Atomineer Pro Documentation

User Guide

www.AtomineerUtils.com

January 2024

Contents

Introduction

Atomineer Pro Documentation is an extension for Visual Studio that aims to save a lot of time and effort when creating and updating documentation comments in source code. As well as providing commands to create and update documentation comments (for single code elements, and across entire solutions), it includes a documentation viewer, live typing aids to improve editing in comments, the ability to word-wrap text in any comment block or align code structures into columns, and some helpers for creating common code structures.

Atomineer supports a wide range of popular languages as well as many documentation comment formats and styles. If the default Atomineer settings are not exactly as you require, it can be easily configured to suit your needs, and once set up you are unlikely to need to configure it again.

In addition to the more commonly used preference settings, Atomineer provides a wealth of advanced options to help you fine-tune Atomineer to your liking – if you would like any assistance in adjusting these options just email us at support@atomineerutils.com. We endeavour to reply within 24 hours and do all we can to help you.

Important Background Information

When documenting, Atomineer uses two mechanisms to find code elements to document, and to understand those elements.

- Atomineer is able to parse the text of your source code.
- The built-in "intellisense" (code model) information provided by Visual Studio.

These two approaches each have their own advantages and disadvantages, and this should be borne in mind when using some Atomineer commands (each command documented below includes details on how to get the best out if it)

Atomineer Parser	Visual Studio Intellisense System
The built-in parser reads the text around the cursor position and interprets it to decide how to document it. **Pros** • Very tolerant of code errors – even if the source code is incomplete or uncompilable, Atomineer can often understand it and document it correctly. **Cons** • Atomineer has no knowledge of your code beyond the few lines it is looking at, so does not know details about user types, macros, etc.	Visual Studio's built-in intellisense system uses information derived from the last build of your code (C++) or is updated live as you type (VB, C#) **Pros** • Provides comprehensive information on code elements. **Cons** • Only available when the code compiles/has been compiled. As a result, it is often unavailable while typing new code.

• To be tolerant, the parser makes educated guesses, so can become confused and produce unexpected results *(if you find a case the parser fails on, please send it to us and we'll do our best to fix it – we can often return a fix to you within 24-48 hours)*	• Users can disable intellisense, making it unavailable. • Intellisense for C++ headers directs Atomineer to the implementations in .cpp files, and often takes a long time to fetch, so intellisense cannot be used in C++ headers. • Intellisense can be unreliable, particularly in older versions of Visual Studio. This problem is much less common from VS2013 onwards. • In Visual Studio 2015, some intellisense became unavailable due to the new Roslyn architecture; this functionality should be restored over time as Roslyn matures.

Programming Language Support

Due to varying levels of Intellisense support for languages, Atomineer's range of capabilities differs slightly from one language to another, as follows:

Language	Level of Support
C# **Visual Basic**	Fully supported. Intellisense for C# and VB is usually very reliable.
C++ **C**	C++ and C are less well supported, with the following caveats: • Intellisense is not available in header files, so batch processing is not available within headers. You can still use **Document All in this Scope** or **Add/Update Doc Comment** however. • Sometimes Intellisense can become slow or cause Visual Studio to lock up or crash. If this occurs, a full clean and rebuild of the solution often restores the performance to normal levels.
Java **JavaScript** **Typescript** **JSharp** **ActionScript** **ECMAScript** **PHP** **Python** **Unrealscript**	Intellisense is unavailable for these languages, so all operations will fall back to using Atomineer's built-in parser, and some of the more advanced features (e.g. duplication of documentation) will be disabled.
Blazor **HTML, cshtml (embedded JavaScript or VBScript)**	These files contain a mixture of two or more languages. Atomineer assumes that they contain C# or VB, so **Add/Update Doc Comment** will usually function well within

	the code sections, but at this time Atomineer does not support documenting within the HTML portions of these files. Blazor (.razor files) is supported to a limited degree by the batch processing commands, as long as the files contain only a single @code section. Header comments will not be added in razor files.
XML **XAML** **HTML** **SQL** **Plain Text**	Atomineer currently offers only minimal support for these languages. It is however able to insert simple boilerplate comments, can align code into columns, and in some cases can word wrap comments, just as it does for other languages.

It should also be noted that Intellisense can be turned off in Visual Studio. If this is done, Atomineer will fall back to its built-in parser, so will not perform to its full capability.

Atomineer Commands

Atomineer provides the following commands:

Command	Description
Add Doc Comment	Creates or updates the documentation comment for the code at the cursor position.
Document All In This Scope	Documents all code elements within a given code scope (e.g. file, namespace, class) using the **Atomineer Parser.** This is ideal for use when you want to document a small portion of a large file, or encounter problems with the other Doc All… commands (usually when Intellisense is unavailable).
Document All In This File **Document All In This Project** **Document All In This Solution**	These commands are shortcuts for common operations. They all work the same way, using the **Visual Studio Intellisense** system to find the code elements to process in a single file, all files in a project, or all files in a solution.
Process All in Chosen Files	The underlying system used by the Document All… commands. This command shows a user interface that allows you to select what you wish to process, as well as additional options such as filtering files based on filenames, etc. This command can be used to create, update, or delete documentation throughout a selection of files, so it is very powerful and flexible.

Hide Documentation Comments using Outlining	Uses Visual Studio's Outlining mechanism to "hide" documentation comments in a file.
Delete Documentation in File	This deletes all documentation comments in the current file, which can be useful when rewriting legacy documentation to be sure that none of the old documentation is accidentally re-used.
Align Code into Columns	In some cases (generally blocks of code where many lines follow a similar pattern, such as a list of assignments, a table of values, repeated calls to the same method with different parameters, or a multi-line if condition), code readability and editability can be greatly improved by lining up the interesting components of each line (such as parameters) into columns. This command uses the context near the cursor to intelligently decide what you would like to line up into a column and how, so it can be used for a wide variety of alignment situations.
Create a Managed Property or A C++ Accessor	Instantly convert a member variable declaration into an auto-property or accessor, or convert an auto-property into a property with a backing field.
Copy As Text	Copies source code to the clipboard *without* any font/formatting information or unnecessary indentation – ideal for copying code snippets into documentation in other programs such as Microsoft Word.

Deprecated Atomineer Commands

Atomineer provides the following commands for Visual Studio versions 2005-2013. These commands are no longer available from Visual Studio 2015 onwards, as their functionality is now available elsewhere, and/or is only rarely used.

Implement/Declare C++ Method	Given a header-file declaration, create a skeleton implementation. Or given a source-file implementation, add a header declaration for it.
Open C++ source/header	Switch easily between C++ source and header files.

Hide Attributes using Outlining	Uses Visual Studio's Outlining mechanism to "hide" attributes in a file.

Other Atomineer Facilities

Atomineer also provides a number of other facilities that can help with documenting your code efficiently. They include:

- **Documentation Viewer.** Documentation comments are often hard to read. The viewer displays the current code element's documentation in a format similar to that on MSDN, or as created by tools like Doxygen or SandCastle, to make them easier to read. It will also expand commands like XML Doc's <include> so you can see the included documentation.

- **Live Typing Aids.** These *optional* aids make it easier to enter documentation within comments:
    - Automatically extend comments when you type Enter. This allows you to continue typing comment text until you are finished. If your documentation block starts with a prefix character on every line (e.g "*") and some indentation, Atomineer will automatically insert this for you on each new line, so you can type without having to worry about the block comment format that contains your description text. Press Enter twice in a row to exit this mode.
    - Automatically continue bullet point lists in comments. If you type a line starting with an obvious bullet point character (-, *, +) or with a numerical id (1, 2, 3, a, b, c), Atomineer can continue the bulleting/numbering on each new line.
    - Improved delete key-press behaviour in comments – if you delete past the start of your description text, Atomineer will automatically append your current line to the line above, stripping out any intervening comment prefixes and indentation – so you can type within a comment block's description "column" as if you were just typing plain text.
    - Tidy up code examples on paste. If you copy code and paste it into a doc comment, Atomineer will integrate it into the comment (e.g. by adding appropriate comment prefix characters and indentation on every line)

Installing Atomineer

Background

Atomineer Pro Documentation is compatible with all versions of Visual Studio from VS 2005 to the current version, as well as Atmel Studio 5.x, 6.x, 7.x. All editions of Visual Studio (e.g. Community, Standard, Professional, Premium, Ultimate, Enterprise, VSTS) are supported *except* for old Express Editions (which did not allow add-ins to be installed).

Atomineer Pro Documentation was originally provided as a Visual Studio "add-in". In more recent versions of Visual Studio, add-ins have been superseded by a new system called "extensions". As a result, you have been provided with **three** Atomineer installers, which have slightly different installation and uninstallation processes. These installers apply to different versions of Visual Studio as follows:

Visual Studio 2005, 2008, 2010, 2012, & 2013, Atmel Studio 5.x & 6.x	**Visual Studio 2015 onwards - and - Atmel Studio 7 onwards**
AtomineerUtilsSetup.exe (add-in) **System requirements** ⓘ A compatible Visual Studio version/edition ⓘ .Net 2.0 ⓘ Windows XP, Windows 7, Windows 8, Windows 10 (32- or 64-bit) ⓘ 1.4 Mb of disk space	Atomineer Pro Documentation.vsix (extension) **System requirements** ⓘ A compatible Visual Studio version/edition ⓘ .Net 4.7.2 ⓘ Windows XP, Windows 7, Windows 8, Windows 10 (32- or 64-bit) ⓘ 1.4 Mb of disk space

If you have several installations of Visual Studio, **both versions** of Atomineer can be installed side-by-side on your PC. They will share preferences and customisations.

Installing

To install either version, quit all running instances of Visual Studio and then run the relevant installer. Installation will proceed as follows:

Visual Studio 2005, 2008, 2010, 2012, 2013 Atmel Studio 5.x & 6.x	**Visual Studio 2015 onwards, Atmel Studio 7 onwards**
Run **AtomineerUtilsSetup.exe** Select an installation location. This must be in a location that can be written to without administrator privileges (e.g. in My Documents). It is recommended that you do not change the default installation location. Click **Install** to install the add-in.	Run **Atomineer Pro Documentation.vsix** (from the appropriate Visual Studio or Atmel folder) Microsoft's VSIX installer will run. Select the edition(s) of Visual Studio or Atmel Studio you wish to install into and click **Install** to install the Extension.

When the installer has finished, Atomineer will be available in all compatible versions of Visual Studio and Atmel Studio.	When the install has completed, Atomineer will be available in all selected editions of Visual Studio or Atmel Studio.

> ⓘ *To upgrade an existing version to a newer version of Atomineer, just run the installer and Atomineer will be upgraded in-place. (There is usually no need to uninstall any version of Atomineer prior to installing another version).*
>
> *The exception to this is that the Free **Trial (extension only)** is seen by Visual Studio as a different product, so it **must be uninstalled** when installing the full version of the extension.*

Quick Setup Wizard

For first-time installs, the **Quick-Setup Wizard** will be shown (for the add-in, this is shown immediately when installation is complete. For the extension, it will be shown the first time you run Visual Studio after installing).

> ⓘ The wizard allows you to choose your preferred documentation style from a set of common defaults.
>
> On each page of the Wizard, just select the style that most closely matches your requirements and click Next.
>
> *Note: You can change all of these settings later or re-run the Wizard from the Atomineer Options. If you skip this step by clicking 'Cancel', Atomineer will just use defaults to start with.*

Preferences and Customisations Location

Once installed, the software will use the following folder(s) to store preferences and customisations:

Visual Studio 2005, 2008, 2010, 2012, 2013 Atmel Studio 5.x & 6.x	**Visual Studio 2015 onwards, Atmel Studio 7 onwards**
Atomineer is installed to the user-supplied location and uses this to store preferences/customisation files. The default location is: My Documents\Visual Studio **20??**\ Addins\AtomineerUtils (where the 20?? indicates the latest version of Visual Studio installed, up to 2013)	The Extension will default to using My Documents\Atomineer However, if the Atomineer add-in is installed, it will instead **share** the add-in's folder.

Using Different Settings for each Solution/Project

Normally, Atomineer will use a single set of preferences to control its functionality, but sometimes it is useful to be able to use different settings within each Visual Studio Solution or Project you are working on (for example, it is quite common in large companies to use a different header layouts and comment styles in different legacy codebases)

Atomineer allows you to set up different preference storage locations for each solution/project as follows:

- In the Atomineer options, switch to the **General Settings** tab
- In the **Preference Storage** section, enter the path(s) of your standardised folder locations, using %solutionPath% or %projectPath% to represent the location of the project/solution file, e.g.

```
%solutionPath%\Resources\AtomineerSettings
```

(For the leaf-name rather than the full path, just use %solutionName% or %projectName%. You can insert the value of any environment variable in the same way, e.g. `%USERPROFILE%`. Note that Atomineer's special variables will be used in preference to any environment variables of the same name)

From this point on, Atomineer will search in your solution's Resources\AtomineerSettings folder for its preferences/customisations. If they are not found there, then it will continue to search in its default location.

To use this in a solution, just set up Atomineer as needed for that solution and then copy your preferences into the solution-relative search folder you configured above. As you switch between different solutions Atomineer will dynamically pick up the relevant preferences.

Multi-user Installation

Atomineer only installs for the current user account on a given PC. To install for multiple users you will need to re-run the installer on each user account. This is to allow each user to have their own independent Atomineer settings.

If you wish to share settings between these users, we recommend keeping your settings in source control as described in the next section.

Sharing Preferences/Customisations across Your Team

Atomineer strives to be very configurable to allow everyone to get what they need out of it. When installing Atomineer for a Team, it is recommended practice to share the preferences and customisations via your Source Control system as follows.

> ⓘ It's recommended that each team member uses a standardised location on their hard drive for their source code (e.g. "C:\Code"). Several similar locations can be used to accommodate different user setups if needed (e.g. "C:\Code" or "D:\Code").

Master preferences set-up

- In the Atomineer options, switch to the **General Settings** tab
- In the **Preference Storage** section, enter the path(s) of your standardised folder locations, with each folder path separated by a semicolon, e.g.

```
C:\Code\AtomineerSettings; D:\Code\AtomineerSettings
```

- Click **Store Preferences on my Search Path** and your preferences will now be saved to the first folder in the above path (i.e. “C:\Code\AtomineerSettings” in this example)
- In Source Control, check in any preferences file(s) that Atomineer saves to this folder, to allow your team members to use them

Team member set-up

- In the Atomineer options, switch to the **General Settings** tab
- In the **Preference Storage** section, enter the path(s) of your standardised folder locations, with each folder path separated by a semicolon, e.g.

```
C:\Code\AtomineerSettings; D:\Code\AtomineerSettings
```

- **Quit** all running instances of Visual Studio
- “Get” the preferences into this location using your source control system

Assigning Custom Hot-Keys for Atomineer Commands

Atomineer will automatically set up hot-keys for the primary commands (but only if they are not already in use for any other commands). However, if you wish to change the default hot-key assignments, follow these steps:

- In Visual Studio, go to Tools > Options
- Find the Environment\Keyboard section
- In "Show commands containing" type "Atomineer"
- The Atomineer commands will be listed. To add a hot-key binding, select a command, click in the "Press shortcut keys" field and press the key combination you wish to use to execute the command. When you are happy with the hotkey, click Assign.

Unattended Command-line Installation

Visual Studio 2005, 2008, 2010, 2012, 2013 Atmel Studio 5.x & 6.x	Visual Studio 2015 onwards, Atmel Studio 7 onwards

You can install or uninstall the Atomineer add-in from the command line for easier deployment to teams. The options are:
 	
AtomineerUtilsSetup.exe –i
Install to the default location, or upgrade an existing installation in-place.
 	
AtomineerUtilsSetup.exe -i <folder-path>
Install to the given folder path (must be on the local hard drive, and not in a protected location like Program Files. This option should only be used for first-time installs)
 	
AtomineerUtilsSetup.exe –u
Uninstall
 	
Please note:
• *Installation/Uninstallation will not proceed if Visual Studio is running.*
• *On Vista/Win7/Win8, UAC will need to be disabled to avoid any UAC prompts stalling the unattended installation (in this case, the user will have to manually choose the 'yes' option before the installation will complete).*
	Unattended installation of the Atomineer Extension can be achieved using Microsoft's 'VSIXInstaller.exe' utility, which is installed as part of Visual Studio.
	
	From the Start menu, search for “Developer Command Prompt for VS 20xx” (where the xx is the latest version of Visual Studio that has been installed).
	
	Alternatively, search for VSIXInstaller in the Visual Studio installation in “Program Files” (VS2022) or “Program Files (x86)” (earlier versions).
	
	In the command prompt that is opened you can then execute these commands:
	
	Installing
	From a command line, execute:
	
	`VSIXInstaller.exe /q "Atomineer Pro Documentation.vsix"`
	
	Uninstalling
	From a command line, execute:
	
	`VSIXInstaller.exe /q /u:fb6af4a4-3321-461d-b213-0300497e8766`

Documenting with Atomineer

The menu commands provided by Atomineer are detailed later in this manual. But first it will help to describe the documenting facilities in more detail, including:

- What documentation comments are, and the formats they can be in
- Using Atomineer to generate new documentation,
- How Atomineer updates existing documentation comments,
- How to configure Atomineer to best suit your needs
- How to batch-convert documentation from a legacy format

Documentation Comment Formats

First, let's take a quick look at the basics – What are documentation comments, and what formats does Atomineer support?

A Documentation Comment is used to describe details of a *code element* (such as a class, interface, method, property, or enumerated type). The description serves two primary purposes:

1) It can tell users of the code why, when and how to use it (call a method, use a class, etc).
2) It can describe implementation details of the code to help us maintain it.

The basic details in a documentation comment can be provided by *Self Documenting Code* (using descriptive names in the source code so that everything clearly describes what it is for). This is a great start, but there are many important details that can't be conveyed easily in self-documenting code – it typically tells you *what* a class or method is *for*, but not *why, when, and how to use it*. For example, here is a descriptive self-documenting method:

```
int FindUserAccount(string userName)
```

It's obvious that it finds a user account based on a user's name. But if we wish to use this method, there are still unanswered questions which significantly influence how we must write our code, and possibly whether this method is the one we want to use at all – these might include:

- What is the return value?
    - Is it an index into an C-style array (a value from 0 to numItems-1)?
    - Is it an index into a Visual-Basic style list (a value from 1 to numItems)?
    - Is it a hash value, unique ID, database table key or other special identifier?
    - Is a special return value (e.g. -1 or an error code) returned or an exception thrown if the user isn't found?
- What will happen if username is `null`? Are blank strings valid?
- Are there any code contracts that this method will enforce?
- Can names include spaces?
- Is userName the display name ("Jonathan R. Doe") or a login name ("JohnDoe")?

- Will the method take a long time to execute (in which case the results may need to be cached), or is it a quick method that can be called whenever needed? Does it involve a round-trip to the database, or is the information held locally?
- If the database doesn't respond or the user name is not found, will an exception be thrown?

Self-documenting code doesn't answer these questions. Of course, *if we had access* to the source code, we could spend a few minutes (or hours) finding it and reading through it (and the tree of methods it calls) to determine the answers we need, but it would be much quicker and easier for us if we could instantly access a summary of the information we need in one short burst of documentation. Just as a picture is worth a thousand words, a good documentation comment can be worth a thousand lines of code.

The documentation comment is where we can gather and describe all these critically important details so that programmers who don't know every detail of the code (other programmers, and ourselves when we return to the code after several months working elsewhere) can quickly and easily get to grips with the essential details needed to use or maintain the code.

We might describe these key features of the method like this:

FindUserAccount	
Summary	Finds a user account in the database.
Parameter "userName"	The **login** name of the user to find. Must not be null/empty.
Exceptions	Throws an InvalidArgumentException if username is null, empty, contains spaces, or is otherwise an invalid Windows Login name.
Returns	-1 if the username is not found, or the database ID (in the UserTable) of the user's record.

In order to make this information more useful, we have separated it into several *entries*. Each describes one specific part of the method, making it easier for us to zone in on the important parts of the information.

This classification of the information also makes it easy for programs to pick through our documentation and build nicely formatted printable content out of it. These programs typically require us to follow some formatting rules to make the content easy to parse. Atomineer supports the following formats:

Format name	Example entry

XML Documentation	`<param name="userName">`
	`The login name of the user to find.`
	`</param>`

JavaDoc (Doxygen)	`@param userName The login name of the user to find.`
Qt QDoc (Doxygen)	`/param userName The login name of the user to find.`
Natural Docs	`Parameters:` `    UserName - The login name of the user to find.`

Here are some of the pros and cons of different formats:

Format name	Pros
XML Documentation	Visual Studio will read XML docs and display them live in tooltips as you type calling code, so you know everything you need to know about the method you are calling without having to stop typing to find and read its source code or external documentation. Widely supported by popular documentation generation programs (e.g. Sandcastle, Doxygen) However, this format is the least human readable, and the end-tags increase the typing required to create the entries.
Doxygen, JavaDoc & QDoc	Three of the most popular documentation systems for C-style languages. Reasonably human-readable. Can be extremely compact and brief. Unfortunately, not supported by Intellisense.
Natural Docs	Natural Docs format is designed to make the comments within source code as simple and readable as possible, while still following a convention that allows them to be parsed to generate external documentation. If you are not concerned about external documentation generation, this can be a very good choice; however, it is not well supported by tools other than Atomineer, so is somewhat more limited than XmlDoc and Doxygen formats.

As a result, our recommendation (especially for Visual Basic, C# and C++code) would be to use XML Documentation – this is supported by the most tools, and the benefits of Intellisense (which grow with every new Visual Studio release) cannot be understated.

Documentation Comments in code

In code, documentation comments are usually embedded in comment blocks, with a special format used to distinguish them from normal comments. Typical styles for this are:

```
/// (a block of one or more single-line comments,
/// using /// rather than // - C++, C#, Java)
```

```
/**
(A multiline comment – C++, C#, Java)
*/
```

```
/*!
(A multiline comment – C++, C#, Java)
*/
```

```
''' (a block of one or more single-line comments,
''' using ''' rather than ' – Visual Basic)
```

Adding the descriptions above to our block format, we get a basic documentation comment for our method. Here's how it would look in the basic XML Documentation format:

```
/// <summary>Finds a user account in the database.</summary>
/// <param name="userName">The login name of the user to find. Must not be null.</param>
/// <exception cref="InvalidArgumentException">Thrown if username is null, empty,
/// contains spaces, or is otherwise invalid.</exception>
/// <returns>-1 if the username is not found, or
/// the database ID in the UserTable of the user's record.</returns>
```

And the same thing in a basic Doxygen/JavaDoc format:

```
/// @brief Finds a user account in the database.
/// @param userName The login name of the user to find. Must not be null.
/// @exception InvalidArgumentException Thrown if username is null, empty,
/// contains spaces, or is otherwise invalid.
/// @returns -1 if the username is not found, or
/// the database ID in the UserTable of the user's record.
```

Each entry starts on a new line, but can continue across several lines, as can be seen for the exception description.

In this form, however, the comments are compact but many would find them quite hard to read – While they can be used to generate external or Intellisense documentation, they don't provide very "human readable" documentation in the source code.

We can improve the readability by adding extra features (all of which are legal, optional, and a matter of personal preference for your team to decide upon):

- Use a consistent ordering of elements so readers can quickly find the information they need.
- Special lines ("separators") can be used at the top and/or bottom of the comment to visually delineate it from the surrounding code.
- Add whitespace to aid readability. This can include:
    - Blank lines within the comment to collect similar entries (such as parameter descriptions) into groups that are visually separated from other entries (such as the summary or returns entries)
    - Blank lines between the comments and the surrounding code
    - Extra whitespace between the entry tags/commands (e.g. `@brief`) and the description text.
    - Extra whitespace/indentation within the comment to create more readable text columns, or blank lines to create paragraphs in long descriptions.

