Solution Builder 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.

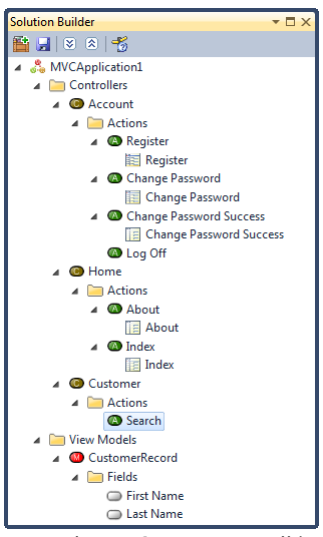
What is NuPattern?

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.

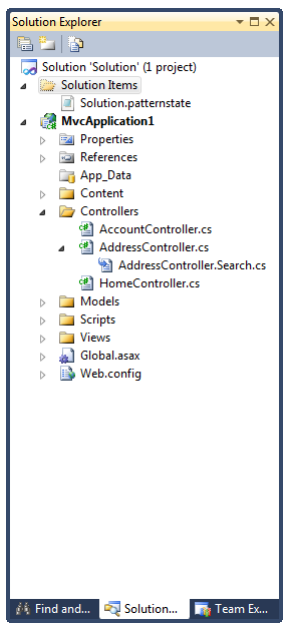


Example MVC Pattern Toolkit

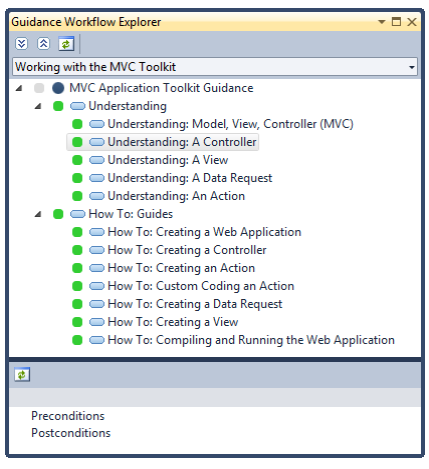
When a ‘Pattern Toolkit’ is installed, a ‘user’ of the toolkit works in a new tool window called 'Solution Builder', which allows them to create, navigate and configure parts of their solution in terms of the components of the working solution instead of in terms of the low-level technological artifacts 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.

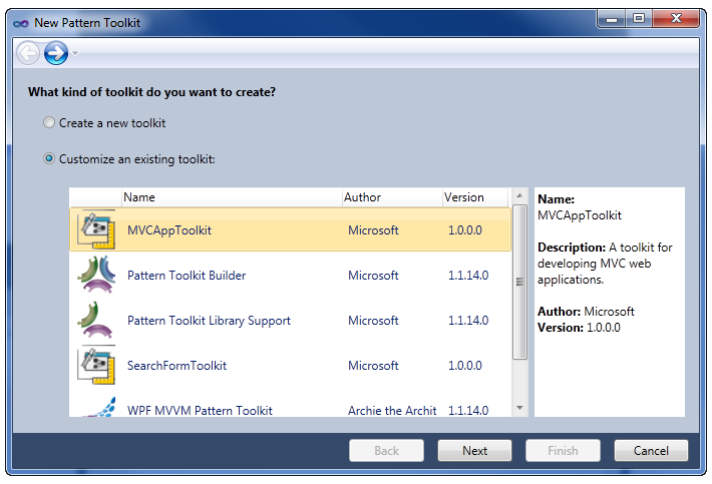


The essential ingredients of a Pattern Toolkit are:

- 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

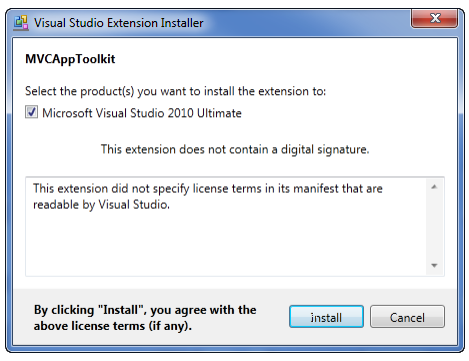
All you need to install and use a pattern toolkit is Visual Studio 2010/2012 Professional, Premium or Ultimate.

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

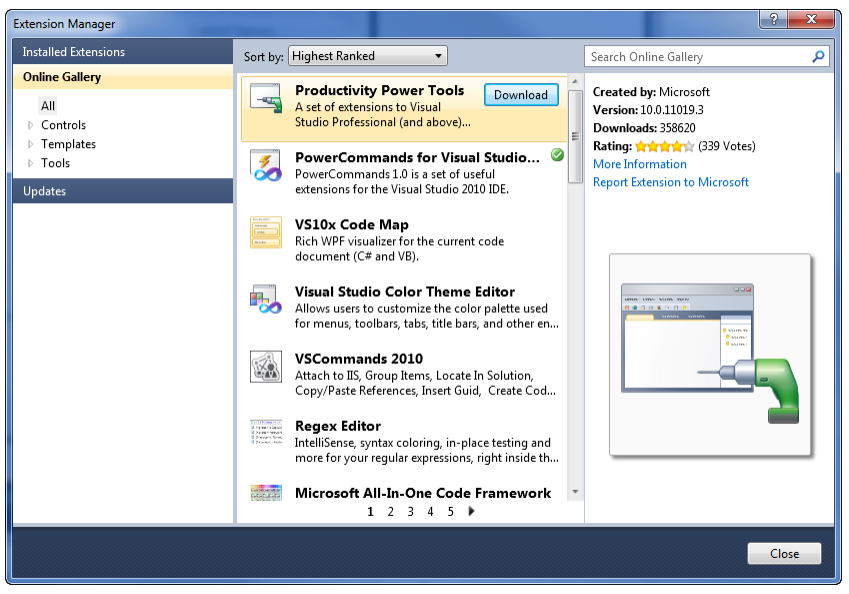
Installing a Pattern Toolkit

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.

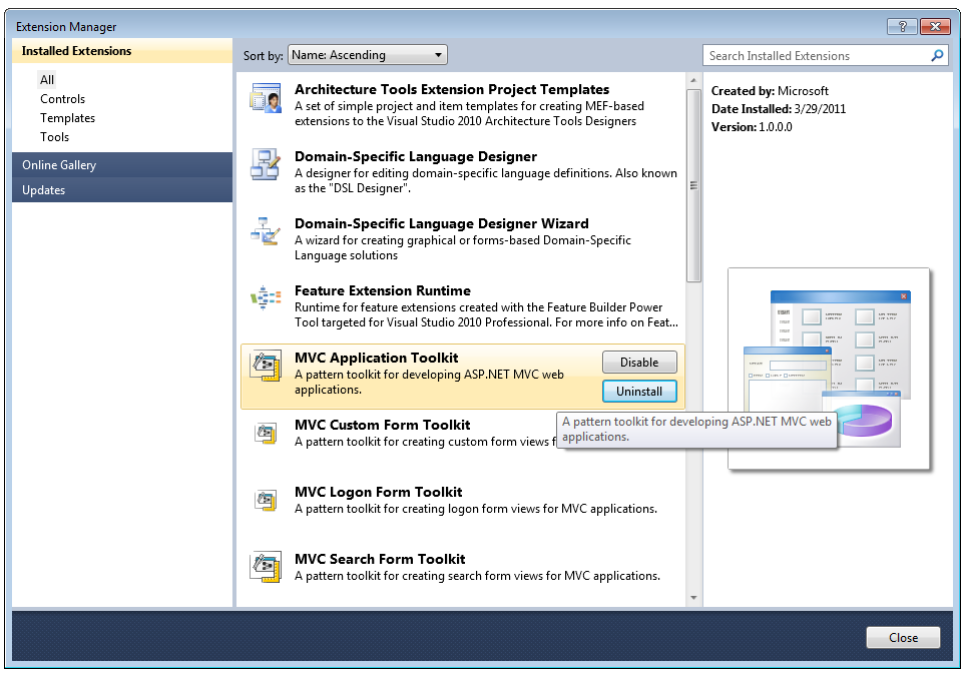


To install a pattern toolkit obtainable from the 'Visual Studio Gallery', you either download the *.vsix file to your machine and open it (as described above), or you use 'Extension Manager' in Visual Studio to 'Download' and 'Install' it in Visual Studio.



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

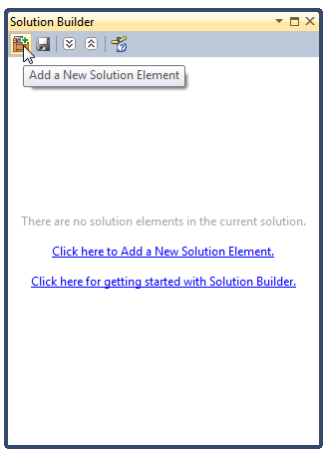
Once a pattern toolkit is installed, the contained pattern within it is available to be used.

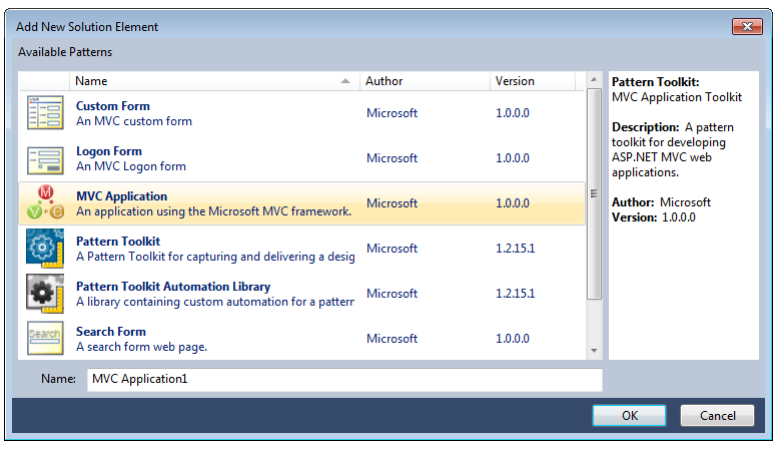
To the user using a pattern toolkit, they are really more focused on applying useful patterns to their solution, and therefore once installed; they really just want to use graphical elements that help them build their solution. For this reason, the terminology used from a user's perspective switches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



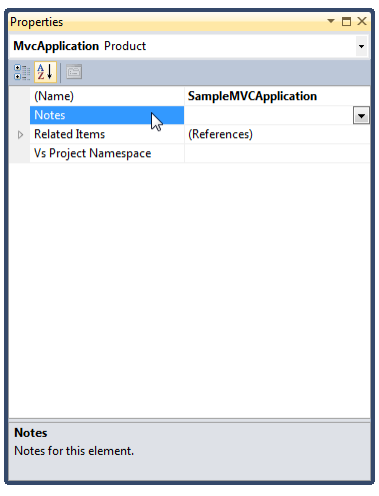


Viewing & Configuring Solution Elements

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



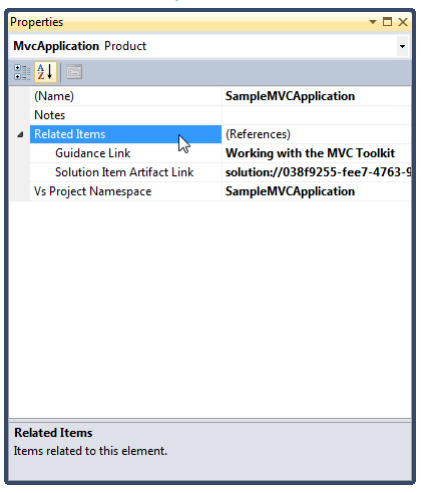
Selecting a solution element allows you to edit its properties in the 'Properties Window':

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.

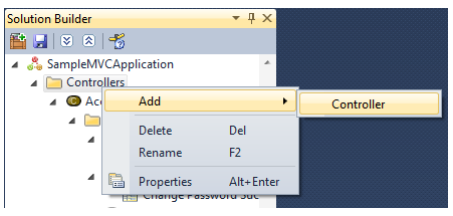
> **Note**: Depending on the type of related item, they can usually be modified (or fixed) to relate to other existing items, should the related item ever become un-related for whatever reason (i.e. deleted).

Notes Properties

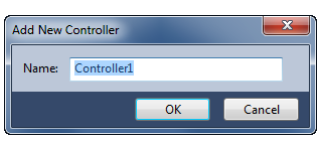
Every solution element has a 'Notes' property that allows you to leave design notes or annotations describing the current solution element.

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.

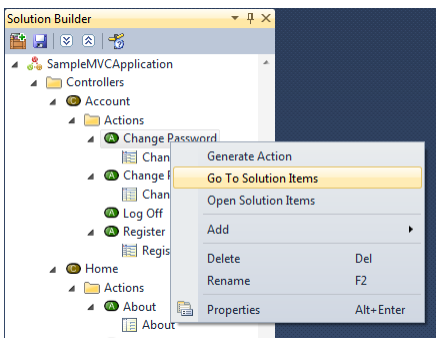


Once the menu is selected, the child element is named in the 'Add New' dialog:



Navigating to Related Solution Artifacts

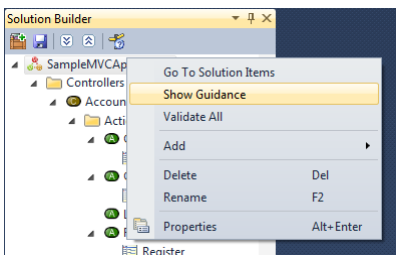
If the solution element has a 'Related Item' (one or more) that refer to solution artifact which are navigable in 'Solution Explorer' (called a 'Solution Item Artifact Link'), then you can navigate to them using the 'Go to Solution Item' menu.



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.

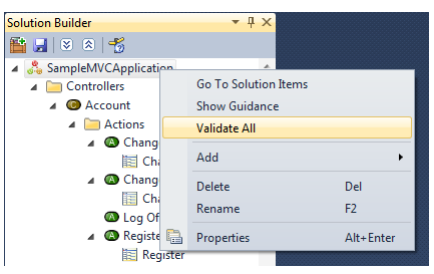


Validating Solution Elements

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.

You must also install the latest version of 'NuGet Package Manager', also available from the Visual Studio Gallery.

> **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.

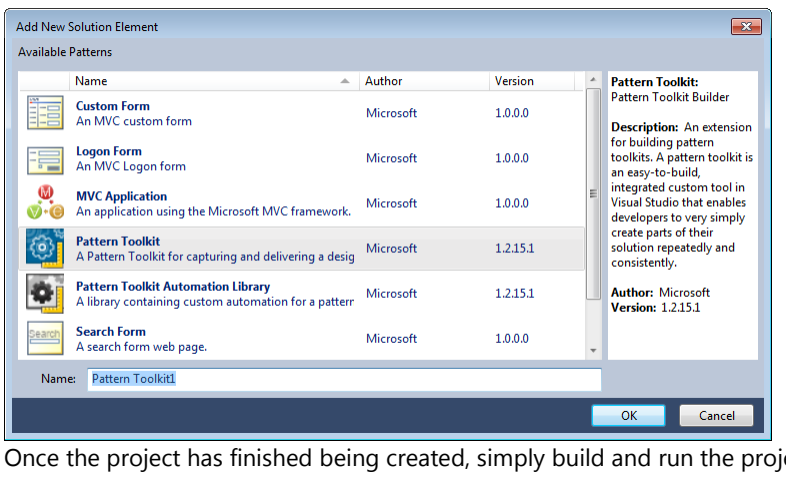
All or some of these steps may occur in any given iteration cycle. The last two steps for Deployment and Customization occur after development of the toolkit in releasing the toolkit, and modifying it or other toolkits later.

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.

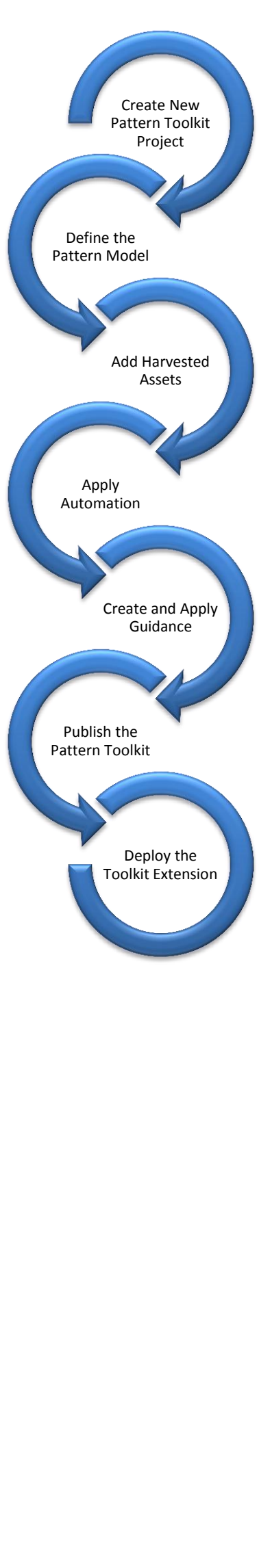
Create a new toolkit project by creating a new **'Pattern Toolkit'** from the Add New Solution Element button in the Solution Builder window.
Complete the project creation wizard and select the 'New Toolkit' option.



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.

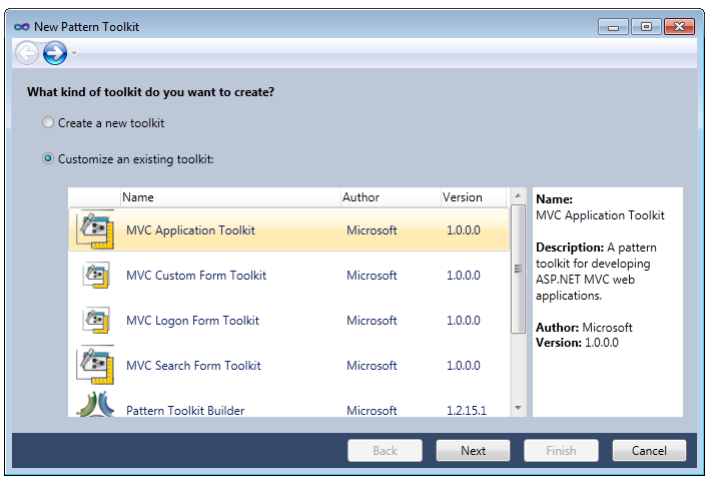
Deploy the Toolkit Extension

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.

