



ABCM Plugin for FrameMaker®

v1.0 User Guide

Contents

Introduction

Advantages over native conditional text	1
What the plugin does	2
Requirements to use the ABCM plugin	3
About the name.....	3
Trademarks and licensing information.....	3

Getting Started

Definitions of terms	5
Important note on native conditional text	5
Specifying attributes	6
Conditionalizing whole chapters.....	7
Preferences	7
About the main settings file.....	8
About scheme categories	9

Filtering, Coloring, and Validating

Filtering	11
Launching a filter	12
Filter types - Source versus duplicate file	12
How filtering works	14

Restoring a document or book	14
Conditionalizing (and filtering out) entire files	14
Coloring	15
Launching a coloring action	15
Removing coloring	16
Attribute validation	16
Automatic validation	17
Validation rule #1 - Simple syntax	17
Validation rule #2 - Ancestor element lacking subordinate condition	17
Rule #3 - Unspecified descendants	19
Rule #4 - Empty attribute not allowed	19

Scheme Setup And Other Main Settings

General information about schemes and categories	21
General scheme editing procedures	22
Coloring schemes	22
Basic coloring scheme behavior	23
Coloring scheme details	24
Attribute/value matching criteria	24
Coloring rule order	26
<no value> and <any value> in a coloring scheme	27
Where the colors come from	28
Other coloring scheme options and features	28
Filter schemes	29
General filter scheme matching behavior	30
<no value> and <any value> in filter schemes	31
Filtering out elements by type, using unique attribute names	32
Validation schemes	34
Advanced scheme options	34
Considering EDD-applied defaults	35
Ignoring case-sensitivity	35
Attribute values delimited by whitespace (Tokenized strings)	35
Processing all flows	37
Master attribute library	37
Master colors list	37
Migrating Sourcerer settings	37

External Calls to ABCM

How to send an external call to ABCM	39
General information on external calls	40
Specifying document and book arguments	41
Specifying Boolean arguments	41
Call reference	41
ColorElement	41
Syntax	42
Usage description	42
Returns	42
ColorFile	43
Syntax	43
Usage description	43
Returns	43
Hello	44
Syntax	44
Returns	44
FilterCheckElement	44
Syntax	44
Usage description	45
Returns	45
FilterFile	45
Syntax	46
Usage description	46
Returns	47
RestoreFile	47
Syntax	47
Usage description	48
Returns	48
SetParm	48
Syntax	48
Usage description	49
Returns	50
SetScheme	50
Syntax	50
Usage description	51
Returns	51
ValidateElement	51
Syntax	51
Usage description	52
Returns	52

Detailed example—Calling ABCM (FDK)	52
--	-----------

1: Introduction

ABCM is a plugin for the structured environment of Adobe FrameMaker that allows you to use attributes and values to indicate and manage conditional content. The functional concept is similar to native conditional text, except that conditions are specified and managed with structural metadata rather than traditional condition tags.

Structured FrameMaker has always allowed conditional metadata to be specified with structural attributes, but it has never included any native features to manage those conditions. This plugin fills that deficit and provides those management functions, such as conditional coloring and the production of conditional output.

Advantages over native conditional text

The use of structural metadata and the ABCM plugin for conditional text brings a host of advantages over native condition tags, including:

- **Easier management of multiple, overlapping conditions** - Structural attributes provide a multi-dimensional matrix for specifying overlapping conditions, and the plugin allows them to be managed much more easily and independently. If you have used native conditional text in the past, you are probably aware of the difficulties that immediately surface once you attempt to overlap multiple conditions, many of which are overcome by the ABCM methodology.

With this plugin, each condition is its own entity that can be managed without conflict with others. You can color any condition or combination of conditions as you see fit, and completely ignore any conditions that you are not concerned with at the present. Similarly, you only deal with the conditions of interest while producing (or *filtering*) output. No more sifting through long lists of conditions trying to decide what to hide versus show.

- **Clear and concise application of conditions** - When your conditions are assigned as attribute values, you know exactly what content is conditional, every time. You never again have to be concerned with issues like a missed character or paragraph mark during condition assignment.
- **“Automatic” conditions** - Because the mere presence of an attribute can represent a condition, a particular element can be designated as always conditional. A common application of this behavior is through the use of designated elements for in-text authoring comments. If you place a unique attribute on such an element, you can have the filtering process automatically remove every single instance of the element, ensuring that every one is removed, every time. Following this example, you would never again need to worry whether you conditionalized all of your personal comments.
- **“Scheme” usage for controlling coloring and filtering actions** - The plugin operates on the concept of “schemes” to control filtering and coloring activities. A scheme allows you to “program” a certain coloring or filtering pattern into your settings one time, then simply run that scheme afterwards. The repetitive decision-making process of the Show/Hide dialog box becomes a thing of the past.
- **Preservation of conditions in markup** - If you export XML or SGML and your conditions are specified as attribute values, your conditions will naturally survive the trip in their native form. Furthermore, they will be readily available for any external post-process to recognize and manage as necessary.
- **Whole-chapter conditionalization** - The functional model of the plugin allows you to conditionalize whole chapters of a book.

What the plugin does

The plugin provides three main functions:

- **Coloring** - Using schemes based on attribute values, the plugin can color your conditional content in a highly-flexible and customizable fashion. The concept is similar to native condition tag indicators, but the functionality is much more advanced. For more information, see [Coloring](#) on page 3-15.
- **Filtering** - Filtering is the process of producing conditional output, similar in concept to using the FrameMaker Show/Hide dialog box. It is functionally different than native conditional text, though, being more flexible and easier to use. For more information, see [Filtering](#) on page 3-11.
- **Validation** - With conditions specified in structural markup, there are certain rules which can be applied to help maintain the accuracy and integrity of those conditions. Validation uses these rules to check your document setup and reports any violations it finds. It has no counterpart in native conditional text. For more information, see [Attribute validation](#) on page 3-16.

Requirements to use the ABCM plugin

- Structured FrameMaker 7.x or 8.x, with any EDD. You may use any attributes and values you want for conditions.
- Microsoft Windows, as supported by your version of FrameMaker.

About the name

ABCM is an acronym for Attribute-Based Condition Management. The plugin is named this because no one could come up with a better name, much to the lament of the developer.

Trademarks and licensing information

This software is provided free and carries a license for unlimited use by anyone in a corporate, business, personal, or other setting. You can use it, not use it, copy it, reverse engineer it, or anything else that amuses you or makes you money in an otherwise legal fashion. If you choose to use it, be aware that YOU are FULLY RESPONSIBLE for any actions or consequences that result from its use, including hardware damage and data loss. West Street Consulting and associates will not be held responsible for any damage caused by its use. USE IT AT YOUR OWN RISK!

Adobe and FrameMaker are registered trademarks of Adobe Systems, Inc. Quadralay and WebWorks are registered trademarks of Quadralay Corporation. FrameScript is a registered trademark of Finite Matters, Ltd. FrameAC is a product of Mekon Ltd. All other marks are trademarks of their respective owners. West Street Consulting is not affiliated with Adobe Systems and this software is in no way developed, endorsed, or approved by Adobe.

With regards to functionality only, this plugin is somewhat of a next generation to the Sourcerer™ plugin, owned and distributed by Advantica, Inc.® Its author is a former employee of Advantica and was involved with the original development of Sourcerer. Currently, West Street and Advantica have no affiliation with one another and Sourcerer and ABCM are separate and completely independent products. The author would like to publicly thank Advantica for the time and inspiration to develop Sourcerer, which has led to the advancements found in this newer software.

2: Getting Started

This chapter contains information about the basic concepts involved with the plugin.

Definitions of terms

Within this document, note the following definitions:

- **Native conditional text** or **conditional text** - Refers to the built-in conditional text feature that comes with FrameMaker, with the standard condition tags and show/hide behavior. While these terms can be accurately applied to elements with conditional attribute values as processed by this plugin, they are reserved for the native feature for clarity.
- **Attributes** and **Values** - Refers specifically to the attributes and values found in structural markup. With regards to ABCM functionality, attribute values are the means of conditional tagging, replacing the condition tags used with native conditional text.

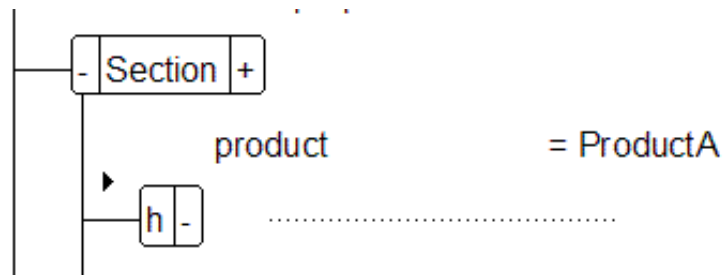
Important note on native conditional text

This software is intended as a replacement for native FrameMaker conditional text, leveraging the power of structural markup to overcome the many limitations associated with the native feature. While the software does not prevent you from using native conditional text at the same time, it is *highly recommended* that you do not. The usage of this software to manage conditional information does not mix well at all with the usage of native conditional text. In any given document set, you should use one or the

other exclusively. Using ABCM in conjunction with native conditional text will likely produce unexpected and disappointing results.

Specifying attributes

For the purposes of this plugin, any attribute and value can represent a condition. For example, the following `Section` element shows a potential “ProductA” condition designated for the `product` attribute:



Or, the following figure shows a `Section` element potentially conditionalized for two different products:



You may use multiple attributes and values as needed to designate conditions, overlapping as necessary, such as:



In short, you may use any attributes and values you wish, provided that you use them consistently and build your scheme logic around them.

NOTE: Using a Strings-type attribute in an EDD, it is convenient to add multiple values to a single attribute in the form of a list. However, ABCM also supports tokenized strings of values, typically found in XML markup. The recognition of this construct is an individual scheme option. For more information, see [Attribute values delimited by whitespace \(Tokenized strings\)](#) on page 4-35.



Figure 2-1 Multiple values as a tokenized string

Conditionalizing whole chapters

One of the key benefits to using ABCM is the ability to conditionalize entire chapters of books. For more information, see [Conditionalizing \(and filtering out\) entire files](#) on page 3-14.