For example, we could write the comments like this, without affecting the quality or layout of any final external documentation, but significantly improving the readability of the source code:

```
//////////////////////////////////////////////////////////////////////
/// <summary> Finds a user account in the database. </summary>
///
/// <param name="userName">
///     The login name of the user to find.
///     Must not be null.
/// </param>
///
/// <exception cref="InvalidArgumentException">
///     Thrown if username is null, empty, contains spaces, or is
///     otherwise invalid.
/// </exception>
///
/// <returns>
///    -1 if the username is not found, or
///    the database ID in the UserTable of the user's record.
/// </returns>
//////////////////////////////////////////////////////////////////////
```

Or:

```
/// ---------------------------------------------------------------------------
/// @brief          Finds a user account in the database.
///
/// @param userName  The login name of the user to find. Must not be null.
///
/// @exception       InvalidArgumentException is thrown if username is null, empty,
///                  contains spaces, or is otherwise invalid.
///
/// @returns         -1 if the username is not found, or
///                  the database ID in the UserTable of the user's record.
/// ---------------------------------------------------------------------------
```

(These are just examples of what is possible. There are many alternative ways the comment can be arranged to provide a compromise between readability and compactness)

Generating new documentation

Atomineer uses thousands of special rules to generate meaningful documentation based on the types of code elements (e.g. constructors/destructors/operators, event handlers, indexers, etc), commonly used names (from .NET, MVC, MVVM, MFC, Qt, stdlib, etc e.g. ToString, GetHashCode), common naming styles (e.g. GetXxx, SetXxx, XxxForm, XxxException), common abbreviations (ptr -> pointer, wnd -> window) and the context in which an element is used (e.g. 'sender' in an event handler, or the number of parameters to an Equals method, etc).

'Perfect' documentation can't always be achieved, but Atomineer attempts to provide the best documentation it can so that *even though you will often need to finish the comments yourself*, the typing involved is *minimised*.

For example, Atomineer (optionally) provides:

- The declaration of the code element being documented.
- A brief summary of the code element, containing an auto-generated description.
- The name of the author and current date.
- All parameters, generic type parameters and any return value are entered into the comment as required, with meaningful auto-generated descriptions where possible.
- Atomineer will attempt to keep related blocks of entries tidy by grouping them together and formatting them with indentation and word-wrapping, and it can automatically insert blank lines between groups to make comments more readable).
- The cursor is positioned in the <summary> section of the comment, ready for you to describe your code.

Atomineer uses some very powerful systems for generating the documentation automatically:

- A user-editable "rules" system containing thousands of rules generates meaningful documentation for code elements based on common naming practices, naming conventions, and can even combine information on parameter and return types (etc) with the naming to deduce more useful documentation.
- If existing documentation is available on an **overridden** base class method/property or an **implemented Interface** method/property, it will be copied into your new documentation comment. [*]
- If existing documentation is available on other **overloads** of a method, the docs from the best-match overload will be copied. [*]
- If a **parameter** has been used and documented in other methods of a class, the documentation from the best match will be copied. [*]
- All **exceptions** thrown directly within a method will be documented automatically, including ones generated by C# Code Contracts.

> ⓘ The three features marked [*] above rely on the Visual Studio Intellisense system. If this is disabled, your documentation is not in Documentation XML format for use by Intellisense, or the Intellisense information is not available (e.g. in C, Java, PHP or UnrealScript code, and sometimes in C++, C# or VB if the code hasn't compiled successfully or is not included in an open Project)

then Atomineer will have no choice but to fall back to its regular auto-documentation generation algorithm.

All of the above is configurable - The style of blocks, the entries they contain and their ordering, indentation styles and word-wrap, and use of blank lines (both within the comment and between the comment and any surrounding code).

In addition to generating documentation comments on code elements:

- If 'Add Doc Comment' is executed in the top line of a file, Atomineer will *replace* any existing file header with a new one in your configured style.

- If 'Add Doc Comment' is executed within a documentation comment or a regular comment, Atomineer will apply word-wrapping to the comment to tidy it up. This is so that creating and updating all code documentation can be done with a single convenient hot-key that applies an appropriate action based on the context.

  ⓘ (Note that word-wrapping does not happen live as you type; you must execute the Add Doc Comment command to re-wrap the comment when you want it to be done. This is because word wrapping can be destructive of important formatting such as in code snippets. By only wrapping when you execute the command, any unwanted side effects can simply be undone)

- Finally, if 'Add Doc Comment' is executed in any other text-based file format, Atomineer will add a file header, footer, or 'catch all' comment to the file. These are somewhat simpler than the documentation comments that are generated for the file formats listed above, but can still save considerable time and effort when documenting other source files (SQL, XML, HTML, python, ruby, etc). Note that these comments will not be updated by Atomineer, only created.

Updating existing documentation comments

If a documentation comment already exists, it will be parsed and updated - for example if you add a new parameter to a method, remove an exception thrown in your method, or change a Property by adding a get/set accessor, the existing comment will be updated to reflect the new details, effortlessly keeping the comment in sync with the code. This feature can also serve to automatically convert most existing Documentation XML/Doxygen comments into the Atomineer format (see below for more details).

In addition, the block will be (optionally) reformatted to keep it tidy:

- Documentation entries will be formatted into a consistent ordering, grouped together into related blocks, with optional blank lines between them. Within each block of entries (e.g. <param> lists), the text for the entries can all be indented to start at the same column to enhance readability.

- If enabled, the word-wrap option will automatically reformat each comment entry to keep it tidy. The indentation level of the first line of each entry will be used in all subsequent lines. The word wrap will preserve blank lines, any lines at a greater indent level than the first line of the entry, and lines starting with certain text (examples (e.g., c.f., i.e.), bullet lists (starting with -, *, +), etc). To deliberately force a new line, you can end the previous line with two spaces.

- Word wrap is suppressed for blocks of text that lie between <pre>, <code>, @code...@endcode, and @verbatim...@endverbatim tags. Note that these start/end tags must be the first nonblank text on the line.

- Documentation comments can include a mixture of XML and/or HTML elements. However, you may also wish to use <, > or & characters in descriptions and code examples, which must be encoded specially in XML and HTML. Atomineer offers three approaches to handling this situation:
    - The default is to convert these special characters into proper XML - the entities '<', '>', and '&' if necessary.
    - Another option is to convert them to legal characters that are more human readable: {, } and +
    - The third option is to leave them as-is. This makes your comments more human readable but no longer compatible with XML or HTML standards.

      Note: In a similar way,< and > characters within XML 'cref' attributes will always be converted to { }, as this is the required syntax for the documentation XML format.

- If enabled, the number of blank lines above and below the doc comment will be corrected to a user-specified standard to help keep code files tidy. The default is to enforce a single blank line above and below the comment. The number of blank lines to enforce after {, #region, #if, public:, private:, protected can also be configured.

'Deleted' entries

Any DocComment entries that are no longer required (e.g. deleted parameters or exceptions no longer being thrown) will be inserted at the end of the comment block with a ### prefix:

```
///### <param name='value'>  This parameter has just been deleted</param>
```

This approach **preserves any text** from these entries in case you wish to re-use it anywhere else (e.g. a renamed parameter), and **ensures you are aware** that Atomineer thinks the text should be deleted.

If you execute the Doc Comment command a second time, any such 'deleted' lines will be automatically removed to save you having to manually clean up the comment.

Note that Atomineer can be configured to **silently delete** unneeded/invalid entries in the “Doc General” tab of the Atomineer Options.

Additional tips

- If you wish to re-generate the auto-doc text for any entry in your comment, or the entire comment, simply delete the entry/comment and execute Add Doc Comment again to update it. There is no need to delete the entire comment as Atomineer will automatically fill in any missing parts while preserving the others.

- To document exceptions that pass through your method from a called method (i.e. uncaught exceptions), document them as **'Passed when'** rather than **'Thrown when'** and the description will be preserved when Atomineer updates the comment. *(Note: The Atomineer variable %except-passprefix% can be set to a different prefix to control the wording Atomineer uses to detect passed exceptions)*

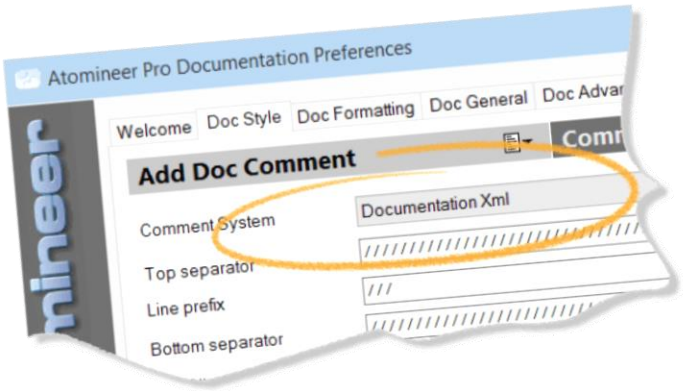
Configuring Atomineer

So how can we configure Atomineer to get the ideal comment format?

Set the Comment Format

First, decide on a core comment format - Xml Documentation, JavaDoc, QDoc or NaturalDoc?

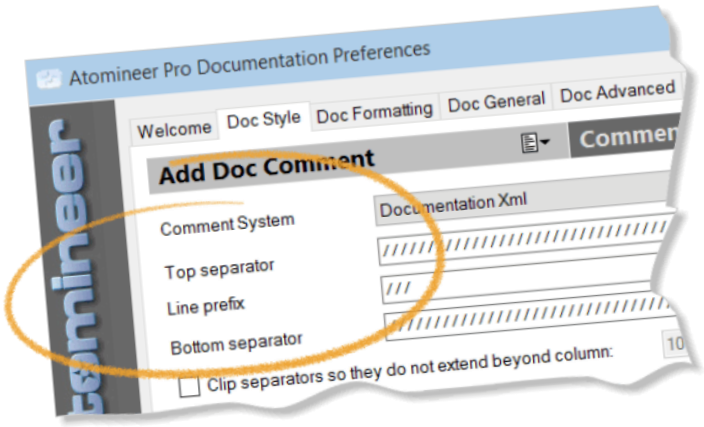
This can be set on the **Doc Style** tab of the Atomineer Options:



(Note that setting this option may set some other preferences to common defaults, so even though all affected options can be changed later, this option should be chosen first)

Choose the comment block style - Separators

Next, let's set the style of the "border" around our comment block – the top and bottom separator lines, and the prefix that goes onto each of the lines.



These fields can be dropped-down to choose from some common/example options, or you can just **type any text you wish** into them. Atomineer doesn't impose many restrictions, but you need to choose something that is compatible with the language you are using as well as any 3rd party documentation extraction programs.

If you wish to use C-language multi-line comments, the top and bottom separators need to form the beginning and end of a valid comment block, so at a minimum something like this is required:

```
/*
*/
```

Or even

```
#if false
#endif
```

(A common style, fully supported by Visual Studio and Doxygen is to use a double asterisk at the start of the comment: `/**`)

If you will use single-line comments (i.e. // in C-languages and ‘ in Visual Basic) to form your doc comments, you don’t need top and bottom separators at all, so you can set them to **(None)** if you wish.

You can also add any text you like “within” the comment separator lines, so you can make separators like these:

```
/*****************************************************************************
 *****************************************************************************/
```

```
/* --- Copyright XYZ Corporation ---
 */
```

```
/** ===========================================================================
    --------------------------------------------------------------------------- */
```

```
/// --==000==------==000==------==000==----==000==----==000==----==000==------==000==--
(None)
```

```
////////////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////////////////////
```

```
///============================================================================
///----------------------------------------------------------------------------
```

As long as it’s a legal comment, and can’t be confused for a doc entry, anything is allowed.

Atomineer also supports a multi-line format for the top separator. Just add `\n` into the text anywhere that you wish to insert a newline. So this format:

```
/* ------------------------\n   Copyright XYZ corporation\n   ------------------------
```

... will produce a 3-line top separator like this:

```
/* ------------------------
   Copyright XYZ corporation
   ------------------------
```

Finally, you can use `%type%` and `%name%` to add the type/name of the code element into the **top** separator, like this:

```
/// --- %name% (%type%) ------------------------------------------------------
```

... which produces a doc comment separator like this:

```
/// --- FindUserAccount (Method) ---------------------------------------------
```

Of course, for Visual Basic, the start of the separator must be ‘ or REM to make it a valid comment.

```
''' ++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
```

The **clip separators** option allows you to limit the width of the separator lines. Normally as you indent a comment block the entire block moves to the right, but this can make viewing or printing code more difficult. To avoid this, set the clip column for separators and they will be truncated at the chosen character column. (This option is usually combined with the **Word Wrap Column** option, described below)

Choose the comment block style - Line Prefixes

Next, you can set a prefix for every line within the comment. For a multiline (`/* ... */`) comment you can use any text within reason, but for a block made of single-line comments, this should *at a minimum* start with the comment prefix // or '. This allows you to create blocks such as this (which has no line prefixes)

```
/**
    @summary Finds a user account
  */
```

Or this (a line prefix of `  *` – including several spaces to line the asterisk up with the top/bottom separator lines):

```
/**
  * @summary Finds a user account
  *
  **/
```

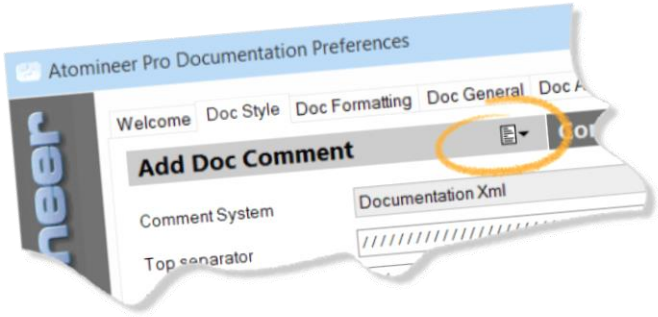
For single-line-comment blocks, it is necessary to start the prefix with a comment header, /// or '''

```
/// --------------------------------------------------------------------------------
/// @summary Finds a user account
/// --------------------------------------------------------------------------------
```

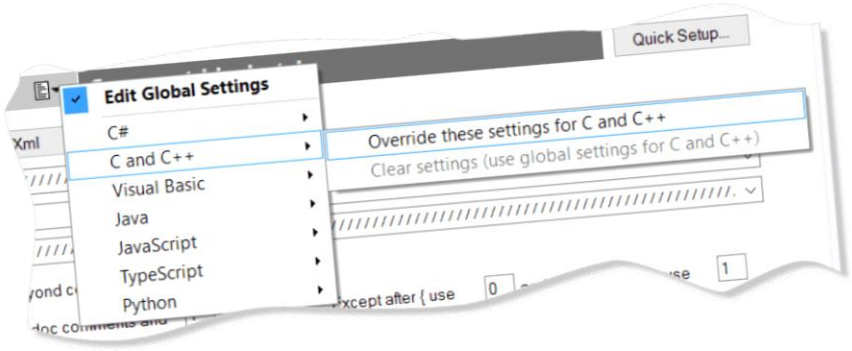
```
''' ------------------------------======0000======------------------------------
''' <summary> Finds a user account </summary>
''' ------------------------------======0000======------------------------------
```

Advanced topic: Choosing different styles for each coding language

Atomineer can use a different comment style for each coding language you use. The settings that you edit by default are the "global" settings – they apply to all languages. However, if you wish you can create a separate set of preferences for each language you use. In the title strip for the section you will see a "drop down menu" icon:



Click this and a menu will be shown:



From the menu, choose "**Override these settings ...**" for any language you wish to use special settings for, and the currently shown settings will be duplicated for the language you have chosen.

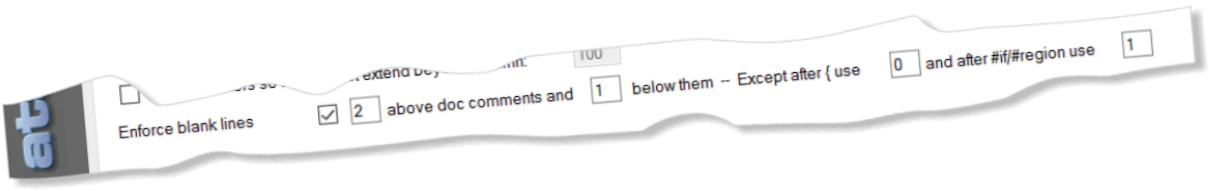
The title strip in the preferences will go blue and indicate the language name you are configuring. Also, the name of the language will go bold in this drop-down menu to indicate that it is currently being overridden. Any changes you make to the preferences will now only affect your doc comments in that one coding language.

You can clear the overridden preferences at any time form the same submenu. This will remove the language-specific preferences and return to using the global settings for this language.

Choose **Edit Global Settings** to return to editing the main preferences.

Automatic Tidying features – Whitespace Control

Now that you have the block outline you require, move down the **Doc Style** tab to the last Style setting:



The **Enforce blank lines** checkbox enables or disables a white-space control feature. With this enabled, when Atomineer generates or updates a documentation comment, it will also add or remove blank lines above and below the comment to help keep the code tidy and consistent. You may enter 0 to remove all blank lines between code and comments, like this:

```
int SomeValue { get; set; }
/// <summary>Opens the database</summary>
void OpenDatabase()
{
}
```

Or you could require 2 blank lines above and 0 below:

```
int SomeValue { get; set; }


/// <summary>Opens the database</summary>
void OpenDatabase()
{
}
```

Or 3 blank lines above and 1 below:

```
int SomeValue { get; set; }



/// <summary>Opens the database</summary>

void OpenDatabase()
{
}
```

You can independently set the gap between #commands and the start of code scopes `{` and the comment, so that you don't get unwanted space at the start of regions/scopes - so you could use 1 line after a #region even if you normally require 3 blanks 'above':

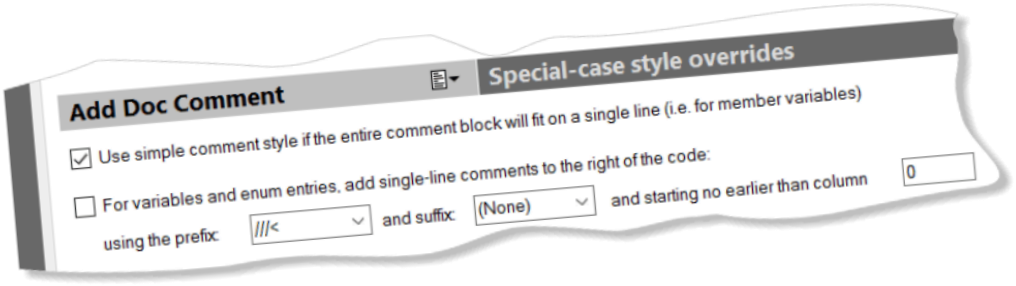
```
#region Database

/// <summary>Opens the database</summary>
void OpenDatabase()
{
}
```

Or if you prefer, disable this feature by un-ticking the check-box, and Atomineer will leave the blank lines above/below the comment exactly as you originally typed them.

Special Cases for 'Simple' Comment Styles

The '**Simple' comment style** options are primarily used for member variable and enum value comments.



If **simple comment style** is enabled, when the entire documentation comment body could fit onto a single line (i.e. it's a single entry comment with a very brief description) it will be output in this form:

```
/// <summary>Opens the database</summary>
```

Instead of a full comment (for example):

```
////////////////////////////////////////////////////////////////////////////////
/// <summary>
///    Opens the database
/// </summary>
////////////////////////////////////////////////////////////////////////////////
```

This option helps to keep the simplest comments compact and brief.

Note however that as soon as the description becomes verbose enough to occupy more than one line, Atomineer will convert the comment into the regular multi-line form when you update the comment.

The remaining options instruct Atomineer to go even further with member variable and enum entry comments, and put the comment on the same line as the code element, as an end-of-line comment. *(Note: This form is not supported by **XML** Documentation tools, as they require the comment to **precede** the code element, but this form of comment is compatible with Doxygen using a special markup).*

For example, a regular doc comment (using Doxygen/JavaDoc markup)

```
/// @brief The maximum speed of this Vehicle
double maxSpeed;
```

...and the equivalent end-of-line comment

```
double maxSpeed;        ///< The maximum speed of this Vehicle
```

The options allow you to set the comment start and end text, so if you prefer you can use `/*<` and `*/` to create a C-style comment:

```
double maxSpeed;        /*< The maximum speed of this Vehicle */
```

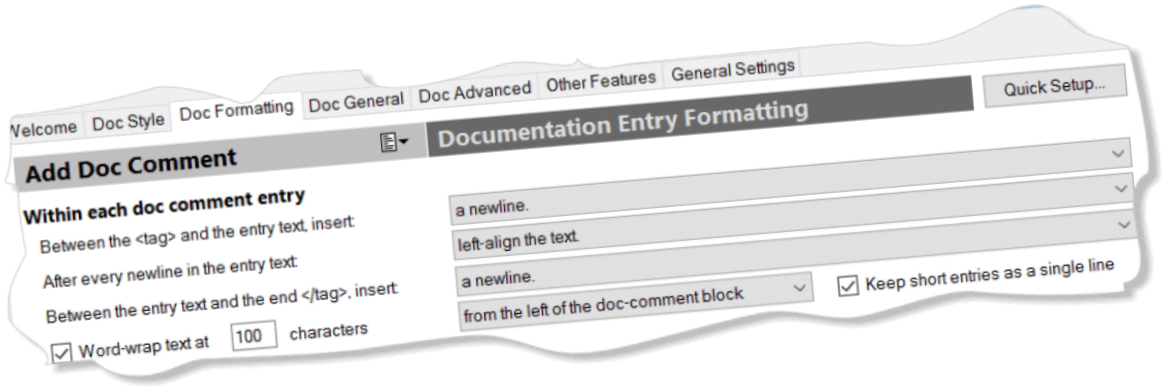
A text column index can be set to force the comments towards the right. This helps to form a tidy column of comments to the right of a block of variables:

```
double maxSpeed;                        ///< The maximum speed of this Vehicle
EngineParameters engineParams;          ///< The engine parameters block
```

Configure Doc Comment Layout/Formatting

Now we can move on to the actual Doc Comment Entries within your comment block.

Switch to the **Doc Formatting** tab and look at the first set of options:



The options within this tab can be set up quickly by clicking the **Quick Setup...** button to re-run the initial setup wizard. However, you can gain much finer control of the formatting if you edit the options on this tab directly.

First, some background to explain these options:

- Each documentation entry is made up of a start ”tag” (command), which may look like `<summary>`, `@summary`, `/summary`, or `Summary:` depending on the comment format in use. This is followed by the description *(entry text)* for the entry. For XML (only) the entry is completed by a corresponding end tag, `</summary>`.
- A documentation entry may occupy a single line, or may be continued over several lines or paragraphs.
- Some documentation entries are standalone (e.g. `summary`, `returns` only occur once in any documentation comment) while others may occur several times, forming a “group” of similar entries (e.g. if a method has 3 parameters, there will be a group of 3 `param` entries)
- In the following, a “Tab” means “moving to the next tab-column in the document”. If your Visual Studio is configured to do this with spaces, Atomineer will insert spaces to achieve this; if it’s configured for tab characters, Atomineer will use tabs.

Atomineer offers a simple but powerful set of options to control how the entries are laid out. Although each of these options is simple, they interact with each other, so understanding exactly what each option does to the text is very helpful when trying to achieve a particular format. This is why the options are worded in a very particular manner.

Documentation Entry Formatting

The first combo box

Atomineer starts to build an entry by outputting the tag, e.g. `<summary>`. In between this tag and the entry text, you may insert:

- **Nothing**. This means the entry text follows directly after the tag:

```
<summary>Finds the user account.
```

- **A space character** causes a single space to be inserted:

```
<summary> Finds the user account.
```

- **A tab character** causes a single tab to be inserted:

```
<summary>	Finds the user account.
```

- **A newline** causes the text to be moved down on to the next line so the tag is on its own:

```
<summary>
Finds the user account.
```

The second Combo Box

This controls how each line in the entry is indented, *if the entry flows on to more than one line*. At the start of each new line in the entry, it can:

- **Left-align the text** so that all lines start at the left edge of the block. Two examples are given here, showing the difference between adding a newline after the tag and adding a space after the tag – this illustrates why this option is for "after each newline..."

```
<summary>
Finds the user account in the
user database.
</summary>
```

```
<summary> Finds the user account in
the user database</summary>
```

- The **Indent the text (one space/tabstop)** options insert a space or tab to indent the entry text:

```
<summary>
    Finds the user account in the
    user database.
</summary>
```

```
<summary> Finds the user account in
    the user database</summary>
```

- **Align the text with the start of the previous line** inserts enough spaces or tabs to indent the entry text so that it forms a column of text:

```
<summary>
    Finds the user account in the
    user database.
</summary>
```

```
<summary> Finds the user account in
          the user database</summary>
```

The third combo box

This combo box is *only relevant for XML Documentation*, as it affects the placement of the end tag, which is not used in the other documentation formats.