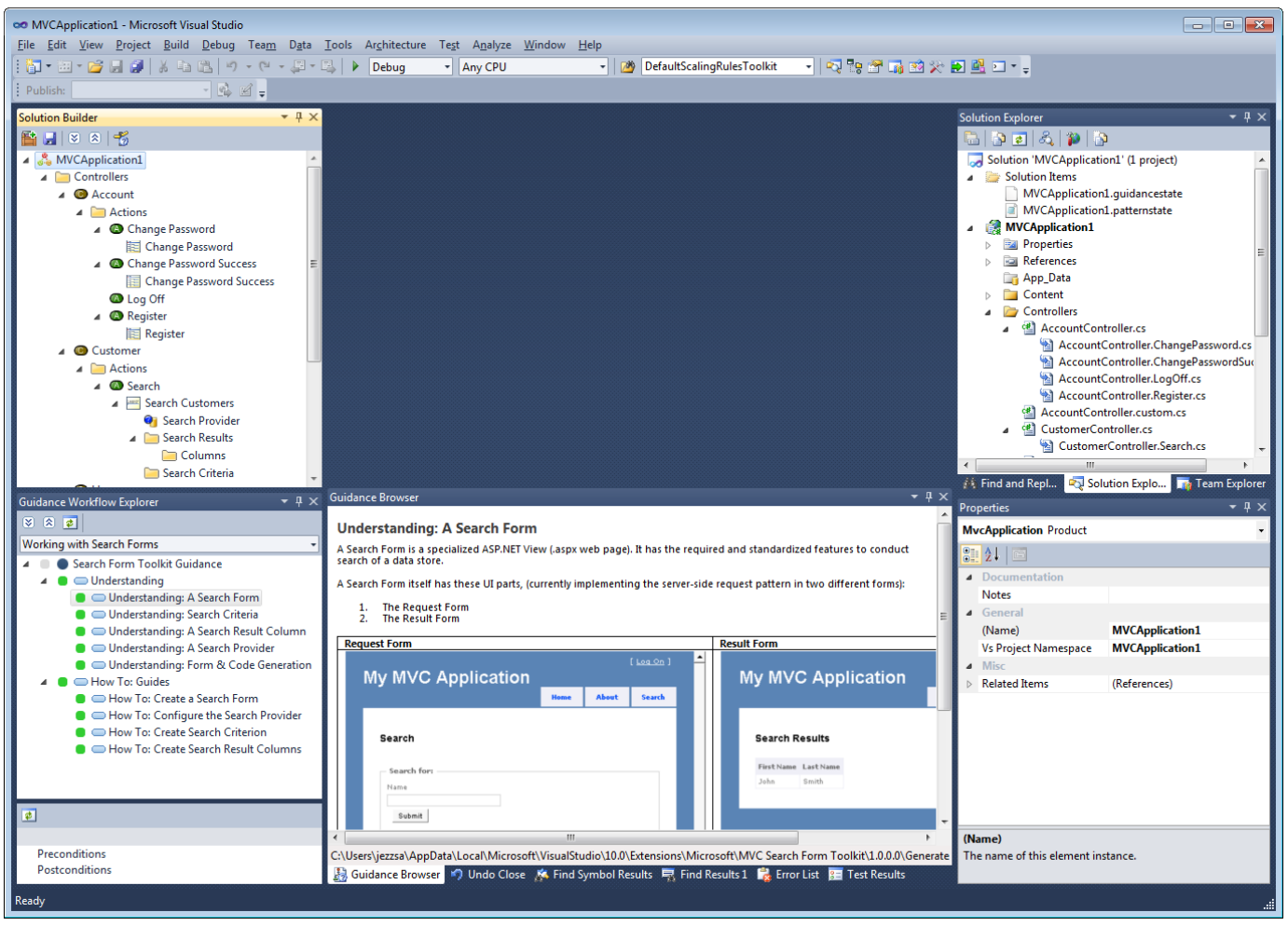
However, in general, applying these customization rules should be an exception to the rule, and should be used sparingly. Constraining customization of the pattern model can in some cases prevent appropriate customization, forcing a re-write of the pattern. For that reason, these rules are not applied 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:

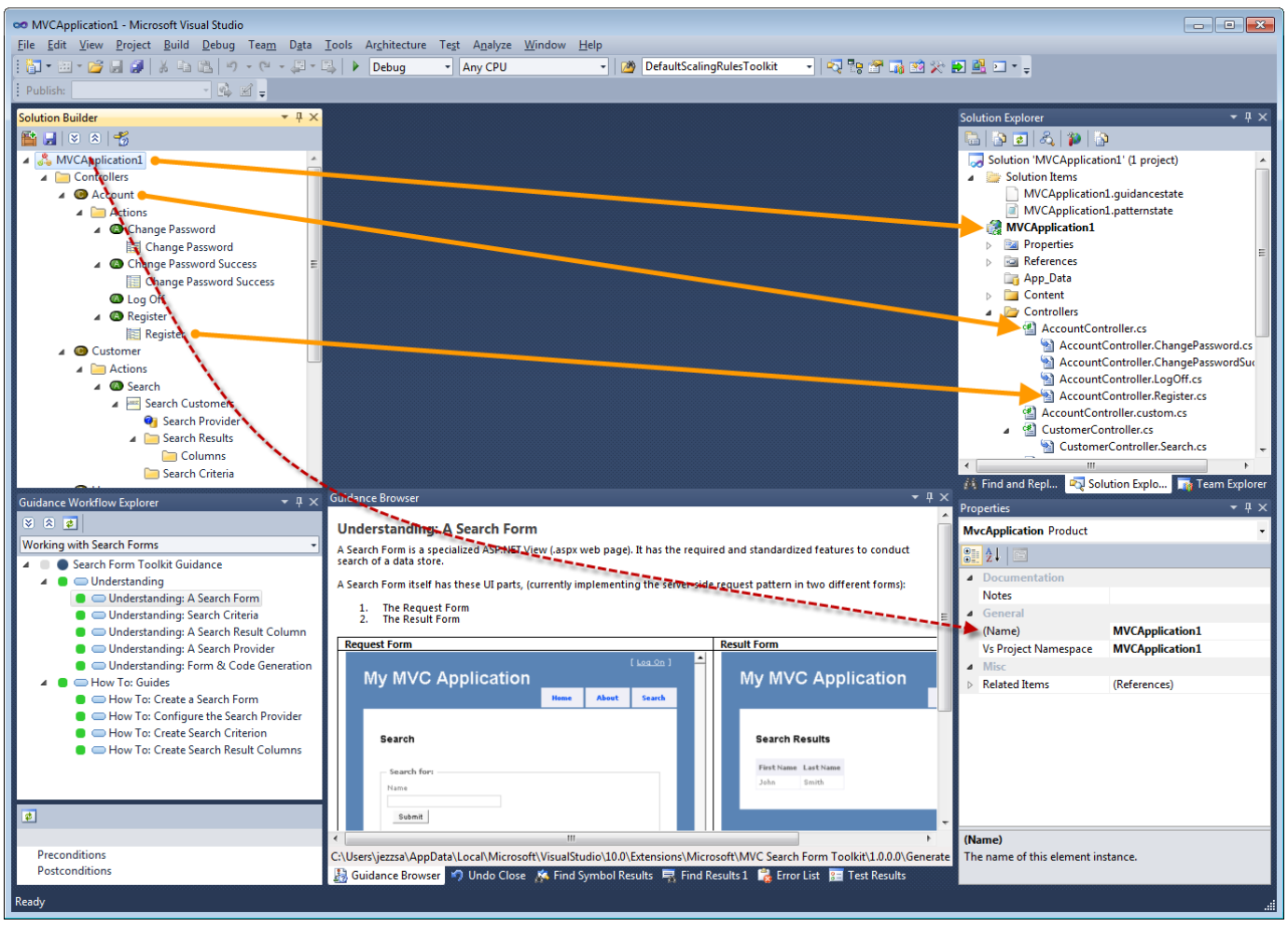
- View, and Browse the existing solution elements for the current solution.
- Configure the solution elements.
- Create new solution elements.
- Compose new or existing solution elements together.
- Invoke Automation.
- Navigate to other Related Artifacts in the development environment.
- Navigate to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



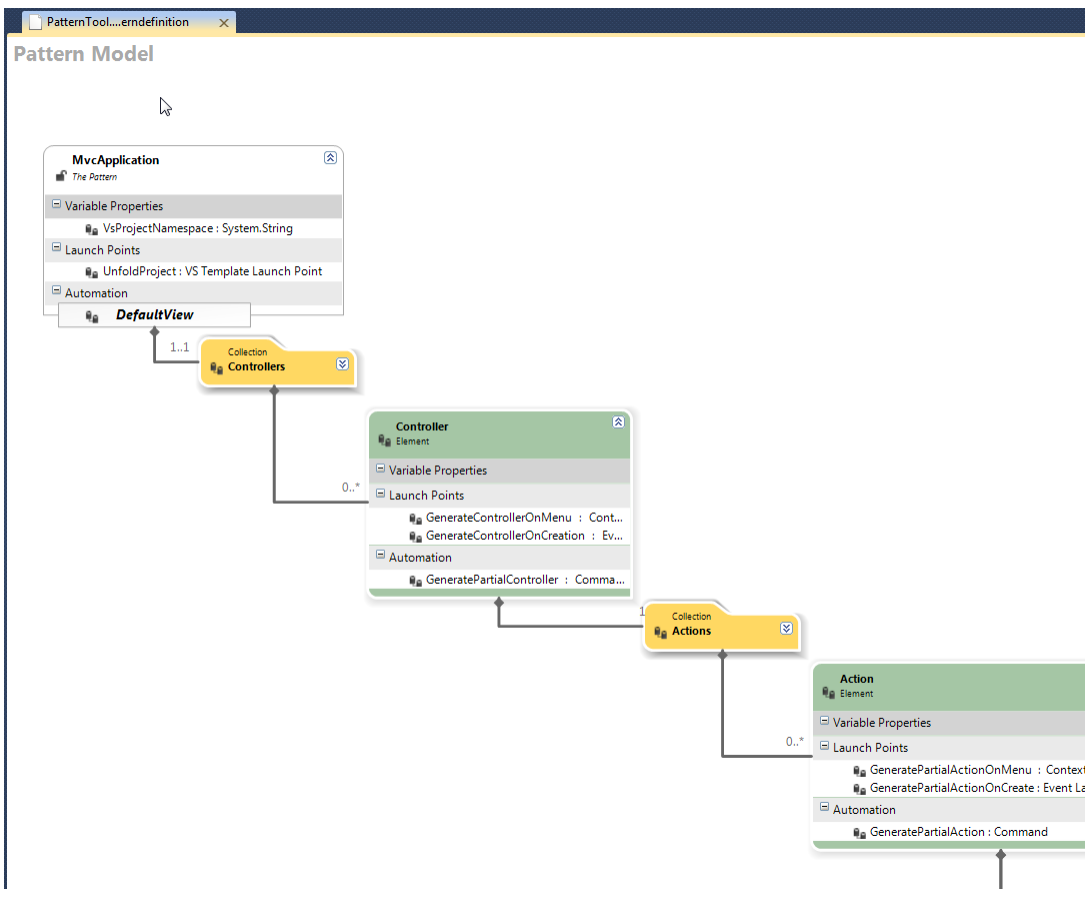
A user of a Pattern Toolkit, and the toolkit's built-in automation, work with defined properties of the solution elements to establish their present configuration and state. When verified as valid, the configured state is then used typically to automate and transform that state into a representative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



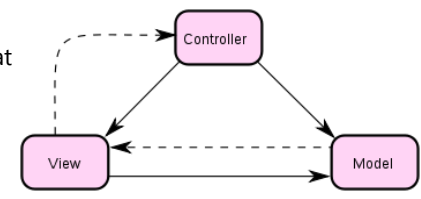
With this model you define all the descriptive elements of your model with names that reflect the pattern being described. Each property on each element (or collection) in the model represents a configuration point or variability point in the pattern, that someone using your model (or automation) can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

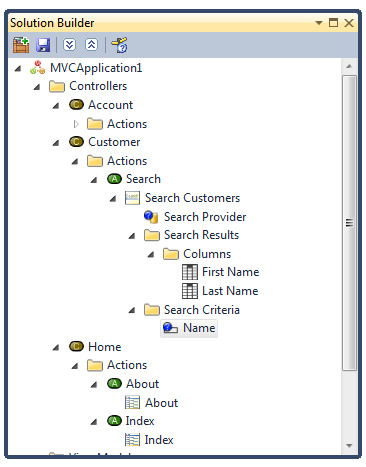
What it is and What it Isn't

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.

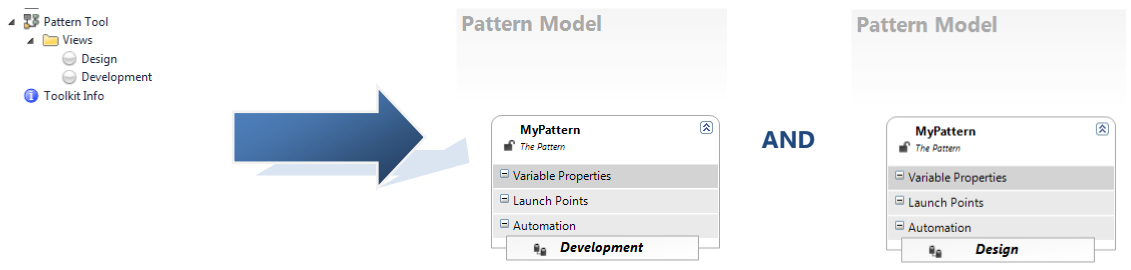
> **Note**: The most minimal variability that can be attributed to a pattern is one with only the pattern element, one view, no elements and no properties. Such a pattern has only one point of variability that can be configured by a user, and that is the name of the instance of the pattern created by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

What are Variability Views?

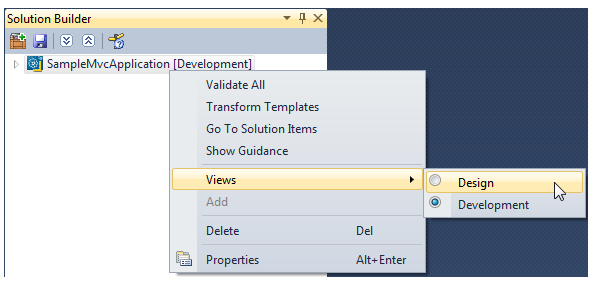
Every Pattern Model has one or more ‘Variability Views’. By default, a Pattern Model has one 'Default' view.

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.

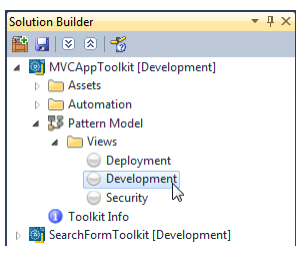


> **Note**: Every pattern has at least one view that is the ‘Default’ view. For smaller or simpler patterns that view is always represented to the user.

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

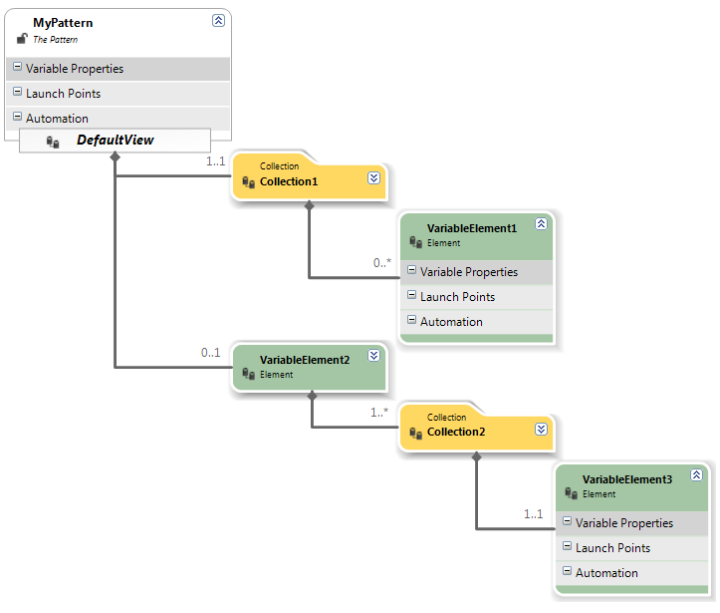
A Pattern Model for building Web Applications could utilize separate views: a view for ‘Development’ of the view navigation (i.e. MVC concepts), a second view for defining ‘Security’ concepts of the web application (i.e. authentication/authorization etc.), and a third view for defining ‘Deployment’ of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

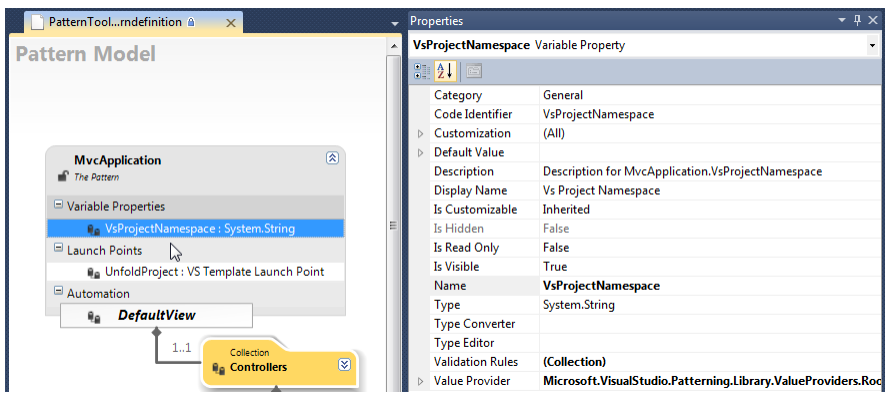
A 'Collection' shape is intended to be used by an author to represent either hierarchy or grouping of elements, even though the 'Element' shape can be used in exactly the same way. The choice is entirely the author's as to how to use collections and elements on the design surface of the Pattern Model.

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.

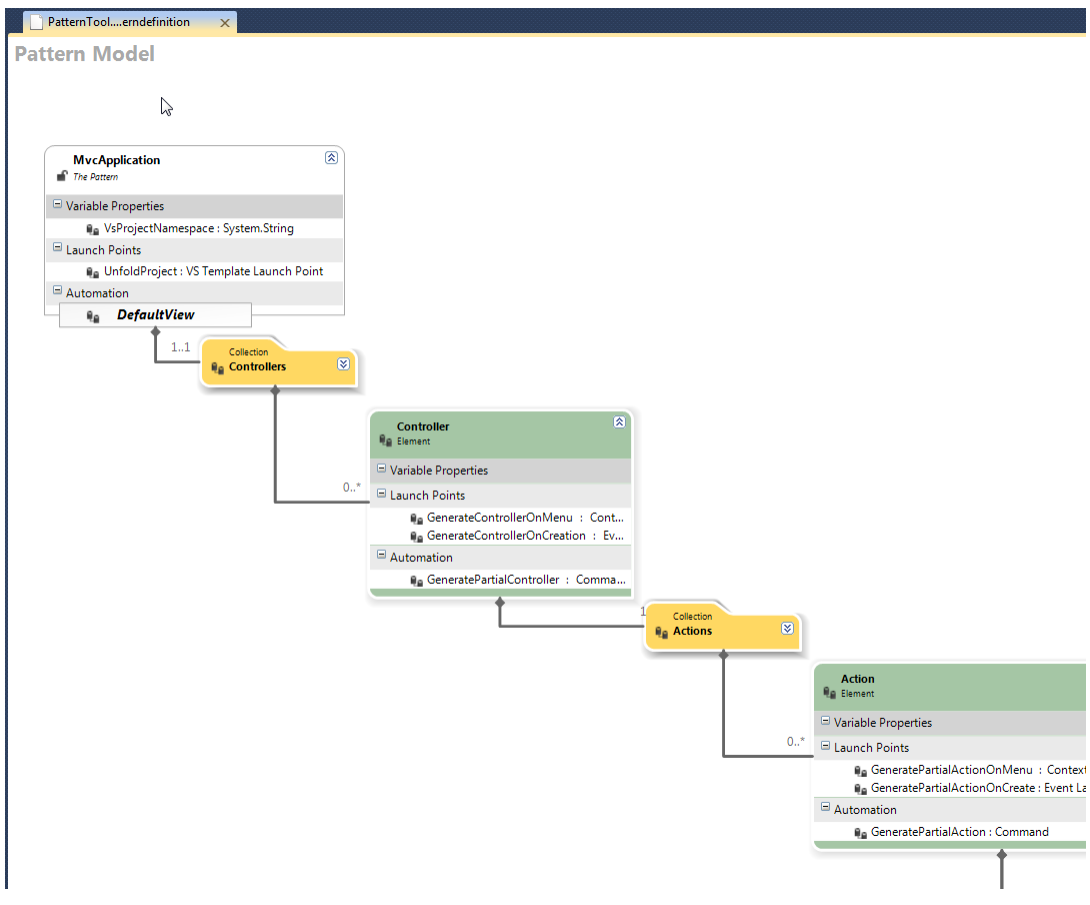
Not all 'Properties' must be visible to or be modifiable by a user; they can be hidden or made read-only, and access restricted to automation only.

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

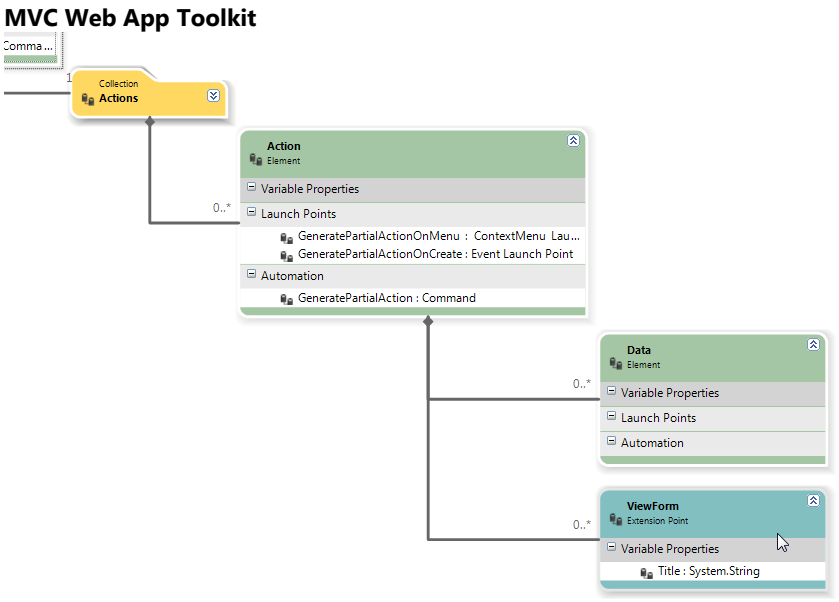
Declaring For Extensibility

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



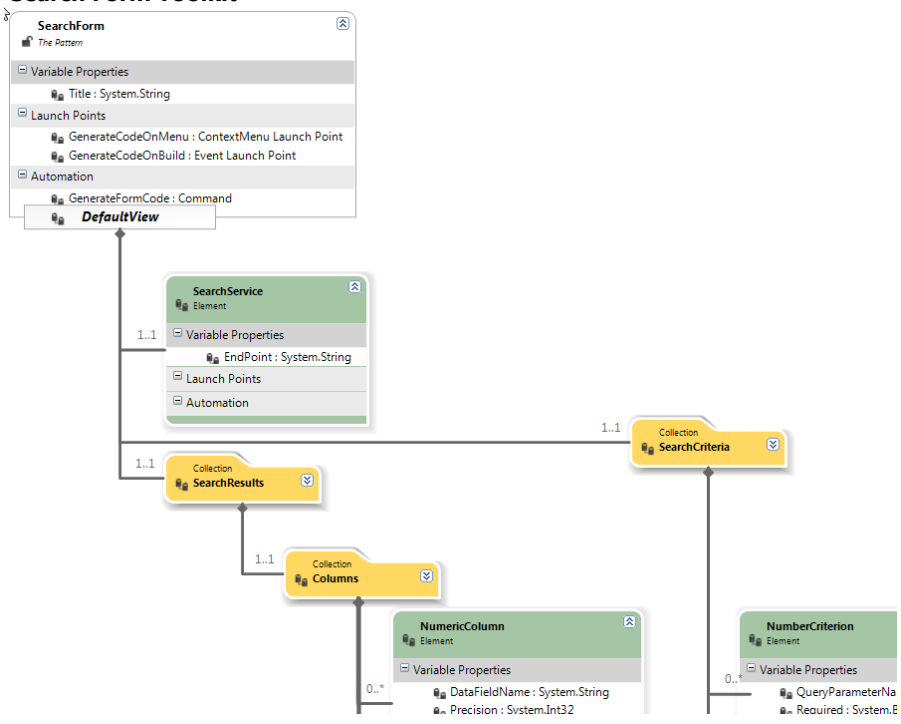
Implementing for Composability

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



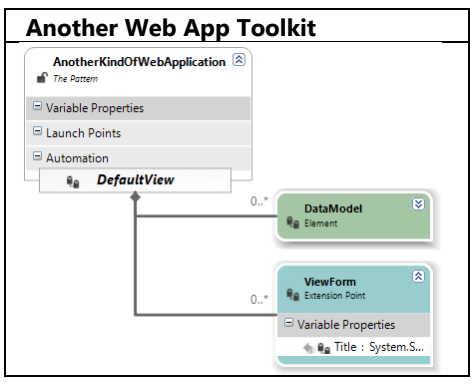
Integrate for Reusability

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.