Preferences

The following table describes the general plugin preferences, accessible by selecting **ABCM > Local Preferences**:

Main settings file location	Sets the path for the primary main settings file. For more information, see About the main settings file on page 2-8.
Default scheme category	Sets the default scheme category upon FrameMaker startup. The default category will initially appear in the coloring, filtering, and validation dialog boxes. It will also be used for locating the default validation scheme if you have auto-validation turned on.
Argument delimiter for external calls	String delimiter to use between arguments for external calls to ABCM. For more information, see General information on external calls on page 5-40.
Default validation scheme	(Optional) Default validation scheme to use upon startup. If you have auto-validation turned on, you should have a valid scheme specified here, otherwise the validation actions will fail until you manually specify a valid scheme.
Auto-validate attributes while authoring	Causes validation to occur during key user events such as inserting and wrapping elements, and setting attribute values. Auto-validation only occurs on the single element in question, such as the one just inserted. For more information on validation, see Attribute validation on page 3-16.
Apply strikethrough text	Causes elements that violate validation rules to be marked with strikethrough text during a validation action. If unchecked, validation will produce warnings and/or a report only.

Active rules	Activates and/or deactivates validation rules on an individual basis. For more information on validation rules, see Attribute validation on page 3-16.
Default filter type	Default filter type that appears initially in the filter dialog box, where it can be changed later. For more information, see Filter types - Source versus duplicate file on page 3-12.
Default coloring scheme	(Optional) Coloring scheme to use upon startup until manually changed.
Politics	Don't lie. We will know.

About the main settings file

The main settings file is the home for all scheme and attribute library data. Whenever you work in the scheme editor, all the data you see is coming from the main settings file, and likewise all modifications are stored therein. Whenever you specify or change a scheme, all scheme parameters come from this file.

The plugin always keeps a local copy of the main settings file to retrieve scheme data whenever necessary. If you want, you can use the local file as the only copy, managing all your data locally. In this scenario, the scheme data you use is private to your installation, and the only means of sharing scheme data with other users is to pass a copy of the file around. When you edit schemes, you are editing data in the local copy only.

Alternatively, you can place a “master copy” of this file anywhere on your computer or corporate network, and point any number of individual users to that master file. In this scenario, FrameMaker will retrieve a copy of the master file upon each startup and store it locally, after which it operates normally with the local copy. With this type of enterprise configuration, you can maintain a central library of scheme data and ensure that all applicable users are using the same scheme parameters.

The location of your “master copy” is specified in your preferences. By default, the location points to the local copy, which normally at the following path (or similar):

```
C:/Program Files/Adobe/FrameMaker7.2/WestStreet/ABCM_MainSettings_LocalCopy.fm
```

With the preferences pointed to this file, you will be working in “local mode” only. If you want to engage the “enterprise” feature, you need only to move a copy of that file to some master location, then edit your preferences to point to the master copy. Once you point to a different file, the plugin will go to that location automatically upon startup, retrieve a copy, and overwrite the local version. The local copy will remain current with the master file and a group of users can remain synchronized. For more information, see [Preferences](#) on page 2-7.

Whenever you edit schemes, you are editing the file that your preferences point to. If your preferences point to the local copy, your edits will appear on your local installation only. If your preferences point to some master copy, your edits will appear there, and your local copy will be refreshed once you are complete. If you edit some master copy that other users also point to, they will see your changes upon their next startup of FrameMaker.

NOTE: If you point to a file on an enterprise location but the software is unable to find it, the most recent local copy is used instead. In this manner, the software will remain functional in the event of network problems that are blocking access to your master copy.

A main settings file is a structured FrameMaker document itself. It may have any name, but it must use the EDD designed for it. If you plan to place a master version somewhere else, you should simply copy and paste the local copy that installs with the software, then point your preferences to the new location.

About scheme categories

The software uses three different types of schemes for processing, as applicable:

- Coloring
- Filtering
- Validation

To help with scheme management and organization, schemes are placed into categories, and you must select the appropriate category when setting or editing a particular scheme. Each category can contain any number of coloring, filtering, and validation schemes, and the way in which you set up your categories is completely up to you. This architecture is designed simply to allow the grouping of common schemes, such that any given list does not get too long.

3: Filtering, Coloring, and Validating

This section of the document describes the three main processes that the ABCM provides to manage your conditional content:

- **Filtering** - Filtering is the process by which you produce publishable output from your composite, conditional source. It is analogous to the “Show/Hide” activity with native conditional text. For more information, see [Filtering](#) on page 3-11.
- **Coloring** - Coloring is a means of denoting your conditions with custom colors, primarily as an authoring convenience to help you visually see your conditions. This function is analogous to the “condition indicators” aspect of native conditional text. For more information, see [Coloring](#) on page 3-15.
- **Validation** - Validation provides an automatic means of detecting common errors with conditional attribute assignment. It is unique to markup-based conditions and has no counterpart in native conditional text. For more information, see [Attribute validation](#) on page 3-16.

You can have as many schemes as you want, and even share them at an enterprise level. For more information, see [About the main settings file](#) on page 2-8.

Filtering

Filtering allows you to produce output from a composite, conditional source, often as one of the final steps before publishing. It has some conceptual similarity to showing and hiding native conditional text, but it is much more advanced and flexible.

The filtering logic is directed entirely by the parameters of your defined filter schemes. That is, for any given document, all decisions about what content gets “shown” versus what gets “hidden” are based on the instructions found in the scheme that you run. This

section does not cover this logic; rather, it describes the general aspects of filtering and file handling. For details on how the logic of schemes work during the filtering process, see [Filter schemes](#) on page 4-29.

Before attempting to filter content, you should read this section carefully. You should be especially sure that you understand the difference between source and duplicate file filtering, as described in [Filter types - Source versus duplicate file](#) on page 3-12.

Launching a filter

To launch a filtering action, bring the desired book or document to the front and select **ABCM > Filtering > Filter {doc type}**. This function will produce the filter dialog box, with the following options:

NOTE: If you intend to filter a whole book, be sure to bring the book window to the front before launching the dialog box.

Scheme category and Filter scheme	Sets the filter scheme you want to run. For more information on scheme construction and behavior, see Filter schemes on page 4-29.
Filtered book folder	Sets the target folder where a duplicated, filtered book will be placed. This option is only applicable to duplicate file book filters. For more information, see Filter types - Source versus duplicate file on page 3-12.
Filter type	Filter type for the current filtering action. For more information, see Filter types - Source versus duplicate file on page 3-12. NOTE: You can set a default value for this option in your preferences. For more information, see Preferences on page 2-7.
Save original files before filtering	Causes the plugin to save all applicable files before launching the filter, including the book file if you are performing a book filter. This option is especially recommended for duplicate file filters, because the filtering action will close your original files during the process and any unsaved changes would be lost otherwise.

Filter types - Source versus duplicate file

ABCM provides two types of filtering which you should be sure to understand before using the filter:

- **“Duplicate file” filtering** - With this type, your files to be filtered are duplicated, then the content to be “hidden” is completely deleted from the duplicate. Your source files are unaffected, and the result is a filtered duplicate of the source file. This form of filtering is very clean and is generally recommended for filtering processes involved with pre-publication document preparation.
- **“Source file” filtering** - This type of filtering works directly on your source files and uses native conditional text as the tool to hide the unwanted content. For all content deemed to be hidden, it applies a condition called “Hidden” and then hides that condition after the filter. Your source files are effectively filtered, but because the unwanted content was hidden with regular conditional text, no content is permanently deleted.

The decision of which filter type to use is purely based on workflow. For publishing, especially books, the duplicate file method is generally preferred, because native conditional text is known to cause crashes and other oddities while generating print or PDF output. On the other hand, the source file filter might be more convenient on a single-document basis while authoring, such that you can get a quick view of what your output will look like.

If you perform a duplicate filter on a single document, the plugin will open up an unsaved duplicate and then filter it, leaving it open on the screen afterwards. If you perform a duplicate file filter on a book, you must specify a target folder to receive the duplicate book. ABCM cannot duplicate a book within the same folder as the original book, because duplicate books use all the same filenames as the original book. The only physical difference with the duplicate book is the missing content that was filtered out, as applicable.

Using the duplicate file filter on a book provides a convenient means to move your publishable output to some refreshable staging area for publication. For example, if you use Quadralay software to generate help systems from your book, you can filter the book at will into a separate project area, then run the help generation software on the filtered duplicate. With this method, you never need to be concerned with the help generation software manipulating your source files or attempting to manage conditions for you.

When you perform a duplicate file filter on a book, ABCM will adjust all cross-references and file reference links automatically. For cross-references between chapter files, the links will be adjusted to point between the respective chapter files of the new, duplicated book. For cross-references outside the book, the plugin will leave them alone and they will continue to point to the same, external source.

If you use the source file filter, you should be aware that the native conditional text is used as a mechanism to hide content only. The native conditional text (i.e., the “Hidden” condition) plays no part in the logic of determining what to show or hide. All show/hide logic is driven by the respective filter scheme. Note the following important items about source-file filtering:

- The filtering and restoration process will interfere with any native conditional text assignment already in the document. Although you should not use native conditional text in conjunction with ABCM, you should be especially sure not to run a source file filter on a document that still contains any.
- The plugin provides an automatic means of restoring a document after a source file filter. For more information, see [Restoring a document or book](#) on page 3-14.

How filtering works

When a structured flow is filtered, ABCM starts at the highest-level element and walks through the entire structure tree in a logical fashion, checking the attributes on each element against the active filter scheme. For any given element, if the element is determined to be a “keeper,” the plugin continues on to the next element. Conversely, if ABCM determines that the element should not be kept, it will hide or delete the element and all its children, according to the filter type. It then backs up to the previous element and continues down the tree.

The active filter scheme contains all the logic used to determine what content should remain, and what should be removed. For details on this logic, see [Filter schemes](#) on page 4-29.

Restoring a document or book

After performing a source file filter, you can restore your document or book to normal by selecting **ABCM > Filtering > Restore {doc type}**. This function will remove all instances of the “Hidden” condition tag in the restored document(s). The intent is to restore the document to the actual condition previous to the filter action.

NOTE: This function may remove other native conditional text from the document. For this reason, you should never run a source file filter or post-filter restoration if you are still using native conditional text manually. For information on native conditional text, see [Important note on native conditional text](#) on page 2-5.

Conditionalizing (and filtering out) entire files

ABCM allows you to conditionalize an entire chapter of a book, and likewise filter out the whole chapter as applicable. To conditionalize a chapter, simply use conditional attributes at the highest-level element of the main flow, like you would any other element. If a filter scheme determines that the HLE should be removed, it assumes that the entire file should be removed from the book.

Note that whole-chapter removal can only occur for duplicate file filters, because it requires a permanent alteration to the filtered book. The plugin never makes

permanent alterations to your source files, so it cannot remove a file if you are performing a source file filter. If a source file filter encounters an HLE that should be hidden, it simply hides all content in the flow, which will normally leave you with blank pages in your output. The ability to conditionalize and remove entire chapters is an important benefit to using duplicate file filters, among others.

Coloring

Coloring text according to conditions is mostly an authoring convenience, allowing you to see visually where your conditions are assigned. In this respect, the purposes of coloring with the ABCM plugin are exactly the same as those associated with native conditional text.