It works just like the first option, inserting nothing, a space or tab, or a new line after the entry text, but before the end tag:

```
<summary>
    Finds the user account in the
    user database.</summary>
```

```
<summary>
    Finds the user account in the
    user database. </summary>
```

```
<summary>
    Finds the user account in the
    user database.    </summary>
```

```
<summary>
    Finds the user account in the
    user database.
</summary>
```

This combo box has one final option, which inserts a newline but *also* indents the end tag to line up with the text column, like this:

```
<summary>
    Finds the user account in the
    user database.
    </summary>
```

Word Wrapping

The check box at the bottom of this section controls word-wrapping. Atomineer can optionally word-wrap the entry text to keep it tidy, using a smart algorithm that tries to preserve indentation and formatting where it is needed, but wraps plain text. To disable word-wrapping, un-tick the checkbox.

Word wrapping is applied at a character column (by default 100 characters). This can be an absolute position, i.e. 100 characters from the left edge of the page, or it can be relative to the start of the comment so that all comments are the same "width" regardless of their indentation:

- From left edge of page, the comment will become shorter as it is indented deeper:

```
/// This is some text that is word wrapped at a particular column
/// so that it forms an absolute boundary, useful if the code will
/// be printed out

            /// This is some text that is word wrapped at a
            /// particular column so that it forms an absolute
            /// boundary, useful if the code will be printed out
```

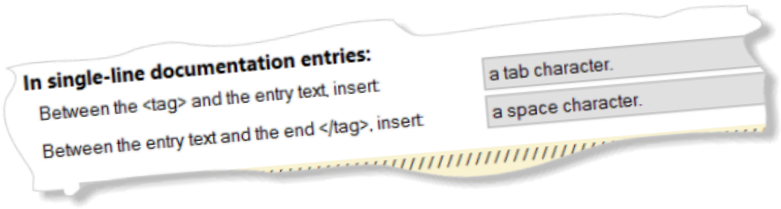
- From left edge of comment, the comment will always be 100 characters wide regardless of indentation:

```
/// This is some text that is word wrapped at a particular column
/// so that it forms an absolute boundary, useful if the code will
/// be printed out

            /// This is some text that is word wrapped at a particular column
            /// so that it forms an absolute boundary, useful if the code will
            /// be printed out
```

Special-case for short entries

Finally, there is a special-case feature for short entries. If the entire entry is short enough to fit onto a single line, then all of the above rules can be **optionally ignored**, and new formatting options are used. These are a bit further down the same tab page:



These options work exactly as described above, but for **single-line** entries only. As they are for a single line entry, you cannot insert newlines into them, so the options are just: none, space, tab.

Special Cases for Groups of Entries

As described above, some documentation entries are standalone (e.g. `summary`, `returns` only occur once in any documentation comment) while others may occur several times, forming a "group" of similar entries (e.g. if a method has 3 parameters, there will be a group of 3 `param` entries).

The options we've looked at so far only apply to the standalone entries. Where a group is involved, a separate set of rules are applied. They work just as described above for the standalone entry options.

These also include a couple of extra options to enhance readability of the comment. The first of these is to add blanks between groups/blocks of entries:

With this disabled, the comment will look like this:

```
/// ------------------------------------------------------------------------
/// @brief Finds a user account in the database.
/// @param username   The login name of the user to find. Must not be null.
/// @param autoCreate true to auto-create an account if it is not found.
/// @exception InvalidArgumentException thrown if username is null, empty,
///            contains spaces, or is otherwise invalid.
/// @returns -1 if the username is not found, or
///          the database ID in the UserTable of the user's record.
/// ------------------------------------------------------------------------
```

But with it enabled, blank lines are inserted between the “groups” to aid readability:

```
/// ------------------------------------------------------------------------
/// @brief Finds a user account in the database.
///
/// @param username The login name of the user to find. Must not be null.
/// @param autoCreate true to auto-create an account if it is not found.
///
/// @exception InvalidArgumentException thrown if username is null, empty,
///            contains spaces, or is otherwise invalid.
///
/// @returns -1 if the username is not found, or
///          the database ID in the UserTable of the user’s record.
/// ------------------------------------------------------------------------
```

(Note here that the two “param” entries are not split by a blank line – they form a group of the same type of entry, so they remain together as a coherent block of entries)

The next option dictates whether all entries in the group are aligned to the same text column, or if each entry is formatted independently.

With the option disabled, each entry is formatted on its own – it maximises use of the available space, but is hard to read:

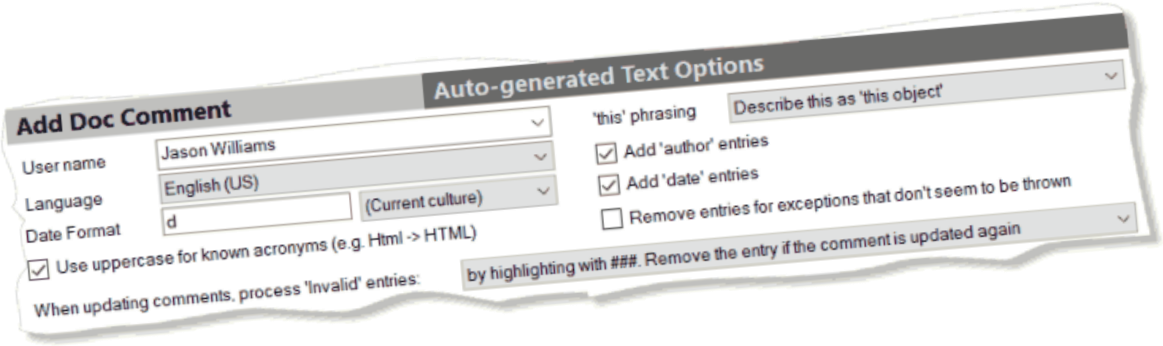
```
/// @param username The login name of the user to find. Must not be null.
/// @param autoCreateIfMissing true to auto-create an account if it is not found.
```

With the option enabled, the entries are aligned with each other, making the parameter-list much easier to read, but at the expense of using more space:

```
/// @param username             The login name of the user to find. Must not be null.
/// @param autoCreateIfMissing  true to auto-create an account if it is not found.
```

Fine Tuning

At this point you should now have a format that you're happy with, and can start tweaking some of the finer details. Let's go back to the **Doc General** tab and look at the other options available there.



- Your **User Name** is optionally entered into Doc Comments in the `<remarks>` or `author` entry. By default, Atomineer will use your Windows Login name, but you can type the name you wish to use into this field to override it.

- **'this' phrasing** controls the way that "this" object is described in descriptions. e.g. If your class was called "Vehicle", these naming options would produce descriptions like:

```
/// <summary> Saves this object to a file </summary>
```

```
/// <summary> Saves this instance to a file </summary>
```

```
/// <summary> Saves this Vehicle to a file </summary>
```

- The **Date** options allow you to set how dates are formatted in the `<remarks>` or `date` entries. These use the standard .Net format strings and cultures, so "d" gives dates like "9/10/2011", and "dd MMM yyyy" produces "4 Feb 2012".

- **'this' phrasing** allows you to control how Atomineer describes "this class" in documentation (as in "Constructs a new instance of this class"). Options include the classname (with or without namespace)

- **Add 'author' entries** and **Add 'date' entries** control whether the author's name and/or date are added to every generated comment. In DocXML these are both added to the `<remarks>` entry (un-tick both options to stop the remarks entry being generated at all), while in Doxygen formats they generate separate `author` and `date` entries.

- By default, when updating a doc comment, Atomineer will try to match any **exception entries** to a throw statement in the method body, to be sure that all exceptions thrown are documented correctly. If an exception is not obviously thrown in a method then its entry will be removed (aside: If you know an exception is thrown by a called method, you can inform Atomineer of this by documenting the exception as 'Passed when…' rather than 'Thrown when…', and Atomineer will know to preserve the exception entry). This option allows you

to disable this feature entirely – with the option ticked, Atomineer will never remove existing exception entries.

- When updating a comment, if Atomineer finds a doc entry such as a `<param>` entry for a parameter that is no longer present in the code, or a custom entry that is not defined in the Atomineer Templates, it will consider it **invalid** and remove it, in one of two ways:
  - Delete the invalid entry immediately.
  - Mark the entry with a ### prefix like this:

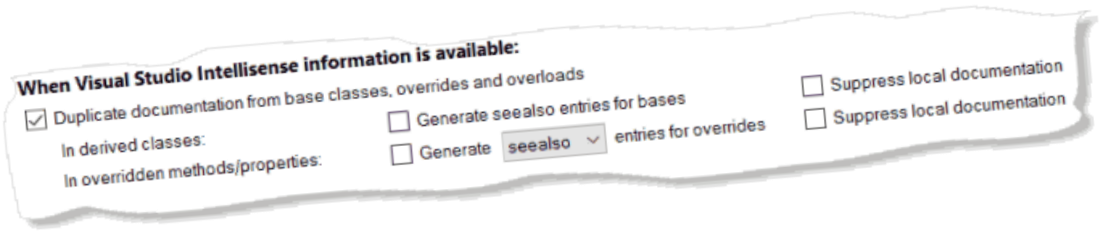
```
/// ### <param cref='missingParam'> This parameter is missing </param>
```

    If you **update the comment a second time**, Atomineer will then remove the marked entry. This allows you to see what you will lose when updating, giving you a chance to rescue any description text that you realise you still want to use somewhere else in the comment.

    Note that Atomineer considers the most common documentation entries "valid" by default, and any entries that are not in this list will be removed. If you wish to use additional entry types, you will need to add custom entries to your documentation Templates. This is a simple proves which is detailed in ***Documentation Rules and Templates****, below.*

Using Intellisense

The next group of options control how Atomineer uses Visual Studio's Intellisense system. This allows Atomineer to search through classes and their base classes for related documentation that it can duplicate.



The first checkbox controls duplication of existing documentation. Examples of this are:

- If you have a parameter named `userId` used throughout a class, Atomineer can duplicate the existing documentation every time it is used, to save you having to rewrite the description of the userId parameter again and again.
- If you have several overloads of a method in a class, Atomineer will duplicate as much documentation as it can between them.
- If you override a virtual method or implement an interface, Atomineer will copy the description from the base class/interface.

The options below this can be used to automatically add "see also" entries to documentation for base classes/interfaces for a **class**, and base classes/interfaces for **methods**. For overrides you can

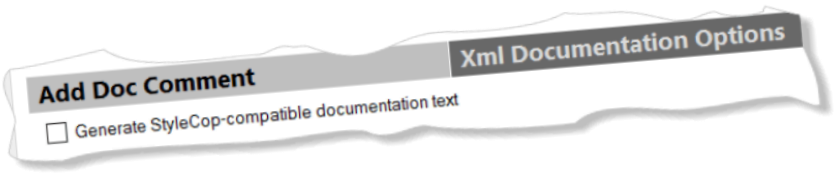
also choose whether this reference to base class documentation is generated as a “seealso” entry or an “inheritdoc” entry (note: inheritdoc is only available in Xml Documentation format). In addition, you can **Suppress Local Documentation**, which stops Atomineer generating any descriptions at all – it will *only* insert a see-also entry that refers to the base class/interface documentation

Notes:

- *For description duplication to work well, it is important that each name used in your class has a unique purpose or usage, so that the description for each name in your code is unique.*
- *Intellisense is a system built in to Visual Studio, which Atomineer uses to duplicate description text. Intellisense information is not always available, so sometimes Atomineer may not be able to duplicate documentation as expected. This is an effect of how Intellisense works rather than a problem with Atomineer. To get the best results, use the most recent version of Visual Studio and make sure your code has compiled. (In general C# and VB are well supported by Visual Studio. C++ is supported but does not always work if the code has not been recently compiled. Other languages (Java) may not be supported at all)*
- *In Visual Studio 2017, the “Lightweight Solution Load” option appears to reduce the availability of Intellisense. You may therefore have to expand projects in your Solution Explorer or even disable Lightweight Solution Load in order for Atomineer to be able to inherit documentation for you. It is hoped that as LSL is developed in future releases of Visual Studio that this situation might improve*

XML Documentation Options

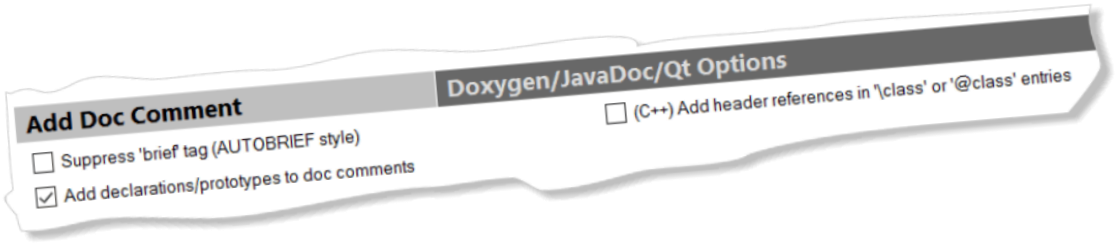
This section provides options for controlling the documentation that is output in XML documentation comments.



- **Generate StyleCop-compatible text** adjusts the way that some code elements (e.g. constructors) are described so that they follow Microsoft's **StyleCop** code analysis tool's description rules.

Doxygen/JavaDoc/QDoc Documentation Options

This section provides options for controlling the documentation that is output in Doxygen/JavaDoc/QDoc documentation comments. The Doxygen, JavaDoc and QDoc options are:



- **Suppress 'brief' tag**. Normally, the description is added as a `@brief` or `/brief` entry.

```
/// @brief Finds a user account in the database
///
/// @param userName The user's login name
```

However, in 'Auto Brief' style, any text (without a tag) at the top of the comment is automatically assumed to be the brief entry, making the comment more readable.

```
/// Finds a user account in the database
///
/// @param userName The user's login name
```

- Doxygen will associate a documentation comment with the code that immediately follows it. However, you can optionally add the **declaration** of the code element in the comment. For example:

```
/// @fn FindUserAccount(string userName)
```

- **(C++) Add header references** will include header references when generating a @class entry, as in:

```
@class Test class.h "inc/class.h"
```

Restricting Documentation by Access Levels

By default, Atomineer allows you to document any supported code element with the “Add Doc Comment” and “Doc All In...” commands. However, if you are creating (e.g.) a library, you may only want to document public/protected members while leaving private/internal members alone.

These options give you control over what Atomineer will document.

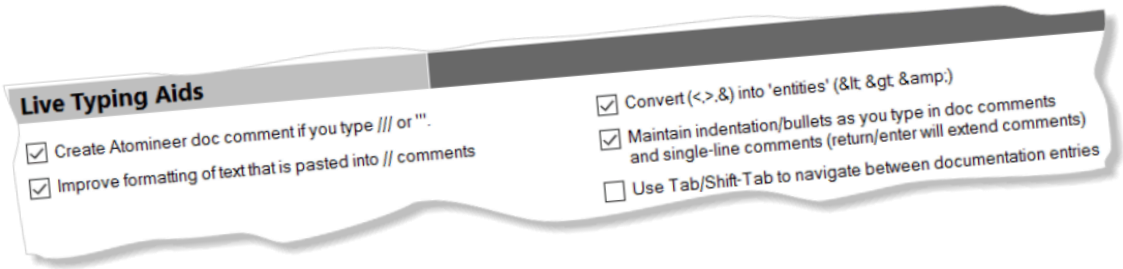
The first option controls whether the settings here apply just to the “Doc All In...” commands, or if you also want to affect “Add Doc Comment” as well. The second selects the access levels for which documentation comments will be allowed (“all” allows all members to be commented; “public,protected” would allow public/internal methods but not private/internal ones, etc)

Note:

- *If documentation is restricted, then attempting to document an “illegal” code element will remove any existing documentation from it, or ignore the attempt if it is not documented. This can make it appear that Atomineer is not working if you do not realise why.*
- *This option is generally only available for languages that include the access level of each code element in its declaration (C#, VB, Java).*

Live Typing Aids

Atomineer provides some special aids while you are typing within a comment or documentation-comment block, to help you to enter your descriptions faster. These aids can be controlled in the **Live Aids** tab.



The aids include:

- If you type /// (in C#) or ''' (in Visual Basic) on a blank line, Visual Studio will add a skeleton Doc Comment for you. Atomineer can optionally extend this behaviour in two ways:
    1. Typing **///** or **/**** or **'''** will add a doc comment in **all** supported languages (C, C++, C++/CLI, C#, Java, Typescript, PHP, UnrealScript, Visual Basic, Python, etc)
    2. Instead of an empty skeleton comment, a complete Atomineer comment will be generated.

- If you type a newline (enter/return), Atomineer will assume you wish to continue typing the comment on the next line. It adds the comment prefix and indents the cursor to match the previous line so you can just type description text without worrying about manually formatting the comment block. In addition, if your line looks like it is part of a bullet-point list (`+`, `-`, `*`) or numbered list (`1. 2. 3.` or `a) b) c)`), Atomineer will automatically add the next entry for you. For example, typing without the aids, you might get:

```
    /// @brief An enum indicating whether the vehicle is:
- A bus,
A train
An aeroplane
A truck
```

  But with the live typing aids, you'll get this result:

```
    /// @brief An enum indicating whether the vehicle is:
    ///        - A bus,
    ///        - A train
    ///        - An aeroplane
    ///        - A truck
```

  Press return **twice** in a row to exit this special "comment editing mode".

- All the documentation comment formats will allow you to embed HTML formatting commands like `<b>bold</b>`, and XML Documentation requires the comment body to be valid XML. For these reasons, you can't use the characters `& < >` within your comments, as they will cause errors in external documentation tools. When updating your comment, Atomineer can detect and correct these standalone characters, converting them into the HTML/XML "entity" codes `&amp;` `&lt;` and `&gt;` so that they remain compatible with XML/HTML formatting.

- If you paste into a comment or documentation comment, Atomineer can automatically reformat the pasted text to integrate it more cleanly into the comment. If you copy a chunk of an existing comment and paste it into another comment, the comment headers will be replaced – so copying this:

```
    // Some text in a regular comment.
    // It has two lines.
```

...and pasting into this (at the arrowed location):

```
/// <summary>
/// My Doc Comment.
/// </summary>
```

...results in:

```
/// <summary>
/// My Doc Comment. Some text in a regular comment.
/// It has two lines.
/// </summary>
```

...instead of the usual Visual Studio behaviour of:

```
/// <summary>
/// My Doc Comment.     // Some text in a regular comment
// It has two lines
/// </summary>
```

Atomineer works hard to ensure that copying and pasting of regular text or code samples usually works better than with the default Visual Studio handling – but if you are ever unhappy with the results, you can simply **undo** to remove the Atomineer reformatting, which will leave the text as Visual Studio normally pastes it.

Note: If you use another Extension or Add-in that monitors keypresses (such as ViEmu), it is possible that a keyboard handling clash could occur as both Extensions try to react to the same keypress. To resolve this you will unfortunately need to disable the keyboard handling in one of the Extensions.

Live Visualisation Aids (Visual Studio 2015 onwards)

The next section of options allows you to configure a highlighting option that makes it easier to differentiate doc comments and source code.

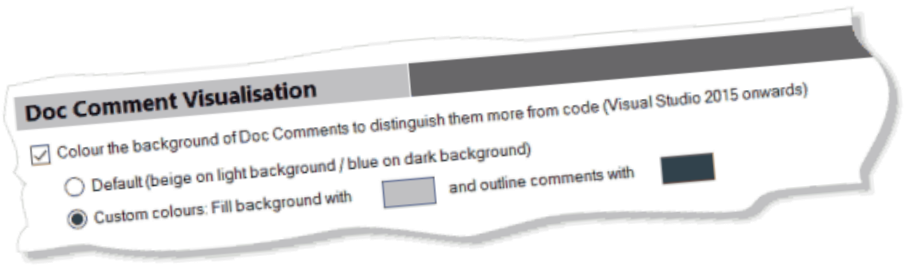
In the comment block style you can configure text-based separator lines at the top and bottom of the comment block to make it stand out – the comments can be used to visually separate methods to aid in navigating the source file.

This feature can be used to augment that approach, or to replace it entirely – it fills the background rectangle of each documentation comment with a specified fill colour, and places a boundary outline across the top and bottom of the block:

```
////////////////////////////////////////////////////////////////////////////////////////////////////
/// <summary>   Opens a database.   </summary>
///
/// <remarks>   Jason Williams, 08/07/2016. </remarks>
///
/// <param name="connectString">
/// The connection string, containing all parameters required to connect to the database.
/// </param>
///
/// <returns>   true if it succeeds, false if it fails. </returns>
////////////////////////////////////////////////////////////////////////////////////////////////////

public bool OpenDatabase(string connectString)
{
}
```

This feature can be enabled in the preferences:



Once enabled, you can choose the default colour scheme (beige when the background colour is light, or dark blue when the background colour is dark), or opt to set custom fill and outline colours.

Note that these colours are used with 50% transparency so that other highlights (e.g. selected text) remain visible, so you may need to use a slightly stronger colour to get the result you desire.

Getting the Most out of Atomineer

Introduction

Atomineer does a lot of work behind the scenes to produce the best documentation it can from your code. However, it can only provide good documentation for code that it understands – so understanding what Atomineer looks for can help you to get better results.

To start with, understanding clearly what Atomineer does can help avoid disappointments and confusion:

- Atomineer **does not** document your code. What Atomineer does is **significantly accelerate** the process of **you documenting** your code. Its aim is to provide you a **starting point** that will minimise your typing effort. Expect to tweak the wording and add information to complete the descriptions.

- Atomineer works hard to save you time when **updating** existing comments. It ensures that:
  - comments conform to your (and StyleCop's) layout and style guidelines,
  - they contain valid XML, Doxygen, QDoc, JavaDoc, NaturalDocs and/or HTML markup,
  - the documentation remains in sync with the code (e.g. each parameter is documented, listed in the order it appears in the method signature, and documentation for any deleted parameter is removed),
  - the comments remain tidy and readable using word wrapping and by controlling the whitespace in and around them.

- Atomineer has many subtle (optional) features to improve your **workflows**. These include:
  - Automatic comment extending. When you type a column of text within a comment, Atomineer will add the comment line headers and indentation for you on each new line, so you can edit the text as easily as if the comment block around it was not there. (Hint: Press return twice to break out of this mode)
  - If you paste code into a comment, Atomineer will add comment line headers and indentation to enclose the pasted code as an example.
  - Automatic bullet-point list extending as you type. Start a line with an obvious bullet point – e.g. **`- + * 1) a)`** and Atomineer will continue the bullets/numbering on the next line.
  - Automatic entity conversions (when updating comments). Characters like <, &, > can't be used in comments as they will break any XML or HTML markup in the comment. Atomineer converts these to the correct form to be sure that your Intellisense, Code Analysis and external doc tools continue to run smoothly.
  - You don't have to place the cursor perfectly to update a comment with Atomineer – anywhere in the whitespace or comment above a code element, or anywhere on the first line of the code declaration is close enough. In .net languages you can also execute Atomineer anywhere within a method's body to document it.

    - After adding or updating a comment, Atomineer tries to leave your cursor in a useful location.
    - When commenting prefixed variables (lpszName, mSize), Atomineer automatically removes the prefixes to make the descriptions cleaner
    - Atomineer separates the words in method names by detecting camelCaseNaming, PascalCaseNaming, and underscore_separated_naming styles. It tries to use this to generate a proper sentence form (words with spaces) for the name, where possible expanding abbreviations (ptr -> pointer, db -> database, etc), applying correct capitalisation for acronyms (HTML, XML, etc), phrase correction (NumItems -> Number of items) and word reordering (FileSize -> size of the file) to generate a more readable natural English description from the name.

- Atomineer is highly configurable.
    - If it is not producing what you require, it is often possible to fix it. If your company has a nonstandard coding convention, Atomineer can probably be adjusted to accommodate it.
    - If it is doing something you find annoying or you think it is interfering with another addin, this can be fixed by disabling certain minor features.
    - Let us know if you can't work out how to achieve your aims, and we'll be happy to tell you if and how it can be done. In a very high proportion of cases the answer is "yes" and the change needed is simple to apply.

Key Coding Strategies

Here are a few simple guidelines that you can follow to help Atomineer produce excellent results:

1. **Use descriptive naming**. Use brief but descriptive names for classes, methods and variables, so that Atomineer can extract useful descriptions from them as a solid basis for your own documentation text. (Consider which of these names gives you the clearest understanding of a variable's meaning: n, name, userName, nameOfTheCurrentUser).