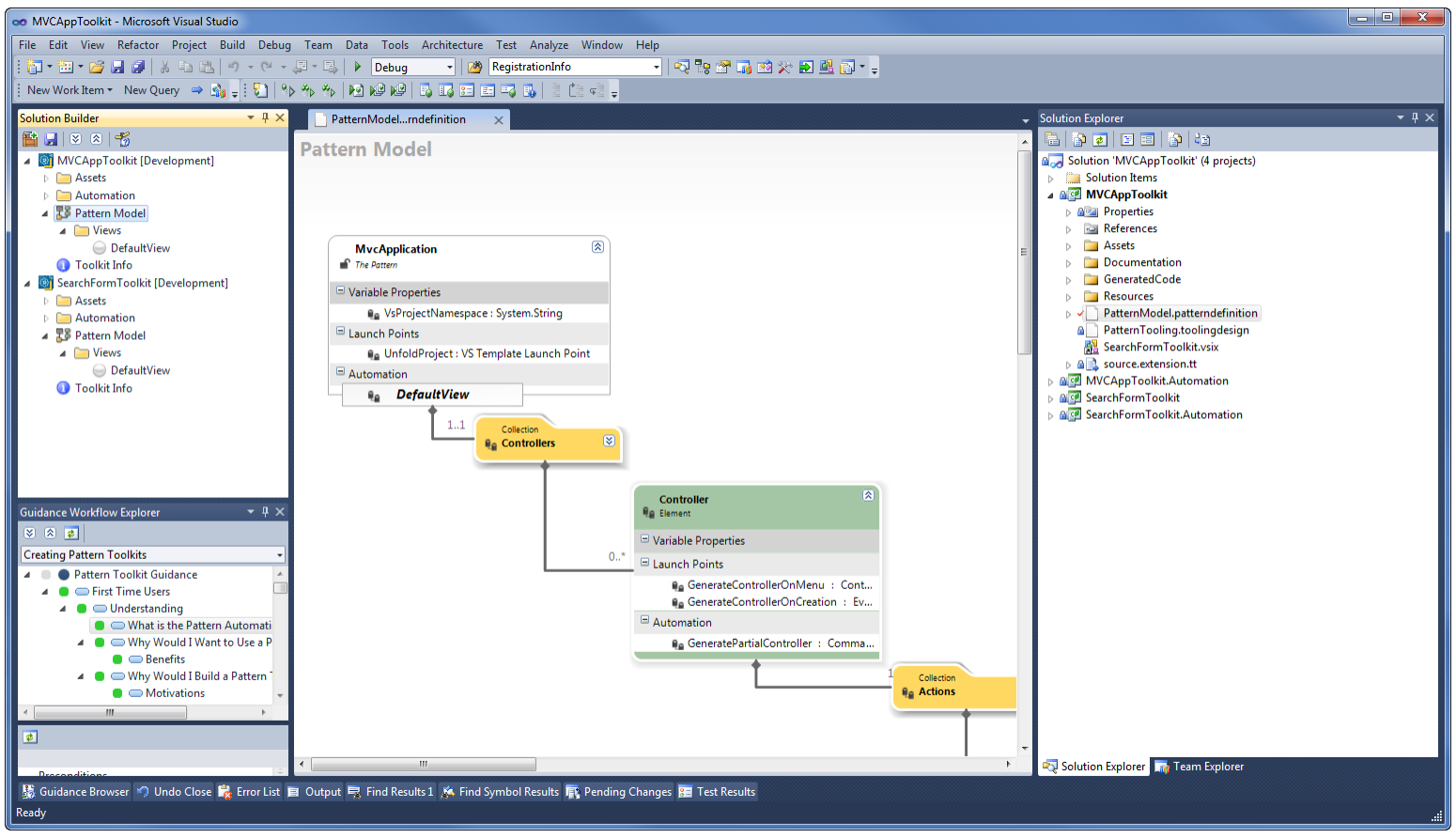
Another Web App Toolkit



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).

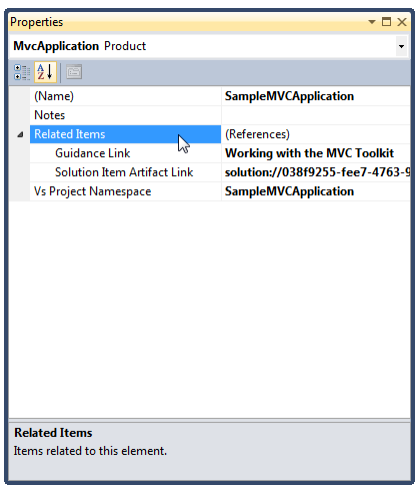
Note: the 'Pattern' element always supports 'OneToMany' Views. This cardinality is not configurable.

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.

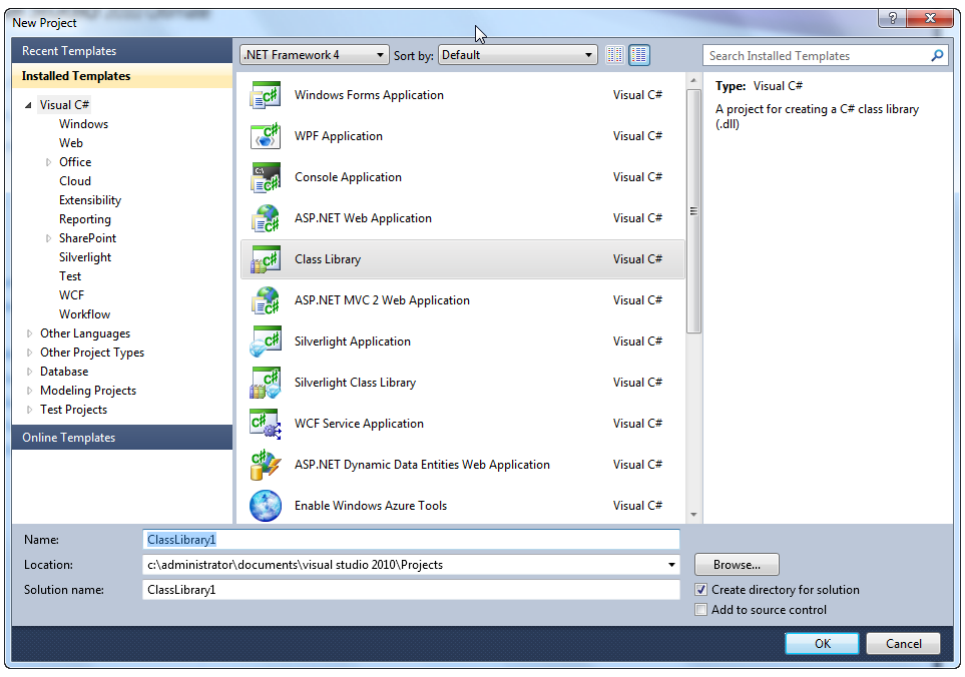
> **Note:** All Automation assets and their dependencies are installed and deployed by the toolkit.

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.



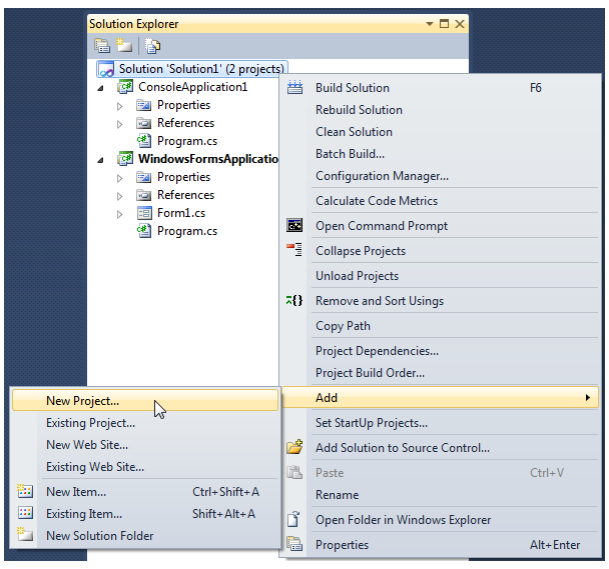
The 'New Project' dialog box displays all the project templates installed in Visual Studio, categorized usually by development language first, and then technology.

> **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.

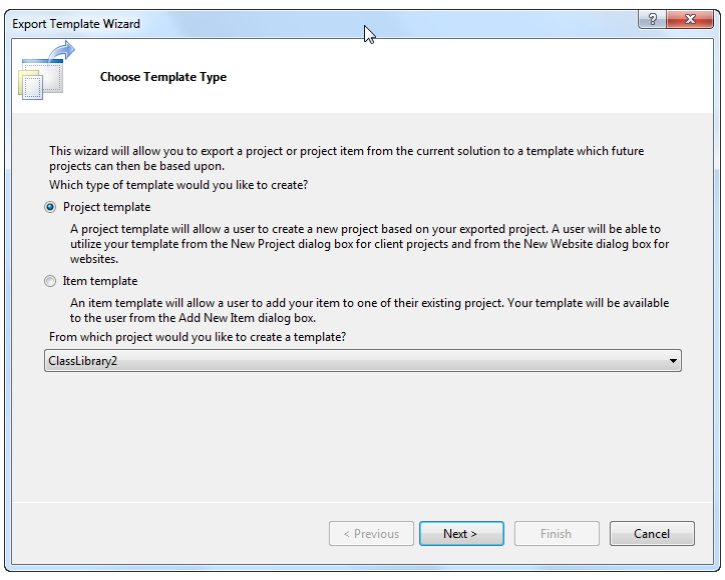
Template Makeup

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.

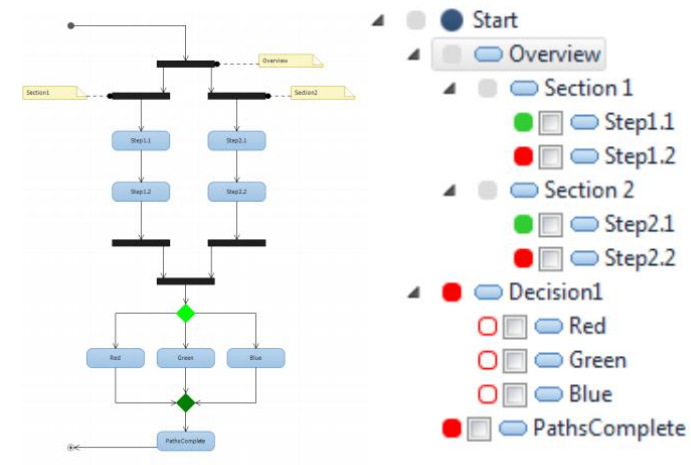
That guidance is composed of one or more a 'Guidance Workflows', each of which is a multi-pathed workflow of individual 'Guidance Documents' (topics), each of which has some browsable content, and each of which may or may not have state to indicate progress through it, and also contain links that automate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

What is a Guidance Workflow?

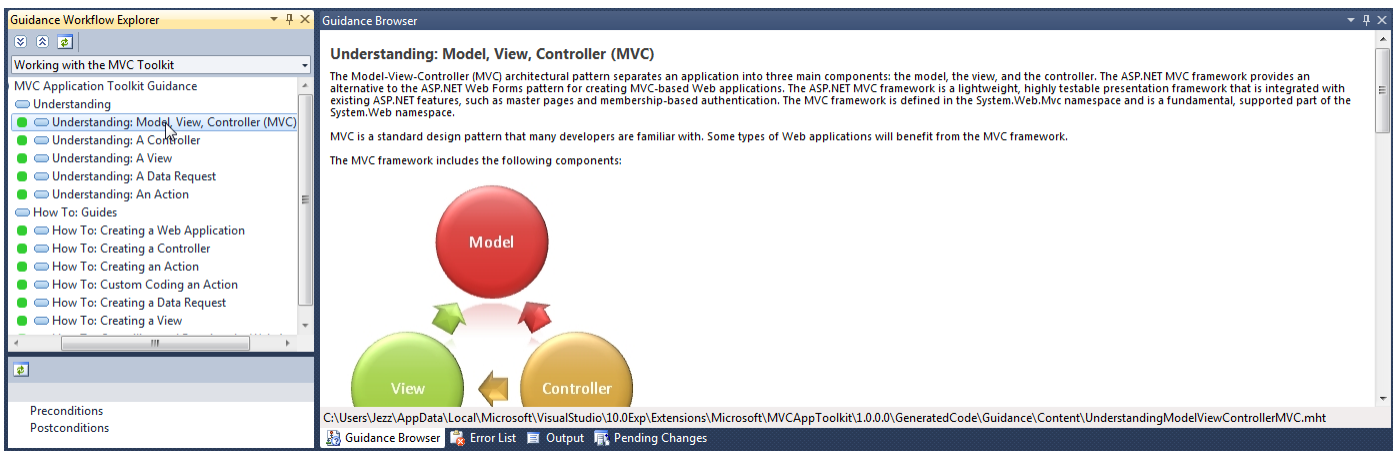
A Guidance Workflow describes the structure (layout) of a set of guidance, that may include both informative and instruction guidance steps.

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?

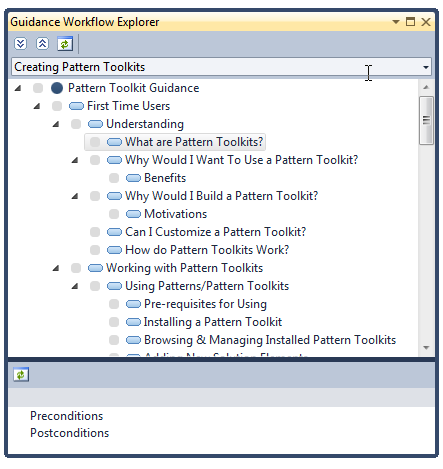
A Guidance Document is simply the content document which can represent a topic of any node in the guidance workflow.

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?

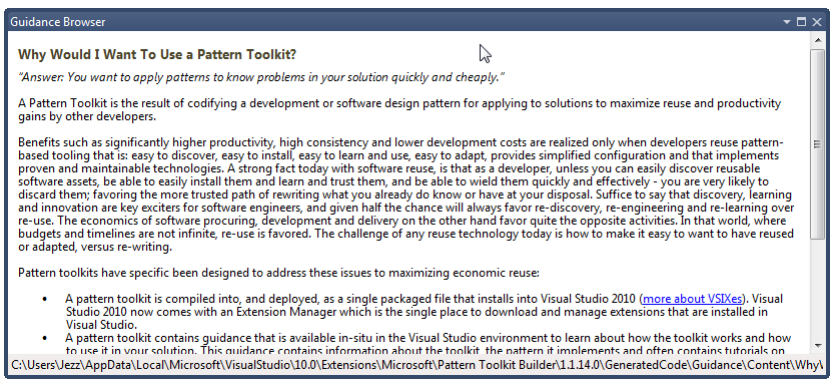
The 'Guidance Explorer' window is the tool window in Visual Studio where you can select, view and interact with the currently active 'Guidance Workflow'.



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

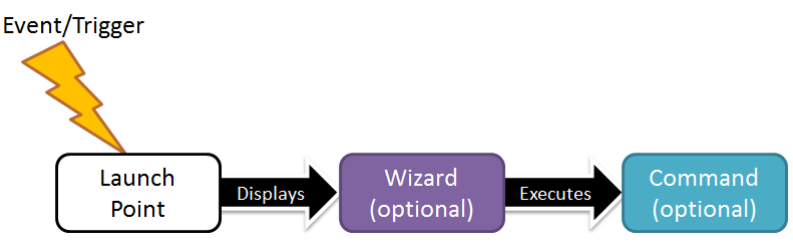
It is really the 'Launch Point' automation extension type that brings together all the other automation extensions to deliver the overall value of automation to a pattern toolkit.

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.

How To: Guides

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

Using

Installing and developing solutions with pattern toolkits using Solution Builder.

For guidance on using pattern toolkits and 'Solution Builder', see 'Using Patterns in Solution Development'.

Understanding: What are Pattern Toolkits?

See What are Pattern Toolkits for the concepts.

How To: Install/Uninstall Pattern Toolkits

See Pre-Requisites for Using.

Obtain the Pattern Toolkit Installer

Pattern Toolkits are deployed in a Visual Studio Extension (VSIX) package (a file with the *.vsix extension). A VSIX is a single file, self-installing package containing everything needed to install and run a Visual Studio extension.

You must first obtain a VSIX installer (i.e. MyPatternToolkit.vsix). You can either obtain the *.vsix 'privately' from within your organization, or browse for it 'publically' from the Visual Studio Gallery

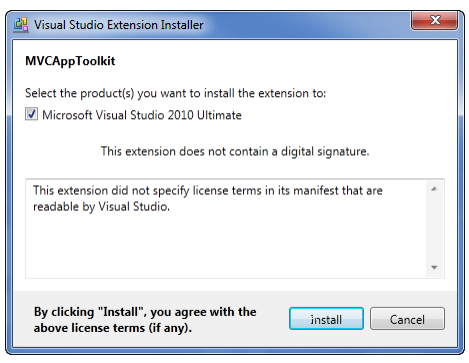
Run the Installer

File Installation

If you have obtained the VSIX file directly, open the file directly from your desktop (hard drive or network location).

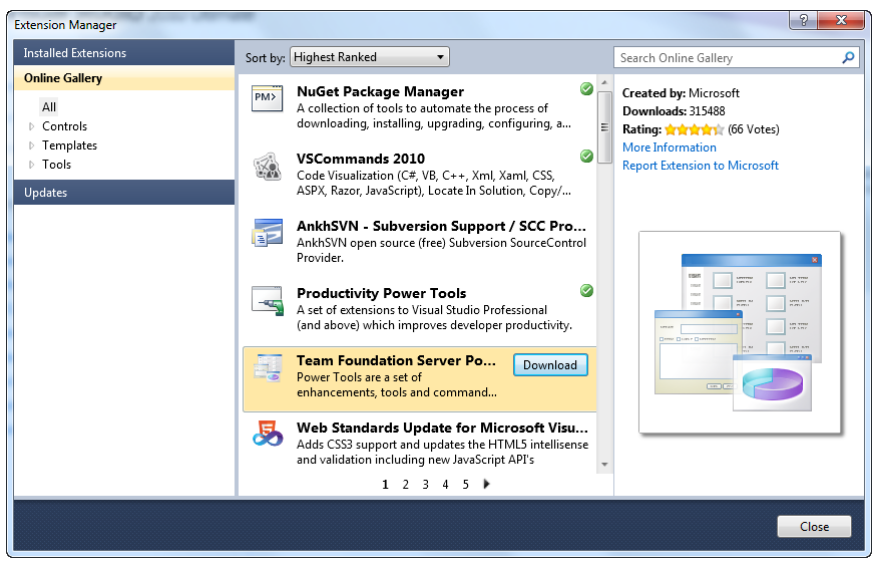
The VSIX Installer will run, to verify your pre-requisites and will install the toolkit into Visual Studio.