The process of conditional coloring with the plugin, however, is functionally much different than native conditional text. Before attempting to set up schemes and perform coloring actions, you should be aware of the following:

- Coloring does not occur until you manually run a scheme through the **ABCM > Coloring** menu or associated shortcuts. The plugin does not include any automatic coloring.
- Coloring is applied as a simple format override, much as if you opened the paragraph designer, selected a color, and clicked Apply. It is therefore easy to remove by refreshing the EDD definitions.
- When you run coloring on a whole document, the EDD definitions are first refreshed to remove any previous coloring. This process will also remove any other format overrides in the document.
- The active coloring scheme contains all the logic used to determine color assignment. For more information, see [Coloring schemes](#) on page 4-22.

NOTE: Coloring should occur with reasonable reliability, but IT IS NOT FLAWLESS. Format overrides do not always mix well with an EDD-driven, structured environment, and some anomalies may occur. It should always work well enough, though, to clearly indicate where your conditions are assigned. Note that in any case, FILTERING SHOULD BE FLAWLESS. If a certain piece of content does not color as expected, it should still filter correctly. If it does not, the software has a critical bug and you should report it to West Street.

Launching a coloring action

To launch a coloring action, you should bring the desired file to the front and select **ABCM > Coloring > Color {doc type}**. Note the following:

- When you choose to color a selection only, the software colors the selected element and all descendants. If no element is entirely selected, the software colors the element that contains the insertion point, and all descendants. If there is no insertion point, nothing happens.
- The software does not automatically apply coloring, so you may want to keep the Esc 1 1 shortcut handy for coloring the current selection. In your preferences, you can specify a default scheme to load upon startup that will be used for selection coloring, until you change it.
- By default, the scheme selection box appears each time you launch a document or book coloring action. You can stop this behavior by unchecking the option at the bottom of the selection box. Afterwards, coloring will initiate as soon as you select the respective command, using the most recently-selected scheme. If you want to restore the appearance of selection box, you can select **ABCM > Coloring > Set Active Scheme** to produce the same selection box, and recheck the option.
- When the software determines that a color should be applied, it looks for it by name in the current document's template. If it cannot find the color, it cannot apply it.
- When an entire book is colored, any chapter files that are closed will be skipped.
- Coloring will override any colors applied by paragraph and character formats, but it will not override colors applied by condition indicators from native conditional text.

Removing coloring

Coloring by ABCM is accomplished by simple format overrides, and are therefore easily removed by refreshing the element definitions. You can do this with the **File > Import > Element Definitions** command, or perhaps with a shortcut provided with plugins such as West Street Structure Tools.

Attribute validation

ABCM includes a validation feature that helps you prevent common issues associated with attribute values and conditional markup. It may be used as an automatic process during authoring, or you can run it on an entire document or book as a post-process before filtering and publishing. When you run it on an entire file, ABCM produces a hyperlinked report detailing all the issues it found.

The following sections contain details on the four potential issues that validation can detect. Before using validation, note the following:

- The software is programmed to recognize four potential issues, but not all four may be important to you. For this reason, you can selectively decide which validation “rules” should be active, using your preferences. For more information, see [Preferences](#) on page 2-7.
- ABCM has no association with element definition validation, launched through the Element > Validate menu command.

Automatic validation

In your preferences, you can opt to have validation occur automatically during key user events, such as inserting elements and setting attributes. Any rule violations are reported instantly with message boxes, and strikethrough text applied as applicable.

For auto-validation to work, the software must have an active validation scheme. In your preferences, you can specify a default validation scheme to load upon startup, such that auto-validation will begin to work immediately. If you do not specify a valid default scheme, auto-validation will fail after startup until you manually set an active scheme. For more information on preferences, see [Preferences](#) on page 2-7.

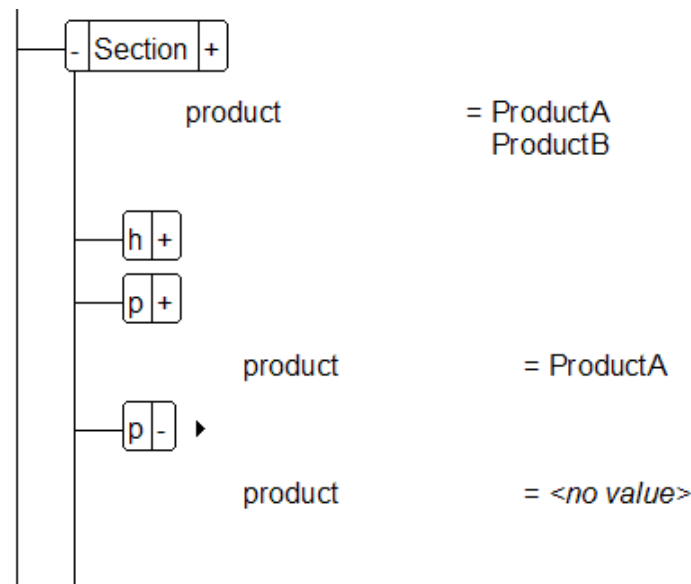
Validation rule #1 - Simple syntax

When using attribute values to denote conditional content, the syntax of the specified values is extremely important. With rule 1 active, the validation feature will scan all attributes contained in the active validation scheme, and ensure that all specified values match those contained in the scheme. If the software finds a value in the document that is not in the scheme, it will report it as an error. Note that it only scans the values of the attributes found in the scheme, and all other attributes are ignored.

Validation rule #2 - Ancestor element lacking subordinate condition

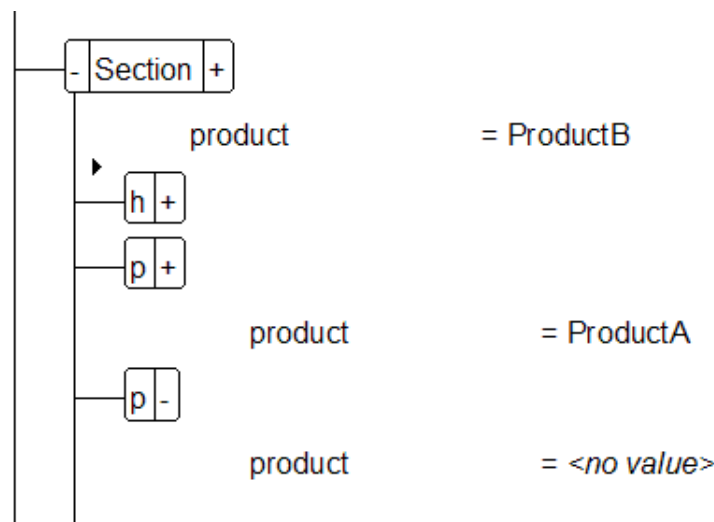
Because of the natural inheritance that flows through a structure tree, it is normally an error when something breaks that flow. When using attribute values to denote conditions, this situation can occur when an element contains a condition that is not shared by all its ancestors.

As an example, consider the following structure fragment:



This fragment is hierarchically sound because all conditions are properly nested and the natural inheritance is not broken. The `Section` element contains all the conditions shared by subordinate elements. This includes the `p` element with no value, because a “no value” situation generally indicates to inherit parent conditions by default. This fragment will filter normally for both products A and B, and no content will get lost.

Conversely, consider the following fragment:



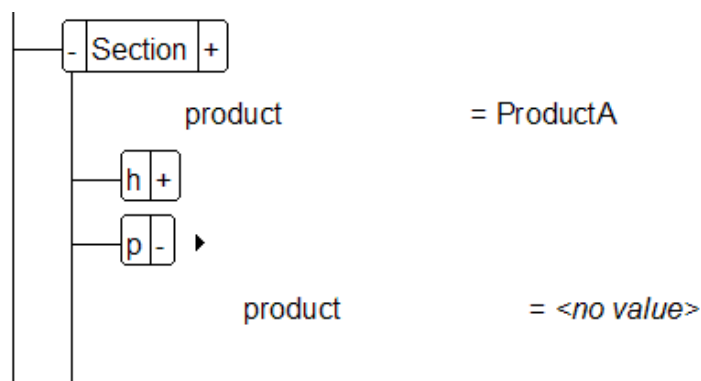
This fragment has hierarchy issues because conditions are not properly nested. In particular, the “ProductA” element is generally orphaned, because its condition is not shared by the `Section` ancestor. To illustrate, consider the case where you are filtering to produce a “Product A” version of this document. The `Section` element will be removed during the filter process, because it does not apply to Product A. This removal, however, will also remove the subordinate elements, including the `p` element

tagged for Product A. So, this element will never appear in a Product A version of the document despite its tag, due to the mismatch in conditional hierarchy.

During validation, this scenario is checked for all attributes found in the validation scheme. Note that this rule does not look at the validity of the values themselves; rather, it only looks for mismatches between ancestors and descendants with any value. Therefore, the specific values specified in the validation scheme are not used for this rule, unlike rule #1.

Rule #3 - Unspecified descendants

In some specialized cases, you may require that all elements explicitly specify all applied and inherited conditions. For example, consider the following structure fragment:



In a purist scenario, the empty `product` attribute on the subordinate `p` element might be considered an error, because it does not explicitly contain the condition of its parent. If copied and pasted elsewhere, its original conditional nature might be lost because it was dependent upon its parent to inherit the Product A condition.

Normally, an empty attribute indicates to inherit ancestor conditions by default, and this situation is not considered an error. Therefore, rule #3 is frequently disabled by users.

Rule #4 - Empty attribute not allowed

This rule flags any attributes that are not permitted to be unspecified, indicated by the lack of a “<no value>” inclusion in the active validation scheme. That is, if a particular attribute in the scheme does not have “<no value>” included in the list of valid values, the attribute is not permitted to be empty.

This rule is a largely a specialized version of rule #1, provided for convenience. For more information on rule #1, see [Validation rule #1 - Simple syntax](#) on page 3-17.

4: Scheme Setup And Other Main Settings

Schemes are the configuration workhorse of the plugin, where you specify exactly how you want it to color, filter, and/or validate your content. Rather than making manual decisions about how to process your content with each action, you put this logic into schemes and simply run the desired scheme whenever necessary.

You can have as many schemes as you want, and even share them at an enterprise level. For more information, see [About the main settings file](#) on page 2-8.

General information about schemes and categories

In most respects, a scheme is little more than a collection of attributes and values, and perhaps some additional options. When run, the plugin navigates the document structure tree(s) and stops at each element, comparing it to the attributes and values in the scheme. If they match, some respective action occurs, such as the coloring of the element during coloring or the preservation of content during filtering.