2. **Be consistent**. Using a consistent naming style makes it easy to set up good documentation rules for Atomineer. You can add custom rules if for any bespoke conventions, but by default Atomineer looks for industry-standard and best-practice patterns such as:
   - Naming conventions like SetXXX, GetXXX, CalcXXX, ToString, Equals, Clone, Serialize, XXXFactory, XXXProxy, XXXException, XXXEventArgs
   - Use words and word-ordering consistently, e.g. Avoid mixing names like "ItemCount", "NumItems", "TotalCount", "CountOfObjects" that mean essentially the same thing – settle on a convention and use it consistently.
   - Boolean properties/parameters are made considerably clearer with the form "IsXXX" or "AreXXX" (this distinguishes **actions** like "Open" from **states** like "IsOpen"). Prefer to use Boolean names with the "positive" meaning (i.e. "IsFound" rather than "NotFound")
   - Abbreviations are useful in code for removing unnecessary verbosity as long as they are consistently applied. Atomineer will recognise many common abbreviations (e.g. ptr -> pointer) and acronyms (e.g. tcpip -> TCP/IP) to generate clean natural-English descriptions, so your documentation can be readable while your code is compact.

     Hint: If your team uses bespoke conventions, it's easy to add custom rules and abbreviations to Atomineer to help it produce good results from your codebase.

3. **Be specific.** Avoid overloading names with many meanings. E.g. Where a parameter has a *different meaning* when used in different methods, use a distinctive name (e.g. replace the vague term "id" with specific, descriptive terms like "userId", "serverId", "fileId"). Atomineer will copy documentation from same-named parameters in your class, so using each name for **one purpose** really maximises the benefits of this feature.

4. **Use OO practices.** e.g. use the single-responsibility principle and store code "one class per file". These practices help because simpler operations are more easily described by brief names; OO code files also tend to be smaller, reducing the scope of names (which improves their consistency) and speeding up Atomineer processing; and with OO, naming generally becomes more consistent as types start to share common naming patterns.

5. **Separate words in symbols.** Use camelCaseNaming, PascalCaseNaming, or underscored_naming to help Atomineer pick out the words in your names.

6. **(C++) Avoid macros.** Atomineer only parses the code in the vicinity of the cursor, and has no knowledge of the types and macros that may be defined elsewhere in your codebase. C++ macros can be replaced with arbitrary text during pre-processing and can thus confuse Atomineer and produce odd results. Many uses of macros can be replaced by typedefs and templates. (However, if necessary Atomineer can be configured with rules to correctly resolve custom macros that are used in your code)

7. If Atomineer produces "bad" documentation for a feature of your code:

    - Check that the description has not simply been copied from an existing doc comment (check the base class for overridden methods; check the same class for overloads, properties, parameters and member variables). Often Atomineer copies this documentation on the assumption that it is useful, but if existing documentation is poor, this strategy can backfire.
    - Consider if your naming approach could be more descriptive. Perhaps try a couple of variations to see if Atomineer works better for slightly different patterns.
    - It is easy to add new documentation rules to handle cases that we haven't envisaged.

Generally the guidelines above are aligned with common programming best practices. If someone has to read through the code in a method to understand what its primary function is, then that method is poorly named. If parameters have ambiguous names, programmers will find the code slightly harder to understand. By helping Atomineer to produce better documentation, you will also be writing code that is easier to read and understand – better self-documenting code.

By using Atomineer many users find that their coding style is gently massaged in the direction of these best practices (if Atomineer documents something well, it usually means it's cleanly written; if not, it can be a sign that Atomineer has been confused by some aspect of the code such as a syntax error or unclear naming)

Finally, if something isn't working as you'd like it, **ask us** (support@atomineerutils.com) for help. Many situations can be improved with only small tweaks to the XML rules that drive Atomineer. We are more than happy to help you fine tune your results, and to hear of ways we can improve Atomineer – and will usually give you a definitive answer in less than 24 hours.

"Self Documenting" Code and Doc Comments

Atomineer can only provide information that it can extract from your source code. That means that it can't usually say much more about your code than your code already tells you. So why document the code at all?

This logic misses a number of key points.

- "Self documenting" code **is seldom** "self **documenting**". It's descriptively written, easier to understand and more memorable, but that ***isn't documentation***. Documentation answers critical questions that users or maintainers of a class/method will have, like:
  - *"When is this method useful to me, and how should I use it?"*
  - *"What will happen if I pass a null for this parameter?"*
  - *"If the item is not found, what will the return value be/will an exception be thrown?"*
  - *"Why did you write this code to work in this way?"*
  - *"How is this class related to the classes over there?"*
  - *"Can you give me an example of how I use this to connect to the remote server?"*

- Reading code takes *time*. Good documentation **summarises** key knowledge clearly and concisely, and doesn't include unnecessary details. It's much quicker to work on well documented code, because the knowledge you need is distilled into a concentrated form. When writing a call to a method, Visual Studio's Intellisense tooltips will show you the documentation **live** as you type, without you having to find and read the code. And if you start coding with a better understanding, you will spend less time debugging to iron out all the little bugs and niggles and misunderstandings.

- Reading the source code requires a context switch from the calling site to the source code. You have to open new windows and jump back and forth between documents. Documentation can be popped up in a tooltip as and when you need it, left open in a separate web browser window, or sit on your desk in printed form.

- Methods are not often used in complete isolation. A task like creating and opening a connection to a server may require several lines of code. Good documentation can supply an example that helps the reader to quickly achieve their most likely task. Because of this, a large number of classes and methods in MSDN are documented with a short example of usage for a typical case – if you ever find yourself copying and pasting a code snippet from MSDN, CodeProject or StackOverflow to get you started on a task, you are relying on documentation.

- When you don't have access to the source code, it's irrelevant whether the source code was self documenting or not. Usually in this case, documentation needs to be provided in an external form (.PDF, .HTML, .CHM, etc) alongside the library binaries. This documentation can be written in external tools, but it is usually much easier to keep in sync if the basis of that documentation is extracted directly from source-code comments.

- Unit tests and TDD are employed to help us to think about and design our code well. In a similar way, writing down how we expect a method to be used forces us to explain it to someone else. We need to *think about it, understand it, and look at it from the point of view of the end user*. Documenting is a powerful way of improving the quality of the code and tests that we write.

Self-documenting code is an important step in writing clean, maintainable code, but in all but the simplest cases it is just the *first* step. For example, here is some hard to understand code:

```
int Opt(string c, string n)
{
    ...
}
```

If we make it "self-documenting" with descriptive names it becomes clearer:

```
int ReadUserOption(string categoryName, string optionName)
{
    ...
}
```

...in fact it seems fairly self-explanatory until you see the same code with *documentation.*

Does it work like this?

```
/// <summary> Reads a named user option value from the UserOpts.xml file. </summary>
///
/// <param name="categoryName"> The name of the category/group containing the option.
///                             For global preferences, pass in null or string.Empty. </param>
/// <param name="optionName">   The name of the option within the parent category.
///                             This must not be null/Empty. </param>
///
/// <exception cref="ArgumentException">
///     Thrown if optionName is null/empty.
/// </exception>
/// <exception cref="InvalidOperationException">
///     Thrown if the options file is missing or could not be read, or the value was not found.
/// </exception>
/// <exception cref="FormatException">
//      Passed if the option value couldn't be parsed as an int.
/// </exception>
///
/// <returns> The option value. </returns>

int ReadUserOption(string categoryName, string optionName)
{
    ...
}
```

Or perhaps it doesn't throw exceptions, but passes back a special return code?

```
/// <summary> Reads a named user option value from the UserOpts.xml file. </summary>
///
/// <param name="categoryName"> The name of the category/group containing the option. </param>
/// <param name="optionName">   The name of the option within the parent category. </param>
///
/// <returns>
///     The option value (always zero or positive), or the built-in default value for
///     the named preference if it is not specified in the user's file.
/// </returns>

int ReadUserOption(string categoryName, string optionName)
{
   ...
}
```

Of course, every programmer will have their own idea of how such a method should be implemented, and will make assumptions about how such a method will work when they call it. This is *why we need documentation* to clarify the details. Documentation records and communicates key points of our design to others.

The time taken to write the documentation once is usually much less than the time that will be saved by the many people who later have to read our code.

Finally, if you are worried about documentation becoming "out of sync" with the code, then stop thinking of documentation and code as being separate things. Comments are an *important and integral part of the code*. The source code is written for the compiler to use; the comments are written for the human being to use.

Why does Atomineer generate "rubbish" sometimes?

It can be frustrating trying to understand why Atomineer sometimes produces something unexpected. Here are the top reasons why this can happen occasionally:

- A syntax error in the code or documentation comment. Atomineer always tries to produce a result, and will usually do so even if you present it with bad syntax. For example, if a method does not have a closing } so that it is not complete, Atomineer can document exceptions that it finds in the following code. So bad results are often a hint to check the syntax of the surrounding code and if you find any problems, to re-document after the error is corrected.

- Low information content. Atomineer can only provide information that it can glean from the code. The richer this information source, the better it can do. If a parameter is named 'n', Atomineer will provide minimal documentation. If it is 'userName', Atomineer can do a much better job.

- In some circumstances, Atomineer copies description text from existing documentation. This works on the theory that overridden methods will operate largely as their base class method does, or that a parameter used throughout a class will have the same usage. However, if file-name and user-name parameters in a class are all called "name", Atomineer cannot

distinguish them and so it can copy inappropriate documentation. Unique naming not only helps Atomineer, but also humans attempting to read the code.

- If Atomineer finds existing documentation for a name, it assumes that the existing documentation must be of good quality, and it will duplicate it. If this existing documentation is of poor quality or the name is overloaded with many unrelated meanings, Atomineer's duplication mechanism can produce undesirable results.

- Some preferences can cause effects that can be confusing. For example, if you tell Atomineer not to document private methods, and then you type /// in front of one, Atomineer will delete the /// that you've typed. It's just doing as it has been instructed, but unless you realise *why* the /// has just vanished, it can be confusing.

- When Atomineer can't produce specialised documentation for a code element it resorts to a catch-all such as just repeating the name in sentence form (i.e. "param "launchRocket" => launch rocket"). This may seem rather useless, but if you think about how you would *document* this parameter ("true to launch the rocket immediately") you will often find that Atomineer has saved you some typing. (If you don't like this behaviour, the catch-alls can be easily disabled in your rules)

- Atomineer handles many languages and myriad code styles by parsing isolated snippets of text. Occasionally a combination of syntax and personal coding style can cause this parser to become confused and produce odd results. Send us any examples that exhibit such problems and we will do our best to improve Atomineer's code handling.

Why doesn't Atomineer add mark-up like <cref>?

By default, Atomineer's aim is to generate human readable documentation comments. Most programmers prefer to work directly in the IDE, and switching to external documentation is time consuming and difficult compared to just jumping to the top of a method or doing a "go to definition" to read the comment and code together.

However, it is often necessary to use external documentation (for example, in libraries where the source code is not available), and under these circumstances it is possible to add mark-up to the comments so that different font styles and hyperlinks can be used to improve the resulting documentation.

Where this markup is present within your documentation, Atomineer will preserve it, but it will not (by default) generate the markup for you.

We will discuss two primary types of mark-up:

References to Types (crefs)

It is common to see hyperlinks used for references to other types, for example:

```
Blends two <see cref="System.Drawing.Color"/> values.
```

…will generate documentation like this:

> Blends two System.Drawing.Color values.

Atomineer never generates crefs like this because in all of the cases where it knows the type (i.e. for a return value or parameter), the external documentation tool *also* knows (and should hyperlink to) the type anyway – so linking to the type in the description would be unnecessary duplication.

Highlighting mark-up (code sections)

Another common case is to mark-up special words, for example

```
Returns <c>true</c> if it succeeds, or <c>false</c> if it fails.
```

…will generate documentation like this:

> Returns **true** if it succeeds, or **false** if it fails.

By default, Atomineer will not generate this mark-up, but special variables (true, false, null, etc) have been added to allow you to change this. These are documented in **Appendix A**.

Advanced Understanding

Understanding how Atomineer generates documentation can help you to get better documentation out of it, and also to understand why you might sometimes get unexpected results.

When you ask it to add a documentation comment, Atomineer takes the following steps:

1. First it looks at the text near the cursor position, and parses it to see if it can find a documentation comment in any of 4 styles (as configured by the user; “tripleslash” (///); “slashstar” (/** ... */); and an optional alternate user-defined style). If found, it will parse the comment so that it can preserve any existing documentation, converting all of these formats to the user’s configured style.

2. Next, Atomineer queries the Visual Studio Code Model (Intellisense) system. Depending on the version of Visual Studio, the programming language, and whether the code has been compiled, varying degrees of information will be available, so Atomineer mostly uses this only to find descriptions that can be copied from existing documentation, such as:
    - For overrides, copy descriptions from the base Class or Interface
    - For overloads, copy descriptions from best-match sibling overloads
    - For all methods, check if parameter documentation can be duplicated from other parameters, properties or member variables in the class.

3. Then Atomineer parses the text around the cursor position. It picks out the elements that still need to be documented (methods, parameters, exceptions, etc), and executes thousands of rules against them to automatically generate a description for them.

4. In cases where Atomineer can’t provide a specific description, it uses a catch-all, such as expanding the name into a sentence form. The intention is that typical *documentation* frequently includes this text – so even though it may not seem like a useful initial description, it is still a useful starting point that will hopefully save you a bit of typing.

So we can write code in a way that maximises the quality of documentation Atomineer will generate, minimising the time and effort required to complete the documentation.

Using Doc All in File versus Doc All in Scope

These two commands are very similar in effect, but have very different underlying implementations. Understanding how they work is crucial if you are to use the best tool for the job.

Doc All in This File/Project/Solution

These commands work by using Visual Studio's Intellisense database to locate all the relevant code elements in a file and document them.

This works very well when the Intellisense information is available. However, as mentioned above: depending on the version of Visual Studio, the programming language, and whether the code has been compiled, varying degrees of information will be available, and these commands can therefore fail. In particular:

- Some languages offer no intellisense support at all, so these commands simply will not work.
- They will miss out code that lies in disabled #if blocks.
- They will miss out code when the code has not been successfully compiled.
- In languages other than C# and VB that support intellisense, these commands may sometimes report that they found nothing to document. This can usually be cured by ensuring intellisense is enabled, rebuilding the solution, and (in some cases) rebooting Visual Studio.

Doc All in This Scope

This alternative command uses Atomineer's built-in code parser rather than intellisense to understand the source code.

It reads the code line by line and attempts to work out the features of the text that represent valid code elements to document, and it then documents them. Because it does not rely on Intellisense, it will generally work on uncompilable/uncompiled code, it will process code that lies within #if blocks even if they are not currently enabled, and it works equally well for all code languages and versions of Visual Studio, and even on incomplete code snippets – in short it gives much more consistent results than Doc All In File.

However, there is a trade-off. Because it relies on interpreting the text in your source code it can occasionally become confused by unexpected syntax (particularly things like macros and #if blocks).

Note that Doc All in Scope can be used to document an entire file scope by placing the cursor at the very top of the file before executing the command.

So if you try one of the 'Doc All…' commands and the results are unsatisfactory, please undo and try the alternative approach to see if that suits your needs better.

As a general rule of thumb **Doc All In File/Project/Solution/Chosen Files** are best for .net languages (C#, VB), while **Doc All in Scope** is frequently more effective for C, C++ and Java.

Documentation Rules and Templates

Atomineer uses special XML **templates** that define the documentation entries that make up a comment and the order in which they appear. Each code element type (classes, methods, etc) has its own template so you can precisely control your documentation comments.

When there's no existing documentation entry for a code element, and Atomineer can't find any appropriate user-supplied documentation in the same class or base classes that it can make use of, it falls back to a simple but surprisingly powerful **rule-based** auto-generation system.

This system is entirely data-driven, from rules defined in a simple XML format, which are easily changed or augmented to suit your own coding style. (If you do modify your rules, or have any suggestions for improving the default rules, please consider emailing us your feedback so we can develop and improve the rules to cover different coding styles and situations better).

Custom Rules Files

To add custom rules, open the Atomineer options and switch to the **Advanced Customisation** tab. For each section of the rules there is a button. Click the appropriate button and that section of the rules will be exported to a separate custom XML file, and then opened in your editor.

> - Note that the default rules are copied *as an example to help you get started*, but you should comment them out or delete them once you have added your own rules. This allows your rules to be executed followed by the default rules, thus adding your rules without losing any existing auto-doc functionality.

The full set of customisation files that can be added is:

- UserVariables.xml
- DocXmlTemplates.xml
- DoxygenTemplates.xml
- File.xml
- Namespace.xml
- Exceptions.xml
- Typedefs.xml
- Enums.xml
- Variables.xml
- Classes.xml
- Interfaces.xml
- Methods.xml
- MethodReturns.xml

- Parameters.xml
- TypeParameters.xml
- WordExpansions.xml
- Replacements.xml (replacements/acronyms)
- NonSplittable.xml (unsplittable terms)
- Prefixes.xml

A similar set of files can be used to override the settings in prefs.xml:

- DocComment.xml
- DocXml.xml
- Doxygen.xml
- Format.xml

The file contents should be in exactly the same format as the corresponding section of rules.xml or prefs.xml. For example, you can augment the method documentation with a Methods.xml containing the following:

```
<?xml version="1.0" encoding="utf-8"?>
<Methods>
    <If name="SelfDestruct" desc="Starts a 5-minute countdown before the computer explodes."/>
</Methods>
```

To remove your customisations, simply delete the custom file.

By using the Rules Path in the Advanced Customisation tab, you can place your customisations in any folder(s) you wish, which allows them to be easily checked into your source control system for safekeeping, and for quick and easy deployment of customisations across your entire team.

Defining Rules and Templates

The rules fall into the following main groups, which are described in more detail below:

- **User Variables.** When generating documentation comment text, Atomineer can replace variable names such as %user% with context-sensitive text. This section allows you to add your own variables, such as %CompanyName%.

- **Templates.** Each template defines the comment block layout to use for a given type of code element, allowing you to specify exactly what information is expected in the comments for (e.g.) methods, properties and classes.

- **AutoDoc rules.** These describe how to generate auto documentation for different code element types (classes, structs, records, enums, exceptions, methods, properties, return codes, parameters, etc). You can easily augment the default logic to add new automatic documentation based on the names and types relating to each code element, to allow Atomineer to recognise and support the naming conventions you use.

- **Abbreviation expansions.** This section allows Atomineer to expand abbreviations to full English terms (e.g. "src" -> "source") to produce more readable auto-doc commenting.

- **Replacements & Acronyms**. This section is used when Atomineer is expanding a code element's name into a sentence format. e.g. 'OpenTcpipConnection' will be converted to 'open tcpip connection'. Each of these words is then looked-up in the replacements list, and if a match is found it is replaced as specified. In this case, the sentence will be converted to 'open TCP/IP connection'. Although this is primarily used for upper-casing acronyms, it can be used to make any substitution desired, e.g. 'atomineer' could become 'At*mineer'.

- **Prefix removals**. This works just like the Abbreviation expansions, but is only ever applied to the first 'word' of a name, and hence can be used to eliminate any coding prefixes your team employs.

- **Conversions**. This special section describes how to convert entries between Documentation XML, Doxygen and JavaDoc formats, allowing you to convert legacy comments into a new format. Note that the conversions section won't handle everything, but if you let us know your requirements, we may be able to incorporate additional features that will support your conversion needs.

<UserVariables> Section

When generating documentation, Atomineer uses variables to pass information into your rules, to allow information to be stored during rule execution, and to control the output that Atomineer generates.

The User Variables section allows you to override the built in variables and add your own custom variables. These can be output in description text or used in subsequent variable definitions using the %varName% syntax (see below).

The default variables also allow you to control certain features or text-replacements in the rules and templates. These variables are as follows, and can contain other %variables% if required. (These can be edited in the Atomineer Options)

%company%	Your company's name.
%copyright%	Your company's copyright declaration.
%useremail%	Your contact email address.
%property-getonly%	Description prefix for a read-only property.
%property-getinit%	Description prefix for a read-only property with a one-time `init` accessor.
%property-setonly%	Description prefix for a write-only property.
%property-getset%	Description prefix for a read/write property.

%indexer-getonly%	Description prefix for a read-only indexer.
%indexer-setonly%	Description prefix for a write-only indexer.
%indexer-getset%	Description prefix for a read/write indexer.
%param-optional%	Text prefixed to optional parameter descriptions.
%except-passprefix%	When Atomineer sees an exception description, it will normally remove it from the comment if it can't find a corresponding 'throw' or code Contract in the method body. By documenting an exception as "Passed when…" you can indicate to Atomineer that the exception is not thrown directly within the method, but from a subroutine that it calls. This variable allows you to change the prefix (normally "pass…") that is used to detect this situation, such as when you are not documenting in English.
%true%	The text used for documenting 'true' boolean values e.g. 'true', 'TRUE', '<c>true<c>'.
%false%	The text used for documenting 'false' boolean values.
%null%	The text used for documenting null pointer/reference values.
%nullptr%	The text used for documenting nullptr pointer values.
%stylecop%	true/false to enable or disable style cop compatible descriptions. Note that this is overridden by the value set in the main preferences.
%docBasesWithSee%	set to true to document the primary base class (only) with a <see cref='...'>>
%docPrimaryBaseWithSee%	set to true to document ALL base classes and/or interfaces with <see cref='...'>
%docOverridesWithSee%	set to true to document overrides (where possible) with <see cref='...'>. (Note that you also need to disable duplication of base documentation in the preferences to allow these rules to be applied for overrides).
%docUseAttributes%	set to true to use Attribute meta-data when generating documentation.
%alwaysAddInForParams%	set to true to always add [in] for parameters that are not [out] or [in,out]. This setting is ignored if %suppressInOut%=true.
%suppressInOut%	set to true to suppress the addition of [in], [out], [in,out] to paramter descriptions. Note that this overrides %alwaysAddInForParams%.

When documenting, each type being documented provides a set of useful variables. (For example, for a method, the method's name, return type, and parameter names and types are all supplied to

the rules being executed). More details of the type-specific variables are listed later in this document.

In addition the following global variables are always provided internally by Atomineer:

%user%	User's login name (or the overridden name provided in the Atomineer Options)
%time%	The current time, in the format specified in the Atomineer Options
%date%	The current date, in the format specified in the Atomineer Options
%day%	The current day, e.g. "Tuesday"
%dayofmonth%	The current day of the month as a number, e.g. "2"
%month%	The current month, e.g. "June"
%monthofyear%	The current month as a number, e.g. "6"
%year%	The current year, e.g. "2009"
%pathname%	Full pathname of the current file, e.g. "C:\MyProject\Source\MenuHandlers\FileOpen.cpp"
%projectpathname%	Pathname of the current file relative to the project folder (or a full pathname if it's outside the project root), e.g. "Source\MenuHandlers\FileOpen.cpp"
%projectfolder%	Project folder containing the current file, e.g. "Source\MenuHandlers"
%folder%	Leafname of the folder containing the current file, e.g. "MenuHandlers"
%leafname%	Leafname of the current file (not including the file extension) e.g. "FileOpen"
%extension%	File extension of the current file (including the ".") e.g. ".cpp"
%project%	Name of the Project containing the current file
%solution%	Name of the Solution containing the current file
%containingclass%	Name of the parent class, struct, record, interface, union or enum containing the cursor (or the file leafname if intellisense information is unavailable). Note that you can use %containingClass:Leaf% if you don't want the fully qualified name.
%namespace%	The namespace portion of containingClass.
%lang%	The source code Language of the code being documented. This will be one of the following:

	• “c” (C, C++, C++/CLI, Unrealscript) • “c#” (C Sharp) • “vb” (Visual Basic) • “typescript” (JavaScript, JScript, TypeScript) • “java” • “python” • “php” • “sql” • “unknown” (any other language)
%assemblyname%	Assembly name for the Project containing the current file
%assemblyversion%	Assembly version
%assemblyfileversion%	Assembly file version
%assemblytitle%	Assembly title
%assemblydesc%	Assembly description
%assemblycompany%	Assembly company name
%assemblycopyright%	Assembly copyright
%assemblytrademark%	Assembly trademark
%newline%	Inserts a newline into the generated description text.

Notes:

- If you create a variable with the same name as a global, the global will be replaced by your definition.
- The type-specific variables (e.g. %retType%) cannot be overridden.

Templates Section

There are three templates sections: <DocXmlTemplates> (used for generating Documentation XML comments), <DoxygenTemplates> (used for Doxygen and JavaDoc comments) and <NaturalTemplates> for NaturalDocs format comments. These sections are identical in operation, but there are slight differences to the actual format used to better support these different systems.

In both cases, the types that can be independently templated are:

- file
- namespace
- enum, bitfield
- class, struct, record, interface
- method, property, indexer, delegate, constructor, destructor (c++), finaliser (c#), operator, eventhandler, eventsender

If no template is supplied for a type, a default Atomineer layout will be used.