> **Note**: Pay close attention to whether the VSIX includes a license agreement, and whether it includes a digital signature to ensure its authenticity and originating source. Both these things are displayed on the front page of the installer when run.



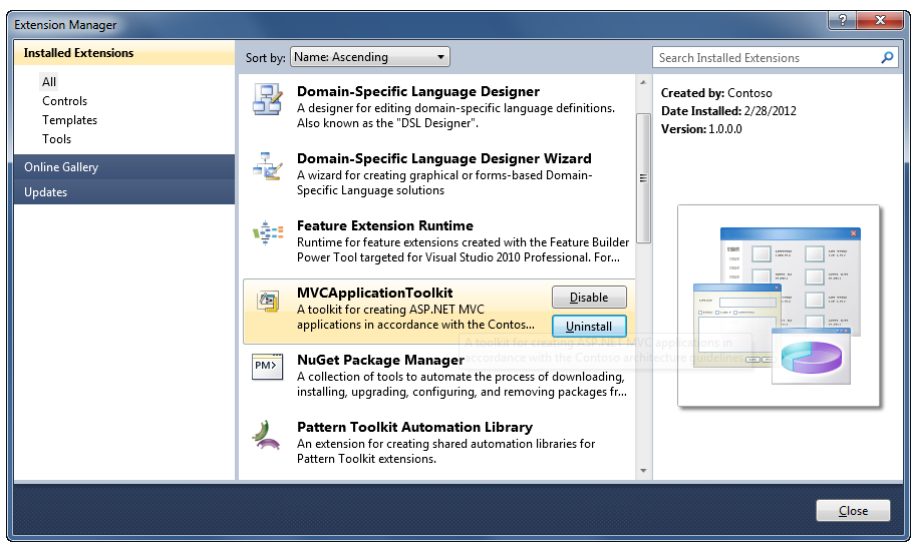
Gallery Installation

If you wish to obtain your VSIX from a gallery, such as the Visual Studio Gallery, then you can search, browse, download and install the VSIX directly from the gallery from the 'Extension Manager' in Visual Studio (Tools | Extension Manager).



Verify Install

Once installed, open Visual Studio, and open the 'Extension Manager' (Tools | Extension Manager) to see your toolkit installed.



Uninstall the Toolkit

You can either 'Uninstall' or just 'Disable' a toolkit directly from the 'Extension Manager'.

> **Note**: Once 'Uninstalled' or 'Disabled', you need to restart Visual Studio to effect the change.

How To: Use a Pattern

> **Important:** In order to use the pattern contained within a Pattern Toolkit, the Pattern Toolkit must be installed. You can verify this by looking at the 'Installed Extensions' in the 'Extension Manager' dialog in Visual Studio (Tools | Extension Manager).

A quick note for understanding; as users, when we say 'use' a pattern (from a pattern toolkit) what we are really doing is using a pattern toolkit to create for us an instance of the pattern that can be parameterized for our solution. This pattern instance is called a 'Solution Element', and we work with it in our solution.

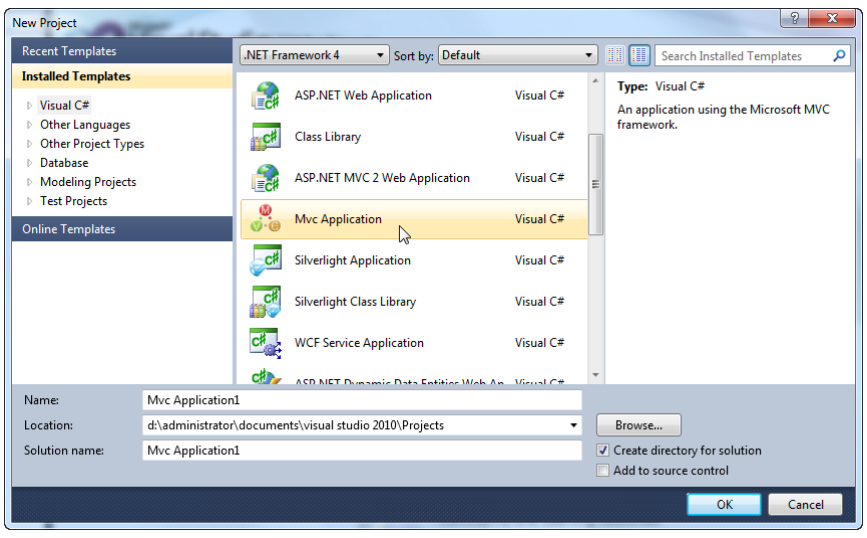
There are two ways to start using the pattern.

Create From the Add/New Project/Item Dialog

If the pattern toolkit includes a project or item template, it will appear in the Add New Project/Item dialog in Visual Studio.

From an empty environment, or existing solution, open the Add/New Project/Item dialog in Visual Studio.

> **Tip**: Use the 'search' box in the top right corner of this dialog to locate the template quicker.



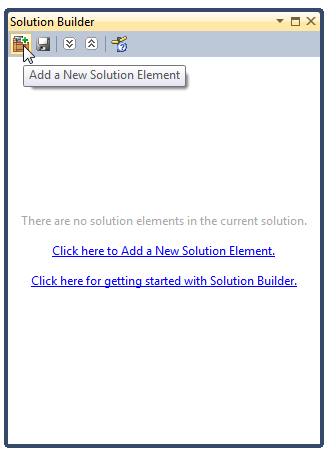
> **Note**: If you cannot find the pattern from this dialog, you can always find it from the next method below

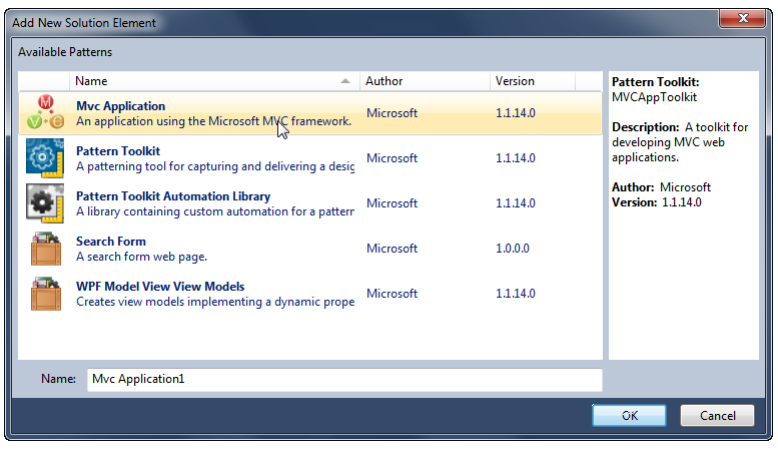
Create From Solution Builder

Otherwise, all patterns can be found in the 'Add New Solution Element' dialog.

> **Note:** You must have an existing solution open in order to create a new pattern this way. Create a new blank solution if you don't already have one open.

From an existing or new solution, open the 'Solution Builder' tool window, and click the 'Add New Solution Element' button (or hyperlink in the window).





> **Note**: Even though a pattern may or may not appear in the Add New Project/Item dialog in Visual Studio (depending on whether it supplies a project or item template), all patterns appear in the 'Add New Solution Element' dialog, in the 'Solution Builder' tool window.

Apply the Pattern

In either case, a new instance of the pattern will be created for you and displayed in 'Solution Builder', and you work with the created 'Solution Elements' in that window with the tools and guidance the pattern toolkit provides for you.

See Viewing and Configuring Solution Elements for examples of the kinds of automation and guidance the toolkit may provide.

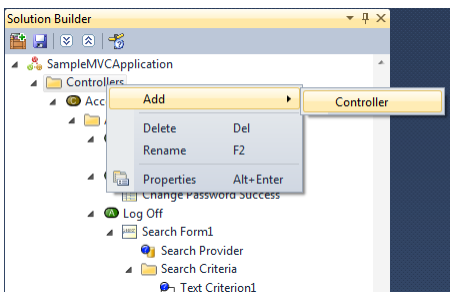
How To: Add New Solution Elements

If a solution element has child elements defined in its pattern, then the 'Add' menu will appear in the context menu of the element.

Add a New Element

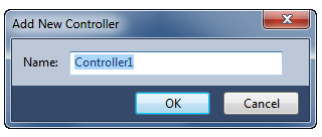
You can add new instances of child elements by right-clicking on the element and selecting the type of child to add from the 'Add' menu.

> **Note**: Some elements in a pattern may only allow one instance of that element, and in those cases, the menu may be disabled when one instance is already present.



Name the New Element

Give the new element a name.



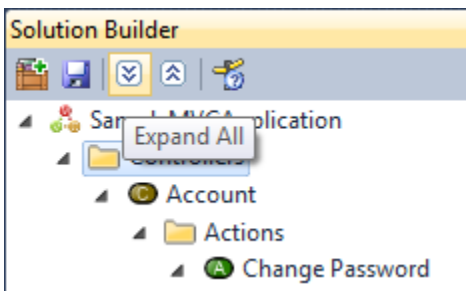
> **Note**: If any automation executes as a result of creating this new element, then you may see activity in the development environment as this element is created.

How To: Control the display of Solution Elements

In 'Solution Builder' you can control how solution elements are presented in the tree layout:

Expand/Collapse

The Expand/Collapse buttons apply to all elements in the Solution Builder.

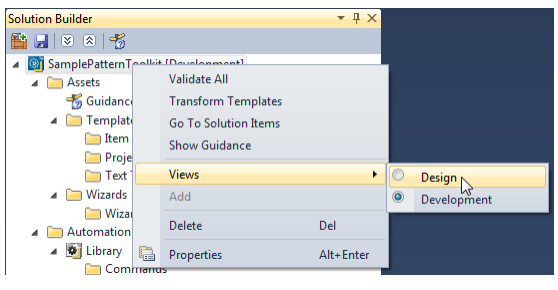


How To: Work with Multiple Views

If a pattern supports multiple views (or aspects), then you can switch between these views in 'Solution Builder'.

Patterns will provide multiple views to focus development on different distinct aspects of the pattern.

> **Note**: Every pattern has at least one view, the 'default' view, which is the current view when the solution is opened.
> If a pattern only has one view, then the 'Views' menu is not displayed.

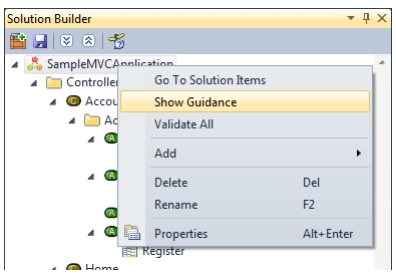


How To: Find the Guidance

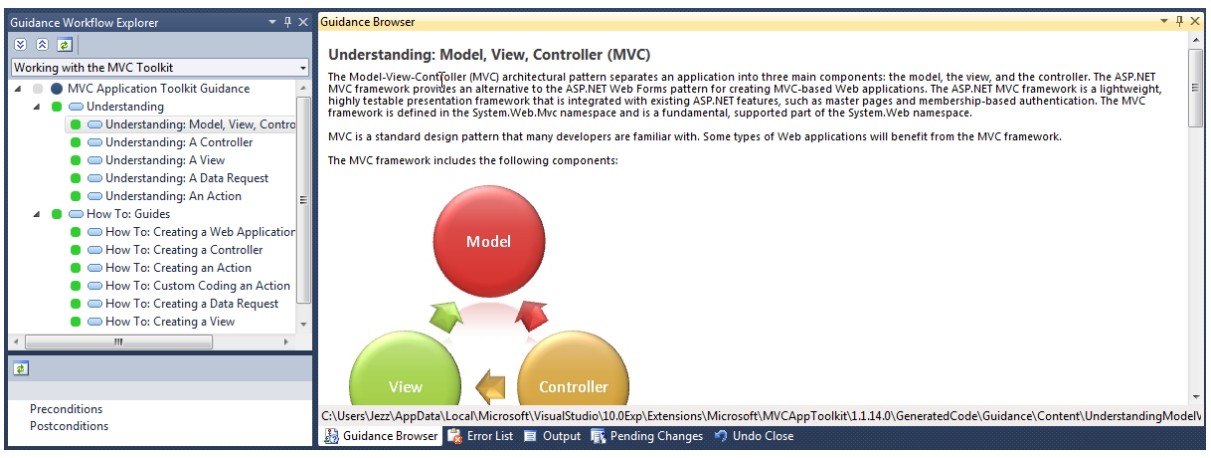
If a solution element is associated to any guidance, then you can locate and browse that guidance from the element.

Show the Guidance

Right-click on the element and select the 'Show Guidance' menu.



The guidance then appears in the 'Guidance Explorer' and 'Guidance Browser' tool windows.



Note: It is not common for this menu to be available for all elements in the pattern. Typically, it is only provided for the top level element of the pattern, or for key child elements that have important distinct processes related to them.

How To: Run the Automation

In most cases, automation automatically runs for you when working with solution elements in Solution Builder. The toolkit providing the pattern takes care of this for you. This is the whole benefit of having toolkits implement solutions for you based on patterns.

For example, you may see new projects and files created or updated in 'Solution Explorer' as you add or change solution elements. You may see errors appear in the 'Error List' window informing you when things aren't configured correctly, or when critical steps have not been followed.

Expect that this kind of automation may be automatically performed when: new elements are created or their properties changed, or when the solution is built, or when saved etc.

Some automation needs to be executed manually.

Some automation may require special conditions to be satisfied before it can execute. Code generation is typical example of such automation.

In any case, automation may be conditional on a number of things. Such as: the state of the solution elements in 'Solution Builder', or the state of related projects and files in 'Solution Explorer', or maybe even the status of other systems and services inside or outside the development environment.

Executing Automation Manually

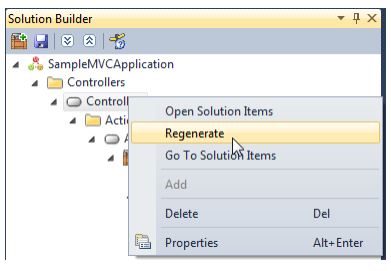
Depending on how, where and when the specific toolkit chooses to apply automation in the application of the pattern, it may need to be executed manually at certain times.

Sometimes, this automation is critical to the patterns development, sometimes they are ancillary. For example, an 'Export' function may not be necessary to get the pattern completed in the solution, although very useful for extracting or persisting some data as a result of applying the pattern to the solution.

The toolkit may guide you to when these things are ready for executing, either with guidance, or perhaps with errors in the 'Error List' to prompt you to complete a task. Sometimes, it is apparent from the pattern you are applying.

Manual automation is typically run from content menu items on solution elements in Solution Builder.

Simply click the menu item to execute the function, and the toolkit will manage the rest.

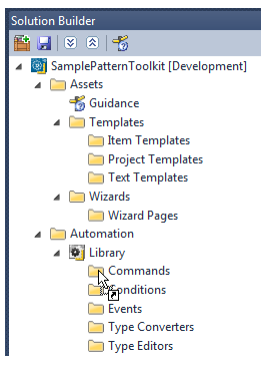


How To: Use Drag and Drop

It is common for solution elements in 'Solution Builder' to allow drag and drop of data from other windows (i.e. 'Solution Explorer').

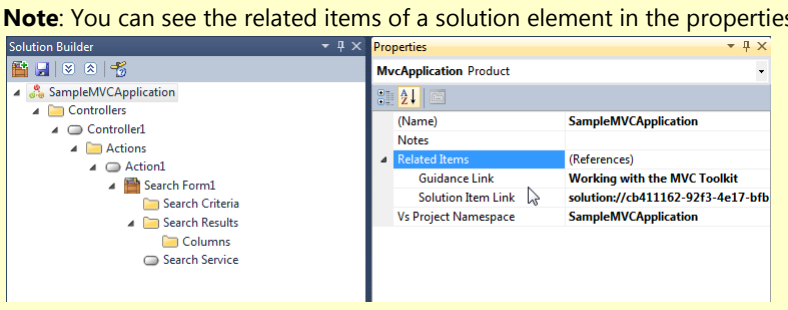
The source for the data can be anything, e.g. files from a hard drive, items in a solution, records from a database etc.

Simply drag data over solution elements in the Solution Builder window, and if the mouse allows, drop the data on the solution element.



How To: Navigate to or Open Solution Items

It is common for solution elements in 'Solution Builder' to be related to one or more solution items displayed in 'Solution Explorer'.

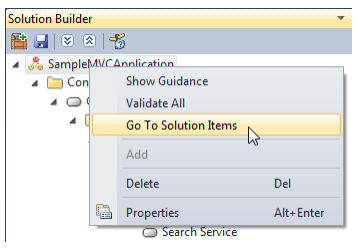
> **Note**: You can see the related items of a solution element in the properties of the solution element.
>
> 
>

For these solution elements, you will be able to navigate from elements in 'Solution Builder' to project items in 'Solution Explorer'.

> **Note**: Some solution items cannot be opened in any editor, such as: the solution, any folders and projects. In these cases you will only be able to navigate to these solution items, which simply selects them, and displays them in 'Solution Explorer'.

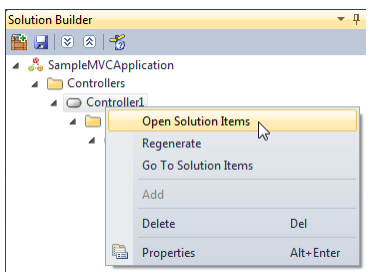
Navigating to Related Artifacts

To select and navigate to the solution items in 'Solution Explorer', right-click and select the 'Go To Solution Items' menu.



Opening Related Artifacts

To open the solution items in the development environment, right-click and select the 'Open Solution Items' menu, or double-click on the solution element.



Fixing Invalid Related Artifacts

If the related item cannot be found, or does not exist in the solution, then the related solution items may have been moved, renamed or deleted from the solution.

You can fix the link to the item. See Fix Related Items for details.

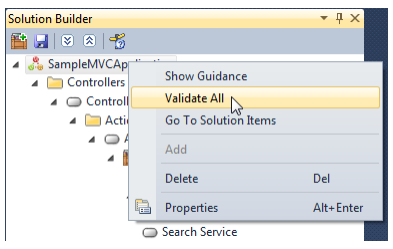
How To: Validate Solution Elements

Use validation to verify that the solution elements, and therefore your application of the pattern, is correctly configured in 'Solution Builder'.

Depending of the pattern toolkit you have installed, validation may occur automatically as you create and configure solution elements (i.e. when a property changes, or when solution built), or you are given menu commands to execute validation manually. Some toolkits offer both.

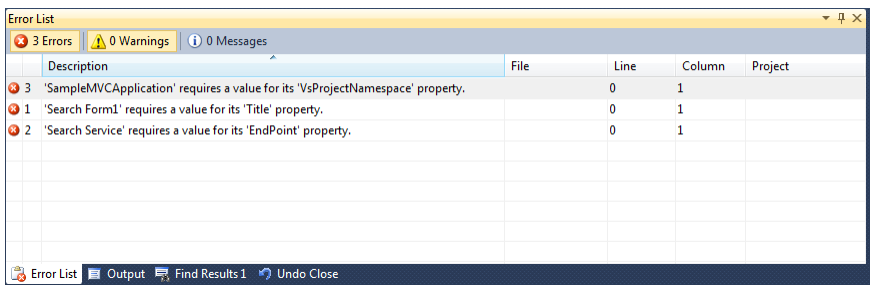
Validating Elements

Typically, menu items like the 'Validate All' menu are provided on the top level solution elements.



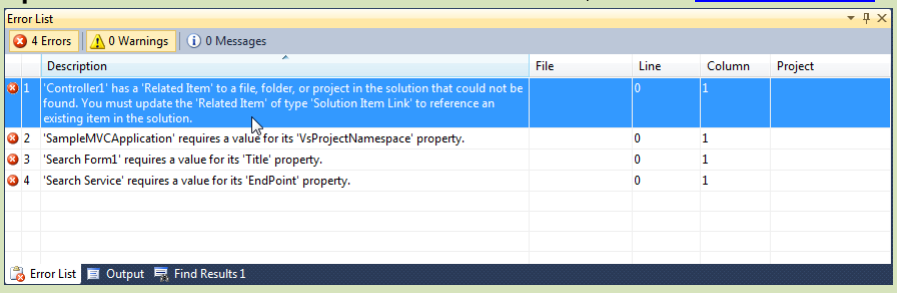
Validation Errors

Validation errors and warnings are reported in the 'Error List' tool window (CTRL + W, E).



