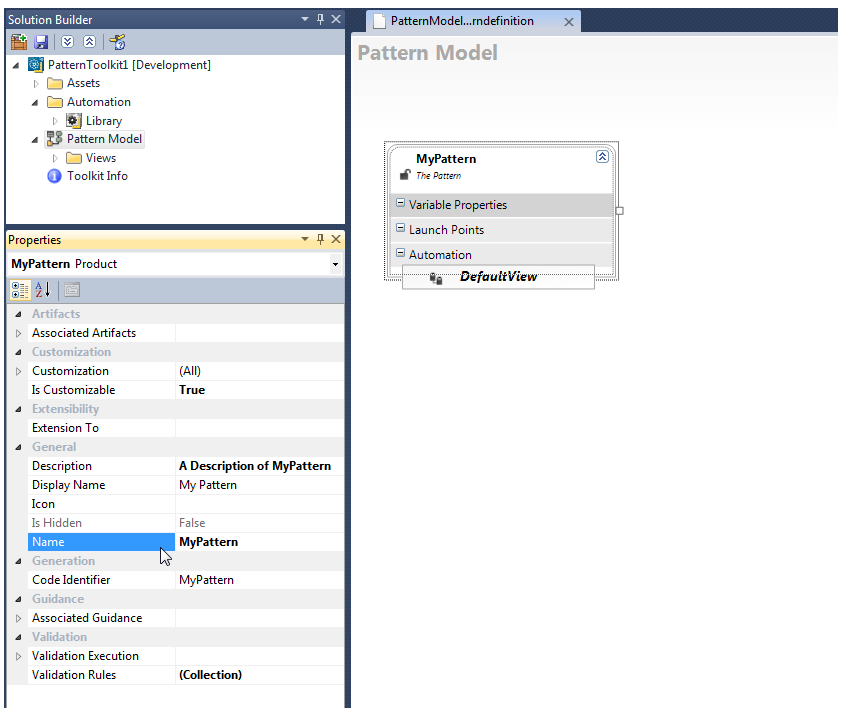
The pattern name and description should reflect what it is that the user is going to create in their solution.

> **Note**: The pattern name should never contain the word or terms “toolkit”, as the user will never create an actual toolkit. They create what is packaged within the toolkit.

This name is displayed to the user in the [‘Add New Solution Element’](#) dialog, from which they select the pattern to use.



> **Note**: When a user creates a new instance of the pattern, this name is used as the default name of the solution element created.



The pattern name and description are initially used to name the root ‘Pattern’ shape of the [Pattern Model](#). This name can be changed independently later at any time

> **Note**: The actual displayed name to a user is in fact the value of the ‘Display Name’ property which by default is the same as the value of the ‘Name’ property, with spaces inserted before any capitalized letters in the ‘Name’.

You can view and modify the name and description of the pattern at any time in the properties of the root ‘Pattern’ Shape.

Naming Automation

When creating and configuring Launch Points, Commands and Wizards on elements in the pattern model, you should use the following naming conventions to ensure the readability of your pattern model.

Configuring Launch Points

(e.g. Template Launch Point, Menu Launch Point, Event Launch Point and Drag Drop Launch Point)

- Start the name with the action being performed. e.g. 'UnfoldProject' or 'GenerateClass' or 'ConfigureElement' etc.
- Complete the name with the name of the event triggering the launch point. e.g. 'UnfoldProjectOnInstantiation', 'GenerateClassOnBuild', 'ConfigureElementOnMenu' etc.

Configuring Commands

- Describe in the name the action being performed. e.g. 'UnfoldProject' or 'GenerateClass' or 'ConfigureElement' etc.

> **Note**: Do not encode in the name of the command the launch point in which it is used, as commands can be later reused by several launch points.

Configuring Wizards

- Describe the action being performed by the wizard as a whole, remembering that the element upon which it is going to be configured is usually very specific. e.g. 'ConfigureUserInformation'.

How To: Run, Debug, Test a Toolkit

Running, and Manually Testing a Toolkit

You build, run and manually test your pattern toolkit in the same way you would any .NET application with Visual Studio. Press CTRL + F5 (Start without Debugging).

> **Note**: You must select the toolkit project as the 'Startup Project' first in the solution, in order to run.

Your toolkit will be loaded in the 'Experimental Instance of Visual Studio', you will see your toolkit already loaded in Extension Manager, and you will then be able to create new instances of its pattern in the 'Solution Builder' window.

> **Note**: In order to create instances of your pattern, you must have an existing solution open, or create a new one.
> An exception to this rule is when your pattern has a 'Template Launch Point' configured on it with a project template that appears in the Add New Project/Item dialog box. You can then use that project template to create an instance of your pattern, as a solution is automatically created for you.

Once a test solution is created in the 'Experimental Instance', use the 'Solution Builder' window to create a new instance of your pattern, and then manually test it.

> **WARNING**: For many of the incremental changes you can make to your pattern in the pattern toolkit project, you can simply manually test them with an existing instance of your pattern already created in an existing solution in the 'Experimental Instance'. However, there are several kinds of changes that you can make in the pattern toolkit which will not be reflected in previously created instances of your pattern. For these kinds of changes you will need to create brand new instances of your pattern in the solution, and delete the older instances. If in doubt, always create a new test instance of your pattern.

Debugging a Toolkit

> **Note**: For the most part, it is unnecessary to launch the debugger to manually test your toolkit, unless you are troubleshooting your automation classes; otherwise the debugger just adds a performance cost to the whole process.

See Troubleshooting Toolkit Problems for more details on how to debug and diagnose your automation classes.

Automated Toolkit Testing

You can build and run automated tests on your toolkit in the same way as you can with any .NET application.

Unit testing/Integration testing of all your automation classes are very highly recommended.

As for automated testing of the toolkits' UI in 'Solution Builder', any UI test framework (such as Coded UI Tests) suitable for testing the user interface in Visual Studio are also recommended.

How To: Troubleshoot Toolkit Problems

There are two main mechanisms for troubleshooting toolkits for authors. You can debug the code of your toolkit using a debugger, or diagnose trace information provided by your toolkit and the toolkit execution environment.

> **Note**: When using toolkits from other authors, you will be unable to debug their code, and will only be able to diagnose using tracing (see below).

Debugging Toolkits

To debug your pattern toolkit in the same way you debug any application with Visual Studio, simply build your toolkit project and run in debug mode.

Your toolkit will be loaded in the 'Experimental Instance of Visual Studio', and you will be able to set breakpoints in the custom automation classes your toolkit uses.

> **Note**: Being that pattern toolkits execute upon an automation framework that you have configured in your toolkit, there is very little code in your toolkit project to actually debug. The only code which you can debug is custom code in your automation classes (i.e. Commands, Conditions, Value Providers, Validation Rules etc.), or code called from those automation classes.
> The code in the automation framework makes extensive use of tracing for debugging and diagnostic purposes.

Diagnosing Toolkits

All toolkits, including the automation framework upon which toolkits execute, uses diagnostic tracing as the primary mechanism for identifying and troubleshooting. See Troubleshooting Toolkits for more details on using trace information to diagnose issues with your toolkit and others you work with.

You are strongly encouraged to participate in this tracing mechanism when building your own pattern toolkits. See Applying Tracing to Automation Classes for more details.

Design

The concept of designing your tooling before implementing your tooling

How To: Create a Production Tooling Workflow

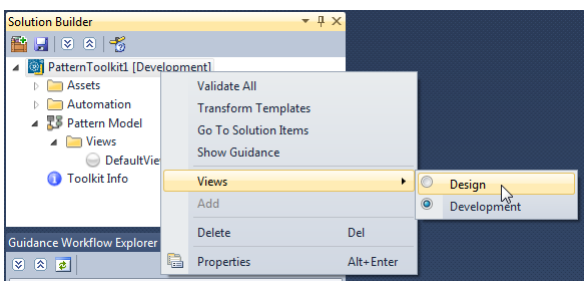
You can create 'Production Tooling Workflow' design for your toolkit which aids you in developing and reasoning about how the assets you have harvested for your pattern interact with proposed automation 'Tools' to complete the pattern in your toolkit.

> **Note**: This diagram's primary purpose is to be a free-form design surface to aid understanding, and has no impact on the pattern model design in the 'Development' view.

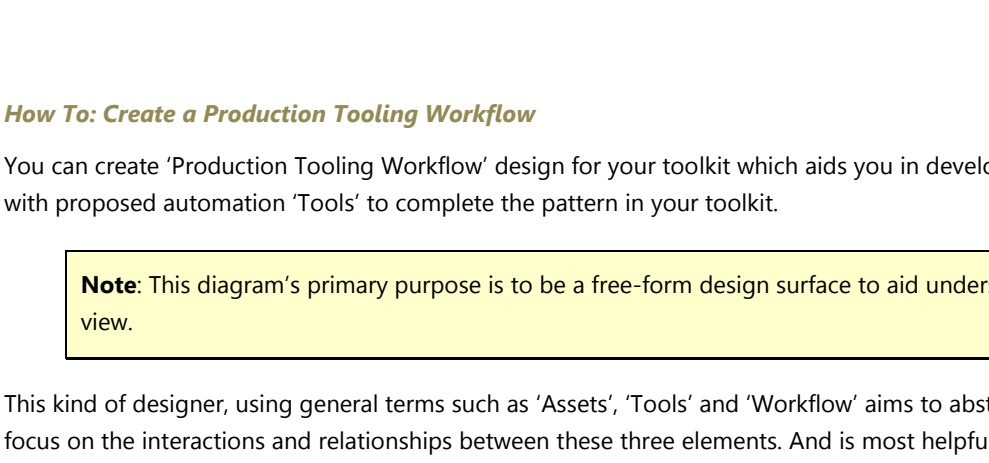
This kind of designer, using general terms such as 'Assets', 'Tools' and 'Workflow' aims to abstract the details of physical assets and automation away, allowing the toolkit author to focus on the interactions and relationships between these three elements. And is most helpful when trying to understand how automation can play a part in increasing the productivity of using your toolkit.

Create the Production Tooling

In Solution Builder, change the 'View' of your pattern toolkit from 'Development' to 'Design'

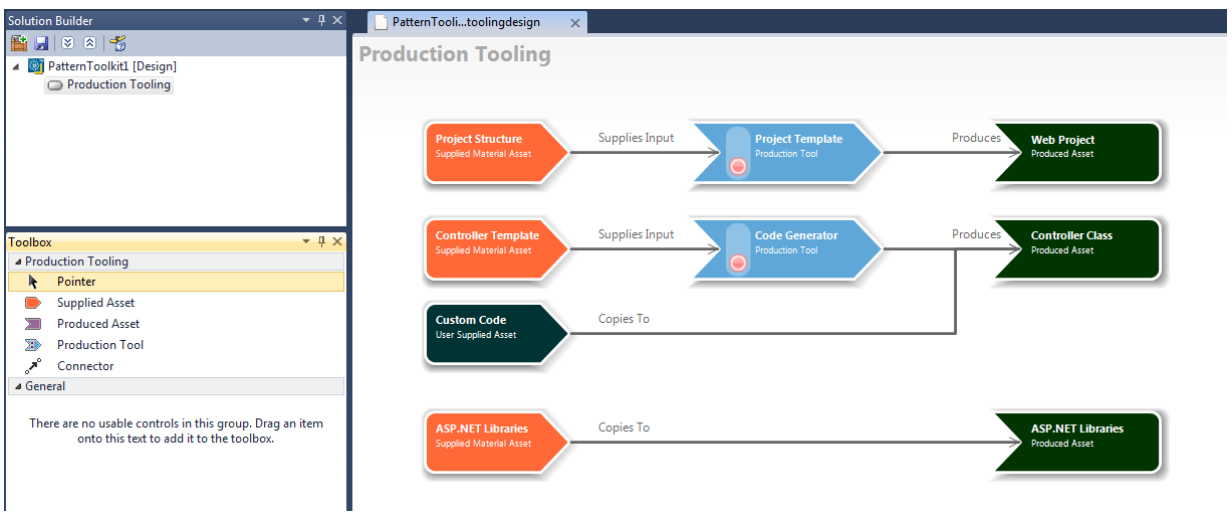


Open the 'Production Tooling' designer.



Now drag and drop elements from the toolbox, and configure their properties to describe and document the proposed assets and automation of the toolkit.

For example:



Modeling

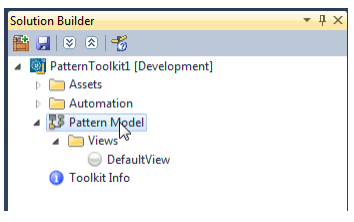
Understanding: What is a Pattern Model?

See What is a Pattern Model for details.

How To: Creating a Pattern Model

Every pattern toolkit has one instance of a Pattern Model, in the 'PatternModel.patterndefinition' file.

In your toolkit project, from the 'Solution Builder' window, open the 'Pattern Model' element.



> **Note**: Either double-click on this element or right-click and choose 'Open Solution Items'.

How To: Modeling Design Guidelines

- Don't include in the name of your pattern with the word 'toolkit'. Your pattern toolkit is simply the package that deploys the pattern. A user creates an instance of your pattern, not an instance of your toolkit.
- Define aspects or concepts that are easily understood in the 'vocabulary' of the pattern being represented and configured.
- Avoid representing aspects or concepts in your pattern that have no variability, or no structural or referential value. This can lead to over complex models which are confusing and complex to configure for a toolkit user.
- Use a 'Collection' shape when you have an 'Element' shape with a Cardinality of more than 'One', or when you have to group multiple different 'Element' shapes that are related together.

> **Note**: A 'Collection' is no different from an 'Element' in any way, except that is represented with a Folder icon by default if no other icon is specified.

- Don't pluralize the names of 'Collection' or 'Element' shapes.
- Try to keep 'Pattern', 'Collection' and 'Element' shapes collapsed (especially when they don't have any 'Variable Properties' or 'Automation' defined) to save real estate on the pattern model diagram.
- Ensure the values of the 'Display Name' and 'Description' properties of 'Pattern' 'Element', 'Collection' and 'Variable Property' shapes is meaningful and up to date. These values are presented to pattern users in various different places, and provide important cues for configuring correctly. See Controlling the Display of the Pattern for more details for more details on where these names and descriptions are presented to users.
- Use the 'Icon' property on 'Pattern', 'Collection' or 'Element' or 'Extension Point' shapes to provide a unique visual distinction for a user.
- Set the 'Auto Create' property to true, to automatically create the first instance of an element or collection, to save users of the pattern having to manually create it. See Controlling Instancing.
- Use Validation, and Validation Rules on 'Pattern', 'Collection', and 'Element' shapes for 'Many' kinds of 'Cardinality' to ensure the user configures the correct number of instances. See Applying Validation for more details.

> **Note**: There is no need to explicitly validate 'One' kinds of 'Cardinality' (i.e. OneToOne and OneToMany), these are validated automatically at all times.

- Use 'Extension Point' shapes when you want to make your pattern to plug-into or integrate with others, or to have other pattern toolkits provide extensions to your pattern that have variable implementation. See Enabling Composition and Pattern Reuse for more details about when and why to use extension points.
- For 'Pattern', 'Collection' or 'Element' shapes that will be associated to artifacts in the solution (i.e. unfold templates or generate code in Automation), set the 'Associated Artifacts' properties, so that users can navigate and open those artifacts from Solution Builder. Enabling Solution Navigation for more information on this capability.
- Always provide default values for all properties used by any automation. See Setting a Default Value for more details.
- Hide property values that are not to be seen or manipulated by a toolkit user. Typically, these properties will only be used exclusively by automation. See Controlling the Display of the Pattern for more details.
- Make all properties 'read only' that are to be seen but not changed by a toolkit user. Invariably, these properties are used exclusively by automation, but suitable for the user to reference. See Controlling the Display of the Pattern for more details.
- Use Validation Rules on 'Variable Properties' to ensure property values are always valid for state of the pattern. See Applying Validation for more details.

Understanding: Model Variability

See Commonality and Variability on the concepts expanded here.

The variability of your pattern is defined in the Pattern, Views, Collections, Elements, and Variable Properties, and the relationships of those.

The Pattern, Views, Elements and Collections (and their relationships to one another) give the pattern a logical structure and vocabulary, and the Properties they have provide the configured state of the pattern to the user.

See the following topics for more information about how variability is represented in a pattern toolkit:

- Variability in Views, and Displaying Aspects with Views for creating new Views.
- Variability in Collections, Elements and Properties, and Controlling Instancing for defining relationships between elements.
- Variability in Extension Points, and Enabling Composition and Pattern Reuse for creating Extension Points.

Understanding: Enabling Composition and Pattern Reuse

See What are Extension Points for the concepts behind using them in your pattern for:

1. Declaring for Extensibility
2. Implementing for Composability
3. Integrate for Reusability

The key benefit of 'Extension Points' is making pattern toolkits composable, and therefore much more reusable.

How To: Declaring Extensions

> **Recommend**: See What are Extension Points for the concepts behind using 'Extension Points' in your pattern for:

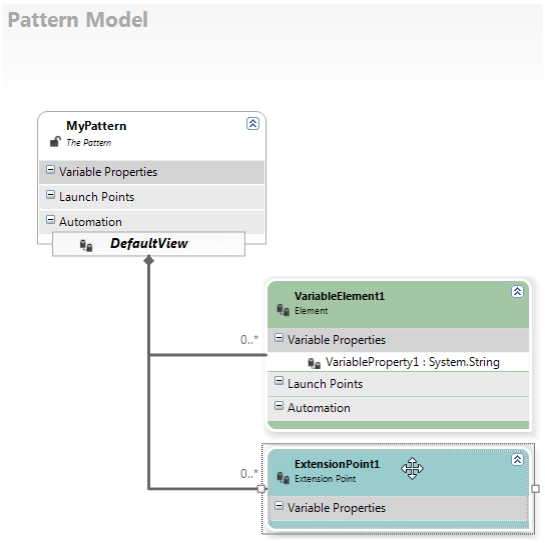
This topic describes how to create extensions to your pattern, so that other patterns can extend you pattern. This is a strategy to decompose pattern toolkits into smaller more reusable toolkits.

Creating the Extension Point

To declare an 'Extension Point' in your pattern, drag and drop an 'Extension Point' shape from the toolbox onto your pattern model diagram. See Adding Elements for more details.

You must name the extension point, and you can define the 'Cardinality' of the extension.

> **Note**: Because it is not known at design-time which pattern (from another toolkit) will plug into instances of this pattern, you cannot 'Auto-Create' instances of other patterns. However, you can create custom commands that

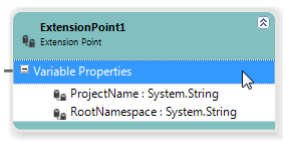


Defining a Data Contract

You can add 'Variable Properties' to the 'Extension Point' to define data that is 'required' by those pattern toolkits 'providing' the extension.

These properties are often required by automation, to integrate the implementation provided by another toolkit into this toolkit.

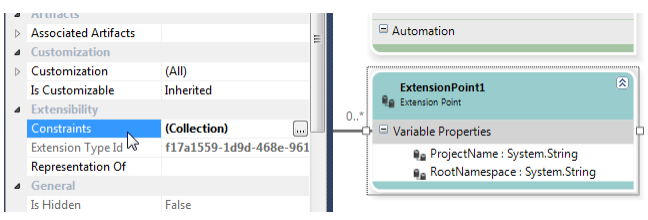
> **Note**: The variable properties defined on an extension point, conceptually, define a 'data contract' between the 'Requirer' and the 'Provider'.



> **Note**: You cannot define any Automation for an 'Extension Point'.

Constraining Extensions

You can constrain the patterns that can be composed by a user on this extension by defining one or more 'Constraints' of the extension point.



These constraints are simply 'Conditions' that determine if the providing pattern is permitted to extend this pattern.

For example, by applying conditions that check the providing pattern's toolkit information, you could constrain that only pattern toolkits created by certain authors, or only certain versions, can be composed into your pattern.

> **Note**: These constraints are only applied when users compose patterns together in the 'Solution Builder' window.

How To: Implementing Extensions

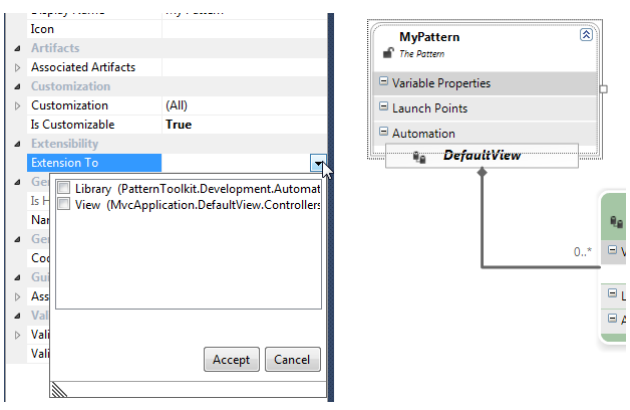
> **Recommend**: See What are Extension Points for the concepts behind using 'Extension Points' in your pattern for:

This topic describes how to implement existing extension points (from other toolkits) so that your pattern can extend other patterns. This is a strategy for providing patterns that integrate with other patterns.

Implementing an Extension Point

To implement an 'Extension Point' defined in another pattern toolkit, you configure the 'Extension To' property on your 'Pattern' element in the pattern model.

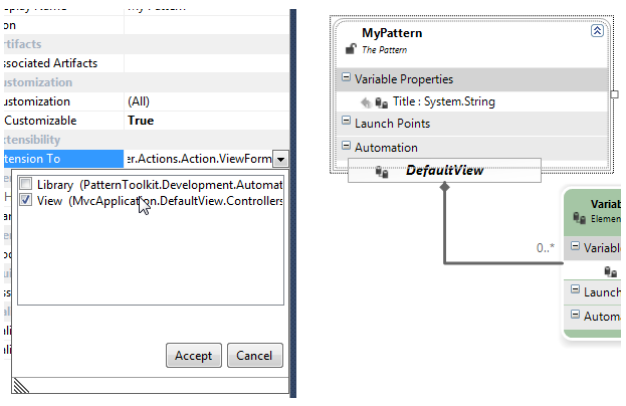
This property defines a list of all the 'Extension Points' you can implement from patterns that are currently installed.



> **Note**: The pattern toolkit that defines the extension point must be installed in Visual Studio first, in order to appear in this list.

You can reuse (or provide) one or more of these 'Extension Points', and when you select the extension point from the list.

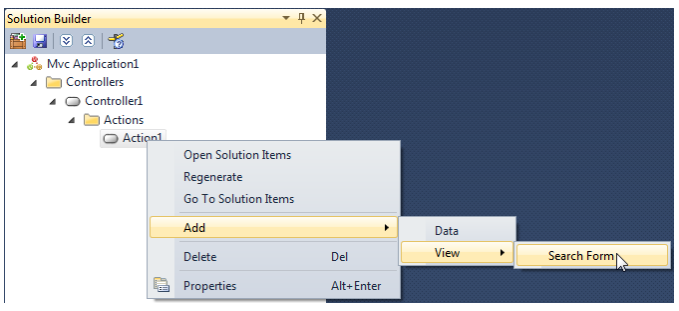
The properties defined on the extension (data contract) are automatically added to your pattern element.



> **Note**: These 'Implemented' properties cannot be removed or renamed, but their appearance, and values can be customized for use in your pattern.

Once an extension point is implemented, your pattern now becomes available as a new element in the implemented pattern. Users can start creating instances of your pattern from the implemented pattern, and automation from the implemented pattern can start working with the data in the data contract provided by your pattern. To the user this extension mechanism is completely opaque.

For example, the 'Search Form Toolkit' implements the 'View' extension point of the 'MVC Application Toolkit', so that when the user comes to create a 'View' they can choose from one of the toolkits that implement this extension point.



How To: Reusing Extensions

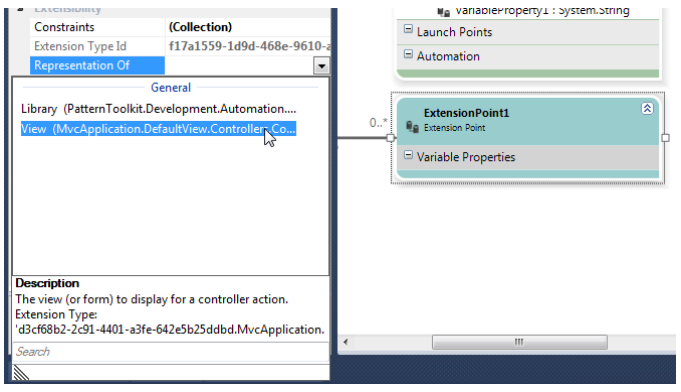
> **Recommend**: See What are Extension Points for the concepts behind using 'Extension Points' in your pattern for:

This topic describes how to reuse an existing extension point (defined by another pattern) so that your pattern can integrate with other patterns that already extend the extension point. This is a strategy for reusing existing patterns in your pattern model.

Reusing an Extension Point

To reuse an existing 'Extension Point' defined by another pattern toolkit, you configure the 'Representation Of' property of an 'Extension Point' shape in your pattern model.

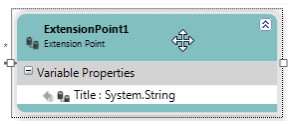
This property defines a list of all the 'Extension Points' you can reuse, from patterns that are currently installed.



> **Note**: The pattern toolkit that defines the extension point must be installed in Visual Studio first, in order to appear in this list.

You can only reuse (or become) one of these 'Extension Points', and when you select the extension point from the list.

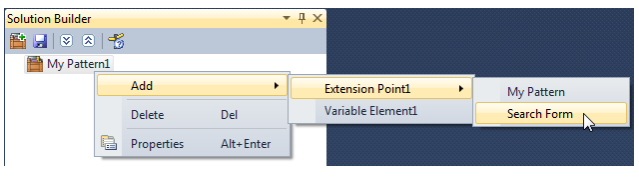
The properties defined on the extension (data contract) are automatically added to your extension point to represent the original extension point.



> **Note**: These 'Implemented' properties cannot be removed or renamed, but their appearance, and values can be customized for use in your pattern.
> **Note:** Any existing 'custom' properties already defined on your extension will be removed.
> **Note:** You cannot add any other new properties to the extension point.

Once an extension point is reused, your pattern now allows users to start creating instances of other patterns integrated with your pattern. To the user this extension mechanism is completely opaque.

For example, following the examples above, a user can now see that this pattern can add 'Views', and they can add any view from any toolkit (that implements a 'ViewForm') that is installed.



> **Note**: In this specific example, displayed above, this particular example toolkit both implements the 'ViewForm' extension point AND reuses the 'ViewForm' extension point, which is why the user see two possible 'View' ('Extension Point1') types on offer ('My Pattern' and 'Search Form') to add to their pattern. The former coming from this toolkit, the later coming from the 'Search Form' toolkit.

How To: Displaying Aspects with Views

See Variability Views for the concepts behind creating views for distinguishing aspects of you pattern.

Create new views for displaying distinct aspects of your pattern.

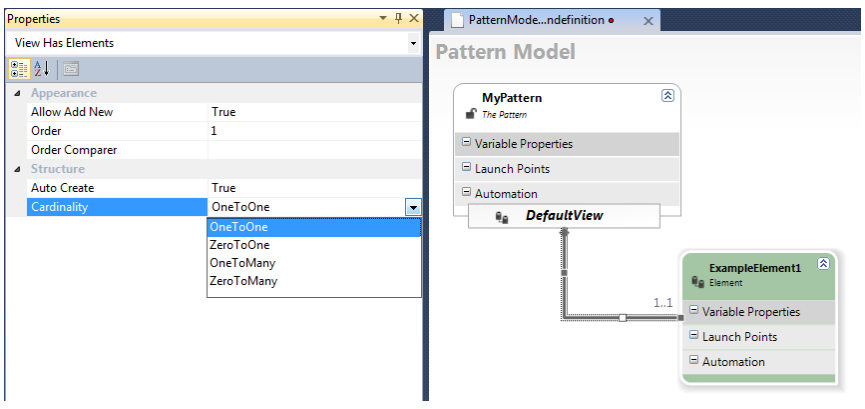
Add a new view to your pattern model, and then configure the view.

> **Note**: 'Collection', 'Element' and 'Extension Point' shapes in each view must be distinct, and have unique names. There is no sharing of elements between views, but the Pattern element and it properties will be present in all views.

How To: Controlling Instancing of Elements

See Collection, Elements and Properties for the concepts behind relationships and cardinality.

You control instancing of ‘Collections’, ‘Elements’ and ‘Extension Points’ by changing the Cardinality property on the connection between them.

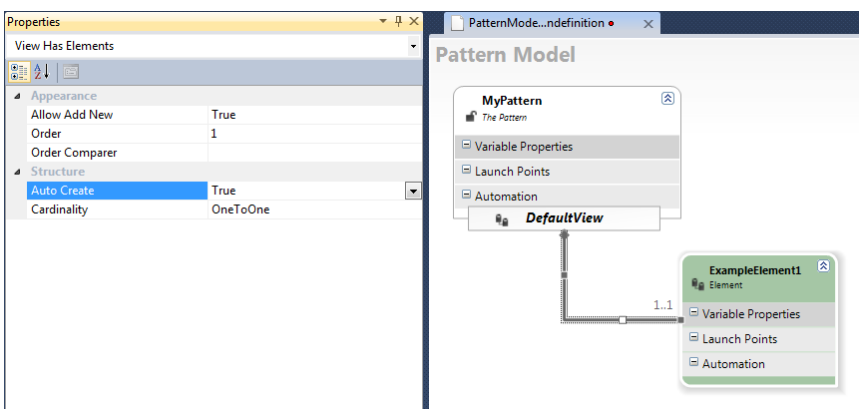


The ‘Cardinality’ property controls how many instances of a child element can be created. However, the creation of those element instances is either performed manually by the user by right-clicking in the parent element and using the ‘New’ menu to create an instance of the child element, or created by automation in the toolkit.

Auto-Creation of Instances

As a convenience, the first instance of any child element can be created automatically by using the ‘Auto Create’ property.

When ‘Auto Create’ is set to ‘true’ the first child instance of the element is created automatically using default values of its properties with the default name.

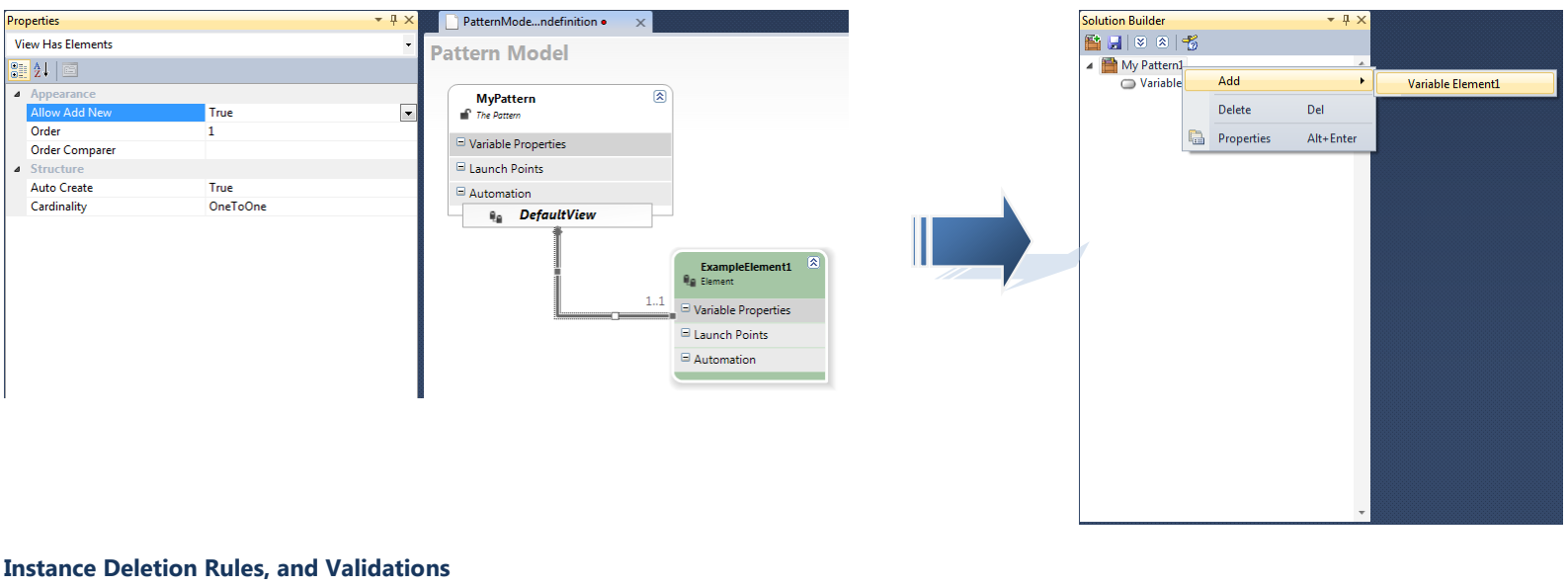


Suppressing Manual Adding of Instances

If you wish to fully control instancing of elements (i.e. using automation in your toolkit), you can prevent the user from creating instances manually using the ‘Allow Add New’ property.

When ‘Allow Add New’ is true, then the user is provided the ‘Add’ menu to add new instances, providing the number of existing instances does not exceed the cardinality of the element. When ‘Allow Add New’ is false, then only automation can create new instances of the elements.

> **Note**: Even if ‘Allow Add New’ is false, if ‘Auto Create’ is true, an initial instance of the element will always created automatically.



Instance Deletion Rules, and Validations

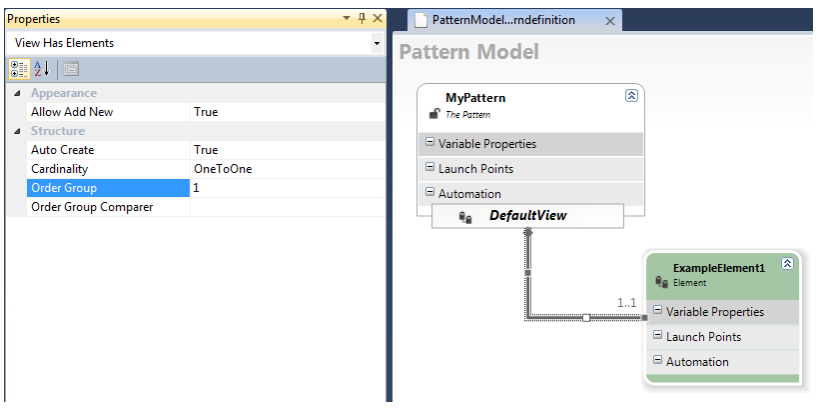
In general, a user can always manually delete any element instance regardless of the ‘Cardinality’ property applied to it; as long as they are able to recreate it manually (i.e. ‘Allow Add New’ is true). This includes deleting the last instance in a ‘One To One’ and ‘One to Many’ cardinality. Otherwise, if not allowed to manually recreate the instance, then a user can only delete instances if they don’t violate the cardinality rules. (e.g. can’t delete the last instance in a ‘One To One’ or ‘One To Many’ relationship).

> **Note**: There are no restrictions to adding and deleting of element instances using automation.
>
> **Tip**: If you want to restrict the user from creating their own instances (i.e. you have special creation rules or constraints), then have instancing managed by your toolkit by using the ‘Allow Add New’ property.

Regardless of how many instances are actually created at any one time, there are built-in validation rules to ensure that at least one instance is created in ‘One To Many’ and ‘One To One’ relationships. There are no built-in validation rules applied to ‘Zero To Many’ and ‘Zero To One’ relationships. If you want to constrain the maximum number of instances allowable for a ‘One To Many’, ‘Zero To Many’ and ‘Zero To One’ relationship, then you must define custom validation rules on the parent element. See Applying Validation for more details on creating and applying custom validation rules.

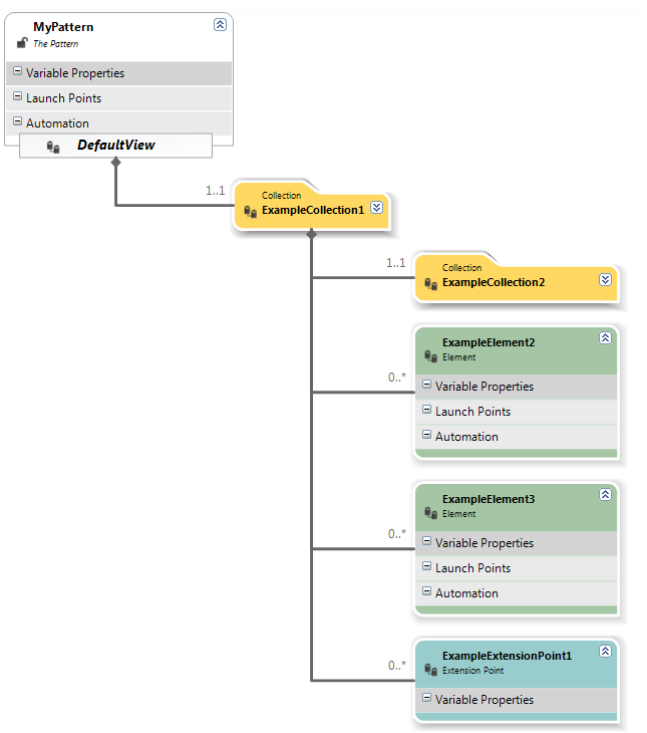
How To: Controlling Ordering of Elements

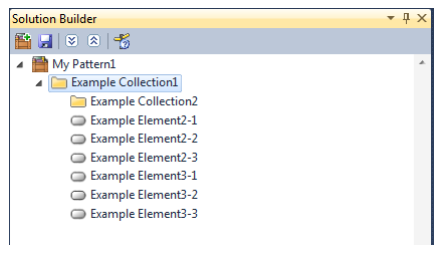
You control the ordering of instances of 'Collections', 'Elements' and 'Extension Points' by changing the Ordering properties on the connection between them.



Since any 'View', 'Element' or 'Collection' can have any number of child 'Element', 'Collection' or 'Extension Point' shapes in the pattern model, it follows that there can be one or more instances of those 'Element', 'Collection' or 'Extension Point' nested under the 'View', 'Element' or 'Collection' in the 'Solution Builder' window when the toolkit is used.

For example, consider this pattern model, where 'ExampleCollection2' has multiple child elements, collections and extension points.





'ExampleCollection1' has 4 child elements in the pattern model.
One 'Collection, two 'Elements' and one 'Extension Point'.

Because of the 'Cardinality' of these child elements, there can only be one instance of ExampleCollection1, but there can be many instances of both 'ExampleElement2' and 'ExampleElement3' and 'ExampleExtensionPoint1'.

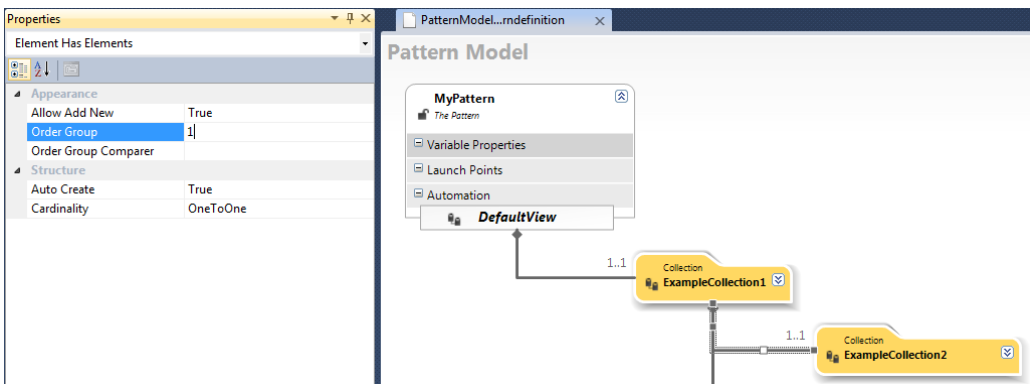
In 'Solution Builder, when a toolkit user creates instances of 'ExampleCollection2', 'ExampleElement2', 'ExampleElement3' and 'ExampleExtensionPoint1', those instances are all nested below the single instance of 'ExampleCollection1', and they must be ordered in some sequence.

This presents the opportunity for a pattern toolkit author to define the order in which the instances of each element, collection and extension points are presented.

Ordering of element, collection and extension point instances is controlled by two properties in the pattern model. Firstly, element instances can be grouped together in 'Order Groups'. Secondly, each of the groups of element instances can be ordered using a specialized 'Value Comparer' called an 'Element Ordering Comparer', that determines the order of the instances within the group.

Group Ordering

To control the order of groups of element instances use the 'Order Group' property of the connector between the parent element and the child 'Element', 'Collection' or Extension Point'. Where the lower the number the higher up the list the elements appear.



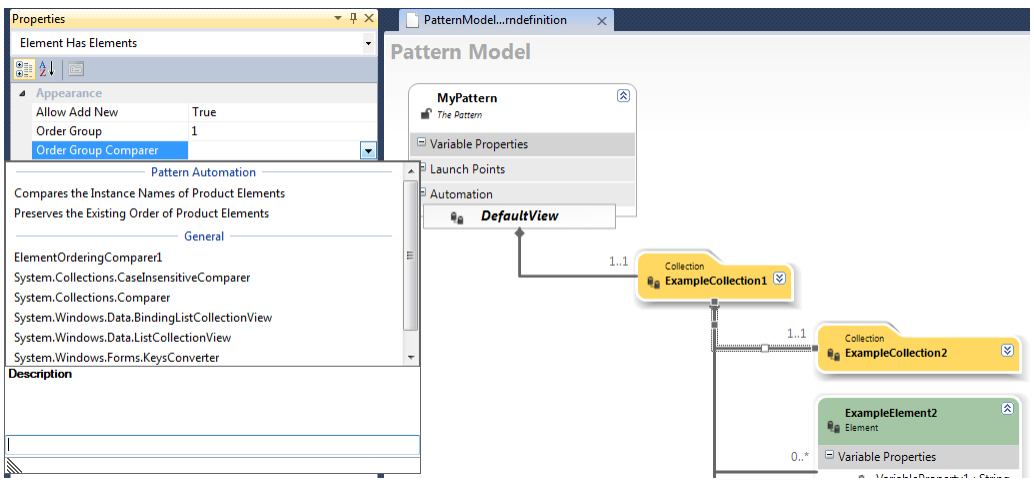
> **Note**: By default, the 'Order Group' property is set to '1' for all 'Element', 'Collection' and 'Extension Point' shapes.

In the example above, because 'ExampleCollection2', 'ExampleElement2', ExampleElement3' and 'ExampleExtensionPoint1' all have the same default value of '1' for their 'Order' property, all of their instances will be grouped together and ordered as a single group.

If you wish to create multiple groups for ordering element instances separately, simply increase/decrease the 'Order group' property of sibling 'Element', 'Collection' or 'Extension Point' shapes.

Instance Ordering

Within each 'Order Group' the instances of all elements of that group are ordered using the 'Order Group Comparer'.



> **Note**: By default, the 'Order Group Comparer' property is left blank for all 'Element', 'Collection' and 'Extension Point' shapes, which implies that instances of the element will be ordered alphabetically by their 'InstanceName' property, using the default built-in comparer.

The value of this 'Order Group Comparer' property can be any custom System.Collections.IComparer class that can order the element instances according to any scheme.

The default comparer orders element instances alphabetically by their 'InstanceName' value. This comparer can be extended to order by other properties of the elements.

> **Tip**: You can create your own comparer class in your toolkit by adding an 'Element Ordering Comparer' and implementing the 'Compare' method. See Creating Your Own Ordering Comparer for more details.

Once a custom 'Element Ordering Comparer' is configured, all instances of all elements in that 'Order Group' are ordered using the custom comparer.

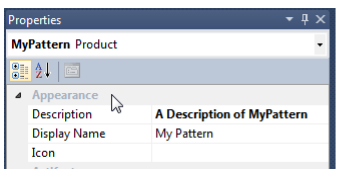
> **Note**: You can only define one 'Order Group Comparer' per 'Order Group'.

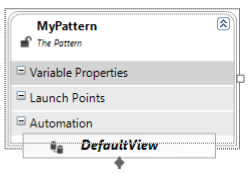
How To: Controlling the Display of the Pattern

There are a number of descriptive properties of the 'Pattern', 'Collection', 'Element' and 'Property' shapes that can be configured to control the appearance of those elements to users when applying new instances of the pattern from the pattern model.

These properties have one or more purposes and are displayed in one or more places to the toolkit user.

Pattern Element

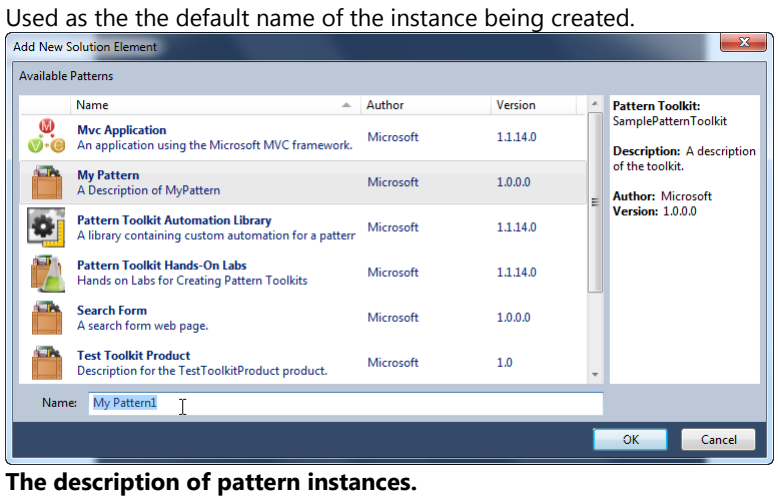




Display Name — **The default name of pattern instances.**

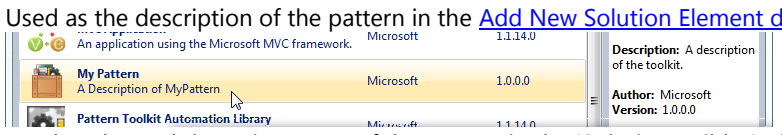
Used as the name of the pattern in the Add New Solution Element dialog.

Used as the the default name of the instance being created.

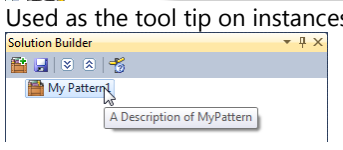


Description — **The description of pattern instances.**

Used as the description of the pattern in the Add New Solution Element dialog.

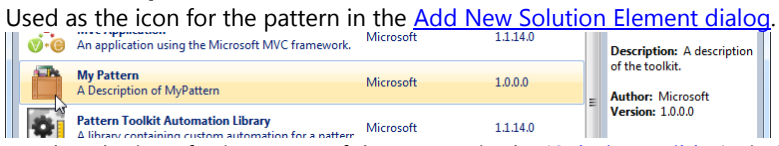


Used as the tool tip on instances of the pattern in the 'Solution Builder' window.

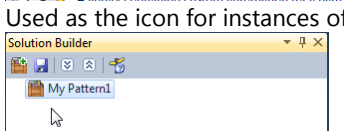


Icon — **The icon for pattern instances.**

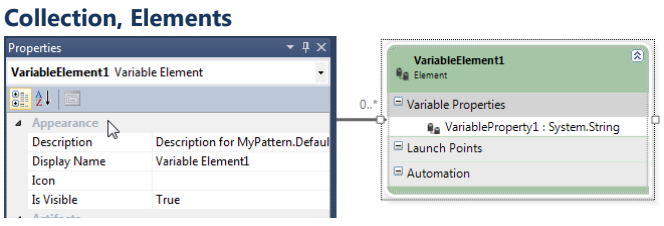
Used as the icon for the pattern in the Add New Solution Element dialog.



Used as the icon for instances of the pattern in the 'Solution Builder' window.

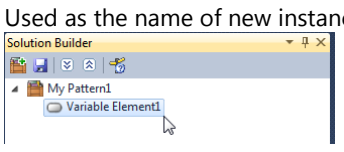


Collection, Elements

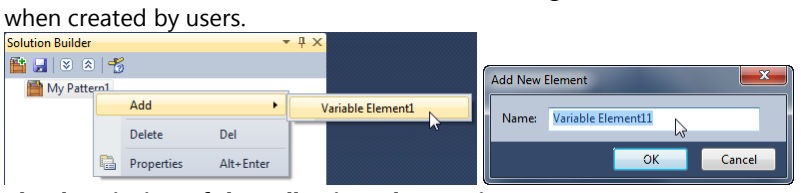


Display Name — **The name of collection/element instances.**

Used as the name of new instances created in the 'Solution Builder' window, when 'auto-created'.

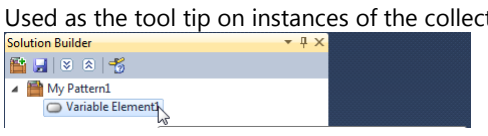


Used as the the default name of new instances being created in the 'Solution Builder' window, when created by users.



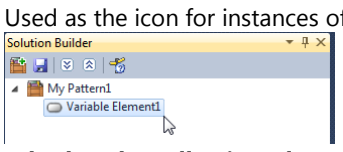
Description — **The description of the collection/element instances.**

Used as the tool tip on instances of the collection/element in the 'Solution Builder' window.



Icon — **The icon for the element.**

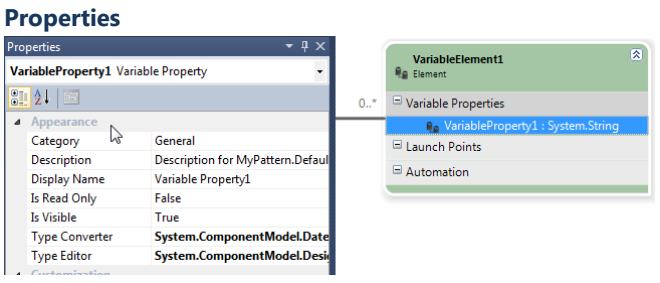
Used as the icon for instances of the collection/element in the 'Solution Builder' window.



Is Visible — **Whether the collection/element is visible to the user.**

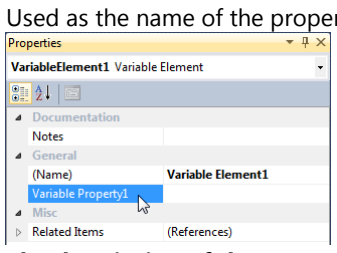
Controls whether the collection/element is seen in the 'Solution Builder' window.

Properties



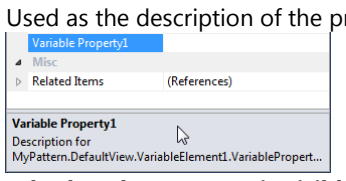
Display Name — **The name of the property**

Used as the name of the property in the 'Properties Window'.



Description — **The description of the property**

Used as the description of the property in the 'Properties Window'.



Is Visible — **Whether the property is visible to the user.**

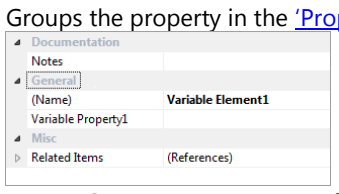
Controls whether the property is seen in the 'Properties Window'.

Is Read Only — **Whether the property is read-only, and cannot be edited by the user.**

Controls whether the property is editable in the 'Properties Window'.

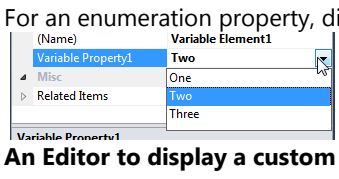
Category — **The Category that the property is grouped in**

Groups the property in the 'Properties Window'



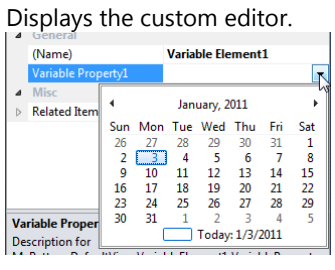
Type Converter — **A TypeConverter to convert the value of the property from its 'Type' to the value the user selects. (Also commonly used to define an enumeration for the property).**

For an enumeration property, displays a dropdown list of values.