There are two parts to creating the templates – defining shared <EntrySettings> and defining the templates themselves. These are described below.

Custom Entry Names and Formatting - <EntrySettings>

Atomineer works with the most common documentation entry types (summary, params, etc) "out of the box", using its default settings. However, there are a number of aspects of the entries that can be fine-tuned to suit your needs in the templates files:

- Although Atomineer understands specific entry types (e.g. "summary") internally, you may wish to use a different name in your comments, such as "brief", "description" or "purpose".
- You may wish to use additional entry types (e.g. "copyright", "owner", "product") that Atomineer knows nothing about.
- You may wish to change the default formatting (whether punctuation is automatically added to the entry, or word-wrapping is applied, and so on)

To achieve this, the Templates begin with an <EntrySettings> section which controls the documentation entries in your templates. Each child element in the <EntrySettings> defines the settings for one documentation entry type.

Tag Names

The pre-supplied elements represent the entries that Atomineer supports directly, such as <summary> and <param>. While Atomineer works with "summary" internally, you may wish to use a different name for this entry – for example, JavaDoc format uses the name "brief" for this entry type. To control this, we add an attribute **_tagName="brief"** to the Xml element, like this:

```
<EntrySettings>
    <summary _tagName="brief" />
</EntrySettings.
```

(note the leading underscore on the attribute's name).

Now, when Atomineer parses or writes out a documentation comment, it will use "brief" rather than "summary" for that entry's tag.

Tag Aliases

To convert legacy documentation to our standardised format we can also specify any number of aliases for this tag, so we can ensure that (for example) "description", "summary", "brief", "details" are all treated as "summary" like this:

```
<EntrySettings>
    <summary _tagName="brief" _aliases="description,summary,brief,details" />
</EntrySettings>
```

Custom Documentation Entry Types

As well as configuring the built-in entry types, we can create our own custom entries too. For example, to use a new “stakeholders” entry (with an alias of “owners”) to our documentation comments, we just add an entry like this:

```
<EntrySettings>
    …
    <stakeholders _tagName=”stakeholders” _aliases=”owners”
</EntrySettings>
```

Other Formatting Controls

In addition, we can add special attributes to control how each of these entry types is formatted by Atomineer:

`_tagName="name"`		This is used to change the name of the documentation entry tag that is parsed and output by Atomineer. For example, Atomineer normally generates 'exception' entries, but some people prefer these to be called 'throws'. Adding **<exception _tagName=“ throws”>** to the template makes Atomineer output the entry using your preferred name.

`_aliases="name1,name2,..."`		This attribute contains a comma-separated list of aliases for the tag - for example, **<exception _aliases=“throw, throws”/>**.
		
		When parsing existing comments, Atomineer will then treat all of these names as aliases for the given entry type, and they will be converted to the new name.
		
		Note that it’s not necessary to add the element tag or the _tagName to the list of aliases, i.e. use:
		`<exception _tagName=”throws” _aliases=”throw”/>`
		Rather than
		`<exception _tagName=”throws” _aliases=”exception,throws,throw”/>`

_minTagLength="12"		(doxygen) This attribute forces Atomineer to pad out all entries of this type to the given number of characters. There is a global preference of the same name which can be applied to line up all entry tags to a given column-width, but with this template attribute you can customise the formatting more finely by overriding the global default on a per-entry basis.
_style="multiline"		For standalone entries such as 'summary' and 'returns', by Atomineer can format the text as a single-line entry if possible - this is controlled by the global option 'Keep short entries as a single line'. However, if you wish to control this behaviour on a per-entry basis, you can add this attribute to force the entry to always use the multi-line format. This attribute would normally be applied to all 'summary' templates, giving more space by default to the summary, but allowing 'returns' to remain in the compact single line style.
_wordwrap="true"		**true** = (default) If the word wrapping preference is enabled, word-wrap the text in this element. **false** = Ignore the preference and disable word-wrap in this element.
_punctuate="true"		**true** = (default) Add punctuation at the end of the element if it does not appear to end in punctuation. **false** = Do not add any additional punctuation.

`_verbatim="true"`		**true** = Copy the text from this element verbatim (no word wrap, no punctuation, no header on each line, etc.) **false** = (default) Format the text from this element, using line headers and word wrap as configured.
`_createOnly="false"`		**true** = This entry will be added by Atomineer when it creates a new documentation comment, and will be preserved if present when a comment is updated. However, if you manually delete the entry from the comment Atomineer will not add it back in when an existing comment is updated. **false** = (default) The entry will be added by Atomineer when creating and updating comments.
`_optional="false"`		**true** = This entry is "legal", and should be formatted to the given position in the final documentation comment, but *should not be added by Atomineer if it is missing*. **false** = (default) This entry should be added by Atomineer if it is not present. This attribute can also use a conditional expression of the form 'variable == value1,value2,value3' or 'variable != value1,value2,value3'. These expressions compare the contents of the given variable with the comma-separated list of values, and return true (==) or false (!=) if there is a match. Note that the values to match can use the # wild-card in the same manner as <If> commands in the auto-doc rules.

		For example, to suppress generation of an entry on methods that are overrides or static, use _optional="specialType == override,static" ...or to suppress generation of an entry for Test methods, you could use _optional="methodName == #Test"
`_copyFromBase="false"`		If you use the Atomineer option to copy base class documentation to derived class methods, entries in the base class will (by default) be copied to the derived class unless there is a matching entry already present. However, in some circumstances you may wish the derived class to *not* inherit the base documentation for that entry (usually these are rare cases where you end up with both a generated entry and one duplicated from the base class). In this case, add _copyFromBase="false" to your template's entry to suppress the duplication of the entry from base class documentation.
`_blankEntry="text"`		In the case of auto-generated entries such as param and exception, the standard is to have no entries if there are no params/exceptions to document. However, some companies require that an entry is retained, as in <param>None</param> If you specify a _blankEntry, this text will be used to create an entry in these circumstances.

Code-Element Templates

The **<EntrySettings>** element is followed by templates for the different comment blocks you will generate.

For example, here is a template for a Method, and the comment that it generates:

```
<method>
    <summary/>
    <_/>
    <remarks>%user%, %date%</remarks>
    <_/>
    <exception/>
    <_/>
    <param/>
    <_/>
    <returns/>
</method>
```

```
////////////////////////////////////////////////////////////////////////////////
/// <summary> An example method for documentation. </summary>
///
/// <remarks> Jason, 12/3/2010. </remarks>
///
/// <exception cref="ExampleCodeException"> Thrown when we feel like it. </exception>
///
/// <param cref="firstParameter">   The first parameter. </param>
/// <param cref="secondParameter">  The second parameter. </param>
///
/// <returns> The result. </returns>
////////////////////////////////////////////////////////////////////////////////
```

When creating new comments, Atomineer builds content for the most common entries for you. The position of the following XML elements specifies where in the comment the entry (or block of entries, in the case of 'exception' and 'param') will be placed.

```
<summary/> <remarks/> <exception/> <param/> <returns/> <value/>
```

The templates control the following aspects of the block's layout:

Entry order	When updating an existing comment, Atomineer will re-order the entries it finds to match the order given in the template, ensuring consistency of commenting across your project(s).
'Legal' entries / Custom entries	The XML elements in your template are not limited to the standard ones that Atomineer knows. You can add any custom entry types that you require, just by adding elements to the XML template, e.g. `<design-document/>` **However,** if you wish to use additional entries like this in your documentation you must define them for Atomineer in your templates so that Atomineer knows which code elements the entry is 'legal' for, and how it should be positioned and formatted within the comment block. Therefore, Atomineer will automatically **delete** any documentation entries that are not explicitly defined in your template for the code element being commented.

	If you are generating external documentation with a tool such as SandCastle or Doxygen, you will need to configure your documentation generator to handle these additional entries correctly too. *Note: Atomineer will automatically add every entry that is defined in your template, unless you also set the **_optional="true"** attribute for this entry type in the <EntrySettings> block – see above.*

Default Body Text	You can specify the text that is used for the "body" of a doc comment entry within the content of the XML element like this:
	
	This text can include Atomineer variables; in this case %user% and %date% will inject the current user's name and the date into the default text.
	
	`<author> This method was written by %user% on %date%. </author>`
	
	The same syntax can be used for OS Environment variables too:
	`<created-by> %USERDOMAIN%\%USERNAME% on %COMPUTERNAME%. </created-by>`
	
	This text can also include newlines and indentation (just type verbatim what is needed - However, note that this may be removed by the word wrapping feature depending on how it is configured).
	
	`<remarks>`
	`  TODO - fill this in`
	`</remarks>`
	
	Note that for auto-generated entries (summary, params, etc), Atomineer will generate its own body text, so the templated body text will be ignored in these cases.

Blank Lines	The special element **<_/>** tells Atomineer where to add a blank line between entries (if the global preference "Add blank lines between blocks of entries" is disabled, these elements are ignored) Note that multiple blank lines will be merged, and any leading/trailing blanks (on the entire block) will be stripped, regardless of what is in the template. Note: this only adds blank lines between entries. To add whitespace within an entry template (e.g. newlines and indentation) just add it verbatim to the XML file. Note however that features like word wrap may still remove the newlines in the final output.

- **Note** that most changes to rules and preferences will take immediate effect in Visual Studio, but if you add new custom entries it will be necessary to **restart Visual Studio** before the templates are picked up by Atomineer.

More Template Entry examples

`<summary>`	Place the summary first, using Atomineer-generated content
`<remarks> No comment </remarks>`	Place remarks next, containing the text "No comment"
`<_/>`	Add a blank line
`<info author="%user%"/>`	Add custom entry: <info author="Jason Williams"/>
`<author> %user% </author>`	Add custom entry: <author> Jason Williams </author>

Adding "TODO" items to the Visual Studio Task List window

You can add TODO, HACK, UNDONE entries to be displayed automatically in the Visual Studio Task Window. However, there are some limitations:

C#	Visual Studio doesn't recognise task comments that directly follow an XML start tag, so you must place the TODO on a blank line:
	
	`<remarks>`
	`TODO: Don't forget to fill in the remarks!`
	`</remarks>`

C++	Visual Studio ignores task comments inside DocXml /// blocks. You must use the _verbatim="true" tag to emit a normal comment into the middle of the documentation block.
	
	Note that the indentation used in your template XML is significant, so you must set the indentation of the element's text carefully.
	
	`<remarks _verbatim="true">`
	`// TODO: Don't forget to fill in the remarks!`
	`</remarks>`

Special attributes for code-element templates

The following attributes can be added to the main template xml element (<file>, <method> etc) to control how this type of code element is documented:

`_filetypes=".h.hpp"`	Target a template to specific set of file extensions, so you can use a different header style in .h and .cpp files, for example. The first template that matches the filetype will be used, so this must precede any file template that doesn't specify any specific filetypes.
	
	`<method filetypes=".cpp">`
	`... special template for use in C++ source files`
	`</method>`
	
	`<method>`
	`... catch-all template for all other file types`
	`</method>`

`access="[AccessLevels]"`	If specified, this attribute overrides the global 'Restrict documentation by access level' preference, allowing finer-grained control over what is or is not documented based on its access levels. For example to only allow methods to be documented if they are public or internal, modify the <method> element like this:
	
	`<method access="public,internal">`
	`...`
	`</method>`
	
	You can also use access="all" to override the default to use an unrestricted access level, or access="none" to totally disable documentation of that type of code element. If an access level is applied to a property, then it will affect the accessors independently (i.e. for { public get; private set; } with private access suppressed, Atomineer will document the property as if it only has a 'get'.

File Header and Footer Templates

The <file>, <filefooter> (C#, C++, C) and <file-vb>, <filefooter-vb> (Visual Basic) templates are different from regular comment blocks because you aren't documenting a code element. Atomineer offers two options:

1. If this template includes any embedded XML elements, it will be treated as a regular Documentation XML comment, and will thus take on the separators and other style configured for your comments. If you apply Add/Update Doc Comment to a file header in this format, it will be updated and word-wrapped just like any other doc comment.

2. If there are no embedded XML elements, this template is treated as a block of freeform text - WYSIWYG. In this case, you have the freedom to use any text you wish, but Atomineer will not update existing comments – in this case it will delete any existing comment and regenerate it.

In both cases, you can insert any of the global variables (in the table above) or the following special file-header variables:

%fileDescription%	Description of this file as generated by the <File> autodoc rules (see below).

Below are examples of different styles of file comment that you may find useful:

XML comment

If you include one or more XML elements in the comment, Atomineer treats the template as a normal XML template, and will update existing comments. The default XML template looks like this:

```
<file>
    <prototype>%projectpathname%</prototype>
    <_/>
    <copyright file="%leafname%%extension%" company="%company%">
        Copyright (c) %company% %year%. All Rights Reserved.
    </copyright>
    <_/>
    <summary/>
</file>
```

Default plain-text Atomineer header (for C#, C++, C, typescript, javascript, java)

```
<file>
    // file:      %projectpathname%
    //
    // summary:   %fileDescription%
</file>
```

Default plain-text Atomineer header (for Visual Basic)

```
<file-vb _separators="false">
    '-------------------------------------------------------------------------------------------
    ' file:       %projectpathname%
    '
    ' summary:    %fileDescription%
    '-------------------------------------------------------------------------------------------
</file-vb>
```

Free-form text

This template simply contains the verbatim text you want in the final comment, with %variables% as needed. If you wish to include XML tags in this style of comment you must use `<` and `>` to represent the < and > characters.

```
<file>
    // project:   %project%
    // file:      %projectpathname%
    //
    // summary:   %fileDescription%
    //
    //            %copyright%
    //
    //            Date        Developer        Change
    //            %date%    %user%    Created
</file>
```

StyleCop-compatible Xml-based file header

Note that this uses < and > so that it will be treated as plain text rather than XML (You will also need to set up the %useremail% user-variable appropriately in the UserVariables section to support this example)

```
&lt;file&gt;
    // &lt;copyright file="%leafname%%extension%" company="%company%"&gt;
    // Copyright (c) %year% All Rights Reserved
    // &lt;/copyright&gt;
    // &lt;author&gt;%user%&lt;/author&gt;
    // &lt;email&gt;%useremail%&lt;/email&gt;
    // &lt;date&gt;%date%&lt;/date&gt;
    // &lt;summary&gt;%fileDescription%&lt;/summary&gt;
&lt;/file&gt;
```

More Options

Further control of file header/footer comments can be achieved using the following attributes:

`_filetypes=".h.hpp"`	Target a template to specific set of file extensions, so you can use a different header style in .h and .cpp files, for example. The first template that matches the filetype will be used, so this must precede any file template that doesn't specify any specific filetypes.
`_separators="false"`	By default, Atomineer will use your configured Atomineer comment style (separators) for the file comment. For full control of the text that is inserted, this can be disabled so that the text can be treated verbatim.
`_addfooter="true"`	In a file header, this will instruct Atomineer to automatically add the file footer as well. This is handy if you use a paired header/footer (such as an #ifdef...#endif in a header file)

Note: Due to the number of options available, achieving the exact file header you want can sometimes be a bit tricky, so if you would like help, just email support with an example of the header text you wish to achieve, and we'll be happy to help you design the Templates.

Conversions Section

Atomineer is capable of converting between different comment styles/skins and also between DocXml, Doxygen, QDoc and JavaDoc comment formats.

Note that at this time, the primary conversion process that is supported is Doxygen/JavaDoc to DocXml, simply because there has so far been no demand to convert DocXml the other way. However, basic conversions are already in place to convert from DocXml if required. Please email us if you encounter any mark-up that is not converted well, and where possible we will try to upgrade Atomineer to support your conversion needs.

To convert between Doc Comment styles/formats, you may do any of the following:

1. To convert between different block format "skins", set the Alt-Separator, Alt-LineHeader and Alt-SeparatorB preferences in your Prefs.xml so that Atomineer knows what the 'old' comment format looks like. (See the User Manual for details)

2. Set your Doxygen or DocXml templates up to indicate the 'legal' entries in your new comment format, and how they should be ordered within the new comment block. Any entries that have the same tag in both old and new formats (e.g. param -> param) will be automatically 'converted' (reformatted in the new style). Any entries that are not considered 'legal' by Atomineer will be marked as 'deleted' with a ### prefix.

3. Common Doxygen/DocXml mark-up is automatically converted. e.g.

```
\c ClassName          ->          <see cref="ClassName"/>
\code ... \endcode    ->          <code> ... </code>
```

4. Where an old entry should be removed entirely, add an entry like this within this <Conversions> section:

```
<ingroup convertTo=""/>            - delete all '\ingroup' entries
```

5. Where an old entry needs to be renamed, add an entry like this:

```
<author convertTo="programmer"/>   - convert all '<author>' entries to '@programmer'
```

6. A special pair of commands can be used to convert simple plain-text lists into a set of tags, e.g.

   To convert:

   ```
   @references
        - \c MyClass
        - \c BaseClass
   ```

   …into a set of separate entries, you can use two forms:

   This form:

   ```
   <references convertEachLineTo="seealso cref" stripPrefix="-" />
   ```

will produce:

```
<seealso cref='MyClass'/>
<seealso cref='BaseClass'/>
```

…while:

```
<references convertEachLineTo="seealso" stripPrefix="- \c" />
```

will produce:

```
<seealso>MyClass</seealso>
<seealso>BaseClass</seealso>
```

AutoDoc Section – Commands

Atomineer's automatically generated documentation is built by executing the rules in the <AutoDoc> section. This is broken into sub-sections for all the main documentation entries (classes, methods, parameters, returns, etc.) plus an additional <WordExpansions> section that defines abbreviations and the text that they should be expanded into.

The rules for the appropriate entry type are executed as a script. There are a few very simple commands:

<If> command

This command evaluates a set of conditions. If any condition is true, the command is executed. Each condition tests a variable using an XML attribute, where the name of the attribute is the variable to test, and the attribute value to test is a string to test against. The matches are case insensitive, and multiple tests on a single variable can be combined by using a comma as a logical 'or':

```
<If name="vehicleSpeed,carSpeed,carVelocity" .../>
```

Different variables can be tested simultaneously by using several attributes. These are combined in a logical 'and' operation. i.e. the following only matches a method called 'GetIndex' that returns an 'int':

```
<If name="GetIndex" retType="int" .../>
```

A special wildcard '#' can be used to match any sequence of characters. It can be used in the following ways: #text, text#, te#xt, #text#. If a string with one wildcard is matched, then Atomineer sets the variable %match% to the matched text. If two wildcards are used, the matches are placed into %match1% and %match2%.

The <If> condition can also test if a variable is defined. This is useful for determining if an indexed argument or meta-data attribute is available for the current code element. As above, a comma-separated list of variables can be tested and the condition will evaluate true if one or more of them are defined:

```
<If defined="argName3"              desc="This method has a 3rd Argument"/>
<If defined="attr-conditional"      desc="This method is marked Conditional"/>
<If defined="attr-description,attr-information"
                                    desc="This has a Description or Information attribute"/>
```

If a string is matched, Atomineer appends the text from the desc="" attribute (if present) to the output description. This text can include variables which will be expanded, for example:

```
<If name="#Width"       desc="The width of the %containingClass%'s %match%." />
```

If the <If> command has child elements, these are then executed, so 'if's can be nested to make the rules more efficient or achieve 'and' logic.

```
<If containingClass="#Rectangle,#Triangle,#Circle">
    <If name="#Area"  desc="The area of the shape." />
</If>
```

Normally, when a match is made, Atomineer will exit the script and use the description that has been built. However, by setting the attribute *continue="y"*, you can instruct Atomineer to continue executing commands. Any text you set in desc will be appended to allow several phrases to be concatenated to build a final output. For example, this:

```
<If isPointer="y"             continue="y" desc="null if it fails, else "/>
<If sName="Clone,Duplicate"   desc="A copy of this object"/>
```

...will generate the description "null if it fails, else a copy of this object" for a method called Clone that returns a pointer type, or just "A copy of this object" if the method returns a reference type. Note that Atomineer corrects the capitalisation to make the sentence structure correct.

Note that as these commands are executed in the order they are defined, you must place more-specific tests before less-specific ones, so that they match correctly.

In a similar way you might wish to add a suffix to augment the main description. Suffixes can be added by replacing the desc="…" attribute with a suffix="…" instead.

```
<If attrs="NotNull"           suffix=". This parameter cannot be null"/>
```

Generally the rules for adding suffixes should be placed before the main rules that they modify (as when a rule is matched it normally causes rule execution to halt, which would stop any following suffix processing from occurring). Because of this, suffix="…" automatically implies continue="y".

<IfNot> command

These work exactly the same way as <If> commands, but the logic is reversed - they only execute if all conditions evaluate to false.

<Set> command

This command is unconditional - It is always executed, and simply sets one or more variables or the ‘desc’ or ‘suffix’. In most cases this is used for a 'catch-all' rule at the end of the rules script, as you usually set an unconditional value for ‘desc’ and processing then ends:

```
<If name="Equals">
    <If numArgs="1"  desc="Specific description for numArgs=1 case"/>
    <Set desc="Catch-all description for all other cases"/>
</If>
```

However, it can also be used to create and alter internal variables, to change the outputs of following rules. In this case you need to use the continue="y" attribute so that rules processing continues after the Set command, as well as setting any variables needed. For example, the following sets the isOptional variable to ‘y’:

```
<If type="_optional #">
    <Set isOptional="y" continue="y" />
</If>
```

The text within the Set attributes can also use any %variables% and the processing commands that can be embedded within them:

```
<Set upperName="%name:UCase%" continue="y" />
```

<Execute> command

This acts like a subroutine call (or a #include) - it executes the given AutoDoc section and (if it doesn't return a description) continues execution at the command following the Execute.

```
<Execute rules="Variables"/>
```

Special 'abortComment' option

Atomineer can optionally only apply comments to code elements with given access levels. This is usually used to stop documentation being generated for private members. A special feature allows you to extend this 'do not document' handling to any specific code element. This is done simply by adding an attribute to an If, IfNot or Set condition:

```
<If access="private" abortComment="yes"/>
```

If this attribute is hit while processing, AddDocComment exits immediately without inserting a comment at all.

AutoDoc Section – Using Variables

Variables may be used as tests in If commands by using the **varName="test"** attributes described above.

Variables may also be embedded in the desc="" text to insert special text into the output. Each variable is delimited by % characters (use %% to insert a literal % symbol). Any variable (except %ip%) can be used as many times as you like within a description.

The following variables are defined:

`%ip%`	Insertion Point. The place where the cursor will be left after the comment is inserted. Use only one %ip% in any doc comment.
`%match%`	For wildcard matches, the %match% command inserts the text that matched the wildcard. (In cases involving two wildcards, the variables %match1% and %match2% are used).
`%???%`	Depending on the context, more variables will exist. In most cases "name" will return the full name of the code element being documented, and "sName" will return it in sentence form (with spaces between words, and abbreviated words expanded to their full name), but some contexts will provide much more information - For example, Methods provide %name%, %sname%, %retType%, %numArgs%, %argName1%, %argType1% ...

Variables used to insert text can have additional processing commands applied to that text. Each command is of the form "`:command`" and you can append as many commands as you like. e.g. `%match:Sentence:StripLastWord:SCase:Plural%` would have the effect of converting the name "DivideByZeroException" into the text "Divide by zeroes".