Because of their similarities, the three scheme types (coloring, filtering, validation) look very much the same and use the same editor (ABCM > Main Settings > Schemes). The scheme editor includes a drop-down menu that allows you to switch between scheme type that you are editing. All schemes are stored in the main settings file, described in more detail under [About the main settings file](#) on page 2-8.

The plugin also provides a higher level of categorizing schemes, known simply as scheme categories. A category is a collection of any number of coloring, filter, and validation schemes, and serves as a mechanism to help you keep schemes in order. The way you categorize schemes has no effect on how the plugin operates. Rather, it is

a basic feature that allows you to group common schemes together and prevent scheme lists from getting too long.

Although all schemes are constructed in a similar manner, the way they behave during processing may differ. For example, the details of attribute/value matching differs between coloring and filter schemes. These details are explored in the individual sections about each scheme type.

NOTE: Schemes are completely document- and book-independent. You can run any scheme of any type on any document or book. In most cases, you will have a number of schemes that you run on any given document set, and you may share schemes between different document sets. The manner in which you name and categorize your schemes is entirely up to you and no scheme is ever restricted to a particular file.

General scheme editing procedures

All schemes are edited using the scheme editor (ABCM > Main Settings > Schemes). A scheme is mostly just a collection of attributes and values, and the three scheme types look similar to each other in many ways. The following are some general tips to keep in mind while editing schemes:

- A scheme can contain multiple attributes and unique values for each. When you are looking at the attributes and values on the right, the list of values will always reflect the currently selected attribute only.
- When you add attributes and values to a scheme, the plugin provides a dialog box with a drop-down menu. This menu is prepopulated based on information found in your master attribute library. For more information, see [Master attribute library](#) on page 4-37.
- When you add coloring rules, the plugin provides a dialog box with a drop-down menu. This menu is prepopulated based on information found in your master colors list. For more information, see [Master colors list](#) on page 4-37.
- Like all ABCM dialog boxes, you can double-click an item in a scroll box to edit it.
- All schemes may include a set of advanced options, accessible by clicking the Advanced Options button. For more information on these options, see [Advanced scheme options](#) on page 4-34.

Coloring schemes

Coloring text according to conditions is mostly an authoring convenience, allowing you to see visually where your conditions are assigned. In this respect, the purposes of

coloring with the ABCM plugin are exactly the same as those associated with native conditional text.

A coloring scheme includes one or more “rules,” each of which indicates a color and the parameters to match in order to apply that color. The order of these rules is often critically important for achieving the desired results. For more information, see [Coloring rule order](#) on page 4-26.

While similar in concept to native conditional tag indicators, the process of conditional coloring with ABCM is functionally much different. Coloring with the plugin is a highly-customizable and specific process. With your schemes, you must indicate exactly what conditions should receive what color, including any details about condition overlap. There is no automatic magenta text with the plugin... if you want overlaps colored a certain way, you must specify as such. Furthermore, a scheme only colors the conditions you want, and ignores any others that might exist. In this manner, it is very different than native conditional text, which forces you to view one coloring pattern only, and all conditions at a time or none at all.

Because of this flexibility, you can have as many schemes as you want for any particular document or structure definition, and run whichever one applies the particular coloring pattern you want to see at the time. It is common to color content in different fashions depending on what you want to see, particularly if you have many conditions and/or conditional overlaps.

NOTE: For more information on running a coloring scheme, see [Coloring](#) on page 3-15.

Basic coloring scheme behavior

When you run a coloring scheme, the process starts at some element, and steps logically throughout each descendant element performing the coloring function. At each element, the plugin stops and compares the attributes/values on the element to those in the rules of the scheme. It goes down each rule in order, looking for a match. It applies the color assigned to the first matching rule it finds, if any. Once a rule is applied or all rules have been exhausted, the plugin steps to the next element and does the same thing until all elements are completed.

If you run coloring on a whole document or book, the starting point is the highest-level element. If you run it on a selection, the starting point is the selected element, or the element that contains the insertion point. All coloring is launched through the **ABCM > Coloring** menu or associated shortcuts. You are encouraged to become accustomed to the shortcut for coloring a selection (Esc 1 1), because you may use it frequently to refresh the area in which you are working.

Coloring scheme details

To create and/or edit a coloring scheme, you should work in the scheme editor (ABCM > Main Settings > Schemes). Be sure that you have the correct category and scheme type selected. For more information on general scheme editing procedures, see [General scheme editing procedures](#) on page 4-22.

A coloring scheme includes one or more rules, each of which contains a set of attributes and values. At a basic level, when the attributes/values of a rule match those on an element, the element receives the color assigned to the rule. There are, however, a number of technical details concerning coloring schemes, described in the following list:

- **Attribute/value matching criteria** - Rule matching behavior is very extensible with the **Match all values** option. For more information, see [Attribute/value matching criteria](#) on page 4-24.
- **Rule order** - Rule order is very important, because for any given element, the rules are processed in the order which they appear, and only the first matching rule is applied. For a more detailed explanation, see [Coloring rule order](#) on page 4-26.
- **<no value> and <any value>** - These standard items can be added to the values for any attribute, and are important to understand. For more information, see [<no value> and <any value> in a coloring scheme](#) on page 4-27.
- **Other formatting capabilities** - Coloring rules support the same additional style options as native conditional text, such as underlining and strikethrough. These options are specified on a rule-by-rule basis and are applied as simple format overrides, like coloring.
- **Changing a rule color** - The color assigned to a rule can be changed by selecting the rule in the list and double-clicking it or clicking Edit.
- **Override coloring of child elements** - This rule option causes all coloring of any child elements to be skipped, if the rule is matched. All child elements will receive the color of the matched element.

Attribute/value matching criteria

During coloring, each element is tested against each coloring rule in the scheme, until one matches or the rules are exhausted. Each rule has an independent setting called “Match all values,” which significantly affects the criteria required for an attribute/value match, as follows:

- **The option is checked** - If “Match all values” is checked, every single attribute and specified value in the rule must appear on the element in order to make a match. The element may have more attributes and values than appear in the rule, but it must at least have all those specified in the rule.
- **The option is NOT checked** - If the option is not checked, the rule will match if a single value from a single rule attribute is matched.

For example, consider the following element:



...and consider the following rule:

Rule color	Attribute(s)	Value(s)
Green	<code>outputformat</code>	PDF

This rule will match whether or not “Match all values” is checked, because it matches at least one value of the `outputformat` attribute, and it also happens that every value in the rule is found on the element. The rule doesn’t care about the `product` attribute at all, because it isn’t specified in the rule.

Similarly, consider the following rule:

Rule color	Attribute(s)	Value(s)
Green	<code>outputformat</code>	PDF
	<code>product</code>	ProductA

This rule will also match in either case, because every attribute and value in the rule is found on the element, so the state of the “Match all values” checkbox doesn’t matter.

Conversely, consider the following rule:

Rule color	Attribute(s)	Value(s)
Green	<code>outputformat</code>	PDF
	<code>product</code>	ProductA ProductC

This rule is different. If “Match all values” is unchecked, the rule will match, because at least one value in the rule is found on the element. However, if it is checked, the rule will not match, because “ProductC” is not found in the `product` attribute of the element.

The “Match all values” option and rule order are your primary tools for designed detailed and effective coloring schemes. For more information on rule order, see [Coloring rule order](#) on page 4-26.

Coloring rule order

Rule order in a coloring scheme is critically important, because for each element evaluated, the plugin will choose the first rule that matches and ignore the rest. Therefore, you must be sure that your most “specific” rules are near the top, and the more “general” rules are near the bottom.

For example, suppose you want to color all your “ProductA” content red, and all your “ProductB” content green. And, to indicate a mix of conditions, you want to color content for both products in blue. In this case, you might be tempted to create the following scheme:

Rule #	Rule color	Attribute(s)	Value(s)
1	Red	product	ProductA
2	Green	product	ProductB
3	Blue	product	ProductA ProductB

At first glance, this scheme seems to have all the rules you need. More than likely, though, you will never get to see your blue color applied. For an explanation, consider the following element:



This element is one that should be colored blue, according to your original intentions. However, the plugin will never get to the blue rule (#3), because the first rule (red) will always match first. Its only criterion is that the `product` element contains “ProductA”, which this element does. So, it colors the element red and never gets to your blue rule.

To make a scheme like this work, you must be more specific with rule order and use the “Match all values” option appropriately. Consider the same basic scheme, with the rules and options rearranged:

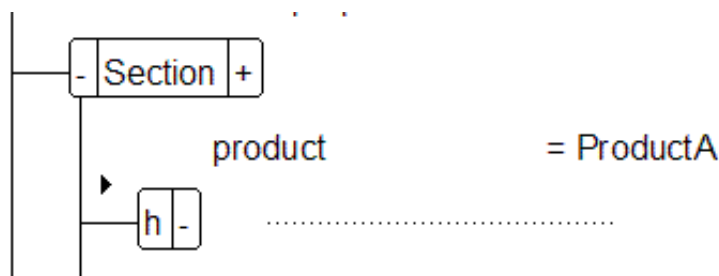
Rule #	Rule color	Attribute(s)	Value(s)	Options
1	Blue	product	ProductA ProductB	Match all values
2	Red	product	ProductA	
3	Green	product	ProductB	

Now, the “composite product” rule is at the top, so it will be evaluated first. And, it is forced to match all the values in order to qualify. In this case, the `Section` element will be properly colored blue, and any elements for ProductA or ProductB only will be colored red or green, respectively.

<no value> and <any value> in a coloring scheme

Like any scheme type, you can specify “<no value>” or “<any value>” for any attribute. These values, which automatically appear in the applicable drop-down menus, mean literally what they say: no value or any value. “No value” is synonymous with an empty attribute, and “any value” is synonymous with an attribute that contains any value.

As an example, consider the following element:



...and consider the following rule:

Rule color	Attribute(s)	Value(s)
Green	product	<any value>

This rule will match, because the `product` attribute contains any value. The converse is true, in that a specification of “<no value>” would only match if the attribute is empty. Note the following about <any value>/<no value>:

- If <any value> and <no value> both appear for any given attribute, it means literally “color me if I have this attribute, and it is either empty or specified.” In other words, it will color any element based on the mere presence of an attribute, regardless of its contents.
- <any value> and the “Match all values” option are generally incompatible, because the option requires an explicit list to work logically.

Where the colors come from

During a coloring action, if a rule matches, the plugin attempts to find the associated color by name in the document being processed. If it finds the color, it will retrieve the associated color definition and apply it. If it does not, it will warn you that the color does not exist.

For this reason, it is important that you specify colors in your schemes that your template(s) actually contains. To help prevent errors, you can customize the drop-down list of colors that appears in the scheme editor to reflect the colors that are valid for your documents. For more information, see [Master colors list](#) on page 4-37.