Type Editor — **An Editor to display a custom UI for editing the the value of the property.**

Displays the custom editor.



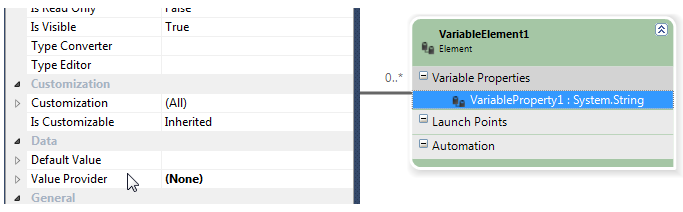
How To: Applying Validation

See Applying Validation to a Property for details on how to define validation rules for elements and properties.

See Executing Validation Rules for more details on how to execute validation in the pattern model.

How To: Setting Default Values for Properties

You can set the default value of a property so that when a new instance of the owning element is created, the property is assigned a specific default value.



> **Note**: Once this value has been set as the default, the value can be changed by either automation or by the user manually.

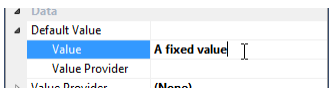
Configure the Property

A default value can be either:

1. A fixed, static value configured by the author.
2. An expression with a dynamic value configured by the author.
3. Calculated dynamically with a Value Provider.

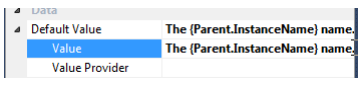
Defining a Fixed Value

In the 'Value' of the 'Default Value' property, type a fixed value.



Defining an Expression

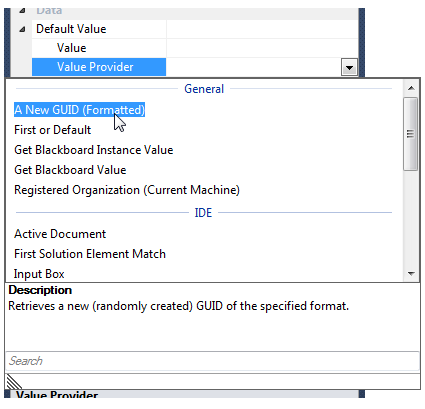
In the 'Value' of the 'Default Value' property, type an expression.



> **Note**: Expressions include substitutions between curly braces than can reference other properties of the current element (i.e. {AnotherProperty}) or reference other elements in the pattern model (i.e. {Parent.Parent.AProperty}). See the Expression Syntax for more information.

Calculating with a Value Provider

In the 'Value Provider' of the 'Default Value' property, select a value provider.

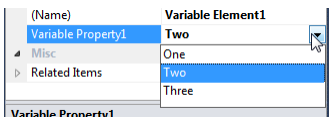


> **Note**: You cannot define a 'Value' and a 'Value Provider' at the same time.

Although there are a number of provided value providers in the toolkit that you can use for your default value, you may need to write your own to provide your specific default value. See Creating your Own Value Provider for more details.

How To: Defining an List of Values for a Property

Since a variable property only supports primitive data types (i.e. String, Int32, Boolean etc.), to get a dropdown list of values for the user to choose from requires the creation of an enumeration.

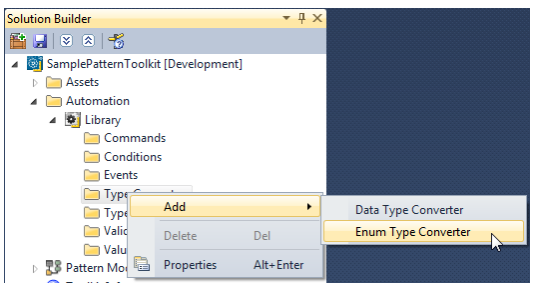


The enumeration may contain fixed values (like those from a C# enum) or dynamic values like a selection of existing element instances elsewhere in the pattern model.

In either case, to provide any list, you need to create a 'Type Converter' that returns the elements in the list and determines if the user can provide their own value or the list is restricted to entries within it.

Create a Custom Type Converter

Create an 'Enum Type Converter' from the 'Type Converters' folder in 'Solution Builder'.



In the code for the type converter:

1. Modify the descriptive properties and add appropriate tracing to the custom class. See Coding Custom Automation Classes for more details on the importance of this step.

For Exclusive Values

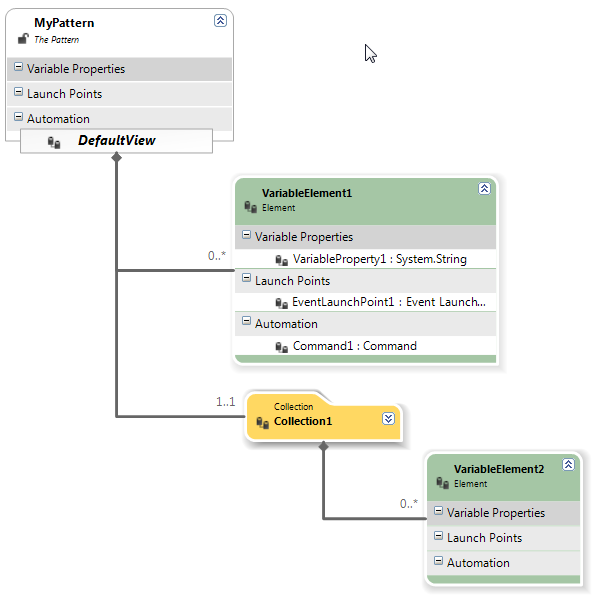
In the 'GetStandardValuesExclusive()' method, return 'true' to ensure a user picks an item from the list. Return 'false' to allow the user to type their own value (that may not be in the list).

For Fixed Values

In the 'GetStandardValues()' method, create a list of items that you want to return. The example given uses a List<string>() with some sample items, but in fact you can return a List<any> any object you like.

For Dynamic Values

If you want to return a list of elements dynamically from elsewhere in the pattern model, say for example you had a pattern model like this:



You can return a list of all the created instances of the 'VariableElement2' element, by using code like the following example:

> **Note**: This assumes that this Type Converter is configured on a property on the 'VariableElement1' element.

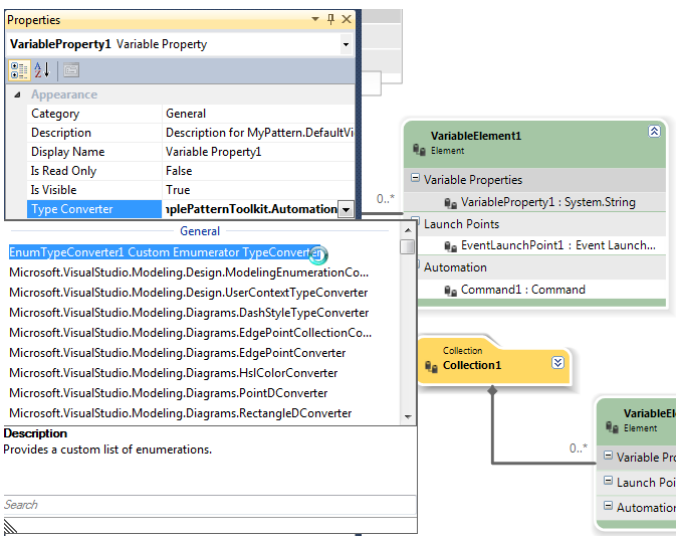
```
var currentElement = context.Instance as IProductElement;
var variableElement1 = currentElement.As<IVariableElement1>();
var collection1 = variableElement1.Parent.Collection1;
foreach (var variableElement2 in collection1.VariableElement2)
{
       items.Add(variableElement2.InstanceName);
}
```

Build the Solution.

You must build the solution in order to proceed with next step.

Configure the Property

In the pattern model, set the 'Type Converter' on the variable property of the element to use new type converter.



How To: Defining a Calculated Property

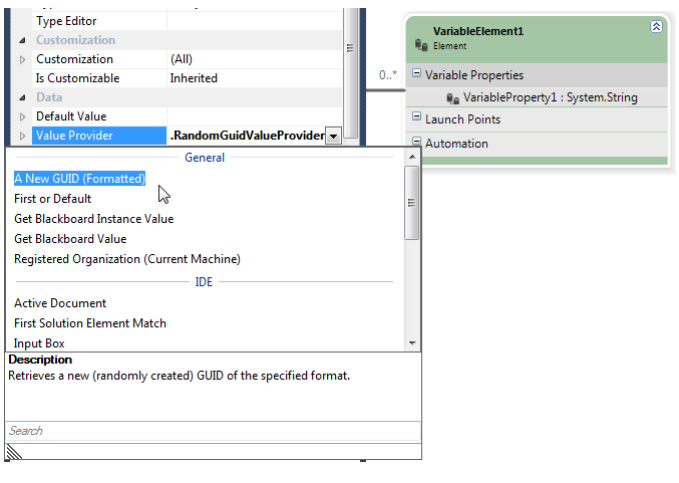
You can define a variable property to always have a calculated value. This can be useful when you need access to values in the environment that change dynamically during the use of the toolkit, or even values outside the development environment.

Another specific use of calculated properties is to have values that cannot be reached dynamically during certain automation operations, such as code generation with T4. In these cases, dynamic values will need to be 'cached' in variable properties to be accessed. These variable properties are also typically hidden from users in these cases.

> **Note**: To facilitate this need, all calculated properties are evaluated just before these automation operations to ensure the values are up to date.

Configure the Property

Select the 'Value Provider' property of the variable property



> **Note**: You cannot define any 'Default Value' and a 'Value Provider' for the same variable property at the same time.

Although there are a number of provided value providers in the toolkit that you can use for your value, you may need to write your own to provide your specific default value. See Creating your Own Value Provider for more details.

How To: Defining Custom Property Data Types

> **WARNING**: Using custom property types is not commonly required for most toolkits and is an advanced topic area. Primitive data types should be used where possible.

Since a variable property only supports primitive data types (i.e. String, Int32, Boolean etc.), to use properties with other data types, you must use a 'Type Converter' that converts to and from the custom type to one of the primitive data types, such as: String, Int32, Boolean etc. System.String is preferable.

> **Note**: Only primitive types are supported in the pattern model to simplify the persistence of the properties values in the pattern state file.

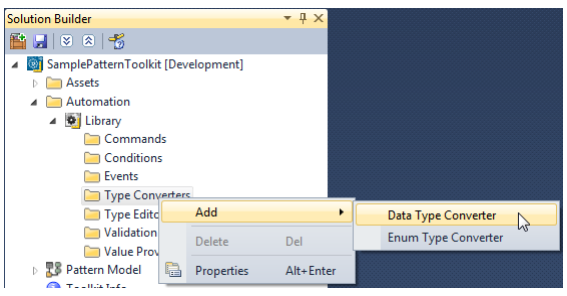
Once you have an appropriate 'Type Converter', you simply configure it on the variable property, and the property will manage the conversion from the custom data type to the primitive data type, and the toolkit will manage the persistence of the value (always in a string format).

> **Note**: If you want users of the pattern toolkit to be able to edit the property value that uses you custom data type, then you will also need to provide a custom UI Editor in addition to the Type Converter, and configure it on the property as well. See Defining a Custom UI Property Editor for more details.

Create a Custom Type Converter

> **Note**: This step is only necessary if there is not already an existing type converter for the custom data type available. You must add an assembly reference to the assembly containing the converter for it to be selectable in the list of known converters.

Create a 'Data Type Converter' from the 'Type Converters' folder in 'Solution Builder'.



In the code for the type converter:

1. Modify the descriptive properties and add appropriate tracing to the custom class. See Coding Custom Automation Classes for more details on the importance of this step.
2. See the .NET Framework guidance lines for Implementing a Type Converter.

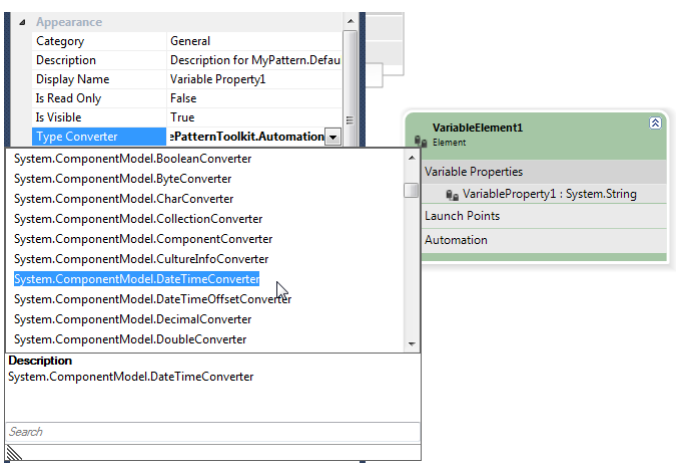
> **Note**: The IsValid() method of the converter need only verify the coarse grained ranges and bounds of the value to be converted, as validation rules on the property will ensure specific values in the pattern.

Build the Solution.

You must build the solution in order to proceed with next step.

Configure the Property

In the pattern model, set the 'Type Converter' on the variable property of the element to use new type converter.

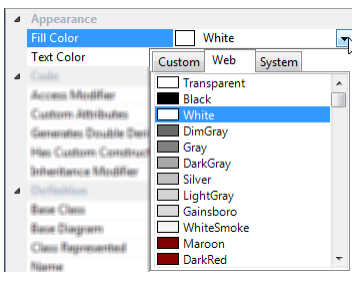


How To: Defining a Custom UI Property Editor

A UI Editor is used for displaying a custom user interface for selecting a value of a property of any type.

For example, if you had a property which is the path to a file, you might wish a give the user of the toolkit a dialog to pick that file rather than manually typing the full path on disk to the file. In this case, you would specify a UI Editor for picking files, and this UI Editor would pop up a dialog box to pick the file when the user goes to change the property value.

A UI Editor can either provide a dialog box to pick a value, or it can provide a special control that appears in a drop down window, like this color picker:

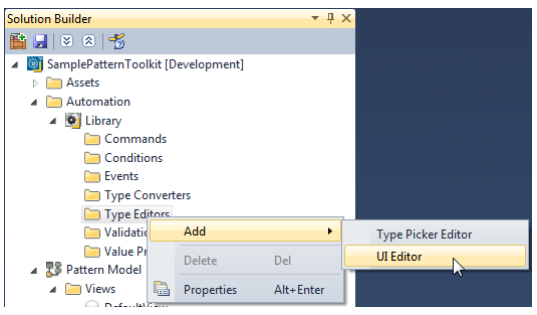


The 'UI Editor' works in conjunction with the 'Type Converter' defined on the variable property to convert the type of the object returned from the UI Editor to the primitive type of the variable property (i.e. String, Int32, Boolean etc.)

Create Custom UI Editor

> **Note**: This step is only necessary if there is not already an existing type editor available. You must add an assembly reference to the assembly containing the editor for it to be selectable in the list of known editors.

Create a 'UI Editor' from the 'Type Editors' folder in 'Solution Builder'.



In the code for the type editor:

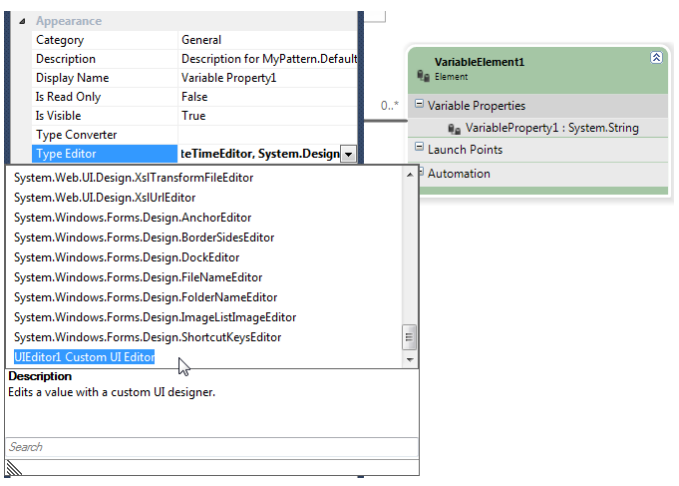
1. Modify the descriptive properties and add appropriate tracing to the custom class. See Coding Custom Automation Classes for more details on the importance of this step.
2. See the .NET Framework guidance for Implementing an UI Type Editor.

Build the Solution.

You must build the solution in order to proceed with next step.

Configure the Property

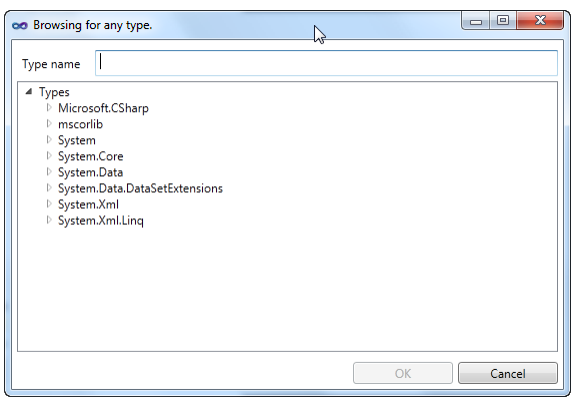
In the pattern model, set the 'Type Editor' on the variable property of the element to use new type editor.



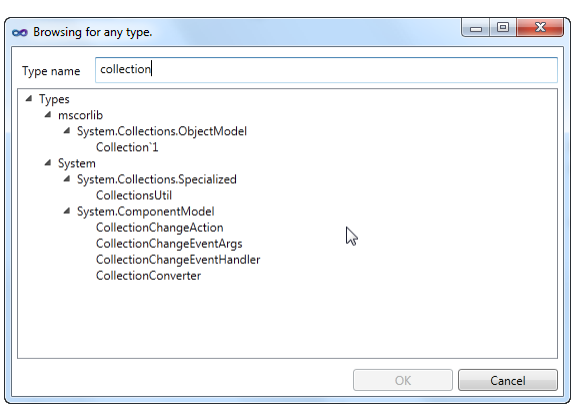
How To: Defining a .NET Type Picker Property Editor

To provide a custom UI Editor that allows a toolkit user to select a .NET type found in any of the referenced assemblies of all projects in the solution, you can create a custom 'Type Picker Editor'. Furthermore, you can constrain the list of .NET types to pick from by applying a filter to the list for types derived from a specific base class, or deriving from a specific interface.

For example, this is what the picker looks like for a user with a simple one-project solution, with no filter applied.

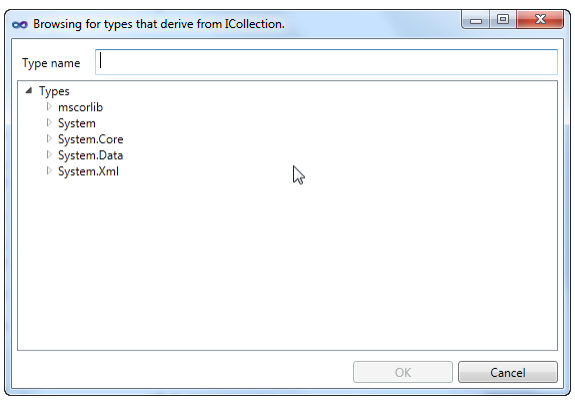


And the user can type a name to further filter the list by the name of the type they are looking for.



When a filter is applied to the picker, the list of available types is already restricted for the user.

For example, this is what the picker looks like when a filter for types deriving from System.Collections.ICollection is applied.



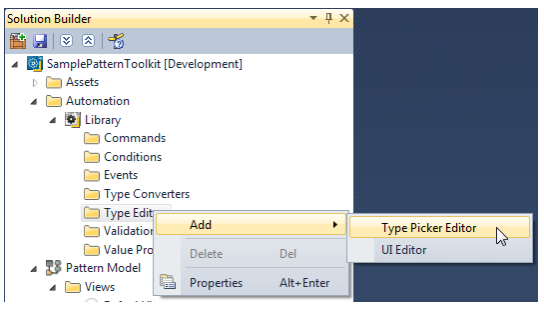
Again the user can filter further by name, but there will not be any types in the list that do not derive from System.Collections.ICollection. This is a good way to ensure the user only picks a type of the given types allowed.

> **Note**: This picker returns a string value of the type, and therefore does not require the use of a Type Converter on the same variable property.

Creating this kind of picker UI Editor is very straightforward.

Create Custom UI Editor

Create a 'Type Picker Editor' from the 'Type Editors' folder in 'Solution Builder'.



In the code for the type editor:

1. Modify the descriptive properties and add appropriate tracing to the custom class. See Coding Custom Automation Classes for more details on the importance of this step.
2. Edit the value of the [EditorBaseType] attribute, and specify the base type or interface of types you want to filter the lists of types to.

For example:

```
[EditorBaseType(typeof(ICollection))]
```

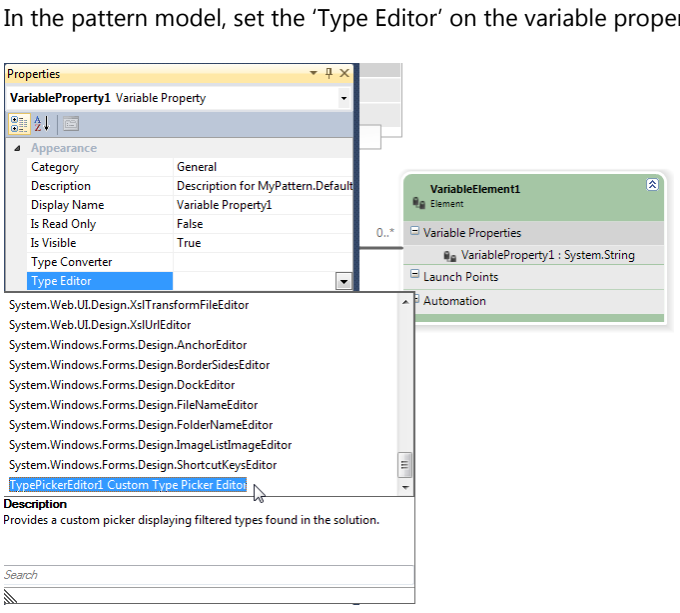
Filters the list of available types in the picker to those who derive from System.Collections.ICollection.

Build the Solution.

You must build the solution in order to proceed with next step.

Configure the Property

In the pattern model, set the 'Type Editor' on the variable property of the element to use new type editor.



How To: Controlling Customization of Elements and Properties

> **Recommend**: See What is Customization for why customization is so much more beneficial than creating a new toolkit from scratch, and for an explanation of what can and can't be customized in a toolkit.

Any pattern toolkit can be customized by creating a new customized pattern toolkit project based upon an existing toolkit. As such, in the process of building a customized toolkit, there are many aspects of the original pattern model that can be customized. These include most of the appearance attributes of elements and properties and attributes related to property values. Pattern toolkits are customized by 'tailors'. 'Authors' create the first original toolkit, which is then customized by the 'tailor'.

> **Note**: You can always add to an existing pattern model.

Furthermore, any toolkit author can refine (and only constrain) what can be further customized (about their pattern model) of the things that they can presently customize. That is to say, that because any toolkit can itself be a customized toolkit of another toolkit, the customization rules for the current author/tailor may be constrained by customization rules defined by the authors/tailors of the original toolkits.

Of the attributes that can be customized in a pattern model, all of them are fully customizable by default. To constrain whether they are customizable by subsequent tailors, an author/tailor must explicitly apply a 'customization policy' to the specific elements they wish to constrain.

'Customization Policies' can be applied to individual elements in the pattern model, or can be defined once on ancestor or parent elements in the model and inherited by all descendant elements. By default, all elements in the pattern model 'Inherit' their customization policy from their parent element. The top-most parent element (the 'Pattern' element) controls the overall customization policy of the whole pattern model.

> **WARNING**: Applying a customization policy to your pattern model constrains what can be customized by subsequent tailors wishing to customize your pattern. If reuse is your goal, apply customization policies very sparingly. By default, everything in the pattern model is customizable (i.e. no customization policy is applied).

Granularity and Inheritance

You can 'lock' customization of all customizable attributes of an element/property with coarse grained control

You can 'lock' customization of single customizable attributes of an element/property with fine grained control of each attribute.

You can 'selectively' configure either coarse or fine grained customization policies to individual elements in the pattern model.

You can 'inherit' the coarse grained customization control from your parent or ancestor elements.

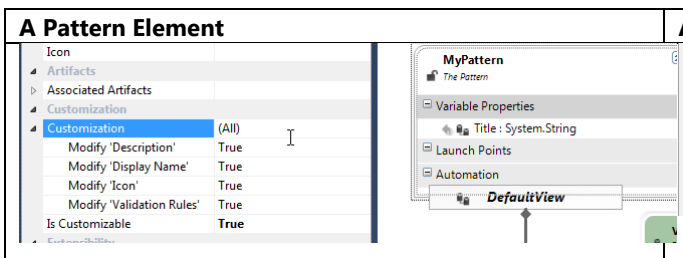
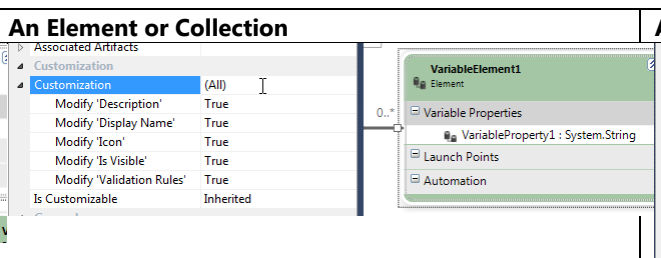
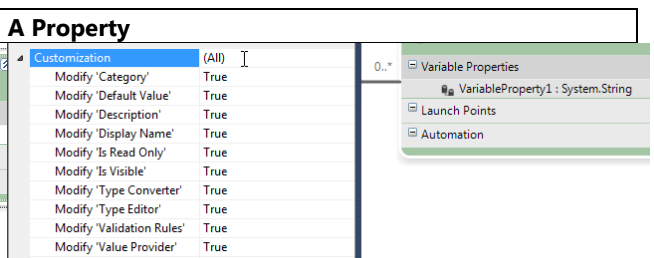
You can 'blanket' configure coarse grained customization policies to descendant elements in the pattern model.

You can override coarse grained control at any descendant level.

For Example, in an arbitrary multi-branched pattern model with several elements and properties, when a parent or ancestor element locks all of its customization (coarse grained), then all descendant elements (and their properties) are also locked (coarse grained) when they inherit their customization policy. But you can still for selected descendant elements/properties, override that coarse grained control and configure fine grained control for just that element, and/or its descendants.

Viewing the Customization Policy

You can view the current customization policy of any element or property, by viewing the 'Is Customizable' and 'Customization' properties.

A Pattern Element	An Element or Collection	A Property
		
'Is Customizable', is by default 'True', (and cannot be set to 'Inherited').	'Is Customizable', is by default 'Inherited'	'Is Customizable', is by default 'Inherited'

> **Note**: If the element is not customizable ('Is Customizable' is false) then the customization policy is not viewable.

The value of the 'Is Customizable' property is also reflected with an icon on every shape in the pattern model.

Is Customizable		
True	**False**	**Inherited**
		

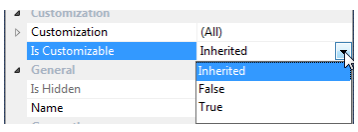
> **Note**: The value of the 'Is Customizable' property and the icon on descendant elements that are 'inheriting' do not change when the 'Is Customizable' property changes on the parent or ancestor element.
> The 'effective' value of the 'Is Customizable' property for any element/property is not calculated and does not take effect until the toolkit is customized by a tailor.

Configuring the Customization Policy

There are basically two kinds of changes you can make for any element/ property for its 'customization policy'. You can either turn-off customization for the whole element/property, or have more fine grained control by controlling customization of individual attributes of the element/property.

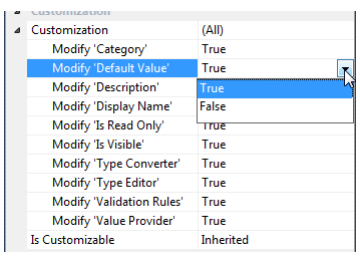
Coarse Grained Customization

Change the value of the 'Is Customizable' property for the element/property.



Fine Grained Customization

Change the value of the individual setting in the 'Customization' property for the element/property.



Assets

Understanding: What are Assets?

See What are Assets for a broader overview of what assets are in pattern toolkits.

Essentially, Assets are the parts of a pattern toolkit that set the foundation for the implementation of the pattern, and determine the shape of the toolkit. Without assets all you really have is a theoretical model without an implementation. The Assets themselves determine how the pattern is represented and automated. This is why it is recommended to consider spending some time developing a draft design of the toolkit, by cataloguing the assets and proposed automation of the solution implementation. See Create a Production Tooling Workflow. That process also identifies candidate Automation which is typically applied to create, configure, tune and exchange the Assets. The two are integrally related.

We make a distinction between the harvested Solution Implementation Assets which can be either fixed or variable, and the Automation Assets that are developed and configured in a pattern toolkit.

In a pattern toolkit, the most commonly harvested and developed assets are:

- VS Templates
- Text Templates
- Guidance

The three main solution-based assets

Understanding: What are VS Templates?

See What are VS Templates for a broader overview of what VS templates are and how they are used in pattern toolkits.

See Visual Studio Templates for references on how to create and configure them.

See VS Template Design Guidelines for designing your VS templates effectively for pattern toolkits.

See Creating, Exporting, Importing, Substitution and Unfolding topics for how to create and integrate VS Templates into your pattern model.

Understanding: VS Template Design Considerations

There is extensive and very flexible support for 'unfolding' VS project and item templates in your pattern toolkit.

Benefits

Traditionally in Visual Studio, when users create projects, and add items to projects, they use the Add New Project/Item dialog to select the kind of project or file they want. And all the project and item templates are listed categorized there.

Your toolkit can also supply project and item templates to the Add New Project/Item dialog, and when they are used they can create the necessary elements in your pattern model. This allows you to apply patterns to the templates that users normally use, which directs them to use your pattern in solution builder to complete the pattern.

In addition, with a pattern toolkit, you can also unfold those project or item templates from any element in your pattern model. This ensures that the elements in your pattern model are correctly represented with the correct files and projects in the solution automatically. The user does not have to choose which file to select to implement the pattern.

Furthermore, a pattern toolkit can unfold projects or items when appropriate, and in response to user gestures or events. E.g. when the user has creates certain elements in the pattern, files are added to the project, or when they have finished configuring certain elements in the pattern, a project with content is unfolded to represent that specific configuration. Templates can also be unfolded using menus and other events.

Traceability & Composition

VS templates are bound to specific elements in a pattern model (i.e. the element that automates their unfolding) that often represent an abstraction of their meaning or content in the solution. When a project or item template is unfolded from an element, that element is bound or related to the project or item using an artifact link. That link enables automation in the toolkit to track the project or file in the solution and perform automation with it.

> **Note**: These kinds of artifact links are resilient to refactoring the project or file by renaming it, or moving around the solution.

An element may have one or more links to one or more artifacts (many-to-many). It follows then that in order to track or perform any kind automation on a project or item unfolded from a VS template an artifact link is necessary. Therefore, by default, when a VS template is unfolded an artifact link is automatically added to the instance of the element that the template was bound to. This is a very important consideration when designing your VS templates for use in a pattern toolkit.

Design for Automation

In general, if you need access to a specific project or item (for automation purposes in your toolkit) and that project or item is unfolded by a VS template from your toolkit, then you need an artifact link. Since only one artifact link is created per project or item template, and that link points only to the project or specific item, then you need to decompose your VS templates into the items and projects which you need automation access to.

For example, let's say you have an existing VS project template that unfolds a specific project with multiple specific files within. Some of those files are code and configuration files that have variable content, which you are either going to automatically generate, or modify parts of using automation with data from your pattern model. Since the filenames and locations in the project could be renamed by a user as they refactor the project over time, the challenge for you is how to determine which files are now the ones you intend to access for automation? The answer is with artifact links. And since you will need an artifact link for each file you automate, you will need to break down what was once a suitable project template into a smaller project template with multiple separate item templates. The pattern model and the automation you configure on it then reconstitutes the whole project from multiple VS templates seamlessly, but with the ability to track and access individual files within it.

Design for Composability

Another key reason to break your VS templates into smaller units is because your pattern represents each of those units with different elements in the pattern model, and those project or items may have a different lifecycle or deployment or versioning model.

Multi-Projects Templates

> **WARNING**: There is very limited support for multi-project (or solution) templates in pattern toolkits (i.e. where type="ProjectGroup"). For the design reasons discussed above they are typically best decomposed into individual project templates.
> They can still technically be used in a limited way for a small set of scenarios, but with the following constraints:
>
> - Can have zero or more projects.
> - No artifact links are created for any project unfolded by them.
> - Cannot set 'Unfold When Created' as true in a 'Template Launch Point', nor configure on an 'Unfold VS Template Command'. (i.e. cannot create a new solution when an element is created, as this closes and replaces the existing solution containing the pattern instance)

How To: Creating a VS Template

> **Warning:** Creating VS templates from scratch is not a trivial process and requires in-depth knowledge of this activity that is not described in detail in this guidance.
>
> **Tip**: Rather than creating a VS Template from scratch), it is strongly recommended to export one from an existing project in Visual Studio. See Exporting a VS Template for more details.

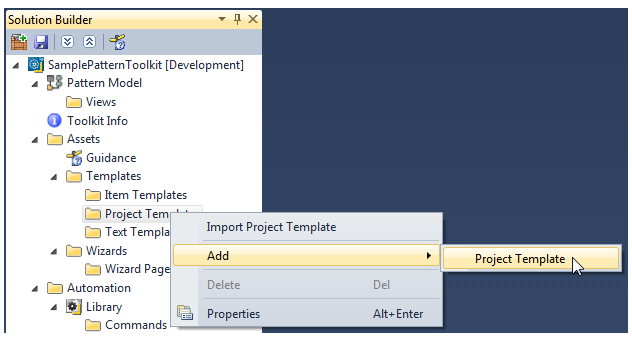
Process Overview

To create a new VS template for use in a pattern toolkit:

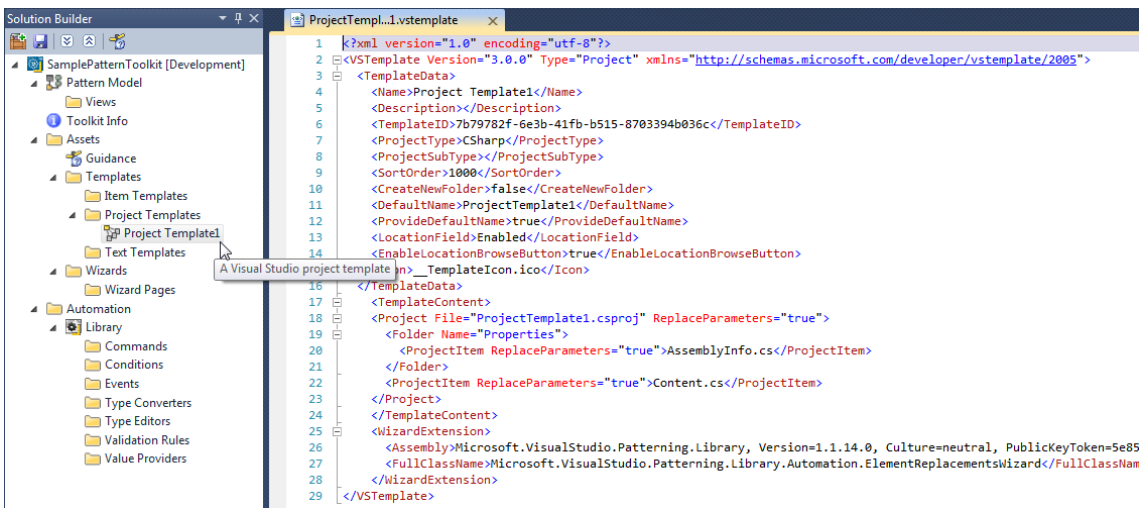
1. Create the new VS template.
2. Edit the VS template manifest to describe the template, and define the structure and items within the template.
3. Automate the unfolding of the template in the pattern model.

Create the Template

To create a new VS Template (Item or Project) from scratch, in your toolkit project, right-click on the 'Project Templates' or 'Item templates' folders in 'Solution Builder', and select Add.



Then double-click on the new element in 'Solution Builder' to open the template for editing.



Edit the Template

VS Templates are configured by editing the XML manifest file (*.vstemplate file) and the contents of the other files included in the template.

> **Note**: Refer to Visual Studio Templates for more details on their structure.

As VS templates support textual substitution, you can edit these files to substitute values from both the environment and from the pattern model when the templates are unfolded. See Substituting Values in VS Templates for more details.

Automate the Template

See Unfolding VS Templates for how to automate the unfolding of the VS template in a pattern model.

How To: Exporting a VS Template

> **Note**: Exporting/Importing a VS template is the recommended practice for creating new VS Templates (or customizing existing VS templates) in a pattern toolkit project. Creating a VS template from scratch is not recommended unless you are fluent in the detailed knowledge of VS template creation. If so, see Creating a VS Template for the best way to create a new VS template from scratch for a pattern toolkit.
> **Important:** Exporting a VS template is the first part of a two part process of adding a VS template to your pattern toolkit project. After exporting the VS template, you must import the template into your toolkit project. See Importing a VS Template for more details on that process.

Process Overview

Before you can export a VS template from Visual Studio, you must create the project and/or items in 'Solution Explorer' to export.

To export a VS template for use in a pattern toolkit:

1. Construct the desired Project or Items (files) you wish to export in a Visual Studio solution. Finalize the structure and naming, configuration of projects, and content of files.
2. Harvest the template, by running the 'Export Template Wizard' to create a VS template (*.zip) archive on disk.
3. Import the VS template package into your toolkit project.
4. Automate the unfolding of the template in the pattern model.

Constructing the Template

To construct the template, create and set up new projects and add files in a solution in Visual Studio. You can use any existing project or item template already installed in Visual Studio to get started.

> **Tip**: Consider creating the project or files in the toolkit solution and harvest them from there. A separate solution is not necessary.

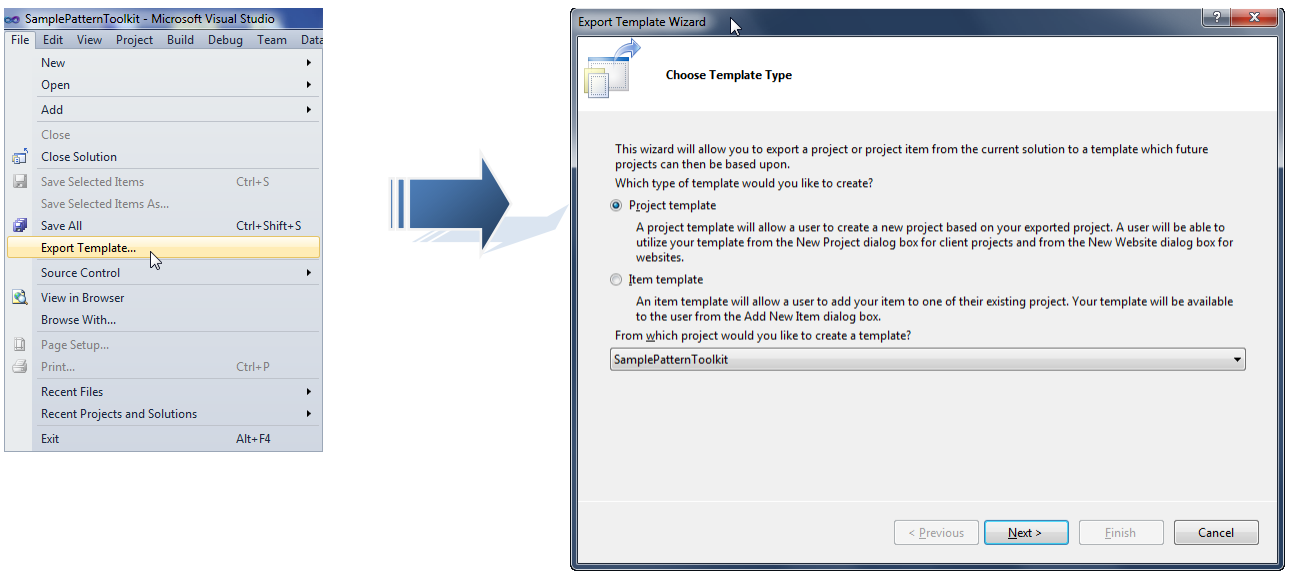
Pay special attention to naming, structure and configuration of projects, as these are also copied to the template.

Remove any files, folders, references or configuration not necessary in the template, and pare it down to its bare essentials.

> **Note**: The basic principal is that the template only contains the bare fixed (or only slightly variable) essentials, and other things that are more variable can be added by your toolkit using automation.

Exporting the Template

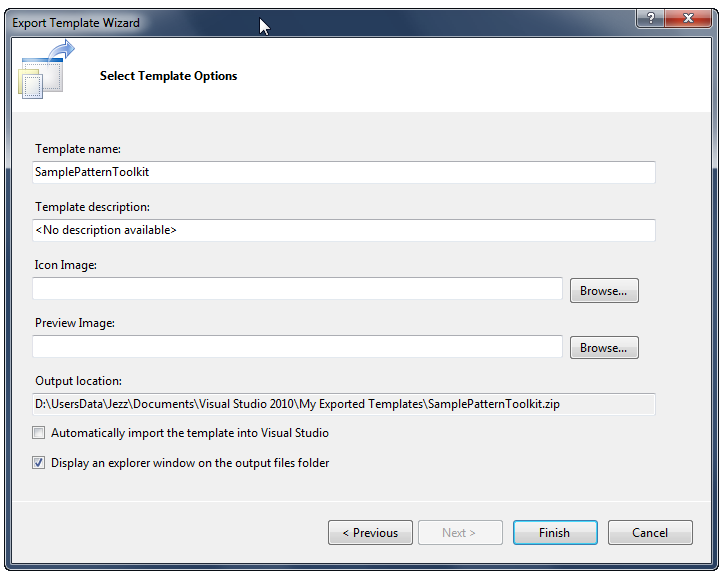
To export the project or just an item from the solution, run the 'Export Template Wizard' in the solution (File | Export Template).



The export wizard asks which kind of template to export (Project or Item), then a set of questions on selecting exactly which project or items to select, and their dependencies.

On the final 'Template Options' page, the wizard asks how to name the template and where to create the template archive (a *.zip file).

> **Important**: It is vitally important that you **deselect** the **'Automatically import the template into Visual Studio'** option in this wizard page, failing to do this will mean that you will have two templates installed into Visual Studio causing unnecessary confusion, one created by this Export Wizard and one created by your pattern toolkit.



> **Note**: The name and description are not so important at this stage as they will be updated when your template is imported and used in the pattern toolkit. Add an existing icon if you have one. A default icon will be included if none provided, and that can be updated later in the pattern toolkit.
> **Note**: The 'Output Location' is where the template archive (a *.zip file) will be deposited. This directory defaults to the 'Exported Templates' folder of the current users default Visual Studio directory. The template importer will start looking for your template in this directory by default also.

Importing the Template

See Importing a VS Template to complete this process.

How To: Importing a VS Template

Importing a VS template is the second part of a two part process in adding a VS template to a pattern toolkit. The first part is to create a VS template archive (a *.zip file) using the process in Exporting a VS Template, or by obtaining an existing VS template archive (i.e. from a Visual Studio SDK or extension).

> **Note**: If you create a VS template from scratch using the method described in Creating a VS Template, the VS template already exists in the pattern toolkit project and there is no need to import the template again.

Process Overview

Before you can import a VS template into your pattern toolkit, you must have already created or obtained a VS template archive (a *.zip file).

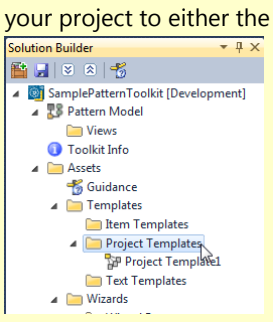
To import a VS template for use in a pattern toolkit:

1. Import the VS template package into your toolkit project.
2. Automate the unfolding of the template in the pattern model.

Import the Template

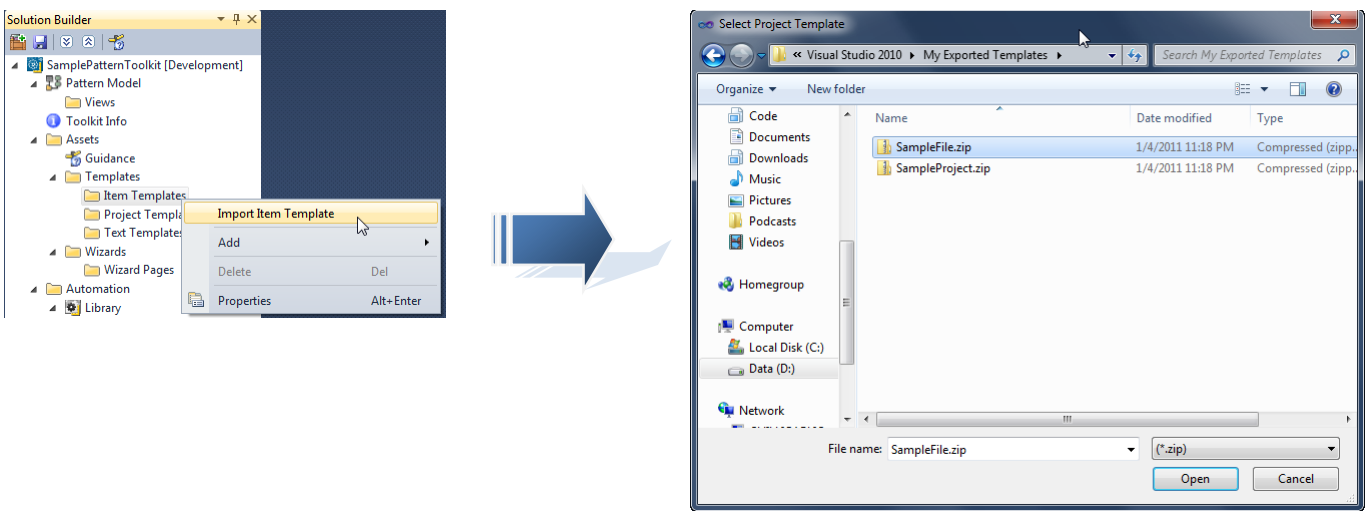
There are two automated methods to import a VS template archive: use the import wizard, or drag and drop from windows explorer.

> **Note**: Although not recommended, you can also manually unzip the VS template archive, and copy the contents into a either the 'Projects' or 'Items' sub folder of the 'Assets' folder in a pattern toolkit project. You will then need to register the new template with the pattern toolkit, by dragging the unzipped *.vstemplate file from your project to either the 'Project Templates' or 'Item Templates' folder of your pattern toolkit in 'Solution Builder'.



Use the Import Wizard

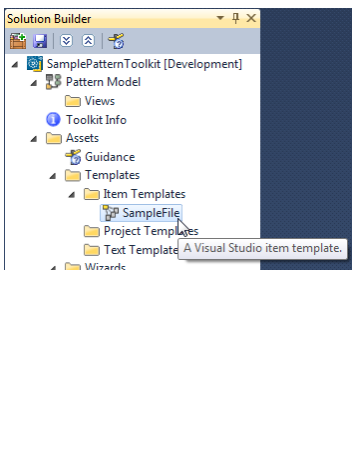
To import an existing or exported VS Template (Item or Project) into your toolkit project, right-click on the 'Project Templates' or 'Item templates' folders in 'Solution Builder', and select 'Import'.

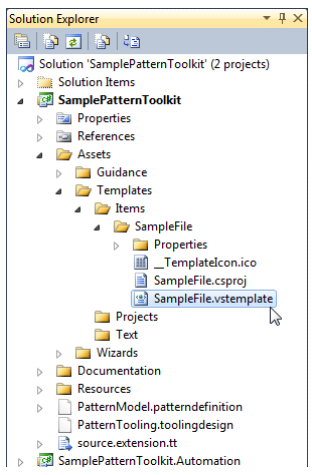


You will then be asked to select the VS template archive (a *.zip file) from the dialog box.

> **Note**: You can select one or more template archives to import.
> **Note**: By default this dialog box imports from the same directory that the 'Export Template Wizard' exports to.

The template is then imported into your pattern toolkit project and registered in your toolkit.

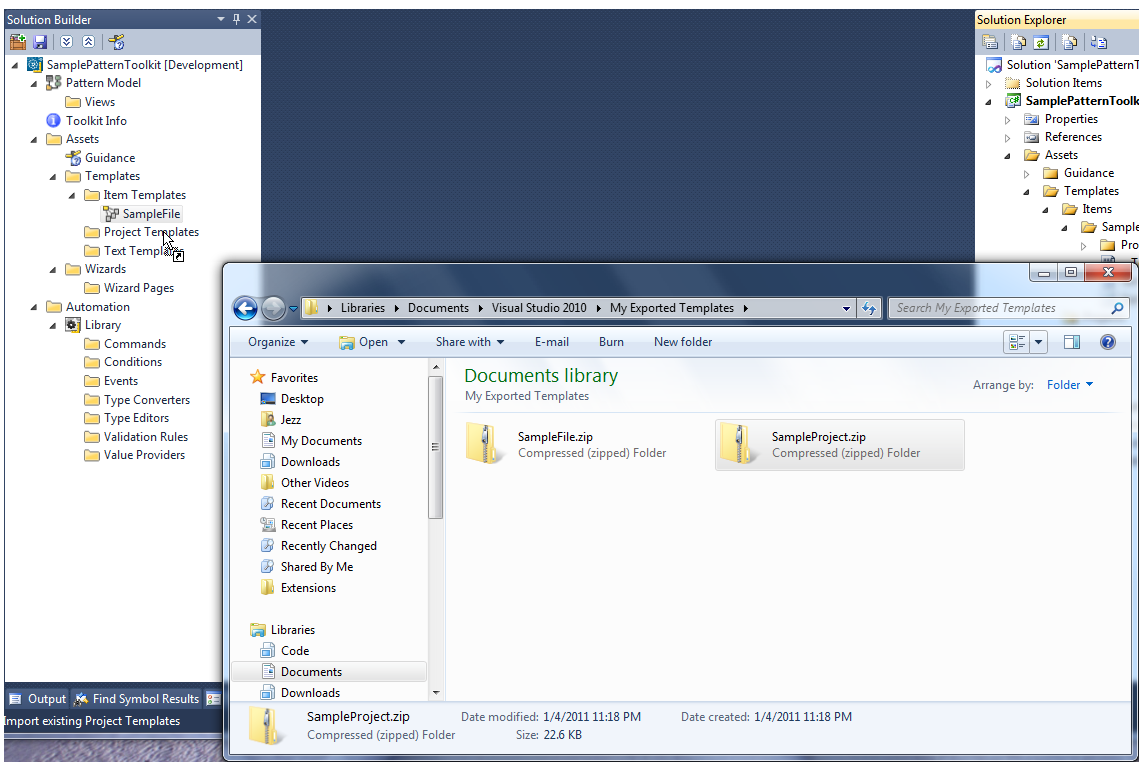




Drag and Drop From Windows Explorer

To import an existing or exported VS Template (Item or Project) into your toolkit project form 'Windows Explorer', simply drag the VS template archive (a *.zip file) from Windows Explorer, and drop on either the 'Project Templates' or 'Item templates' folders in 'Solution Builder'.

> **Note**: You can drag and drop one or more template archives to import.



Similarly to using the import wizard, the template is then imported into your pattern toolkit project and registered in your toolkit.

Edit the Template

VS Templates are configured by editing the XML manifest file (*.vstemplate file) and the contents of the other files included in the template.

> **Note**: Refer to Visual Studio Templates for more details on their structure.

As VS templates support textual substitution, you can edit these files to substitute values from both the environment and from the pattern model when the templates are unfolded. See Substituting Values in VS Templates for more details.

Automate the Template

See Unfolding VS Templates for how to automate the unfolding of the VS template in a pattern model.

How To: Substituting Values in VS Templates

> **Note**: Refer to Visual Studio Templates for more details on how to customize VS templates with substitutions.

A VS Template supports textual substitutions in both the manifest file (*.vstemplate file) and the content files within the template. All substitutions are inserted into the source files between '$' characters. For example, to substitute a random GUID into one of the files you would insert $guid1$ or $guid2$, where you want to make the substitution.