The available commands are:

`(no suffix)`	Use the text verbatim `DivideByZeroException`
`:Sentence`	Insert spaces between words, retaining existing capitalisation `Divide By Zero Exception` Conversion to sentence form also expands any abbreviated words, e.g. "InfoMgr" becomes "Information Manager"
`:FirstWord`	The first word of the text `Divide`
`:LastWord`	The last word of the text `Exception`
`:FirstCSV`	The first value in a comma-separated list
`:LastCSV`	The last value in a comma-separated list (used rarely for extracting the 'result' type parameter from a generic/template type-list)
`:StripFirstWord`	Everything except the first word `ByZeroException`
`:StripLastWord`	Everything except the last word `DivideByZero`

:LCase	Lowercase the text `Dividebyzeroexception`
:UCase	Uppercase the text `DIVIDEBYZEROEXCEPTION`
:WCase	Uppercase the first letter of each word in a (space-separated) sentence `This Is A Sentence`
:SCase	Sentence-case the text (uppercase the first character, lowercase the rest) `This is a sentence`
:Camel	Camel-case a multi-word symbol (lowercase the first character of the first word, uppercase the first character of the remaining words) `thisIsASentence`
:Pascal	Pascal-case a multi-word symbol (uppercase the first character of each word in the symbol) `ThisIsASentence`
:Underscored	Converts spaces and file-path separators into underscore characters. e.g. “Exceptions\Divide By Zero” would become “Exceptions_Divide_By_Zero”. This is usually used with UCase to generated C++ constant style naming like “#ifndef EXCEPTIONS_DIVIDE_BY_ZERO” for creating include-only-once implementations in header files.
:Plural	Attempts to pluralise a word using heuristics for English. This will give results like: square squares box boxes party parties …usually it works well, but in some cases the heuristics will fail: goose gooses `DivideByZeroExceptions`
:Leaf	The 'leafname' in text assuming that it is a filename or a fully qualified type name. e.g. 'System.Exception' would return only 'Exception'.
:UnixDir	Any Windows path separator characters ‘\’ will be replaced by Unix path separators, ‘/’.
:StripGen	For generic/template types such as List<int> this returns the main type without its type-params, i.e. 'List'
:SafeGen	Converts the <> characters from a template/generic type into {} so it can be safely embedded in XML documentation, e.g. ‘List<int>’ becomes ‘List{int}’
:AsSee	Generates a complete ‘see’ entry using the variable value as the content for the cref. For example, for a class called MyList<T>, the variable %containingClass:AsSee% will expand to: `<see cref='MyList{T}'/>`

:Match(REGEX)	Attempts to match the given regular expression. The output value will be the first match for the regex within the input. For example, processing a method name of "Polygon3D" with %methodName:Match(\d+)% would return "3" , the matched digit(s). (Note that the expression cannot include "%" or "):" as these are the two terminators that Atomineer uses to extract the variable and commands from the rules)
:MatchIC(REGEX)	As :Match but using **case insensitive** matching.
:Strip(REGEX)	Attempts to match the given regular expression. All matches for the regex within the variable will be removed. Examples: • Processing a method name of "Polygon3D" with %methodName:Strip(\d+)% would return "PolygonD" by removing all matched digit(s). • Strip the start of paths to header files in an 'include' folder with %file:Strip(^.*include\\)%. This would convert "C:\MathApp\include\math\matrix.h" into "math\matrix.h" • … and if you require Unix-style folder separator characters for external tools like doxygen, you can use a pair of commands: %file:Strip(^.*include\\):UnixDir% to produce "math/matrix.h".
:StripIC(REGEX)	As :Strip but using **case insensitive** matching.
:Default(TEXT)	While processing, any undefined variable expands to an empty string, and sometimes the processing commands above can result in an empty string. In this case 'default(…)' will replace the empty string with arbitrary text. You can use any text for the default as long as it does not contain "%" or "):" e.g. %arg2:Default(No parameter)%

Most of these commands are of little use on their own, but once combined with Sentence, (e.g. :Sentence:SCase or :Sentence:LCase) can produce much more useful or readable text output.

Special handling for better quality output

The following special handling is provided to improve the quality of autodoc output. These methods use heuristics to detect and correct the English, which do not always succeed, but do improve the output text in the majority of cases:

- Plural command can be used to attempt to pluralise a word as it is expanded.

  e.g. "Box" → "Boxes"

- Any text that has the word "a" in it is processed to replace the "a" with "an" or "the", so it is generally better to use 'a' in preference to 'an' or 'the' in your rule description text. Typical results of this are:

  "Report a warning"

"Report an error"
"Report the warnings"
"Report the errors"

AutoDoc Section - Documenting specific Code Elements

Various types of code elements (e.g. classes, exceptions, methods) require different approaches for documenting them. The AutoDoc section is thus divided into a number of type-specific sections, which are applied as appropriate to the item being documented. These are described below.

The global variables (listed above) and any User variables you have set can be used within these rules. In addition, the following variables are always set for each code element being documented:

%name%	The name of the item being documented (e.g. 'm_SomeVariable', 'some_variable')
%sName%	The name of the item being documented, converted to a 'sentence' form (words separated by spaces. Anything that looks like a common prefix (e.g. 'm_') will be removed. e.g. 'some variable')
%sNameRaw%	As %sName%, but without any prefix removal (e.g. 'm some variable').

In addition to the rules described above, each of these sections can add an optional prefix or suffix to the generated description text. These are set as attributes on the section, e.g.

```
<Exceptions prefix="Thrown when " suffix="."> 
  ...
</Exceptions>
```

The special-cases for the various code elements that can be documented are handled in a number of sub-sections of the AutoDoc section, and are described in detail below:

<File>, <Namespace>, <Interfaces>, <Exceptions>

Rules used to generate the description used in (respectively):

- for File header comments (%fileDescription%)
- for Namespace comments
- for Interfaces
- for any Exceptions thrown in a method
- for any Generic Type Parameters in a class or method

For these sections, no additional variables are available.

<TypeParameters>

Rules used to generate the description for generic/template type parameters. The following special variables can be used in these sections:

%isOut%	"y" or "n", indicating if the type parameter is covariant ('out').
%isIn%	"y" or "n", indicating if the type parameter is contravariant ('in').

<Enums>

Rules used to generate the description for enumerated types. The following special variables can be used in this section:

%specialType%	The type of the enum (enum, bitfield)

<Typedefs>, <Variables>, <Parameters>

Rules used to generate the description for parameters, variables and C++/C typedefs. (Note: The Execute Command is used to 'include' (duplicate) the Variables rules into both the Typedefs and Parameters sections). The following special variables can be used in these sections:

%type%	The type of the variable (const int*)
%typeBase%	The type of the variable, not including any modifiers (int)
%index%	The 1-based index of the variable within the method's variable list
%word1%	The first word of the variable (often used for prefixes, e.g. m_, lpsz, etc)
%coreName%	The name with the first word removed (to allow processing of prefixed names)
%isPointer%	"y" or "n", indicating if the variable type includes a "*" or "^"
%isReference%	"y" or "n", indicating if the parameter type includes an "&" or "ref"

In addition, the following variables are available only for the Variables section:

%isEvent%	"y" or "n", indicating if the variable is an event declaration

In addition, the following variables are available only for the Parameters section:

%isOut%	"y" or "n", indicating the parameter type includes an "out"

%isOptional%	"y" or "n", indicating if the parameter is optional
%methodName%	The name of the method for which this is a parameter
%numArgs%	The total number of arguments for the method
%attrs%	A semicolon-separated list of any attributes found on the parameter (e.g. a [NotNull] attribute would result in attrs="NotNull")

<Classes>

Rules used to generate the description for classes, structs and records. The following special variables can be used in this section:

%numBases%	The number of base classes/interfaces this class inherits from/implements
%baseNameX%	Base class/interface name, where X is the 1-based index of the base (e.g. baseName1, baseName2)
%isStatic%	'y' or 'n', indicating if this class is static
%isPartial%	'y' or 'n', indicating if this class is partial
%isSealed%	'y' or 'n', indicating if this class is sealed
%isAbstract%	'y' or 'n', indicating if this class is abstract
%isGeneric%	'y' or 'n', indicating if this class is generic
%isTemplate%	'y' or 'n', indicating if this class is a template
%attr-????%	When a class has one or more meta-data Attributes assigned to it, each one generates an attr-???? variable, e.g. attr-conditional, attr-description. The value of the variable is the list of parameters to the attribute (e.g. the description text from a DescriptionAttribute). This is usually used with the defined='attr-description' syntax of the <If> command (see above) to detect when attributes are applied to a class, and add appropriate notes to the documentation.

<Methods>

Rules used to generate the description for methods (including VB subs and functions), constructors, destructors, finalisers, properties, indexers, delegates and event handlers. The following special variables can be used in this section:

%methodType%	The type of the method (one of: method, property, indexer, delegate, eventhandler)
%methodName%	(alias for %name%) The name of the method
%rawMethodName%	The name of the method, including any generic type specifiers
%specialType%	Extra info on the method: normal, static, inline, abstract, virtual, override
%retType%	The return type for this method (const int*)
%retTypeBase%	The core return type for this method, without modifiers (int)
%numWordsInName%	The number of words detected in %name%
%numArgs%	The number of arguments for this method
%argTypeX%	Argument type, where X is the 1-based index of the argument (e.g. argType1, argType2)
%argTypeBaseX%	Core argument type (without modifiers), where X is the 1-based index of the argument (e.g. argType1, argType2)
%argNameX%	Argument name, where X is the 1-based index of the argument (e.g. argName1, argName2)
%getSet%	(For methodType="property" or "indexer") the appropriate "Gets", "Sets", or "Gets or sets" text for this property/indexer
%eventSender%	(For methodType="eventhandler" only) the sender of the event
%eventType%	(For methodType="eventhandler" only) the type of the event
%attr-????%	When a method has one or more meta-data Attributes assigned to it, each one generates an attr-???? variable, e.g. attr-conditional, attr-description. The value of the variable is the list of parameters to the attribute (e.g. the description text from a DescriptionAttribute). This is usually used with the defined='attr-description' syntax of the <If> command (see above) to detect when attributes are applied to a method, and add appropriate notes to the documentation.

<MethodReturns>

Rules used to generate the description for the return values for methods (VB functions). The following special variables can be used in this section:

%methodType%	The type of the method (one of: method, property, indexer, delegate, eventhandler)

%type%	The return type for the method (const int*)
%typeBase%	The return type for the method, not including any modifiers (int)
%isPointer%	"y" or "n", indicating if the variable type includes a "*" or "^"
%isReference%	"y" or "n", indicating if the parameter type includes an "&" or "ref"
%attr-????%	Attributes applied to the method (as for <Methods>, see above)

<WordExpansions> and <Prefixes>

This is a special set of rules that are executed to detect and expand abbreviations, and remove prefixes. They are called for each word in turn in a generated description in order to allow abbreviations to be expanded. The 'desc' that they generate should usually be given in lowercase to create tidy results. For example:

```
<If name="ack"      desc="acknowledge"/>
<If name="fg,fore"  desc="foreground"/>
```

To delete unwanted words (such as library naming prefixes) from documentation, simply replace the word with desc="-" to delete it from the final documentation. That is, a member like 'lpszUserName' might be documented as 'the lpsz user name'. By removing the prefix 'lpsz', this can be cleaned up to 'the user name'.

The difference between the handling of the WordExpansions and Prefixes sections is that for Prefixes, only the first word of any symbol is processed; all words are processed for abbreviation expansions.

Automated Comment Conversion

Atomineer can use its unique ability to parse multiple formats of comments to convert from legacy formats to a new style. Any existing documentation in a valid XmlDoc, Doxygen, JavaDoc, QDoc, JSDoc, or JSDuck format (or any other sufficiently similar format) can be converted in this way.

In addition, if your legacy comments are not in a form that Atomineer recognises directly, it is also possible to apply custom pre-processing of the comment text during the parsing process to convert your comments into a form that Atomineer can successfully parse.

In this way it is possible to automatically convert any machine-readable legacy format to an Atomineer-supported style, and this process can even be applied to the comments throughout an entire project or solution.

As this conversion process is a relatively rarely-used option, it must be manually configured by opening the Atomineer **Prefs.xml** file (in your install folder, usually Documents\Visual Studio 20??\Addins\AtomineerUtils, or Documents\Atomineer) and adding the following elements to the <DocComment> section:

```
<Alt-Separator  value="/** " />
<Alt-LineHeader value=" * " />
<Alt-SeparatorB value=" */" />
```

You can set any text for the Separator (top line of the comment block), SeparatorB (bottom line of the comment block) and LineHeader (prefix for all lines within the block). If the separators and/or line prefix are not used, you must set `value="(None)"`

If you wish to convert from JavaDoc/JSDoc/JSDuck, QDoc or Doxygen format, then you must also make sure that the Doxygen command prefix is correctly configured. Look for the <Doxygen> section in the Prefs.xml file, which contains a <CommandPrefix> entry. Set this as appropriate, i.e. one of these values:

```
<CommandPrefix value="@" />
<CommandPrefix value="\" />
```

When you execute ‘Add Doc Comment’ on a comment in this alternate format, it will be converted into your configured 'primary' format. The standard tags (summary/brief, returns/return, seealso/sa etc) will be converted automatically as required, including embedded mark-up like c/see.

Any custom tags used in your old format will simply be treated as embedded text appended to the previous entry, unless they match a custom entry tag that you have added to your comment Templates - if a matching template entry tag is found, the old comment data will be converted to the new documentation style and inserted into the final block as dictated by that template. Note that you can also use the ‘_aliases’ attribute in your Template to allow conversion of the entry name (for example you may wish to change your comments from using ‘@throws’ to ‘@exception’).

Custom <Conversion> Rules

In addition, Rules.xml contains a special section, <Conversions>, which tells Atomineer how to convert the entries it finds when converting legacy comments. This can:

- Delete unwanted entries (e.g. delete '@ingroup' entirely)
- Rename entries (e.g. '@description' to '<remarks>')

Note that at this time, the primary conversion process that is supported is Doxygen/JavaDoc/QDoc to DocXML, simply because there has so far been no demand to convert DocXML the other way. However, basic conversions are already in place to convert from DocXML if required. Please email support@atomineerutils.com if you encounter any mark-up that is not converted well, and where possible we will try to upgrade Atomineer to support your conversion needs. (Also, the macro/extension pre-processing approach described below may help you to handle your conversion needs for yourself)

To convert between Doc Comment styles/formats, you may do any of the following:

1. To convert between different block format "skins", set the Alt-Separator, Alt-LineHeader and Alt-SeparatorB preferences in your Prefs.xml file so that Atomineer knows what the 'old' comment format looks like.
2. Set your Doxygen or DocXML Templates up to indicate the 'legal' entries in your new comment format, and how they should be ordered within the new comment block. Any entries that have the same tag in both old and new formats (e.g. param -> param) will be automatically 'converted' (reformatted in the new style). Any entries that are not considered 'legal' by Atomineer will be marked as 'deleted' with a ### prefix.
3. Common Doxygen/DocXml markup is automatically converted, e.g.:

```
\c ClassName                    ->  <see cref="ClassName"/>
\code ... \endcode              ->  <code> ... </code>
```

4. Where an old entry should be removed entirely, add an entry like this within the <Conversions> section:

```
<ingroup convertTo=""/>          - delete all '\ingroup' entries
```

5. Where an old entry needs to be renamed, add an entry like this:

```
<author convertTo="programmer"/>  - convert all '<author>' entries to '@programmer'
```

6. A special pair of commands can be used to convert simple plain-text lists into a set of tags, e.g. To convert:

```
   @references
       - \c MyClass
       - \c BaseClass
```

…into a set of separate entries, you can use two forms:

`<references convertEachLineTo="seealso cref" stripPrefix="-" />` will produce:

```
    <seealso cref='MyClass'/>
    <seealso cref='BaseClass'/>
```

`<references convertEachLineTo="seealso" stripPrefix="- \c" />` will produce:

```
    <seealso>MyClass</seealso>
    <seealso>BaseClass</seealso>
```

If an 'incorrect' entry is converted (such as a param or exception entry that describes a parameter or exception that is not present in the code), the entry will be converted correctly, but then added at the bottom of the new comment with '###' prepended ('deleted' entry), just as would happen if you removed that param/exception and updated an existing comment. If unwanted, just execute the Add Doc Comment command a second time to clean away these 'deleted' entries.

Note that if the old comment format diverges too far from what Atomineer expects (such as using Doxygen aliases or custom Documentation XML elements not mentioned in the Conversions rules), the conversion may not work so well - you may need to 'tidy up' the comment (manually or perhaps using a Macro/Extension pre-processing script - see below) before applying the Atomineer conversion.

For example, the following Doxygen/JavaDoc comment uses the alternate format described by the prefs XML above, with JavaDoc-style entries:

```
/**
  * @brief  This is the brief description
  * @returns true if it feels like it
  * @param exampleParam An example parameter.
  * @date 15/04/2010
  * @author Jason
  */
```

...and it is converted to the following Documentation XML (in this case, using the default Atomineer format, with the entries rearranged in the order dictated by the default template):

```
////////////////////////////////////////////////////////////////////////////////////////////
////
/// <summary>   This is the brief description. </summary>
/// <author>    Jason. </author>
/// <date>      15/04/2010. </date>
///
/// <remarks>   Jason Williams, 16/04/2010. </remarks>
///
/// <param name="exampleParam"> An example parameter. </param>
///
/// <returns>   true if it feels like it. </returns>
////////////////////////////////////////////////////////////////////////////////////////////
////
```

Note that the author and date are recognised tags and thus preserved in the resulting comment - but if these are not set as 'legal' entries in your templates, they can be removed by simply executing the Add Doc Comment command a second time to update the new comment. (Note also that the 'remarks' was not supplied, so a new auto-generated entry is added that records the author/date of the conversion). However, as you can see, Atomineer does the bulk of the conversion work for you.

Pre-processing to help Atomineer Parse your Code

You may find some cases in your codebase where Atomineer cannot convert your existing comments (e.g. if they are in a special in-house format), or cannot parse your code correctly (such as C++ code where in-house macros are used within class/method declarations).

To deal with these situations, Atomineer provides several approaches that can be used to pre-process the text of your source code before it is fed into Atomineer's parsing engine.

Regular Expression Replacements

Occasionally, Atomineer hits something that confuses it, resulting in a lower quality of automatic documentation than expected. Typical examples are when a typedef or #define macro are used to represent special types in your code - Atomineer only parses the code near the location you wish to document, so it has no way of understanding what these special code symbols represent.

When parsing class and method declarations, Atomineer can apply optional regular-expression replacements to the text before it is parsed. This does not change the code in the document being processed, but just the view Atomineer has of it. This allows users to convert in-house types, typedefs and macros into a form that Atomineer can handle better.

To add regex replacements, edit the Pre-processing Rules (in the Atomineer preferences, go to the 'Doc Advanced' tab and click the Pre-processing button) and add one or more entries like the examples that are provided - for Methods:

```
<MethodDecl>
   <Replace pattern="OurTrueFalseType" replacement="bool" continue="true"/>
   <Replace pattern="OUR_TYPE\(bool\)" replacement="bool" continue="true"/>
</MethodDecl>
```

... or for Classes/Structs/Records/Interfaces:

```
<ClassDecl>
   <Replace pattern="OurTrueFalseType" replacement="bool" continue="true"/>

   <Replace lang=”c” pattern="C_LANGUAGES_ONLY" replacement=" " continue="true"/>
</ClassDecl>
```

The 'pattern' is a standard regular expression to be matched in the source code; when it is found, each match will be replaced by the plain-text 'replacement' value. Typically the value will be a blank string “” to remove the macro entirely, or the name of an intrinsic type to replace a macro with a type that Atomineer can make more sense of.

The ‘**continue**’ attribute can be set to ‘true’ or ‘false’ to control whether or not any following Replace rules are executed if a rule causes the text to be altered. If not specified, ‘true’ is assumed.

The optional ‘**lang’** attribute can be used to restrict the rule so that it only applies to certain languages. The possible values for this are:

‘lang’ value	**Languages that the rule will apply to**
c	C, C++, UnrealScript
c#	C#
java	Java
typescript	Typescript, JavaScript, Jscript
vb	Visual Basic
php	PHP
sql	SQL
python	Python

Preprocessing with a C# Custom Extension Command

Visual Studio has a built-in extensibility model that allows end users to write their own code to automate activities within the IDE – for example, adding new menu commands. This is done by writing a "VSIX" Extension (in older versions of Visual Studio, this was achieved by creating an Add-In Package or VBScript Macro)

Atomineer can also execute these extension commands as required, so it uses this mechanism to allow you to insert custom pre-processing code as follows:

- When Add Doc Comment is executed, it locates the existing comment to be parsed (if any)
- It selects the text in the comment, and then executes your extensibility Command, which can read the selection, process the text it contains, and write back a block of replacement text.
- Atomineer then reads the new selection and tries to parse it as normal.

In this way, any legacy comment can be converted by writing a simple parser for your old format, and applying a rough conversion to make it look like a Doxygen or XmlDoc format. This conversion doesn't need to provide "perfect" output, just something that conforms enough to the syntax Atomineer expects to allow it to parse the text correctly. The main consideration is that Atomineer will try to preserve indentation from the original comment, so it's best to strip or preserve indentation during your conversion step.

Enabling this conversion requires the following steps (which are explained in detail below).

- Customise our pre-created example extension to parse and process your comments. This requires a few simple string-processing calls in C#, so the coding effort required is pretty minimal.
- If it differs from your configured style, set Atomineer to recognise this as "alternate comment style" that it will convert into your preferred style.
- Tell Atomineer the name of your extension command so it knows what to execute to convert the comments.

In order to build and debug the custom extension you will first need to install the "Visual Studio Extension Development" Workload using the Visual Studio Installer.

Next, download the source code for the extension from here:

https://www.atomineerutils.com/AtomineerCustomPreprocessorExtension.zip

Open the solution. When you build and run it, a new "experimental" copy of Visual Studio should start up (note that this is a special debugging instance which uses completely independent preferences from your normal Visual Studio, so it will act as if it is the first time you have run it. Any settings you change here will not affect your normal Visual Studio). Once it has started up you should see "Preprocess DocComment" command on the Tools menu. Create a blank document and copy this example text into it:

```
/************************************************************
* Usage:           Open the datafile for reading
*
* Return Value:    true if the file was opened successfully
*
* Notes:
*           Preconditions:
*                    The filename must have been set in Init()
************************************************************/
```

Select the entire text of this comment, and execute the Tools > Preprocess DocComment command and the comment will be converted into a more Atomineer-compatible (Doxygen) form by the custom command:

```
/************************************************************
* \brief            Open the datafile for reading
*
* \returns     true if the file was opened successfully
*
* \remarks
*           Preconditions:
*                    The filename must have been set in Init()
************************************************************/
```

If you stop debugging, in your extension solution you can edit the PreprocesssDocCommentDoxygen() or PreprocesssDocCommentDocXml()method in PreprocessDocumentationCommentCommand.cs. There are two #defines set at the top of this source file that will switch the extension to output DocXML format or Doxygen (Javadoc "@command") or Doxygen (Qt QDoc documentation "\command") comment formats.

This reads the selected text from Visual Studio's active document into a list of strings, applies a conversion process to them, and then inserts the processed text back into the active document, replacing the original text. So now you can place examples of your legacy comments into a source file, and test your conversion by selecting the comments and executing Tools > Preprocess DocComment. By setting debugger breakpoints in this method you can even single step through the execution of the command to debug it.

Once you have the conversion process working, you can integrate it into Atomineer.

First, install the extension into your normal Visual Studio instance. Quit all running instances of Visual Studio, and double-click the AtomineerCustomPreprocessorExtension.vsix file in the Debug or Release output folder for your custom pre-processor extension. (Note: To install updates you can uninstall the extension from the Extensions > Manage Extensions menu and then reinstall it, or increment the version number in the "source.extension.vsixmanifest" file in the custom solution (double click to edit this file and the Version is shown in the top right corner of the page) and then double-click the vsix to update your install.

Now a couple of simple steps are needed to configure Atomineer. If the comment format produced by the custom extension doesn't match your final block format, you will need to set up an alternate format so that Atomineer can recognise this partially-converted comment. For the example above, adding the three preference options below to Documents\Atomineer\prefs.xml will set this up (see **Automated Comment Conversion**, above, for further details about alternate formats)

```
<DocComment>
    <Alt-Separator  value="/************************************************************" />
    <Alt-LineHeader value=" * " />
    <Alt-SeparatorB value="************************************************************/" />
    ...
</DocComment>
```

Next, enable execution of the pre-processing command whenever Atomineer is about to parse a comment:

- Edit the Pre-processing Rules (in the Atomineer preferences, go to the 'Doc Advanced' tab and click the ‘Pre-processing’ button)
- Uncomment the example ConvertComment entry, and make sure it uses the correct name (as shown here for our custom extension) for the command in the “macro” attribute:

```
<ConvertComment macro="Tools.PreprocessDocComment"/>
```

Now, when you execute Add Doc Comment, Atomineer should detect the legacy comment, select the entire comment block, execute your command code to pre-process it, and then parse and update the output of your command to finish the conversion process.

- Please Note:
    - The mechanism that must be used involves selecting each comment as it is processed, causing unpleasant, flickery redraws of the code.
    - Executing the command can slow down comment processing quite significantly.
    - There is also a risk that changes made by this pre-processing code might cause unexpected problem(s) when commenting other parts of your code-base. This can be confusing and difficult to diagnose.

ⓘ It is recommended that when you have finished converting legacy comments, you disable the pre-processing Rule so that Atomineer no longer tries to execute your command – just comment the ConvertComment element back out. You may also like to uninstall the custom extension.