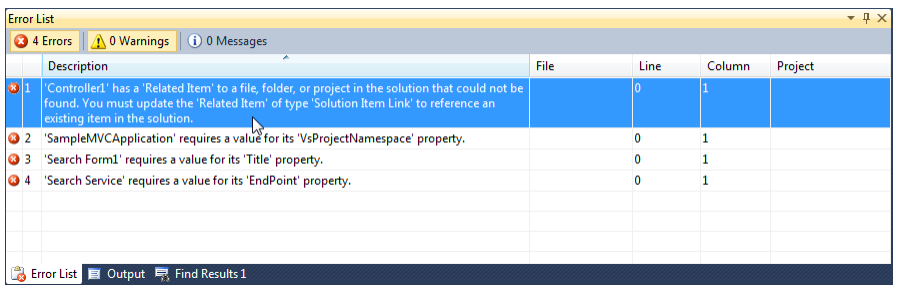
Note: Double-clicking on a validation error does not currently navigate you to the offending element in 'Solution Builder'.

Tip: If a validation error refers to broken 'Related Items', then see Fix Related Items.



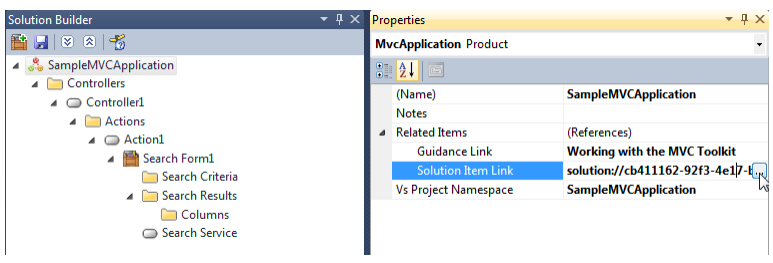
How To: Fix or Related Items

If a related item of a solution element cannot be found in the current solution, then an error appears in the 'Error List' window similar to this:

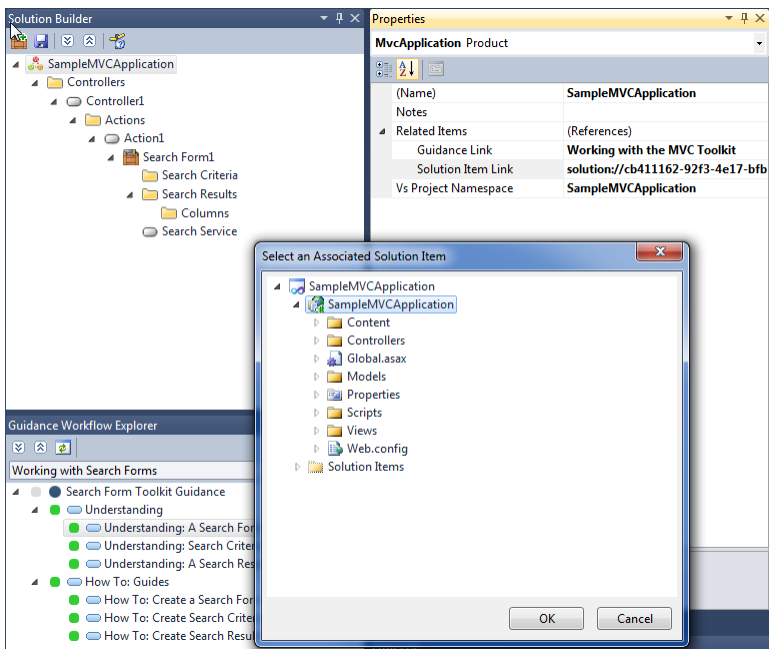


There are many types of related items for a solution element. They may refer to items in the Solution Explorer, or in fact any other source in Visual Studio. Some related items may even refer to sources and services outside the Visual Studio environment.

Depending on the type of Related Item, the value of that item in the 'Properties Window' will provide the necessary editors to help re-select or define a new value.



For the 'Solution Item Link' kind of related item, a solution picker helps you select the correct file, folder or project to relate to.



How To: Troubleshoot Pattern Problems

Occasionally, problems with specific toolkits may occur. Some problems are known temporary glitches and can be worked around, others may result in blocking issues that prevent further toolkit usage.

Diagnosing and Reporting Issues

Either way, these problems need to be identified and reported to the toolkit author by using their provided feedback mechanism.

See Troubleshooting Toolkits for more details on these processes.

Authoring

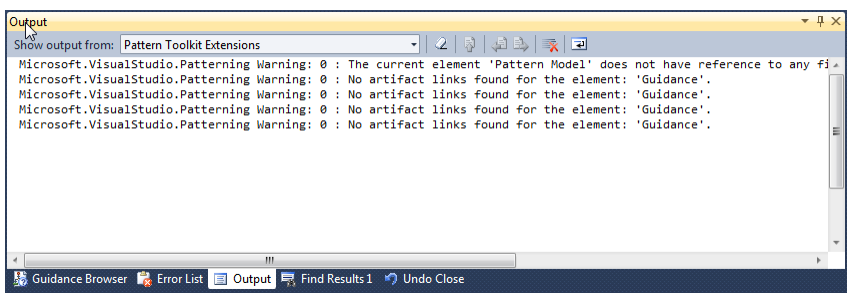
For ‘How To’ style guidance on authoring pattern toolkits, you must have the ‘Pattern Toolkit Builder VS2010’ or ‘Pattern Toolkit Builder VS2012’ (VS012) extension installed, and you must create a new ‘Pattern Toolkit’ project in Visual Studio to get access the guidance.

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.

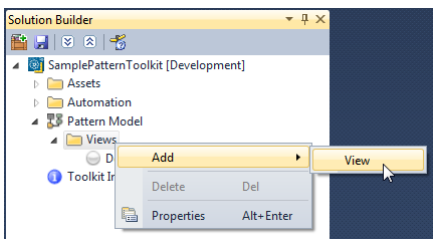
Authoring

Designing and creating patterns, Pattern Toolkits, assets, automation.

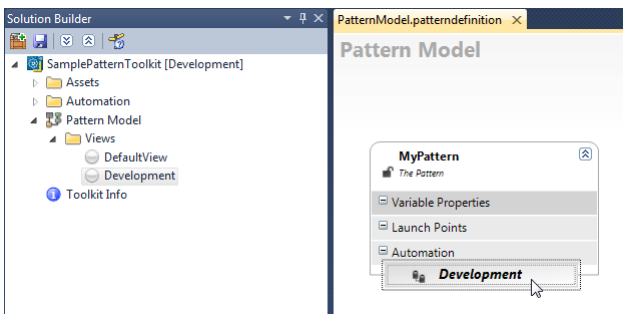
Modeling

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'

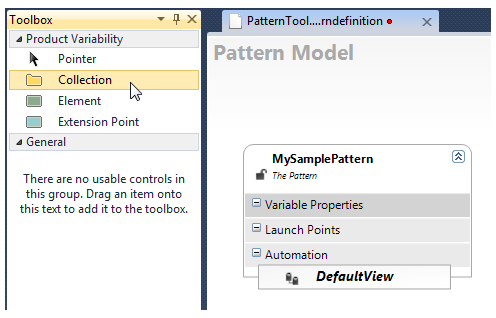


Name the view and it will be created, and double-click on the new view to open it in the pattern model.



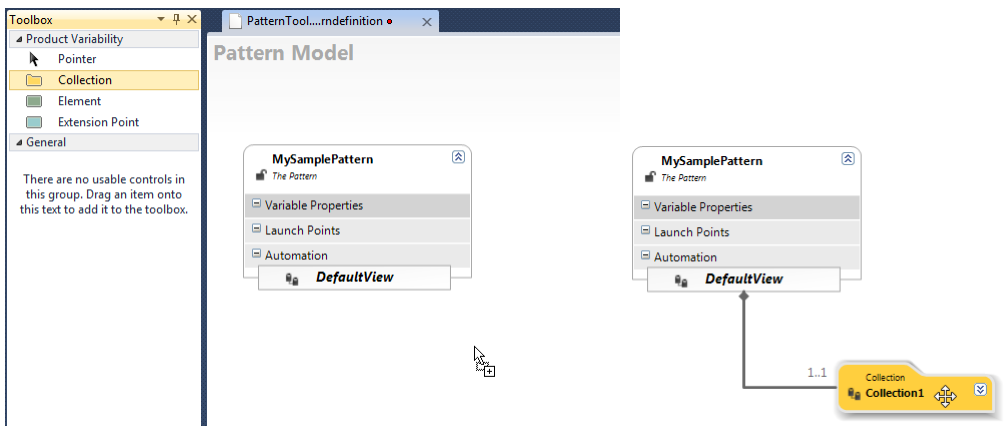
Adding a Collection or an Element

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.

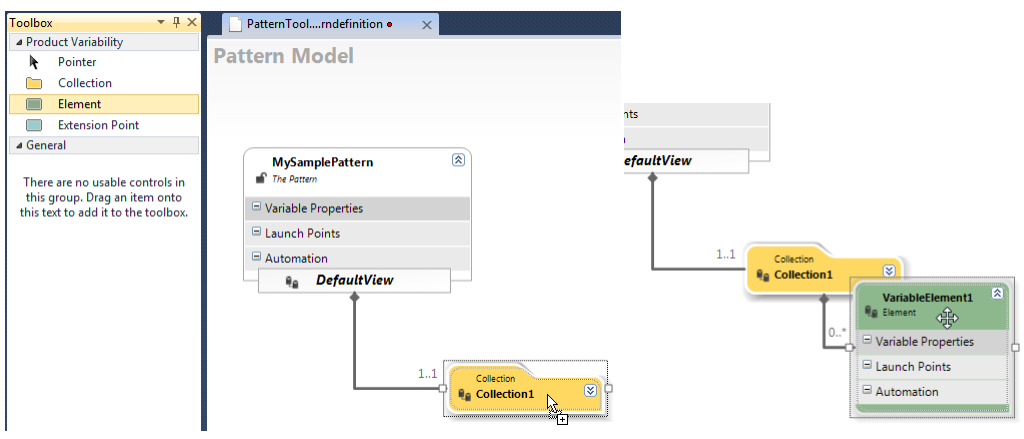


IMPORTANT: It matters where you drop the shape!

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'.



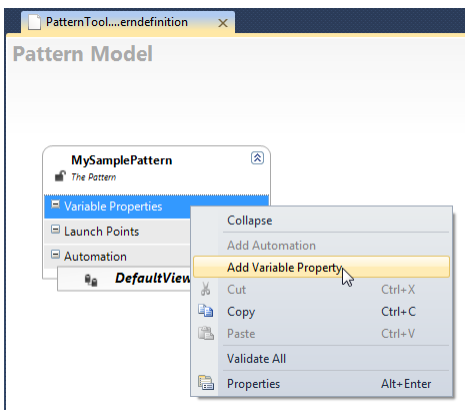
If you drag and drop the shape onto another shape, then the 'Collection' or 'Element' becomes a child of that shape.



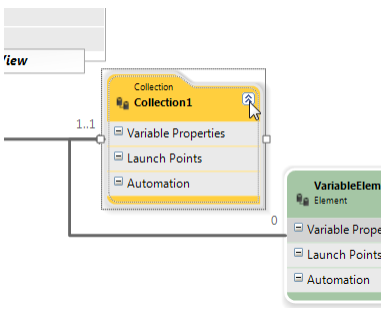
Adding a Variable Property

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.

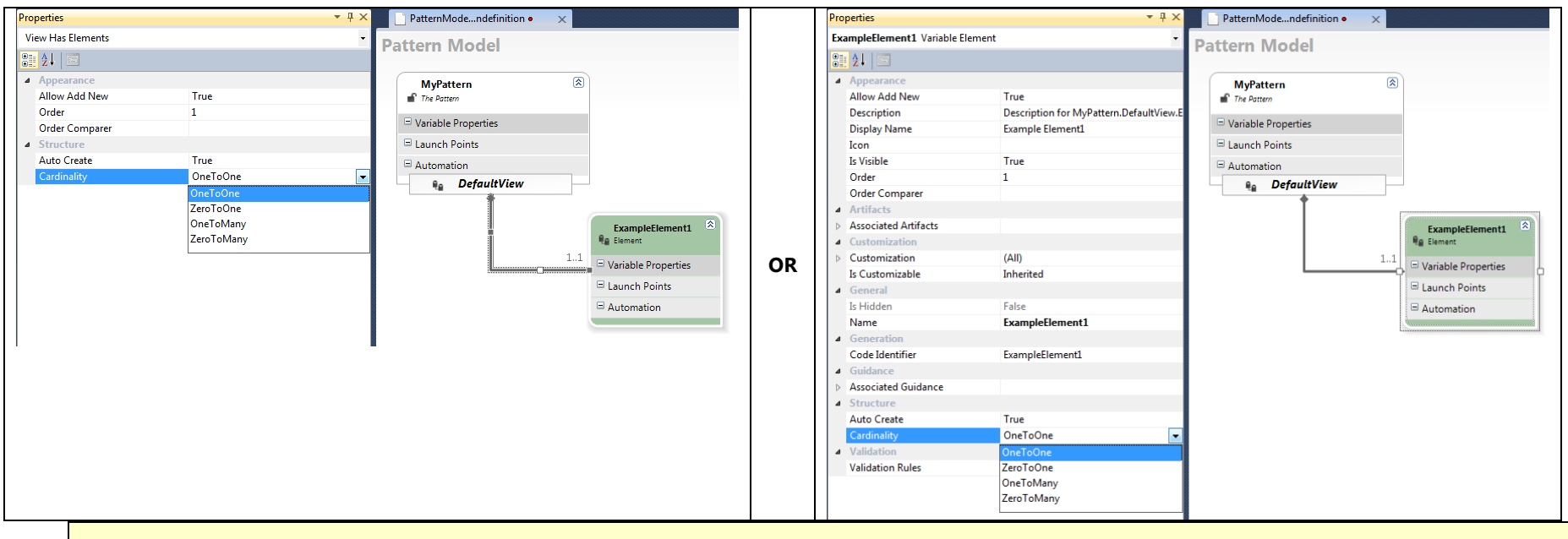


> **Note**: For 'Collections', you need to expand the shape to see the 'Variable Properties' compartment.



Changing the Cardinality

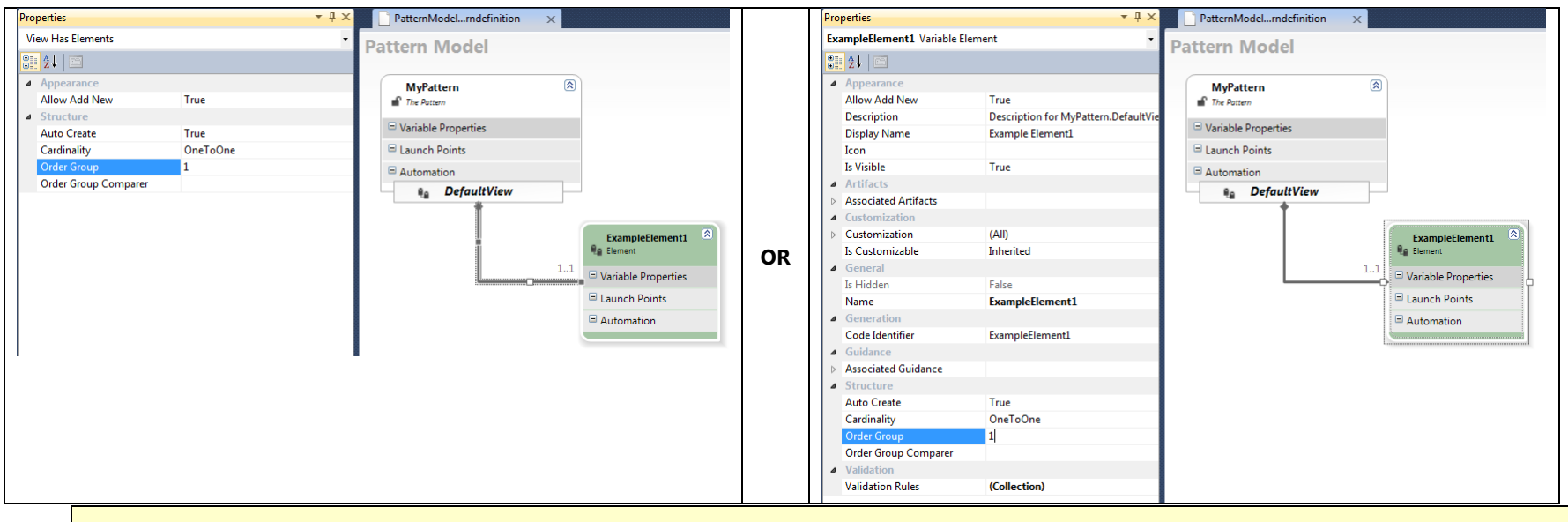
You change the cardinality of the relationship between the 'View', and its child 'Collection' or 'Element' or 'Extension Point' shapes, or between 'Collection', 'Element' and 'Extension Point' shapes, by changing the 'Cardinality' property of either the connector between the shapes, or on the properties 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'.

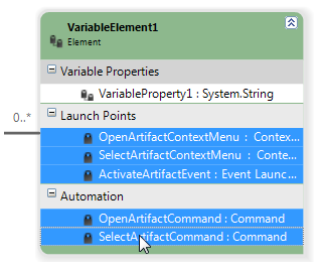
Showing 'Hidden' Elements

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.

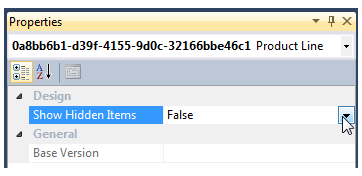
For example, when you configure the 'Associated Artifacts' or 'Associated Guidance' properties of a pattern element, several Command and Launch Points are automatically configured on the current element, which are normally hidden from view.



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.



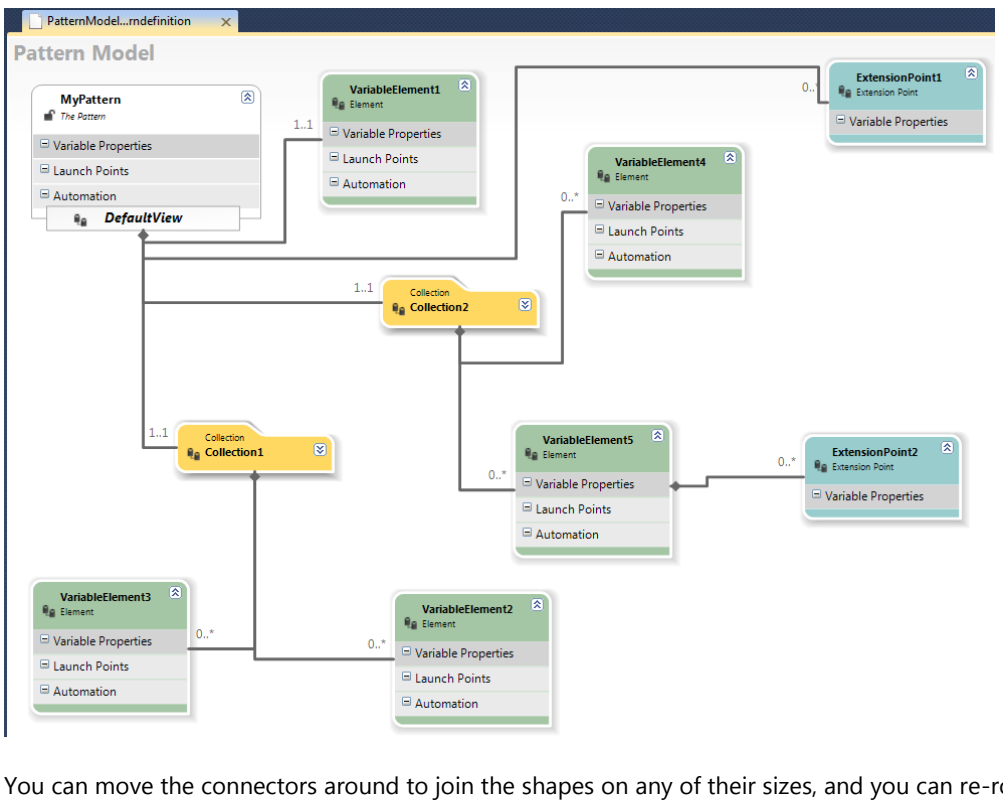
> **Note**: Showing hidden items usually expands the shapes of the diagram interfering with the current layout of the pattern model. Hide these items to restore the original layout.

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.

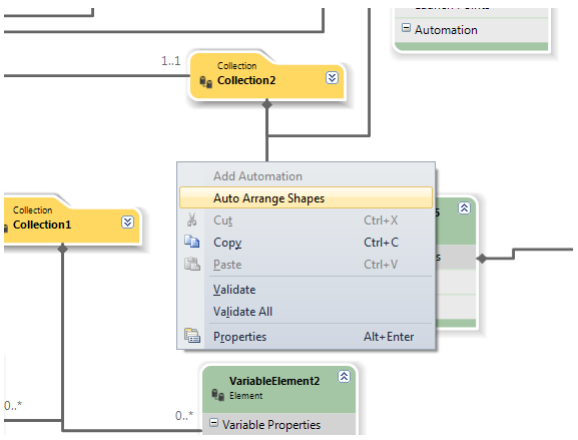
For example:



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.