Built-In Substitutions

There are a number of built-in common substitutions for both project and item templates. Refer to Template Parameters for a complete list of built-in substitutions for VS templates.

> **Note**: Some of these substitutions are only valid for project templates, some only for item templates.

Pattern Model Substitutions

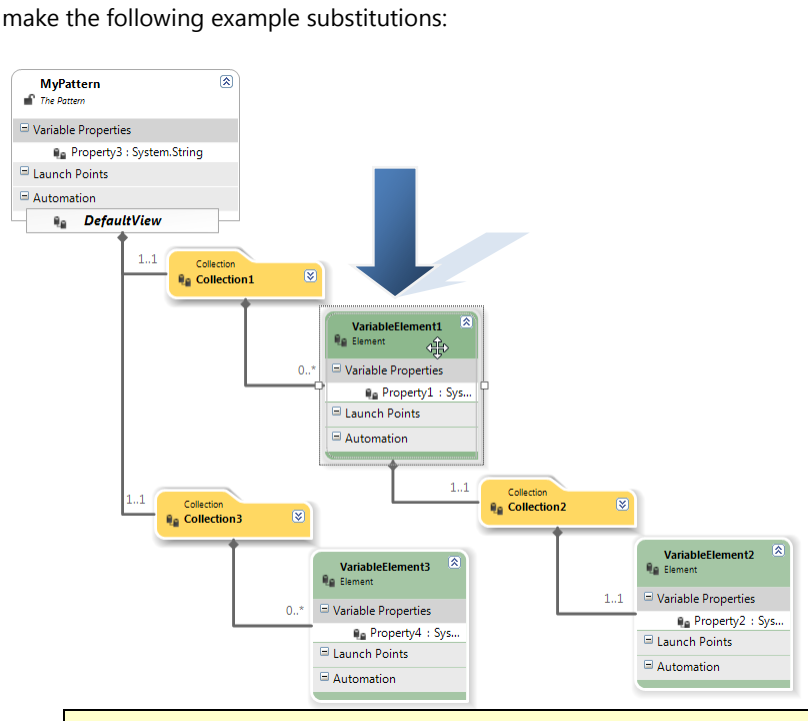
As well as the set of built-in substitutions that come as default for VS templates above, when a VS template is unfolded from automation a pattern model, you are able to substitute any variable property (including the user defined name of the solution element instance) from any element in the pattern model. These are called pattern-model substitutions.

> **Note**: The critical thing to know when inserting pattern model substitutions is from which element the template is being unfolded from.

From any element in the pattern model you can define an expression to be substituted that can reach any property of any instance of any pattern element. The syntax is the same as the built-in substitutions and follows this syntax: $ElementName.PropertyName$. You can also navigate up the ancestry from the current element and down into their descendancy.

For Example

In the given pattern model with the given elements, collection and cardinalities of the relationships, where we unfold a VS template from the 'VariableElement1' element, we can make the following example substitutions:



Expression	Notes
$Property1$	Properties of the current element.
$InstanceName$	'InstanceName' is a built-in property for the user-defined name of the instance of the current element
$Collection2.Property2$	Properties of descendant elements in OneToOne or ZeroToOne relationships
$Parent.Parent.Parent.Property3$	Properties of ancestor elements
$Parent.Parent.Collection3.VariableElement3[*n*].Property3$	Properties of relative's elements in OneToMany or ZeroToMany relationships, where the *n* is the zero-based index of the instance of the element.

> **Note**: In the substitutions above, the current view of the pattern element 'DefaultView' in this case, is also a parent element of 'Collection1' and 'Collection3' and must be included in the expression.

Additional Custom Substitutions

It is possible to provide additional substitutions to your template using two methods:

> **Tip**: Both the following methods below use native VS mechanisms that are superseded with pattern-model substitutions above, and are only mentioned here for completeness. With pattern-model substitutions (and variable properties with static values or with value providers) you can achieve all that you can achieve with the native VS methods.

Static Custom Substitutions

If you have static values that can be defined at design time that you want to substitute into more than one file in the VS template, you can define these globally in the *.vstemplate file in the <CustomParameters> element.

> **Note**: This approach only works for substitutions that do not vary with your pattern, and where your goal is simply to reuse the same static value across multiple files in the template.

For example:

```
<TemplateContent>
    ...
    <CustomParameters>
        <CustomParameter Name="$MyParameter1$" Value="MyValue2"/>
        <CustomParameter Name="$MyParameter2$" Value="MyValue2"/>
    </CustomParameters>
</TemplateContent>
```

See Pass Custom Parameters to Templates for more details on this mechanism

Dynamic Custom Substitutions

> **Tip**: The following is a very advanced Visual Studio integration topic, and requires deep knowledge and experience in VS template Wizards. Consider instead an approach where dynamic values are defined in variable properties (with value providers) in your pattern model and substituted as pattern model substitutions instead.

If you want to provide dynamic values that are determined when the pattern in unfolded based on the environment, or calculated by automation, and which are not in the pattern model, you can implement your own TemplateWizard to provide custom dynamic values that can also be substituted.

See the Template Wizard reference for more details on how to add values to the replacementsDictionary in the RunStarted() method.

How To: Unfolding VS Templates from a Pattern

See Unfolding VS Templates for the process of configuring automation to unfold of the VS template in a pattern model.

How To: Troubleshooting VS Templates

> **Note**: See Troubleshooting Toolkits for more details on viewing the trace window and diagnostic information for toolkits when errors occur.

Creating and refining VS templates gives rise to a number of errors which are often very difficult to troubleshoot and resolve.

Typically they are first seen as failures to unfold template in pattern toolkits. Usually with errors and stack traces in the Tracing Window that look similar to this:

```
System.IO.FileNotFoundException: Could not find file
'C:\Users\Administrator\AppData\Local\Microsoft\VisualStudio\10.0Exp\Extensions\Microsoft\SamplePatternToolkit\1.0.0.0\Assets\Templates\~PC\Projects\Project
Template1.zip\ProjectTemplate1.csproj'.
File name: 'C:\Users\Administrator\AppData\Local\Microsoft\VisualStudio\10.0Exp\Extensions\Microsoft\SamplePatternToolkit\1.0.0.0\Assets\Templates\~PC\Projects\Project
Template1.zip\ProjectTemplate1.csproj'
   at System.IO.__Error.WinIOError(Int32 errorCode, String maybeFullPath)
   at System.IO.File.InternalCopy(String sourceFileName, String destFileName, Boolean overwrite)
   at System.IO.File.Copy(String sourceFileName, String destFileName, Boolean overwrite)
   at Microsoft.VisualStudio.TemplateWizard.Wizard.Execute(Object application, Int32 hwndOwner, Object[]& ContextParams, Object[]& CustomParams, wizardResult& retval)
   at EnvDTE._Solution.AddFromTemplate(String FileName, String Destination, String ProjectName, Boolean Exclusive)
   at Microsoft.VisualStudio.TeamArchitect.PowerTools.VsIde.VsProjectTemplate.Unfold(String name, IItemContainer parent)
   at NuPattern.Library.Commands.UnfoldVsTemplateCommand.UnfoldTemplate(ISolution solution, IFxrUriReferenceService uriService, IServiceProvider serviceProvider, IProductElement owner,
UnfoldVsTemplateSettings settings, Boolean fromWizard) in D:\Projects\Platu\Dev\Team\Src\Library\Source\Commands\UnfoldVsTemplateCommand.cs:line 220
   at NuPattern.Library.Automation.TemplateAutomation.<Execute>b__2() in D:\Projects\Platu\Dev\Team\Src\Library\Source\Automation\Template\TemplateAutomation.cs:line 123
   at NuPattern.Extensibility.TracingExtensions.DoShield(ITraceSource traceSource, Action action, String format, Boolean showUI, String[] args) in
D:\Projects\Platu\Dev\Team\Src\Common\Source\Extensibility\Extensions\TracingExtensions.cs:line 94
```

The causes for these kinds of difficult to diagnose errors are often to do with the caching of templates that Visual Studio does for performance reasons, and for that reason require special steps to resolve manually.

Resolving Issues

Verify the following steps, and apply the appropriate steps to remedy the issue:

- The pattern model validates without error and the pattern toolkit project builds without errors.

> **Remedy**: Fix all errors, clean and rebuild solution.

- The VS template is zipped up an deployed as a *.zip file to the Extension folder of the Experimental Instance of Visual Studio at: %localappdata%\Microsoft\VisualStudio\10.0Exp\Extensions\<Author>\<ToolkitName>\<Version>\Assets\Templates\<ItemsOrTemplates>

> **Remedy**: If the zip file is not deployed in your toolkit extension folder then you need to re-associate it to automation:
>
> 1. Re-associate the VS template to automation of a pattern element in the pattern model.
> 2. Clean and Rebuild the pattern toolkit solution.

- When you open the Zip file, you see more than one *.vstemplate file contained within.

> **Remedy**: If there is more than one *.vstemplate file in the Zip file, then you need to delete them:
>
> 1. Find and delete all the other *.vstemplate files found in either the project folders or on the disk at the same location as the project.
> 2. Shutdown Visual Studio, and delete all 'bin' and 'obj' sub-directories found in the directory of your solution.
> 3. Restart Visual Studio, Clean and Rebuild the pattern toolkit solution.

- When you open the Zip file, you see all the content files referred to in the *.vstemplate file, correctly named and in the correct structure.

> **Remedy**: If any of the files in the *.vstemplate are named incorrectly or missing they need to be fixed in the pattern toolkit project:
>
> 1. Fix the issue in the pattern toolkit project
> 2. Re-associate the VS template to automation of a pattern element in the pattern model.
> 3. Clean and Rebuild the pattern toolkit solution.

- When you open the Zip file, and the *.vstemplate file, the <ProjectType>, <TemplateID> matches that of the URI associated to the element in pattern model.

> **Remedy**: If the <ProjectType> or <TemplateID> does not match then the template is not associated to the pattern element correctly then you need to re-associate it to automation:
>
> 1. Re-associate the VS template to automation of a pattern element in the pattern model.
> 2. Clean and Rebuild the pattern toolkit solution.

- When you open the Zip file, and change the value of the <Hidden> element (if any) to false, and save the zip file, you are able to see and create a new project/new item from the template in Visual Studio.

> **Remedy**: If you cannot see the template in the 'Add New Project/Item dialog' in Visual Studio then you need to:
>
> 1. Shutdown Visual Studio
> 2. Reset the Visual Studio Experimental Instance
> 3. Re-associate the VS template to the pattern element in the pattern model again, and clean and rebuild the solution.

> **Remedy**: If you cannot create a project or item from the template in the 'Add New Project/Item dialog' in Visual Studio then your template may be either corrupt or incorrectly configured, and needs fixing in the pattern toolkit project.
>
> 1. Fix the issue in the pattern toolkit project
> 2. Re-associate the VS template to automation of a pattern element in the pattern model.
> 3. Clean and Rebuild the pattern toolkit solution.

> **Remedy**: If you cannot create a project or item from the template in the 'Add New Project/Item dialog' in Visual Studio your template may not be supported by Visual Studio, and you are missing a vital extension or SDK required to open the specific project or file type in the template.
>
> 1. Install the SDK or extension that supports that project type
> 2. Restart Visual Studio
> 3. Clean and Rebuild the pattern toolkit solution.

- If none of these remedies resolve the problem please try the following last resort measures, and then contact Feedback with a detailed description of your issue.

> **Emergency Remedies**:
>
> 1. Shutdown Visual Studio, and delete all 'bin' and 'obj' sub-directories found in the directory of your solution.
> 2. Reset the Visual Studio Experimental Instance
> 3. Restart Visual Studio, clean and rebuild your toolkit solution.

Understanding: What are Text Templates?

See What are Text Templates for a broader overview of what Text Templates are and how they are used in pattern toolkits.

See Code Generation and Text Templates for references on how to create and configure them.

See Creating, Templatizing, Writing and Generating Code topics for how to create and integrate Text Templates into your pattern model.

How To: Creating a Text Template

A Text Template is the input artifact used for creating a code generator in a pattern toolkit. The template reads a data source (such as the pattern model), interprets the data in that model and generates some textual output. (i.e. code files or XML configuration, etc.)

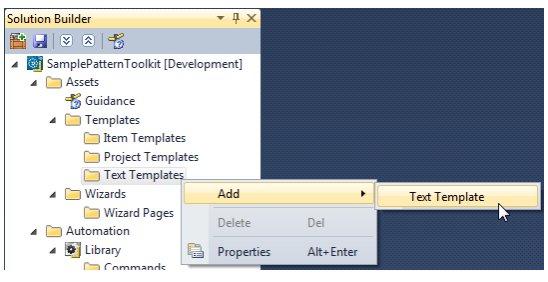
Process Overview

To create a T4 code generator for use in a pattern toolkit:

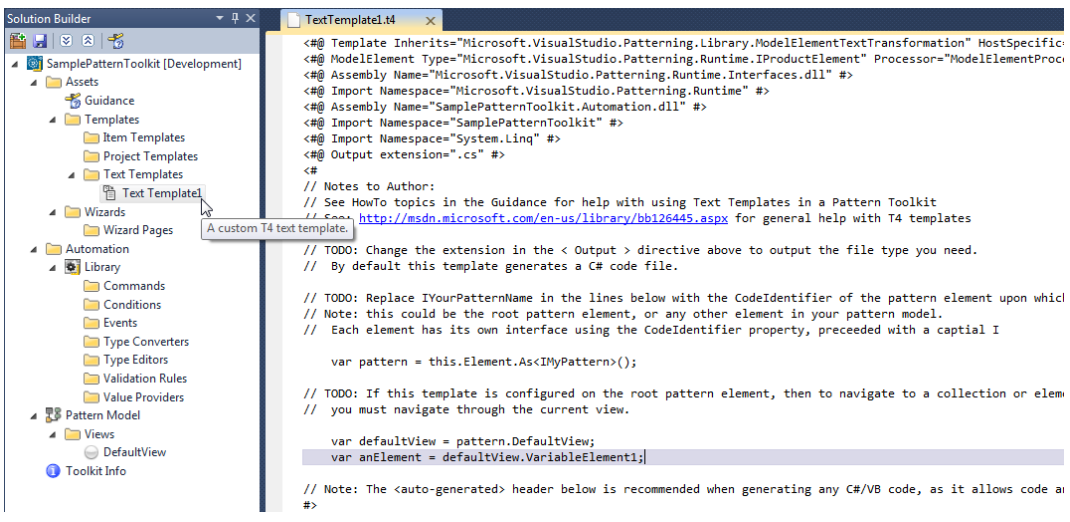
1. Create a new Text Template in the pattern toolkit
2. Harvest code from existing solutions and build-out the template content.
3. Automate the generation of the code in the pattern model.

Create the Template

To create a new text template into your toolkit project, right-click on the 'Text Templates' folder in 'Solution Builder', and select 'Add'.



Then double-click on the new element in 'Solution Builder' to open the template for editing.



Edit the Template

> **Tip**: Although you can manually edit the contents of a text template from scratch, for code that represents part of an established solution implementation of a pattern, it is strongly recommended that you use the process in Templatizing Existing Code to reduce manual errors and maintain consistency in your solution implementations.

See Writing Text Templates for more details on how to edit them to create code from a pattern model.

Automate the Template

See Generating Code for how to automate the creation of code from a Text Template in a pattern model.

How To: Templatizing Existing Code

Templatizing existing code files (i.e. code, configuration etc.) from existing solution is a practice that ensures that the generated code from text templates in a pattern models is accurate and consistent with that harvested from existing implementations of patterns. And is a quicker, less error prone, means to creating text templates rather than writing them from scratch.

> **Note**: Templatizing existing code also ensures that important whitespace in source files is preserved as originally intended.

Process Overview

To Templatize existing code, you:

1. Create a new text Template
2. Harvest existing code and paste into the text template
3. Substitute text in the template with values from properties of elements the pattern model.

Create the Template

See Creating a Text Template for details.

Set the extension of the Output directive to create the correct file extensions

```
<#@ Output extension=".cs" #>
```

Harvest Existing Code

From existing source/configuration files copy all the code and paste to the end of the text template.

Alternatively, you can also copy the source file itself into the pattern toolkit project, and rename its extension to *.t4, then copy and paste the standard text template header to the start of the file.

> **Note**: Text template files for pattern toolkits use the extension .t4 rather than .tt as these files are not self-generating in the pattern toolkit project. By default text templates with extension *.tt are automatically assigned a custom tool by Visual Studio that self-generates the code in the toolkit project whenever the file is changed.

Substitute Text

Once the text template can generate real content, it is now a just matter of substituting the variable parts of the code with calculated values from the data source of the text template.

In pattern toolkit generated code the data source for all text templates is the element upon which the text template is configured in the pattern model. From that element you can navigate to any other elements and read the state of all elements in the pattern model.

See Writing Text Templates for more details.

How To: Writing Text Templates

> **Tip**: By default, Visual Studio does not support color syntax coding of text templates. You are strongly recommended to obtain 3rd party tools (i.e. Visual T4) to edit text templates. These editors significantly increase productivity in templatizing code, and decrease the development cost and effort in creating and debugging text templates.

Writing text templates for pattern toolkits is no different than writing text templates in general except that the data source provided to them is the pattern model. All other aspects should be common to all text templates. See Code Generation and Text Templates for references to writing text templates in general.

> **Warning**: In pattern toolkits, text templates are executed in their own AppDomain, and in this AppDomain there is no programmatic access to the services of Visual Studio (i.e. IServiceProvider.GetService() will always return null for any service). For this reason alone, all data which the text template requires must be present as properties on elements in the pattern model. And to assist with this, all Value Providers on all properties in a pattern model are evaluated before code generation execution.

The following sections describe each part of the standard header that is present in each text template of a pattern toolkit. What they do, how they work and how they can be changed for individual text templates.

Template and ModelElement Elements

```
<#@ Template Inherits="NuPattern.Library.ModelElementTextTransformation" HostSpecific="True" Debug="True" #>
<#@ ModelElement Type="NuPattern.Runtime.IProductElement" Processor="ModelElementProcessor" #>
```

These directives are required.

All pattern toolkit text templates that require access to the pattern model must derive from the 'ModelElementTextTransformation' class, and use the 'ModelElementProcessor' processor. This inherited base class that gives access to the 'Element' property of the template, that then gives access to the pattern model in the template.

The 'ModelElement' directive determines the type of the element provided to the template. In the case above that type is the 'IProductElement' type, which all elements that support automation (i.e. pattern, collection and element) in a pattern model derive from.

> **Note**: In the 'Initializers' section below, the type is then cast to the specific type in the pattern model, to access it programmatically.

Assembly and Import Elements

```
<#@ Assembly Name="NuPattern.Runtime.Interfaces.dll" #>
<#@ Import Namespace="NuPattern.Runtime" #>
<#@ Assembly Name="SamplePatternToolkit.Automation.dll" #>
<#@ Import Namespace="SamplePatternToolkit" #>
<#@ Import Namespace="System.Linq" #>
```

These directives are required.

The first directive 'Assembly' directive is where the 'Template' base class exists. The second directive, 'Import' directive is the namespace for classes needed in the text template to navigate the pattern model.

The third and fourth directives, are the assembly and root namespace of the types in this specific pattern toolkit that are generated from the pattern model and allow the text template access to the specific types of the pattern model. The directives refer to the assembly and root namespace of the automation library project for the toolkit.

The fifth directive includes the commonly used 'System.Linq' namespace only to make the code in the text template be able to immediately use LINQ expressions.

> **Note**: You need to add additional 'Import' and 'Assembly' directives to any other types you use in the text template code.

Output Element

```
<#@ Output extension=".cs" #>
```

This directive is optional, and if not provided the generated file will default to having a .cs file extension.

> **Note**: The T4 code generation automation command that commonly is used to execute the text template allows you to override this directive by specifying the file name and extension in the command settings. However, if only a filename is provided with no extension then the extension from this directive is used.

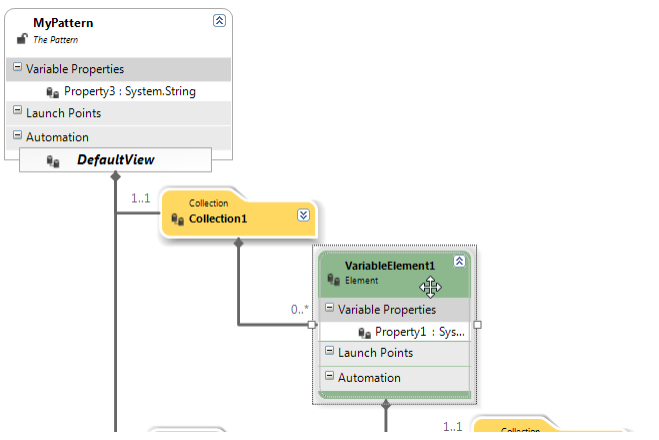
Initializers

```
<#
        var pattern = ((IProductElement)this.Element).As<IMyPattern>();
        var defaultView = pattern.DefaultView;
        var anElement = defaultView.VariableElement10;
#>
```

This code block is not required.

The data source for the text template is in the 'this.Element' property of the text template. This property is of the type defined in the second directive above (i.e. IProductElement). To access the specific elements of the pattern model in terms of their defined types (rather than through the generic IProductElement family of interfaces), then 'this.Element' should be cast (using the .As<T>() method) to the type of the element upon which this template is being generated from.

For Example



Given the example pattern model here, and given that we will configure the text template from the 'VariableElement1' element in that model, the following code would get access to the specific pattern model.

```
<#
        IVariableElement
element = ((IProductElement)this.Element).As<IVariableElement>();
        IDefaultView defaultView = element.Parent.Parent.DefaultView;
        IMyPattern pattern = defaultView.Parent;
#>
```

Code Generation Comment

```
//------------------------------------------------------------------------------
// <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>
//------------------------------------------------------------------------------
```

This code block is not required, but is strongly recommended if the file being generated is .NET Code, and you want to want to be compatible with static analysis tools that can ignore or process this code differently than hand-written code.

> **Note**: By default, this header presumes C# generated code. A similar header would be required for other languages like VB.NET.

Contents

See Code Generation and Text Templates for references to writing text templates in general.

How To: Generating Code from a Pattern

See Generating Code for the process of configuring automation to generate code from a Text Template in a pattern model.

Guidance

Understanding: What is Guidance?

See What is Guidance for the concepts of why guidance is such a fundamental and important part of pattern toolkit development.

How To: Adding Guidance to Your Pattern

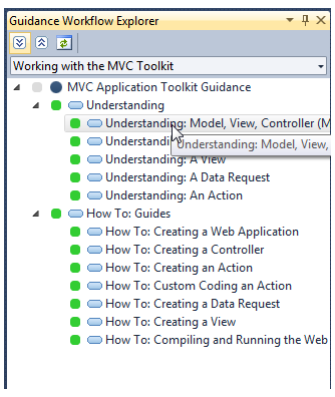
Every element in a 'Pattern Model' can have associated to it a 'Guidance Workflow'. This ability to provide guidance to every element in the pattern helps the users using the toolkit understand and be most productive with the toolkit.

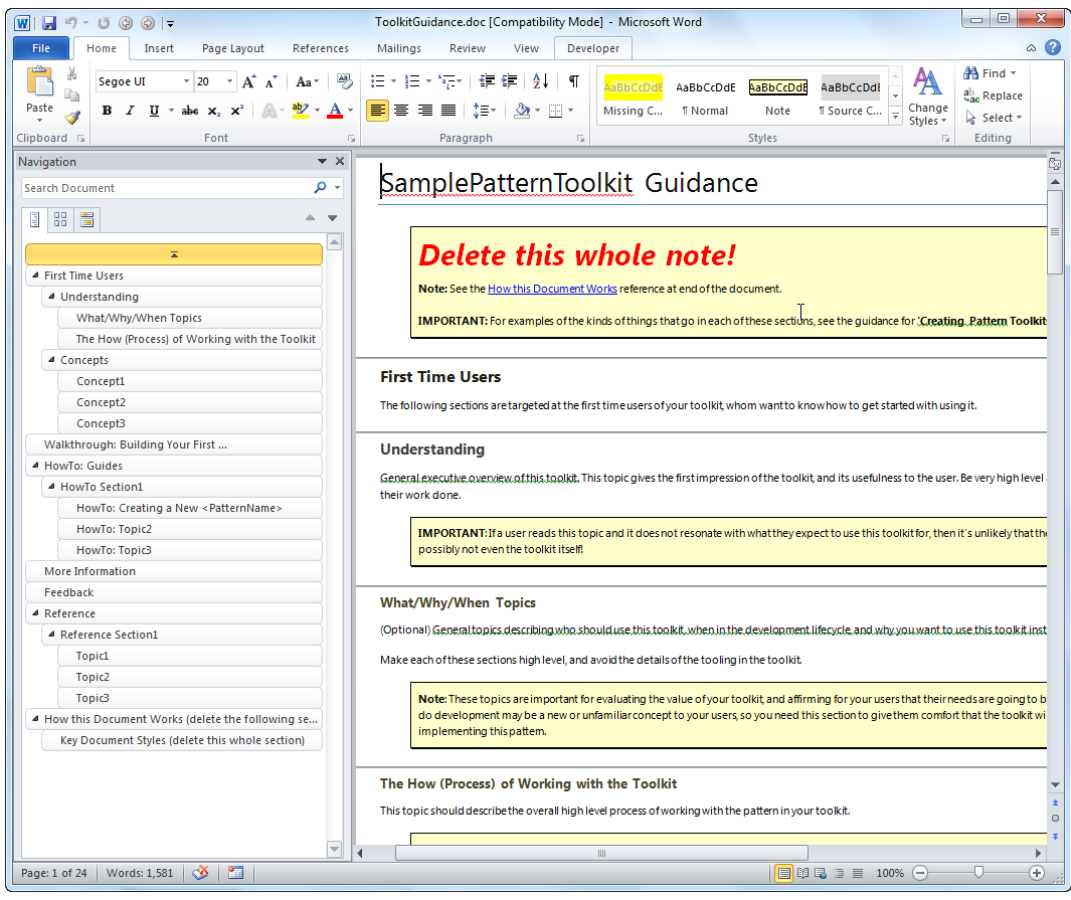
Typically, guidance can not only educate the users on what the toolkit does, and how, but will often provide prescriptive or instructional guidance on how to use the toolkit. In some cases, prescriptive instructional guidance is combined with state and automation to create a 'managed' sequence of 'smart' steps that each requires completion in some pre-defined order to get a task done. This is very useful for guiding the user very closely through difficult or complex tasks. The automation applied to them, can make calculations about what has or has not been done already, or automates parts of the process automatically for the user.

Every Pattern Toolkit project is equipped with a default Guidance Workflow intended to provide guidance for the toolkit as a whole. This specific Guidance Workflow is typically (but not always) associated to the topmost pattern element in the Pattern Model. It can in fact be associated to any, or indeed, all of the elements in the pattern model. And in addition, any element in the pattern model can be associated with any Guidance Workflow. The choice is very flexible.

Default Toolkit Guidance

The default Guidance Workflow provided in every toolkit project is generated automatically for you. The workflow of the guidance is a style of guidance that is akin to a standard document with a 'Table of Contents' that organizes its structure, and uses paging, multi-level headings, and links between pages. Best for understanding a set of organized, related topics.





It is constructed in a single Microsoft Word document, where you simply create new pages in the document each with a leveled heading and some content, and link the pages between each other using standard page hyperlinks. The result is single document containing all the guidance, which is easy to create, refactor and rearrange.

Once the document is saved, the Guidance Workflow can be 'built' from inferring the structure of the Word document. The Word document is automatically 'shredded' into individual guidance topics (individual *.MHT files), and a Guidance Workflow is automatically assembled to represent the same 'Table of Content' structure represented in the document.

The only thing that remains to do is associate the Guidance Workflow to the appropriate elements in the pattern model, and the guidance will be automatically provided when the toolkit is used.

> **Note:** There are others styles of guidance, such as more instructional step-by-step guidance that require state management and automation. See Add Additional Guidance to Your Pattern.

Creating the Guidance Document

See Editing Guidance, and Building Guidance for details on the creation process.

See Writing the Toolkit Guidance Document for tips on writing the Word document.

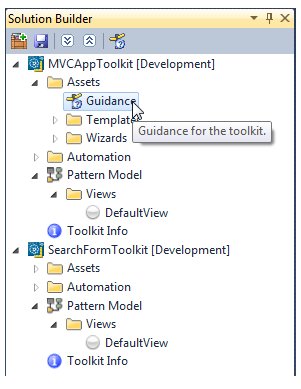
Associating Guidance to the Pattern Model

Once the guidance is built, guidance workflows can be associated to one or more element in the Pattern Model.

See Associating Guidance to elements in the pattern.

How To: Writing the Toolkit Guidance Document

These are tips on how to write the document of the default guidance for a toolkit. You should be aware of all the information below to create a successful document.



In the Pattern Toolkit project, double-click on the 'Guidance' element in 'Solution Builder'. This opens the guidance document in Microsoft Word.

> **Note**: You will need Microsoft Word 2007 or later to edit this document.

Creating guidance in this Word document is very straight forward and natural for technical writing and documenting guidance. Basically, you write the document like you would write any structured technical document, with structured headings, text, tables, pasted images etc.

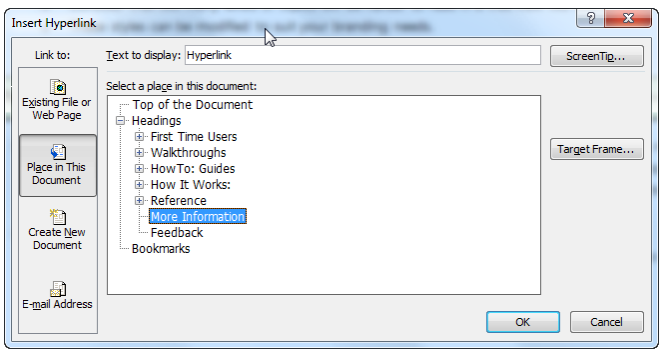
> **Tip**: Turn on the 'Document Map' or 'Navigation Pane' in Word, this presents the topics in the guidance document very similar to how it will look in the 'Guidance Workflow Explorer' window when your toolkit is used.

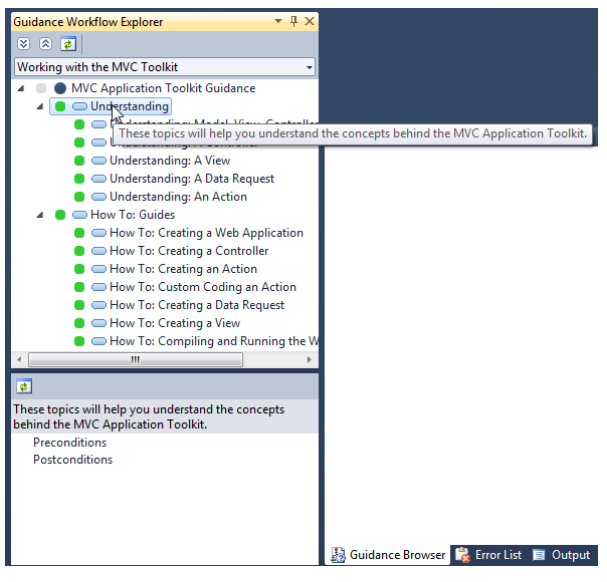
There are some constraints and processing rules that you need to be aware of to make effective use of the automatic generation of the guidance workflow.

1. The document must start with a Title paragraph, and followed by no other content.
2. The title must be followed immediately by a topic, using one of the 'Heading' styles (Heading 1 – Heading 4).
3. No 'Topic' exceeds a single page in length.
    - Content flowing over a single page will not be included in the guidance topic.
    - Pages in these documents are as large as they can physically be (22 inches tall x 11 inches wide).
    - Consider that content overflowing this limited volume will be difficult to read, and possibly very tedious, when presented to a reader in the 'Guidance Browser' tool window.
    - They can contain only web compatible content (i.e. text, tables, images, fonts, styles etc.).
    - Consider that all content is going to be transformed into single web archive (*.MHT) file.
    - Consider that all content will be resized according to the size of the 'Guidance Browser' tool window in Visual Studio, which is resizable in nature.

> **Tip**: Use the 'Web Layout' feature in Word to view how the content will look as a web page, and how the content changes when the window is resized.

4. Every 'Topic' starts with a single line 'Heading', using the provided styles from (Heading 1 to Heading 4) which all have page-breaks before them.
    - Heading 5, 6 and so on do not have page breaks before them and therefore do not result in individual topics.
        - These heading styles are most useful for splitting up the content within a topic.
    - Consider that more than about 4 levels of headings will be harder to read in a tree view like tool window in Visual Studio, as the content is indented far off to the right of the window.
5. A 'Topic' with only one sentence, or no content at is termed a 'Headline'
    - 'Headline' are useful for organizing and structuring the 'Topics', or as placeholders as the document evolves.
    - The first sentence of a headline is used to describe the node in the 'Guidance Explorer'. The description is used as a tooltip, and displayed in the description pane.
    - 'Headline' headings do not need to be unique throughout the document.
    - 'Topic' headings must be unique throughout the document, as individual files are created using the text in the heading.
6. Hyperlinks are used between topics, to aid in browsing the guidance.
    - When creating the Hyperlink to another topic in this document, insert the hyperlink and use the 'Place in this Document' tab to select the other heading to link to.





    - Only link to the heading of a Topic, links to the content of topics is invalid.
    - Don't link to a heading of a Topic, these links are also invalid.
    - Don't link to other bookmarks within the document.
    - You may link to any external site or URL.
7. Modify the document styles can to suit your branding needs.
    - Use styles to differentiate normal text from side notes or source code snippets.

Once you have completed writing the documentation, save and close the document, and proceed to Building Guidance.

How To: Adding Additional Guidance to Your Pattern Toolkit

Create the Guidance Workflow

At this time, you cannot create additional guidance workflows to your pattern toolkit.

Associate the Guidance

See Associating Guidance for details.

> **Important**: The 'Feature Id' for additional guidance workflows from separate feature extensions is found from the 'Id' attribute of the 'Identifier' element in the source.extension.vsixmanifest file of the feature extension project.
>
> 
>

Automation

Understanding: What is Automation?

What is Automation describes what automation is and how it works in pattern toolkits.

Applying Automation to Your Pattern describes the physical process of applying automation to your pattern model

See the other topics below for more details on the different types of automation classes.

What are Events?

- What are Events for what they are and how they work.
- Which Events to Use for which standard events are supported, and when to use them.
- Create Your Own Events for creating your own custom events.

What are Launch Points?

- What are Launch Points for what they are and how they work.
- Unfolding VS Templates for using a template Launch Point.
- Showing a Context Menu for using the Context Menu Launch Point.
- Executing a Command for using an Event Launch Point.
- Supporting Drag & Drop for using a Drag & Drop Launch Point.

What are Commands?

- What are Commands for what they are and how they work.
- Executing a Command for how to apply a command to your pattern model.
- Controlling When Commands are Executed for how to use conditions to make commands contextual.
- Creating Your Own Commands for creating your own custom commands.

What are Wizards?

- What are Wizards for what they are and how they work.
- Creating a Wizard for applying a wizard to your pattern model.
- Creating Your Own Wizards for creating your own custom wizards.

What are Conditions?

- What are Conditions for what they are and how they work.
- Controlling When Commands are Executed for how to use conditions to make commands contextual.
- Creating Your Own Conditions for creating your own custom conditions.

What are Value Providers?

- What are Value Providers for what they are and how they work.
- Creating your Own Value Providers for creating your own custom value providers.

What are Validation Rules?

- What are Validation Rules for what they are and how they work.
- Applying Validation to a Property or Element for how to apply a validation rule to a variable property.
- Executing Validation Rules for how to execute validation rules in your pattern model.
- Creating Your Own Validation Rules for creating your own custom rules.

Using & Creating Automation

- Automation Design Guidelines describes guidelines for using automation in your pattern model.
- Applying Automation to Your Pattern describes the physical process of applying automation to your pattern model
- Creating Your Own Custom Automation Classes gives you guidance on writing your own automation classes.

Understanding: Applying Automation to your Pattern

> **Note**: See What are Automation Extensions for more background on the process discussed below.

Most of the Automation provided for pattern toolkits works in two parts.

The first part is a Command of a given type (i.e. a code generation command). The command is then executed by the second part, a Launch Point of a given type, which typically responds to a gesture by the user or an event raised from the development environment (or from an external system).

> **Note**: Both the specific Command type and the Launch Point type will have their own specific set of settings that control their defined behavior.

A Launch Point detects the user gesture or event, may apply any pre-conditions to it to determine if it should execute and if so, first executes an optional Wizard to gather data from the user, and then executes the Command.

> **Note**: Commands and Wizards are optional, but either a Command and/or a Wizard are required to be configured for most (but not all) Launch Points.
> For example, a 'Template Launch Point' requires neither a Command nor a Wizard.

The Launch Point, Command and optional Wizard work together as the basic operational unit of any automated task.

In order to apply automation to your pattern, you:

- First determine which element in the pattern model you want to execute the automation on.

> **Note**: The element upon which the Command/Wizard and Launch Point are configured defines the initial context and data for the automation to execute. And from that element you can also reach any other element, and their properties, in the pattern model.

- Next, you determine what kind of automation to apply. The kind of automation determines the kind of Launch Point and Command to use.

> **For Example**: If you want to unfold a VS Project Template when the pattern is first instantiated, and you want that template to appear in the Add New Project dialog of Visual Studio – you simply configure a 'Template Launch Point'.
> **For Example**: If you want to generate a code file, from data in an element in your pattern, and whenever the solution is built - you configure a Command, with the 'GenerateCodeCommand' command type, and configure an 'Event Launch Point' to execute the command when it detects the 'OnBuildStarted' event of Visual Studio.
> See Which Automation to Use for more examples of common automation scenarios, and which combinations of Launch Points and Commands to use.

See Executing a Command for details on how to configure a Launch Points with Commands and Wizards.

How To: Automation Design Guidelines

Things to keep in mind when applying automation to your pattern model.

General

- Don't rely on the fact that any guidance provided in your toolkit will actually be read or followed initially, but ensure that topics are named accordingly and can be easily found when needed. See Adding Guidance to Your Pattern for more details.

> **Note**: Guidance is usually only discovered when problems with using your toolkit are encountered. Only then do users go in search of answers in the guidance!

- Avoid executing commands (and other automation) when the solution elements of the pattern model are in an invalid state, especially if the automation creates other artifacts or commits any changes to any data. Use conditions on the launch points that guard against invalid property values of solution elements of the pattern model. See Controlling When Commands are Executed for more details.
- For automation that is automatically triggered at key times, and that performs 'synchronization' or 'generation' kinds of tasks, consider in addition providing menus that can be run by a user manually. See Showing a Context Menu for more details.
- For the automation kinds in the previous point above, that also generally take a while to execute, consider providing boolean properties on those elements that control whether that automation is automatically executed or not. For example, a 'Generate On Build' boolean property that by default is true, but can be manually turned to false by a user to avoid the performance cost, when they are knowingly working on other 'non generation' areas with the toolkit.
- In custom automation classes (i.e. Commands, Conditions, Value Providers, Validation Rules, etc.) ensure your classes are well descried and categorized, and make extensive use of tracing for troubleshooting. See Creating Your Own Custom Automation Classes for more details.

Validation Rules

- Use validation rules for all property values to ensure the model is always configured correctly, so automation can run correctly. See Adding Validation to a Property for more details.
- Ensure that the violation message of a validation rule always tells the user which element is at fault, and guides the user in how to resolve the violation. Be very specific where possible. See Creating Your Own Validation Rules for more details.
- Configure when validation occurs. See Executing Validation Rules for more details. And avoid executing validation 'On Save' or when properties change unless absolutely necessary, as these events occur very often and may take some time to finish validating, and frustrate users.

Add From Existing?

- If elements of the pattern model represent artifacts or data in either the solution or other repositories (such as: on the hard disk, within databases, from remote services, etc.) then consider enabling the user to create new element instances from existing artifacts and data, either with Drag and Drop or with custom 'Import' kinds of automation. This ability also helps your pattern be integrated with existing solutions that may not have been created wholly by your pattern model. See Supporting Add from Existing Artifacts for more details.

Event Launch Points

- When configuring the 'Event Launch Point' with element specific events (i.e. those in the family starting with 'IOnElementXXX'), you must set the 'Current Element Only' flag to true, otherwise your automation will fire for every single instance of every element for that event. The 'Element Only' flag ensures you handle only these events for this specific instance. For all other events, set the 'Current Element Only' flag to false, otherwise the commands may never fire.

Code Generation

- See Unfold or Generate to determine when to use code generation or VS templates to get artifacts into the solution.
- Use a 'Command' with the 'GenerateProductCodeCommand' type and a launch point to generate a single file of code using a T4 template.
- Don't generate code until the parts of the pattern model which are used by the code generator T4 are valid. Use the 'ValidElementCondition' type on the launch point to ensure the pattern model is valid.
- Consider generating code when the solution is built. i.e. using an 'Event Launch Point' configured with the 'IOnBuildStarted' event type.
- Avoid code generation using an 'Event Launch Point' configured with either the 'IOnProductStoreSavedEvent' or 'IOnElementPropertyChangedEvent' event types. This event will fire too often, and code generation usually takes some time. So the user will experience frustration with unexpected delays when changing the properties of solution elements.

Unfolding VS Templates

- See Unfold or Generate to determine when to use code generation or VS templates to get artifacts into the solution.
- Use a 'Template Launch Point' only on the 'Pattern' shape. Use a 'Command' configured with the 'UnfoldVsTemplateCommand' type, and a Launch Point on a 'Collection' or an 'Element' shape.
- Don't use multiple project VS templates, instead unfold each one from a separate 'UnfoldVsTemplateCommand' command. You can use the same launch point to trigger the commands.
- If you have a 'Template Launch Point' defined on the 'Pattern' element, then ensure the icon used in that project or item VS template is the same value as the Icon property of the 'Pattern' shape.

See Which Automation to Use to understand better the provided automation classes or whether to build your own custom automation classes.

How To: Executing a Command

Commands are a basic executable unit of automation. Typically they are executed in response to events either from the toolkit user or from the development environment or systems the toolkit integrates with by Launch Points. It is the Launch Point that determines when to execute a command. A Launch Point can also display an optional wizard to capture data from the user before executing the command. All data required by the launch point and the command is fetched from the properties of the element upon which they are configured.

> **Note**: See Applying Automation to your Pattern for more details in understanding how automation works.

Overview

Once you know what command and launch point to use, the process of configuring them in your pattern model is straightforward.

> **Note**: See Which Automation to Use for examples of Launch Points and Commands you can configure to automate your toolkit.

Choose the element in your Pattern Model upon which the automation is to execute, then:

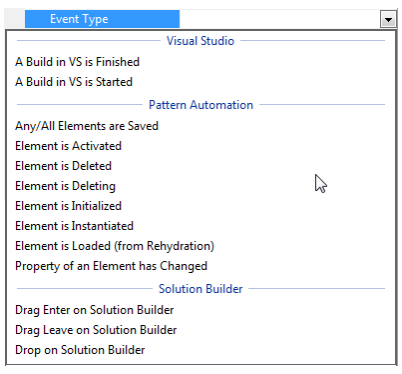
1. Add a Command and/or Wizard to the element in the Pattern Model.
2. Select the type of Command/Wizard you need, and Configure any Settings that type has.
3. Add a Launch Point to the element in the Pattern Model.
4. Select the Command and/or Wizard, and Configure any Conditions and other Settings that type has.

Once configured, you can build the toolkit and test its operation.

See the Hands-On-Lab for Creating Pattern Toolkits for a walkthrough of configuring a simple commands and launch points.

Understanding: Which Events to Use?

There are a number of events which can be used to trigger automation. The 'Event Launch Point' lists several when selecting the event which it will trigger the execution of its configured command.



Even though many of these events should be self-explanatory, some will be easily confused with their intended purpose. This topic is intended to direct you to using the correct event for common automation.

Important: Always use the 'OnElementInstantiatedEvent' (and not the 'OnElementCreatedEvent' or 'OnElementLoadedEvent') events when you want to trigger automation for when an element is first created, which is the most common case.
Warning: The 'OnElementCreatedEvent' should only be used in the cases where you want your automation to be triggered whenever the solution is opened, and the element is 'rehydrated' into 'Solution Builder', which is a rare case of automation.

Event Name	**Usage** (Use this event to trigger automation …)	**Notes**
A Build in VS is Started	As part of the build process	The two events respond to the global build commands in the solution. They require a solution to be opened.
A Build in VS is Finished	After the build process has complete	
Any/All Elements are Saved	(rare usage) After the pattern state file has been saved.	This event is raised whenever the pattern state file is saved, which can occur at any time during using and configuring a pattern instance in the solution builder window.
Element is Activated	When the element instance is double-clicked or ENTER key pressed when selected.	These events are raised for every instance of every element in the pattern model. Using these events requires that you also set the 'Current Element Only' property to true, otherwise the automation for your element will be called multiple times, once for each instance of the element.
Element is Deleted	After the element is deleted	
Element is Deleting	Before the element is deleted	
Element is Initialized	(rare usage) After an element is rehydrated.	
Element is Instantiated	(very common) After an element is first created.	
Element is Loaded	(rare usage) After an element is rehydrated.	
Property of an Element has Changed	After any property of the element has changed value.	
Drag Enter on Solution Builder	When any data is dragged over an element.	These events are raised during a drag and drop gesture over elements in the Solution Builder window.
Drag Leave on Solution Builder	When any data is dragged out of solution builder.	
Drop on Solution Builder	When any data is dropped on an element.	

Understanding: Which Automation to Use?

The automation framework that underpins commands and launch points is very flexible and highly extensible. Part of the intrinsic problem with being powerful is that it's not always immediately clear what automation techniques and which combinations of commands and launch points to use for any given scenario. For exactly this purpose, there are the following topics to help determine whether to configure built-in automation or create your own custom automation.

- The Provided Automation Types topic lists the out-of the box provided automation classes you can configure right-away on your pattern model
- If you need to create your own automation, see the Creating Custom Automation Classes for guidance on how to write your own custom automation classes.

How To: Controlling When Commands are Executed

You control when commands are executed typically with defining conditions on an automation extension that executes the command, i.e. launch points, such as the: Event Launch Point, Context Menu Launch Point and Drag Drop Launch Point etc.

These conditions must evaluate to true in order for the launch point to execute the command.

> **Note**: If a Wizard is also configured on the launch point, and the user cancels the wizard, then typically the command will not execute either.

Conditions for every launch point are configured differently depending on the launch point's function. See Configuring an Event Launch Point, Configuring a Menu Launch Point, and Configuring a Drag Drop Launch Point for more details.

How To: Executing a Sequence of Commands

You can execute a sequence of commands from a single launch point by using the 'Aggregator Command'.

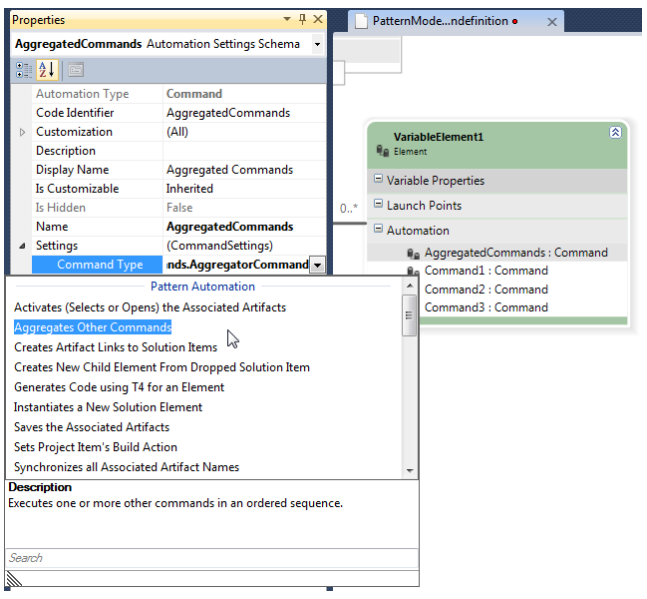
This command aggregates and executes an ordered list of commands configured on the same element.

Configure Your Command

Add new commands and Configure them on an element in the pattern model.

Create an Aggregator Command

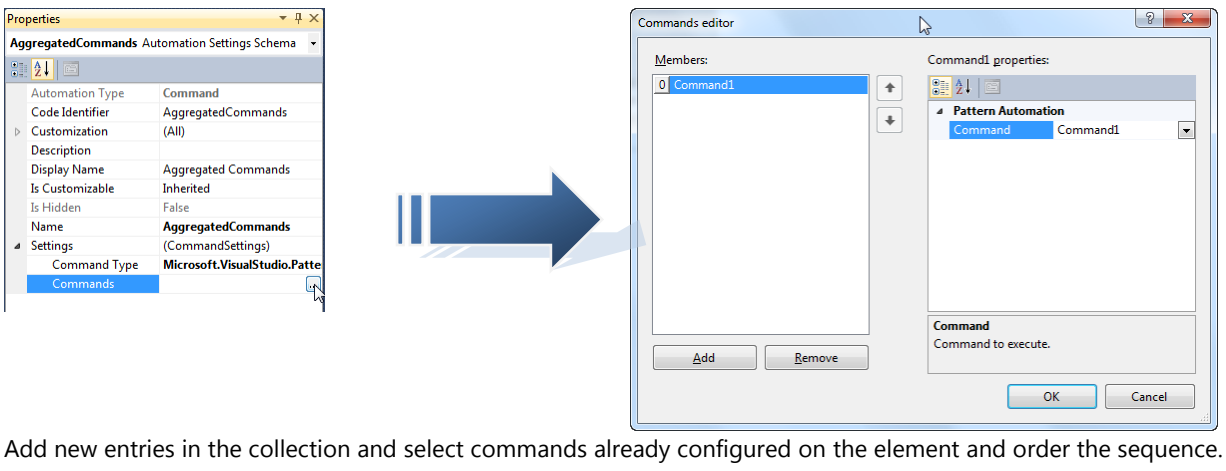
Add another command and Configure that command with the 'Command Aggregator' command type, on the same element.



Configure Aggregator

The command aggregator has a property called 'Commands' that you can edit to define the sequence of commands you want to execute.

Click the ellipsis to edit this collection.



Add new entries in the collection and select commands already configured on the element and order the sequence.

Configure Launch Point

Add a new Launch Point and Configure the Launch Point on the element, and set its 'Command' property with the aggregator command on the element.

Understanding: Wizards

See What are Wizards for more details on where they are used.

Wizards are the primary means to gather values of variable properties from a toolkit user. Their primary purpose is to present information to a user in a suitable format for the task at hand, with suitable UI aids and controls that manipulate that data efficiently.

They are typically displayed as a pre-step to running a command that automates the data gathered by the Wizard. In this role, they guide the user to configuring the required data to execute the command.

> **Note**: All data displayed in a wizard is usually bound to properties of an element, although Value Providers on those properties can be used to gather data from other sources.

A Wizard is simply a container for Wizard Pages. The wizard manages the sequence of pages.

The Wizard is simply a standard XAML window control with predefined, consistent looking 'chrome' that all wizards in toolkits assume. The wizard hosts one or more of the Wizard Pages, and has a title, shown in the title of the window.

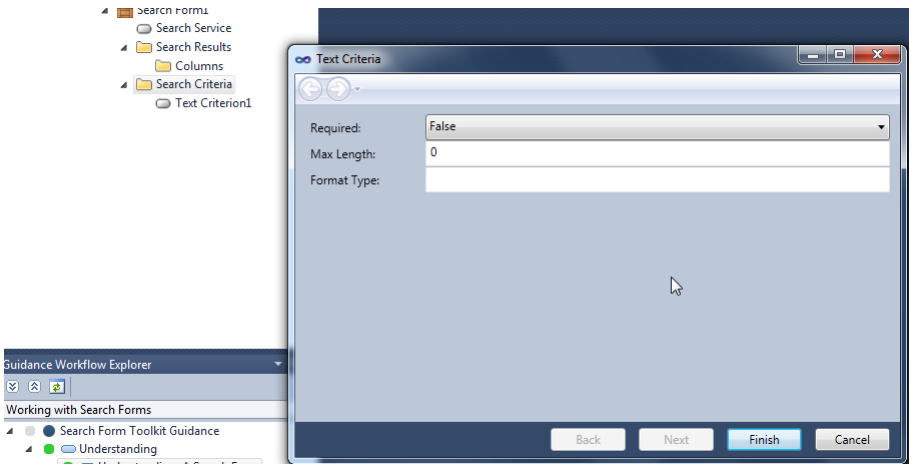
Each Wizard Page is a standard XAML page control that has a title and data-bound controls to the properties in an element.