Other coloring scheme options and features

Note the following miscellaneous items about coloring schemes:

- **<refresh EDD> as a color** - The “<refresh EDD>” option always appears in the colors list, and can be used in the place of a color for any rule. This option will cause the plugin to refresh the EDD definition for any element it matches, removing all format overrides for that element and any descendants. No colors are applied, unless the EDD definition directs as such.
- **<skip> as a color** - The “<skip>” option always appears in the colors list, and can be used in the place of a color for any rule. This option will cause the plugin to do nothing if the rule matches, and simply step to the next element. This option is intended for certain specialized use only and may not be commonly found in schemes.
- **Override coloring of child elements** - This option is available on a rule-by-rule basis, near the “Match all values” option. If a rule matches and this option is specified, it causes the color to be applied to the respective element and all descendant elements, with no consideration for descendant attribute conditions. In essence, it causes the plugin to discontinue its walk down the current branch being processed, and back up to start down the next logical branch. Any descendant elements therefore remain unprocessed.

Filter schemes

Filtering is the process by which you produce conditional output. It is loosely analogous to the “Show/Hide” process for native conditional text, with far more flexibility and options.

A filter scheme is mostly a simple collection of attributes and values. During the filtering process, the plugin examines each element in a logical fashion, starting at the highest-level element and walking logically throughout each branch. If the attributes on the element match those in the scheme, the element is preserved. If they do not, the element and all descendants are removed or hidden.

A filter scheme is structurally more simple than a coloring scheme because it has no rules, only one set of attributes and values. This difference is because the filtering process is a simple yes or no decision... not one where multiple colors might need to be evaluated and applied.

Note the following important items about filter schemes:

- **You are specifying content to “keep,” not content to hide** - This aspect of filter schemes makes them fundamentally different than showing/hiding native conditional text. With the native Show/Hide dialog box, you must focus on what conditions to hide, in order to produce the output you want to keep. This logic is counterintuitive and is overcome by filter schemes, in which you specify what you want to stay in your output. All other conditions not specified are either ignored or hidden by default, depending on the logic of the scheme.
- **Attributes not specified in the scheme are ignored** - Only those attributes found in the scheme are evaluated, and only then if the respective element contains them. All other attributes on the element are ignored, regardless of their contents. In this way, you can set up schemes that only consider certain conditions. This process is much different than showing/hiding native conditional text, in which case you must always consider all conditions every time you want to show/hide, because they must all be dealt with in the Show/Hide dialog box.
- **Matching requires at least one value match for each scheme attribute** - If at least one value matches on the element between the attributes in the scheme and the attributes on the element, the element is preserved. That is, for each attribute in the scheme, at least one value must be found at the element, otherwise it is removed. For examples of this behavior, see [General filter scheme matching behavior](#) on page 4-30.
- **<no value> and <any value>** - These items, which mean literally what they say, may be critically important for proper scheme behavior. For more information, see [<no value> and <any value> in filter schemes](#) on page 4-31.

NOTE: For general information on running a filter, see [Filtering](#) on page 3-11.

General filter scheme matching behavior

The logic for matching (versus discarding) during a filter action is generally simple. For each element evaluated, the following rules apply:

- If an attribute is on the element and in the scheme, that attribute is evaluated. At least one value must match between the two, otherwise the element is flagged for removal.
- If an attribute in the scheme does not exist on the element, or vice versa, it is simply ignored.

For example, consider the following element:



...and consider the following scheme setup:

Attribute(s)	Value(s)
outputformat	PDF

During evaluation, the `Section` element will be preserved. It has an `outputformat` attribute which is also contained in the scheme, and at least one value (PDF) matches between the two. The `product` attribute is not included in the scheme, so the scheme doesn't care about it at all.

Conversely, the following scheme setup would cause the removal of the element:

Attribute(s)	Value(s)
outputformat	HTML

The following scheme setup would allow the preservation of the element, because at least one value matches on both scheme attributes:

Attribute(s)	Value(s)
outputformat	PDF
product	ProductA

The same applies to the following scheme. This scheme adds another value to match on the `product` attribute that the element doesn't have, but it doesn't matter because the ProductA value does match:

Attribute(s)	Value(s)
<code>outputformat</code>	PDF
<code>product</code>	ProductA ProductC

The following scheme would also preserve the element. The `customer` attribute is never evaluated because it doesn't exist on the element:

Attribute(s)	Value(s)
<code>outputformat</code>	PDF
<code>customer</code>	CustomerA

The following scheme, however, would not match and would flag the element for removal. Even though the `outputformat` attribute makes a match, the `product` attribute does not, and at least one match must occur for all attributes evaluated:

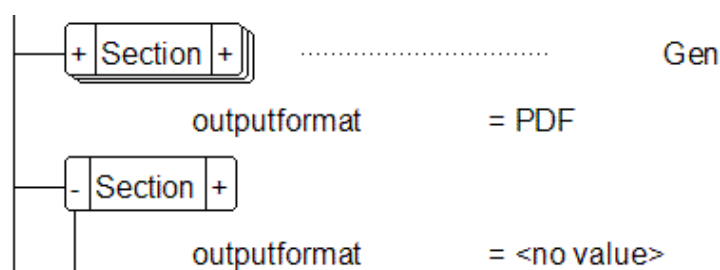
Attribute(s)	Value(s)
<code>outputformat</code>	PDF
<code>product</code>	ProductC

<no value> and <any value> in filter schemes

Like any scheme type, you can specify "<no value>" or "<any value>" for any attribute. These values, which automatically appear in the applicable drop-down menus, mean literally what they say: no value or any value. "No value" is synonymous with an empty attribute, and "any value" is synonymous with an attribute that contains any value.

These items, especially "<no value>", may be critically important for a properly-functioning filter scheme. An "unconditional" situation is frequently indicated by an empty attribute, analogous to the native conditional text practice of simply assigning no tag to unconditional content. Unlike native conditional text, however, the plugin does not mandate this assumption, and if you intend to use empty attributes to denote an unconditional situation, you must provide for this convention in your schemes using "<no value>" as applicable.

As an example, consider the following two elements:



The first element is designated for PDF output, while the second element has no specification. This convention would typically indicate that the second element is unconditional with regards to output format, and should always be preserved through any `outputformat`-based filter. Assuming this is the case, the scheme must specify the following to produce PDF output:

Attribute(s)	Value(s)
<code>outputformat</code>	<code><no value></code> PDF

With this scheme, both elements will be preserved. If the scheme failed to include “<no value>”, the second element would be filtered out, because “PDF” does not match a literal state of no value. Only the special “<no value>” flag does. Therefore, any architecture that uses empty attributes to denote an unconditional state must be incorporated with schemes that use “<no value>” to designate as such.

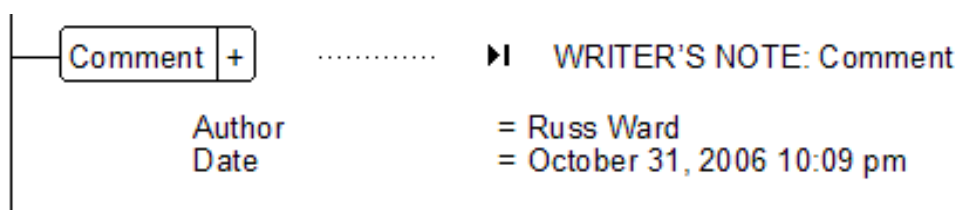
The “<any value>” flag is similar in concept, except that it matches if the respective attribute is populated with any value at all. The “<any value>” flag is likely to be used much less often, if at all.

Filtering out elements by type, using unique attribute names

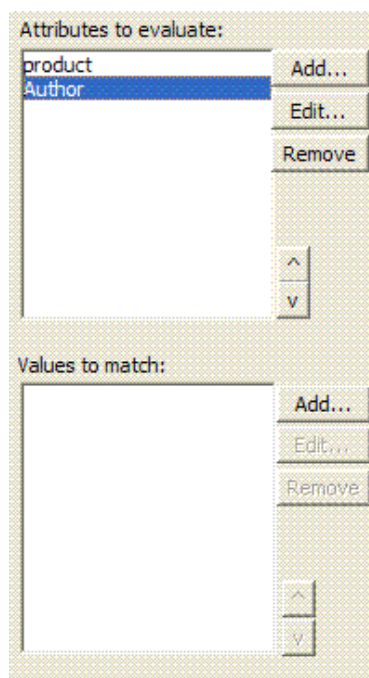
By way of scheme setup, the plugin provides a way to filter out all elements with a given attribute, regardless of its contents. This feature provides a way of filtering all elements of a certain type in a blanket fashion, if they are elements that you want removed from the output regardless of attribute contents.

A common usage of this feature is with authoring comments, when a structure definition reserves a special element for them. If you put authoring comments directly in your text, it is likely that you will want to filter them all out before producing deliverable output. And, it is likely that you will want them all out period, without any regard for attribute contents.

As an example, consider the following element:



Before producing output for customers, it is likely that the author would like to remove all `Comment` elements like this completely, regardless of attribute contents. To accomplish this with a filter scheme, you can specify an attribute and leave the Values box blank. This effectively says to the plugin, “There are no valid values at all for this attribute, so the attribute itself is invalid. Whenever the attribute is encountered, therefore, just remove the parent element.” The scheme dialog box might look something like the following:



To make this feature work effectively, the attribute you choose should be unique to the element(s) you are trying to filter out. For example, the attribute chosen in this example is `Author`. This scheme setup will cause the removal of any element with an `Author` attribute, so if you only intend for `Comment` elements to be removed in this fashion, the `Author` attribute must be unique to `Comment` elements.

The use of this convention allows great flexibility with storing non-publishable data in your documents, because once a scheme is set up, you can be assured that it will all be removed during filtering, every time. The risk that an errant comment will remain in the document because of misassigned conditional text is eliminated.

Validation schemes

Validation schemes direct the behavior of attribute validation feature, and are essentially a collection of the attributes and valid values that you use for your conditions. For a complete explanation of how the validation feature works, see [Attribute validation](#) on page 3-16.

Validation schemes are constructed with the same editor as other schemes, using the standard controls. Note the following about validation schemes:

- Because a validation scheme is a collection of valid attributes and values, you should have a separate scheme for each document set that uses different attributes and values for conditions. In other words, for any set of documents that uses the same exact attributes and values for setting conditions, there should be one unique validation scheme.
- <no value> and <any value> may be specified for attributes in a validation scheme, and like other schemes, they mean literally what they say. If you specify <no value> for an attribute, that attribute is not permitted to contain any values. If you specify <any value>, the attribute can contain any value, but it cannot be empty. Either of these specifications should appear alone, because combining them with explicit values and/or each other is illogical and would serve no purpose.
- The advanced scheme options apply, as with all schemes. For more information, see [Advanced scheme options](#) on page 4-34.