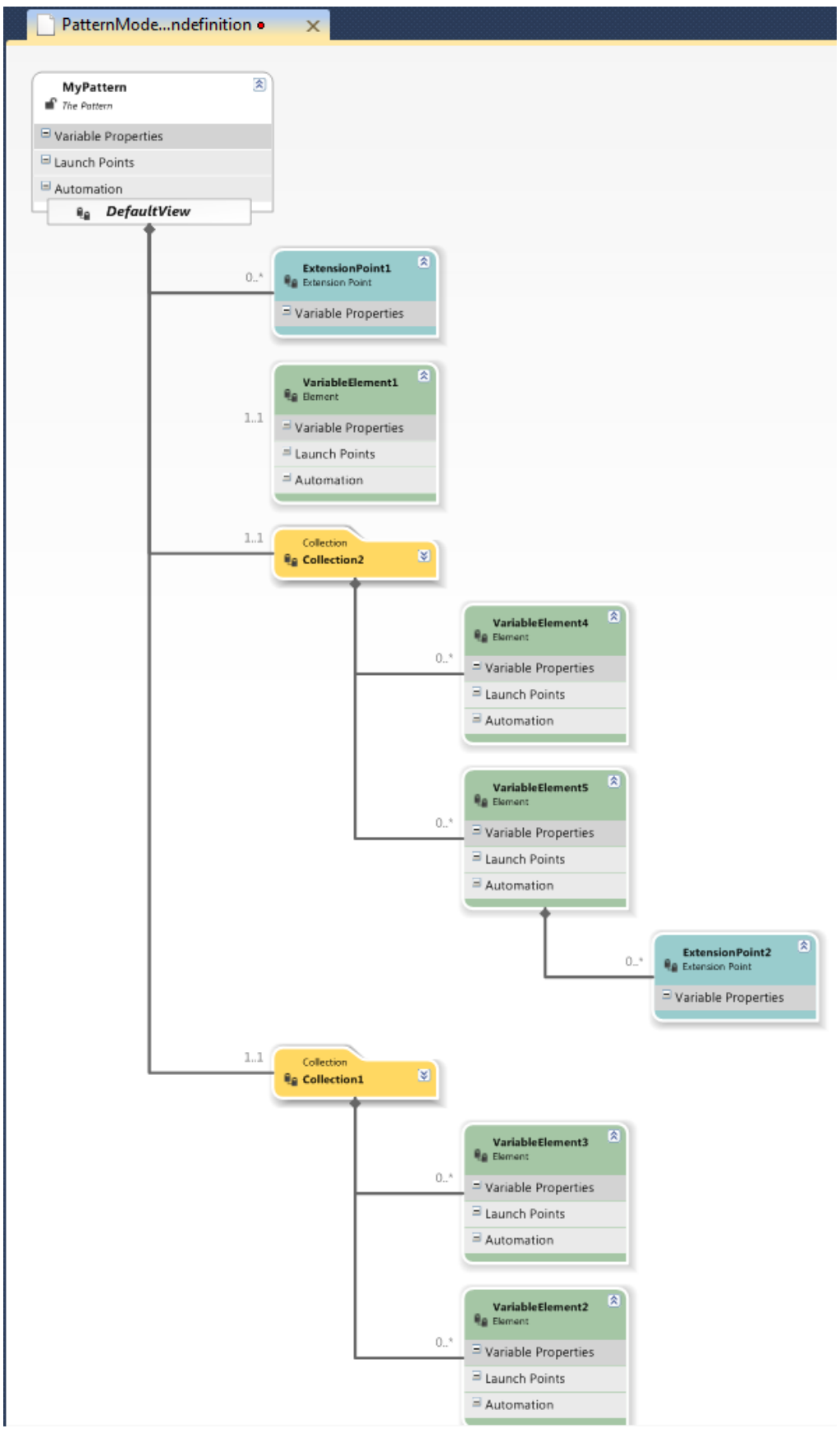
Hierarchical Layout

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:

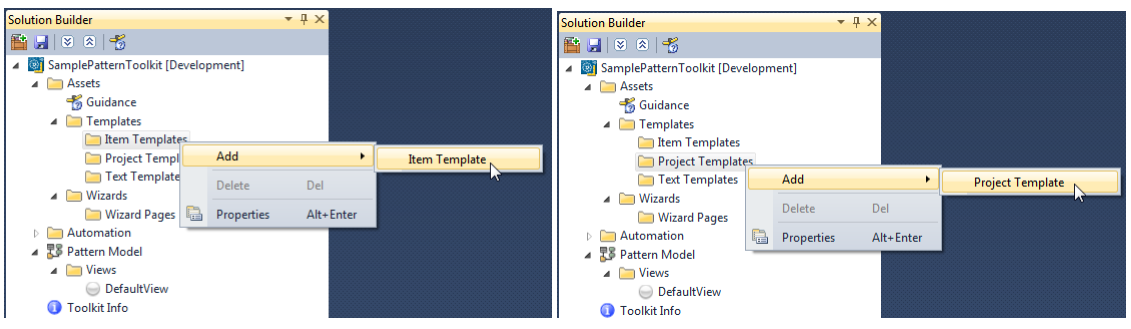


> **Note**: If you wish to modify the ordering of the shapes in the hierarchy, simply drag them higher up the page than their sibling shapes, and use the 'Auto Arrange Shapes' menu again.
> **Note**: If you wish to undo any auto-arrangement, simply use the 'Undo' menu in Visual Studio.

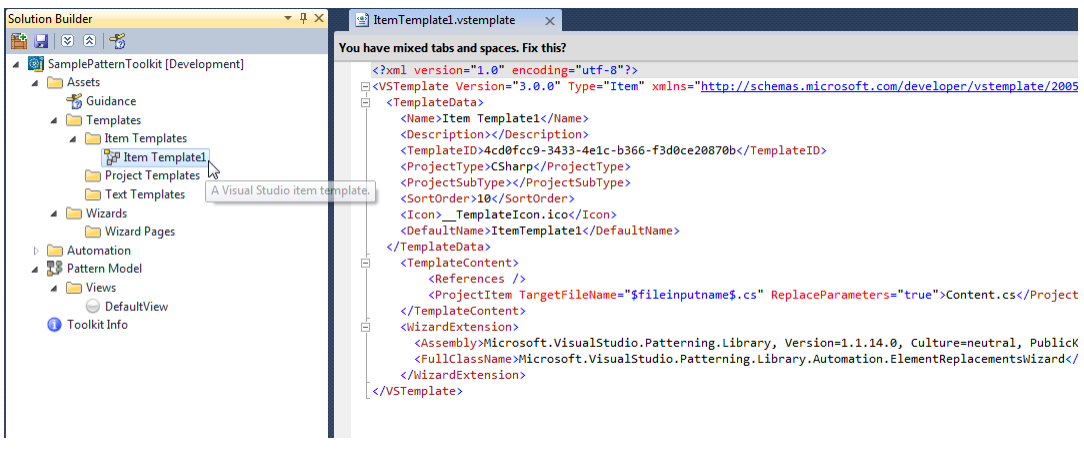
Assets

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.

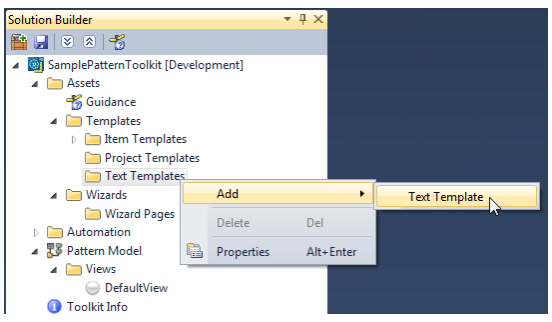


Then double-click on the newly created element, and the *.vstemplate file will open for editing.

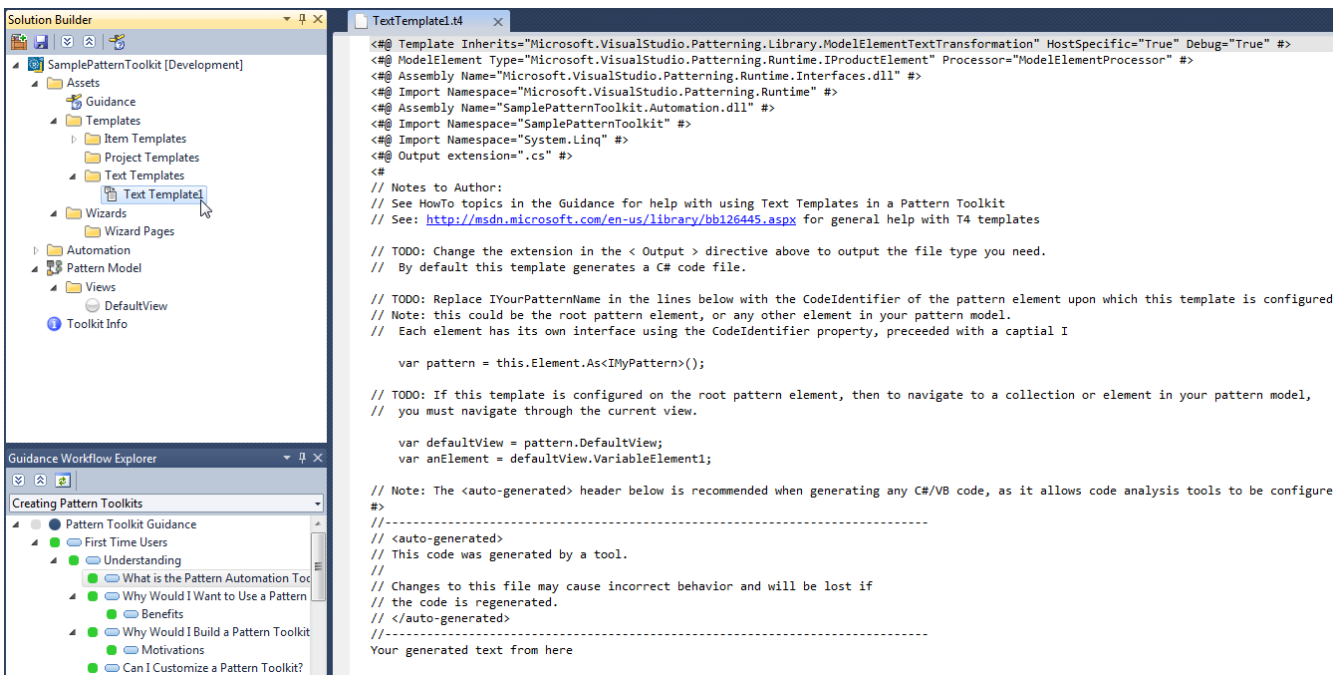


Adding a Text Template

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.



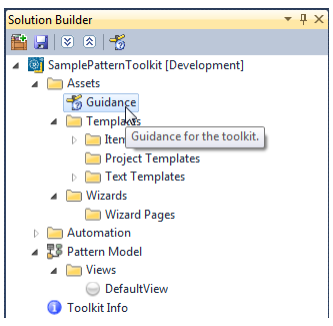
The *.t4 file will open for editing.



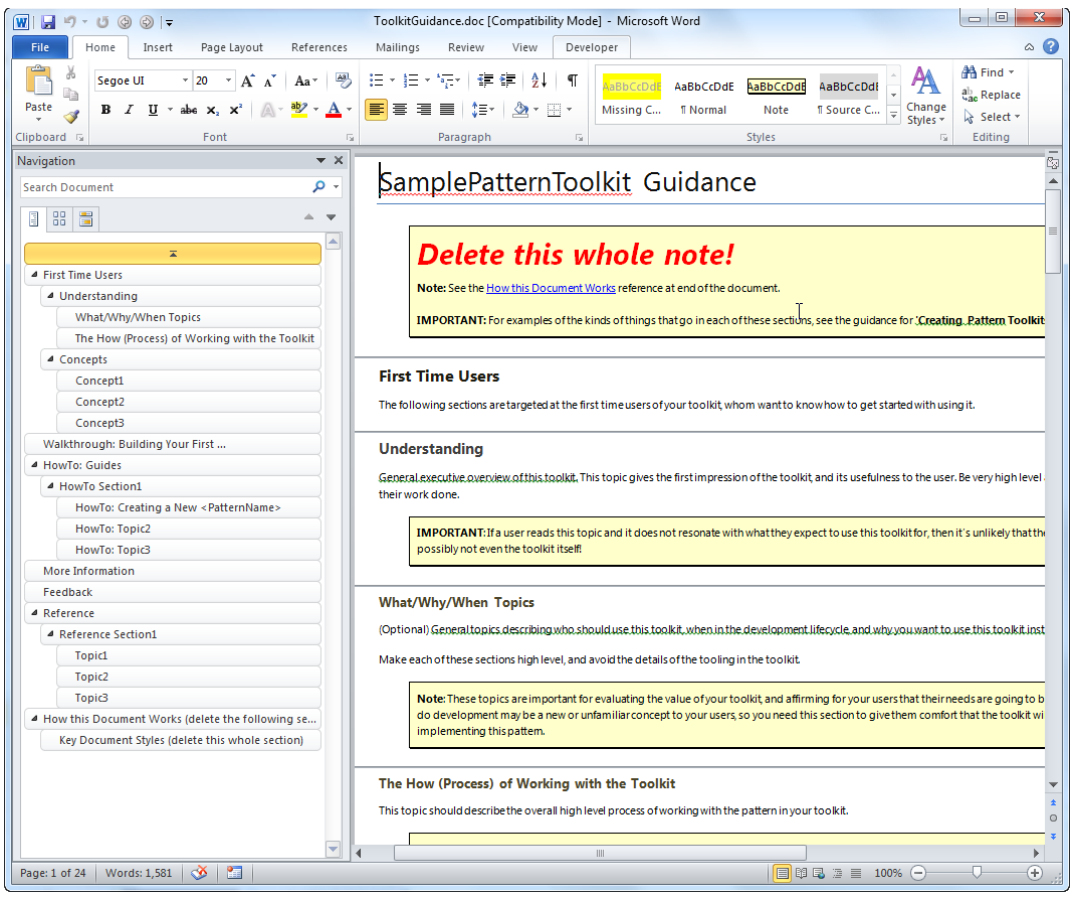
Guidance

Editing Guidance

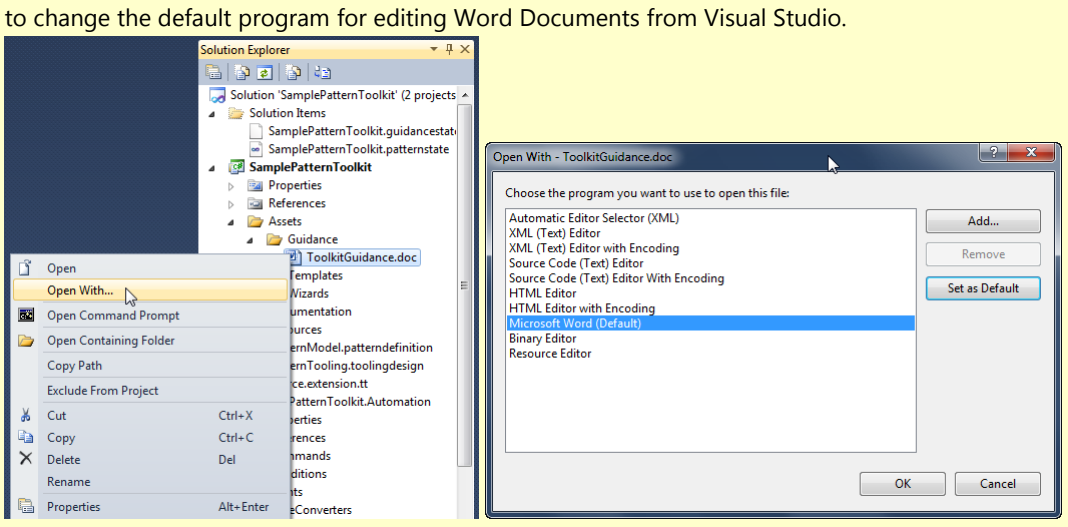
To edit the guidance provided for each toolkit project, in 'Solution Builder', double-click on the 'Guidance' element.



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.

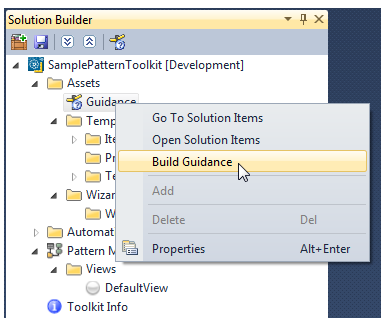
Once the guidance has been created, close and save the document.

Then proceed to Building Guidance.

Building Guidance

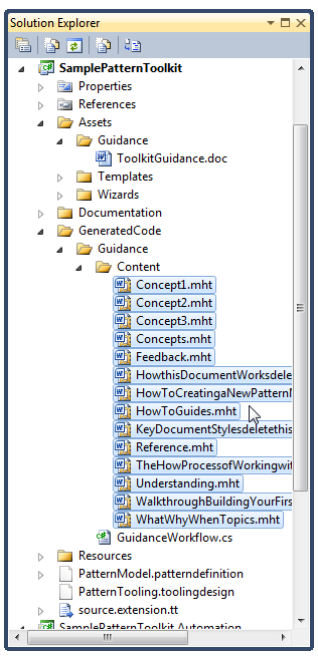
> **Note**: To build the guidance document, it must be saved and closed in Microsoft Word.

To build the guidance document provided for each toolkit project, in 'Solution Builder', right-click on the 'Guidance' element, and select 'Build Guidance'.

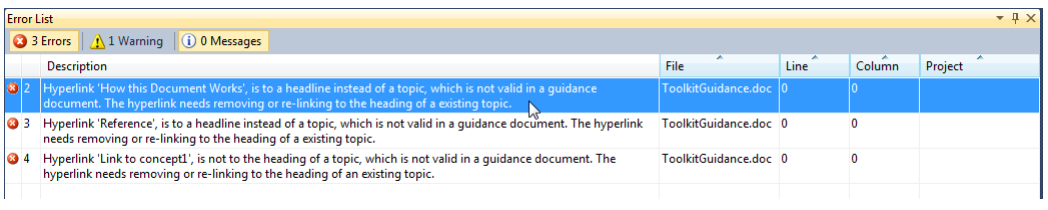


> **Note:** The building of guidance may take quite some time, especially if there are many guidance topics to generate.

If the guidance document is correctly formatted, the guidance document is shredded into many other documents generated into the toolkit project.



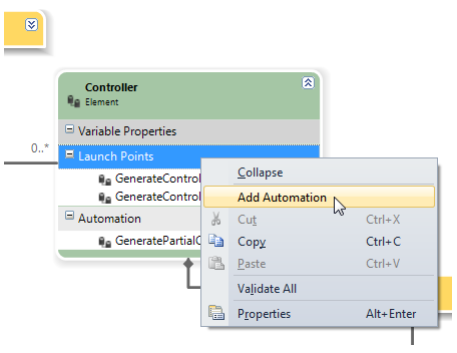
If the guidance document is not correctly formatted, errors will be displayed in the 'Error List'. These errors will need to be resolved before the guidance document can be built.



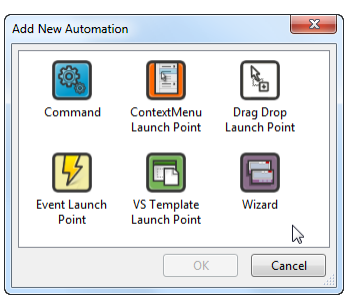
Automation

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.

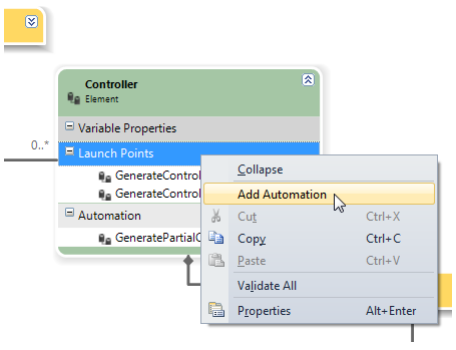
Tip: The 'Status Text' property value is displayed in the status bar of Visual Studio when data is being dragged over the current element. This text value can be either static (i.e. defined as a fixed string), or a Value Provider can return the text dynamically.

3. Optionally, you can configure a custom 'Wizard' to be displayed before the 'Drop Command' command is executed.

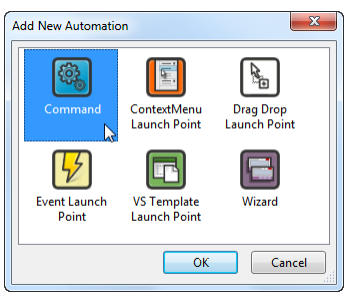
Adding a Command

> **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'.