Wizard Pages can use a built-in control called a <ValueEditor> control that displays the variable properties of an element using its defined Type Converter, Type Editor and descriptive attributes.

In order to build a wizard page, it is simply a matter of using the <ValueEditor> control for each property, arranged in a simple layout on the page. The Wizard Framework does all the heavy lifting of displaying the wizard, displaying the pages with the correct controls, managing navigation between pages, and validating the values entered into the controls.

Example Wizard

This is an example of a custom Wizard built for an example toolkit for editing 'Text Criterion' elements, showing the 3 variable properties of the 'Text Criterion' element, which have their own display formats and validation rules.



To create a Wizard, see Creating a Wizard for details.

How To: Creating a Wizard

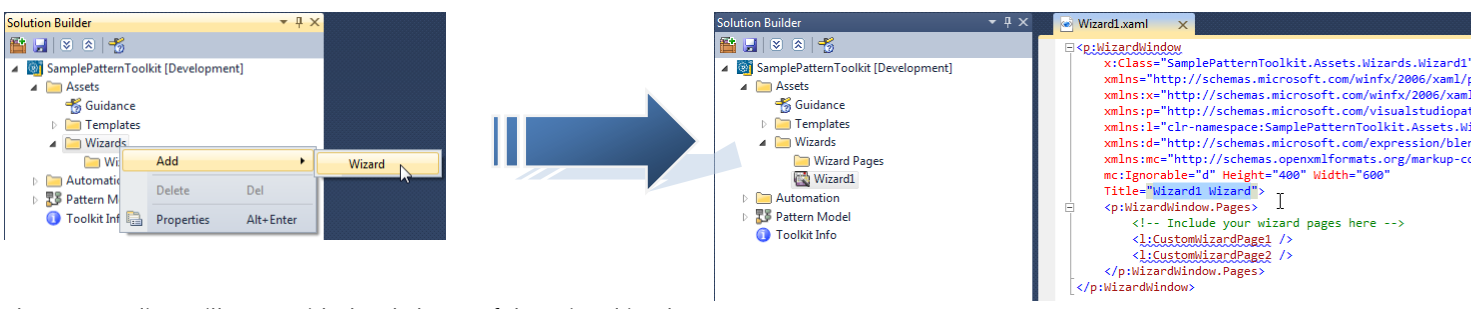
> **Recommend**: See Understanding Wizards for details on why and where they are implemented and used.
>
> **Note**: Wizards are created and configured to run on a specific element in a pattern model, but require the creation of a Wizard and one or more Wizard Page assets first.

The basic process for creating a Wizard is:

1. Create a Custom XAML Wizard
2. Create one or more XAML Wizard pages
3. Add a Wizard on an element
4. Configure the Wizard to be displayed

Create Custom Wizard (XAML)

To create a new 'Wizard', in the pattern toolkit project, right-click on the 'Wizards' folder in 'Solution Builder' window, and click 'Add'.



The XAML editor will open with the skeleton of the Wizard in place.

> **Note**: There is very little visual appearance to customize in the XAML for a Wizard, so maximize the 'XAML' view.

Updating the Title

Change the 'Title' property of the <WizardWindow> element, which will be displayed on the dialog of the wizard.

Adding Wizard Pages

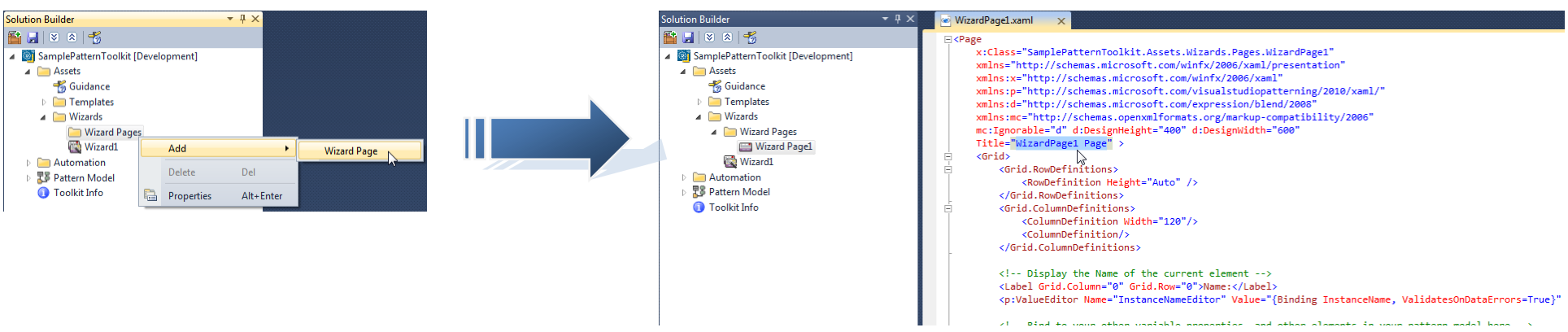
> **Note**: The <WizardWindow.Pages> collection will be populated after you create one or more Wizard Pages (below).

Descriptive Information

Use the naming guidelines in Naming Custom Automation Classes for configuring the descriptive properties of the wizard's class (*.cs in the code-behind), so that it is correctly documented, named, described and categorized.

Create Custom Wizard Pages (XAML)

To create a new 'Wizard Page', in the pattern toolkit project, right-click on the 'Wizard Pages' folder in 'Solution Builder' window, and click 'Add'.



The XAML editor will open with the skeleton of the Wizard Page in place (in 'Split' mode).

Updating the Page Title

You should now change the 'Title' property of the <Page> element, which is displayed on the wizard page.

Adding Controls

The 'Data Context' of the XAML page will be the current element from the pattern model upon which the Wizard will be configured, which gives access to data bind the properties of the current element.

Variable properties of elements in the pattern model already define appearance attributes for aiding the editing of the property in the 'Properties Window' of Visual Studio (i.e. IsReadOnly, Type Converters, and Type Editors etc.). For UI consistency, you can reuse those already defined attributes by using the <ValueEditor> control in the XAML of the Wizard Page.

The ValueEditor control will display the variable property value using its configured: Type Converter, Type Editor and other appearance attributes of the property automatically. You can even configure the ValueEditor to display error cues if invalid input is given, which use the configured validation rules of the variable property.

> **Note**: In the default Wizard Page that is created for you, controls are organized into a simple <Grid> control for convenience. You can use other layout controls as you see fit.

1. Add a <ValueEditor> control element to the page
2. Give it a unique value for the 'Name' property. The convention is to use the name of the variable property it is bound to, and suffix the word 'Editor'.
3. Set the 'Value' property binding to the 'Code Identifier' value of the variable property.
4. Set 'ValidatesOnDataErrors' = true, to honor the validation rules configured on the variable property.

For Example:

```
<p:ValueEditor Name="InstanceNameEditor" Value="{Binding InstanceName, ValidatesOnDataErrors=True}"/>
<p:ValueEditor Name="MyPropertyNameEditor" Value="{Binding MyProperty, ValidatesOnDataErrors=True}"/>
```

> **Note**: You are also free to use other controls bound to other variable properties or even collections of elements throughout the pattern model.
> (i.e. to showing lists of elements from the pattern model)

See XAML overview for WPF guide more information about using XAML to build more advanced wizard pages.

Adding Page to Wizard Sequence

The final step in creating a wizard page is to add the page to the wizard page sequence.

In the XAML for the Wizard (created in earlier step), ensure there is a namespace declaration to the Wizard Page class (by default it is the same namespace as the Wizard class), and add the type of the page to the <WizardWindow.Pages> element. For Example:

```
<p:WizardWindow
    x:Class="SamplePatternToolkit.Assets.Wizards.Wizard1"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    xmlns:p="http://schemas.microsoft.com/visualstudiopatterning/2010/xaml/"
    xmlns:l="clr-namespace:SamplePatternToolkit.Assets.Wizards.Pages"
    xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
    mc:Ignorable="d" Height="400" Width="600"
    Title="Wizard1 Wizard">
    <p:WizardWindow.Pages>
        <!-- Include your wizard pages here -->
        <l:WizardPage1 />
        <!--<l:CustomWizardPage2 />-->
    </p:WizardWindow.Pages>
</p:WizardWindow>
```

Add and Configure the Wizard

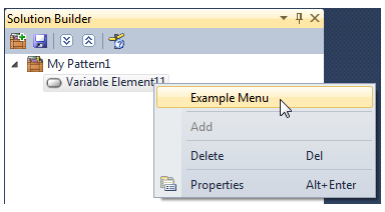
Once the XAML Wizard and Wizard Page assets are created, configure the automation to display the wizard on an element in the pattern model.

See Adding a Wizard and Configuring a Wizard.

How To: Showing a Context Menu

Context menus are configurable on all elements in the pattern model. They can display a 'Wizard' and/or execute a 'Command' when clicked (one or other, or both).

Menus support both a textual Caption and an Icon. The display of the menu is controlled by a collection of 'Conditions' that must evaluate successfully before the menu is displayed to the user. For Example:



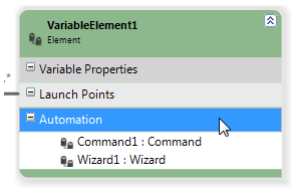
> **Note**: A context menu is provided by the 'Context Menu Launch Point'.

The basic process for creating a Context Menu is:

1. Add a Command, and optional Wizard to an element.
2. Add a Context menu Launch Point to the element
3. Configure the Launch Point

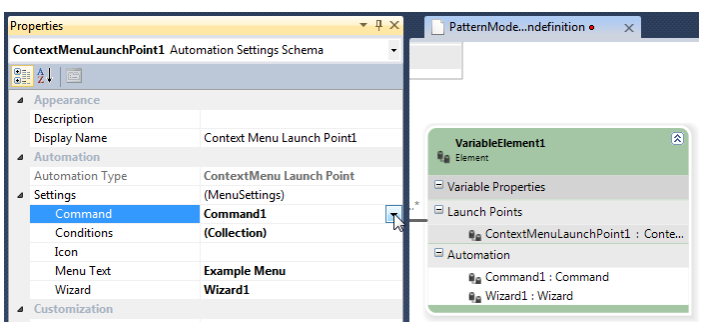
Add the Command and Wizard

In the pattern model, add a Command, and configure it. Optionally, add a Wizard and configure it.



Add and Configure the Launch Point

To the same element, add a Context Menu Launch Point and configure it with the configured command (and configured Wizard).



How To: Applying Validation to a Property or Element

See Validation Rules for more information.

All elements and all properties of elements support a collection of validation rules, used to ensure that the pattern model structure and data is correctly configured at all times by users.

> **Note**: Validation violations are displayed to toolkit users in the 'Error List' window.

A pattern toolkit author must configure when the validation rules are evaluated and errors displayed to users. See Executing Validation Rules for more details.

There are a number of built-in validations rules provided to all toolkits that can be configured for basic data type validation. For more complex validation scenarios simple custom validation rules are typically written. See Creating Your Own Validation Rules for more details.

Property Validations

You can configure one or more validation rules on any variable property of any element in the pattern model. These rules typically verify values manually entered by toolkit users or automatically calculated. These rules may range from basic data type checking (i.e. is this value a valid date?) through range and bound checking (i.e. is this date greater than today's date?) to contextual value checking (i.e. is this date now valid relative to the date of this other property given the current state of the applied pattern?).

Element Validations

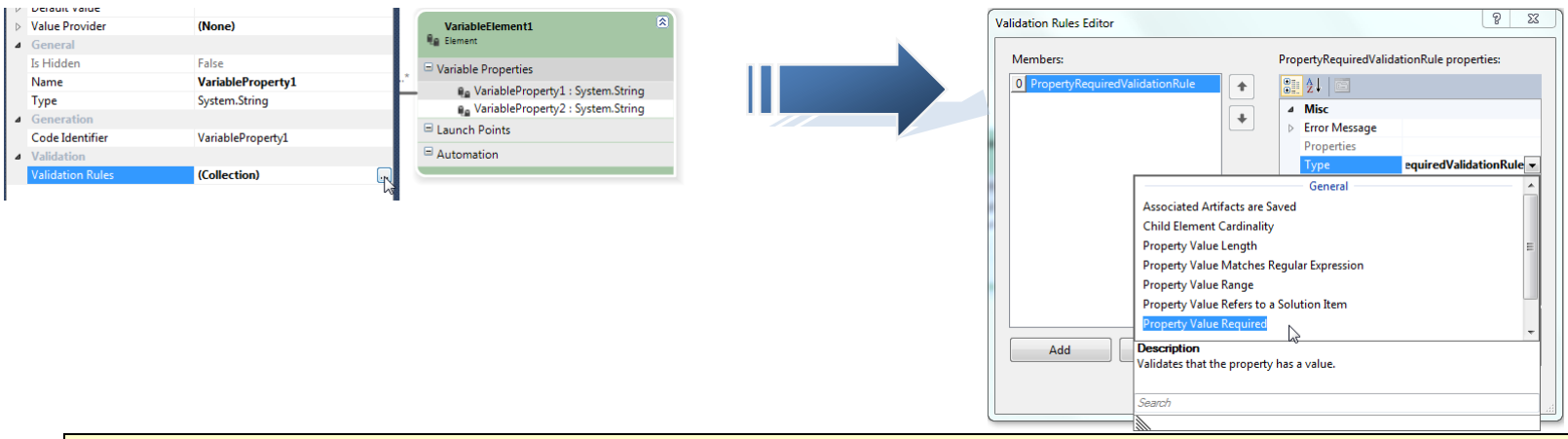
You can configure one or more validation rules on any element (itself) in the pattern model. These rules typically verify the structure of the applied pattern created by automation or toolkit users. These rules may range from simple cardinality checks (i.e. do I have more than two instances of a specific kind of child element?) through to structural integrity checks (i.e. has the user created and configured a specific subset of elements in the applied pattern?)

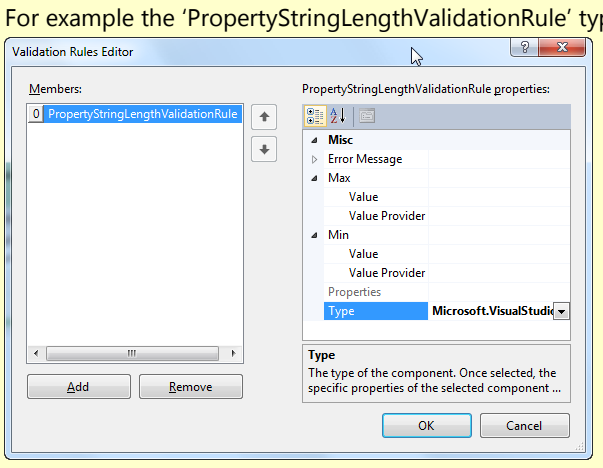
> **Note**: The order in which multiple defined validation rules on the same element or property run in is in-determinant. They must be able to run in any order, and at any time.

Configure a Validation Rule on a Property

To configure a validation rule on a variable property, in the pattern model select the property shape and click the 'ellipsis' on the 'Validation Rules' property.

This brings up the 'Validation Rules Editor', where you can add and configure one or more validation rules.



> **Note**: Each configured Validation Rule may require configuration of additional properties it provides.
> For example the 'PropertyStringLengthValidationRule' type requires a value for the 'Min' and a value for the 'Max' length allowable.
>
> 
>

Configure a Validation Rule on an Element

To configure a validation rule on an element, in the pattern model select the element shape and click the 'ellipsis' on the 'Validation Rules' property.

As above, this brings up the 'Validation Rules Editor', where you can add and configure one or more validation rules.

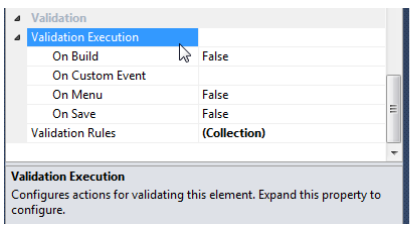
How To: Executing Validation Rules

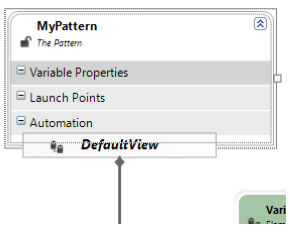
Once you have applied one or more validation rules to properties and elements of the pattern model, that validation is not automatically executed, and you must configure when to execute that validation.

> **Note**: By default, there are a number of validation rules that are built-in to verify things like cardinality that are executed independently from the validation rules a pattern toolkit author defines.

Configure Validation Execution

To configure when to execute validation in your pattern model, select the 'Pattern' element and configure the 'Validation Execution' properties.





On Build Execute validation when solution is built.

On Save Execute validation when pattern state is saved

> **Note**: Use sparingly. Saving the pattern state file occurs very regularly, (whenever a property value changes for example) and if your pattern model has lots of elements, validating on this setting can generate a noticeable performance issues for toolkit users.

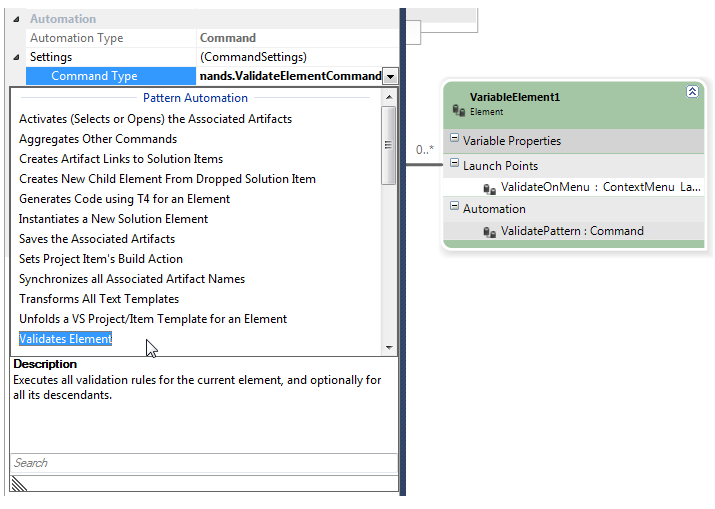
On Menu Provide a 'Validate All' menu on the root element, to manually validate the pattern.

On Custom Event Execute the validation on a custom selectable event.

> **Note**: By default, these properties need configuring manually by the pattern toolkit author. And when modified automatically configure (hidden) launch point and command automations to perform the requested actions. These automations are not intended to be modified directly, and are hidden to reduce clutter in the element. See showing hidden elements for more details.

(Optionally) Custom Validation Execution

If these validation execution options are not sufficient, or you require additional ones, you can configure additional automation on any element, using any launch point with the 'Validate Element' command to control when to execute validation.



Understanding: Unfold or Generate?

Use these guidelines to determine whether to use T4 code generation or VS templates for creating artifacts in the solution.

Unfold VS Templates

> **Note**: When you unfold a project or item using a VS template an artifact link is added to the current element instance to either the project or first item in an item template.

Use VS templates only when:

1. The content of the template has more than one file, or contains folders.
2. The content of the files in the template is text based or binary.
3. The content of the file does not require condition logic to determine the contents.
4. Calculates simple text substitutions using properties in the pattern model.
5. You require an entry in the Add New Project/Item dialog in Visual Studio. (Pattern element only)
6. Do not require updates to the content of items when the state of the pattern model changes.
7. Do not require changes to an items filename or extension.

> **Note**: A common scenario for using VS templates is unfolding a project into the solution, with a pre-determined folder structure, and pre-determined files (although there are many other possibilities).

See Unfolding VS Templates for details on how to unfold VS templates from your elements.

Code Generation

> **Note**: When you generate text using a T4 template an artifact link is added to the current element instance to the generated file. This link is reused for subsequent generations of the same file.

Use code generation only when:

1. The content of the file is text based.
2. The content of the file is variable, and requires condition logic to determine its contents.
3. Calculates simple or composite text substitutions using properties in the pattern model.
4. Require regeneration of the content of a file when the state of the pattern model changes.
5. Require a variable filename or extension of the file.

> **Note**: A common scenario for using Code Generation is generating a code or configuration file, with content (or filename) dependent on the current state of the pattern model (although many other possibilities).

See Generating Code for details on how to add a code generator to your elements.

How To: Unfolding VS Templates

> **Recommend:** See What are VS Templates for understanding how they can be used in pattern toolkits.
> **Recommend:** See VS Template Design Considerations for designing your VS templates effectively for pattern toolkits
> **Recommend**: Study the Unfold or Generate topic to determine if unfolding a VS template is the right fit for toolkit's your needs.

Acquire the Template

> **Important**: You must first acquire a VS template asset by either importing an existing one, or creating one from scratch.
> See Exporting, Importing, and Creating for creating VS template assets.

Configure the Automation

On an element in the pattern model from which you wish to unfold the VS template, configure the appropriate automation.

The kind of automation you configure depends on which element in the pattern model you unfold from.

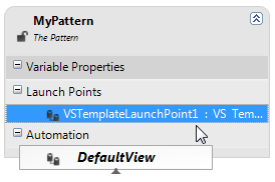
Unfolding from the Pattern Element

VS templates configured on the 'Pattern' element of a pattern model have the option to be configured to instantiate a new instance of the pattern when unfolded. When selected this option displays the template in the Add/New Project/Item dialog in Visual Studio. This capability provides a convenient segue between traditional project based development and pattern based development. There are several other unique options specific to templates unfolded from the 'Pattern' element.

Being that unfolding a VS template in Visual Studio is a specific gesture by a toolkit user, the 'Template Launch Point' is provided to configure the automation for a VS template on the pattern element.

> **Note**: These options and capabilities are not permitted for VS templates configured on other elements in the pattern model.

To configure a VS template to be unfolded from the 'Pattern' element of a pattern model, add a 'Template Launch Point' to the 'Pattern' element.



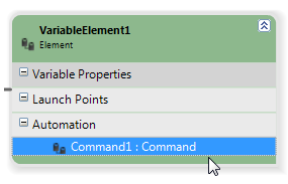
Configure the 'Settings' property of the 'Template Launch Point' to unfold the template asset you have imported or created, and define how it is presented to a toolkit user.

> **Note**: You can only add one 'Template Launch Point' per 'Pattern element.
> **Note**: If you want to unfold other VS templates from the 'Pattern' element, you can add additional commands using the 'Unfold VS Template' command type, and either execute these commands from the 'Template Launch Point', or from another launch point you define on the 'Pattern' element. See below for details.

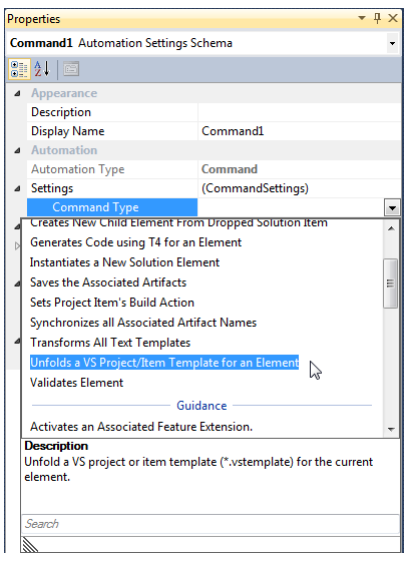
Unfolding from a Child Element

VS templates configured on child elements in the pattern model do not have the ability to create instances of those elements from the Add/New Project/Item dialog in Visual Studio. Were this the case, then the user adding or creating a project or item from that dialog has the additional complication of selecting an parent element *instance* in the 'Solution Builder' window to create a *new* instance of the element under, before unfolding the template. This complexity adds little value to the user, and makes for a complex user experience, compared to just adding a new element instance directly in the 'Solution Builder' window. For this reason, VS templates configured on child elements are hidden from the Add/New Project/Item dialog in Visual Studio.

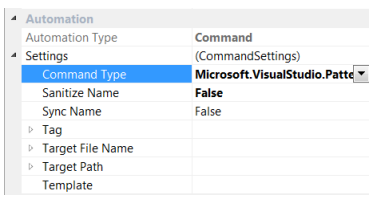
To configure VS template on the child elements of a pattern model, add a 'Command' to the child element.



Configure the 'Settings' property, and select the 'Unfold VS Template' command type.



Configure the following properties of the 'Unfold VS Template' command type:



Template — Select an existing *.vstemplate file from the assets in the toolkit project.

> **Note**: When you select the *.vstemplate file from the template picker dialog, the contents of the *.vstemplate file are automatically re-configured, and synchronized with the values of the current element. All content files of the template are also configured so that when the solution is built the VS template archive is automatically created and packaged in the toolkit.

Target File Name — Configure this to be the default name you wish to give the item or project being unfolded.

> **Note**: If left blank, the item or project being unfolded from the template will assume the actual name of the instance of the element.
> **Tip**: You can also define expressions in the value of the 'Target File Name' property to substitute property values in calculated names. (i.e. {InstanceName} or {Parent.PropertyName}. See the Expression Syntax for more details). Or you can use a Value Provider to calculate the name.

Target Path — Configure this to determine where in the solution the project or item will be unfolded.

> **Note**: If left blank, then the path defaults to '~', and will unfold into the solution item of the first ancestor element found, or if, none found then unfold to the solution. See the Target Path Syntax for more details.
> **Tip**: You can also define expressions in the value of the 'Target Path' property to substitute property values in calculated paths. (i.e. {InstanceName} or {Parent.PropertyName}. See the Expression Syntax for more details). Or you can use a Value Provider to calculate the path.

Sanitize Name — Configure this to remove spaces and other non-conventional characters from the name of the project or item unfolded from this template.

> **Note**: By default this is true. If false, then the project or item created from this template will be named with precisely the same name (including all spaces and non-conventional characters) as the name of the instance of the element. This is not normal convention with naming projects and files in a solution.

Sync Name — Configure this to synchronize changes in the name of the element with a change in the name of the project or item created.

> **Note:** This only has effect when the Target File Name property value includes property substitutions (i.e. {InstanceName} or {Parent.PropertyName}, etc. See the Expression Syntax for more details) or when the value is calculated by a value provider.
> **Note**: By default this is false. If true, then the project or item will be renamed when properties of the element instance in Solution Builder are changed.
> **Note**: For project artifacts, this command also synchronizes the 'AssemblyName' and the 'RootNamespace' properties of the project.
> **Note**: For C# code files, when 'Sync Name' is true, and the user changes the name in Solution Builder, a rename/refactor is automatically performed on the class in the code as well.

Tag — Configure this to add an arbitrary text value that will be used to tag the artifact link that is automatically created for the unfolded solution item.

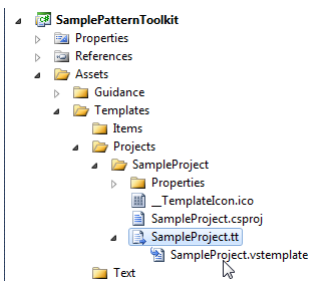
> **Note:** This tag value is an arbitrary textual value of any format.
> **Note:** This tag can be used by automation to filter the available artifact links on the instance of this element.

Lastly, add an appropriate launch point and configure the launch point to execute the command.

Understanding: Generating VS Templates from Text Templates

In some cases, it is desirable or even necessary to generate VS template (*.vstemplate) files from a text templates (*.tt files) in a pattern toolkit project.

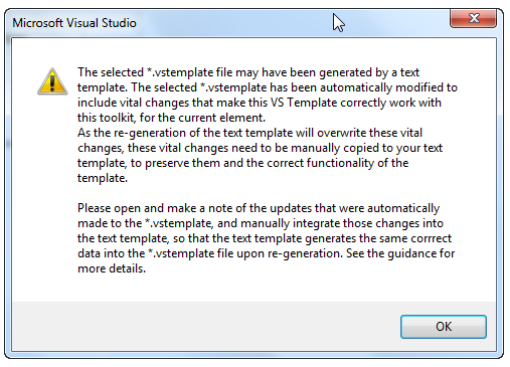
This scheme is setup manually by the pattern toolkit author.



In these cases, there are special considerations when configuring the automation to unfold the template.

When a VS template asset is configured by either a 'Template Launch Point' or by the 'Unfold VS Template' command type, the selected *.vstemplate file is modified automatically in several ways to ensure the template will be packaged and installed correctly by the toolkit, and the descriptive information in the template is synchronized with the element upon which it is configured.

However, if the *.vstemplate file is generated from a text template (*.tt file), then the automation cannot effectively update the *.tt content which generates the content of the *.vstemplate. The automation warns the user with this message.



> **Important**: The automation does perform all necessary updates to the content of the *.vstemplate file, but the toolkit author must manually determine all the changes and copy those changes back into the content of the *.tt in order for the *.vstemplate to be configured correctly next time the text template generates.

Updates to VS Template File

These are some of the types of modifications the automation performs on behalf of the toolkit author, depending on values configured in the automation for unfolding the VS template:

<TemplateData> Element

Updates the <Name>, <Description>, and <DefaultName> element to be synchronized with the current element that the template is configured upon.

Updates the <TemplateID> element to have a unique value, every time the template is selected, ensuring there is never two templates with same identifier deployed by any toolkit.

Updates the <Hidden> element for conditions when the templates must not be displayed in the Add/New Project/Item dialog of Visual Studio.

<WizardExtension> Elements

Multiple Wizard extension elements may be added to the template, some depending on the automation options, e.g.:

```
<WizardExtension>
  <Assembly>NuPattern.Library, PublicKeyToken=ecdd31353928a4a5</Assembly>
  <FullClassName>NuPattern.Library.Automation.ElementReplacementsWizard</FullClassName>
</WizardExtension>
<WizardExtension>
  <Assembly>NuPattern.Library, PublicKeyToken=ecdd31353928a4a5</Assembly>
  <FullClassName>NuPattern.Library.Automation.InstantiationTemplateWizard</FullClassName>
</WizardExtension>
```

Updates to Files in Solution Explorer

These are the modifications made to the VS template file (*.vstemplate) and files and folders included in the template.

Template File (.vstemplate)*

- Build Action: 'ProjectTemplate' or 'ItemTemplate'
- Copy Action: 'Do not Copy'
- Include In VSIX: False

Content Files

- Build Action: 'Content'
- Copy Action: 'Copy if Newer'
- Include In VSIX: False

How To: Generating Code

> **Recommend:** See What are Text Templates for understanding how they can be used in pattern toolkits.
> **Recommend**: Study the Unfold or Generate topic to determine if generating with a text template is the right fit for toolkit's your needs.

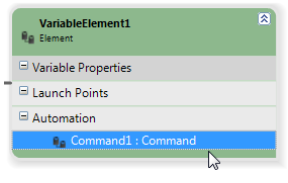
Acquire the Template

> **Important**: You must first acquire a text template asset by creating a new one from scratch.
> See Creating for creating text template assets.

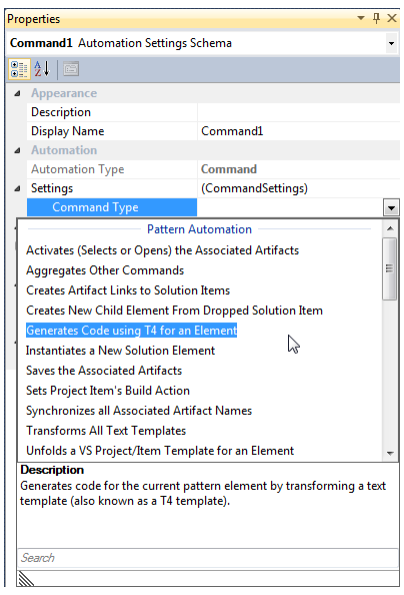
Configure the Automation

On an element in the pattern model from which you wish to generate with the template, configure the appropriate automation.

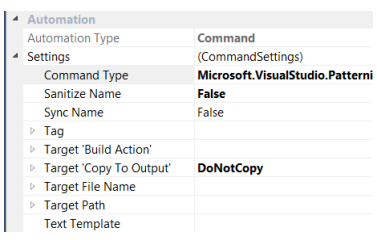
To configure a text template on any element in the pattern model, add a 'Command' to the element.



Configure the 'Settings' property, and select the 'Generate Code using T4' command type.



Configure the following properties of the 'Generate Code using T4' command type:



Text Template Select an existing *.t4 or *.tt file from the assets in the toolkit project.

Target File Name Configure this to be the name of the generated file or project.

> **Note**: Cannot be left blank. May include a file extension or not. If you define a file extension for this name, then this extension overrides the extension in the <#@ Output #> directive defined in the text template.
> **Tip**: You can also define expressions in the value of the 'Target File Name' property to substitute property values in calculated names. (i.e. {InstanceName} or {Parent.PropertyName}. See the Expression Syntax for more details). Or you can use a Value Provider to calculate the name.

Target Path Configure this to determine where in the solution the project or item will be generated.

> **Note**: If left blank, then the path defaults to '~', and will be generated into the first solution item of the current element or the first ancestor element found, or if, none found then generated to the solution. See the Target Path Syntax for more details.
> **Warning:** If left blank, or if this path begins with '~' then when the file is re-generated subsequent times then this path will be evaluated against the solution item of the current element, not a solution item of any ancestor element. To avoid this fully qualify the path with a relative path to the specific intended parent. (i.e. ..\..\~\etc.)
> **Tip**: You can also define expressions in the value of the 'Target Path' property to substitute property values in calculated paths. (i.e. {InstanceName} or {Parent.PropertyName}. See the Expression Syntax for more details). Or you can use a Value Provider to calculate the path.

Sanitize Name Configure this to remove spaces and other non-conventional characters from the name of the file generated from this template.

> **Note**: By default this is true. If false, then the file generated from this template will be named with precisely the same name (including all spaces and non-conventional characters) as the name of the instance of the element. This is not normal convention with naming files in a solution.

Sync Name Configure this to synchronize changes in the name or properties of the element instance with a change in the name of the file generated.

> **Note:** This only has effect when the Target File Name property value includes property substitutions (i.e. {InstanceName} or {Parent.PropertyName}, etc. See the Expression Syntax for more details) or when the value is calculated by a value provider.
> **Note**: By default this is false. When false, the filename is never changed from the name it was originally generated with. If true, then the generated file will be renamed whenever the file is re-generated.
> **Note**: For C# code files, when 'Sync Name' is true, and the user changes the name in Solution Builder, a rename/refactor is automatically performed on the class in the code as well.

Target 'Build Action' Configure this to change the 'Build Action' property of the generated file.

> **Note:** By default, the 'Build Action' will be determined by Visual Studio based on the file extension of the generated file.

Target 'Copy To Output' Configure this to change the 'Copy To Output Directory' property of the generated file.

> **Note:** By default, this value is 'Do not copy'.

Tag Configure this to add an arbitrary text value that will be used to tag the artifact link that is automatically created for the generated solution item.

> **Note:** This tag value is an arbitrary textual value of any format.
> **Note:** This tag can be used by automation to filter the available artifact links on the instance of this element.

Lastly, add an appropriate launch point and configure the launch point to execute the command.

Understanding: Recommended Code Generation Patterns

Writing T4 code generation templates can become complicated very quickly.

T4 is flexible, allowing you to mix control flow logic (like iterating over collections) with rendering and text replacement logic. The fact that you are writing code that generates code makes it even more difficult because when you look at the template it can be hard to tell the difference between the code that executes in the template and the code being generated.

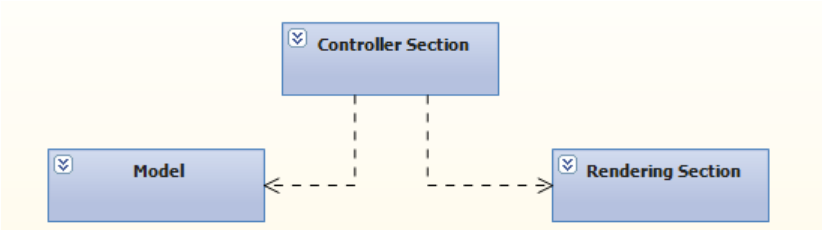
> **Note**: A T4 editor that has syntax coloring, intellisense, validation and preview is highly recommended, such as the 'Visual T4' editor.

Code generation is similar to rendering dynamic HTML pages, so it follows that patterns applied to writing server-side HTML rendering can also apply to rendering code. The purpose of applying a pattern to code generation templates is to separate concerns, thereby reducing the complexity of each concern.

Developing websites often utilize triad patterns like Model View Controller (MVC) or Model View Presenter (MVP). The model is the data used by the page, the view is purely presentation markup, and the controller (or presenter) contains the logic of the page. We can apply the same type of pattern to generating code from toolkits. The pattern instance is the model, and the T4 template is divided into a control section and a rendering section.

The T4 Triad Pattern

The T4 triad pattern involves dividing the T4 template into two sections: one for control flow called the 'Controller', and the other for rendering called the 'Renderer'.



The 'Controller' section strictly does not render any code. In other words, there are no calls to "WriteLine()" or markup sections like "<#= #>". The sole responsibility of this section is to control the logic of the template, calling into rendering methods of the 'Renderer' when rendering is required.

The 'Controller' section also isolates the 'Renderer' section from the 'Model' by passing only simple value types as parameters to the rendering methods.

> **Note**: The 'Controller' Section depends on 'Renderer' and the 'Model', but 'Renderer' and 'Model' have no dependency on 'Controller'.

The 'Renderer' section is strictly responsible for rendering text. It contains no control flow code, i.e. no "if" or "foreach" statements. It contains methods that accept only string or other simple value type parameters. Each method is very simple. Here is the example from above.

See Applying the T4 Triad Pattern for more details on how to apply this pattern to your text template.

How To: Applying the T4 Triad Pattern

> **Recommend**: See the previous topic Recommended Code Generation Patterns for details on what this pattern is.

Consider the following T4 text template example that uses a Triad pattern specifically designed for T4. (notes below)

```
<#@ 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="CodeGenerationPatternExampleToolkit.Automation.dll" #>
<#@ Import Namespace="CodeGenerationPatternExampleToolkit" #>
<#@ Import Namespace="System.Linq" #>
<#@ Output extension=".cs" #>
<#
       var currentElement = (IProductElement)this.Element;
       var thing = currentElement.As<IThing>();
       var patternRootNode = currentElement.Parent.Parent.Parent.As<IExample>();

       // This section contains controller code. It is responsible for the workflow of the template.
       // It does not directly render any text.  It calls render methods for rendering text.

       RenderCommentBlock();

       RenderClassStart(patternRootNode.RootNamespace, thing.InstanceName);

       foreach(IWidget widget in thing.Widgets)
       {
              RenderProperty(widget.InstanceName);
       }

       RenderClassEnd();
#>
<#+

// This section contains render methods. All methods in this section are responsible for rendering text.
// They do not contain any template logic. All parameters are string types.

void RenderCommentBlock()
{
#>//------------------------------------------------------------------------------
// <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>
//------------------------------------------------------------------------------
<#+
}

void RenderClassStart(string namespaceName, string className)
{

#>
namespace <#=namespaceName#>
{
       public partial class <#=className#>
       {
<#+
}

void RenderClassEnd()
{
#>     }
}<#+
}

void RenderProperty(string name)
{
#>            public virtual string <#=name#> { get; set; }

<#+
}#>
```

The ‘Controller’ section (Orange) of the example above is responsible for all the calculation and conditional logic.

> **Tip**: Do not put any rendering code in this section. It will be difficult to differentiate between the template code and the code being rendered by the template.
> For example, “namespace <#=namespaceName#>”.

The ‘Renderer’ section (Green) is responsible for writing all the textual output to the target file.

> **Tip**: Do not put “if” or calculation or conditional statements in this section, and do not pass anything but simple string parameters or other simple value types to the methods.
> These methods can be very simple, like “RenderClassEnd()” above, which only renders two closing curly braces.

The fact that rendering methods in the ‘Renderer’ section are very simple and granular makes it easier to call this code from the ‘Controller’ section and easier to understand and maintain.

How To: Getting a Root Namespace in Generated Code

All text templates in pattern toolkits are executed in a separate AppDomain, and in this AppDomain there is no access to the services of Visual Studio, and therefore any calls to IServiceProvider.GetService() will always return null.

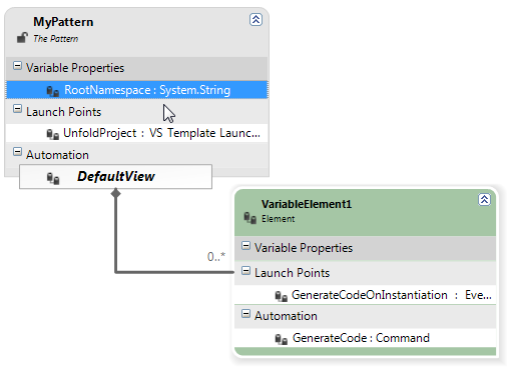
For this reason, all variable properties of all elements in the pattern model that have 'Value Providers' associated to them are evaluated and their evaluated values are cached in the property just prior to executing a text template. This allows the text template to access these properties to get their latest evaluated values.

In many cases, when generating .NET code from a text template, you need to generate that code into classes under the default root namespace of the target project. And therefore, need to obtain the current value of the root namespace of the project in Solution Explorer. Since, the AppDomain prevents you from getting access to any Visual Studio service from the T4 code, you need to use a different approach.

Create a Variable Property

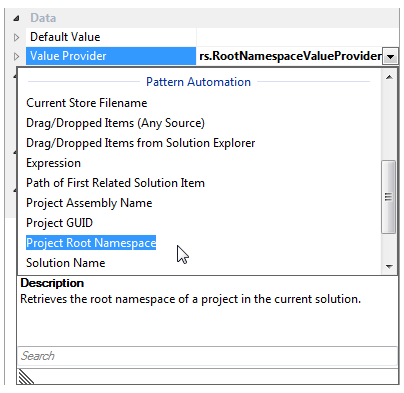
Create a variable property named 'RootNamespace' in your pattern model.

> **Note**: Typically, this property would exist on the element in the pattern model that either unfolded or generated the project in the first place, or has an existing artifact link to it.



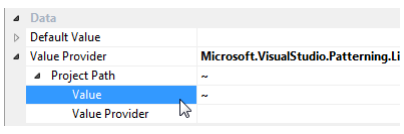
Configure the Value Provider

Configure the 'Value Provider' property to use the 'Project Root Namespace' provider.



This provider obtains the current value of the target project's root namespace when evaluated. The target project must be reachable from an element in the pattern model through an existing artifact link. Typically this element either has the artifact link, or a parent element has the link to the project.

Configure the 'Project Path' property to reference the element in the pattern model that has the artifact link to the project. See Target Path Syntax for more details on this value.



> **Note**: By default, this value is blank, but is equivalent to '~'. To mean, navigate up the pattern model from the current element to the first ancestor element that has an artifact link and traverse the link to the solution item it refers to.

Generate the Root Namespace

In the text template, simply access the 'RootNamespace' property of the element the property is configured upon, and output its value.

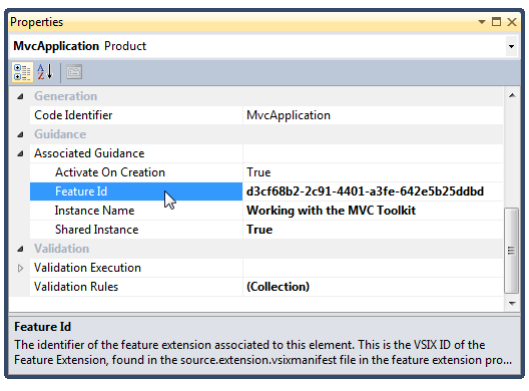
In the above example:

```
<#
    var element = ((IProductElement)this.Element).As<IVariableElement1>();
    var pattern = element.Parent().Parent();

#>
//------------------------------------------------------------------------------
// <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>
//------------------------------------------------------------------------------
namespace <#=pattern.RootNamespace #>
{

}
```

How To: Associating Guidance

In the 'Pattern Model Designer', select the element you want to associate guidance to.

In the properties of the element, expand the settings for the 'Associated Guidance' property.



The 'Associated Guidance' properties control which Guidance Workflow is associated to this element, and how that guidance is created.

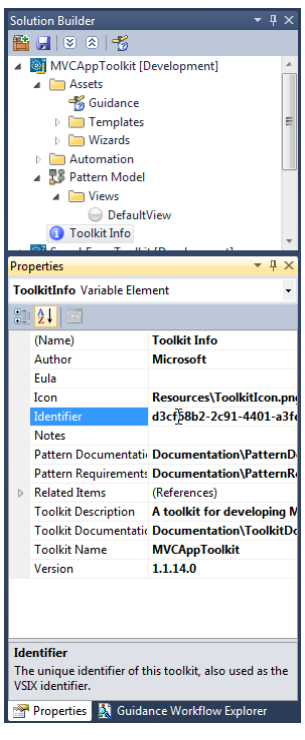
Select the Guidance Workflow to be Associated

It is possible to associate an element in the pattern model to any guidance workflow installed on the user's machine, and the 'Feature Id' property is the unique identifier of each Guidance Workflow. Usually this is a GUID, but can be a textual identifier also.

> **Note:** A pattern toolkit, only provides one guidance workflow by default, see Adding Additional Guidance to Your Pattern, for how to build additional Guidance Workflows.

Toolkit-Built Default Guidance

The 'Feature Id' for the default toolkit guidance is found from the 'Identifier' property in the 'Toolkit Info' element, in 'Solution Builder'.



Copy this value to associate the toolkit guidance to this element.

Additional Guidance

The 'Feature Id' for additional guidance workflows from separate feature extensions is found from the 'Id' attribute of the 'Identifier' element in the source.extension.vsixmanifest file of the feature extension project.

Copy this value to associate the additional guidance to this element.

(Optionally) Determine the Instancing of the Guidance

When guidance is associated to an element in a pattern model, an instance of a Guidance Workflow is created, and displayed in the 'Guidance Explorer' tool window.

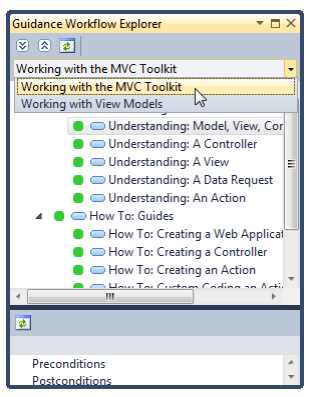
Guidance Workflows themselves can support state of their activities, meaning they need separate instances for each use of the workflow.

For example, if there was a Guidance Workflow for the process of creating a pattern toolkit project that remembered where you are in the process, creating a second pattern toolkit project would require a second instance of that workflow so it could individually record your progress with each project. These states should not be shared, but maintained individually. For guidance that does not maintain any state, one instance of that Guidance Workflow can be reused for all uses of it.

The toolkit provided guidance workflow is by default state-less, and therefore there will only ever need to one instance of this guidance workflow created for all uses of the pattern within it. The guidance workflow can be shared between them all.

Set the 'Shared Instance' property to false only when the associated guidance for this element requires individual instances of guidance workflows to be created for each instance of the associated element. Otherwise, setting to true, creates one and only one instance for all instances of the element.

(Optionally) Provide a Default Name for the Guidance



You can override the built-in default name of any Guidance Workflow by setting the 'Instance Name' property. This is useful for tailoring the guidance for the particular element it is associated with. Or for contextualizing multiple instances of the same Guidance Workflow, i.e. the name of the guidance instance can be tailored to include the name (or any property) of the associated element.

For example: If you want to have a separate instance of the guidance for each instance of the element to which it is associated, then you can contextualize the name the guidance instance with the name of the element instance, by setting the value of the 'Instance Name' property to an expression like: "Using this process for element {InstanceName}".

> **Note**: You can use any property of the current element in this expression.

The value of the 'Instance Name' property is what appears in the 'Guidance Explorer' drop down. If left blank, the default name of the Guidance Workflow is used.

(Optionally) Determine if the Guidance is to be Shown (Activated) when the element is first created.

You can decide whether the 'Guidance Explorer and 'Guidance Browser' tool windows are forced opened and the associated guidance is selected when the associated element is created, by setting the 'Activate On Creation' setting to true.

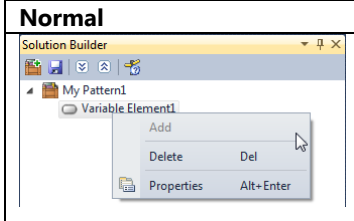
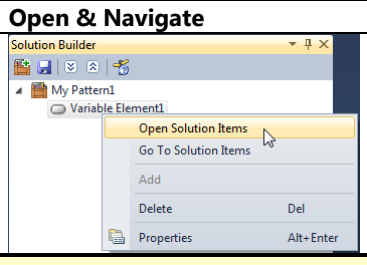
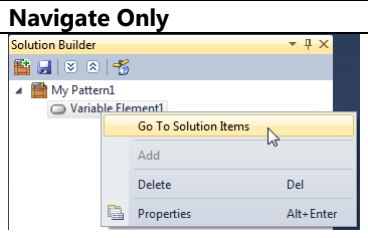
Setting this value to false, just means that the guidance tool windows are not forced opened, and the associated guidance is not presented to the user using the toolkit.

This is useful when your pattern has multiple elements, each with guidance associated to them, and you want to control which guidance workflow is displayed when those elements are first created.

How To: Enabling Solution Navigation

All elements which have associated one or more artifact links have the ability to navigate to those artifacts or open the artifacts in their default editors.

By settings some simple properties on elements in the pattern model, appropriate menus for the toolkit user are provided.

Normal	Open & Navigate	Navigate Only
		

> **Note**: The menus only show for elements that have artifact links that resolve to existing items in the solution. If no solution items are related at present, then the menus are not shown.

For some types of solution artifacts (i.e. code or configuration files, forms and controls) it makes sense to automate the opening of these items for modification when the toolkit user double-clicks on the element instance in the 'Solution Builder' window, or when they hit the ENTER key and it is selected.

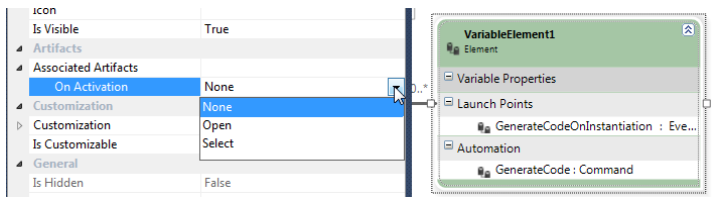
For some types of solution artifacts (i.e. projects, folders etc.) there is no default designer or editor, and it makes sense only to automate the ability to navigate to them in the solution, so the toolkit user might perform additional actions on them in the 'Solution Explorer' window.

The ability to control either opening or navigating to solution items is controlled by the 'Associated Artifacts' property on all elements in the pattern model. These properties only have relevance if the element will have artifacts links associated it, typically created by other automation configured on the element.

> **Note**: By default, these properties need configuring manually by the pattern toolkit author. And when modified automatically configure (hidden) launch point and command automations to perform the requested actions. These automations are not intended to be modified directly, and are hidden to reduce clutter in the element. See showing hidden elements for more details.

Turn On Navigation

To turn on automatic navigation for an element in the pattern model, configure the 'On Activation' property.



The 'OnActivation' property provides menus on the current element that either 'Select' the associated solution items, or 'Open' the associated solution items for editing.

> **Note**: this property has no effect on pattern elements that have no artifact links associated to them.

Understanding: Supporting Add from Existing Artifacts

For elements in pattern toolkits that are related to or integrated with others data, items, editors, tools in Visual Studio or with other external repositories or services such as the hard disk, or databases, web services etc. it makes for a better, more productive user experience to provide gestures to the user of the toolkit that enable them to create new instances of elements from existing data or artifacts.

Pattern Toolkits can support Importing of artifacts or data, or can support Drag and Drop of artifacts and data from any source. Once data or artifacts selected or dropped, a toolkit can create and configure properties of specific solution elements in an existing pattern model with either the dropped items themselves or with data within or derived from them. The specific mapping between dropped artifacts and elements in the pattern model is entirely arbitrary and pattern specific.