Advanced scheme options

For any scheme, you can access the advanced options by clicking the **Advanced Options** button under the schemes list, in the scheme editor. This section describes the options available, on a scheme-by-scheme basis.

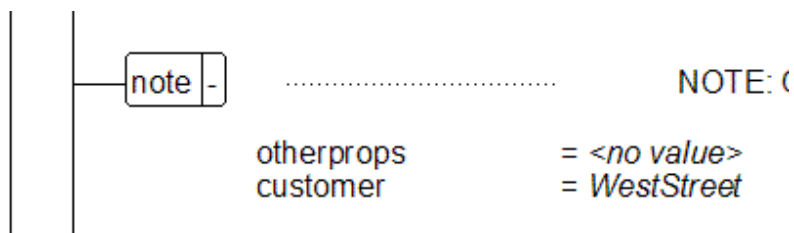
NOTE: Each scheme has its own independent set of advanced options. When you edit these options, you are editing them for the selected scheme only.

Advanced options include:

- **When evaluating attribute values, consider EDD-applied defaults** - See [Considering EDD-applied defaults](#) on page 4-35.
- **When evaluating attributes and values, ignore case-sensitivity** - See [Ignoring case-sensitivity](#) on page 4-35.
- **Consider whitespace as an attribute value delimiter** - See [Attribute values delimited by whitespace \(Tokenized strings\)](#) on page 4-35.
- **Process all flows** - See [Processing all flows](#) on page 4-37.

Considering EDD-applied defaults

In an EDD, you can specify default values for attributes which appear in the Structure View in italics. For example, the following `note` element has `customer` attribute with a default value of “WestStreet” assigned:



While they do appear in the Structure View, default values aren’t actually “real” and do not appear in the FrameMaker attribute editor. By default, ABCM will ignore them while processing, unless you check this advanced scheme option. If you do check it, all default values are regarded the same as any value that was explicitly set.

Ignoring case-sensitivity

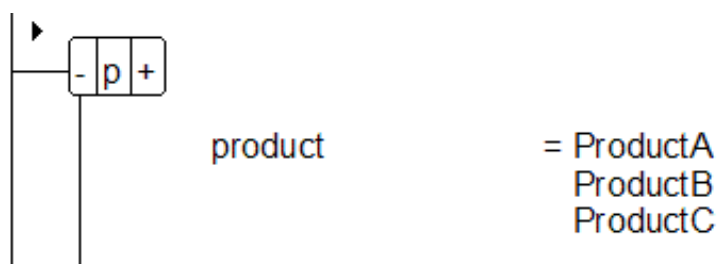
In most processes involving XML and other structured content, case-sensitivity is generally the rule. This is especially true when referring to markup qualities, such as attribute names and values. By default, all processing by ABCM adheres to strict case-sensitivity when comparing scheme parameters to attributes and values in your documents.

However, if you check the “ignore case-sensitivity” advanced option, ABCM will completely ignore the case of attributes and values during evaluation. For example, the attributes `PRODUCT`, `Product`, and `product` will look the same.

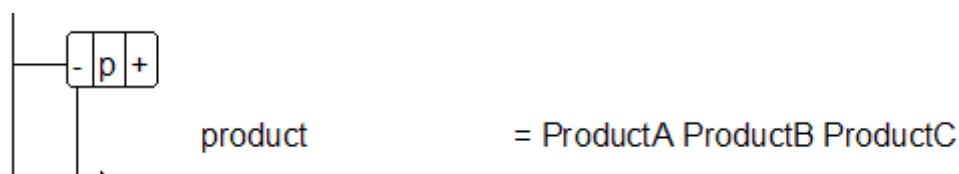
NOTE: This option is always off by default, and is recommended for special situations only. The case-sensitivity of markup is usually very important.

Attribute values delimited by whitespace (Tokenized strings)

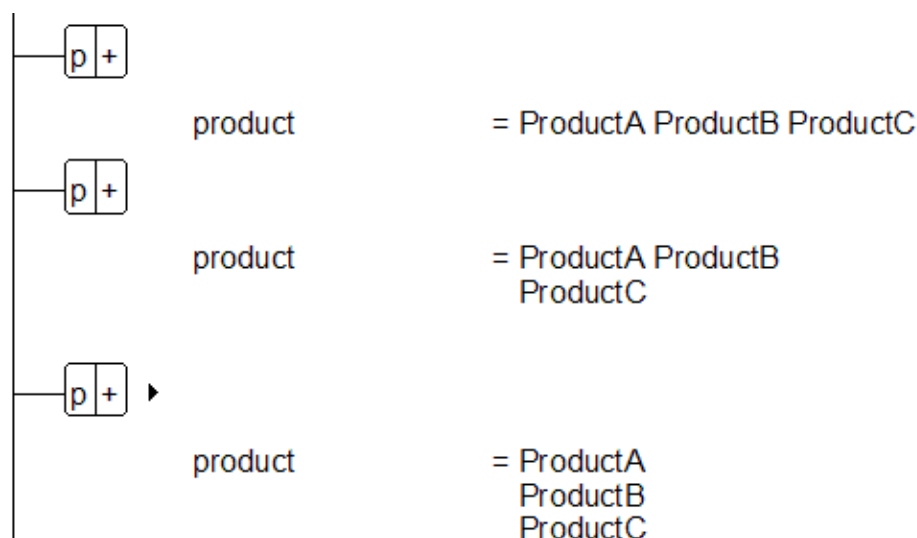
The FrameMaker interface makes it convenient to manage multiple values on a single attribute, separating individual values with a carriage return in the attribute editor and the Structure View. For example, the following `product` attribute has three separate values:



This convention is easily managed with a Strings-type attribute in an EDD and is recognized by ABCM. However, in many applications (especially those dealing with XML), multiple values are found on a single line, delimited with whitespace. For example, the same “conditionalization” could appear as follows:



With this advanced option checked, ABCM will recognize the individual values in this type of combined string (also called a *tokenized string*). That is, ABCM will recognize all individual *tokens* separated by whitespace, such that the following sample elements all appear the same to the plugin:



When checked, this option applies to all attributes and values, and only applies to tokenized values delimited by whitespace. Without this option checked, the default behavior is to ignore individual tokens and consider whole strings only.

Processing all flows

By default, a scheme will process the main flow only, normally flow A. With this option checked, however, all structured flows will be processed independently. This includes flows in text frames and on the master and reference pages. Any unstructured flows are ignored in either case.

Master attribute library

By selecting **Main Settings > Attribute Library**, you can define default attributes and values to populate the drop-down dialog boxes in the scheme editor. For example, when you add an attribute to a scheme, the drop-down dialog box will contain the names of any attribute specified in this library. This list is a scheme editing convenience only and has no effect on the coloring, filtering, or validating of content. It is provided because the same attributes and values are typically reused between multiple schemes, and spelling and case-sensitivity are very important.

NOTE: The scheme editor dialog boxes will allow you to type any values you want. You are not restricted to the entries in the master attribute list.

Master colors list

By selecting **Main Settings > Colors**, you can define a set of default colors to appear in the drop-down dialog box when adding a new rule to a coloring scheme. This feature is provided as a convenience because the same colors are typically reused frequently, and spelling and case-sensitivity are very important.

This list is intended to enhance the scheme editing process only. It has no effect on the coloring feature of the plugin, and it has no inherent correlation with any templates or color definitions. If you use it, you should simply populate it with the names of the colors that appear in your templates, especially those that you intend to use in coloring schemes. In the scheme editor, you can always type any color you want, and you are never restricted to this list alone.

Migrating Sourcerer settings

If you are a Sourcerer user, ABCM provides a means of automatically migrating your Sourcerer master settings to ABCM. These settings include:

- All Sourcerer filter, coloring, and validation schemes
- The current Sourcerer master attributes library

To perform the migration, you should open your Sourcerer master settings file and select **ABCM > Main Settings > Migrate Sourcerer Settings**. Note the following about running this function:

- The settings will be migrated to the ABCM main settings file that your preferences point to. Following the migration, you should be able to open the scheme editor and see them.
- The schemes will be placed in a new category named “Sourcerer.” If that category already exists, ABCM will add a number to the category name and increment it as necessary to find a unique name. For example, “Sourcerer2”, “Sourcerer3”, etc.
- The master attributes will be transferred as-is. If any of the attributes currently exist in your ABCM settings, any new values for them from the Sourcerer settings will be appended.
- Sourcerer did not have a feature comparable to the colors list. You will need to create that manually with ABCM.

At a core functional level, ABCM works very similarly to Sourcerer. In most cases, you should be able to migrate your settings and begin using ABCM in the place of Sourcerer right away.

5: External Calls to ABCM

Like many FrameMaker plugins, you can make external calls to ABCM to invoke certain plugin activities, often for purposes of automation. Specifically, you can call this plugin to:

- Set the active filter, coloring, and validation schemes ([SetScheme](#) on page 5-50)
- Set various options for filtering and other processing ([SetParm](#) on page 5-48)
- Filter-check an element ([FilterCheckElement](#) on page 5-44)
- Filter a document or book ([FilterFile](#) on page 5-45)
- Restore a document or book from a source-file filter ([RestoreFile](#) on page 5-47)
- Color an element ([ColorElement](#) on page 5-41)
- Color a document or book ([ColorFile](#) on page 5-43)
- Validate an element ([ValidateElement](#) on page 5-51)

These functions are fully exposed through the FrameMaker API and allow you to programmatically mimic the behavior of the plugin as used interactively through the GUI.

How to send an external call to ABCM

To call ABCM, you can use one of three methods:

- **With the FDK `F_ApiCallClient()` function, from another API client** If you are working on another FDK client, you can use `F_ApiCallClient()` to call ABCM. This function is part of the normal FDK library and does not require any changes to your normal project settings. For more information on the function itself, see the *FDK Developer's Reference* provided by Adobe with the FDK.
- **With FrameScript** FrameScript®, a scripting tool by Finite Matters, Ltd®, has a comparable function for calling FDK clients, `CallClient`. When called from FrameScript, ABCM behaves identically to a regular API call.
- **With FrameAC** FrameAC by Mekon® (www.mekon.com) is a COM-based utility that enables developers to use Visual Basic to control FrameMaker. FrameAC also provides the ability to script calls to other API clients.

For any supported operation, you pass a string to ABCM which contains a command and any applicable parameters, and ABCM sends back a numeric code indicating the results. The syntax of these strings is the same for either API or scripting calls, and is explained in detail in this document.