ⓘ If you have any trouble getting your conversion working, don’t hesitate to email us and we’ll be happy to help.

Atomineer Menu Commands

This section details the specific menu commands available to you.

Add/Update Doc Comment

Default hot-key: Ctrl+Shift+D

This is the core command of the Atomineer Pro Documentation extension. It creates or updates a Documentation Comment for any supported programming language (C#, C++/CLI, C++, C, Visual Basic, TypeScript, Java, JavaScript, JScript, Python, PHP or UnrealScript)

It will create, or parse and update, any comments that conform to the configured Atomineer Style (XML Documentation, Doxygen, JavaDoc or QDoc formats, with user customisable comment block separators (top and bottom lines to delimit the doc comment) and line prefixes).

Usage

Place the cursor in an appropriate line:

- Anywhere in the first line of the file (to generate/replace a file header)
- Anywhere in the whitespace at the end of the file (to generate a file footer)
- Anywhere in the first line of a "code element" (namespace, macro, typedef, struct, enum, class, record, interface, method, function, property or variable definition, etc) or in any blank lines or documentation comments immediately preceding the element.
- Anywhere within a normal (non-doc comment) comment block to simply reformat the text with word wrapping. (This can be used on single line // and multi-line /*…*/ comments as long as the comment is the first thing on the line. This can also be used on <!-- … --> comments in XML or XAML files)

...and execute the command. This can be done from the Tools > Atomineer menu, the context menu, or by pressing the assigned hotkey (normally ctrl+shift+D).

Description

When you execute the command, the code around the cursor will be interpreted and an appropriate header comment will be inserted just above the current code element. As much useful information as possible will be extracted from your code (and any existing documentation comment), and where possible default descriptions will be inserted to save you unnecessary typing.

Reformat Doc Comment

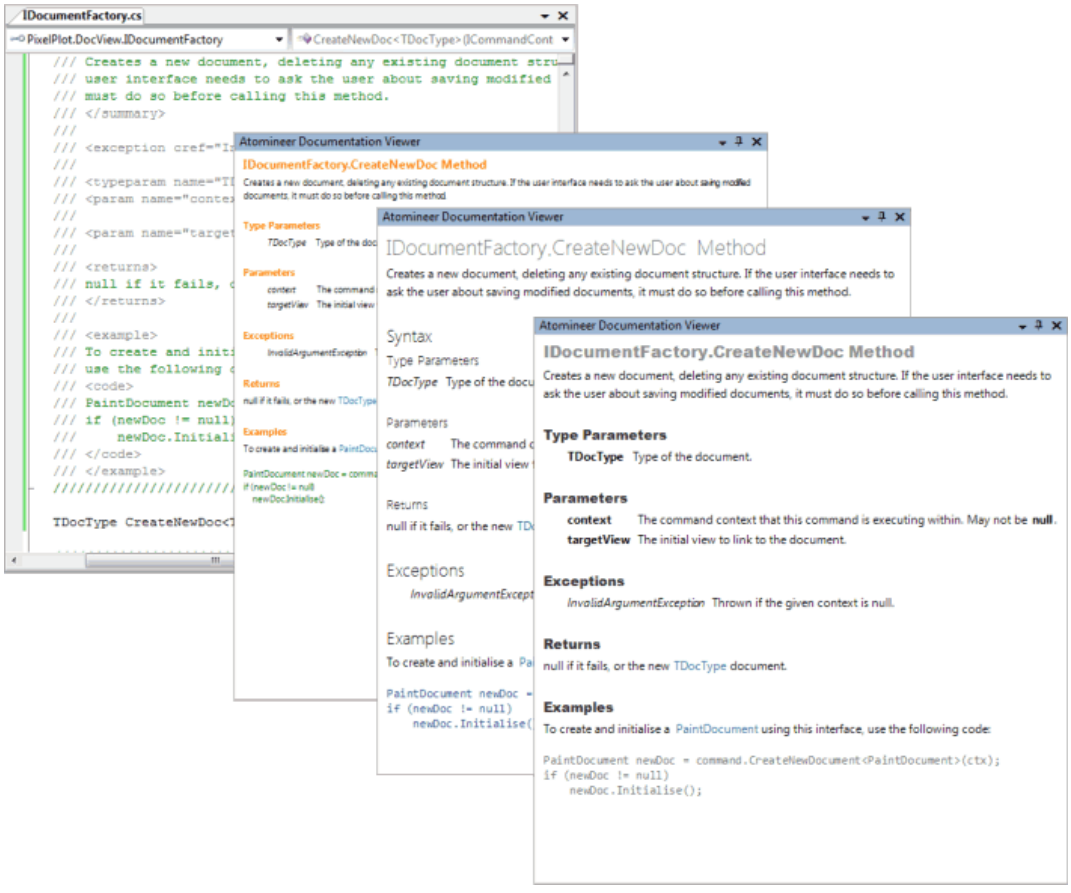
Default hot-key: **Ctrl+Shift+A, Ctrl+Shift+R**

This command works just like Add/Update Doc Comment, except that it does not generate any new entries, but simply reformats an existing documentation comment to update its style to match the current preferences and layout.

Documentation Viewer/Editor

Default hot-key sequence: Ctrl+Shift+A, Ctrl+Shift+V

This Atomineer command opens a documentation viewer/editor window. The read-only **viewer** is provided in older Visual Studio versions (2005-2013) while the **editor** is provided in Visual Studio versions from 2015 onwards. This window displays documentation comments in a human readable style *similar* to the sort of output you can generate with tools like Sandcastle, Doxygen, and JavaDoc.



If you open the viewer, it will automatically display any available documentation for the nearest code element as the text cursor is navigated through your source code. *(Please note that this feature is not currently available in C/C++ header files, nor for end-of-line comments)*

The viewer handles common documentation mark-up including the following:

- Lines that start with *, +, - are treated as bullet point lists
- Lines that end with two spaces, and blank lines, are treated as paragraph separators
- HTML elements such as

  ```
  <br/> <p> <b> <i> <strong> <h1> <ul>
  ```

- XML documentation markup including

  ```
  <para> <c> <code> <list>
  ```

The viewer display will automatically reformat to suit the window size, and can be resized and docked just like any other Visual Studio tool window.

Display Options

Click the cog icon within the viewer/editor window to change display options:

- Select from a range of pre-defined templates. These change the documentation layout and fonts used to emulate different styles that can be produced by tools like Sandcastle, Doxygen and JavaDoc.
- Scale all the text in the window to a size that suits your preferences.

Atomineer will remember your settings the next time you start Visual Studio.

Editor Features

The editor provides the ability to also edit your descriptions in place – Use “Add Doc Comment” as usual to add a new comment to a code element, and then you can click in the description text in the editor to place the cursor and begin typing.

You can edit the documentation in the source code or the editor window, and periodically any changes made will be automatically synchronised between the two.

Within the editor you can only change the description text for documentation entries, so each is added as a separate editing field – click on the entry description to place the cursor in it and begin editing. You can navigate to next/previous fields with Tab and Shift+Tab.

The editor provides buttons for:

- Cut, Copy and Paste of text within the descriptions
- Applying Bold, Italic, Code and Bullet-list mark-up to the selected text
- Adding a new optional documentation entry to the comment (see below)
- Navigating to the previous or next code element in the current source file
- Switching the editor between the normal editable mode and a locked (read-only) mode that makes it act as a viewer only, to prevent accidental changes to the comments as you read.
- Setting editor display options

Adding new entries

Optional entries may not initially be present in the documentation comment. Click the Add Entry (+) icon (or type **Ctrl+E**) to select an entry type to add, and it will be inserted in the appropriate position in the editor.

Any entries that are already present will be shown in the menu, so that you can see the full template for this code element type, but they will be disabled so that you cannot add duplicate entries.

In most cases new entries will be added immediately, but for exceptions it is necessary to fill in the exception name in a dialog first. Once entered this cannot be edited in the documentation editor, only the description text can then be changed. As Atomineer will usually add entries automatically for any exceptions thrown within the method itself, it will add "Passed when" as the start of your description text – this is how Atomineer documents exceptions that are thrown in methods that your method calls, so that it knows that the exception is thrown externally and should not be removed from the comment block.

Navigation

The Next and Previous buttons allow you to advance the editor through the code elements in the current file. If the code element is not yet documented, a documentation comment will be generated. The editor will then switch to editing the documentation comment for this new code element. In this way you can easily work your way through a source code file, adding or updating all of the documentation in a single pass.

You can work your way through the class without needing to use the mouse, by using **Alt+Up** to navigate to the previous code element, or **Alt+Down** to navigate to the next one.

Please note that this feature requires Intellisense information from Visual Studio to locate each code element. This information is only available for some languages, and usually only available while your code is in a compilable (recently compiled) state. If the information is unavailable an error dialog will be shown to inform you why the feature is disabled.

Inserting references (paramref and see)

When editing a method's description, right clicking to open the context menu in any description text will allow new 'see' references or 'paramref' references to parameters within the method (if any) to be inserted into the text. These reference entries are highlighted in the text, and if you click on them you can change the reference.

- 'Paramrefs' can be changed to any of the parameters in the method from a drop-down menu.
- 'See' references can be edited as plain text. Delete the text to remove the reverence entirely.

When the edited text is written back to your source code comment, it will use the normal format (e.g. **<paramref name="xyz"/>** or **<see cref="xyz"/>**).

Editor Preferences

Editor Preferences can be found in the Atomineer Options, in the 'Live Aids' section.

Paragraph markup format can be set to:

- Paragraphs will be written to source code as a double newline (blank line) in the source comment. This is the default style.
- Paragraphs can be enclosed within XML Documentation <para>…</para> markup.
- Paragraphs can be enclosed within HTML … markup.

List (bullet point) markup can be set to:

- Use '*', '+' or '-' characters at the start of a line to denote a bullet point list, or numbers for a numbered list.
- Use HTML markup <ul> and <li> to define bullet point lists, or <ol> for numbered lists.
- Us a simple XML markup (a <list> containing <item> entries) to define a bullet or numerical list.
- Use XML documentation markup a <list> containing <item> entries. Each <item> contains an optional <term> and a <definition> to define lists of bullet or numbered points.

The formats look like this:

Bullet list	Numbered list

`* One`
`* Two`
`* Three`
	`1. One`
	`2. Two`
	`3. Three`

`<ul>`
`  <li>One</li>`
`  <li>Two</li>`
`  <li>Three</li>`
`</ul>`
	`<ol>`
	`  <li>One</li>`
	`  <li>Two</li>`
	`  <li>Three</li>`
	`</ol>`

`<list type="bullet">`
`  <item>One</item>`
`  <item>Two</item>`
`  <item>Three</item>`
`</list>`
	`<list type="number">`
	`  <item>One</item>`
	`  <item>Two</item>`
	`  <item>Three</item>`
	`</list>`

`<list type="bullet">`
`  <item>`
`  <description>One<description>`
`  </item>`
`  <item>`
`  <description>Two<description>`
`  </item>`
`  <item>`
`  <description>Three<description>`
`  </item>`
`</list>`
	`<list type="number">`
	`  <item>`
	`  <description>One<description>`
	`  </item>`
	`  <item>`
	`  <description>Two<description>`
	`  </item>`
	`  <item>`
	`  <description>Three<description>`
	`  </item>`
	`</list>`

It is also possible to build definition lists like this:

- **Width** – The width of the house.
- **Height** – The height of the house.

This can be done in simple markup, HTML and XML by adding HTML <b> markup around the term, while XmlDoc syntax defines specific “term” and “description” for each item:

```
* <b>Width - </b>The width of the house
* <b>Height> - </b>The height of the house
```

```
<ul>
    <li><b>Width - </b>The width of the house</li>
    <li><b>Height> - </b>The height of the house</li>
</ul>
```

```
<list type=”bullet”>
    <item><b>Width - </b>The width of the house</item>
    <item><b>Height> - </b>The height of the house</item>
</list>
```

```
<list type=”bullet”>
    <item><term>Width</term><description>The width of the house</description></item>
    <item><term>Height></term><description>The height of the house</description></item>
</list>
```

Atomineer supports this style in the editor if the term and description are entered with a separator made of a space, a dash character and a space (‘ - ’). The text to the left of this separator is written back to the code comment as the ‘term’, and the text to the right as the ‘description’. If there is no separator, only the ‘description’ will be written back.

Important notes

- This view provides a text viewer/editor rather than a 'WYSIWYG preview'. The formatting of the text should not be expected to match the output of external documentation generation applications or web browsers

- The documentation shown in the viewer/editor is only that contained within the documentation comment. Atomineer will not augment it with other information (such as lists of class members), as the process of building this 'comprehensive' documentation is time consuming, which defeats the primary purpose of providing an instantaneous and lightweight viewer/editor.

- The editor supports some HTML (e.g. bold and italic) and documentation (e.g bullet list, parameter/see reference) markup; we will continue to work to bring support for more markup in future versions of Atomineer, but any markup which is not currently supported by the editor will simply be edited as plain text (e.g. XML tags). Please let us know if there is markup you require that is not currently supported.

- Although the editor accepts standard keypresses such as ctrl+C and ctrl+V for copy/paste, or ctrl+B (bold) and ctrl+I (italic), in some cases you may find that key bindings in Visual Studio will override these keys to apply different actions. In this case, if you wish to use the keyboard shortcuts, you will need to edit your global key bindings to allow the keypress through to the editor window: In Tools > Options, find the Environment > Keyboard page.

Place the cursor into the “Press Shortcut Keys” text field and press the keypress (e.g. ctrl+B). The dialog will then show the key bindings for that keypress (e.g. in “global”, ctrl+B may be assigned to “Debug.FunctionBreakpoint”). If you then enter that command name in the “show commands containing” field, you will be able to Remove the key binding. (Note that you can add the key binding for the same command, but restrict the scope, e.g. to the Text Editor window, rather than assigning it as a global key, so that it can still be used while you are editing source code, but it no longer interferes with the documentation viewer)

Hide Doc Comments using Outlining

Default hot-key sequence: Ctrl+Shift+A, Ctrl+Shift+O

Uses Visual Studio's Outlining feature to hide all Documentation Comments in the current file.

Usage

Outlining must be enabled for the current source file. Execute the command at any time, and the documentation comments will be hidden within outlining regions.

Description

This command hides all multi-line Doc-Comments (lines beginning with ///) in your file using the Outlining feature of the Visual Studio Editor. This allows you to see 'just the code' without the comments interfering with your view

To see the text of the comments, just hover the mouse over the [...] to show a tool-tip.

To edit a comment, double-click the [...] to re-display the outlined text.

Document All in this Scope

Default hot-key sequence: Ctrl+Shift+A, Ctrl+Shift+S

This command searches a file, namespace, class, interface, struct, record, or enum scope **using Atomineer's built in code parser**, and applies the Add Doc Comment command to each major code element within it.

Usage

To document all code elements at global scope in a file, place the cursor in the very first line of the file and execute the command.

To document all code elements within a specific scope, place the cursor in the declaration that introduces the scope (i.e. somewhere in the line with the 'namespace', 'class' (etc) text) and execute the command.

Description

When you execute the command, the code within the next scope (i.e. for C-style languages, from the '{' following the cursor position) will be scanned for child code elements to document. The Add Doc Comment command will be executed on each, adding or updating the comment in the usual way.

Directly nested classes, structs, records, unions and interfaces within the scope will also be documented (to any level of nesting).

This processing can be undone.

Some example to illustrate how useful this automation can be:

- Quickly and easily build the documentation structure for all the entries in an enum definition,
- Document all elements within a single class, even when you have several classes in a source file,
- Convert existing comments (e.g. in the Visual Studio default XmlDoc /// format) into the configured Atomineer format,
- Quickly update existing documentation after changing the Atomineer block format or preferences,
- Automate an initial documentation pass.

After executing this command it is highly recommended that you read through and update the resulting documentation to be sure that it is complete, correct and accurate.

Important Notes

- This command is affected by the 'restrict documentation by access level' preference. By default, all code elements will be documented. However, in C#, Visual Basic and Java code, this option can be used to restrict the generated documentation so that it only applies to code elements with the correct access level. Generally this is used to apply documentation to just the public members of a library, or suppress documentation of private members while allowing public/protected/internal members to be documented. Please note that as C++, PHP, Java/JavaScript, Typescript and Python access levels are not usually specified within the code element declaration, this option is unavailable or only partially available for those languages.

- This command operates in two modes. Initially it will try to use Visual Studio's Intellisense/CodeElement database to determine what to document (This mode may fail when the Solution has not been successfully compiled, or where Visual Studio itself does not provide the CodeElement information needed, such as in C++ header files). If this information is unavailable, Atomineer will parse the code in the scope for itself, searching for code elements to document. This makes the command very useful in older versions of Visual Studio, or when your project has not been recently compiled, as it will succeed in conditions where the other Doc All In... commands are unable to operate.

Document All in this File / Project / Solution

Default hot-key sequence: Ctrl+Shift+A, Ctrl+Shift+F

These commands search one or more source code Files and apply the ‘Add Doc Comment’ command to each major code element within them.

These commands will only be able to process languages for which the required CodeElement (Intellisense) information is available (i.e. it works best for .net languages, and in more recent versions of Visual Studio).

The commands are similar to 'Document all in This Scope', but use Intellisense, which can produce more reliable results in .net languages.

Usage

Place the cursor anywhere in a source code file and execute the 'Document All in This File/Project/Solution' command.

(Note: To document only the members of a single class at a time, use the 'Doc All in Scope' command instead)

Description

When you execute the command, Atomineer will search the Visual Studio Intellisense CodeElement database for all code elements in the current File/Project/Solution. It will then proceed to document each in turn.

Progress will be shown in the Visual Studio status line, and you may hold down the **Escape** key at any time to abort processing. After processing, details of what was documented are reported to the Visual Studio 'Output' window (in the 'Atomineer Pro Documentation' section)

This command can be used in the following ways:

- Convert legacy comments into the configured Atomineer format,
- Quickly update all existing documentation after changing the Atomineer block format or preferences,
- Automate an initial documentation pass on a large codebase.

After executing this command it is highly recommended that you read through and update the resulting documentation to be sure that it is complete, correct and accurate.

Important Notes

- This command is affected by the 'restrict documentation by access level' preference. By default, all code elements will be documented, but this option can be used to restrict the generated documentation so that it only applies to code elements with the correct access level.

- The command relies on Visual Studio Intellisense information to determine what to process, so in some cases it may not be able to process the code (primarily this depends on the version of Visual Studio in use, whether the solution has been successfully built, and how well Visual Studio supports the code language - it works best with .net languages, and in later versions of Visual Studio). If this command cannot process your code, try the 'Doc all in This Scope' command, as it uses Atomineer's own code parser to find the code elements to document. (This command will be greyed out if it knows that Intellisense information is

unavailable, but in some circumstances intellisense information can be incomplete, in which case the command may report 'No code elements found to document', or it may only document a subset of the code items present. This should not be considered not a fault in Atomineer, as it is simply restricted by the availability of Intellisense information).

When documenting entire Projects or Solutions:

- Note that processing hundreds or thousands of code elements in a single pass can be quite time-consuming, and requires Visual Studio to work hard (possibly using a lot of memory. This memory is **not** used by Atomineer directly, but results from the volume of processing being done by Visual Studio to deliver required Intellisense information, especially for C++ projects. In **extreme** cases this can lead to a crash, as VS only runs as a 32-bit process. If this occurs, try documenting one project or file at a time and then allow Visual Studio to recover memory from its heap before attempting another processing burst).

- Depending on the version of Visual Studio, you may see code windows updating during processing. This is normal behaviour.

- In Visual Studio versions 2010 and 2012, the status line progress indicator can sometimes cease updating, thus making it look as though the processing has locked up. If this occurs, status updates can sometimes be recovered by moving the main Visual Studio window. This lack of feedback does not affect processing, which usually completes successfully in due course.

- Open files will be processed and left open, and a new undo step will be generated for the changes made. Non-open files will be opened, processed, saved, and closed - no undo will be available in these files. If you wish to review all the changes it is suggested that you check all work in to source control before executing this command, so that the results can be easily diffed at leisure.

Process All in Chosen Files

This command applies any of the other Doc All In... commands to create, update or delete documentation from a set of files, first allowing you to set a number of additional options controlling exactly which file(s) should be processed, which code elements to process, and how the processing should be customised.

Usage

Place the cursor anywhere in a source-code file (within the project or solution you wish to document), and execute the 'Process All in Chosen Files' command.

Description

When you execute the command, Atomineer will show an options dialog, with 4 sections.

- In the first section, choose the main set of files to process (the current file, all open files, all files in the current Project, all files in the current Solution)

- If more than one file is selected, the second section allows you to apply filtering to further restrict which files are processed:

    - A **regex** filter can be applied. Any file whose leaf-name matches the regex will be processed; any which does not match will be skipped. For example, **.*\.cs** will only process files with ".cs" file extension; **Database\.*** only process files with a "Database" prefix.

    - **Partial classes**. Atomineer can be instructed to only document a partial class if the class name matches the filename (e.g. 'partial class MyClass' will be processed in 'MyClass.cs' but not in 'MyClass.Statics.cs', ensuring that the class itself is only documented in a single location. Note that the *contents* of the class will still be documented in these files)

    - **Read-only** files can be skipped. This can be used with some source control systems to only document files that are checked out for editing.

- Atomineer normally documents all code elements in a file. However, sometimes you might like to only update existing comments to 'refresh' them or convert them to a new style. At other times you might like to leave existing documentation alone, and only add new comments for code elements that were previously undocumented. Or you may wish to delete all documentation comments from processed files.

- Finally, (when documenting) you can control whether or not author/date entries will be added during the processing, whether file headers should be added and whether various code elements should be processed during the scan. The code element filters include:

    - Types – interface, class, struct, record, union (this does not affect whether their members will be processed, only the type declaration itself)
    - Enums – enum and enum-values
    - Members – member variables
    - Properties
    - Methods – method, function, sub, macro

Progress will be shown in the Visual Studio status line, and you may hold down the **Escape** key at any time to abort processing. After processing, details of what was documented are reported to the Visual Studio 'Output' window (in the 'Atomineer Pro Documentation' section)

Important Notes

- Please see the notes on **Doc All in this File** (above) for details of the commenting operation of this command.

- Note that processing hundreds or thousands of code elements in a single pass can be quite time-consuming, and requires Visual Studio to work hard (possibly using a lot of memory). In extreme cases lack of memory can then cause Visual Studio to fail. If this occurs, reduce the scope of the processing (e.g. to individual projects or files) to allow Visual Studio time to 'recover' after each batch of processing.

- Depending on the version of Visual Studio, you may see code windows updating during processing. This is normal behaviour.

- In Visual Studio versions 2010 onwards, the status line progress indicator can sometimes cease updating, thus making it look as though the processing has locked up. If this occurs, status updates can sometimes be recovered by moving the main Visual Studio window. This lack of feedback does not affect processing, which usually completes successfully in due course.

- Open files will be processed and left open, and a new undo step will be generated for the changes made. Non-open files will be opened, processed, saved, and closed - no undo will be available in these files. If you wish to review all the changes it is suggested that you check all work in to source control before executing this command, so that the results can be easily diffed at leisure.

Delete Documentation from this File

This command deletes all documentation comments from the active source code file, allowing the documentation to be regenerated from scratch.

Usage

Place the cursor anywhere in a source-code file you wish to process and execute the 'Delete Documentation from this File' command.

Description

When you execute this command, Atomineer will process the active source code file, removing any block documentation comments (including end-of-line doc comments) that either use the /// or /**…*/ (C-style languages) or ''' (Visual Basic) line prefixes, or your configured Atomineer comment style.

This operation can be undone if you are unhappy with the result.

Important Notes

- Please make absolutely sure that you wish to delete all documentation comments before confirming this action. On completion, review the results to ensure that Atomineer has only removed those comments that you intended - It will preserve file header comments and non-documentation comments, but it is possible that it might remove something you did not intend. You can undo to restore the previous state if you are unhappy with the results. For extra assurance, we recommend only working on source-controlled code so you can recover any removed information easily at a later date if necessary.

- Note that this command will remove comments that match your configured Atomineer documentation comment format. If you use an unusual or nonstandard format it is possible that Atomineer might only partially remove some comments, or may inadvertently remove non-documentation comments. Please review the results and discontinue use if your comment format causes side effects with this command.

Align Code into Columns

Default hot-key sequence: Ctrl+Shift+A, Ctrl+Shift+C

Quickly align text patterns into tidy columns and tables. This can be used to align assignments, commas, arithmetic/logical/bitwise/comparison operators, brackets, strings, numbers, variables and keywords.