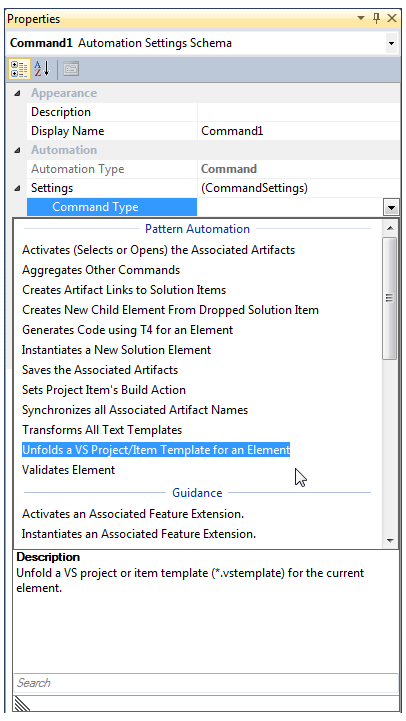


This displays the 'Add New Automation' dialog, where you select the 'Command' shape.

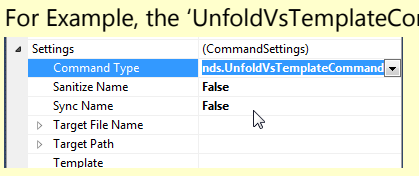


Once added, simply type the name of the command.

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.

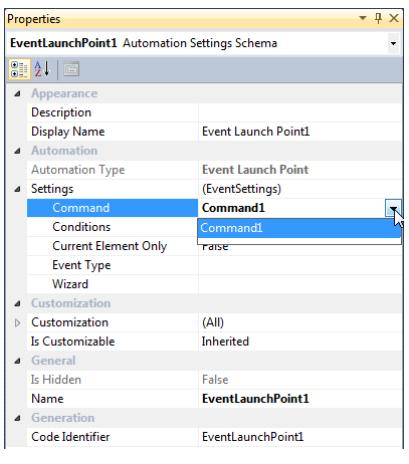
Configuring a Command

> **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

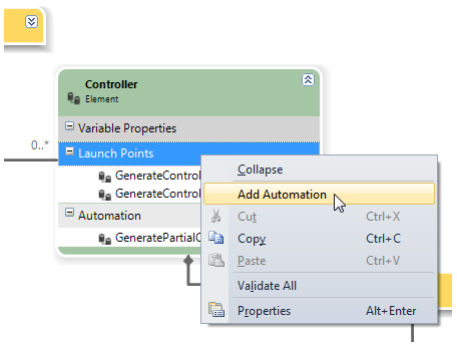
In the 'Settings' property of the launch point, select the 'Command' from the list of commands already added to this element.



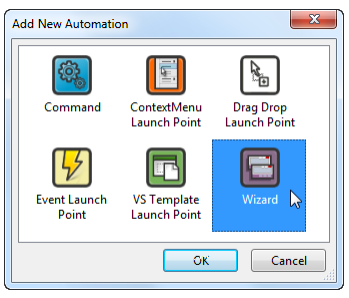
Adding a Wizard

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'.

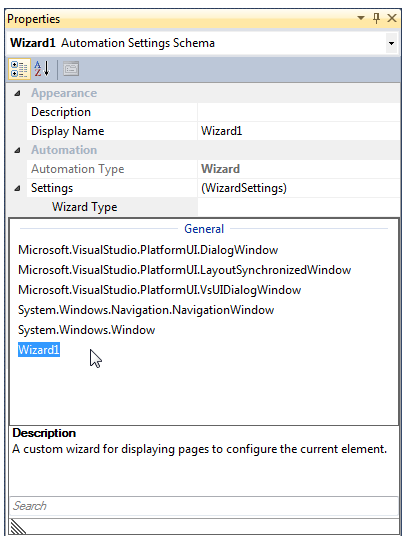


This displays the 'Add New Automation' dialog, where you select the 'Wizard' shape.



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.

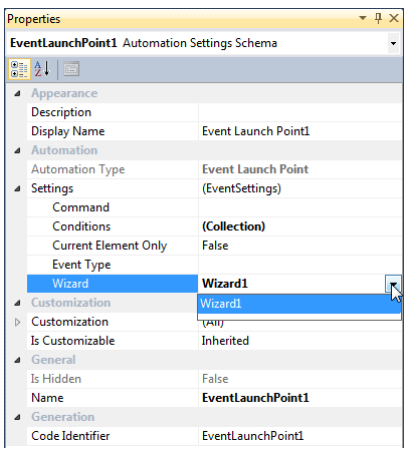
Configuring a Wizard

> **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.