NOTE: The call descriptions and examples in this document are written from an FDK/API perspective, using `F_ApiCallClient()`. If you are using FrameScript or FrameAC, the basic call syntax will be the same, sent using the mechanism supported by the respective tool.

General information on external calls

Before you attempt to call ABCM, note the following:

- Certain commands require that you specify a document or book, which can be done by one of three methods. For more information, see [Specifying document and book arguments](#) on page 5-41.
- The default delimiter string between arguments in a call to ABCM is three dashes (---). This delimiter can be changed in your preferences. In this document, the syntax descriptions of external calls use the default delimiter, which you should adjust accordingly if you decide to change the delimiter. For more information on setting the delimiter string, see [Preferences](#) on page 2-7.
- Several calls to ABCM return zero (0) to indicate a command failure, consistent with the behavior of other FDK functions. However, `F_ApiCallClient()` also returns zero if it fails to communicate at all with the specified API client. If you aren't sure whether your calls are reaching ABCM, you can call the special [Hello](#) command to verify that communications are getting through.

- With the exception of scheme categories and names, call string arguments are generally not case-sensitive. For example, to set an active scheme, you can send any case variation of the `SetScheme` command name, such as `SETSCHEME` or `SetScheme`.
- To effectively use the external interface to ABCM, you should be familiar with the functionality and workflow of the plugin through the GUI.

Specifying document and book arguments

When a document or book identifier is required, you may use any of the following three methods:

- **An object handle ID** - The integer form of the `F_ObjHandleT` object ID for the file.
- **A filename** - A non-qualified filename, such as `MyDocument.fm`.
- **A file path** - A fully-qualified file path, such as:

```
C:\MyDocs\MyDocument.fm
```

With this method, you may substitute forward-slashes for backslashes. For example:

```
C:/MyDocs/MyDocument.fm
```

In all cases, the file must be currently open. ABCM will not open any files.

Specifying Boolean arguments

When an argument requires a Boolean true or false, you can specify it as follows:

- For **true**, you can specify `1`, `true`, or any word that begins with “t”, including just `t`.
- For **false**, you can specify `0`, `false`, or any word that begins with “f”, including just `f`.

Boolean arguments are not case-sensitive.

Call reference

This section details the external calls you can make to ABCM.

ColorElement

Colors an element according to the active coloring scheme.

Syntax

```
F_ApiCallClient( "ABCM",  
  "ColorElement---Document---ElemId---IncludeDescendants---DoWarnings" );
```

where:

<i>Document</i>	Document that contains the element to be colored. For more information on specifying this parameter, see Specifying document and book arguments on page 5-41.
<i>ElemId</i>	The F_ObjHandleT object ID of the element to color.
<i>IncludeDescendants</i>	(Boolean) Indicates whether to evaluate and color any descendant elements. If set to False, descendant elements may receive any coloring applied to the main element only. For more information on setting Boolean arguments, see Specifying Boolean arguments on page 5-41.
<i>DoWarnings</i>	(Boolean) Indicates whether to perform interactive user prompting. If set to False, no message boxes are produced under any conditions, including critical errors. For more information on setting Boolean arguments, see Specifying Boolean arguments on page 5-41.

Usage description

`ColorElement` evaluates the specified element against the active coloring scheme and applies any formatting as applicable. The command requires that an active coloring is scheme set, perhaps with [SetScheme](#).

Returns

Value	Meaning
0	Communication with ABCM appears to have failed. Use Hello to test connectivity.
1	Coloring occurred successfully.
101	Unrecognized command. Check the syntax of the command itself.
103	Incorrect number of arguments sent with command.
104	Bad document argument. See Specifying document and book arguments on page 5-41.
105	Bad element ID.
106	Bad category and/or scheme name. This error will occur if an active coloring scheme is not set before the command is run.

Value	Meaning
116	Coloring failed for an unknown reason.
117	Interactive user cancellation occurred.

ColorFile

Colors a document or book according to the active coloring scheme.

Syntax

```
F_ApiCallClient("ABCM", "ColorFile---File---DoWarnings");
```

where:

<i>File</i>	Document or book to be colored. For more information on specifying this parameter, see Specifying document and book arguments on page 5-41. NOTE: If you specify a book, only currently-open chapter files will be processed. Closed book components are ignored.
<i>DoWarnings</i>	(Boolean) Indicates whether to perform interactive user prompting. If set to False, no message boxes are produced under any conditions, including critical errors. For more information on setting Boolean arguments, see Specifying Boolean arguments on page 5-41.

Usage description

`ColorElement` runs the active coloring scheme on the specified document or book. The command requires that an active coloring scheme is set, perhaps with [SetScheme](#).

Returns

Value	Meaning
0	Communication with ABCM appears to have failed. Use Hello to test connectivity.
1	Coloring occurred successfully.
101	Unrecognized command. Check the syntax of the command itself.
103	Incorrect number of arguments sent with command.
104	Bad file argument. See Specifying document and book arguments on page 5-41.

Value	Meaning
106	Bad category and/or scheme name. This error will occur if an active coloring scheme is not set before the command is run.
108	No structure found in main flow. This error only occurs if you are processing a single document, and the “Process all flows” scheme option is turned off in the active scheme.
116	Coloring failed for an unknown reason.
117	Interactive user cancellation occurred.

Hello

Tests whether ABCM is initialized and receiving external calls.

Syntax

```
F_ApiCallClient("ABCM", "Hello");
```

Returns

Value	Meaning
0	ABCM is not initialized and/or communication failed. Possible causes include: <ul style="list-style-type: none"> ABCM is not running at all. Check the FrameMaker interface for an ABCM menu. Your call uses a syntax that differs from the plugin name in the <code>maker.ini</code> file. In the ABCM installation instructions, the following line is to be entered into <code>maker.ini</code>: <pre>ABCM=Standard,ABCM,WestStreet\ABCM.dll,structured</pre> Whatever string you use to call ABCM (as the first argument of <code>F_ApiCallClient()</code> must match the name assigned there.
1	ABCM is ready.

FilterCheckElement

Checks the specified element against the active filter scheme and returns whether the element should be “filtered out” or not.

Syntax

```
F_ApiCallClient("ABCM", "FilterCheckElement---Document---ElemId");
```

where:

<i>Document</i>	Document that contains the element to be checked. For more information on specifying this parameter, see Specifying document and book arguments on page 5-41.
<i>ElemId</i>	The F_ObjHandleT object ID of the element to check.

Usage description

`FilterCheckElement` evaluates a single element against the active filter scheme and determines whether the element should be hidden, or in other words, filtered out. It does not perform any filtering action itself, returning a flag only. This command requires a valid, active filtering scheme, perhaps set with [SetScheme](#).

NOTE: If you want ABCM to perform a whole file filter, including all the functionality associated with an interactive GUI filtering action, use [FilterFile](#) instead.

Returns

Value	Meaning
0	Element should be preserved. NOTE: This value may also be returned if the currently-active filter scheme is not valid. Be sure to call SetScheme successfully before running this command the first time.
1	Element should be filtered out.
101	Unrecognized command. Check the syntax of the command itself.
103	Incorrect number of arguments sent with command.
104	Bad document argument. See Specifying document and book arguments on page 5-41.
105	Bad element ID.
106	Bad category and/or scheme name. This error will occur if an active filtering scheme is not set before the command is run.

FilterFile

Filters a document or book according to the active filtering scheme and current filter parameters.

Syntax

```
F_ApiCallClient("ABCM", "FilterFile---File---DoWarnings");
```

where:

File

Document or book to be filtered. For more information on specifying this parameter, see [Specifying document and book arguments](#) on page 5-41.

NOTE: If you specify a book, all book components must be currently open.

DoWarnings

(Boolean) Indicates whether to perform interactive user prompting. If set to False, no message boxes are produced under any conditions, including critical errors.

NOTE: Interactive prompting does not include the Filter dialog box.

For more information on setting Boolean arguments, see [Specifying Boolean arguments](#) on page 5-41.

Usage description

`FilterFile` filters an entire document or book and returns the object ID of the filtered file. The filter occurs in accordance with the current filter settings which can be set beforehand with `SetParm` and `SetScheme`. The filter scheme and other settings can have a significant impact on how the filter behaves and you should be sure that you understand and set them properly.

If you filter a single document successfully, the command returns the ID of the filtered document. In the case of a duplicate file filter, this is the ID of the new, filtered file. For a source file filter, it returns the same ID that you originally sent (if you sent the file parameter in ID form).

If you filter a book, the behavior is the same. Note, however, that a duplicate file book filter requires a target folder to receive the duplicated book. It is critically important that you set this parameter before running the filter and make absolutely sure that you have done it correctly. The filter will overwrite any files it finds in the target folder. You can set the target folder path with `SetParm`, along with other important filter settings.

In the unlikely event that a book filter fails in the middle of processing, you should capture this error and close all affected files without saving changes. A failure in the middle of a book filter will leave your files in an unpredictable state.

For more information on filter types, see [Filter types - Source versus duplicate file](#) on page 3-12. For more information on other filter options, see [Launching a filter](#) on page 3-12.

NOTE: Following a source file filter, you can restore the document or book with [RestoreFile](#).

Returns

Value	Meaning
0	Communication with ABCM appears to have failed. Use Hello to test connectivity.
101	Unrecognized command. Check the syntax of the command itself.
103	Incorrect number of arguments sent with command.
104	Bad file argument. See Specifying document and book arguments on page 5-41.
106	Bad category and/or scheme name. This error will occur if an active filtering scheme is not set before the command is run.
111	An interactive user cancellation occurred, or some other unknown filter error occurred.
112	The file could not be duplicated, for an unknown reason. This error is only applicable to duplicate file filters.
113	One or more book components are not open, applicable only to book filters.
114	The specified path for the filtered book is the same as the book to be filtered. The filter must abort because it would otherwise overwrite the original book with the filtered book, permanently deleting content. This error is only applicable to duplicate file book filters. For more information on setting this parameter, see SetParm
115	The specified path for the filtered book is inaccessible or does not exist. This error is only applicable to duplicate file book filters. For more information on setting this parameter, see SetParm .
Any number over 1000	An integer form of the object ID for the filtered file, indicating a successful filter.

RestoreFile

Restores a document or book following a source-file filter.

Syntax

```
F_ApiCallClient("ABCM", "RestoreFile---File");
```

where:

File

Document or book to be restored. For more information on specifying this parameter, see [Specifying document and book arguments](#) on page 5-41.

NOTE: If you specify a book, only currently-open book components are restored. ABCM does not open any files.

Usage description

`RestoreFile` removes the Hidden condition from a document or book as applied during a source-file filter. This command is not applicable if you use duplicate-file filters. For more information, see [Restoring a document or book](#) on page 3-14.