For example: by supporting drag and drop, a user can drag files from a disk onto solution elements in the pattern model, and as a result new elements are created for them that are automatically linked to those files. Or perhaps, a user can drag records from a database from another tool window in Visual Studio onto solution elements in the pattern model, and have data within those records parsed in and used to configure new or existing solution elements. In other cases where the mapping is not necessary so straightforward, an 'Import-like' command can be quickly created that asks the user in a Wizard to select a range of files from the disk, and enter some supporting text values. The 'Import' command then processes that data together and creates a number of new elements in the model. There are many possibilities and combinations.

See Supporting Drag & Drop for how to leverage this mechanism for dropping data dragged from any source onto solution elements in Solution Builder.

See Creating an Importing Command for how to leverage that mechanism for dynamically creating new elements in the model from existing data.

> **Tip:** In many cases supporting both import and drag & drop is necessary to have a rounded user experience with the toolkit, as drag and drop is harder to discover than import commands for users.

How To: Supporting Drag & Drop

Drag & Drop in the 'Solution Builder' window is supported to enable toolkit users to drag data or artifacts from other sources that the pattern represents or integrates with (i.e. windows, the hard disk, databases, external web services etc.). Dragging and dropping data from these sources allows a pattern author to create newly configured solution elements or configure existing solution elements configured with data and links from the dropped data. A common example of this is for importing data from other sources into the 'Solution Builder'.

> **Note:** At the present time, 'Solution Builder' only supports dragging from other sources and dropping on Solution Builder (i.e. from tool windows in Visual Studio such as 'Solution Explorer' and desktop applications such as 'Windows Explorer', etc.), but does not support dragging elements from 'Solution Builder'.
>
> **Note**: This is one strategy for Supporting Add from Existing Artifacts.

You can support drag & drop of any kind from any number of sources inside or even outside of Visual Studio (e.g. files from a hard disk) onto elements in Solution Builder using the 'Drag Drop Launch Point'

> **Note**: By definition, solution elements displayed in Solution Builder can be abstractions of zero or more related development artifacts within the scope of the current solution. Therefore, it is common that there will be more than a simple one-to-one relationship between solution elements and their related artifacts. (i.e. one specific C# source file being represented by one instance of a specific solution element). In fact, there is a many-to-many relationship between solution elements and their related artifacts.
> The pattern author and pattern model ultimately determine the specific relationships and abstraction between solution elements and related artifacts. Therefore, in general it is not possible to infer much information about the creation or configuration of any given solution element based upon what is dragged and dropped on it or its parent element. However, there are some common simple cases accommodated for such as dragging and dropping files from the solution, and dropping files from windows explorer.

The pattern author will be required to implement, at a minimum, a custom condition that determines whether the data being dragged is valid for the solution element being dropped upon, and a custom command must be implemented to configure or create the solution elements with any dragged data.

Create or Configure a Drop Command

Creating a Custom Drop Command

Create a custom command that will handle the data being dropped, and then may create or configure the necessary solution elements in response to the drop event.

> **Note**: This command will be called only after the dragged data has been accepted (by the Drag Condition) and dropped on an element in 'Solution Builder'.

This command must interrogate and extract the data being dropped, and perform whatever action is necessary on the pattern model in response to the drop.

See the topic Creating Your Own Drag Drop Commands for more details and options available for creating your own drag drop commands.

Configuring an existing Drop Command

There are a number of existing drop commands for common cases that can simply be configured in your pattern model. See the Provided Commands for more details.

These commands are used for processing dropped files in 'Solution Builder' that are dragged from either: the 'Solution Explorer' tool window in Visual Studio, or from the Windows desktop or from Windows Explorer. In all of these cases, when the files are dropped, they are linked automatically with the elements that they create.

For more complex cases, you must implement your own drop commands, which may use these provided command classes or their base classes to gain access to the data within these dropped files. See the topic Creating Your Own Drag Drop Commands for more details.

Create or Configure a Drag Condition

Creating a Custom Drag Condition

Create a custom condition that will evaluate whether the dragged data is valid and permitted to be dropped on the current element.

> **Note**: The condition will be called (on any instances of the element it is configured on) when data is dragged over it in the 'Solution Builder', and once again when the data is actually dropped, to ensure validity.

This condition must interrogate the dragged data and determine if it can be dropped on this element in the pattern model.

See the topic Creating Your Own Drag Conditions for more details and options available for creating your own drag conditions.

Configuring an existing Drag Condition

There are a number of existing drag conditions for common cases that can simply be configured in your pattern model. See the Provided Conditions for more details.

These conditions are used for processing dragged over 'Solution Builder' that are dragged from either: the 'Solution Explorer' tool window in Visual Studio, or from the Windows desktop or from Windows Explorer. In all of these cases, when the files are dragged they are filtered by file type.

For more complex cases, you must implement your own drag conditions, which may use these provided condition classes or their base classes to gain access to the data within these dragged files. See the topic Creating Your Own Drag Conditions for more details.

Configure the Launch Point

1. Make sure your solution is built and is up to date.
2. In your Pattern Model, add a 'Drag Drop Launch Point' to the element which supports the drag and drop.
3. Configure the Drag and Drop Launch Point

(Optional) Create a Wizard To Prepare/Process Dropped Data

Optionally, you may want to create a Wizard and present the user with it before processing the dragged data.

This wizard will be displayed before the command is executed after the data is dropped.

> **Note**: A wizard would be a good place to confirm the data being displayed or prepare the solution elements or organize the data before any solution elements are created or configured.

(Optional) Create a Value Provider for Status Text

Optionally, you may decide to create a custom Value Provider that provides dynamic 'Status Text' that is displayed when data is dragged over the current element, so you can contextualize the text with details in the data being dragged.

You must add the following property import to access the dropped data:

```
[Required]
[Import(AllowDefault = true)]
private DragEventArgs DragData { get; set; }
```

You must return a System.String value to be displayed as the text when data is dragged over the element.

How To: Creating Your Own Drag Drop Commands

A 'Drag Drop Command' is a custom command that creates new, or configures existing instances of solution elements in a pattern model from data provided to the command, usually from other sources the pattern represents or integrates with (i.e. the hard disk, databases, external web services etc.)

In general, a 'Drag Drop Command' will parse the dropped data from a source (i.e. dragged file from the hard disk), and then create new or configure existing solution elements in the pattern model with data derived from what is dropped.

> **Note**: This is one strategy for Supporting Add from Existing Artifacts.

This class of command can be unique because when it creates instances of new elements, those elements do not raise the usual 'OnElementInstantiated' event that triggers the typical 'creation' type automation to execute. Instead, these kinds of commands suppress those events, and create the elements 'silently'. This is necessary because, most of the automation that is triggered by the 'IOnElementInstantiated' event, not only presumes the element is brand new and not initialized, but in many cases executes automation (unfolding templates etc.) that creates newly related artifacts for the element. This automation must be suppressed if the artifacts are already provided by the drop function.

Create the Command

Create a custom command that will handle the data being dropped, and then may create or configure the necessary solution elements in response to the drop event.

You must add the following property import to access the dropped data:

```
[Required]
[Import(AllowDefault = true)]
private DragEventArgs DragData { get; set; }
```

> **Note:** As an alternative approach, you could define a property on your command, and then on the pattern model configure that property with a Value Provider (with the above Import) and returning the correct data in the form required to this command.

Typically, you would then process the dropped data by calling various methods on the DragData property to gain access to the dropped data, such as:

```
if (dragArgs.Data.GetDataPresent(DataFormats.FileDrop))
{
    var files = (dragArgs.Data.GetData(DataFormats.FileDrop) as string[];
}
```

Leveraging Built-In Drop Command Classes

Handling dropped data in your own custom commands tends to be very complex. There are several command base classes included in the Library that help you process dropped data. These base classes can be used to handle many of the common cases of dropped files and data, and can be adapted to handle many uncommon cases.

The following is a hierarchy of these classes and what their responsibilities are. Select the base class that suits your needs best.

namespace *NuPattern.Library.Commands*:

CreateElementFromItemCommand - This (abstract) command creates a new child element for each item that is provided to the GetItemIds() method.

> **Tip:** Derive a custom command from this class and override the GetItemIds() method to provide your own items, override GetElementNameFromItem() to provide a custom name for each new child element, and override InitializeCreatedElement() to initialize the properties of each new child element.

CreateElementFromDroppedItemCommand – This (abstract) command creates a new child element for each item that is obtained from any kind of dropped item.

> **Tip:** Derive a custom command from this class and override the GetItemIds() method to add your own items from the dropped data.

CreateElementFromFileCommand - This (abstract) command creates a new child element for each file that is provided to the GetFilePaths() method, and will add that file to the solution (AddFileToSolution()), and set up an artifact link on the newly created child element to the newly added file (GetItemInSolution()).

> **Tip:** Derive a custom command from this class and override the GetFilePaths() method to provide your own files, optionally override the AddFileToSolution() method to add the processed dropped files to the solution.

CreateElementFromDroppedFileCommand – This (abstract) command creates a new child element for each file obtained from dropped files dragged from 'Windows Explorer' matching the given file 'Extension'.

> **Tip:** Derive a custom command from this class and override the GetFilePaths() method to process the dropped files, optionally override the AddFileToSolution() method to add the processed dropped files to the solution.

CreateElementFromDroppedWindowsFileCommand – This command creates a new child element for each file obtained from dropped files dragged from 'Windows Explorer' matching the given file 'Extension', and will subsequently add those files to the solution given a 'Target Path'.

> **Tip:** Use and configure this class directly (no customization) in your pattern model to create new elements in your pattern model that adds the dropped files to the solution dragged from 'Windows Explorer' with a specific file 'Extension'.
> Derive a custom command class from this class and provide your own 'FileImporter' to process dropped files from 'Windows Explorer'.

CreateElementFromDroppedSolutionItemCommand – This command creates a new child element for each file obtained from dropped files dragged from 'Solution Explorer' matching the given file 'Extension'.

> **Tip:** Use and configure this class directly (no customization) in your pattern model to create new elements in your pattern model that add the dropped files to the solution dragged from 'Solution Explorer' with a specific file 'Extension'.
> Derive a custom command class from this class and provide your own 'FileImporter' to process dropped files from 'Solution Explorer'.

How To: Creating Your Own Drag Conditions

A 'Drag Condition' is a custom condition that determines whether dragged data is permitted to be dragged onto an element in the pattern model that originates from other sources the pattern represents or integrates with (i.e. the hard disk, databases, external web services etc.)

In general, a 'Drag Condition' will parse the dragged data from a source (i.e. dragged file from the hard disk), and then permit or deny the drag to an element in the pattern model.

> **Note**: This is one strategy for Supporting Add from Existing Artifacts.

Create the Condition

Create a custom condition that will handle the data being dragged, and determine its validity.

You must add the following property import to access the dragged data:

```
[Required]
[Import(AllowDefault = true)]
private DragEventArgs DragData { get; set; }
```

The 'Evaluate' method of your condition must return 'true' if the data can be dragged onto the current element and 'false' otherwise.

Typically, you would then process the dragged data by calling various methods on the DragData property to gain access to the dragged data, such as:

```
if (dragArgs.Data.GetDataPresent(DataFormats.FileDrop))
{
    var files = (dragArgs.Data.GetData(DataFormats.FileDrop) as string[];
}
```

Leveraging Built-In Drag Condition Classes

Handling dragged data in your own custom condition tends to be very complex. There are several condition base classes included in the Library that help you process dragged data. These base classes can be used to handle many of the common cases of dragged files and data, and can be adapted to handle many uncommon cases.

The following is a hierarchy of these classes and what their responsibilities are. Select the base class that suits your needs best.

namespace *NuPattern.Library.Conditions:*

DropItemCondition - This (abstract) condition evaluates whether any items are obtained from dragged data, provided to the GetDraggedItems() method.

> **Tip:** Derive a custom condition from this class and override the GetDraggedItems() method to evaluate your own items.

DropFileCondition – This condition evaluates whether any files obtained from dragged data, dragged from 'Windows Explorer', match the given file 'Extension'.

> **Tip:** Use and configure this class directly (no customization) in your pattern model to evaluate dragged files from 'Windows Explorer' with a specific file 'Extension'.
> Derive a custom condition class from this class and override GetDraggedFiles() method to evaluate dragged files from 'Windows Explorer'.

DropSolutionItemCondition – This condition evaluates whether any files obtained from dragged data, dragged from 'Solution Explorer', match the given file 'Extension'.

> **Tip:** Use and configure this class directly (no customization) in your pattern model to evaluate dragged files from 'Solution Explorer' with a specific file 'Extension'.
> Derive a custom condition class from this class and override GetDraggedFiles() method to evaluate dragged files from 'Solution Explorer'.

DropItemFormatCondition – This condition evaluates whether each item obtained from dragged data matches a given data 'Format'

> **Tip:** Use and configure this class directly (no customization) in your pattern model to evaluate dragged data with a specific data 'Format'.

How To: Creating an Importing Command

An 'Importing Command' is a custom command that creates new or configures existing instances of solution elements in a pattern model from data provided to the command, usually from other sources the pattern represents or integrates with (i.e. the hard disk, databases, external web services etc.)

In general, an 'Importing Command' will prompt the user to select or provide data from one or more sources, and then create new or configure existing solution elements in the pattern model with that 'imported' data.

> **Note**: This is one strategy for Supporting Add from Existing Artifacts.

This class of command is unique because when it creates instances of new elements, those elements should not raise the usual 'OnElementInstantiated' event that triggers the typical 'creation' type automation to execute. Instead, these kinds of commands should suppress those events, and create the elements 'silently'. This is necessary because, most of the automation that is triggered by the 'IOnElementInstantiated' event, not only presumes the element is brand new and not initialized, but in many cases executes automation (unfolding templates etc.) that creates newly related artifacts for the element. This automation must be suppressed if the artifacts are already provided and initialized by the import command.

The automation process for an Import command typically goes like this:

1. A Context Menu, (or other Launch Point) is provided on some parent element in the pattern to execute an Import.
2. When invoked, an 'Import' wizard may be displayed, and the user is guided to select, sort, filter values to be imported.
3. That data is persisted into properties in the elements of the model (typically hidden properties).
4. The 'Import' command then executes and reads the data from the model, and creates new, or configures existing, solution elements in the pattern model.
5. The data remains persisted in the model, should a wizard be run again, and be used as the next default settings for the wizard.

Create the Command

Create a custom command that will handle the data being imported, and then may create or configure the necessary solution elements.

The only difference for implementing an 'Import'-kind of command is how it would create new instances of solution elements in the model.

> **Note**: Configuring existing solution elements (i.e. setting their properties) in the model is no different than other commands.

When creating new instances of solution elements in the pattern model, the command will call the following methods on the current element (IProductElement) such as:

```
ICollection CreateCollection(Action<ICollection> initializer = null, bool raiseInstantiateEvents = true);
IElement CreateElement(Action<IElement> initializer = null, bool raiseInstantiateEvents = true);
IProduct CreateExtension(Action<IProduct> initializer = null, bool raiseInstantiateEvents = true);
```

- Where the first parameter is an optional initializer used to set initial properties on the created element/collection/extension.
- But, where the second parameter **must be false**, to ensure that that instantiation events are not raised.

> **Note**: In regular commands that create new solution elements, the second parameter is typically omitted or left true, so that the correct instantiation events are fired.

Now, when the Import command is run, it creates new instances of solution elements using these methods, setting their initial (imported) values in the initializer, and setting the second parameter as false.

Leveraging Built-In Import Command Classes

Handling the import of data in your own custom commands can be very complex. There are several command base classes included in the Library that help you importing certain kinds of data. These base classes can be used to handle many of the common cases of importing files, and can be adapted to handle many uncommon cases.

The following is a hierarchy of these classes and what their responsibilities are. Select the base class that suits your needs best.

namespace *NuPattern.Library.Commands*:

CreateElementFromItemCommand - This (abstract) command creates a new child element for each item that is provided to the GetItemIds() method.

> **Tip:** Derive a custom command from this class and override the GetFileIds() method to provide your own items.

CreateElementFromFileCommand - This (abstract) command creates a new child element for each file that is provided to the GetFilePaths() method, and will add that file to the solution (AddFileToSolution()), and set up an artifact link on the newly created child element to the newly added file (GetItemInSolution()).

> **Tip:** Derive a custom command from this class and override the GetFilePaths() method to provide your own files, optionally override the AddFileToSolution() method to add the processed dropped files to the solution.

CreateElementFromPickedFileCommand – This (abstract) command creates a new child element for each file selected from 'Windows Explorer' matching the given file 'Extension'.

> **Tip:** Derive a custom command from this class and override the GetFilePaths() method to provide your own files, optionally override the AddFileToSolution() method to add your files to the solution.

CreateElementFromPickedWindowsFileCommand – This command creates a new child element for each file selected from 'Windows Explorer' matching the given file 'Extension', and will subsequently add those files to the solution given a 'Target Path'.

> **Tip:** Use and configure this class directly (no customization) in your pattern model to create new elements in your pattern model that adds files to the solution selected from 'Windows Explorer' with a specific file 'Extension'.
> Derive a custom command class from this class and provide your own 'FileImporter' to process files selected from 'Windows Explorer'.

Configure the Command

The process for configuring an 'Import'-kind of command is exactly the same as for that of creating any kind of command, with optional wizard and a launch point.

See Adding a Command and Configuring a Command.

How To: Creating Your Own Custom Automation Classes

Configure Built-In or Create Custom?

NuPattern includes a number of built-in automation classes that you can configure right away in your pattern models.

- The Provided Automation topic lists those classes and what they are used for. The Common Automation Scenarios topic lists the common automation scenarios that use the Provided Automation in the toolkit.

There are also a number of other topics in this section that are discussed separately to help you configure certain kinds of specific automation:

- Associating Guidance to elements in your pattern
- Enabling Solution Navigation from elements in your pattern
- Executing Validation on elements in your pattern model.
- Unfolding VS Templates into the solution
- Generating Code into the solution
- Supporting Drag & Drop from any source

> **Note**: If neither the provided automation, nor the topics above address your needs you may need to implement your own custom automation classes specific to your pattern toolkit and the domain it represents.

Writing Custom Automation

Writing custom automation classes (i.e. Commands, Conditions, Value Providers, Validation Rules etc.) is a common activity in pattern toolkit development that requires you to custom code specific classes in your toolkit. These classes, once built, will then be available to be configured on your pattern model the same as the provided automation classes.

The following topics guide you to developing your custom automation:

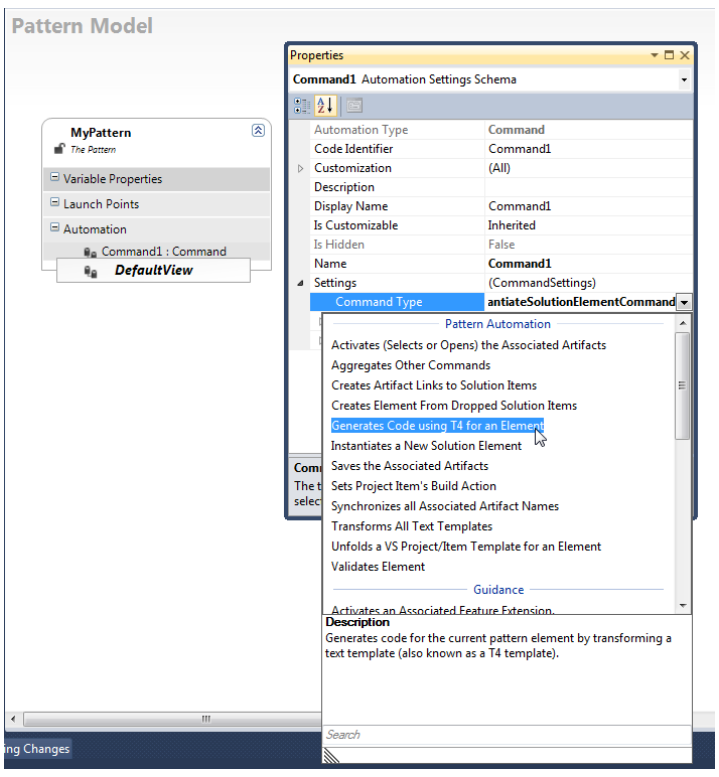
- Coding Your Own Automation Classes – guidelines for coding your automation classes
- Naming Your Own Automation Classes – naming and description guidelines for your automation classes
- Implementing Your Automation Own Automation Class – how to implement the functionality of your automation classes to work correctly in pattern models.
- Applying Tracing to Your Own Automation Classes – how to make use of the tracing facility for troubleshooting your classes.
- Common Automation Tasks – references to libraries and classes that can help you custom code your automation more efficiently.

How To: Naming Your Own Automation Classes

When creating custom automation classes such as: Commands, Conditions, Value Providers, Validation Rules, Wizards and Wizard Pages etc. in your Automation Libraries, you should use the following naming conventions to ensure maximum discoverability and re-usability of them in pattern toolkits.

All Automation Classes

Every automation class uses metadata description attributes (on the C# or VB automation class) which are used for displaying the custom type to a toolkit author, when the author browses the type from the list of types when configuring their automation on the pattern model.



For consistency, use the following guidelines for setting the [DisplayName], [Description] and [Category] attributes in the attributes of the automation class.

```
/// <summary>
/// A custom automation class.
/// </summary>
[DisplayName("SampleClass ")]
[Category("General")]
[Description("Provides some automation.")]
[CLSCompliant(false)]
public class SampleClass : SomeAutomationBaseClass
```

	Display Name	Description	Category
Commands	Should reflect the action it performs. e.g. “Transforms All Text Templates” or “Generates Code”, or “Validates the Model”	Describe the action it performs, with its parameters.	If this class is a general reusable automation class, then the [Category] should reflect the generalized area of automation (e.g. ‘General’, ‘Visual Studio’, ‘Pattern Automation’, ‘UI’, etc.) If this class is more specific to a specific pattern toolkit, then the [Category] should reflect the toolkit itself. (e.g. ‘MVC Automation’)
Conditions	Should reflect the correctly evaluated condition. e.g. “Artifacts Are Saved”, “String Values are Equal”, “Element is Validated”	Starts with: “Used to verify if …”	
Value Providers	Should reflect the value being provided. e.g. “Project Assembly Name”, “Current Time”, “First Related Item”	Starts with: “Retrieves the value of the ….”	
Validation Rules	Should reflect the rule being evaluated. e.g. “Artifacts Are Saved”, “Property Value Required”, “Matches Current Time”	Starts with: “Validates that the …”	
Enum Type Converters	Should reflect the values being returned.	Starts with: “Returns …”	
Data Type Converters	Should reflect the data type being converted.	Starts with: “Converts …”	
UI Type Editors	Should reflect the data type being edited.	Starts with: “Edits …”	

Use the guidelines for naming the automation class.

Custom Command Class	• Describe the action being performed by the command as a whole. e.g. ‘ConfigureUserInformation’. • Complete the name with the suffix ‘Command’. e.g. ‘ConfigureUserInformationCommand’.
Custom Condition	• Describe the condition being evaluated as a whole. e.g. ‘UserIsFemale’. • Complete the name with the suffix ‘Condition’. e.g. ‘UserIsFemaleCondition’.
Custom Value Provider	• Describe the value being provided as a whole. e.g. ‘UserFullName’. • Complete the name with the suffix ‘Provider’. e.g. ‘UserFullNameProvider’
Custom Validation Rule	• Describe the subject being validated by the rule as a whole. e.g. ‘ValidateUserInformation’. • Complete the name with the suffix ‘Rule’. e.g. ‘ValidateUserInformationRule’.
Custom XAML Wizard	• Describe the action being performed by the wizard as a whole, remembering that the element upon which it is going to be configured is usually very specific. e.g. ‘ConfigureUserInformation’. • Complete the name with the suffix ‘Wizard’. e.g. ‘ConfigureUserInformationWizard’. **Note**: Wizards are generally not very reusable, and so are very specific to the specific elements in the pattern model.
Custom XAML Wizard Page	• Describe the sub-set of information being displayed on this page. e.g. ‘ProfileInformation’ • Complete the name with the suffix ‘Page’. e.g. ‘ProfileInformationPage’. **Note**: Wizard pages are generally not very reusable, and so are very specific to the specific elements in the pattern model.
Custom Enum Type Converter	• Describe the enumeration values being returned from the converter. e.g. NumbersBetweenOneandTen • Complete the name with the suffix ‘Converter’. e.g. NumbersBetweenOneandTenConverter
Custom Data Type Converter	• Describe the data type being converted by the converter. e.g. DateTime • Complete the name with the suffix ‘Converter’. e.g. DateTimeConverter
Custom UI Type Editor	• Describe the data type being edited by the editor. e.g. DateTime • Complete the name with the suffix ‘Editor’. e.g. DateTimeEditor

How To: Coding Your Own Automation Classes

Every custom automation class that is generated for you (i.e. Commands, Conditions, Value Providers, Validation Rules, Wizards, Enum Type Converters, etc.) has two very important aspects to it that require careful attention when coding it, so that the custom automation class integrates properly into the authoring experience, and aids in troubleshooting the automation when used in the pattern toolkit.

> **WARNING**: Do not overlook these steps, they are strongly recommended practices for your toolkit to operate effectively.

Descriptive Information

Use the naming guidelines in Naming Custom Automation Classes for configuring the descriptive properties of the automation class, so that it is correctly documented, named, described and categorized.

Tracing

Apply appropriate tracing statements throughout the code. See Applying Tracing to Automation Classes for consistency.

Implementation

See Implementing Your Automation Own Automation Class for details on how to implement the automation in your class.

How To: Applying Tracing to Your Own Automation Classes

Adding tracing to your automation classes, such as Commands, Value Providers, Conditions and Validation Rules, provides both authors and users with diagnostic information and tools for identifying and troubleshooting issues with the automation classes in your toolkit.

When automation fails for unexpected reasons (i.e. bad configuration, missing data, unexpected runtime conditions), errors are automatically reported to users using error message boxes and stack traces are echoed to the trace window for reference. Without additional detailed diagnostic tracing the error mechanism is often the only way to identify issues that went wrong with toolkit automation, but rarely do these errors contain enough information to understand what led to the errors, or what state the toolkit is in. Tracing becomes that necessary powerful diagnostic tool for authors to debug their toolkits.

Trace information appears for both authors and users in the 'Output Window', in the 'Pattern Toolkit Extensions' pane (see Tracing Window). As an author and a user you can see information about how the toolkits work, and a record of all information, warnings and errors there.

By default, the level of trace messages seen in the output window is 'Warning' and above. But if you want to see more diagnostic information, you can configure the level of tracing in the options dialog of Visual Studio. See the options in the Tracing Window for more details.

In order to leverage tracing in your automation classes, follow the steps below.

How to Add Tracing

Add the Tracer variable to your automation class, remembering to replace the type of your class in the method GetSourceFor<T>():

```
		private static readonly ITraceSource tracer = Tracer.GetSourceFor<AutomationClass>();
```

What and When to Trace

Your automation class will have a single actionable method such as: Execute() (for Commands), or Evaluate() (for Conditions and Value Providers) or Validate() for (Validation Rules), these methods should always start with a call to:

```
			this.ValidateObject();
```

This call to ValidatorObject() ensures that all required configurable properties on the automation class are correctly assigned at runtime, and all imports are satisfied. If none are, then this method throws an exception and automatically traces the issues for you. Tracing before this method call is unnecessary, and could potentially lead to inadvertent null exceptions as the parameters of the class, that you may include in the trace statements, may not be valid prior to this call.

Start your method with a call to TraceInformation(), tracing what the automation will do, using data from the most important configured properties on the automation class. This trace entry tells the troubleshooter what is being automated next by what automation class.

Use calls to TraceVerbose() throughout the logic of the code for informing user on decisions being made by the code. These calls are only seen when the level of tracing is turned up very high, and allow a troubleshooter to see the decision paths through the code.

Use calls to TraceWarning() to identify where the logic fails expectedly, but continues. These calls are seen by toolkit users by default, so should only contain data and conditions that indicate a recoverable failure.

Trace the final result of the automation using TraceInformation(). Together with first trace call, this pair of traces provides information about what automation was executed, and whether it succeeded with its results.

Don't use TraceError() for error conditions, instead throw exceptions (i.e. InvalidOperationException) where the logic fails unexpectedly. This will automatically get traced using TraceError() by the runtime framework.

Tracing Statement Recommendations

For consistency, you may want to keep the format of tracing statements consistent based upon the automation type. This aids in the readability of the tracing statements in the trace window.

For example, Value Providers and Conditions both 'Evaluate' values with parameters, Validation Rules 'Validate' rules and Commands 'Execute' actions with parameters. You can reflect that in trace statements consistently.

Here are some recommended trace statement formats that you will see traced from the provided built-in automation.

	Commands	**Value Providers**	**Conditions**	**Validation Rules**
Initial Trace statement	Starts with "Executing …. with parameters …."	Starts with "Evaluating …. with parameters …."	Starts with "Determining …. with parameters …."	Starts with "Validating …. with parameters …."
Result Trace statement	Starts with "Executed …. as …."	Starts with "Evaluated …. as …."	Starts with "Determined …. as …."	Starts with "Validated …. as …."

This is an example of the ExpressionValueProvider automation, with recommended tracing statements (comments and other necessary display attributes removed for this example):

> **Note**: Resource strings would normally be defined in a resource file for localization. They are displayed in the code below for example purposes only.

```
	public class ExpressionValueProvider : Microsoft.VisualStudio.TeamArchitect.PowerTools.Features.ValueProvider
	{
		private static readonly ITraceSource tracer = Tracer.GetSourceFor<ExpressionValueProvider>();

		[Import(AllowDefault = true)]
		public IInstanceBase CurrentElement { get; set; }

		[Required(AllowEmptyStrings = false)]
		public string Expression { get; set; }

		public override object Evaluate()
		{
			this.ValidateObject();

			tracer.TraceInformation(
				“Evaluating expression'{0}' on element '{1}'.”, this.Expression, this.CurrentElement);

			var result = ExpressionEvaluator.Evaluate(this.CurrentElement, this.Expression);

			if (result == null)
			{
				tracer.TraceWarning(
					“Resolved expression'{0}' on element '{1}', to null. The expression maybe invalid or may resolve to the wrong element in the element's ancestry.”, this.Expression, this.CurrentElement);
			}

			tracer.TraceInformation(
				“Evaluated expression'{0}' on element '{1}', as '{2}'.”, this.Expression, this.CurrentElement, result);

			return result;
		}
	}
```

How To: Implementing Your Own Automation Class

Automation classes such as command, conditions, value providers etc. with a few exceptions don't add a great deal of value unless they are configurable by authors and have access to the development environment, tools, and external services, and can interact with the pattern model upon which they are configured by authors. It is the combination of these abilities which makes automation classes very powerful and highly reusable.

Typed vs. Generic Automation

There are two classifications for all automation classes: Typed and Generic.

Typed automation classes are those that are only specific to specific elements in your pattern model, and are only intended to be configured on that element in that pattern model. These automation classes have intrinsic knowledge about the specific type of that element and the rest of the pattern model. These automation classes are targeted at specific problems in this pattern model and are rarely if ever re-used. Because of their intrinsic knowledge of the pattern model, they rarely require additional configuration from the author, the context for their operation is the state of the pattern element (and model) they are configured on.

Generic automation classes are those that are explicitly designed to be re-used across different pattern elements and in different pattern models. Because of their generic nature, and the fact they can make no assumptions about the pattern model upon which they are configured, these automation classes will almost always require additional configuration from the pattern author to define for them all the context they need which to operate with.

All of the Provided Automation Types classes are examples of generic automation.

Adding Configurable Properties

You typically add custom properties to your automation class to make your class configurable by a toolkit author when your automation class is selected and configured for use on an element in the pattern model. This is most common when designing generic re-usable automation classes. These properties provide the configurable context that authors can control.

These properties gather values at design-time from authors, which are then persisted in the pattern model, and are used to change the behavior of the automation class when executed at runtime by a user.

These properties can be configured with either static values (typed or selected by the pattern author), or configured with Value Providers which will fetch the property values dynamically.

See Adding Properties to Automation Classes for more details on how to create and declare properties.

Accessing Useful Services

You typically import services for use in your automation class through MEF exported services provided by other extensions in Visual Studio, so that you can access other services and perform rich automation with the provided services in the development environment.

Examples of commonly imported services include:

- ISolution – for access to the project system (i.e. solution, projects and items in Solution Explorer)
- IProductElement – for access to elements in your pattern model.
- IFeatureManager – for access to guidance workflows in the solution.
- IFxrUriReferenceService – for resolving URI's that are passed around the model as string values.

See Importing Services into Automation Classes for more details on how to declare imports to access other services via MEF.

Accessing the Current Element

Many automation classes require access to the current element of the pattern model upon which the automation is configured. From there you can navigate the pattern model to the current elements: parent, ancestors, siblings and descendant elements.

In order to get a reference to the current element in your automation class you must import the element through MEF, as you do for any service (described above).

Generic automation classes will always import the current element as an IProductElement instance. Which gives the class access to the pattern model in the generic scheme.

Typed automation classes will import the current element as its typed interface name. i.e. IElementName. Which gives the class access to the pattern model with correctly typed properties and methods.

See Importing the Current Element into Automation Classes for more details on how to declare imports to access the current element via MEF.

Navigating a Pattern Model

Once you have a reference to the current element, you can navigate the pattern model and manipulate it programmatically. You can change values of properties, and add and remove instances of elements anywhere in the model.

Navigation in a pattern model can be done in one of two ways depending on what the target of your automation class is designed for.

Generic Hierarchies

For automation classes intended to be reused across multiple pattern toolkits, then you will have limited knowledge about the actual pattern models you are navigating except that they are composed of elements, collection and variable properties in some arbitrary arrangement, and that there is a single root node called the pattern.

These elements are not typed to any specific toolkit, and elements are related in collections indexed by ordinal. However, you can navigate up the hierarchy predictably using the 'Parent' property, and navigate down the hierarchy predictably using collections again by ordinal. There are a set of standard interfaces to enable this navigation that treats the pattern model as a 'Generic' or 'Untyped' hierarchy of generic things. There are a number of different interfaces that describe elements which have child elements, and elements which have properties, and interfaces that distinguish between collections and elements and the root pattern element.

See Navigating Pattern Models Programmatically (Generically) for more details on how navigate your way around a pattern model using the generic interfaces for reusable automation classes.

Typed Hierarchies

For automation classes that are optimized for specific toolkits, and will not be reused across other toolkits, you can get more compiler type-checked and granular information about the hierarchy you are navigating using 'Typed' interfaces. These 'Typed' interfaces are auto-generated for every toolkit and construct an object model for that specific toolkit. These strongly typed interfaces reflect the configured structure and cardinalities of relationships in the pattern model.

See Navigating Pattern Models Programmatically (Typed) for more details on how navigate your way around a pattern model using the typed interfaces for optimized automation classes.

Moving Between Generic and Typed Hierarchies

It is possible and even desirable, to move between the generic and typed schemes, especially when calling some of the automation framework methods and helper functions, which naturally all depend on the generic interfaces.

> **Note**: You cannot simply cast instances of automation classes to move between these different interface schemes.

See Moving Between Generic and Typed Pattern Model Schemes for more details on the syntax for doing this.

Common Automation Actions

There are a number of common things many toolkit builders need to do to manage solution items that are referenced by their pattern models.

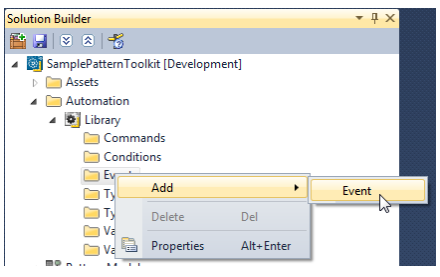
See the Common Automation Tasks topic which has references to libraries and classes that can help you custom code your automation more efficiently

How To: Creating Your Own Event

You can create your own custom automation events, by adding a new 'Event' to your pattern toolkit project, and implementing the interfaces defined within.

Create the Event

In your pattern toolkit project, in 'Solution Builder', add a new 'Event'



Implement the Event

Follow guidelines in the code provided.

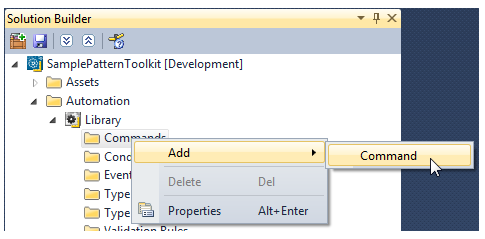
> **Recommend**: See the Naming Custom Automation Classes topic for guidelines on naming and preparing your class for integration into a toolkit.

How To: Creating Your Own Commands

You can create your own custom automation commands, by adding a new 'Command' to your pattern toolkit project, and implementing the class defined within.

Create the Command

In your pattern toolkit project, in 'Solution Builder', add a new 'Command'



Implement the Command

Follow guidelines in the code provided.

> **Recommend**: See the Naming Custom Automation Classes topic for guidelines on naming and preparing your class for integration into a toolkit.
> **Recommend**: See Implementing Your Automation Own Automation Class topic for adding properties, importing services and navigating the pattern model.

Implement the Settings Validator

You can provide design-time validation for the properties defined by your custom command, so that toolkit authors using the command are enforced to provide correct configuration for it on their pattern model.

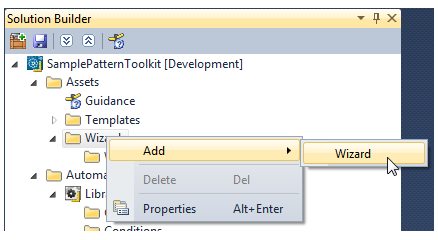
> **Note**: A default implementation of a command validator is provided with the command as a nested file.

How To: Creating Your Own Wizards

You can create your own custom wizards, by adding a new 'Wizard' to your pattern toolkit project, and configuring the XAML within.

Create the Wizard

In your pattern toolkit project, in 'Solution Builder', add a new 'Wizard'



Implement the Wizard

Follow guidelines in the code provided.

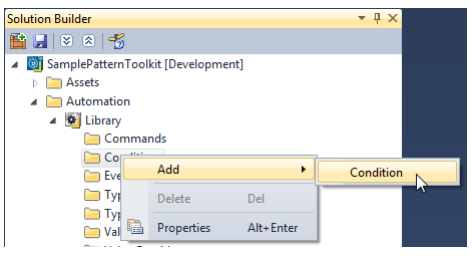
> **Recommend**: See Creating a Wizard for more details on how to implement the wizard.
> **Recommend**: See Implementing Your Automation Own Automation Class topic for adding properties, importing services and navigating the pattern model.

How To: Creating Your Own Conditions

You can create your own custom automation conditions, by adding a new 'Condition' to your pattern toolkit project, and implementing the class defined within.

Create the Condition

In your pattern toolkit project, in 'Solution Builder', add a new 'Condition'



Implement the Condition

Follow guidelines in the code provided.

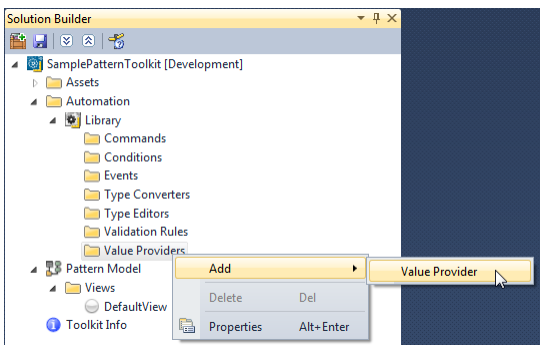
> **Recommend**: See the Naming Custom Automation Classes topic for guidelines on naming and preparing your class for integration into a toolkit.
> **Recommend**: See Implementing Your Automation Own Automation Class topic for adding properties, importing services and navigating the pattern model.

How To: Creating Your Own Value Providers

You can create your own custom automation value providers, by adding a new 'Value Provider' to your pattern toolkit project, and implementing the class defined within.

Create the Value Provider

In your pattern toolkit project, in 'Solution Builder', add a new 'Value Provider'



Implement the Value Provider

Follow guidelines in the code provided.

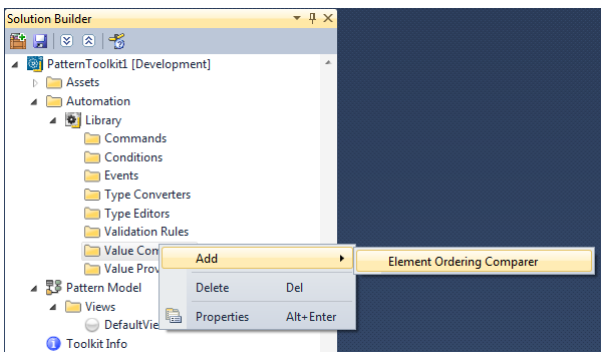
> **Recommend**: See the Naming Custom Automation Classes topic for guidelines on naming and preparing your class for integration into a toolkit.
> **Recommend**: See Implementing Your Automation Own Automation Class topic for adding properties, importing services and navigating the pattern model.

How To: Create You Own Ordering Comparer

You can create your own custom ordering value comparer, by adding a new 'Element Ordering Comparer' to your pattern toolkit project, and implementing the class defined within.

Create the Ordering Comparer

In your pattern toolkit project, in 'Solution Builder', add a new 'Element Ordering Comparer'



Implement the Comparer

Follow guidelines in the code provided.

> **Recommend**: See the Naming Custom Automation Classes topic for guidelines on naming and preparing your class for integration into a toolkit.
> **Recommend**: See Implementing Your Automation Own Automation Class topic for adding properties, importing services and navigating the pattern model.

How To: Creating Your Own Validation Rules

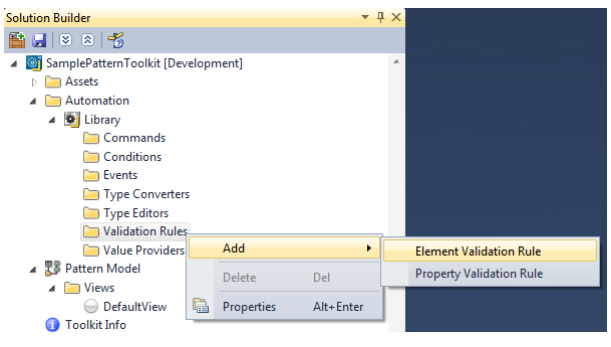
You can create your own custom validation rules, by adding a new 'Element Validation Rule' or 'Property Validation Rule' to your pattern toolkit project, and implementing the class defined within.

An 'Element Validation Rule' is intended for validating elements, and their attributes (i.e. configuration, relationships, cardinality, etc.)

A 'Property Validation Rule' is intended for validating individual variable properties of an element in the pattern model (i.e. input validation, bounds checking etc.).

Create the Validation Rule

In your pattern toolkit project, in 'Solution Builder', add a new 'Element Validation Rule' or a new 'Property Validation Rule'



Implement the Validation Rule

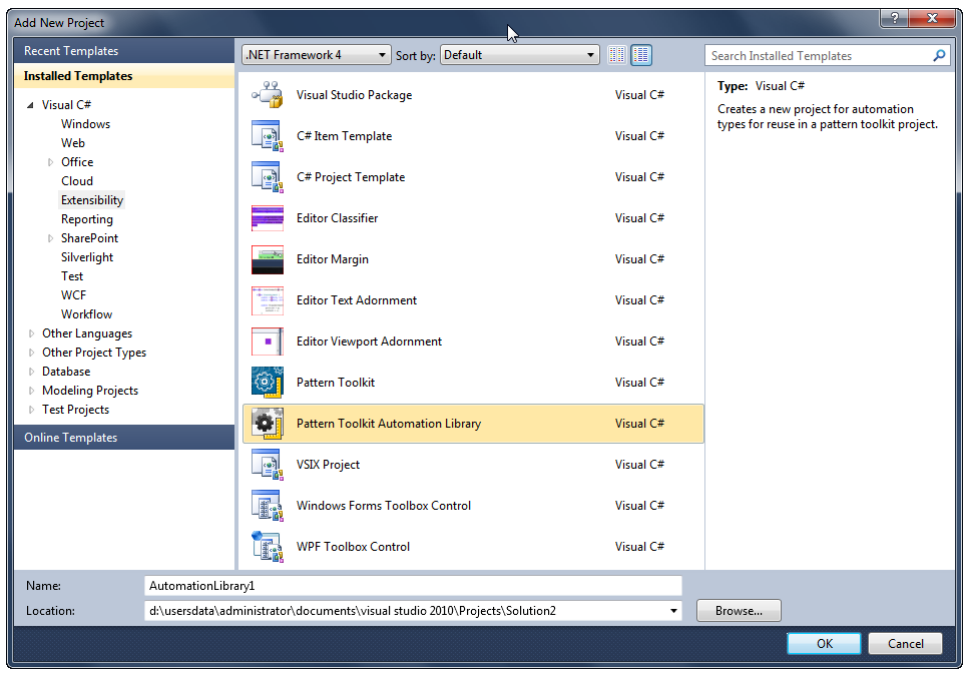
Follow guidelines in the code provided.

> **Recommend**: See the Naming Custom Automation Classes topic for guidelines on naming and preparing your class for integration into a toolkit.
> **Recommend**: See Implementing Your Automation Own Automation Class topic for adding properties, importing services and navigating the pattern model.

How To: Add Additional Automation Libraries

By default, each pattern toolkit project comes with its own automation library project that contains all its automation classes. However, you may decide to factor out automation into additional reusable automation libraries to be shared with one or more other toolkit projects. This is a great strategy for increasing reuse of commonly used automation.

You can do this by simply adding a new 'Automation Library' project to your solution, and configure it to be used by each toolkit.



Add the Automation Library Project

In the toolkit solution, add a new 'Pattern Toolkit Automation Library' project.

> **Note**: A 'Pattern Toolkit Automation Library' is nothing more than a standard C# class library project with a specific structure and naming.

Configure Toolkit Projects

From each toolkit project that wants use automation in this new library project, do the following:

1. Add a 'Project Reference' from the toolkit project to the new library project.
2. Edit the properties of the project reference, and set the value of the 'Output Groups Included in VSIX' property to 'BuiltProjectOutputGroup'
3. In the source.extension.tt file, add a <MefComponent> element that references the library project in the solution.

   i.e. <MefComponent>|NewAutomationLibrary1|</MefComponent>

4. Rebuild the solution

You should now see all automation classes in each shared automation library project appear in the lists of automation (i.e. Commands, Conditions, Value Providers etc.) in the pattern models of each toolkit project.

How To: Common Automation Scenarios

Use these examples as starting points for determining which of the Provided Automation Types to configure to automate your pattern toolkit.

Project or Item Templates

> **Note**: These scenarios require that you have VS Template Assets ready to use. See Unfolding VS Templates for more details on how to create them.

If you want to unfold a VS Project Template when the pattern is first instantiated, AND you want that template to appear in the Add New Project dialog of Visual Studio

- On the Pattern Element
  - Add a 'Template Launch Point'
    - Set the 'Template' property to your project template file (*.vstemplate)

If you want to unfold a multiple VS Project Templates when the pattern is first instantiated, AND you want one of those templates to appear in the Add New Project dialog of Visual Studio.

- On the Pattern Element:
  - Add a 'Command', with the 'UnfoldVsTemplateCommand' command type.
    - Set the 'Template' property to one of your secondary project template files (*.vstemplate)
  - Add a 'Template Launch Point'
    - Set the 'Template' property to your primary project template file (*.vstemplate)
    - Set the 'Command' property to the previous command.

> **Note**: If you have more than two project templates, use the 'Aggregator Command' to execute additional Commands with the 'UnfoldVsTemplate' command type.

Code Generation

> **Note**: These scenarios require that you have Text Template Assets ready to use. See Generating Code for more details on how to create them.

If you want to generate a code file from element in your pattern whenever the solution is built.

- From the Element:
  - Add a Command, with the 'GenerateCodeCommand' command type
    - Set the 'Template' property to your text template file (*.tt or *.t4)
  - Add an 'Event Launch Point'
    - Set the 'Event Type' property to the 'OnBuildStarted' event of Visual Studio.

Other Scenarios

For scenarios that are omitted here (this list is always being updated with the common scenarios), there may still be a built-in automation class that can address your scenario. See the Creating Your Own Custom Automation Classes topic to determine whether there already exists automation for your scenario, and if not, how to create your own custom automation to address that scenario.

Building

These topics help in understanding the building of toolkit projects.

Understanding: Building a Toolkit Project

Building a Pattern Toolkit project is the same process as building any project in Visual Studio. Hit F6 to build the solution.

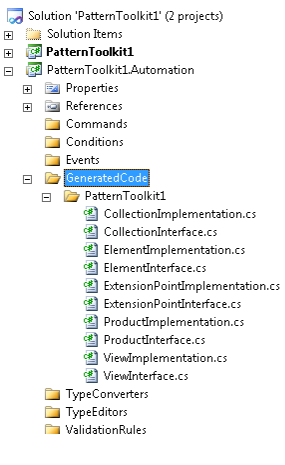
However, when a toolkit project is built, a number of additional activities occur that ensure that the toolkit project is correct formed before compiling, deployment and testing it.

Pattern Model Validation

When a toolkit project is built, the pattern model is automatically validated to ensure completeness and correctness. Any errors found in in the pattern model prevent additional generative steps occurring, and instruct the author to fix those first.

Typed Interface Generation

When a toolkit project is built, a set of strongly typed C# interfaces and classes are re-generated into the 'Automation' project of the toolkit automatically. These interfaces (and their implementations) allow your automation classes to have access to the pattern model in a strongly typed manner.



> **Note**: When adding or modifying new views, elements and properties in the pattern model, the project needs to be rebuilt in order for those modified elements and properties to be reflected in the typed interface code.

VISIX Manifest Generation

When a toolkit project is built, the VSIX manifest file is re-generated into the 'toolkit' project of the toolkit automatically. This manifest defines the packaging of the toolkit, and includes information defined in the 'Toolkit Info' element in Solution Builder.

> **Note**: When modifying properties of the 'Toolkit Info', the project needs to rebuilt in order for this information to be reflected in the deployed pattern toolkit VSIX.

Generated Code Attributes

All generated code in a toolkit project is generated with attributes that designate it as 'generated by a tool' for analysis tools (i.e. Fx Cop, Style Cop etc.) to distinguish it from manually created code.

Specifically, all code files are prefixed with a standard code generation header that can contains special markers in the comments to support code analysis tools (e.g. Style Cop). All classes and interfaces are also marked with the GeneratedCodeAttribute to support compiled assembly analysis tools (e.g. Fx Cop).

Furthermore, all classes can also be marked with the ExcludeFromCodeCoverageAttribute to permit testing tools to exclude the generated code from the calculation of their code coverage metrics.

Experimental Instance Preparation

When a toolkit project is built it is automatically installed (and enabled) into the Experimental Instance of Visual Studio so that you can test and debug it. There is an option that controls this behavior, and it can be found on the property pages of the toolkit project, under the 'VSIX' tab.

In addition to installing the extension, the project will also copy (and enable) any missing dependent VSIX dependencies that the toolkit needs to work, in the Experiment Instance of Visual Studio. Copying these vital dependencies can be controlled by adding an MSBUILD property to the toolkit project file.

<CopyVsixDebuggingDependencies>true</CopyVsixDebuggingDependencies>. By default this property is set to true.

See the next topic on Building a Toolkit Project topic for how to control automatic code generation, validation and generated code.

How To: Building a Toolkit Project

See Understanding Building a Toolkit Project for details on what activities automatically occur when building your toolkit project.