Alignment can be used to significantly improve code readability and clarity in repetitive blocks.

Usage

There are two ways to use this command:

- Place the cursor next to a symbol that you wish to align to, and execute the command. Atomineer will search up and down from this starting point to auto-discover the block of similar code lines to be aligned.

- Select a block of lines to be processed, starting the selection from the symbol you wish to align to, and execute the command. Only lines that overlap the selection will be processed.

Description

When executed, this command will look at the **closest** text to the left or right of the current cursor position (or the top point of the selection). It then searches the block of lines to be processed for similar lines, and inserts tabs/spaces into these lines to align the matches into a column.

When used with a selected block, lines that don't match the pattern (e.g. blank lines) will be skipped, allowing tables with gaps in them to be processed in one pass.

For example (the red line shows where the cursor can be placed when executing the command):

	Text before aligning	Result
Align before assignments	`int age |= 5;` `int height |= 97;`	`int age    = 5;` `int height = 97;`
Align after assignments	`int age =| 5;` `int height =| 97;`	`int age =    5;` `int height = 97;`

Align after commas	`AddUser("Arabella",| "bunny");`
	`AddUser("John",| "abc");`
	`AddUser("Susanna",| "password");`
		`AddUser("Arabella", "bunny");`
		`AddUser("John",     "abc");`
		`AddUser("Susanna",  "password");`

Align before numbers	`int[] values =`
	`{`
	`  4379, |81,`
	`  323692, |1079233,`
	`  12, |36846,`
	`}`
		`int[] values =`
		`{`
		`  4379,   81,`
		`  323692, 1079233,`
		`  12,     36846,`
		`}`

By applying the command several times (left to right) you can align multiple parts of the lines:

Align variable names + Align assignments + Align comments	`byte flags = 0; // options`
	`int volume = 72; // m^3`
	`long displacement = 134; // in water`

	`byte flags        = 0;   // options`
	`int  volume       = 72;  // m^3`
	`long displacement = 134; // in water`

Align after colons + Align breaks	`case Left: t = "left"; break;`
	`case Top: t = "top"; break;`
	`case Right: t = "right"; break;`
	`case Bottom: t = "bottom"; break;`

	`case Left:   t = "left";   break;`
	`case Top:    t = "top";    break;`
	`case Right:  t = "right";  break;`
	`case Bottom: t = "bottom"; break;`

The command will also align equivalent code elements such as logic operators, work on non-code text such as XML or CSV data, and tidy up code indentation:

Align logic operators	`if (readFile &&`
	`    container.IsEmpty &&`
	`    file.IsOpen ||`
	`    reader != null)`

	`if (readFile          &&`
	`    container.IsEmpty &&`
	`    file.IsOpen       ||`
	`    reader != null)`

Align words + Align XML element ends	`<rec cn='Pitson' cc='PT' val='12'/>`
	`<rec cn='Vea' cc='VE' val='7'/>`
	`<rec cn='Lardy' cc='LAL' val='149'/>`

	`<rec cn='Pitson' cc='PT'  val='12'  />`
	`<rec cn='Vea'    cc='VE'  val='7'   />`
	`<rec cn='Lardy'  cc='LAL' val='149' />`

Align to tidy up indentation	`    int a = 0;`
	` int b = 1;`
	`  int sum = i + j;`

	`  int a   = 0;`
	`  int b   = 1;`
	`  int sum = i + j;`

Making the most of Alignment

Arguably one of the best features of the Visual Studio editor is also one of its least-known features. Hold down Alt while dragging (or use Alt+Shift with the cursor keys) and the selection becomes an arbitrary rectangular region (column) within the code. These columns can be copied and pasted, deleted, and overtyped (as you type, the text is repeated into every line in the selection).

Once you have aligned the code into columns with Atomineer, this feature makes editing tables of information incredibly fast and easy, as you can move entire columns, add and delete columns as easily as though you were editing a single line of text. This feature is incredibly powerful once you get used to it, and can save a huge amount of time.

Preferences

When inserting whitespace, either tabs or spaces will be inserted as defined by your Visual Studio preferences. However, there is one dedicated preference (on the 'Other Features' tab) for this command: When aligning code to a column, it can be aligned to any column (using spaces as necessary to minimise the inserted whitespace), or to the next tab-column (using only tabs).

Notes:

- The automatic block detection works best on blocks of code separated by blank lines. If you don't separate blocks in this way, you may find you need to explicitly select the block to be processed to help Atomineer align the text as required.

- Although it is targeted at code, this command works on **any text**, including text within documentation comments, XML or CSV files, etc.

- The algorithm used by Atomineer is a generalised one. The advantage of this is that it is not limited (e.g. to only aligning assignments), but the compromise of the design is that on occasion you may find an example where Atomineer aligns something you didn't want it to. In these cases, you may need to be more explicit (select only the lines you wish to affect), or apply the alignment by hand. You can always undo if the alignment doesn’t achieve what you want.

- Visual Studio provides automatic formatting as you type - this can remove the extra whitespace provided by the Align Columns command. To avoid this happening, go to Tools > Options > Text Editor > {language} > Formatting, and adjust the Spacing options provided. (You can test the results by using the Edit > Advanced > Format Selection menu option to reformat a chunk of code to see how Visual Studio treats the aligned code)

Create C# or C++/CLI Property, Create C++ Access Methods

Default hot-key sequence:	Ctrl+Shift+A, Ctrl+Shift+A Ctrl+Shift+A, Ctrl+Shift+P

These commands both convert member variable declarations into Properties/Accessors (as appropriate for the current language). The “Property” variant applies to C++ (CLI), while the “Accessor” applies to C++ (standard). Both commands can be used in C#, and they operate identically.

- **C++:** Creates simple inlined "Get" and "Set" methods to access a member variable, making it much quicker and easier to make members private and properly encapsulate your class data.

- **C++/CLI:** Creates a Property for your member variable, implementing default get/set methods.

- **C#:** Creates a Property for your member variable, implementing default get/set methods. Each time it is executed the code will be converted through the following forms: It will convert a member variable into an auto-property, then convert an auto-property into an explicitly implemented property with a backing field (optionally using C# 7.0 expression-bodied (lambda) syntax or pre-7.0 syntax). Finally, the expression-bodied property syntax can be converted into a XAML/Prism implementation using a BindableBase.SetProperty() call.

Usage

In your C++ header or C# class, place the cursor anywhere in the middle of a member variable or auto-property declaration, then execute the command. In C# you can execute the command multiple times to cycle through various implementation choices, or use undo to revert each change.

Description

Atomineer will convert a member variable declaration to a property/accessor. The style/conventions used in the generated code can be configured in the preferences (“Other Features” tab, in the “Create Property/Accessor” section).

The source member declaration need not match the configured naming style to be processed correctly - it may have a lower-case first character, or an underscore as a prefix/suffix (e.g. ‘variable’, ‘_variable’, ‘Variable_’, ‘mVariable’ or ‘m_Variable’).

- **C++:** Get and Set methods will be created, and inserted into the document **above** the preceding "public/protected/private:" tag. For known simple types (int, float, etc.) the accessors will pass by value. For unknown types, it assumes complex types (struct/class) and will pass by const-reference. The names used for the getter/setter can be configured in the preferences.

- **C++/CLI:** A property is defined, and inserted into the document **above** the preceding "public/protected/private:" tag.

- **C#:** This operates in a series of stages to give you control over the generated code:
  1. When applied to a member variable, it is converted into an auto-property.
  2. When applied to an auto-property, it is converted into an explicitly implemented property with a backing field. The backing field will use a configurable variable naming style, and can use the newer C# 7.0 expression-bodied member (lambda) syntax – these can both be controlled in the Atomineer Options ("Other Settings" tab).
  3. When applied to an expression-bodied property-with-backing-field it will convert it once more, into a WPF/Prism BindableBase.SetProperty() syntax.

You can convert from a member variable to an explicit implementation by just executing the command multiple times in succession. Each conversion step can be undone if you go too far or decide you do not like the resulting implementation.

The preferences allow you to set a template for the backing field naming style. For an auto-property called 'MyProp', the default (a blank prefix) is to generate a backing field 'myProp'. With a template specified, other styles (e.g. '_MyProp', 'm_MyProp', 'mMyProp', '_Member_MyProp_' can be generated.

Example: C# Property

Before (a member variable)	`private int mSpeed;`
After 1st execution (an auto property)	`public int Speed { get; set; }`

After 2nd execution (as a property with backing field)	`public int Speed`
	`{`
	`  get { return mSpeed; }`
	`  set { mSpeed = value; }`
	`}`
	`private int mSpeed;`

After 2nd execution (using expression-bodied syntax)	`public int Speed`
	`{`
	`  get => mSpeed;`
	`  set => mSpeed = value;`
	`}`
	`private int mSpeed;`

After 3rd execution (using expression-bodied syntax)	`public int Speed`
	`{`
	`  get => mSpeed;`
	`  set => SetProperty(ref mSpeed, value);`
	`}`
	`private int mSpeed;`

Example: C++/CLI Properties

Before	`public:`
	`private:`
	`  int  mInteger;`
	`  MyClass  mComplexClass;`

After	`public:`
	`  property int Integer`
	`  {`
	`  int get()  { return mInteger; }`
	`  void set(int value) { mInteger = value; }`
	`  }`
	
	`  property const MyClass& ComplexClass`
	`  {`
	`  const MyClass& get()  { return mComplexClass; }`
	`  void set(const MyClass& value) { mComplexClass = value; }`
	`  }`
	
	`private:`
	`  int  mInteger;`
	`  MyClass  mComplexClass;`

Example: C++ Accessors

Before	`public:`
	`private:`
	`  int  mInteger;`
	`  MyClass  mComplexClass;`

After	`public:`
	`  int GetInteger(void) const  { return mInteger;  };`
	`  void SetInteger(int Integer) { mInteger = Integer; };`
	
	`  const MyClass &GetComplexClass(void) const`
	`  { return mComplexClass;  };`
	`  void SetComplexClass(const MyClass &ComplexClass)`
	`  { mComplexClass = ComplexClass; };`
	
	`private:`
	`  int  mInteger;`
	`  MyClass  mComplexClass;`

Copy As Text

Default hot-key: Ctrl+Shift+C

Copies selected text onto the clipboard in a simple, clean format for use in other programs.

Usage

Just as with a regular copy: Select the text you wish to copy and execute the command. If no text is selected, the entire line containing the cursor will be copied.

Description

When you copy text from Visual Studio to other applications, it normally includes colour codes for the syntax colouring scheme. This is not always very useful when you want to put the text into a document that will be printed or emailed, especially if you use a non-white background colour.

Additionally, Visual Studio copies the text verbatim - including all the indentation. If you are just copying a code snippet, this pushes your text to the right when pasted, and any included Tab characters can cause the formatting to become a complete mess as few other programs treat Tabs as 4 spaces as Visual Studio does.

This command addresses these issues by copying the selected text in your document to the clipboard as **plain text**. It (optionally) strips unnecessary indentation from the block, and (optionally) converts tabs into spaces.

Note that for the text formatting to appear correct in the pasted text, you must display it using a monospaced font (e.g. “Courier New” or “Consolas”).

Other Menu Items

Atomineer Options...
Shows a dialog for setting the Atomineer Preferences. Please see the **Preferences** section below for more information on setting up Atomineer Preferences.

Help and User Guide...
Opens the latest version of this User Guide page (direct from the AtomineerUtils.com website) in your default Internet Browser/PDF viewer.

Check for updates...
Accesses the AtomineerUtils.com website to determine if a newer version is available for download. (Note: This may be unable to connect if your PC is behind a Proxy Server).

About Atomineer Pro Documentation...
Shows a dialog providing information on Atomineer, the installed version, and the number of times you have used it.

Internal SuppressUI and AllowUI commands

These commands are not available on the menus. They are used to suppress user-interface prompting to allow automated execution from a Visual Studio macro or extension.

Usage
The SuppressUI and AllowUI commands are only available to macros and extensions (and hotkeys). They suppress or allow reporting of any problems via information dialogs.

Suppression of this UI allows macros/extensions to automate execution of Atomineer commands without being stalled by dialogs popping up.

Just include the following commands at the start and end of your macro/extension command:

```
DTE.ExecuteCommand("Atomineer.Utils.Commands.SuppressUI")

... your macro/extension code here ...

DTE.ExecuteCommand("Atomineer.Utils.Commands.AllowUI")
```

Note that this option suppresses all UI prompts from Atomineer, but Visual Studio itself may throw up UI in some cases (for example, if it encounters a read-only file - a solution to this is to check out/make all files writable before running your automated task)

Live Comment Editing Enhancements

Atomineer can optionally track changes within comments and documentation comments as you edit them, and offers the following enhancements to the standard Visual Studio behaviour. These are enabled by default, but can be disabled if you wish - see the 'typing aids' box on the first tab of the Atomineer Options.

Background highlight colour for Doc Comments

In Visual Studio 2015 onwards, Atomineer is able to highlight Documentation Comments by placing a filled background rectangle behind the comment. This can provide a clear visual separation between documentation and code (much as the top and bottom separator lines do)

By default Atomineer will use a light cream colour when applied to a light-coloured editing colour theme, or a dark blue colour when applied to a dark-coloured editing theme. However, the colours used can be set to specific colours in the preferences.

Auto creation of comments when you type ///

For .NET projects, Visual Studio has a handy feature: If you type the start of a documentation comment block (/// in C#, or ''' in Visual Basic) above a code element, then it automatically creates a skeleton comment block. Atomineer extends this behaviour in the following ways:

1. The Atomineer feature works in all supported languages, not just the .NET languages.
2. The comment block is generated using the Atomineer documentation engine, so the comment uses your configured comment style, and of course all the auto-generated documentation is filled in for you. It works equally well for Doxygen, QDoc, JavaDoc, NaturalDocs and XML documentation.
3. The comment block will be indented to the same level as the code element it documents, rather than being dumped at the place where you typed the /// or ''', and the other Atomineer formatting helpers (such as controlling the number of blank lines between code elements) will all apply as normal.

Live typing aids

In addition, Atomineer offers enhanced editing within **///** (**'''** in Visual Basic, **##** in Python) Documentation Comment blocks and any regular comment blocks made up of single-line comments (// in C#, C++, C, Java, JavaScript, TypeScript, PHP and UnrealScript, **'** in Visual Basic, or # in Python). These features help you edit the comment body text without having to continually insert and delete comment // or ' prefixes by hand to keep the comment block tidy. These typing aids include:

1. When you press Return/Enter to insert a new line in a comment, Atomineer will automatically extend the comment onto the new line, duplicating the comment header and indentation from the previous line so that you can continue typing within the comment without having to manage the comment and text indentation. If you wish to exit this 'comment editing mode', press Return twice in a row (the first press will extend the comment, and the second will remove the comment header to leave you on a blank line).
2. If the current line of your comment seems to start with what looks like a bullet-point (-, *, +) or a numbered list entry (e.g. 1) 2) 3) or 1. 2. 3. or a. b. c. etc), Atomineer will automatically extend the list by entering the next bullet/number on the new line.
3. When typing in comment block, if you press Delete to delete the newline at the end of a line, Visual Studio will append the following line. If the following line continues the comment block, Atomineer will strip the indentation and single-line-comment prefix to concatenate the comment body text in a cleaner way.
4. When pasting text into a comment block, Atomineer will automatically reformat it as appropriate to integrate it into the destination comment: Plain text will be converted into comment text; Comment text is converted if necessary between normal and doc-comment formats; code examples (that include a mixture of text and comments) are embedded into the destination comment as a code example, retaining the original prefixes on comment lines. (Note: This only happens if you paste into a commented line - pasting on a blank line adjacent to a commented line will not apply the reformatting. If the paste-reformatting produces an undesirable result in any case, simply undo once to remove the Atomineer reformatting and leave the text as it would normally have been pasted).

Notes:

- To use the standard Visual Studio behaviour to 'exit the comment' when pressing Enter, **press Enter twice in a row**, or type a Shift-Enter (add a newline at the current cursor position) or Ctrl-Enter (add a newline before the current line).

- If you enable/disable any of these features, you will need to execute 'Add Doc Comment' once (or restart Visual Studio) before the change to key-press handling will take effect.

- Due to the way Visual Studio handles key-presses, this feature may interfere with, or be affected by, other installed add-ins/extensions that also process Enter key-presses within comments. If this occurs you can reconfigure Atomineer (or the other add-in) to disable the clashing feature and restart Visual Studio. Where possible Atomineer tries not to interfere with other add-ins, but the known issues are:

- Visual Assist X has an option, (Advanced > Corrections : Auto-extend multi-line comments), which provides much the same new-line behaviour in comments as Atomineer.

Unfortunately these two features will result in the comment prefix being added twice on each line. The recommended solution is to disable the Visual Assist X feature (as it operates in isolation, while Atomineer offers many typing aids that are enabled/disabled together).

- Vim add-ins intercept Enter key-presses and under some circumstances may clash with Atomineer. Disable the Atomineer live aids if you encounter any problems using Vim key-presses in comments.

Appendix A:
Additional Preference Options

Atomineer provides a number of small tweaks that are not exposed in the main preferences user interface, as they are not required by most users. These are exposed either as preference options to affect the extension's behaviour, or as user-defined variables to affect the generated documentation.

Additional Preferences

The preferences XML file is usually saved to <My Documents>\Atomineer\Prefs.xml (unless you have opted to store your preferences on the Preferences Search Path, then these preference options will have been saved into a PrefSets.xml file in the first folder on the path. For long-term users of Atomineer a different location may be used - If you have any trouble locating your preferences, please just email support@atomineerutils.com and we will be happy to help you)

The preferences file is a simple XML-based format, so can be safely hand-edited as long as you are careful to keep the XML format valid. It is recommended that you always keep a back-up of your last good settings so that you can revert if you get unexpected results in Atomineer after you make any changes.

Most preference changes will be picked up the next time you execute Add Doc Comment, but a few preferences may require Visual Studio to be restarted before they will take effect.

Category	Preference name	Description
<Misc>	AddKeyBindings	Set this to "false" to disable Atomineer's default key-bindings if they clash with other bindings you would prefer to use.
<Doxygen>	InOutSep	This is a text string that is used to separate the in/out text from the rest of the entry. Normally this is a space or tab.
	InOutFollowsName	Set this to "false" to place the [in,out] specifier between the @command and the parameter name; "true" to place it after te parameter name, at the start of the description text.
	AlignInOut	Set this to "false" to disable column alignment for [in,out] text. This saves space but can make comments harder to read.

<DocComment>	AllowMacroReturns	Normally Atomineer will be confused if it encounters a macro used as a return type: SPECIAL(int) MyFunction() These macros can be handled generically by setting AllowMacroReturns="true", or specifically by adding pre-processing rules to deal with your specific macros. (Preprocessing is recommended, as enabling Macro Returns relies on heuristics and therefore may cause side effects)
	ConvertDoubleSlashComments	If "true", Atomineer will assume that any // comment directly above a code element is intended to be documentation, and should be converted into a documentation comment when Add Doc Comment executes. If "false", Atomineer will ignore/preserve // comments (the doc comment will use auto-generated description text instead)
	ConvertEolComments	If "true", Atomineer will assume that any end-of-line comments on code elements are intended to be documentation, and should be converted into a documentation comment when Add Doc Comment executes. If "false", Atomineer will ignore/preserve EOL comments (the doc comment will use auto-generated description text instead)

	EmitParamType	If "true", when generating <param> and <returns> entries, Atomineer will add an attribute to the entry indicating the variable's type, for example:
		
		`<param name="size" type="int"> The size </param>`
		
		`@param Size {int} The size.`

	EmitParamTypeWithBrace	If EmitParamType is enabled, then in non-DocXml formats, this will instruct Atomineer to add braces around the type, e.g. {int}

	IgnoreTypedef	If “true”, C style "typedef enum" and "typedef struct" etc are treated as simple "enum" or "struct"
	PreserveExceptions	If “false”, Atomineer will check that all documented exceptions are thrown in the body of a method, and will remove documentation for exceptions that no longer seem to be thrown. If “true”, Atomineer will preserve all exception documentation – if you remove a “throw”, you must manually update the comment to reflect the change.
	PunctuateBlankDescs	If “true”, Atomineer will auto-punctuate blank descriptions (so they will appear as “.”). If “false”, Atomineer will leave blank descriptions completely blank.
	ReplaceSeeAlsos	If “true”, Atomineer will remove any see-also (or sa) entries from existing comments as it updates them, to keep auto-generated docs in sync with base class refactoring. The default is “false”, as seealso entries might also be used to cross-reference other relevant documentation.
	RootPaths	Normally, Atomineer will output project-relative paths for filenames. However, when using a makefile there may not be a project root folder, in which case you can use this preference to strip off common root folders from filenames. Any number of root folders can be specified, separated by semicolons.
	TimeFormat	The main preferences allow the DateFormat to be set using a standard .net date formatting string. This can be used to set a time formatting string, if you use %time% anywhere in your rules.

The preferences file can also include <PrefSet…> entries. Each of these provides a set of preferences for a list of specific file-types. The elements within a PrefSet are the same as those in the main preferences, and if present, each will override the main preference of the same name. (For example, a PrefSet is used to allow different comment prefixes for Visual Basic than for the other supported languages)

Additional Documentation-Generation Variables

In the Atomineer Options **Doc Advanced** Tab, click the **User %variables%** button to edit the user-defined variables. As well as adding custom variables to use in your own rules, you can override the following built-in variables to affect the documentation that Atomineer generates.

Mark-up replacement variables

Atomineer's default aim is to generate human readable comments in your source code. However, if you wish to generate external documentation, you may require additional mark-up to be included in the descriptions to provide a better quality of external documentation. For example, you might want to use:

```
<c>false</c>        (DocXML)
@c false            (Doxygen)
```

…so that generated documentation will show a **false** value in bold.

Variable name	Description
false	The text for false / False / FALSE values
null	The text for a null or NULL
nullptr	The text for a null pointer
true	The text for true / True / TRUE values

Control variables

The following variables more generally control the behaviour of the generated documentation.

Variable name	Description
alwaysAddInForParams	By-value parameters can be documented with **[in]**, reference/pointers with **[in,out]** and out parameters as **[out]**. true = place [in] on the by-value parameters false (default) = do not place [in] on by-value parameters (See also the **suppressInOut** variable, below)
company	The name of your company, for use in header comments, e.g. `Atomineer`

copyright	The copyright text to use in header comments. This usually includes the company and year variables to generate the final text, e.g.
	`Copyright © %year% %company%`
	
	Or you can hard-code it:
	`Copyright © Atomineer`

except-passprefix	When documenting exceptions, Atomineer will (by default) delete entries for any exceptions that it cannot see being thrown in the body of the method. In order to document exceptions that "pass through" the method (i.e. are thrown by other methods that it calls) without Atomineer removing the entries, you can start the description with the "pass prefix" text. By default the pass prefix is "pass", so that a description like "**Pass**ed when…" will be preserved by Atomineer. If you prefer a different phrasing, this prefix can be changed to match it, e.g. "**Thrown by called** method when…"
indexer-getonly	The text to use as a prefix for get-only indexers, e.g. `get` Will produce comments like: "Get the width"
indexer-getset	The text to use as a prefix for get-and-set indexers, e.g. `get or set` Will produce comments like: "Get or set the width"
indexer-setonly	The text to use as a prefix for get-only indexers, e.g. `set` Will produce comments like: "Set the width"
param-optional	The text to prepend to a description when describing an optional parameter, usually "(Optional)"
property-getonly	The text to use as a prefix for get-only properties, e.g. `get` Will produce comments like: "Get the width"

property -getset	The text to use as a prefix for get-and-set properties, e.g. `get or set` Will produce comments like: "Get or set the width"
property -setonly	The text to use as a prefix for set-only properties, e.g. `set` Will produce comments like: "Set the width"
suppressInOut	Atomineer will usually add [in], [in,out] or [out] specifiers on parameters. Set this variable to **true** to disable this behaviour.
userEmail	The user's email address, e.g. `support@atomineerutils.com`

Undocumented Customisations

There are a number of ways that Atomineer can be configured that are undocumented – either because they are rarely needed, or because the documentation would be overwhelming if every possible nuance were described in full.

Even when the options you need are documented it may still be hard to find out what they are or to realise how a setting can be used to achieve what you need.

If you have a requirement that you cannot easily achieve then please do email us (support@atomineerutils.com) and ask – in many cases we may be able to provide preference settings, templates or rules to quickly achieve what you need.