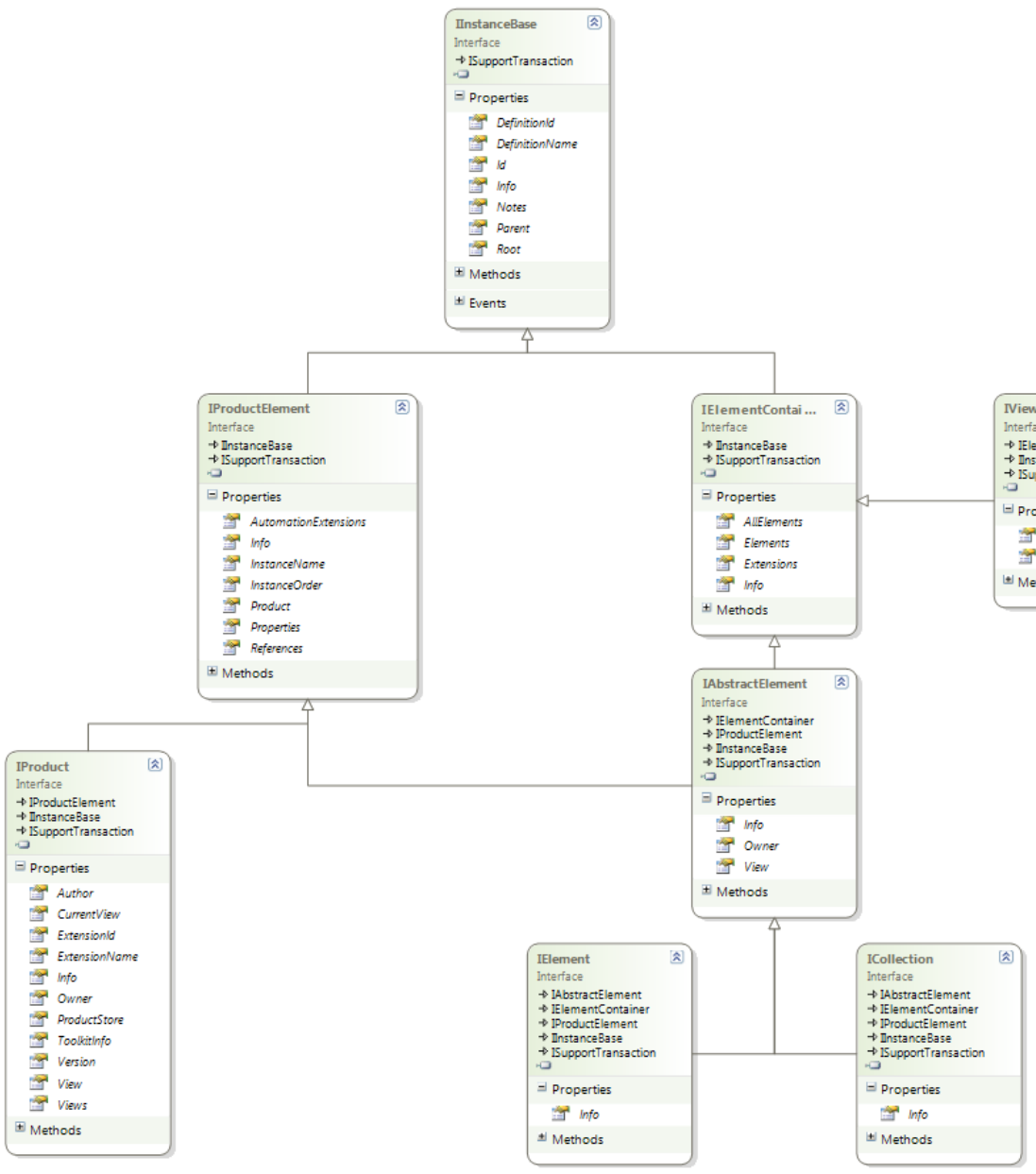
To access the schema information of any element, you use the .Info property.

```
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).

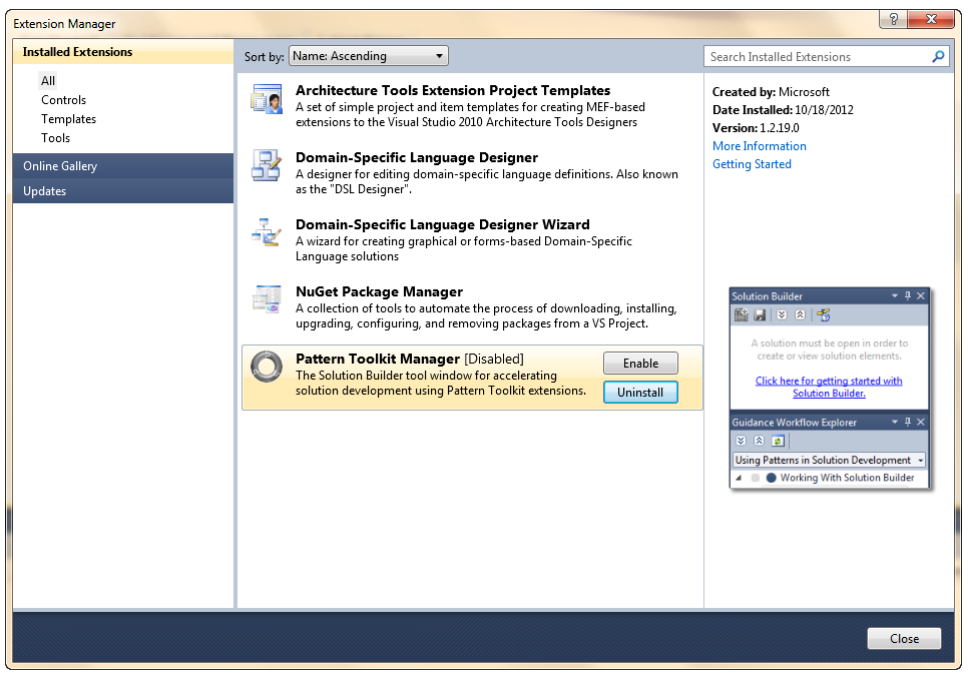
Resetting Experimental Instance

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.

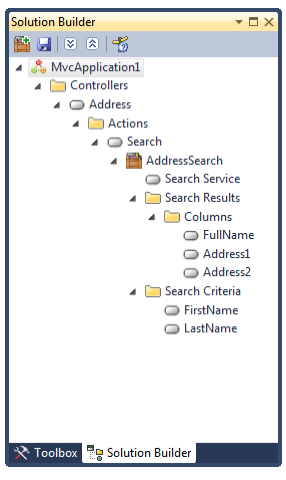


> **Note**: If you see any of your toolkits here that are under development at this point, you should probably uninstall them also.

5. Close the 'Experimental Instance' of Visual Studio.
6. Rebuild your toolkit solutions

Solution Builder

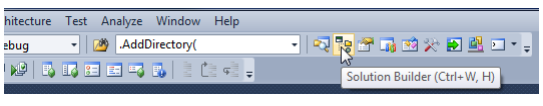
The 'Solution Builder' tool window is a new tool window for working with patterns in your solution.



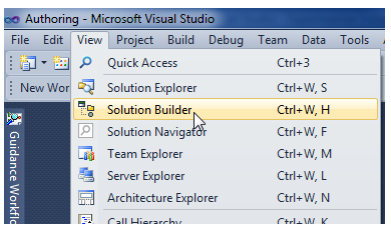
You use this window for creating 'New Solution Elements' that help automate the creation of projects in your solution using installed patterns.

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)



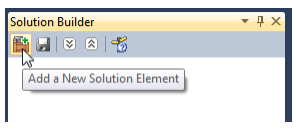
- Press **CTRL + W, H**

> **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.

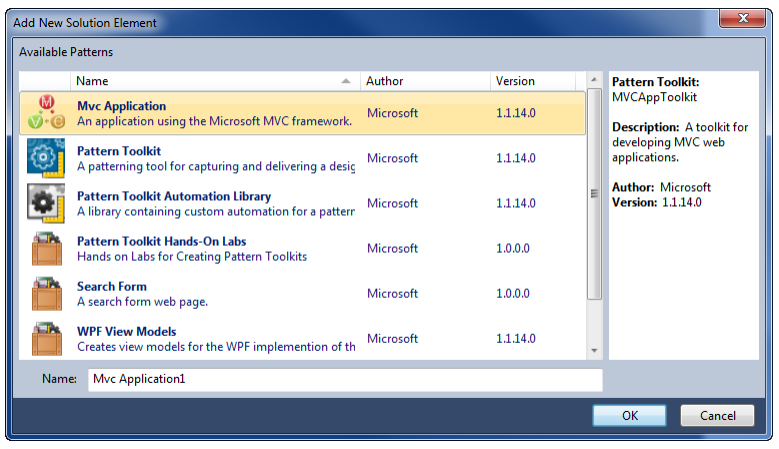
Add New Solution Element Dialog

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.

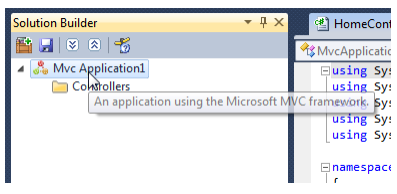


> **Note**: You will need to have an existing or new solution already open to create new solution elements with this dialog.



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.



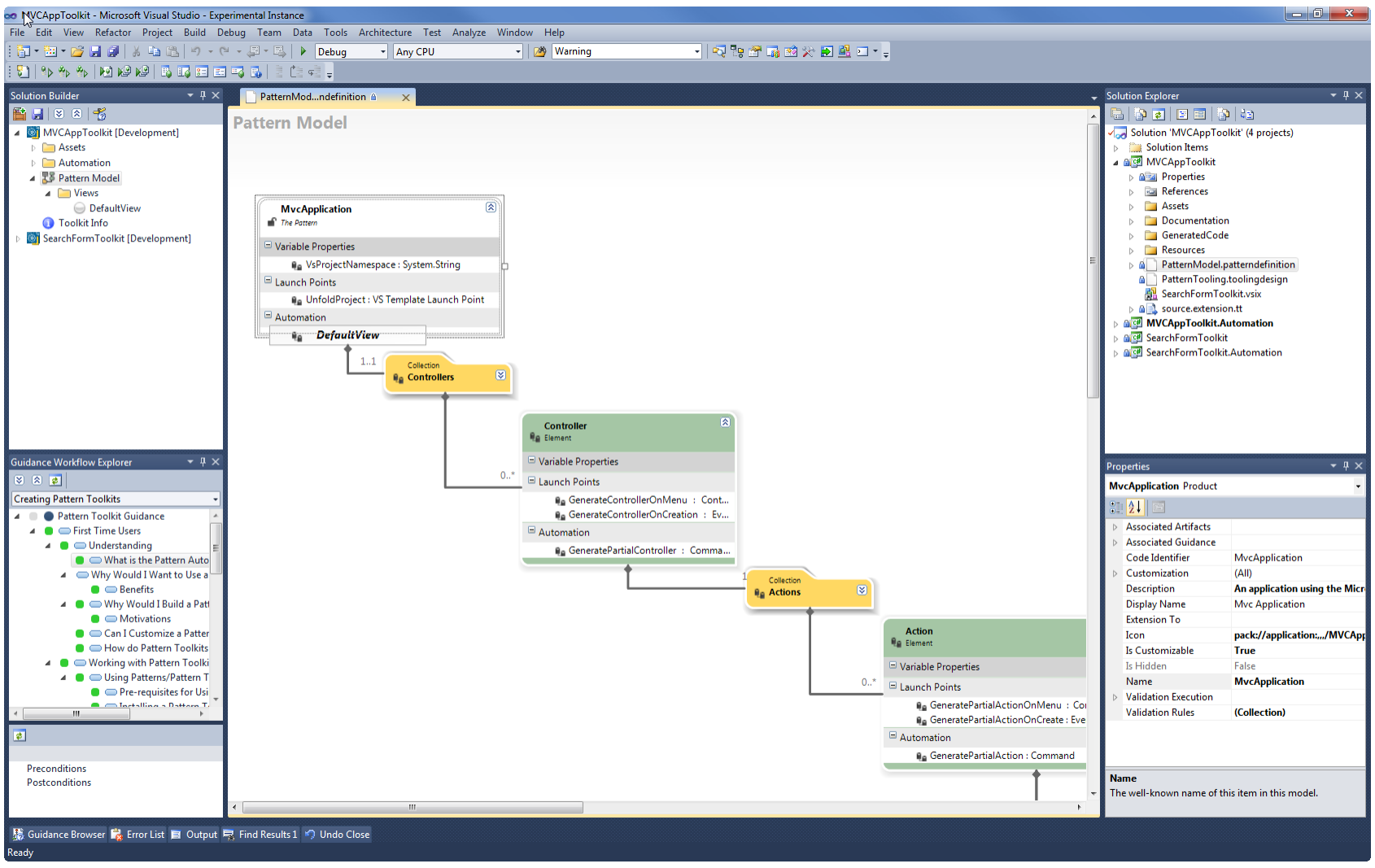
Pattern Model Designer

The 'Pattern Model Designer' is a designer for describing the pattern within a Pattern Toolkit.

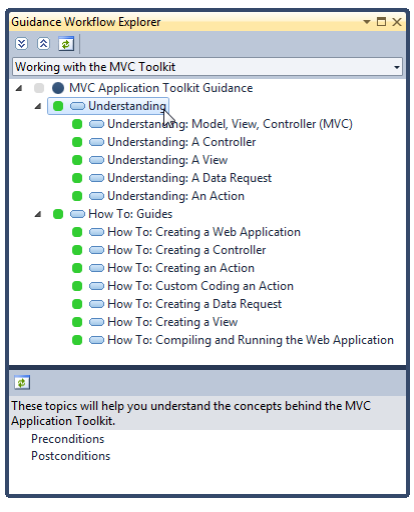
This designer defines the pattern, and the elements which allow a user to configure its variability for their solution.

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

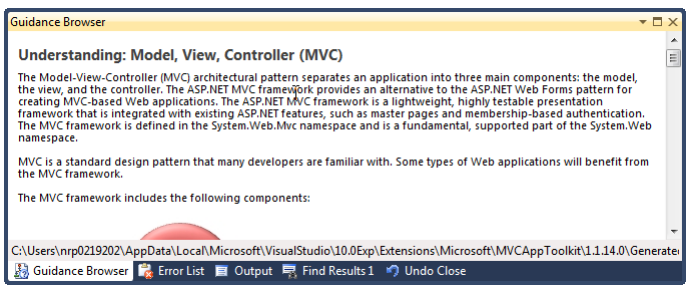


The 'Guidance Workflow Explorer' window (View | Other Windows menu) in Visual Studio displays the available guidance workflows that can be used to work with other tools and processes in Visual Studio.

This window allows you to select a workflow from the drop down list, click on each step in the workflow.

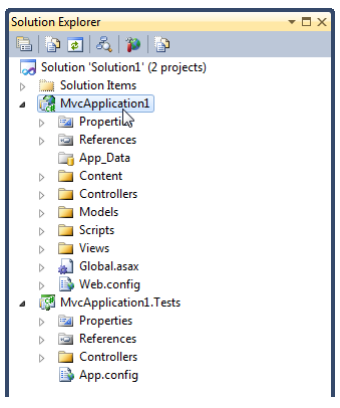
Each step in the guidance workflow, may be either active (green), blocked (red), or disabled (grey) to indicate whether the user can proceed with that step in the guidance. Some steps will include a checkbox that allows the user to mark the step as complete. Other steps may determine if they are complete 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.

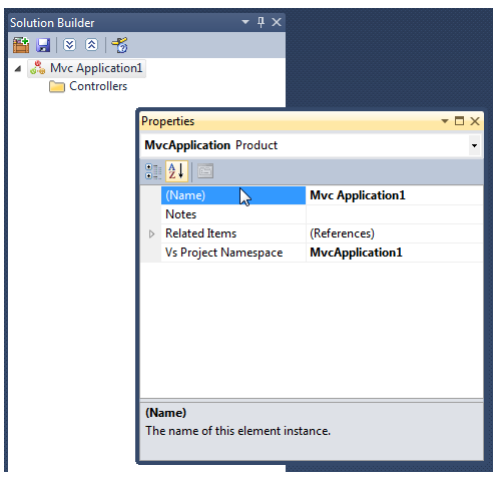
Solution Explorer



'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



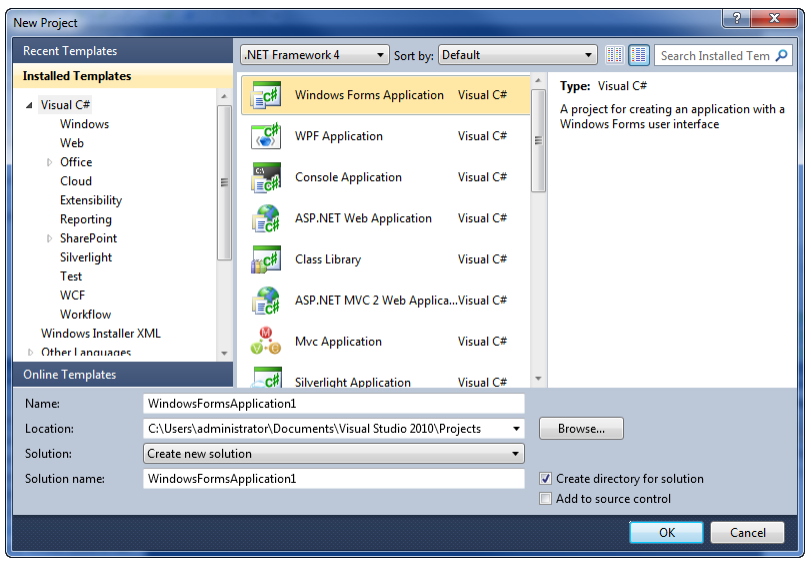
The 'Properties Window' (CTRL + W, P) in a common window in Visual Studio for displaying the properties for the selected object in the currently active window.

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.

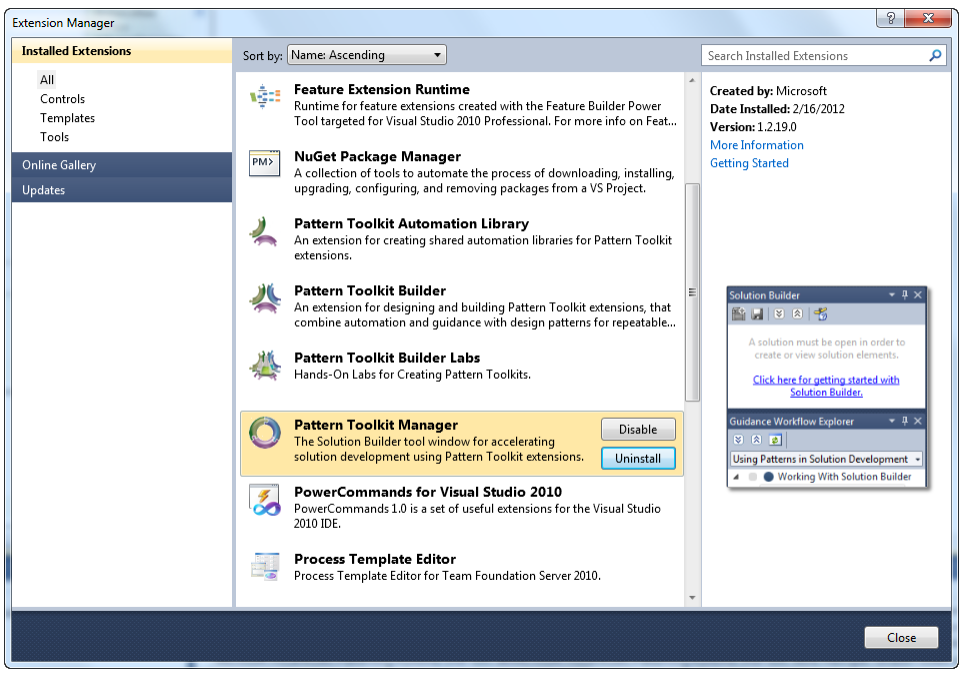
This dialog lists all the project and item templates installed on the current machine.



> **Note**: Item templates are filtered based upon the projects the items will be added to.

Extension Manager

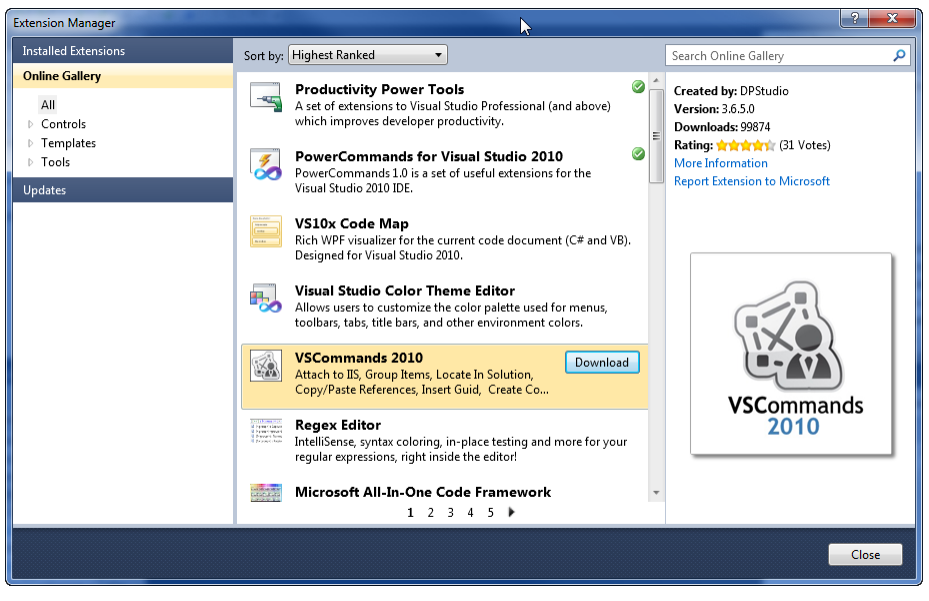
The Visual Studio 'Extension Manager' (Tools | Extension Manager menu) is where you view and manage the installed extensions to Visual Studio.



In the dialog above you can see the currently installed extensions.

> **Note**: NuPattern extensions are listed as extensions in 'Extension Manager'.

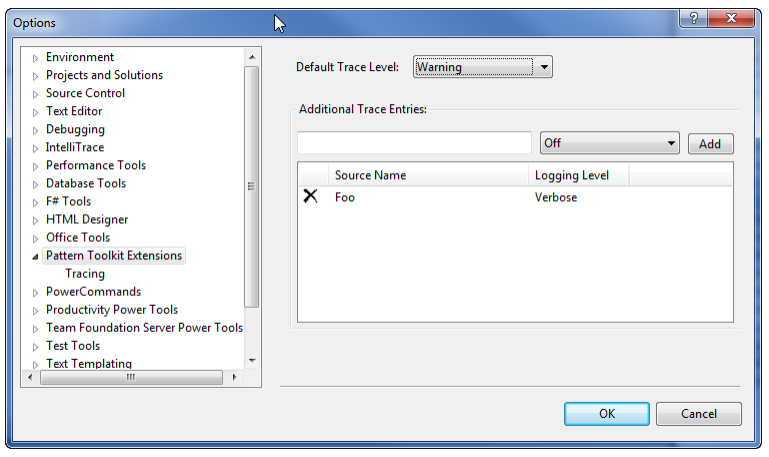
You can also download and install additional extensions from the [Visual Studio Code Gallery] in this dialog.



Options

There are several options that control how patterns toolkits work in the Visual Studio environment.

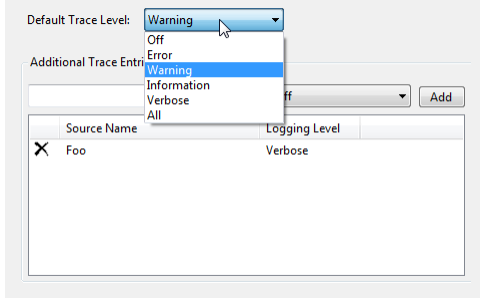
They are modified in the Visual Studio 'Options' dialog, (from the Tools | Options menu)



Tracing Options

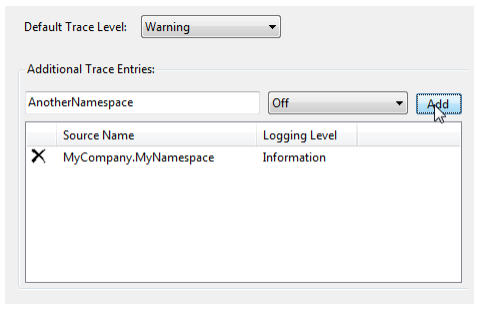
In this page you can control the tracing levels and tracing sources that appear in the Tracing Window.

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.



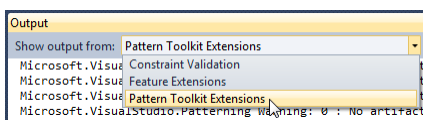
> **Note**: The format of trace sources is typically a name of an application or namespace of the types that produce trace output.
> See Introduction to Instrumentation and Tracing in the .NET Framework.

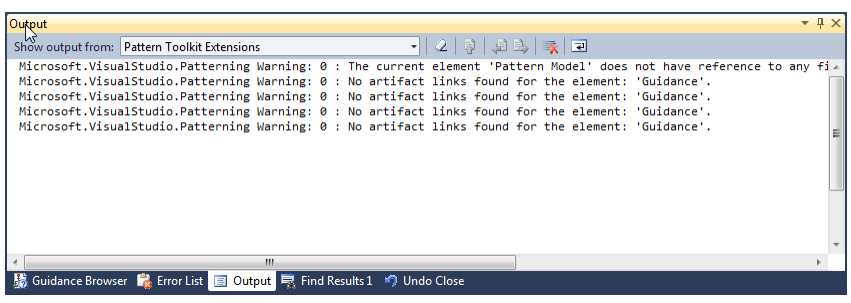
Tracing Window

The Tracing Window is where status and troubleshooting information is displayed for all pattern toolkits.

Opening the Trace Window

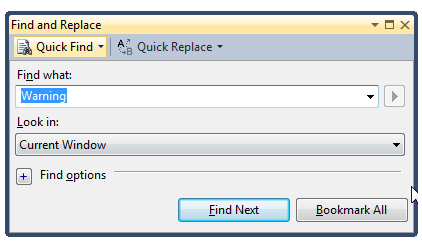
Open the 'Output Window' (CTRL + W, O) in Visual Studio, and select the 'NuPattern Toolkit Extensions' pane from the drop down list at the top.





Working with the Trace Information

The trace information can be searched using 'Find' (CTRL + F), for searching for specific text.



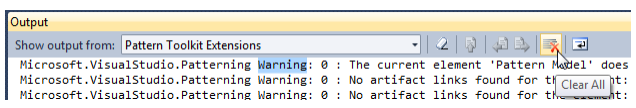
Select the 'Current Window' option to search this information.

> **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

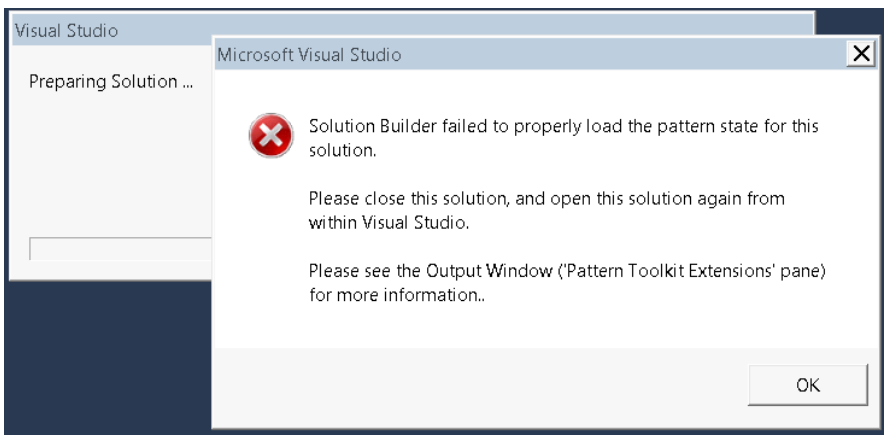
Known Issues

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).

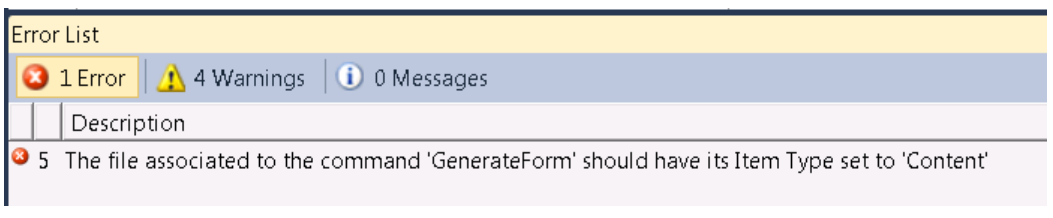
Corruption of pattern model when performing an Undo

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.

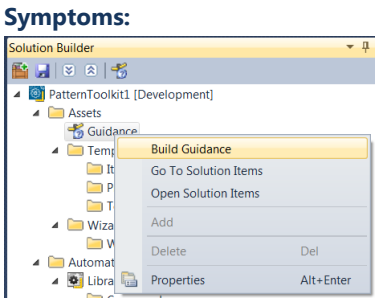
Cause:
The version of the Visual Studio 2010 SDK is not compatible with the version of Visual Studio 2010.

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.

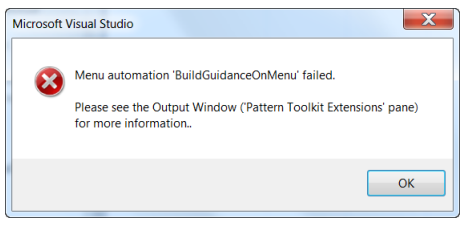
Solution:
You must uninstall the current version of the VSSDK installed on your machine, then install Service Pack 1 of the Visual Studio 2010 SDK.

Error, building guidance with 64bit versions of Microsoft Word.

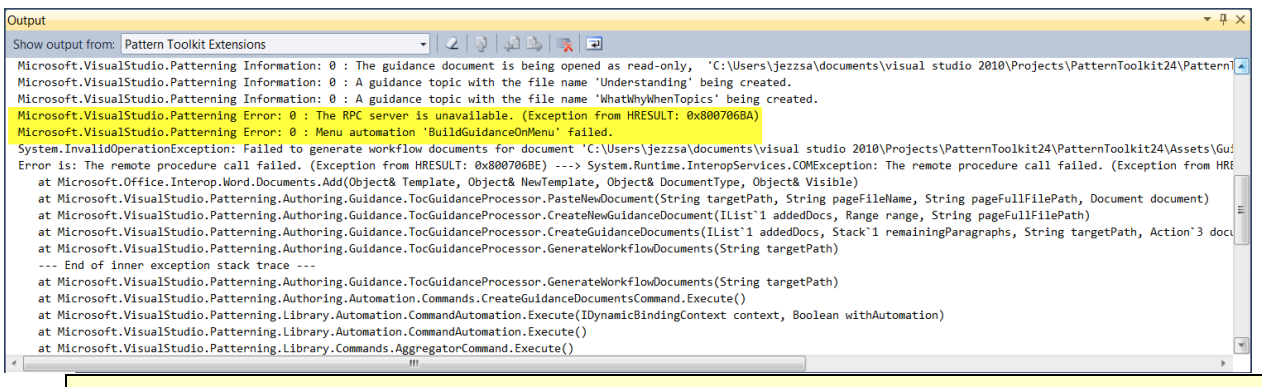
Symptoms:



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.

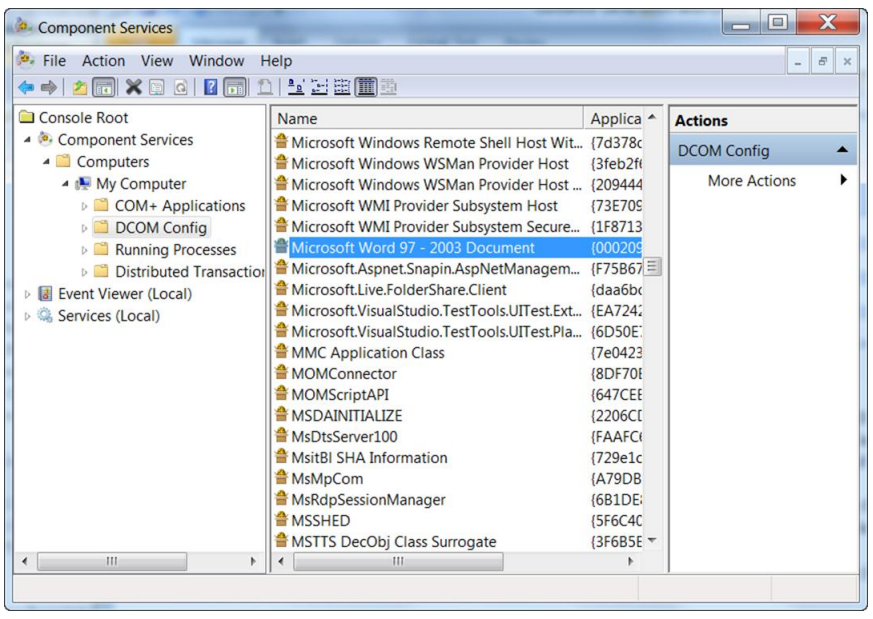
> **Note**: The 64 bit version of Office has been confirmed to experience this issue, whereas the 32 bit version of Office may not.

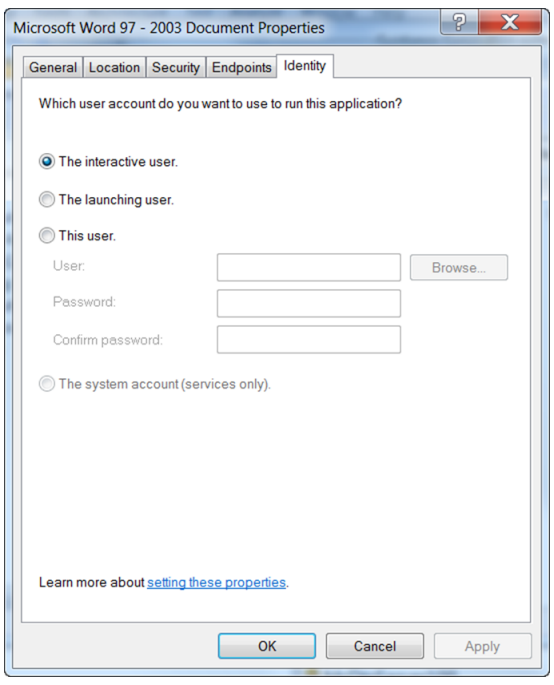
Solution:

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.

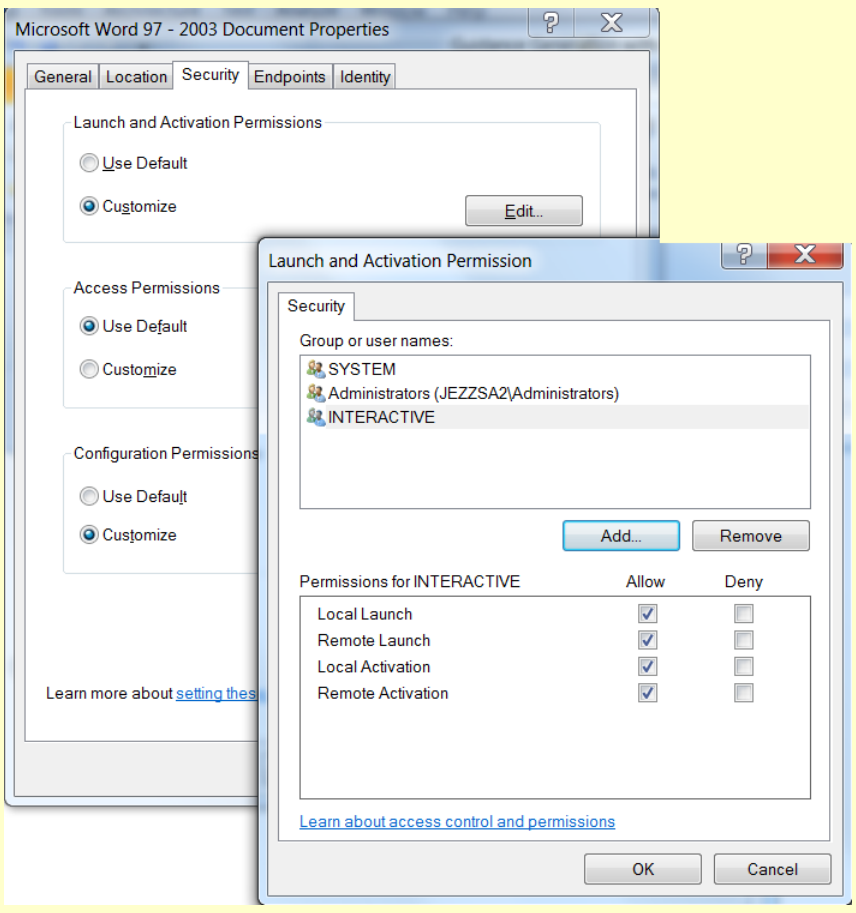
In the properties dialog, select the "Identity" tab.

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