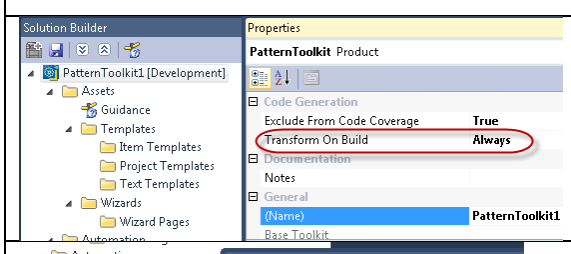
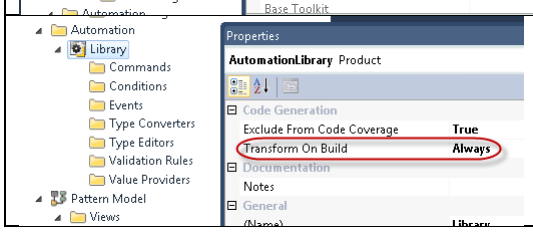
This topic guides you through controlling those activities, and how to enact them manually when the automatic processes are supressed.

Controlling Code Generation

The generation of all code is performed automatically when the project is built to aid in rapid development of the toolkit, especially in the earlier stages of development. This ensures that changes in the pattern model are always synchronized in the generated code. However, as a pattern matures and the pattern model stabilizes, the generation of artifacts to stay synchronized with the model may hinder or slow down rapid build cycles, such as when you want to create, build and test automation classes.

Suppressing Automatic Code Generation

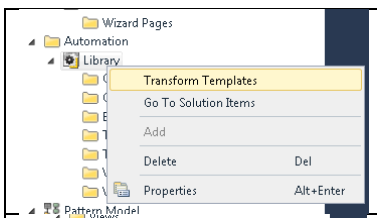
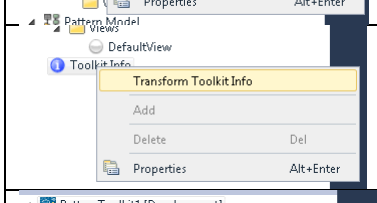
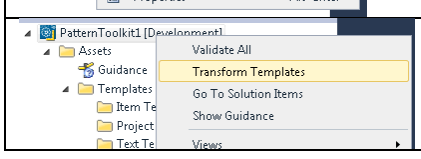
For reasons like this, you can control whether the code generation automatically occurs on build with the 'Transform On Build' property. This property appears on both the 'Toolkit' element and the 'Library' element in Solution Builder. Both have the same meaning, but control different artifacts generated into the projects of the toolkit.

	Transform On Build	Description
	Always	When the solution is built, toolkit information and the vsix manifest file are re-generated.
	Never	The re-generation of the toolkit info and vsix manifest are skipped.
	Always	When the solution is built, the typed interface code is re-generated.
	Never	The re-generation of the typed interface code is skipped.

Manual Code Generation

If you suppress automatic code generation, as described above, and you make certain modifications to the pattern model or any modifications to the 'Toolkit Info', you will need to manually re-synchronize generated artifacts in the projects of the toolkit, by doing the following:

In Solution Builder (in this order),

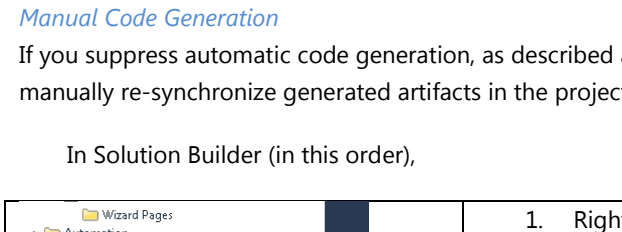
	1. Right-click on the 'Library' element, and select the 'Transform Templates' menu.
	2. Right-click on the 'Toolkit Info' element, and select the 'Transform Toolkit Info' menu.
	3. Right-click on the 'Toolkit' element, and select the 'Transform Templates' menu.

Excluding Generated Code from Code Coverage

All generated code in a toolkit project will be included as part of any calculation of code coverage for tests.

This may not be desirable because this code cannot be controlled by a toolkit author, and may not warrant the development of tests that cover it.

For reasons like these, you can control whether the code is excluded from the code coverage calculation with the 'Exclude From Code Coverage' property. This property appears on both the 'Toolkit' element and the 'Library' element in Solution Builder. Both have the same meaning, but control the attribute on different artifacts generated into the projects of the toolkit.

	Exclude From Code Coverage	Description
	True	Generated code will be, marked with attributes that will exclude it from code coverage calculations.
	False	Generated code is not marked with attributes that exclude it from code coverage calculations.

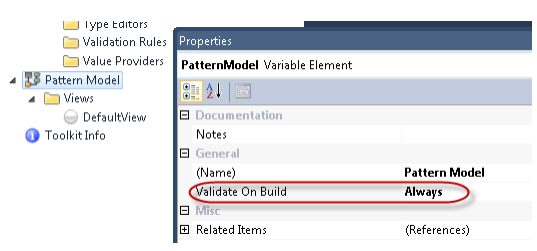
Note: By default, this property is set to 'true', to exclude all code from code coverage calculations.

Controlling Validation

Validation of the pattern model is performed automatically when the project is built to aid in rapid development of the toolkit, especially in the earlier stages of development. This ensures that changes in the pattern model are always verified. However, as a pattern matures and the pattern model stabilizes, the verification of the model may hinder or slow down rapid build cycles, such as when you want to create, build and test automation classes.

Suppressing Automatic Validation

For reasons like this, you can control whether validation automatically occurs on build with the 'Validate On Build' property. This property appears on the 'Pattern Model' element in Solution Builder.

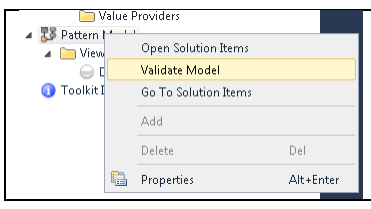
	Validate On Build	Description
	Always	When the solution is built, the pattern model is validated.
	Never	The validation of the pattern model is skipped.

Note: Validation of the pattern model is always performed, regardless of the above property value, whenever you open the pattern model designer, make a change and try to save it.

Manual Validation

If you suppress automatic validation (on build), as described above, and you want to validate it manually.

In Solution Builder (in this order),

	1. Right-click on the 'Pattern Model' element, and select the 'Validate Model' menu.

How To: Building a Toolkit in Team Build

Configuring a Build Server

> **Note**: This process applies to Team Foundation Build 2010.

Pattern Toolkit projects are simply ordinary C# projects with some special MS Build targets and dependencies on assemblies installed by NuPattern. These special build targets and dependencies will need installing on the team build server by installing the 'NuPattern Toolkit Builder' on the build server, (as the build server account).

You can do this with this procedure (assuming TFSBUILDS represents the account that executes build on the build server):

1. Login to the build server as a local administrator.
2. Copy the '**NuPatternToolkitBuilder.vsix'** file to the local disk.
3. Open the Visual Studio Command prompt:
    a. **runas /user:domain\TFSBUILDS "vsixinstaller C:\NuPatternToolkitBuilder.vsix"**
    b. Enter the password for the build account
4. Delete directory: **C:\Users\TFSBUILDS\AppData\Local\Microsoft\VisualStudio\10.0Exp**
5. In the Visual Studio Command Prompt:
    a. **runas /user:domain\TFSBUILDS "cmd.exe"**
    b. Enter the password for the build account
6. In the Visual Studio Command Prompt:
    a. **"C:\Program Files (x86)\Microsoft Visual Studio 201X SDK\VisualStudioIntegration\Tools\Bin\CreateExpInstance.exe" /Reset /VSInstance=10.0 /RootSuffix=Exp && PAUSE**
7. In the Visual Studio Command Prompt:
    a. **runas /user:domain\TFSBUILDS "C:\Program Files (x86)\Microsoft Visual Studio 10.0\Common7\IDE\devenv.exe /rootSuffix Exp"**
    b. Select: "Visual Studio General Settings"
    c. Open Extension Manager, and ensure the following extensions are enabled.
        - NuPattern Toolkit Manager
    d. Close Visual Studio

Building Pattern Toolkit Projects

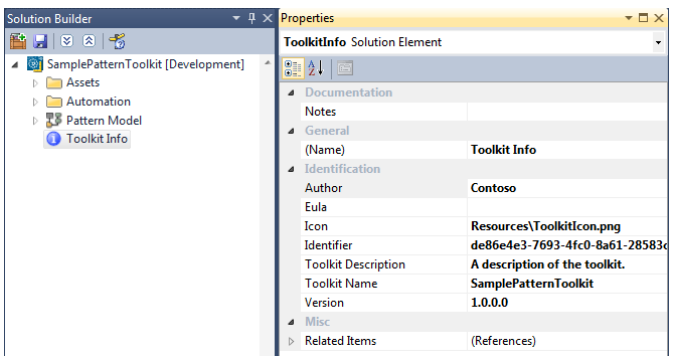
By default, solutions containing pattern toolkit projects should build just fine in Team Build with no changes to them.

However, if you are nesting toolkits for single VSIX deployment, then it is necessary to modify the value of the <CopyVSIXtoFolder> element to include the Team Build workspace of the build definition. See Packaging Multiple Toolkits Together for more details.

Deployment & Releasing

Understanding: What is Toolkit Information?

Toolkit information is used to build the toolkit package that installs and registers a pattern together with its assets, automation and guidance.



The toolkit packaging technology used is VSIX technology, and this technology requires detailed information about the package that can be read by tools that install these kinds of packages into Visual Studio. One such tool is the '[Extension Manager]' in Visual Studio, which reads the toolkit information form the package and presents it to viewers.

This topic requires more information.

How To: Packaging Assemblies for Deployment

This topic requires more information.

Key points: Don't package non-redistributable assemblies in your VSIXes. CopyLocal = false for assembly references, etc.

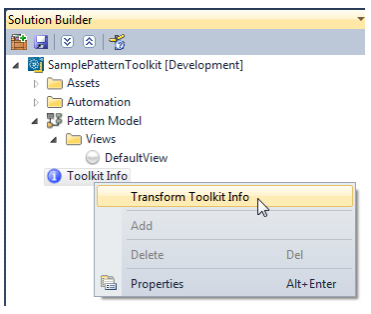
How To: Updating Toolkit Information

When the toolkit information for a pattern toolkit changes in the 'Solution Builder' window, this information needs to find its way into the source.extension.vsixmanifest file to correctly build the VSIX package for the pattern toolkit.

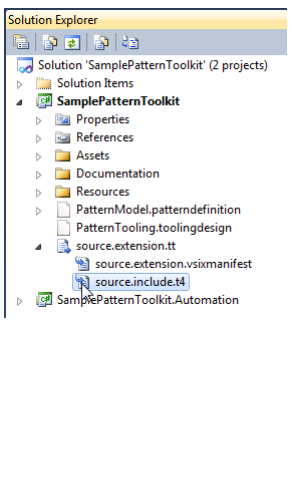
By default, this information is automatically propagated when you build the toolkit project. However there may be times when you want to manually enact the process.

Update Information Manually

On the 'Toolkit Info' element in 'Solution Builder' for the pattern toolkit, right-click and select the 'Transform Toolkit Info' menu.



This menu transforms the information in the 'Toolkit Info' element into a temporary file in the toolkit project (source.include.t4) that will be transformed into the source.extension.vsixmanifest file on the next build of the toolkit.



How To: Versioning a Toolkit

This topic requires more information.

How To: Installing a Toolkit for Customization

> **Note**: We refer to any pattern toolkit being customized as the 'base' toolkit.

Installing a pattern toolkit ready for customization is no different than installing a pattern toolkit for use. Basically, all you need to do is run the *.vsix file, and it installs into Visual Studio.

The key concept to be aware of is that unlike developing a pattern toolkit where you run and test the toolkit in the Experimental Instance of Visual Studio, when you develop a customized toolkit you have to have the 'base' toolkit installed in the main instance of Visual Studio. And the customized toolkit is run and tested in the Experimental Instance again.

> **Note**: You can confirm that any pattern toolkit is installed by finding it in the 'Extension Manager' of the instance of Visual Studio you are running.
>
> **Tip**: If you want to customize a pattern toolkit that you are currently developing, install the toolkit into Visual Studio by running the *.vsix file generated by your toolkit project. You can find this *.vsix file in the output directory (i.e. bin\Debug') of the toolkit project.
>
> **Warning**: It is not recommended to develop the base toolkit and customized toolkits in parallel on the same development machine. Changes in the base toolkit must be continuous propagated from Experimental Instance to the normal instance of Visual Studio before the customized toolkit will benefit from them. This process gets too confusing too quickly and leads to complicated debugging issues.

How To: Creating an MSI for Toolkit Deployment

Technically speaking you need only deploy a pattern toolkit in an MSI if your toolkit requires resources that cannot be deployed directly with the VSIX packaging technology. For example, you want to install registry settings in windows, or you want to deploy ancillary files to the hard disk. IN some cases, MSI is required by organizational policy for re-distribution of tools.

MSI packages can be created, and VSIXes packaged in them, with MSI development tools such as WIX, or other 3rd party products.

VSIXes support being packaged in MSI, and have special properties in the source.extension.vsixmanifest file to indicate to 'Extension Manager' that the VSIX has been deployed by MSI to prevent Visual Studio users from uninstalling the VSIX from Extension Manager manually.

See Visual Studio Extension Deployment for more details on this process.

How To: Packaging Multiple Toolkits Together

It is possible to package one or more (child) toolkit VSIXes inside a single (master) toolkit VSIX, to aid in deployment. You may want to do this when you only want to deploy a single VSIX file containing a number of related toolkits. With this technique a user installs only one (master) VSIX, and that master VSIX installs all the other (child) packaged VSIXes automatically.

> **Note**: Even though there is only one master VSIX file to deploy and install, the master VSIX will install distinct child VSIX entries into Visual Studio, and both the master and all child VSIXes still appear individually in the Extension Manager. This means potentially that a user can remove one or more of the child toolkits manually.

Modify the Child Toolkit Projects

1. In each child toolkit, *.csproj project file, remove the (hidden) file link:

```
    <Content Include="$(LocalAppData)\Microsoft\VisualStudio\1X.0\Extensions\NuPattern\NuPattern Toolkit Builder\X.X.X.X\NuPatternToolkitManager.
vsix">
      <Visible>false</Visible>
      <Link>NuPatternToolkitManager.vsix</Link>
      <FixedLink>
      </FixedLink>
      <IncludeInVSIX>true</IncludeInVSIX>
    </Content>
```

2. In each child toolkit, source.extension.tt file, remove the <reference> to the NuPatternToolkitManager VSIX:

```
       <Reference Id="c869918e-f94e-4e7a-ab25-b076ff4e751b" MinVersion="1.X.X.X">
             <Name>NuPattern Toolkit Manager</Name>
             <VsixPath>NuPatternToolkitManager.vsix</VsixPath>
       </Reference>
```

Modify the Master Toolkit Project

1. In the master toolkit, source.extension.tt file, add a new <reference> to child toolkit VSIX:

```
       <Reference Id="TOOLKITIDENTIFIEROFCHILDVSIX" MinVersion="1.X.X.X">
             <Name>TOOLKITNAMEOFCHILDVSIX</Name>
             <VsixPath>ASSEMBLYNAMEOFCHILDTOOLKITPROJECT.vsix</VsixPath>
       </Reference>
```

> **Action:** You must update the highlighted names above, with values from the source.extension.vsixmanifest file found in the child Toolkit project.

2. In master toolkit, *.csproj project file, add a file link:

```
    <Content Include="..\Binaries\ASSEMBLYNAMEOFCHILDTOOLKITPROJECT.vsix">
      <Link>ASSEMBLYNAMEOFCHILDTOOLKITPROJECT.vsix</Link>
      <IncludeInVSIX>true</IncludeInVSIX>
    </Content>
```

> **Action:** You must update the highlighted paths and names.

3. Assuming the Parent and Child Toolkits exist in the same solution, either: change the build order of the solution to ensure the child toolkit is built before the master toolkit, or add an Assembly Reference to the child toolkit, from the master toolkit project and ensure CopyLocal = false on the assembly reference.

Modify the Child Toolkit Projects for Team Build

In each child toolkit *.csproj project file, uncomment and update the value of the <CopyVSIXtoFolder> elements:

```
  <!-- Copy VSIXES to Binaries directory -->
  <PropertyGroup>
    <CopyVSIXtoFolder Condition=" '$(TeamBuildConstants)' == '' ">$(SolutionDir)Binaries</CopyVSIXtoFolder>
    <CopyVSIXtoFolder Condition=" '$(TeamBuildConstants)' != '' ">$(SolutionRoot)\TeamProjectRoot\Projects\BranchPath\Binaries</CopyVSIXtoFolder>
  </PropertyGroup>
```

> **Action:** You must update the highlighted paths to point to the local folder of the 'Workspace' mapping from the build definition.

Customization

Tailoring, customizing and re-deploying existing Pattern Toolkits.

What is a Customized Toolkit?

> **Recommend:** Refer to What is Customization for details on what a customized toolkit is, and what is and isn't customizable.

See Creating a Customized Toolkit for details on how to create a customized pattern toolkit.

> **Definition**: The toolkit that is being customized will become the 'base' toolkit to the 'customized' toolkit. As customized toolkits can be further customized, the 'base' term is relative to the toolkit as whole families of toolkits can emerge, each one successively customizing the next.
> **Definition**: A pattern toolkit 'author' is the role of the person creating a toolkit. A pattern toolkit 'tailor' is the role of the person customizing an existing toolkit.

How To: Creating a Customized Toolkit

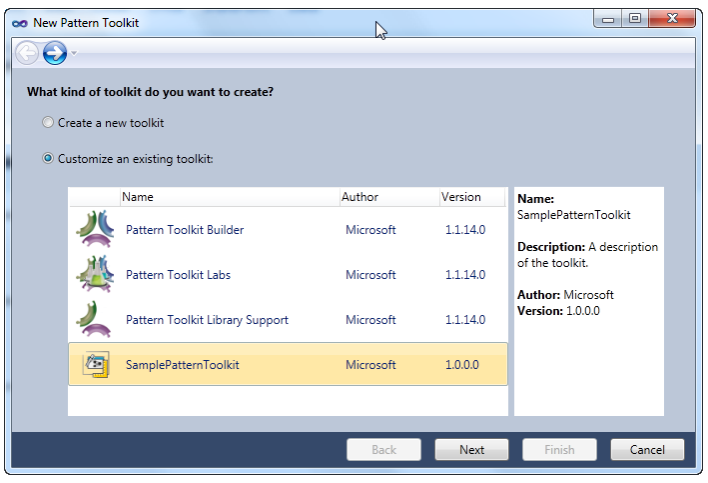
> **Note**: To customize a pattern toolkit, you must have the toolkit installed into Visual Studio first.
> See Installing a Toolkit for Customization for details on this process.

Create the Project

To create a customized pattern toolkit, create a new pattern toolkit project from either the Add/New Project dialog, or from 'Solution Builder' window.

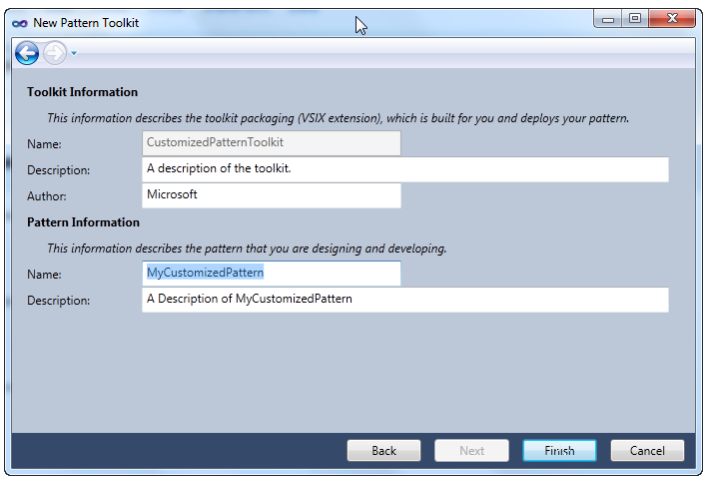
In the first page of the 'New Pattern Toolkit' wizard, select 'Customize an existing toolkit' option.

From the list of installed toolkits, select the toolkit you wish to customize, and click Next.



In the next page, type a name and description of the new tailored toolkit.

You can also modify the name and description of the tailored pattern, and click Finish.



A new toolkit project is created, and you can now proceed to customize the toolkit to create your own tailored version of the toolkit.

Customize the Toolkit

See the following topics:

- Customizing the Pattern Model
- Customizing Assets
- Customizing Guidance
- Customizing Automation
- Customizing Toolkit Info

Modeling

How To: Customizing the Pattern Model

> **Recommend:** Refer to What is Customization for details on what is and isn't customizable in a pattern model.

In general, most aspects, except the original structural integrity of a pattern model, in a pattern toolkit is customizable, but can be further constrained by the base toolkit authors. You are free to change and add a different appearance and new behavior to elements and automation.

This topic requires more information.

Assets

How To: Customizing the Assets

In general, all assets in a pattern toolkit are customizable, since the assets themselves are just associated to elements in the pattern model. You are free to re-associate elements to your own version of the assets or different assets.

Guidance is packaged in the base pattern toolkit in document form (typically *.mht files). New guidance workflows can be created using the same documents or customized version of them.

> **Note**: Currently, pattern toolkits do not support any automation for customizing assets provided from base toolkits directly. The pattern toolkit tailor needs to extract the guidance assets manually, and reassemble customized versions manually.

This topic requires more information.

Guidance

How To: Customizing the Guidance

In general, all guidance used by a pattern toolkit is customizable, since the guidance is itself just associated to elements in the pattern model. You are free to re-associate elements to your own version of the guidance or different guidance.

Guidance is packaged in the base pattern toolkit in document form (typically *.mht files). New guidance workflows can be created using the same documents or customized version of them.

> **Note**: Currently, pattern toolkits do not support any automation for customizing guidance provided from base toolkits directly. The pattern toolkit tailor needs to extract the guidance assets manually, and reassemble customized versions manually.

This topic requires more information.

Automation

How To: Customizing the Automation

In general, all automation used by a pattern toolkit is customizable, since the automation is itself just associated to elements in the pattern model. You are free to re-associate elements to your own version of the automation or different automation, or reconfigure the automation with different parameters if the automation lasses support them.

This topic requires more information.

Deployment

How To: Customizing the Toolkit Info

In general, all toolkit information about a pattern toolkit is customizable. This is in fact this step is mandatory, so that a new toolkit identity is created for the customized version.

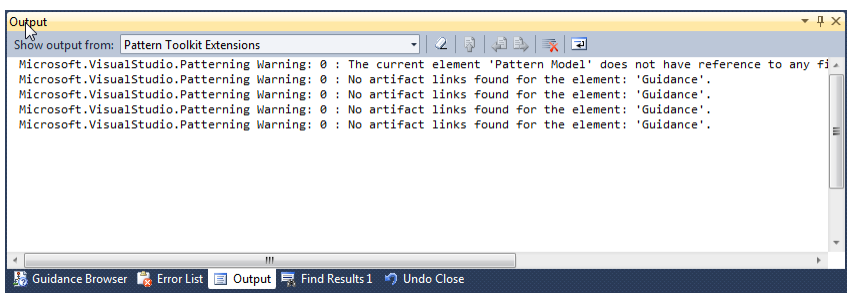
This topic requires more information.

Reference

Troubleshooting Toolkits

If errors or inconsistencies are experienced with building or using pattern toolkits, you can find out more information about what went wrong by looking at the diagnostic trace information displayed by the 'Output Window' of Visual Studio, in the 'NuPattern Toolkit Extensions' pane.

See the Tracing Window for more details.



This kind of detailed information is useful for identifying the causes and the data leading up to the issue.

Diagnosing Issues

By default only 'Warning' and 'Error' diagnostic level traces are captured in the trace window, but you can increase/decrease the fidelity of the information by changing the tracing levels up to 'Information' or 'Verbose', or down to 'Warnings' or 'Errors'.

You change the level of diagnostic information displayed by changing the trace level in the 'NuPattern Toolkit Extensions' Options.

> **Note**: Changing the tracing level only affects new traces, so if troubleshooting a specific issue, you will need to clear the trace window and reproduce the issue again to see the new level of trace messages.

Reporting Issues

Issues with toolkits that cannot be worked around will need to be reported back to the authors of the toolkits being used.

> **Note**: The provided feedback mechanism is specific to each toolkit.
> In general, you should refer to any troubleshooting type sections of guidance provided by the specific toolkit for details about how to contact toolkit owners and report issues.

When reporting any problem try to give as much detail on what problem you are seeing and what set of actions led up to the problem. Including detailed trace logs, that is, with the trace level or at least 'Information' is most desirable.

> **WARNING**: Detailed trace logs, depending on the toolkit, may contain sensitive information. Please ensure this information is sanitized before copying or sending.

Screen shots also help a great deal, short lists of the actions you took to reproduce the problem and more detailed information gathered from trace logs also helps to identify the root cause of the issues.

Resetting the Environment

If you are building a pattern toolkit, and are using the Experimental Instance of Visual Studio to test your toolkit, you may also try resetting the Experimental Instance it to clear out any remnant or duplicate data or settings which could be causing issues.

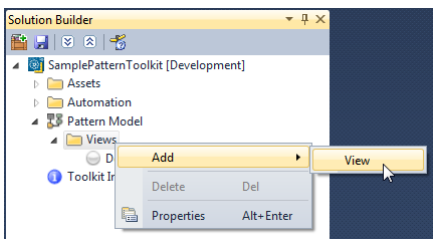
Authoring

Designing and creating patterns, Pattern Toolkits, assets, automation.

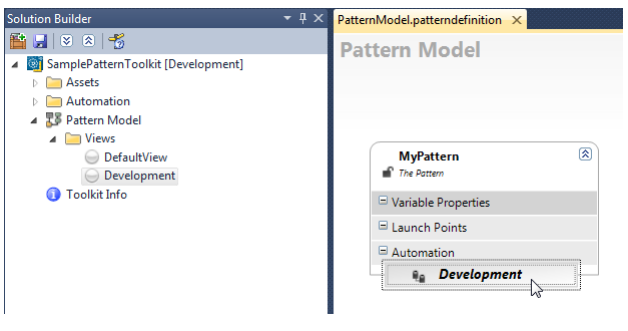
Modeling

Adding a View

To add a 'View' to a pattern model, in 'Solution Builder' right-click on the 'Views' folder of the 'Pattern Model' element an select 'Add View'

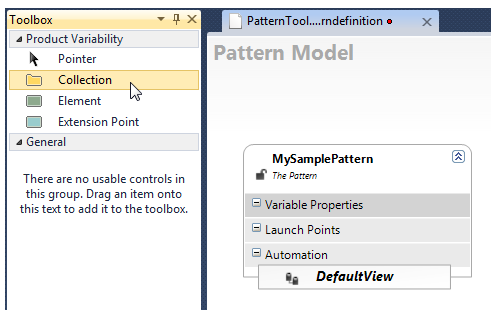


Name the view and it will be created, and double-click on the new view to open it in the pattern model.



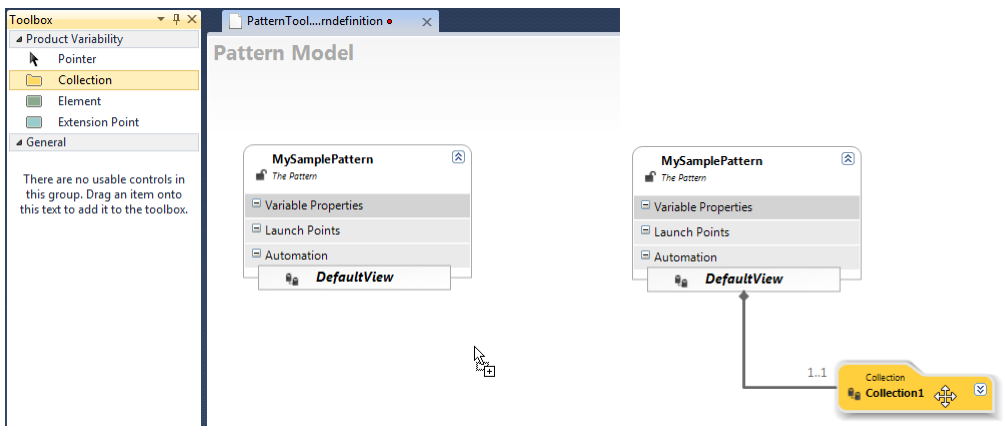
Adding a Collection or an Element

To add a 'Collection' or an 'Element' to a pattern model, open the 'Pattern Model Designer', and simply drag and drop the shape from the toolbox onto the design surface.

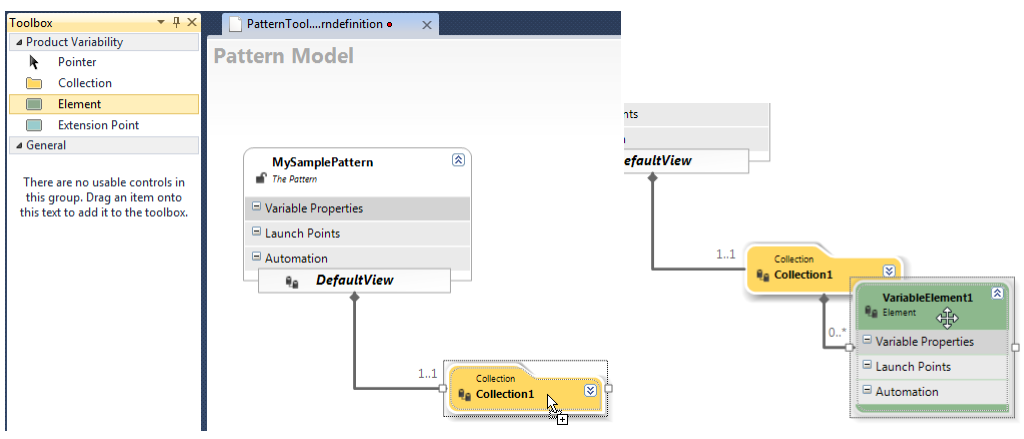


IMPORTANT: It matters where you drop the shape!

If you drag and drop the shape onto the canvas, then the 'Collection' or 'Element' automatically becomes a child of the pattern in that 'View'.



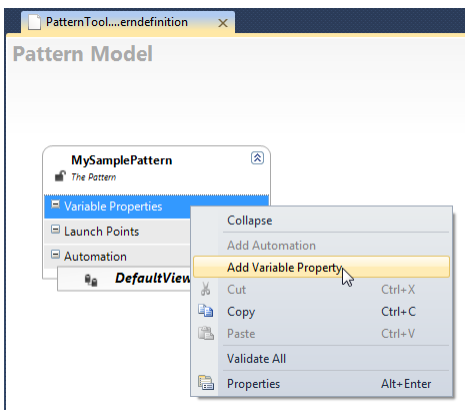
If you drag and drop the shape onto another shape, then the 'Collection' or 'Element' becomes a child of that shape.



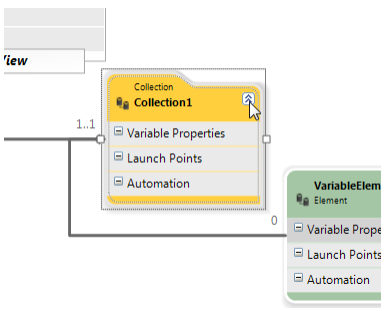
Adding a Variable Property

To create a 'Variable Property' on the pattern, an element, or a collection, right-click on the 'Variable Properties' compartment and click 'Add Variable Property'.

Simply type its name.

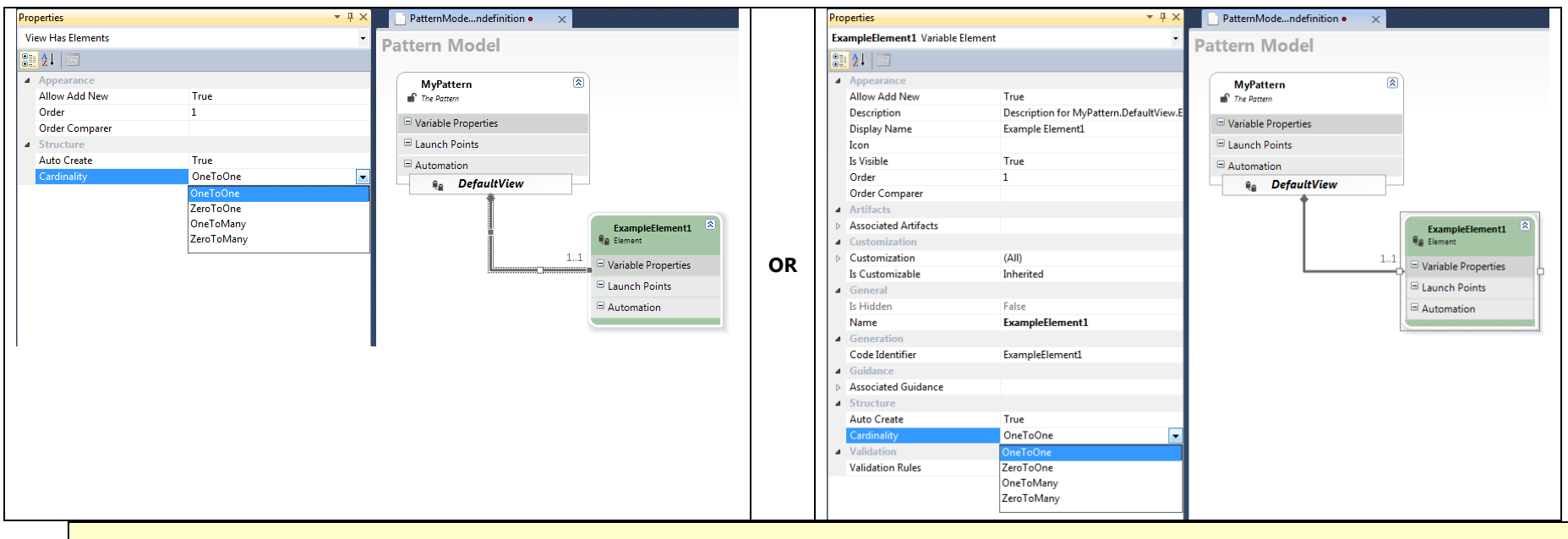


> **Note**: For 'Collections', you need to expand the shape to see the 'Variable Properties' compartment.



Changing the Cardinality

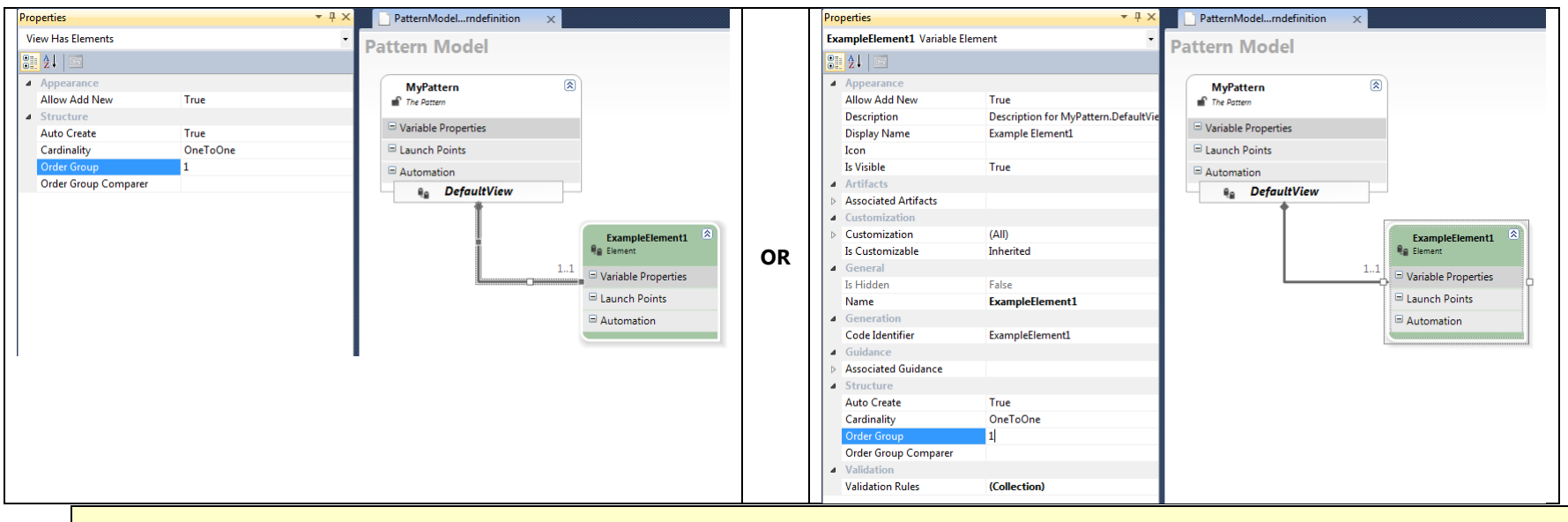
You change the cardinality of the relationship between the 'View', and its child 'Collection' or 'Element' or 'Extension Point' shapes, or between 'Collection', 'Element' and 'Extension Point' shapes, by changing the 'Cardinality' property of either the connector between the shapes, or on the properties on the child shape.



Note: By default, a Collection is created with a 'Cardinality' of 'OneToOne'. An 'Element' and an 'Extension Point' are created with a 'Cardinality' of 'ZeroToMany'.

Changing the Ordering

You change the ordering of element instances on the relationship between the 'View', and its child 'Collection' or 'Element' or 'Extension Point' shapes, or between 'Collection', 'Element' and 'Extension Point' shapes, by changing the 'Order Group' and 'Order Group Comparer' properties of either the connector between the shapes, or on the properties on the child shape.



Note: By default, a Collection, an Element and an Extension Point are created with an 'Order Group' of '1', which would result in all element instances from sibling Collection, Element and Extension Points being ordered together.
By default, a Collection, an Element and an Extension Point are created with an 'Order Group Comparer' which is blank, which would result in all child instances being ordered alphabetically according to their 'InstanceName'.

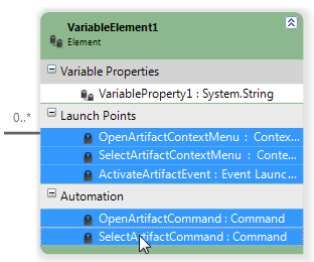
Showing 'Hidden' Elements

A pattern model supports the showing/hiding of elements in the model that are: either not required for manual configuration, or that are automatically configured by automation in the pattern model itself. These elements are not shown to an author to save on real estate and to remove the clutter of these automation elements which the author would otherwise not normally need to view, modify or interact with.

> **Note**: The general rule is not to modify elements that are hidden.

There are several examples of these elements, most of which are provided by automation extensions to aid configuration of a pattern model.

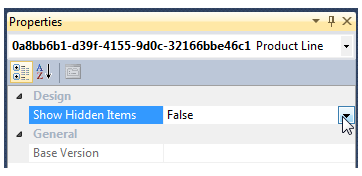
For example, when you configure the 'Associated Artifacts' or 'Associated Guidance' properties of a pattern element, several Command and Launch Points are automatically configured on the current element, which are normally hidden from view.



Showing/Hiding

You can show the hidden items by changing the value of the 'Show Hidden Items' property on the design surface of the pattern model.

Click the pattern model designer surface, anywhere but on a shape, and change this property.



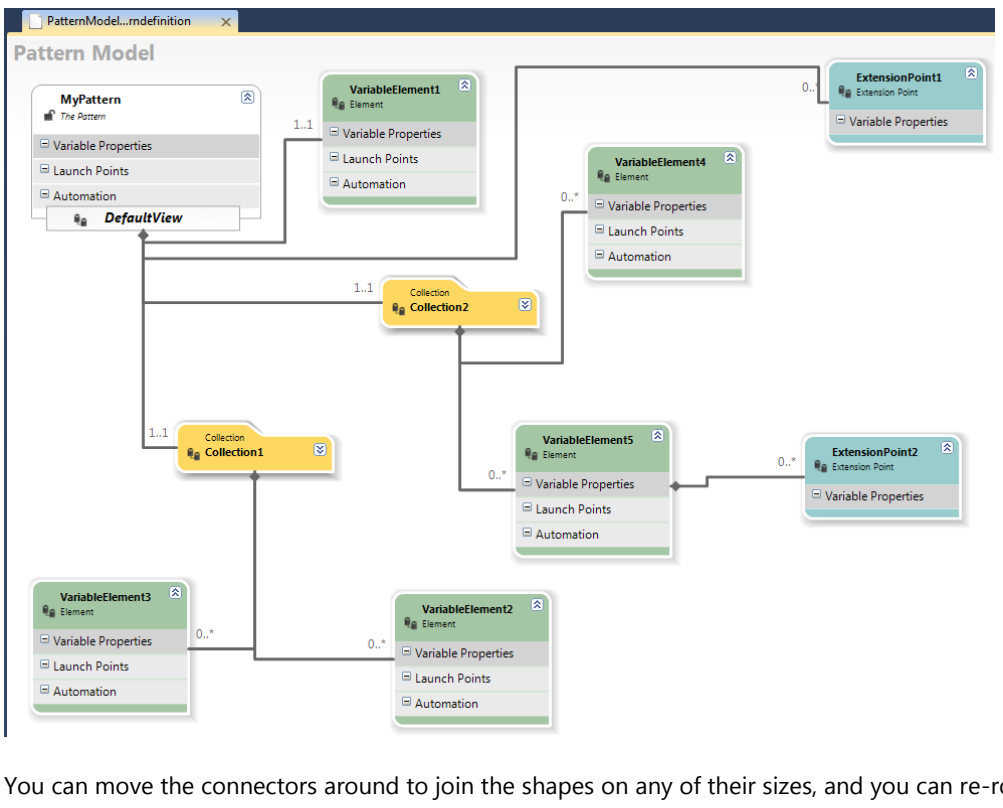
> **Note**: Showing hidden items usually expands the shapes of the diagram interfering with the current layout of the pattern model. Hide these items to restore the original layout.

Arranging Shapes on Pattern Model

Free Layout

With the exception of the 'Pattern' element (and the attached 'View' element) which are fixed in the top left corner of the 'Pattern Model', you are free to arrange your 'Element', 'Collection' and 'Extension Point' shapes in any arrangement on the diagram.

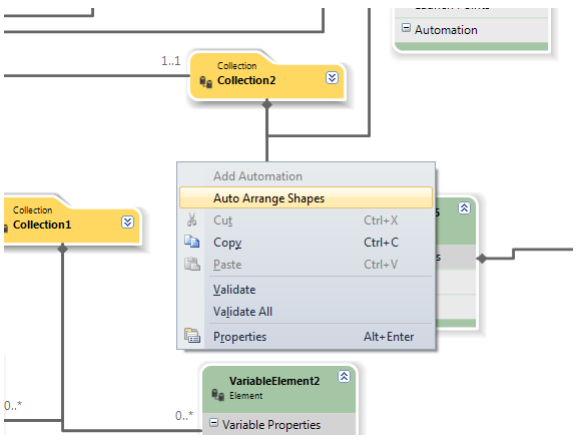
For example:



You can move the connectors around to join the shapes on any of their sizes, and you can re-route the connectors manually if they don't agree with your layout needs.

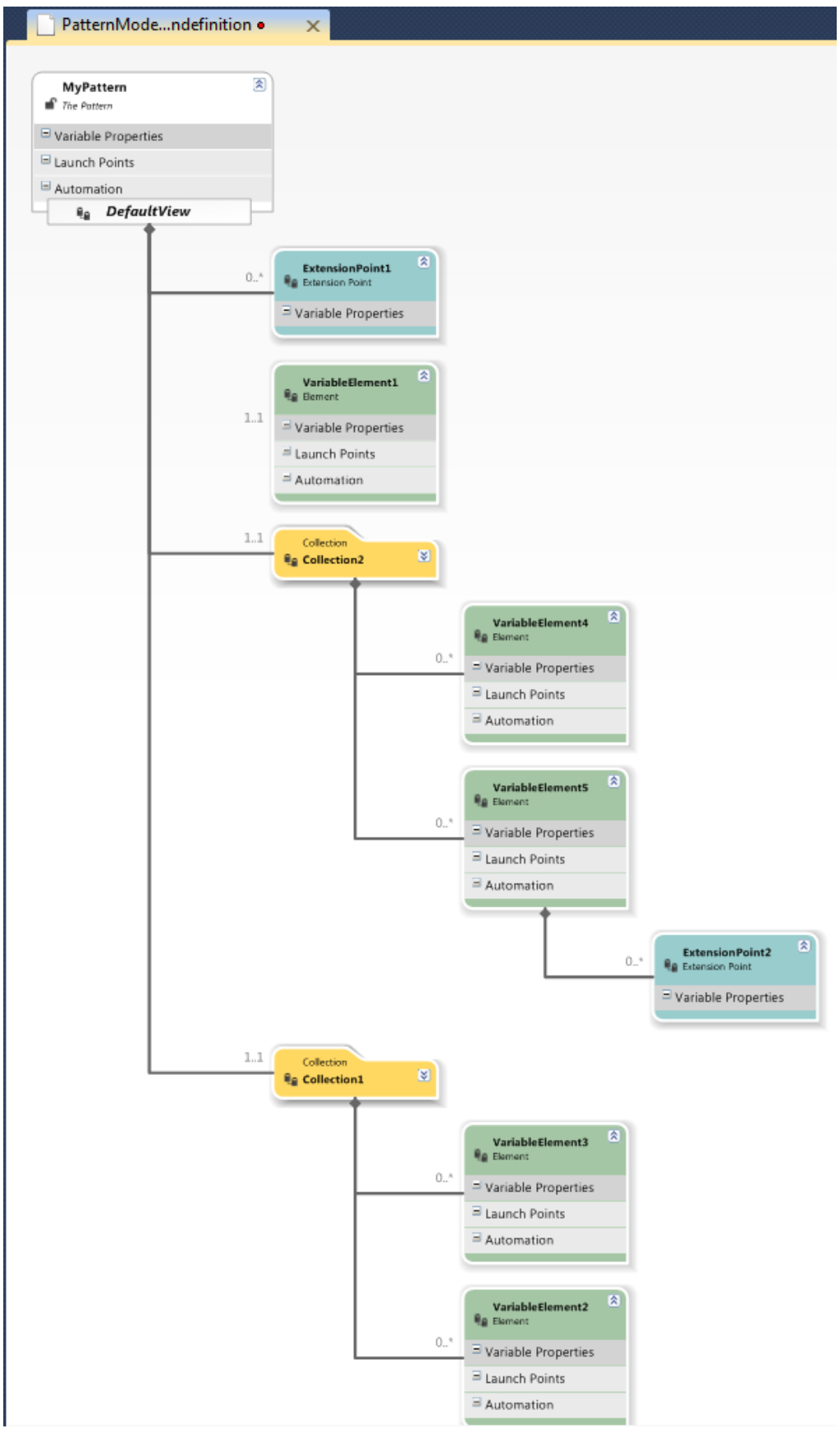
Hierarchical Layout

If you want to arrange your shapes in a hierarchical structure that favors vertical scrolling as opposed to horizontal scrolling to pan around, you can use the 'Auto Arrange Shapes' context menu anywhere on the diagram.



This will automatically sort and arrange your shapes into the hierarchical arrangement, sorted by vertical precedence.

For example:

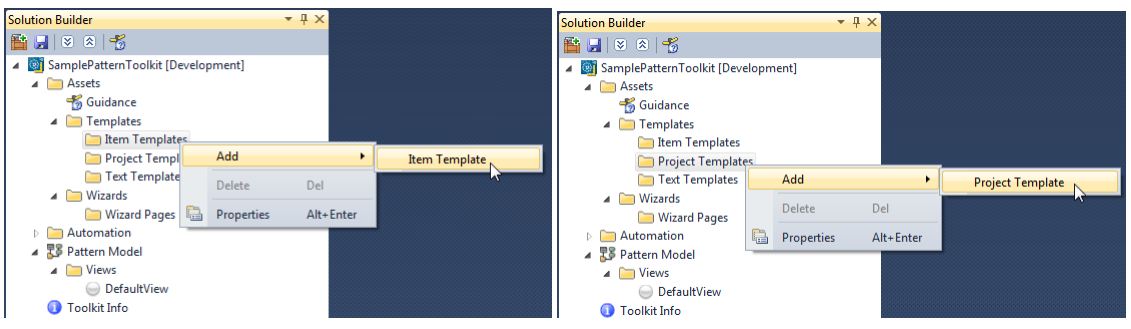


> **Note**: If you wish to modify the ordering of the shapes in the hierarchy, simply drag them higher up the page than their sibling shapes, and use the 'Auto Arrange Shapes' menu again.
> **Note**: If you wish to undo any auto-arrangement, simply use the 'Undo' menu in Visual Studio.

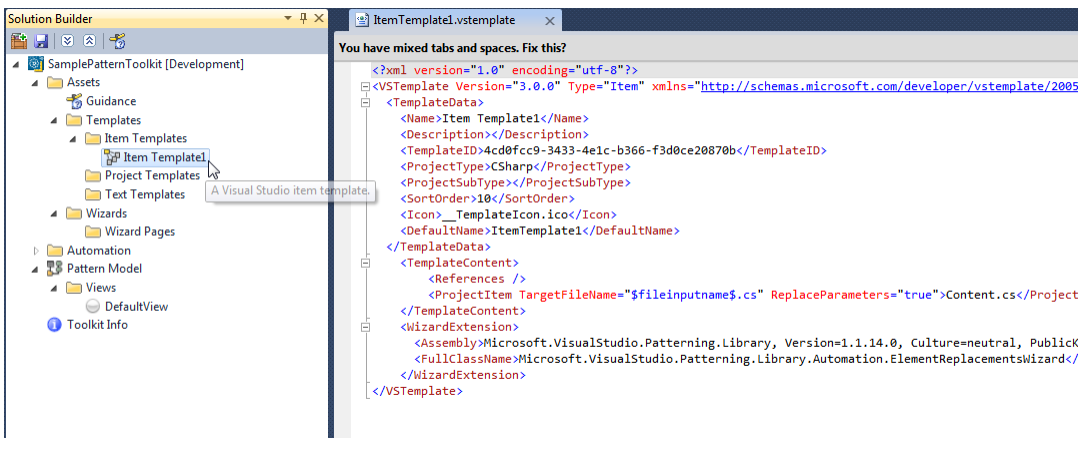
Assets

Adding a VS Template

To add either a VS item template or a VS project template to your toolkit project, in 'Solution Builder', right-click on the 'Templates' folder in the 'Assets' folder and create new one.

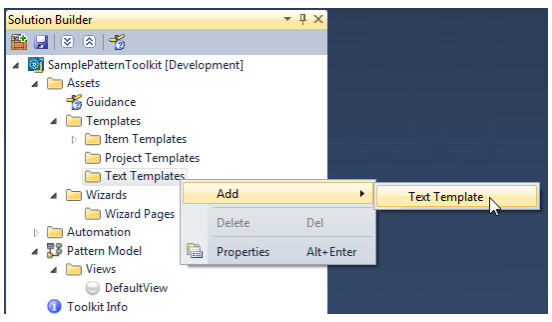


Then double-click on the newly created element, and the *.vstemplate file will open for editing.

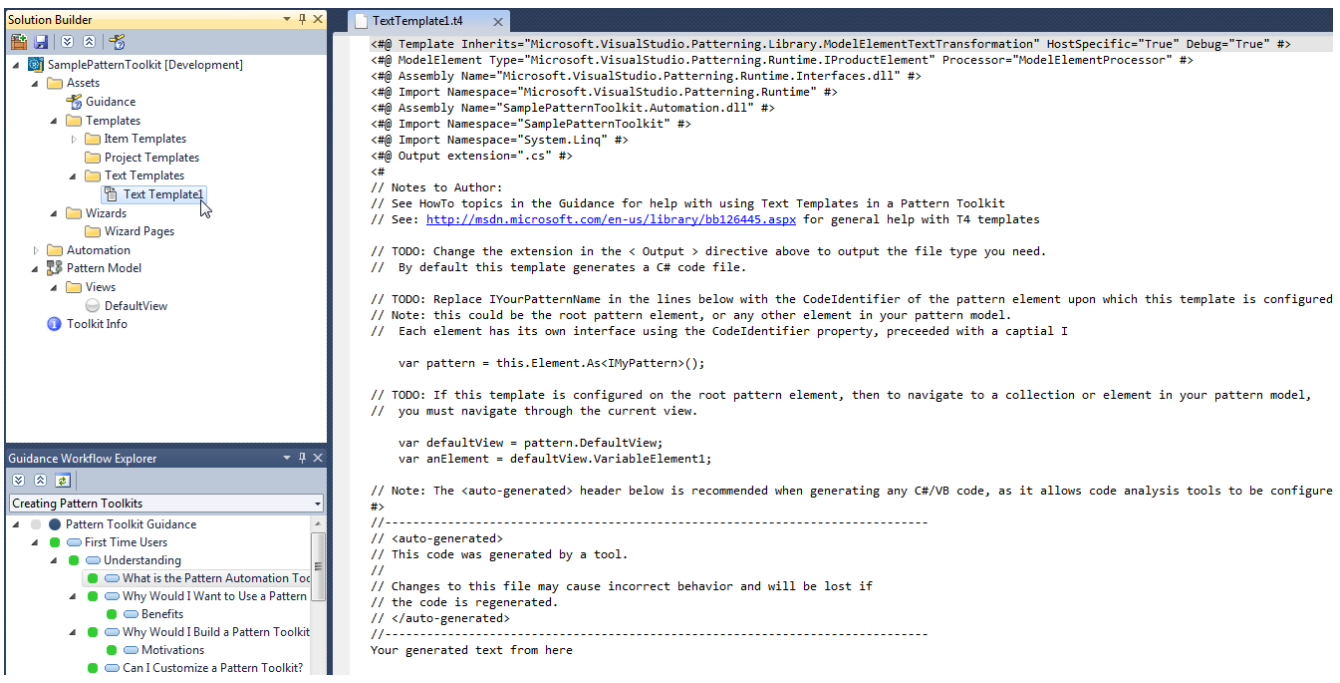


Adding a Text Template

To add a T4 text template to your toolkit project, in 'Solution Builder', right-click on the 'Templates' folder in the 'Assets' folder and create new one.



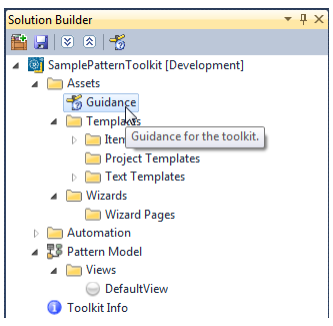
The *.t4 file will open for editing.



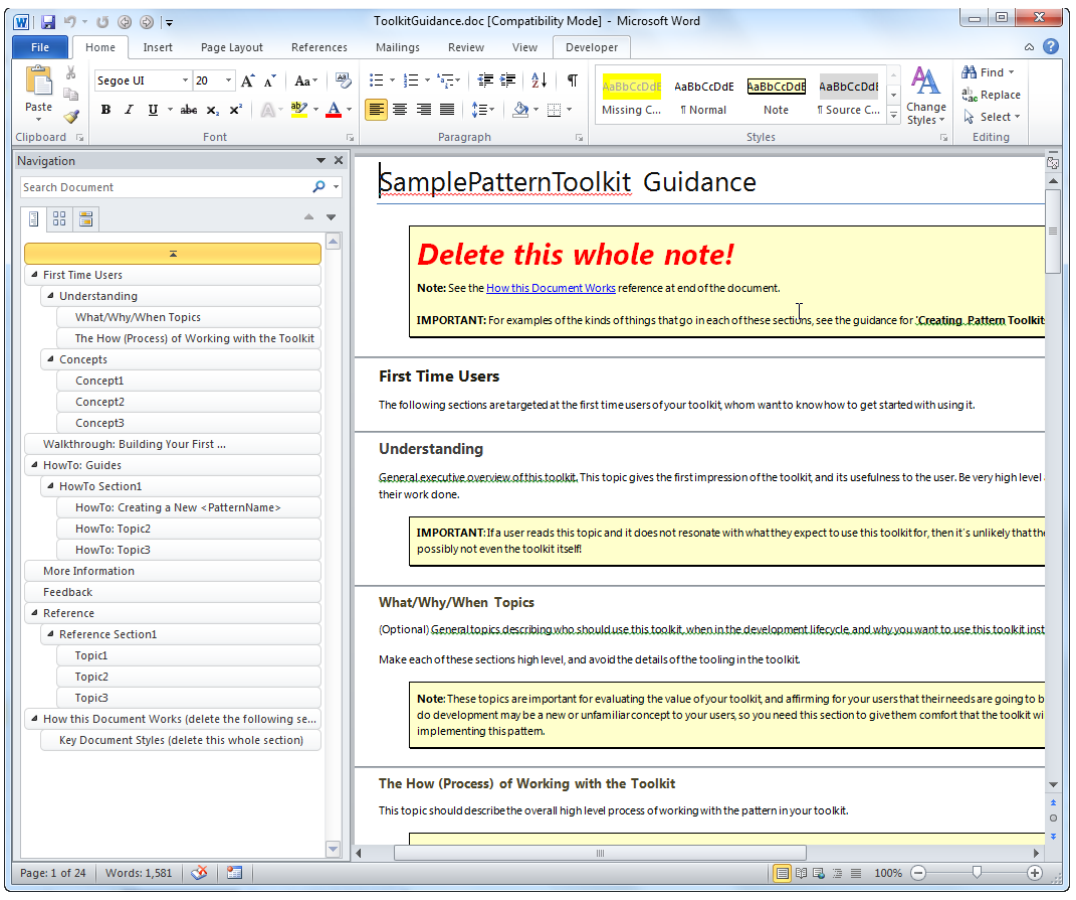
Guidance

Editing Guidance

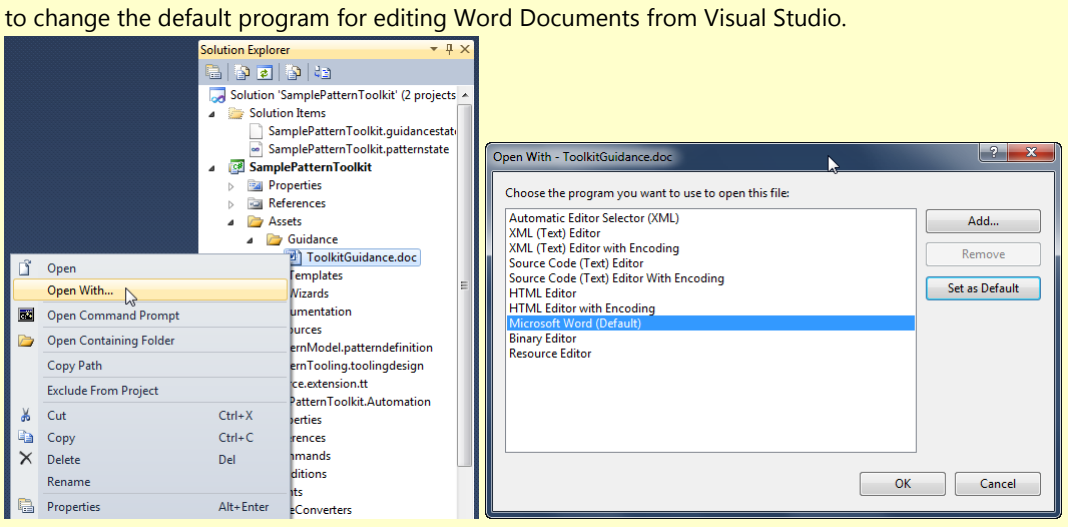
To edit the guidance provided for each toolkit project, in 'Solution Builder', double-click on the 'Guidance' element.



The guidance document will open for editing in Microsoft Word.



Note: If the document does not open in Microsoft Word, and instead opens in Visual Studio, then either Microsoft Word is not installed on your machine, or you need to change the default program for editing Word Documents from Visual Studio.



Once this setting is applied, the document should then open in Microsoft Word.

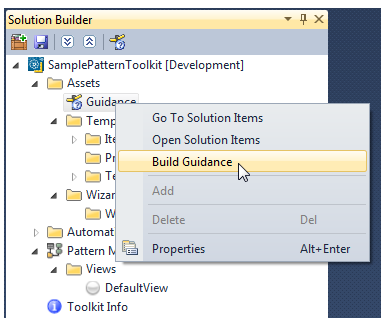
Once the guidance has been created, close and save the document.

Then proceed to Building Guidance.

Building Guidance

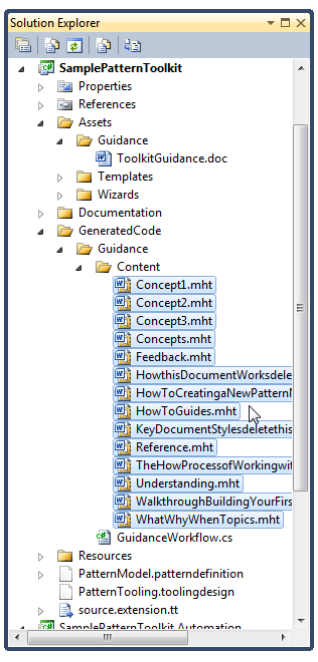
> **Note**: To build the guidance document, it must be saved and closed in Microsoft Word.

To build the guidance document provided for each toolkit project, in 'Solution Builder', right-click on the 'Guidance' element, and select 'Build Guidance'.

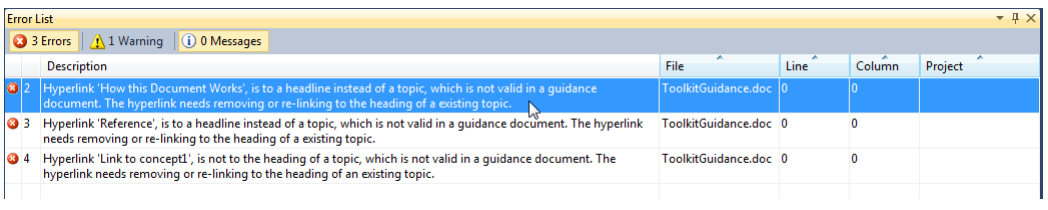


> **Note:** The building of guidance may take quite some time, especially if there are many guidance topics to generate.

If the guidance document is correctly formatted, the guidance document is shredded into many other documents generated into the toolkit project.



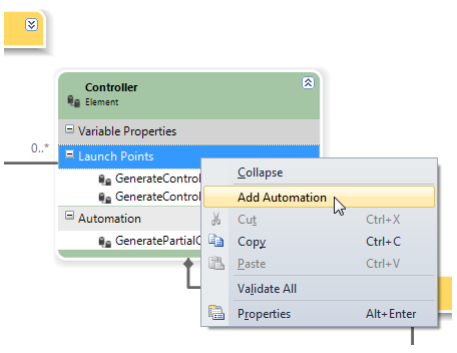
If the guidance document is not correctly formatted, errors will be displayed in the 'Error List'. These errors will need to be resolved before the guidance document can be built.



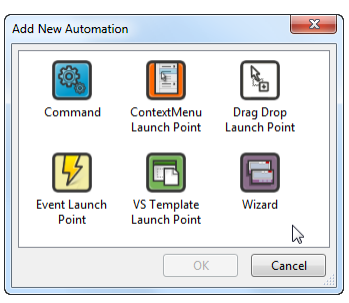
Automation

Adding a Launch Point

To add any 'Launch Point' in the Pattern Model, right-click on either the 'Automation' or 'Launch Points' compartments of an element and select 'Add Automation'.



This displays the 'Add New Automation' dialog, where you select the 'Launch Point' to add.



Once added, simply type the name of the launch point.

See Configuring a Launch Point to configure it with automation that it executes.

Configuring a Launch Point

The following launch points are supported:

- Configuring a Template Launch Point
- Configuring an Event Launch Point
- Configuring a Context Menu Launch Point
- Configuring a Drag Drop Launch Point

Configuring a Template Launch Point

Note: In order to configure a 'Template Launch Point', you must have already created new or imported an existing VS Item or Project Template into the Pattern Toolkit project.

1. Configure the 'Template' property by selecting an existing *.vstemplate file from the assets in the toolkit project

Note: When you select the *.vstemplate file from the template picker dialog, the contents of the *.vstemplate file are automatically re-configured, and synchronized with the values of the current element. All content files of the template are also configured so that when the solution is built the VS template archive is automatically created and packaged in the toolkit.

2. Optionally, configure the 'Target File Name' property to be the default name you wish to give the item or project being unfolded.

Note: If left blank, the item or project being unfolded from the template will assume the actual name of the instance of the element.

Tip: You can also define expressions in the value of the 'Target File Name' property to substitute property values in calculated names. (i.e. {InstanceName} or {Parent.PropertyName}. See the Expression Syntax for more details). Or you can use a Value Provider to calculate the name.

3. Optionally, configure the 'Target Path' property to determine where in the solution the project or item will be unfolded.

Note: If left blank, then the path defaults to '~', and will unfold into the solution item of the first ancestor element found, or if, none found then unfold to the solution. See the Target Path Syntax for more details.

Tip: You can also define expressions in the value of the 'Target Path' property to substitute property values in calculated paths. (i.e. {InstanceName} or {Parent.PropertyName}. See the Expression Syntax for more details). Or you can use a Value Provider to calculate the path.

4. Optionally, configure the 'Unfold When Created' property, to automatically unfold the template whenever an instance of the element is created.

Note: By default this is true. If false, then this template will never unfold when the user creates an instance of the element in solution builder.

5. Optionally, configure the 'Create When Unfolded' property, to automatically create an instance of the element whenever the template is manually used to create or add a project or item from the 'Add New Project/Item' dialog.

Note: By default this is true. If false, then this template can be used to create an item or project but no element instance will be created for it.

Note: If both 'Create When Unfolded' and 'Unfold When Created' are false, then effectively the template is not associated to any specific element – as there is no dependency between them. This may be a desirable scenario for just including an ancillary project or item template in a toolkit.

6. Optionally, configure the 'Sanitize Name' property, to remove spaces and other non-conventional characters from the name of the project or item unfolded from this template.

Note: By default this is true. If false, then the project or item created from this template will be named with precisely the same name (including all spaces and non-conventional characters) as the name of the instance of the element. This is not normal convention with naming projects and files in a solution.

7. Optionally, configure the 'Sync Name' property, to synchronize changes in the name of the element with a change in the name of the project or item created.

Note: This only has effect when the Target File Name property value includes property substitutions (i.e. {InstanceName} or {Parent.PropertyName}, etc. See the Expression Syntax for more details) or when the value is calculated by a value provider.
Note: By default this is false. If true, then the project or item will be renamed when properties of the element instance in Solution Builder are changed.
Note: For C# code files, when 'Sync Name' is true, and the user changes the name in Solution Builder, a rename/refactor is automatically performed on the class in the code as well.

8. Optionally, configure the 'Tag' property to associate an arbitrary text value to the artifact link being created for this solution item. This tag can then be used by automation to filter for the specific artifact link for this element that is automatically generated.
9. Optionally, configure the 'Wizard' property to display a wizard before the template is unfolded to gather values from the user that could be used to substitute into the content of the template's files or as filenames.
10. Optionally, configure the 'Command' property to execute a command after the template has unfolded.

Configuring an Event Launch Point

1. Configure the ‘Event Type’ property by selecting a triggering event form the list.
2. Configure the ‘Command’ property to execute a command after the event has been raised.

> **Note**: The ‘Command’ is optional if a ‘Wizard’ is configured.

3. Optionally, configure the ‘Current Element Only’ property

> **Important**: By default this is false. This property must be true if an element specific ‘Event Type’ is selected (e.g. ‘Element Instantiated’). If this property is false for an element specific event, then the event is raised for every instance of every element, and will lead to unexpected results.
>
> **Note**: When this property is true, an automatic filtering condition is added to the ‘Conditions’ property to ensure that the event is only handled for instances of the specific element upon which the launch point is configured.

4. Optionally, configure the ‘Conditions’ property to define conditions that constrain the event from being handled.
5. Optionally, configure the ‘Wizard’ property to display a wizard when the event is raised to gather values from the user that could be used to configure properties of the pattern, before the command is executed.

Configuring a Context Menu Launch Point

1. Configure the ‘Menu Text’ property to define the caption of the menu displayed to the user.
2. Configure the ‘Command’ property to execute a command when the menu is clicked.
3. Optionally, configure the ‘Icon’ property for the menu.
4. Optionally, configure the ‘Menu Order’ property to position the menu relative to other menus you have on this element.

> **Note**: All menus that are displayed, on all elements, are sorted first by their ‘Menu Order’ value and then alphabetically in their ‘Menu Text’ value.
> If you want your menu to appear higher up the list then give it a lower ‘Menu Order’ value.

5. Optionally, configure the ‘Conditions’ property to define conditions that constrain the menu from being shown.

> **Note**: If the conditions do not evaluate, then the menu is not visible.

6. Optionally, configure the ‘Wizard’ property to display a wizard when the menu is clicked to gather values from the user that could be used to configure properties of the pattern, before the command is executed.

Configuring a Drag Drop Launch Point

Note: In order to configure a Drag and Drop Launch Point, you must have already created a custom 'Drop Command' to handle the data being dropped, and a custom 'Drag Condition' that determines if the data contains valid data to drop on the element.

1. Configure the 'Drop Command' and 'Drag Conditions' properties with the custom types above.

Note: If no conditions are configured, then any and all data will be permitted to be dragged and dropped on instances of the current element.
This may leave the user confused as to whether the drag and drop was successful or not, and whether the data being dragged was valid or not.

2. Optionally, configure the 'Status Text' property to display a message to the user when dragging the data over instances of the current element.

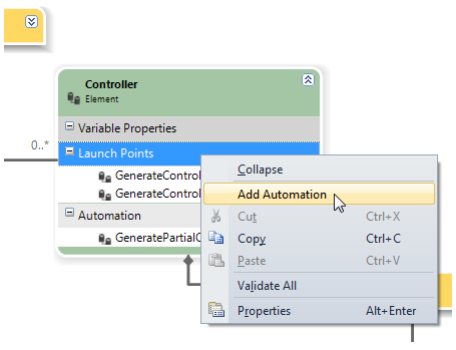
Tip: The 'Status Text' property value is displayed in the status bar of Visual Studio when data is being dragged over the current element. This text value can be either static (i.e. defined as a fixed string), or a Value Provider can return the text dynamically.

3. Optionally, you can configure a custom 'Wizard' to be displayed before the 'Drop Command' command is executed.

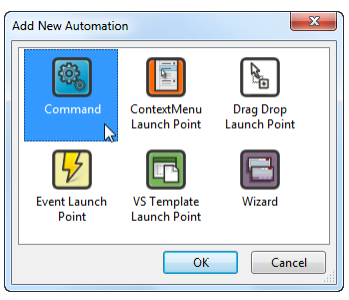
Adding a Command

> **Note**: In order to add and configure a 'Command' automation completely, you must use an available command type, or have already created a custom 'Command', and have built the solution.

To add a 'Command' in the Pattern Model, right-click on either the 'Automation' or 'Launch Points' compartments of an element and select 'Add Automation'.

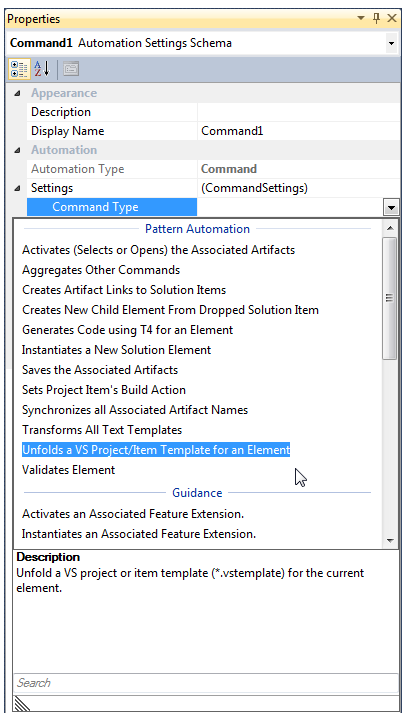


This displays the 'Add New Automation' dialog, where you select the 'Command' shape.

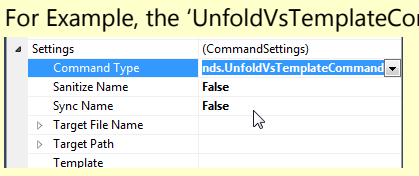


Once added, simply type the name of the command.

In the 'Command Type' property of the 'Settings', select the specific type of command to execute.



> **Note**: If the command supports any additional properties, configure the values of those properties.
> For Example, the 'UnfoldVsTemplateCommand' command type supports number of optional and mandatory properties it requires to execute correctly.



See Configuring a Command to configure it with automation that executes it.

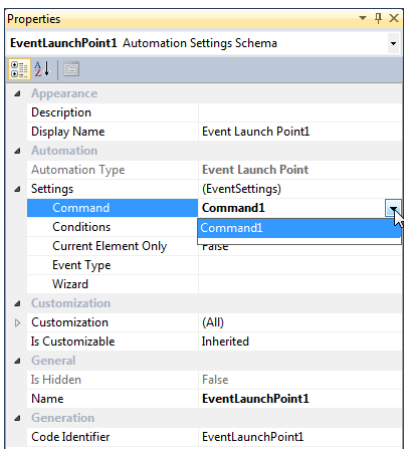
Configuring a Command

> **Note**: To configure a Command, you must have already added a 'Command Automation' to an element in the pattern model.

Commands are used in the configuration of the following Launch Points:

- Template Launch Point
- Event Launch Point
- Context Menu Launch Point
- Drag Drop Launch Point

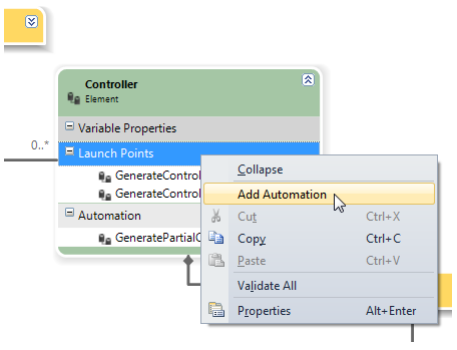
In the 'Settings' property of the launch point, select the 'Command' from the list of commands already added to this element.



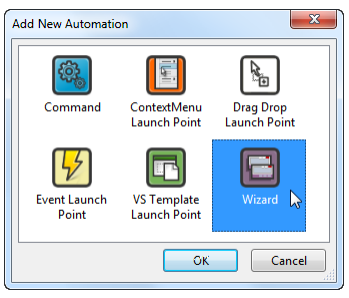
Adding a Wizard

Note: In order to add and configure a 'Wizard' automation completely, you must have already created a custom 'Wizard' (XAML) and one or more custom 'Wizard Pages' (XAML) to show in that wizard, and have built the solution.

To add a 'Wizard' in the Pattern Model, right-click on either the 'Automation' or 'Launch Points' compartments of an element and select 'Add Automation'.

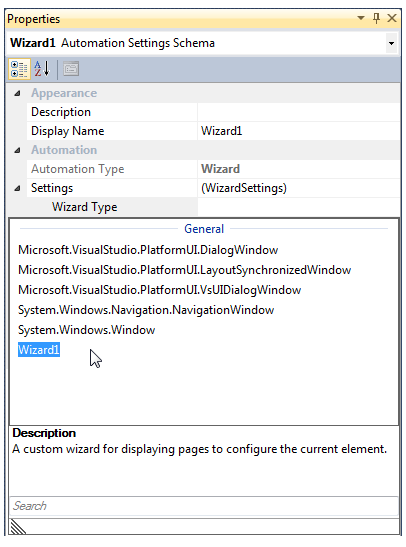


This displays the 'Add New Automation' dialog, where you select the 'Wizard' shape.



Once added, simply type the name of the wizard.

In the 'Wizard Type' property of the 'Settings', select the specific type of wizard to display.



See Configuring a Wizard to configure it with automation to display it.

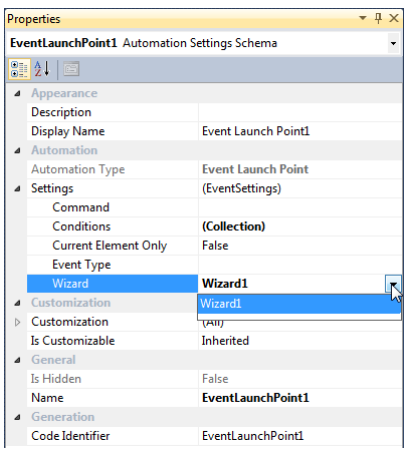
Configuring a Wizard

> **Note**: To configure a Wizard, you must have already added a 'Wizard Automation' to an element in the pattern model.

Wizards are used in the configuration of the following Launch Points:

- Event Launch Point
- Context Menu Launch Point

In the 'Settings' property of the launch point, select the 'Wizard' from the list of wizards already added to this element.



Common Automation Tasks

There are a number of common custom automation tasks that can be performed in custom automation classes for manipulating solution items that are referenced by elements in a pattern model.

These are examples of the recommended code required to perform these tasks.

> **Note**: NuPattern ships two 'authoring' toolkits that use automation to help you build your pattern toolkit project. The capabilities provided by these toolkits can be leveraged also in your toolkit. You can see by example the kinds of automation the authoring toolkits provide and can use them as examples to help understand how to write your own automation. Although you are not provided the source code for those toolkits, you can learn from their disassembled implementation using a disassembler such as the .NET Reflector.

Resolving Target Paths

Many of the provided commands used in pattern toolkits allow configuration of a 'Target Path' or 'Project Path' property that resolves from an element to a solution item through an artifact link found by traversing the configured path. The syntax for these kinds of paths is described in Target Path Syntax.

You can provide a similar kind of property on any automation class that asks a toolkit author to provide a target path, and have it resolved to an actual solution item that your automation can then perform additional actions with.

Add References

- NuPattern.Extensibility.dll
- NuPattern.Runtime.Interfaces.dll;
- Microsoft.VisualStudio.TeamArchitect.PowerTools.dll

Declare the Property and Imports

```
[Required(AllowEmptyStrings = false)]
public string TargetPath { get; set; }

[Import(AllowDefault = true)]
public IProductElement CurrentElement { get; set; }
[Import(AllowDefault = true)]
public IFxrUriReferenceService UriService { get; set; }
[Import(AllowDefault = true)]
internal ISolution Solution { get; set; }
```

Implement Code

```
var resolver = new PathResolver(this.CurrentElement, this.UriService, this.TargetPath);
resolver.Resolve();
if (!string.IsNullOrEmpty(resolver.Path))
{
    var solutionItem = this.Solution.Find(resolver.Path).FirstOrDefault();
}
```

Resolving Expressions

Many of the provided commands used in pattern toolkits allow configuration of a 'Target File Name' or similar property that resolves a pattern author defined expression and substitutes values from properties of element instances in the pattern. The syntax for these kinds of paths is described in Expression Syntax.

You can provide a similar kind of property on any automation class that asks a toolkit author to provide an expression, and have it resolved to actual values of properties that your automation can then perform additional actions with.

Add References

- NuPattern.Extensibility.dll
- NuPattern.Runtime.Interfaces.dll;
- Microsoft.VisualStudio.TeamArchitect.PowerTools.dll

Declare Property and Imports

```
            [Required(AllowEmptyStrings = false)]
            public string PropertyName { get; set; }

            [Import(AllowDefault = true)]
            public IProductElement CurrentElement { get; set; }
```

Implement Code

```
                    var evaluatedValue = ExpressionEvaluator.Evaluate(this.CurrentElement, this.PropertyName);
```

Resolving Artifact Links

For any element in a pattern model that has one or more artifact links associated to it, you can resolve those links to solution items.

Add References

- NuPattern.Extensibility.dll
- NuPattern.Runtime.Interfaces.dll;
- Microsoft.VisualStudio.TeamArchitect.PowerTools.dll

Declare Property and Imports

```
            [Import(AllowDefault = true)]
            public IProductElement CurrentElement { get; set; }
            [Import(AllowDefault = true)]
            public IFxrUriReferenceService UriService { get; set; }
            [Import(AllowDefault = true)]
            internal ISolution Solution { get; set; }
```

Implement Code

```
                var references = SolutionArtifactLinkReference.GetResolvedReferences(this.CurrentElement, this.UriService);
                var item = references.Where(solutionItem => Path.GetExtension(solutionItem.Name) == ".xml").FirstOrDefault();
```

Note: This example finds the first artifact link that refers to an .XML file in the solution.

Adding Artifact Links

You can explicitly add an artifact link to any element to any item in the solution.

Add References

- NuPattern.Extensibility.dll
- NuPattern.Runtime.Interfaces.dll;
- Microsoft.VisualStudio.TeamArchitect.PowerTools.dll

Declare Property and Imports

```
        [Import(AllowDefault = true)]
        public IProductElement CurrentElement { get; set; }
        [Import(AllowDefault = true)]
        public IFxrUriReferenceService UriService { get; set; }
        [Import(AllowDefault = true)]
        internal ISolution Solution { get; set; }
```

Implement Code

```
    var item = this.Solution.Traverse()
    .Where(solutionItem => !string.IsNullOrEmpty(solutionItem.PhysicalPath) && solutionItem.PhysicalPath.EndsWith(".zip"));
    SolutionArtifactLinkReference.AddReference(this.CurrentElement, UriService.CreateUri(item));
```

Note: This example adds an artifact link to first .ZIP file it finds in the solution.

Source Control Integration

You can ensure that changes to files in the solution are correctly tracked by source control by checking out the file before modifying its content.

Add References

- NuPattern.Extensibility.dll
- Microsoft.VisualStudio.TeamArchitect.PowerTools.dll

Implement Code

```
        var item = this.Solution.Traverse()
        .Where(solutionItem => !string.IsNullOrEmpty(solutionItem.PhysicalPath) && solutionItem.PhysicalPath.EndsWith(".zip"));
        item.Checkout();
```

Adding Properties to Automation Classes

You declare custom properties on your automation classes to make your class configurable by a toolkit author when they configure your automation on an element in their pattern model.

These properties gather values at design-time from authors and persisted in the pattern model. At runtime, when a user invokes the automation the properties values are used to configure the behavior of the automation class.

> **Note**: Not only do properties allow authors to specify 'static' configuration at design-time, but each property can also be configured to use a 'Value Provider' instead to fetch its value dynamically. It goes without saying that the configured 'Value Provider' on the property can also have configurable properties itself.

Declaring a Property

To declare a property, simply define a public property on the automation class with a public getter and public setter.

```
        public string MyProperty { get; set; }
```

> **Note**: All public properties of an automation class are considered configurable by an author, unless they only have a getter, or have attributes that make them read-only or hide them at design time.

Properties that must have values defined at runtime for safe execution can declare that they are required using the [Required] attribute. This attribute ensures that the automation is not executed if the value of this property is not defined.

```
        [Required(AllowEmptyStrings = false)]
        public string MyProperty { get; set; }
```

> **Note**: You can also specify other System.ComponentModel.DataAnnotations attributes such as the [RegularExpression] attribute to define other constraints on the legal configured safe value to execute the automation class.
> **Note**: If a required (or other validation attribute) property is not satisfied at the time the automation is executed, the automation framework will not execute the automation, and will both notify the user of the problem, and trace the issue to the Tracing Window automatically.

Properties that declare default values can declare those using the [DefaultValue] attribute, but the initial value also must be initialized in the constructor of the automation class.

```
        [Required(AllowEmptyStrings = false)]
        [DefaultValue("Some Value")]
        public string MyProperty { get; set; }
```

> **Note**: The [DefaultValue] attribute simply declares what the default is, which may lead to a visual distinction of the value it when displayed; it does **not** actually set the value to the default value. You must still add code to set the default value in the constructor of the class.

```
        private const bool DefaultMyPropertyValue = false;

        public MyAutomationClass()
        {
            this.MyProperty = DefaultMyPropertyValue;
        }

        [DefaultValue(DefaultMyPropertyValue)]
        public bool MyProperty { get; set; }
```

Properties that are displayed to authors should use the [DisplayName] and [Description] attributes to give the author information about what they are used for and how they are to be configured.

```
        [DisplayNameResource("Solution Path")]
        [DescriptionResource("The path to the project in the solution where this file will be generated.")]
        public string TargetPath { get; set; }
```

Properties can be of any type, but in order to be displayed and persisted correctly they must be convertible from their native type to a string type using a TypeConverter defined on the property.

```
        [TypeConverter(typeof(ColorConverter))]
        public Color BackgroundColor { get; set; }
```

For properties of special types, in order to be displayed as to the author correctly (using the appropriate editor controls) they must use a Type Editor which also must be defined on the property.

```
        [TypeConverter(typeof(ColorConverter))]
        [Editor(typeof(ColorEditor), typeof(UITypeEditor))]
        public Color BackgroundColor { get; set; }
```

> **Note**: Common value types such as string, boolean, integer etc. don't require you to specify a TypeConverter or TypeEditor. Those converters and editors are already built-in.

For simple properties where you don't want to forbid an author to configure a 'Value Provider' for its value, you can use the [DesignOnly] attribute.

```
        [DesignOnly(true)]
        public bool MySimpleProperty { get; set; }
```

Supporting Special Property Values

Many of the provided automation classes, allow an author to specify certain special syntaxes of values for certain properties that are pre-processed at runtime to resolve to other elements in the pattern model or items in other coordinate systems, such as the Expression Syntax or Target Path Syntax.

The ability to support these syntaxes and resolve to other things is not built-in to all properties by default, and must be programmed in each case in each automation class.

There is support that helps you easily support these syntaxes on your custom properties in their automation classes.

See the resolving topics in Common Automation Tasks for more details.

Importing Services into Automation Classes

You declare imports to various services in Visual Studio (available through MEF) on your automation classes to give your class access to these services in order to build rich automation.

These imports assume the service has already been exported at the time the automation class is invoked.

> **Note**: Prior to Visual Studio 2010, it was common to obtain an instance of an IServiceProvider and call the GetService() method on it to obtain the type of service you required. This practice is no longer required for the most part. Now most services in Visual Studio are exported to MEF, and you only need to import them directly. However, there may be some less common services which are still not exported to MEF and only accessible through an IServiceProvider. In these cases, simply import the IServiceProvider and then call GetService() method from there.

Commonly Imported Services

The following are some of the common services you can import into your automation classes for integrating your automation with the Visual Studio development environment.

`IProductElement`	The current element, un-typed to your pattern. **Note**: Use the .As<T>() method to cast to the type of your current element. (see below)
`IProperty`	The current variable property, if the automation is set on a variable property in the pattern model. (i.e. for validation rules, value providers, etc.)
`IEvent<EventArgs>`	The current event being raised that triggered the automation (i.e. for conditions)
`IPatternManager`	The global service for managing pattern instances (as seen in the 'Solution Builder' window).
`ICommandSettings`	The currently configured settings on the current automation.
`ISolution`	The global service for the current solution (as seen in the Visual Studio 'Solution Explorer' window).
`IUserMessageService`	The global Visual Studio service for displaying UI, which can be used to display message boxes etc.
`IFxrUriReferenceService`	The global service to help dereference URI's, such as artifact links, guidance links etc.
`IFeatureManager`	The global service for managing guidance workflows (as seen in the Visual Studio 'Guidance Workflow Explorer' window)
`IServiceProvider`	The global Service Provider in Visual Studio (see 'Special Services' below). **Note**: You can use this service provider to reach any other service (i.e. the 'DTE') in Visual Studio. See the GetService() methods.
`DragEventArgs`	The current dragged and dropped data.
`IStatusBar`	The global Visual Studio 'Status Bar', which can be used to add progress messages.
`IErrorList`	The global Visual Studio 'Error List', which can be used to add error, warning or information messages. **Note**: As well as adding your own items to this list, you will also need to manage clearing them from the list.

Declaring an Import

To import a service, define a public property on your automation class and decorate it with the [Import] attribute.

```
[Import(AllowDefault = true)]
public ISolution Solution { get; set; }
```

> **Note**: Use the "AllowDefault" parameter to ensure the import gets its default value if not satisfied.

For imports that are required to be not null when the automation is executed, use the [Required] attribute.

```
[Required]
[Import(AllowDefault = true)]
public ISolution Solution { get; set; }
```

> **Note**: If a required import is not satisfied at the time the automation is executed, the automation framework will not execute the automation, and will both notify the user of the problem, and trace the issue to the Tracing Window automatically.

Importing Special Services

There are a number of noteworthy services that require slightly modified import statements to work correctly in Visual Studio. Importing IServiceProvider and several other legacy Visual Studio services require you define explicitly the type of service being imported.

The correct way to import IServiceProvider is:

```
[Import(typeof(SVsServiceProvider), AllowDefault = true)]
public IServiceProvider ServiceProvider { get; set; }
```

Importing the Current Element into Automation Classes

It is almost always the case that an automation class will need access to the underlying pattern model upon which it has been configured, and in many cases to the specific element upon which it was configured.

For convenience, the current element can be imported into the automation class the same way as other services in Visual Studio.

There are in fact always two imports that are valid for the current element. One to the element in using the 'Generic' interface scheme, and the other using the generated 'Typed' scheme. Depending on how general or reusable your automation class is to be, will determine which scheme you use in the specific automation class.

> **Note**: You won't need to declare both imports; you can always convert from one to the other. See Moving Between Generic and Typed Pattern Model Schemes.

Declaring the 'Generic' Scheme Import

To import the current element regardless of the specific pattern model, define a public property on your automation class and decorate it with the [Import] attribute as shown.

```
[Required]
[Import(AllowDefault = true)]
public IProductElement CurrentElement { get; set; }
```

> **Note**: The import should always be required and always declared exactly as above.

Within your automation class you can now navigate the pattern model using the generic scheme. See Navigating Pattern Models Programmatically (Generically) for more details.

Declaring the 'Typed' Scheme Import

To import the current element as a specific type in a specific pattern model, define a public property on your automation class and decorate it with the [Import] attribute as shown.

```
[Required]
[Import(AllowDefault = true)]
public IMySpecificElement CurrentElement { get; set; }
```

> **Note**: The import should always be required and always declared exactly as above.
> The type of the property can be inferred from the 'CodeIdentifier' property of the element upon which the automation class is presently configured, pre-fixed with an 'I' character.
>
> **Warning**: When declaring the 'Typed' scheme import, you are asserting that the automation class is not reusable on any other element in the pattern model or with any other pattern toolkit. In other words, this automation class is exclusive to the typed element of the current pattern model.

Within your automation class you can now navigate the pattern model using the typed scheme. See Navigating Pattern Models Programmatically (Typed) for more details.

Navigating Pattern Models Programmatically (Generically)

Once you have a reference to an element in a pattern model in the generic scheme, you can navigate up and down the model in un-typed manner. Note, that you are navigating instances of elements in the current pattern model as the user sees it.

> **Note**: You can move from the generic scheme to the typed scheme if you wish, but doing so makes your automation class bound to a single pattern model. See Moving Between Generic and Typed Pattern Model Schemes for more details.

In the generic scheme there is no type checking. The pattern model is composed of elements with parent/child relationships. Each element implements a number of interfaces depending on how it was configured in its original pattern model. For example, it will implement the IElementContainer interface if it has children elements. It will implement the IView interface if it is a view, etc. All elements will derive from the IInstanceBase interface, which really only gives you access to its parent and some other basic information.

All examples below assume you have an instance of an element in the pattern model, which is commonly imported into your automation class. i.e.

```
public IProductElement CurrentElement { get; set; }
```

Accessing the Parent

To access the parent element of the current instance, use the .Parent property.

```
IInstanceBase parent = this.CurrentElement.Parent;
```

Accessing the Pattern Instance

To access the root product element of any instance, use the .Root property.

```
IProduct product = this.CurrentElement.Root;
```

Accessing Child Elements

To access the children elements of any element, you must use the .Elements property of the IElementContainer interface.

```
IElementContainer element = this.CurrentElement as IElementContainer;
IEnumerable<IAbstractElement> children = element.Elements;
```

> **Note**: The cast to IElementContainer will only succeed for elements that have either child elements or child collections (i.e. IView or IAbstractElement). IAbstractElement is a base interface for both IElement and ICollection.

To access specific children, find children by their DefinitionName (which is the 'Name' of the element in the pattern model).

```
IElementContainer element = this.CurrentElement as IElementContainer;
IAbstractElement childElement = element.Elements.FirstOrDefault(child => child.DefinitionName == "ChildElement1");
```

Accessing Properties

To access the properties of an element, you must use the .Properties property of the IProductElement interface.

```
IEnumerable<IProperty> properties = this.CurrentElement.Properties;
```

To access a specific property, find that property by its DefinitionName (which is the 'Name' of the variable property in the pattern model).

```
IProperty colorProperty = this.CurrentElement.Properties.FirstOrDefault(prop => prop.DefinitionName == "Color");
```

Accessing the Instance Name

To access the instance name of an element, use the InstanceName property of the IProductElement interface.

```
string instanceName = this.CurrentElement.InstanceName;
```

Accessing Schema Information

All elements in a pattern model have additional schema information associated to them, which is the information defined about all instances of that kind of pattern element. This is the information defined in the pattern model.

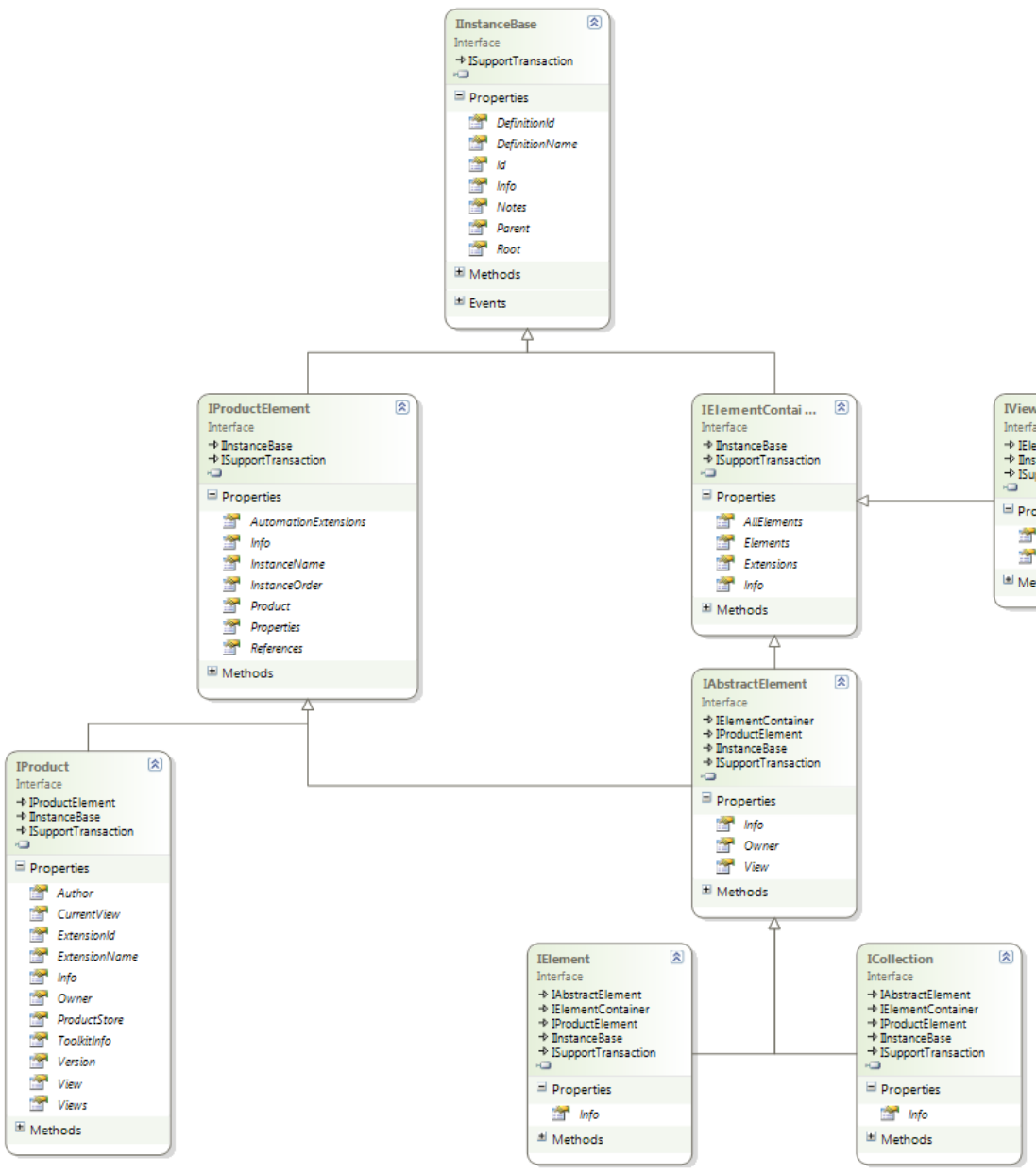
To access the schema information of any element, you use the .Info property.

```
IPatternElementInfo schemaInfo = this.CurrentElement.Info;
```

> **Note**: From the .Info property you can see all the configured descriptive information about the element.

Inheritance Tree

The following graph shows the inheritance tree for a number of the most important interfaces in the generic scheme.



Navigating Pattern Models Programmatically (Typed)

Once you have a reference to an element in a pattern model in the typed scheme, you can navigate up and down the model in typed manner. Note, that you are navigating instances of elements in the current pattern model as the user sees it.

> **Note**: You can move from the typed scheme to the generic scheme as needed to call some of the automation frameworks API's. See Moving Between Generic and Typed Pattern Model Schemes for more details.

In the typed scheme everything (i.e. Parents, Children, Properties etc.) are accessible by name as the type they were defined as in the pattern model, unlike the generic scheme where there was no type checking, and no naming. The pattern model is composed of elements with parent/child relationships. Each element implements a single interface based upon how it was configured in its original pattern model.

All examples below assume you have an instance of an element in the pattern model, which is commonly imported into your automation class. i.e.

```
        public IMyPatternElement1 CurrentElement { get; set; }
```

Accessing the Parent

To access the parent element of the current instance, use the .Parent property.

```
            IParentElement1 parent = this.CurrentElement.Parent;
```

> **Note**: The returned parent is of the type of the parent in the pattern model.
> **Note**: There is no parent property for the pattern element.

Accessing the Pattern Instance

To access the root pattern element of any instance, use the .Parent property successively.

```
            IPattern1 pattern = this.CurrentElement.Parent.Parent.Parent;
```

> **Note**: The parent of a top-most element or collection in a pattern model is actually the view, whose parent is the pattern.

Accessing Child Elements

To access the children elements of any element, use the named collection for the element.

```
            IChildElement1 children = this.CurrentElement.ChildElement1;
```

Accessing Properties

To access the properties of an element, use the named property on this element.

```
            string myPropertyValue = this.CurrentElement.MyProperty1;
```

> **Note**: the returned type of the property will depend on the configured type in the pattern model.

Accessing the Instance Name

To access the instance name of an element, use the InstanceName property.

```
            string instanceName = this.CurrentElement.InstanceName;
```

Accessing Schema Information

All elements in a pattern model have additional schema information associated to them, which is the information defined about all instances of that kind of pattern element.

To access the schema information of any element, you use the .Info property of one of the generic interfaces, which requires first moving to the generic scheme.

```
            IElementInfo schemaInfo = this.CurrentElement.AsElement().Info;
```

> **Note**: From the .Info property you can see all the configured descriptive information about the element.

Moving Between Generic and Typed Pattern Model Schemes

Once you have a reference to an element in a pattern model, usually after having imported into your automation class, you can traverse between the generic and typed schemes using specific methods provided in those schemes.

This is typically required to use many of the provided API's and helper methods of the automation framework.

> **WARNING**: You cannot simply type-cast from one scheme to the other. The result will always be null.

Generic to Typed

Element, Collections, View and Patterns

Use the .As<T>() method, to get the typed interface.

```
Given:
            public IProductElement CurrentElement { get; set; }
Then:
            IMyPatternElement1 element = this.CurrentElement.As<IMyPatternElement1>();
```

> **Note**: The call to .As<T>() will return null if the current element is not of the specified type.

Typed to Generic

Elements and Collections

Use the .AsElement() method, to get the generic IElement interface.

Use the .AsCollection() method, to get the generic ICollection interface.

```
Given:
            public IMyPatternElement1 CurrentElement { get; set; }
Then:
            IElement element = this.CurrentElement.AsElement();
```

Views

Use the .AsView() method to get the generic IView interface.

```
Given:
            public IMyView1 CurrentView { get; set; }
Then:
            IView view = this.CurrentView.AsView();
```

Pattern Element

Use the .AsProduct() method to get the generic IProduct interface.

```
Given:
            public IMyPattern1 CurrentPattern { get; set; }
Then:
            IProduct pattern = this.CurrentPattern.AsProduct();
```

Target Path Syntax

Several of the automation commands, conditions and value providers that work upon items in Solution Explorer use a specific syntax for referencing files, folders, projects and solution folders in the solution. This syntax is complex because it mixes both the hierarchical structure of the pattern and the physical solution structure in Solution Explorer.

For example, the 'UnfoldVsTemplateCommand' can be configured to unfold a project or file into a specific project, or folder in the current solution.

For many pattern toolkits, the solution structure is either well known by the pattern, as it is the pattern that would have determined that project structure via automation, or certain artifacts in the solution are referenced by elements in the pattern model.

The syntax of these paths always starts with establishing the current element in the pattern model, then traverses a known artifact link (on that element) moving from model path space to physical solution path space, and then from there through solution containers such as projects, folders etc.

The general syntax of a path is in 3 optional parts: **[Model Path Space]~[Solution Path Space]**, where the '~' character denotes the traversal from model to solution path space.

The starting point for these paths is the current element in the model. The first part of the path (model space) allows you to define a relative path in the model from the current element. For example: a path starting with '..\..\..\' moves the scope from the current element to the grandparent element of the current element.

Next in the path is the tilda '~' character. This character resolves the first artifact link of the element in scope to a physical solution item (such as a: file, folder, project or solution folder or even the solution itself).

Without an artifact link on some element in the model, no traversal from model to solution is possible, and the '~' character can be omitted. In this case the solution is the assumed starting point. You do this if you want to specify only a solution path.

The last part of the path is relative from the resolved solution item, (or from the solution when '~' is omitted). This solution path is the logical path in solution explorer and can traverse solution folders, projects, project folders and even files to nested files.

Example Syntaxes:

Path	Meaning	Notes
\	The solution	Paths starting with '\' character imply the solution root and solution folders or project within.
~	Traverse the artifact link of the current element.	This traverses the first artifact link found, starting with the current element and then ancestor elements (if none found on current element). By default, if ~ is specified as first character, an artifact link is searched for by traversing up the ancestry from the current element (starting with the current element) until an element is found that has an artifact link, and that link is then resolved. Be aware that if specifying ~ as the first character in the path on an element that has an artifact link, will resolve to that artifact link and not to any link on any ancestor. It is recommended that ~ is always explicitly fully qualified with a relative ancestral path (i.e. ..\..\) to the ancestral element to avoid errors in resolving the artifact links at runtime. If no ancestor is found with an artifact link, then the ~ is resolved to being the solution itself.
..\	Traverse to parent element in either the model or solution.	Before a ~ is resolved in the path, this path traverses up the model path to a parent element. After a ~ is resolved in the path, this path traverses up the solution path from current solution item. Note: that if the intended ancestor element is the root pattern element, then do not include a ..\ for the View (which is actual the immediate child of the root pattern element).
..\~\Dummy	Move to the parent element in model, resolve its artifact link over to the solution, and then navigate down into its 'Dummy' container in the solution (solution folder, project or project folder).	Typically used to define a path into a solution container that is already referenced by a specific ancestor element. For example: if the parent element had an existing artifact link to a project or folder, then this path points to a subfolder of that project or folder. This path explicitly identifies that the artifact link to traverse is on the parent element, and therefore the search for an ancestor's artifact link is not executed. This is useful when many of the ancestors have artifact links to different solution elements.
~\Dummy	Search the ancestry of the current element (including the current element first) for an artifact link, then resolve the artifact link over to the solution , and then navigate down into its 'Dummy' container in the solution (solution folder, project or project folder).	Typically used as shorthand to avoid explicitly identifying which ancestor element has the artifact link. Can be useful when composing patterns together, and relying on parent patterns having artifact links somewhere in their ancestry. Note: Use caution with specifying the ~ as first character. See notes above.
..\~\Dummy\{Parent.InstanceName}.cs	Search the ancestry of the current element (including the current element first) for an artifact link, then resolve the artifact link over to the solution , and then navigate down into its 'Dummy' container in the solution (solution folder, project or project folder), then navigate to a C# source file with the name of the instance of Parent element. (presumably already in the solution)	Typically used to define a path to file in the solution with a dynamic file name based on properties of elements in the model ancestry.

Expression Syntax

Many values of many properties in the pattern model, and used by various automation classes, support an expression syntax for substituting values from the pattern model for easier configuration by an author.

The expression syntax uses a simple substitution format that uses curly braces '{}' to surround the expression within an expression. These are then substituted when the value is evaluated. (i.e. "The parent elements' name is: '{Parent.InstanceName}'", or "This value of the 'Something' of this element is: '{Something}'").

The substitutions which are possible depend entirely on the specific pattern model being used, and the expression is rooted from the current element being configured.

Example Syntaxes

Expression	Meaning	Notes
{InstanceName}	The value of the given name of the current element instance.	'InstanceName' is a special built in property which is the value of the name of the instance of the current element. This value is typically defined either automatically by automation, or more usually manually by the pattern user.
{AProperty}	The value of the property called 'AProperty' on the current element.	Where 'AProperty' is a variable property on the current element.
{Parent.InstanceName}	The value of the name of the parent element of the current element.	
{Parent.Parent.Parent.Parent.AProperty}	The value of the property called 'AProperty' on an ancestor element of the current element.	You can traverse up into any ancestor of the current element that is reachable, including through extension points, for pattern instances that are 'parented' on other pattern instances.
{Parent.OtherElement.AProperty}	The value of the property called 'AProperty' belonging to a sibling element instance of the current element.	You can traverse down any branch of the current pattern model, through OneToOne or ZeroToOne relationships.
{Parent.OtherCollection[3].AProperty}	The value of the property called 'AProperty' belonging to the third instance of a sibling element of the current element.	You can traverse down any branch of the current pattern model, through 'OneToMany' or 'ZeroToMany' relationships, by instance index.

Provided Automation Types

The following automation types are provided to be shared across all pattern toolkits, and can be used to apply automation to pattern models.

- Commands
- Conditions
- Value Providers
- Validation Rules
- Events

> **Note**: These automation classes are provided to all pattern toolkits in a shared automation library that ships with the 'NuPattern Toolkit Builder' extension.
> There may be additional automation classes seen in various lists throughout the design time experience, and these may come from additional installed toolkits.

Commands

Display Name/Category	Description
Guidance	
Activates an Associated Feature Extension.	Activates an existing feature instance making it the currently selected guidance workflow in the 'Guidance Explorer' window.
Instantiates an Associated Feature Extension.	Creates a new feature instance, with an optional default name, and optionally makes the feature the currently selected feature in the 'Guidance Explorer' window.
Pattern Automation	
Activates (Selects or Opens) the Element's Associated Artifacts	'Opens' or 'Selects' all the associated artifacts for the current element.
'Opens' or 'Selects' a single associated artifact for the referenced element.	'Opens' or 'Selects' a single associated artifact for the referenced target element.
Aggregates Other Commands	Executes one or more other commands in an ordered sequence.
Creates Artifact Links to Solution Items	Creates solution item artifact links for each of the given Items.
Creates New Child Element From Dropped Solution Item	Creates new instances of the specified child element for each dropped solution item of the specified file extension, and sets an artifact link to the solution item.
Creates New Child Element From Dropped File from Windows Explorer	Creates new instances of the specified child element for each dropped windows explorer file of the specified file extension, adds the file to the solution at the given 'Target Path', and sets an artifact link to the solution item.
Creates New Child Element From Selected Files on the Computer	Creates new instances of the specified child element for each selected file of the specified file extension, adds the file to the solution at the given 'Target Path', and sets an artifact link to the solution item.
Generates Code using T4 for an Element	Generates code for the current pattern element by transforming a text template (also known as a T4 template).
Instantiates a New Solution Element	Creates and Instantiates a new element in the Solution Builder window.
Saves the Associated Artifacts	Saves the associated artifacts to this element.
Sets Project Items Build Action	Sets the build action of an existing project item.
Synchronizes all Associated Artifact Names	Synchronizes the name of the associated unfolded artifact (i.e. file, project, folder) to the Name of the current element. For project artifacts, this command also updates the 'AssemblyName' and the 'RootNamespace' properties of the project.
Transforms All Text Templates	Transforms all text templates found on, and contained within, the target solution item.
Unfolds a VS Project/Item Template for an Element	Unfold a VS project or item template (*.vstemplate) for the current element.
Validates Element	Executes all validation rules for the current element, and optionally for all its descendants.
Visual Studio	
Collapse All Solution Items	Collapses all items in the solution, except projects.
Runs a Visual Studio Command	Executes any configured command in the Visual Studio IDE.
DSL Automation	
Generates Code using T4 for a DSL Model Element	Generates code by transforming a text template (also known as a T4 template), for any model element of any DSL model file.

Conditions

Display Name/Category	Description
General	
String Values are Equal	Used to verify that the configured left and right string values are equal, using the configured comparison kind.
Pattern Automation	
Associated Artifacts are Saved	Used to verify that all associated artifacts to the current element are saved.
Dropped Item has DataFormat	Used to verify that the current dragged data is of the specific System.Windows.DataFormats format. (i.e. FileDrop, Text, Xaml, Html, Bitmap, etc.)
Dropped Items are Files	Used to verify that the current dragged data includes one or more files of the specified file extensions.
Dropped Items are Solution Items	Used to verify that the current dragged data includes one or more solution items of the specified file extensions.
Element is Validated	Used to verify that all validation rules for the current element, and optionally for all its descendants, are satisfied.
Element Reference Kind Exists	Used to verify the existence of a reference of the given kind on the current element.
Element Solution Artifacts Exists	Used to verify the existence of any artifact references on the current element. Does not verify associated artifacts that are projects, folders or the solution.
Event Sender Is Current Element	Used to verify that the raised event is specific to the current element instance only.
Pattern Is Parented	Used to verify if the pattern (parent of the current element) is parented within another pattern.
Property Changed Match	Used to verify if the property that changed (i.e. the one that raised the OnPropertyChanged event of the current element) is the given property.
Property Exists	Used to verify the existence of a variable property on the current element, and whether that property must have a value.

Value Providers

Display Name/Category	Description
General	
A New GUID (Formatted)	Retrieves a new (randomly created) GUID of the specified format.
Current Date and Time	Retrieves the current date and time from the system.
Registered Organization (Current Machine)	Retrieves the registered organization for the current windows installation on this machine.
Pattern Automation	
Current Store File name	Retrieves the currently open product store file name from the product manager. Returns null if a store is not open.
Drag/Dropped Items (Any Source)	Retrieves the items of the specified Data Format being (dragged and) dropped from any source
Drag/Dropped Items from Solution Explorer	Retrieves the items being (dragged and) dropped from Solution Explorer
Expression	Retrieves the value of the evaluated expression that accesses properties of the current element, or its ancestry. (i.e. {PropertyName}, or {Parent.InstanceName} or {Parent.Parent.PropertyName} etc.)
Path of First Related Solution Item	Retrieves the full physical path to the first referenced solution item of the current element with the given extension.
Project Assembly Name	Retrieves the assembly name of a project in the current solution.
Project GUID	Retrieves the GUID identifier of a project in the current solution.
Project Root Namespace	Retrieves the root namespace of a project in the current solution.
Solution Name	Retrieves the name of the current solution.
Variable Property Value	Retrieves the value of a variable property of the current element in the pattern model.
Replace Forbidden Characters in Expression	Replaces forbidden characters in the value of the evaluated expression that accesses properties of the current element, or its ancestry. (i.e. {PropertyName}, or {Parent.InstanceName} or {Parent.Parent.PropertyName} etc.)

Validation Rules

Display Name/Category	Description
General	
Associated Artifacts are Saved	Validates that the associated artifacts to the current element are currently saved. This rule does not validate associated artifacts that are projects, folders or the solution.
Child Element Cardinality	Validates that the number of child element instances is within configured cardinalities range.
Property Value Forbidden Characters	Validates that the value of the property does not contain any forbidden characters.
Property Value Length	Validates that the length of the property is between (or equal to) the minimum and maximum lengths.
Property Value Matches Regular Expression	Validates that the property value satisfies the given regular expression.
Property Value Range	Validates that the property value is within the given maximum and minimum range.
Property Value Refers to a Solution Item	Validates that the property value refers to an existing item in the solution. Either a relative solution path (i.e. \ProjectName\FolderName\File.txt) or an absolute path (i.e. C:\Folder\File.txt).
Property Value Required	Validates that the property has a value.

Events

Display Name/Category	Description
Visual Studio	
A Build in VS is Finished	Raised when a build in Visual Studio has finished.
A Build in VS is Started	Raised when a build in Visual Studio starts.
Pattern Automation	
Any/All Elements are Saved	Raised when any and all elements in Solution Builder are saved (occurs automatically and frequently).
Element is Activated	Raised when a product/element/collection is activated (i.e. double-clicked).
Element is Deleted	Raised after a product/element/collection has been deleted.
Element is Deleting	Raised before a product/element/collection is to be deleted.
Element is Initialized	Raised when any product/element/collection is initialized by any mechanism (i.e. first created, deserialization, programmatically, via automation etc).
Element is Instantiated	Raised when any product/element/collection is created for the first time. Raised when user manually creates an element in Solution Builder, or automation creates element programmatically with instantiation flag. Not raised when programmatically creating an element without instantiation flag.
Element is Loaded (from Rehydration)	Raised when any product/element/collection is rehydrated from serialization. This typically occurs whenever the solution is opened, and Solution Builder is populated.
Property of an Element has Changed	Raised when a property of an element is changed.
Drag and Drop	
Dragged Data Enters	Raised when dragged data has just entered the solution builder window.
Dragged Data Leaves	Raised when dragged data has moved outside of the solution builder window.
Dragged Data Dropped	Raised when any dragged data source is dropped onto an element in solution builder window.

Environment

The development and tooling environment for using, authoring and customizing Pattern Toolkits.

Visual Studio Experimental Instance

The 'Experimental Instance of Visual Studio', is a special testing version of Visual Studio that is primarily used to test and debug Visual Studio extensions under development. As opposed to the 'Normal' instance of Visual Studio where regular code development takes place.

> **Note**: All the settings needed for Visual Studio are kept in the registry, and the experimental instance using a different set of registry settings than the normal instance of Visual Studio.

This special experimental instance of Visual Studio can be reset at any time, to clear out any detritus from testing or developing extensions against it, without affecting the 'Normal' instance of Visual Studio where you do your regular development.

Running Experimental Visual Studio

To run to the 'Experimental Instance', run the **"Start Experimental Instance of Microsoft Visual Studio 201X"**, from the Start Menu (All Programs | Microsoft Visual Studio 201X SDK | Tools).

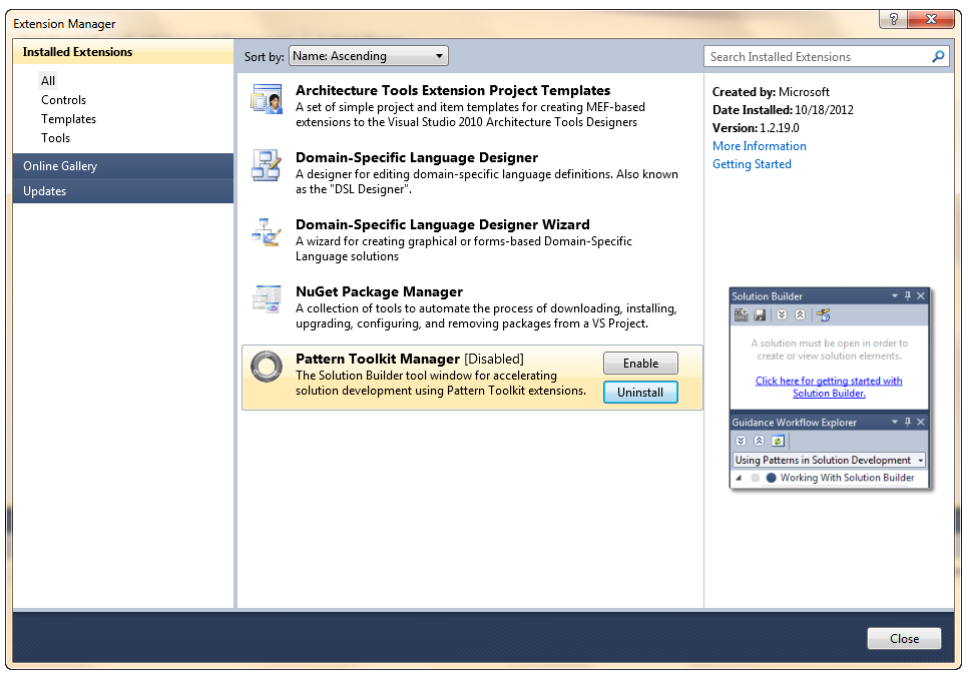
Resetting Experimental Instance

To reset the Experimental Instance of Visual Studio, (specifically for pattern toolkit development) requires these steps:

1. Close all running instances of Visual Studio 201X.
2. Run the **"Reset the Microsoft Visual Studio 201X Experimental instance"**, from the Start Menu (All Programs | Microsoft Visual Studio 201X SDK | Tools).

> **WARNING**: In some cases this command prompt fails to run correctly, and does not reset the experiment hive.
> The symptom of this is if the command window flashes (immediately opens and closes), and does not ask you to "Press any key to continue…" after it finishes the process.
> If this occurs, simply delete the directory at: **%localappdata%\Microsoft\VisualStudio\1X.0Exp** and run the command again.

3. Run the **"Start Experimental Instance of Microsoft Visual Studio 201X",** from the Start Menu (All Programs | Microsoft Visual Studio 201X SDK | Tools).
4. Open the 'Extension Manager' dialog (Tools | Extension Manager…), and enable all the extensions which have the **'[Disabled]'** status.

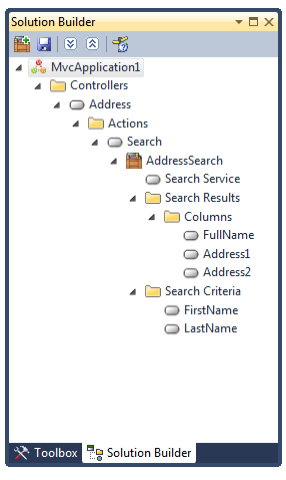


> **Note**: If you see any of your toolkits here that are under development at this point, you should probably uninstall them also.

5. Close the 'Experimental Instance' of Visual Studio.
6. Rebuild your toolkit solutions

Solution Builder

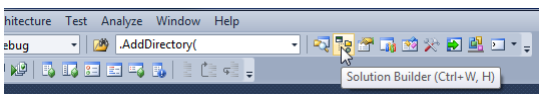
The 'Solution Builder' tool window is a new tool window for working with patterns in your solution.



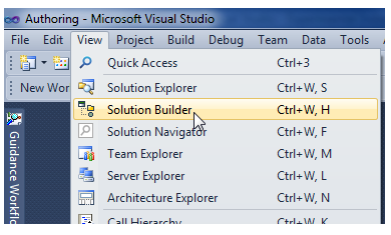
You use this window for creating 'New Solution Elements' that help automate the creation of projects in your solution using installed patterns.

To show the tool window, either:

- Click on the 'Solution Builder' button on the main toolbar in Visual Studio.



- Click on the 'Solution Builder' menu (View menu)



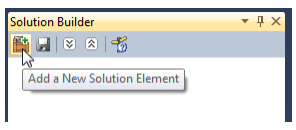
- Press **CTRL + W, H**

> **Tip**: Move and dock the tool window in a separate area of the main window of Visual Studio (i.e. on the opposite side of the window from 'Solution Explorer'), so that you can see both 'Solution Builder' and 'Solution Explorer' at the same time. This is useful for working on the solution using both these windows.

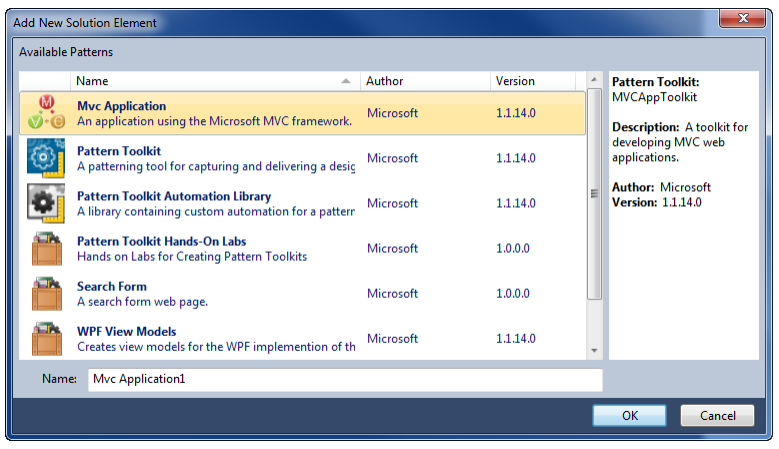
Add New Solution Element Dialog

The 'Add new Solution Element' dialog is where you create new instances of patterns in your solution, these new instances called 'Solution Elements' are then created in the 'Solution Builder' window.

You can open this dialog box from 'Add' button  the 'Solution Builder' window.

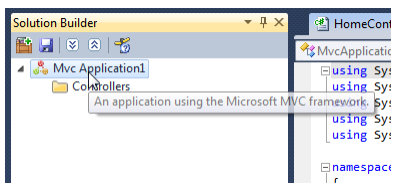


> **Note**: You will need to have an existing or new solution already open to create new solution elements with this dialog.



From this dialog, you select the pattern you want to use, and name it.

A new instance of that pattern, with that name is then created in the 'Solution Builder' window.



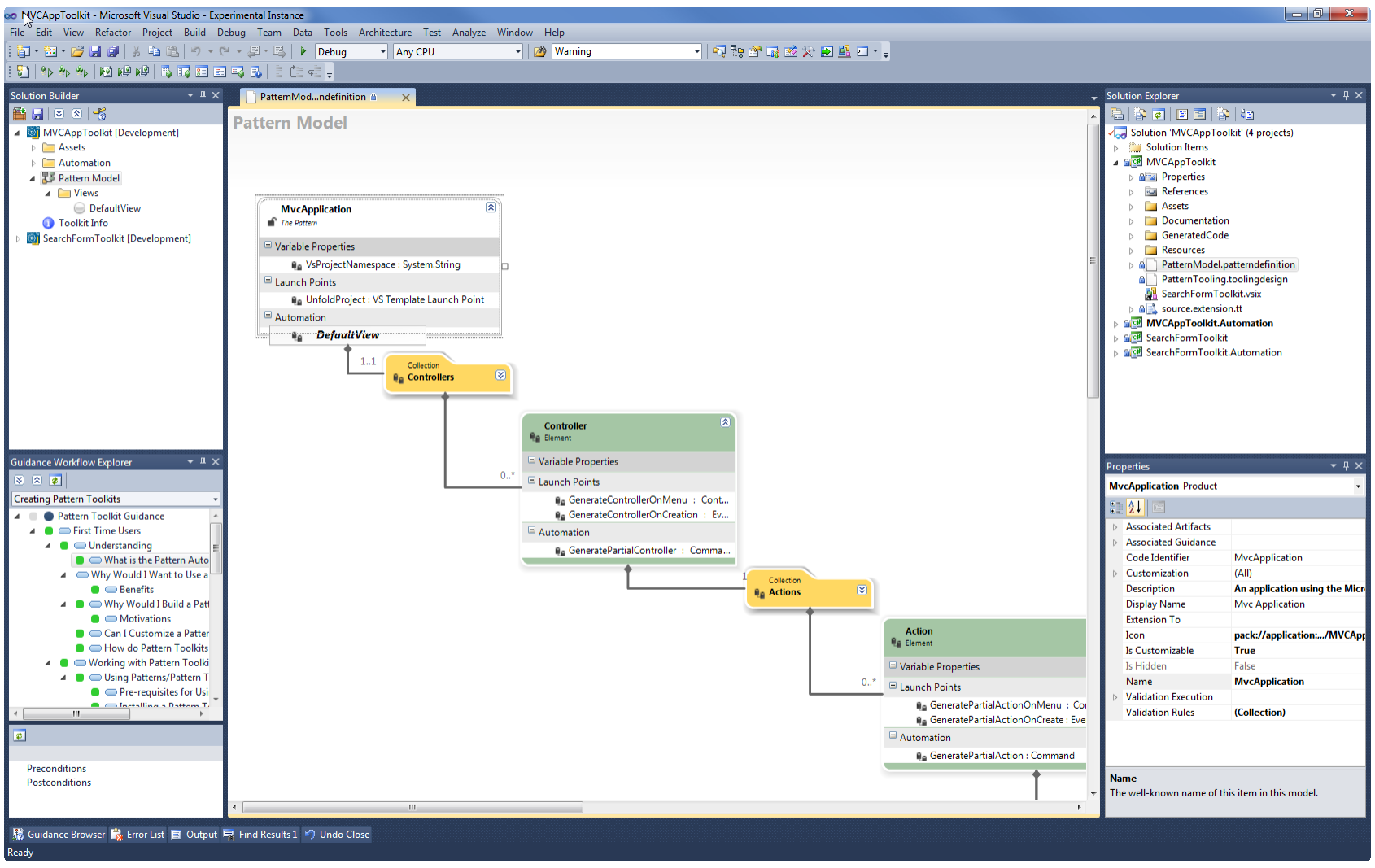
Pattern Model Designer

The 'Pattern Model Designer' is a designer for describing the pattern within a Pattern Toolkit.

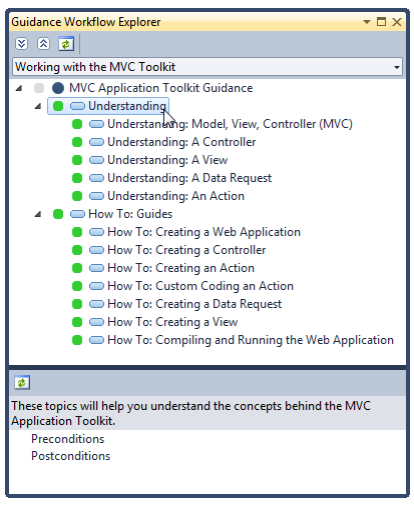
This designer defines the pattern, and the elements which allow a user to configure its variability for their solution.

You open the designer from the 'Pattern Model' element in 'Solution Builder', or the PatternModel.patterndefinition file in 'Solution Explorer'.

You modify the shapes on the designer with the 'Properties Window'.



Guidance Workflow Explorer

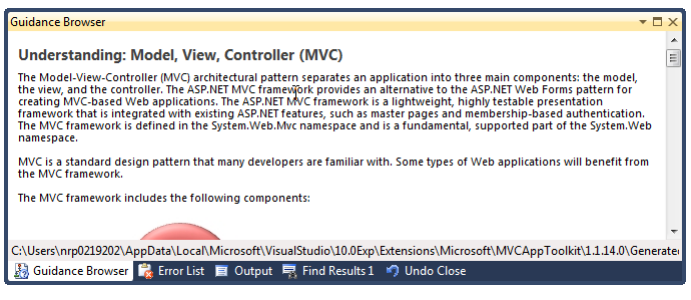


The 'Guidance Workflow Explorer' window (View | Other Windows menu) in Visual Studio displays the available guidance workflows that can be used to work with other tools and processes in Visual Studio.

This window allows you to select a workflow from the drop down list, click on each step in the workflow.

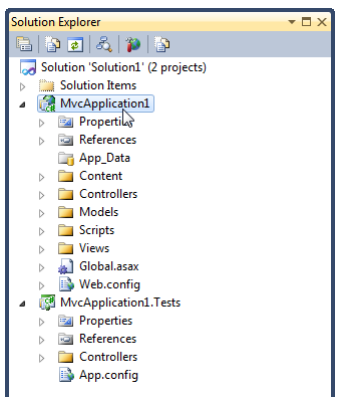
Each step in the guidance workflow, may be either active (green), blocked (red), or disabled (grey) to indicate whether the user can proceed with that step in the guidance. Some steps will include a checkbox that allows the user to mark the step as complete. Other steps may determine if they are complete using automation and conditions.

For steps that have more information, you can select the step in the workflow and browse the guidance in the 'Guidance Browser' window.



More details on 'Guidance Workflows' and how they work will be provided soon.

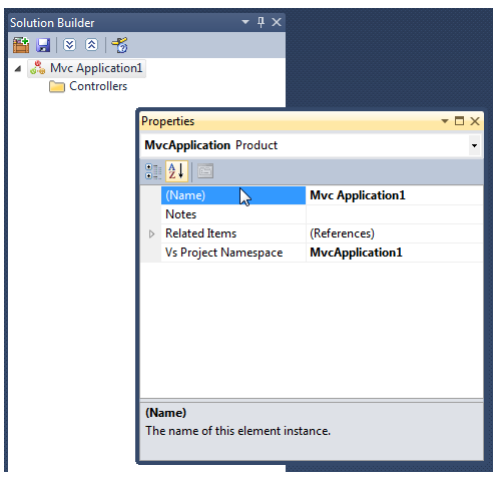
Solution Explorer



'Solution Explorer' (CTRL + W, S) is a common tool window in Visual Studio for displaying the solution and projects under development.

This window shows the physical representation of a solution, with its files, folders and projects.

Properties Window



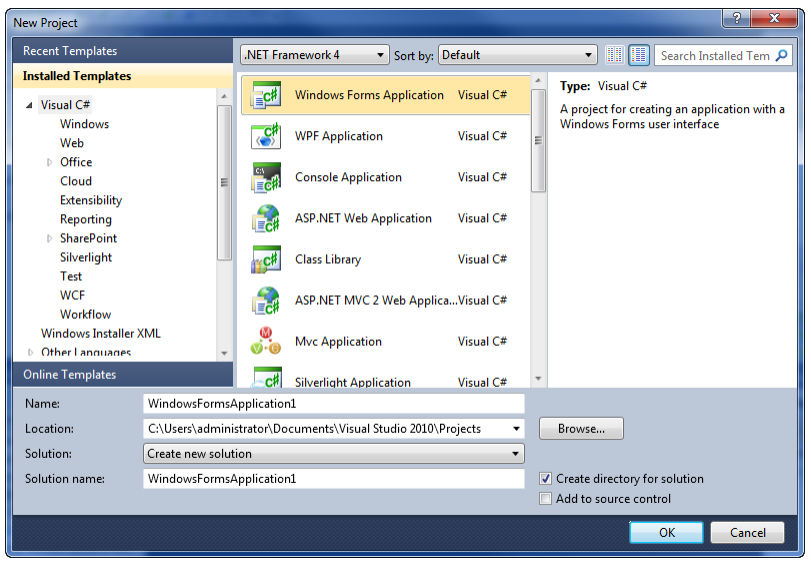
The 'Properties Window' (CTRL + W, P) in a common window in Visual Studio for displaying the properties for the selected object in the currently active window.

In this window you can edit the properties of the selected object, and it provides editors and dialog for manipulating these properties.

Add/New Project/Item Dialog

The Visual Studio 'Add New/Add Existing Project or Item' dialogs (File | New or File | Add menus) are where you add new projects and items to projects in your solution.

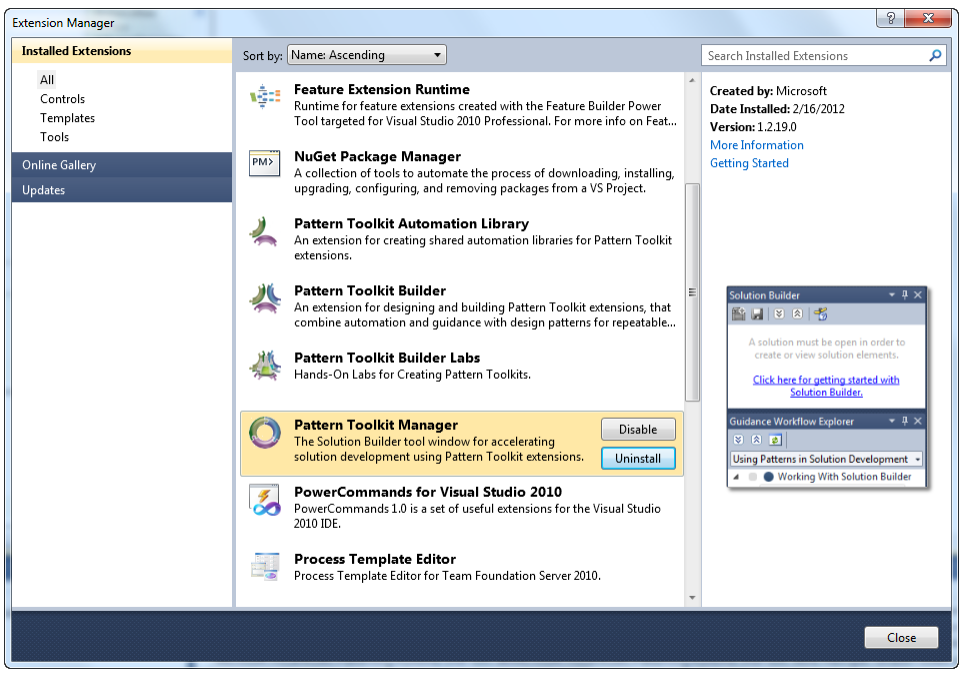
This dialog lists all the project and item templates installed on the current machine.



> **Note**: Item templates are filtered based upon the projects the items will be added to.

Extension Manager

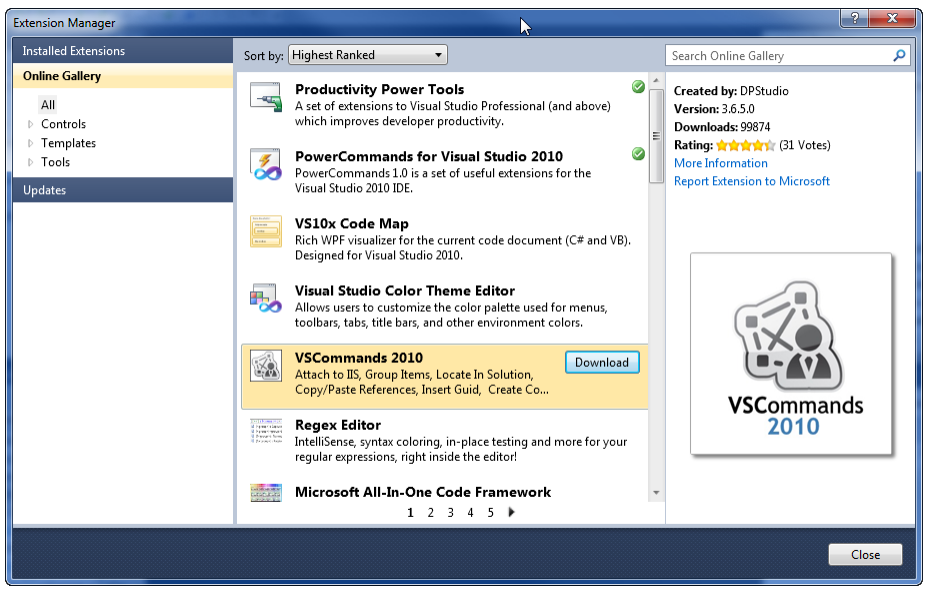
The Visual Studio 'Extension Manager' (Tools | Extension Manager menu) is where you view and manage the installed extensions to Visual Studio.



In the dialog above you can see the currently installed extensions.

> **Note**: NuPattern extensions are listed as extensions in 'Extension Manager'.

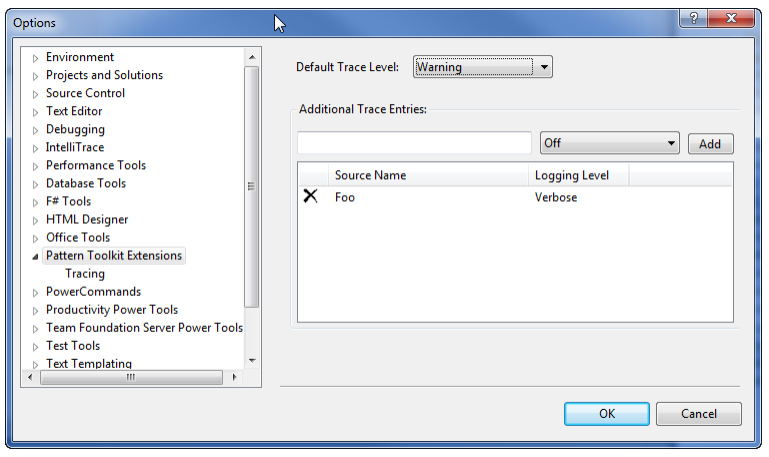
You can also download and install additional extensions from the [Visual Studio Code Gallery] in this dialog.



Options

There are several options that control how patterns toolkits work in the Visual Studio environment.

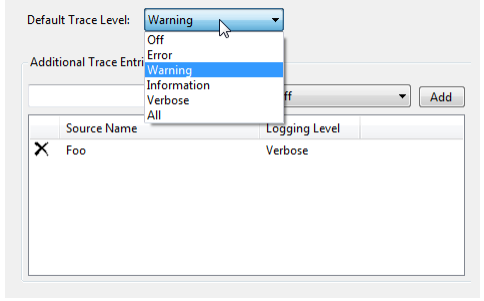
They are modified in the Visual Studio 'Options' dialog, (from the Tools | Options menu)



Tracing Options

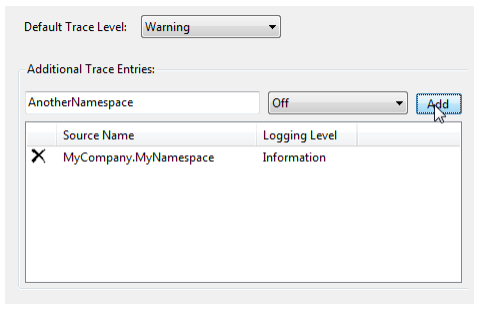
In this page you can control the tracing levels and tracing sources that appear in the Tracing Window.

To change the default trace level for all toolkit diagnostic information, select the 'Default Trace Level'.



> **Note**: The different levels are ordered such that the items lower in the list include all items above them. So for example, by choosing 'Information' you are choosing 'Information', 'Warning' and 'Error' levels.

To add additional trace sources for deeper diagnosis, add a new trace entry to the 'Additional Trace Entries' list.



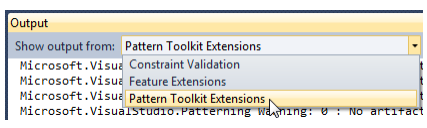
> **Note**: The format of trace sources is typically a name of an application or namespace of the types that produce trace output.
> See Introduction to Instrumentation and Tracing in the .NET Framework.

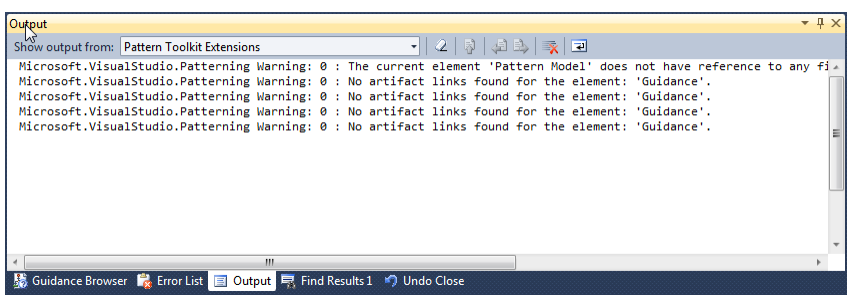
Tracing Window

The Tracing Window is where status and troubleshooting information is displayed for all pattern toolkits.

Opening the Trace Window

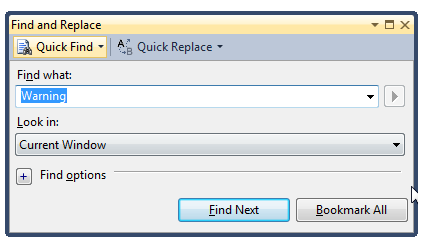
Open the 'Output Window' (CTRL + W, O) in Visual Studio, and select the 'NuPattern Toolkit Extensions' pane from the drop down list at the top.





Working with the Trace Information

The trace information can be searched using 'Find' (CTRL + F), for searching for specific text.



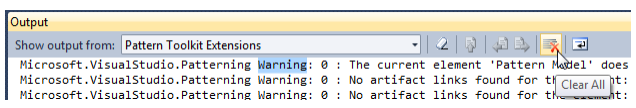
Select the 'Current Window' option to search this information.

> **Note**: Make sure the 'Output Window' is first pinned, and the mouse is first clicked in the window before searching.

Trace information can also be copied to the clipboard.

> **Tip**: This is useful for sending the trace information to others, for example in an error report.

You can also clear the information by clicking the 'Clear All' button in the tool bar of the 'Output Window'.



Adding More Trace Information

You can change the level of diagnostic information displayed in this trace window, or add trace information from other sources by changing the options in the 'NuPattern Toolkit Extensions' Options.

For example, you can increase the level of diagnostic trace information for all toolkits, and add additional trace sources for other extensions running in Visual Studio.

More Information

- Pattern Toolkits are compiled into Visual Studio extensions which are packaged and deployed as VSIX files.
  - What is a VSIX?
  - Visual Studio Extension Deployment
  - VSIX Deployment

You can find more information about NuPattern on the codeplex project site at http://nupattern.codeplex.com

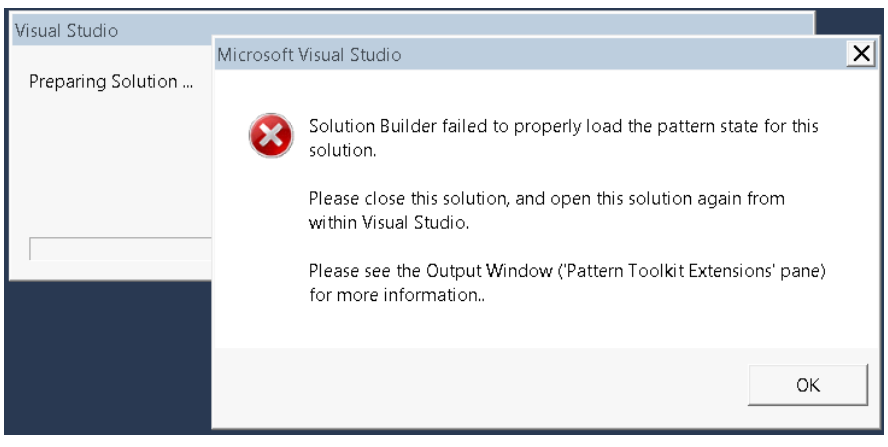
Known Issues

This is a list of the critical known issues in the current version of NuPattern.

Error when opening a pattern toolkit solution file

Symptoms:

When you open a solution that contains a pattern state file (*.sln file) from a shortcut link (i.e. from outside Visual Studio), as the solution opens you may see the following error:



And the 'Solution Builder' window is empty.

Workaround:

From within Visual Studio, close the solution (File | 'Close Solution' menu), and then re-open the solution from the list of recent projects (File | 'Recent Projects And Solutions' menu).

To avoid this problem entirely next time, open Visual Studio to either the 'Start Page' or 'Empty Environment', and open the solution from hard disk (File | Open menu) or from the list of recent projects (File | 'Recent Projects And Solutions' menu).

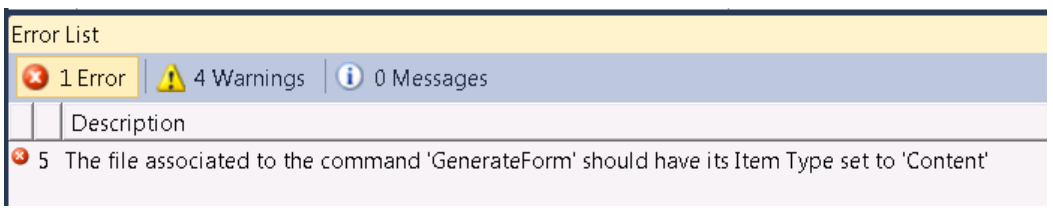
Corruption of pattern model when performing an Undo

Symptoms:
In the pattern model designer, when you Undo after adding a new Automation or Launch Point to any element, the element is visually removed but a placeholder remains, which corrupts the pattern model file.

Workaround:
Instead of doing an ‘Undo’, simply ‘Delete’ the newly added Automation or Launch Point manually.

Error in Text Templates when using the Visual T4 Editor

Symptoms:
If you install and use the Visual T4 Editor for editing text templates, and have associated a text template to a command on the pattern model, and are editing that text template in Visual Studio at the same time you build or try to debug your toolkit, you receive the error below.



Where 'GenerateForm' is the name of your command that you have configured using the 'Generate T4' command.

Cause:
When you open a text template for editing using the Visual T4 editor, Visual T4 temporarily changes the 'build action' of the *.tt or *.t4 file in solution explorer from 'Content' to 'Compile' while the file is open for editing. This is necessary for intelli-sense to work as you work with code within the file. When the file is closed, Visual T4 restores the build action to 'Content', and 'IncludeInVSIX' property back to true, as they were before opening the file. The problem is that the 'Build Action' property needs to be 'Content' and 'IncludeInVSIX' property needs to be 'True' for the template to work in your toolkit.

The problem above, only occurs if the file is open when you build the solution, as the build action is temporarily set to 'Compile' instead of 'Content' as it needs to be to work in the deployed pattern.

Solution:
Close all *.tt and *.t4 files before building and debugging, and ensure the 'Build Action' property is set to 'Content', and the 'IncludeInVSIX' property is set to 'true'.

If the problem persists after closing the files, try closing all open documents (including hidden open document) using the Window | Close All Windows command.
The properties of the files should be restored to:

- Build Action = Content
- IncludeInVSIX = true.

If problem persists, unfortunately the only reliable permanent work around is to [disable] the 'Visual T4' extension in 'Extension Manager'.

Build error: "store must be open for this operation"

Symptoms:
In Visual Studio 2010, when you build a pattern toolkit project you get a build error "store must be open for this operation", and the output window indicates a problem with a MS Build target in the Microsoft.VsSDK.targets file.

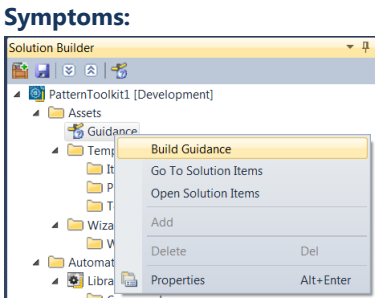
Cause:
The version of the Visual Studio 2010 SDK is not compatible with the version of Visual Studio 2010.

It is likely you have Service Pack 1 for Visual Studio 2010 installed, but do not have Service Pack 1 of the Visual Studio 2010 SDK installed.

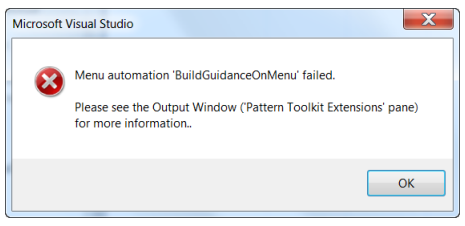
Solution:
You must uninstall the current version of the VSSDK installed on your machine, then install Service Pack 1 of the Visual Studio 2010 SDK.

Error, building guidance with 64bit versions of Microsoft Word.

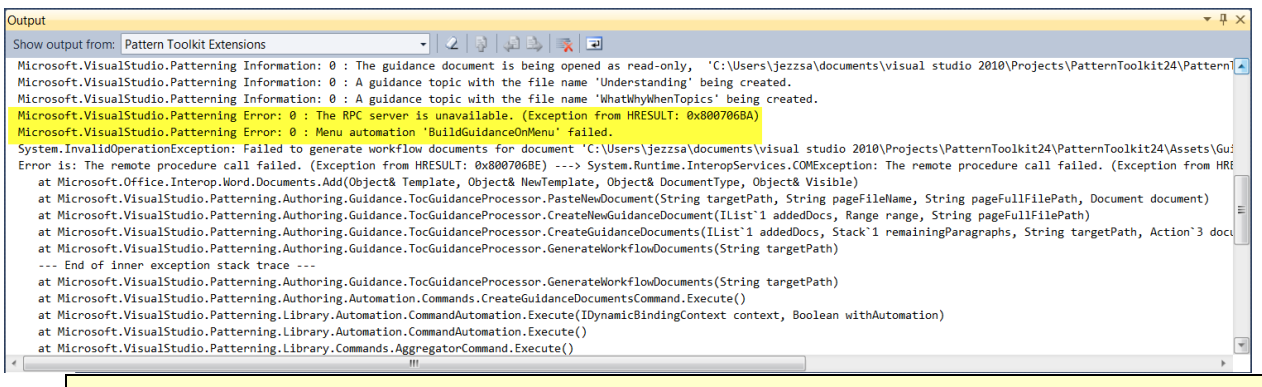
Symptoms:



When you right click on the 'Guidance' node in Solution Builder and select 'Build Guidance', Microsoft Word crashes and you receive an error reporting that the build guidance action failed.



The guidance fails to build, and the following error can be found in the 'Output Window'.



> **Note**: In some installations of Microsoft Office, Microsoft Word may no longer start correctly, and the machine may require rebooting.

Cause:

Automating Office applications such as Microsoft Word from Visual Studio (as is the case for building guidance in pattern toolkits) requires that the correct user identity be configured for Microsoft Word in the Windows DCOM configuration.

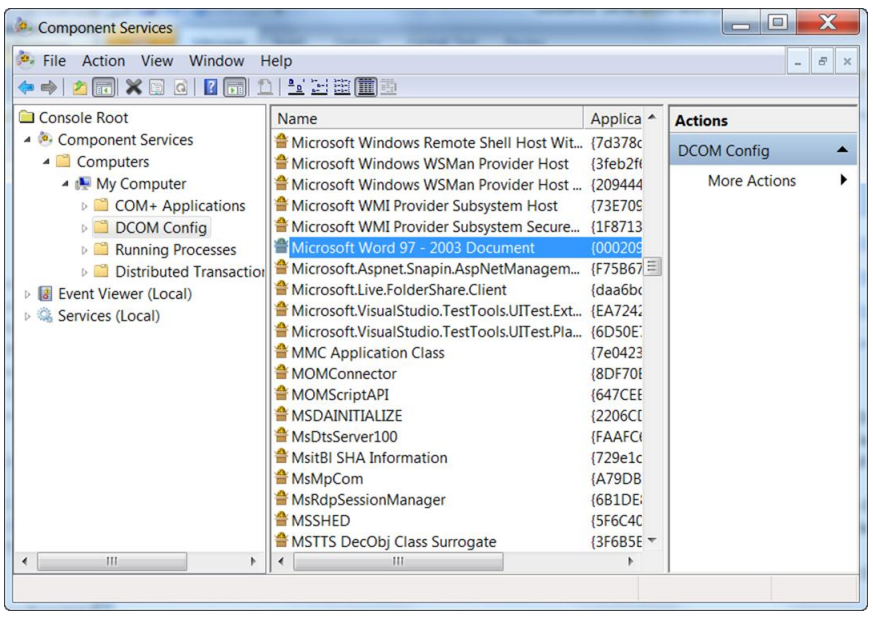
> **Note**: The 64 bit version of Office has been confirmed to experience this issue, whereas the 32 bit version of Office may not.

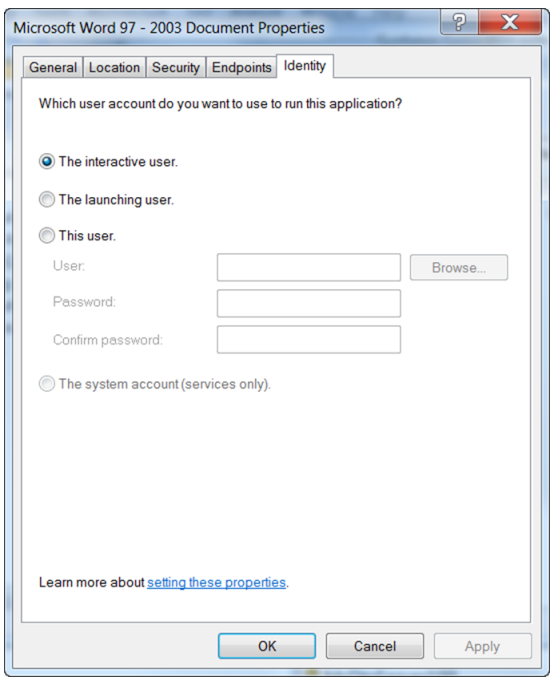
Solution:

In Windows, at the Start menu, type: "Component Services". The "Component Services" MMC console window will open.

Expand the "Component Services | Computers | My Computer | DCOM Config" node.

Select the "Microsoft Word 97 -2003 Document" entry.





Right-click on this entry, and select the "Properties" menu.

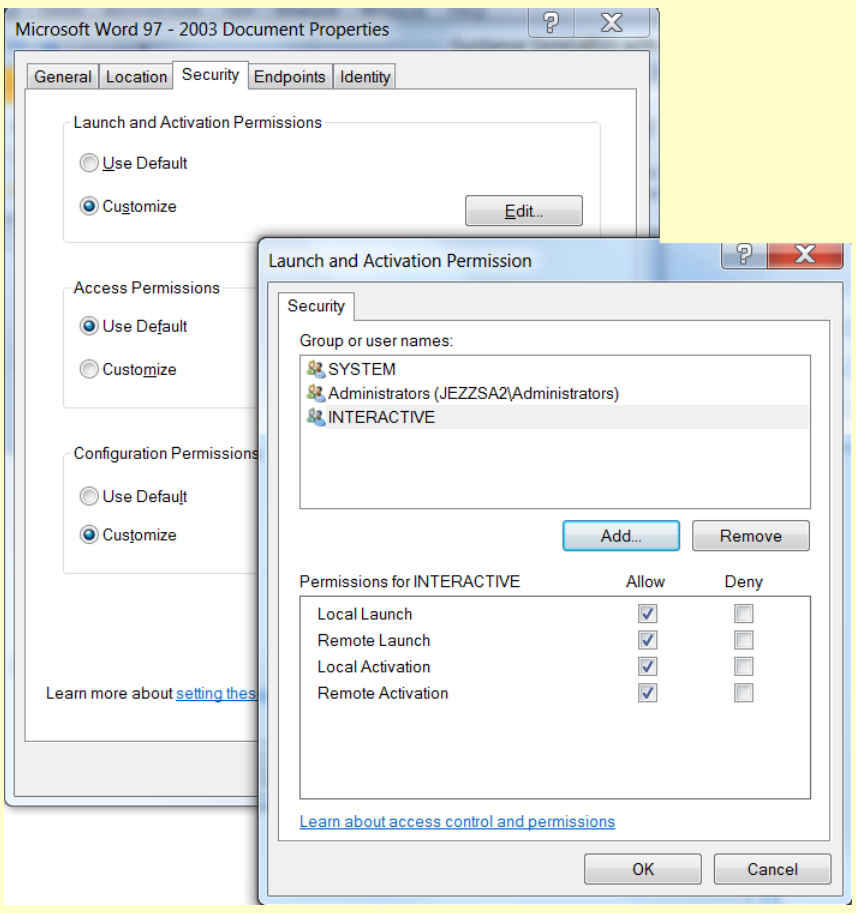
In the properties dialog, select the "Identity" tab.

Change the identity account from "The launching user" to be the "The interactive user".

> **Note**: Some installations of Microsoft Office and Visual Studio may require you to specify explicitly the identity account of your current user and password. In these cases, select the third option in this page, and type your full account with domain, and password for you're the current user.

Press OK, to close the dialog.

> **Note**: This solution presumes that the account that you use to work in Visual Studio is in the "Local Administrators" group for the machine.
> If it is not then this solution requires additional configuration.
> Select the "Security" tab.
> In the "Launch and Activation Permissions" setting, click on "Customize", and then on the "Edit" button.
> Click "Add", and add the 'INTERACTIVE' account.
> Ensure that the "INTERACTIVE" account allows both 'Local Launch' and 'Local Activation' permissions.



Press "OK", to close all dialogs.

Feedback

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