Returns

Value	Meaning
0	Communication with ABCM appears to have failed. Use Hello to test connectivity.
1	Restoration occurred normally..
101	Unrecognized command. Check the syntax of the command itself.
103	Incorrect number of arguments sent with command.
104	Bad file argument. See Specifying document and book arguments on page 5-41.

SetParm

Sets a parameter that affects ABCM processing, such as filter parameters.

Syntax

```
F_ApiCallClient("ABCM", "SetParm---Parm---Value");
```

where:

Parm

and

Value

Parameter and value to set. For a list of parameters and valid values, see, see [Usage description](#) on page 5-49.

Usage description

SetParm is a means to set important parameters via external calls. These parameters would typically be set by a user in a dialog box during an interactive session, and represent important settings that can significantly affect how ABCM processes content.

The following table lists the parameters supported by this command and the valid values for each:

Parameter	Description	Valid values
Filter_Type	Filter type, normally set in the Filter dialog box. See Launching a filter on page 3-12.	<ul style="list-style-type: none"> • D or Duplicate • S or Source
Filter_SaveFirst	Option to save all files prior to a filter action, normally set in the Filter dialog box. See Launching a filter on page 3-12.	Boolean true/false. See Specifying Boolean arguments on page 5-41
Filter_RemoveOverrides	Option to remove format overrides following a filter action, normally set in the Filter dialog box. See Launching a filter on page 3-12.	Boolean true/false. See Specifying Boolean arguments on page 5-41
Filter_Path	<p>Target path for a duplicate-file book filter, normally set graphically in the Filter dialog box. For more information, see Launching a filter on page 3-12.</p> <p>NOTE: This parameter must be set before you can run a duplicate-file book filter</p>	<p>A valid folder path. Note the following:</p> <ul style="list-style-type: none"> • You can optionally use forward slashes instead of back slashes. For example: C : /MyDocs / • You MUST include the trailing slash. See the example above. • You MUST NOT include the target file name. Specify the folder path only. • You should be VERY CAREFUL to set this property correctly. A book filter will overwrite files in the target folder.

Parameter	Description	Valid values
Val_Rule1Active	Specifies whether the respective validation rule is active, normally set in the Preferences dialog box. See Preferences on page 2-7.	Boolean true/false. See Specifying Boolean arguments on page 5-41
Val_Rule2Active		
Val_Rule3Active		
Val_Rule4Active		
Val_ApplyStrikethrough	Specifies whether validation should apply strikethrough text, normally set in the Preferences dialog box. See Preferences on page 2-7.	Boolean true/false. See Specifying Boolean arguments on page 5-41

NOTE: Once a parameter is set, it remains set for the duration of the session unless changed.

Returns

Value	Meaning
0	Communication with ABCM appears to have failed. Use Hello to test connectivity.
1	Parameter set successfully.
101	Unrecognized command. Check the syntax of the command itself.
103	Incorrect number of arguments sent with command.
109	Unrecognized parameter.
110	Invalid value.

SetScheme

Sets the active coloring, filter, or validation scheme.

Syntax

```
F_ApiCallClient( "ABCM",
    "SetScheme---SchemeType---Category---Scheme" );
```


where:

<i>SchemeType</i>	Scheme type, one of: <ul style="list-style-type: none"> • F or Filter • C or Coloring • V or Validation
<i>Category</i>	Case-sensitive scheme category.
<i>Scheme</i>	Case-sensitive scheme name.

Usage description

`SetActiveScheme` sets the active scheme prior to a coloring, filtering, or validation action. It reads the data from your local copy of the main settings file. Once a scheme is set, it remains set until you change it.

Returns

Value	Meaning
0	Communication with ABCM appears to have failed. Use Hello to test connectivity.
1	Parameter set successfully.
101	Unrecognized command. Check the syntax of the command itself.
102	Bad scheme type argument.
103	Incorrect number of arguments sent with command.
106	Unrecognized category and/or scheme.
107	Main settings file critical error, possibly missing or corrupt.

ValidateElement

Validates the specified element against the active validation scheme and returns the first rule violated, if any.

Syntax

```
F_ApiCallClient("ABCM", "ValidateElement---Document---ElemId");
```

where:

<i>Document</i>	Document that contains the element to be checked. For more information on specifying this parameter, see Specifying document and book arguments on page 5-41.
<i>ElemId</i>	The F_ObjHandleT object ID of the element to check.

Usage description

`ValidateElement` evaluates a single element against the active validation scheme and returns an integer representing the first rule found that was violated, if any. It operates using the currently-active validation rules and validation options, all of which can be set with `SetParm`. It does not create any reports.

Note that this command only returns a single integer and therefore can only return a single rule number. An element, however, can violate more than one rule at once. This command will simply return the first rule violation it finds, but you should be aware that more violations may exist.

Returns

Value	Meaning
0	Element's attributes are valid.
1 - 4	Number of the validation rule that the element violated. See Attribute validation on page 3-16.
101	Unrecognized command. Check the syntax of the command itself.
103	Incorrect number of arguments sent with command.
104	Bad document argument. See Specifying document and book arguments on page 5-41.
105	Bad element ID.
106	Bad category and/or scheme name. This error will occur if an active filtering scheme is not set before the command is run.
119	An unknown error occurred during validation.

Detailed example—Calling ABCM (FDK)

The following example contains a C language code sample for use with the FDK. It is a self-contained function that has been designed for and tested against the `External_Calls_Sample.fm` file, found in your `ABCM_SampleFiles` subfolder. For proper operation, this function relies on `External_Calls_Sample.fm` to be in its

original state, and the external call schemes to be unaltered in the `Samples` category. This category is found in the default main settings file provided with the ABCM installation.

With `External_Calls_Sample.fm` active, this function does the following:

1. Sets the active coloring scheme
2. Colors the original document
3. Sets up and performs a duplicate-file filter on the original document
4. Colors the new, filtered duplicate
5. Closes the duplicate
6. Sets up and runs a source-file filter on the original document
7. Restores the original and removes coloring
8. Retrieves the ID of the first `Section` element, colors it, then filter checks it
9. Retrieves the ID of the second `Section` element, colors it, then filter checks it

NOTE: You can also find an electronic copy of this function in the `External_Calls_Sample.c` file which installs into the `ABCM_SampleFiles` subfolder, under `WestStreet`.

```
VoidT ABCM_Sample()
{
    F_ObjHandleT docId,
        docId2,
        flowId,
        elemId;
    UCharT msg[512];

    IntT returnVal;

    UIntT i;

    //Let's store the ID of the active document, which MUST BE the
    //External_Calls_Sample.fm file for this whole function to work
    docId = F_ApiGetId(0, FV_SessionId, FP_ActiveDoc);
    if(!docId) return;

    //Message
    F_ApiAlert("Preparing to color. Click OK to set the active scheme...",
        FF_ALERT_CONTINUE_WARN);

    //Set the active coloring scheme. Let's do a bogus scheme first, just as
    //an example of a call failure
    returnVal = F_ApiCallClient("ABCM", "SetScheme---C---BogusCategory---BogusScheme");

    //send a message about the failure. Should indicate a return value of 106
    F_Sprintf(msg, "The command to set the scheme failed, because the category\
and scheme name were bogus. The call returned the following integer: %d\n\n\

```

```
Click OK to try again.",
    returnVal);

F_ApiAlert(msg, FF_ALERT_CONTINUE_WARN);

//Set the active coloring scheme for real this time
returnVal = F_ApiCallClient("ABCM", "SetScheme---C---Samples---External_Call_Sample");

//Evaluate the response. We're going to back out if something went wrong.
if(returnVal == True)
    F_ApiAlert("The scheme was set successfully. Click OK to launch the coloring.",
        FF_ALERT_CONTINUE_WARN);

else
{
    F_Sprintf(msg, "The command failed, returning the following integer: %d\n\n\
Your main settings do not include the factory samples necessary for this test.\
The test will abort.", returnVal);

    F_ApiAlert(msg, FF_ALERT_CONTINUE_WARN);

    return;
}

//Launch the coloring.
returnVal = F_ApiCallClient("ABCM", "ColorFile---External_Calls_Sample.fm---False");

//Let's make sure it worked, before continuing. If you forgot to open the file
//beforehand, this would have caused it to fail.
if(returnVal == True)
    F_ApiAlert("The file was successfully colored. Click OK to launch the filtering.",
        FF_ALERT_CONTINUE_WARN);

else
{
    F_Sprintf(msg, "The coloring failed. Did you forget to open the file?",
        returnVal);

    F_ApiAlert(msg, FF_ALERT_CONTINUE_WARN);

    return;
}

//Set the filtering scheme
returnVal = F_ApiCallClient("ABCM", "SetScheme---F---Samples---External_Call_Sample");

//Set the filter type to duplicate file
returnVal = F_ApiCallClient("ABCM", "SetParm---Filter_Type---D");

//Set the "save files first" option to off
returnVal = F_ApiCallClient("ABCM", "SetParm---Filter_SaveFirst---0");

//Filter the document
returnVal = F_ApiCallClient("ABCM", "FilterFile---External_Calls_Sample.fm---False");

//If it filtered successfully, we should have received an object ID back of the
//filtered duplicate. Just as a sample of what we can do with it, let's color the
//filtered duplicate now.
```

```
//Let's save the document ID first.
if(returnVal > 1000)
{
    docId2 = (F_ObjHandleT)returnVal;
    F_ApiAlert("The file was successfully filtered. Click OK to color the new file.",
        FF_ALERT_CONTINUE_WARN);
}
else
{
    F_ApiAlert("Something went wrong. Aborting the test.", FF_ALERT_CONTINUE_WARN);
    return;
}

F_Sprintf(msg, "ColorFile---%d---False", docId2);
returnVal = F_ApiCallClient("ABCM", msg);

//Get ready to close the filtered duplicate
F_ApiAlert("The duplicate file was colored. Click OK to close it.",
    FF_ALERT_CONTINUE_WARN);

F_ApiClose(docId2, FF_CLOSE_MODIFIED);

//Now let's do a source-file filter

F_ApiAlert("The duplicate file was closed. Now click OK to perform a\
source-file filter on the original document.", FF_ALERT_CONTINUE_WARN);

//Set the filter type to source file
returnVal = F_ApiCallClient("ABCM", "SetParm---Filter_Type---S");

//Filter the document
returnVal = F_ApiCallClient("ABCM", "FilterFile---External_Calls_Sample.fm---False");

//Report
F_ApiAlert("The file was filtered. Click OK to restore it.",
    FF_ALERT_CONTINUE_WARN);

//Restore the filtered file.
returnVal = F_ApiCallClient("ABCM", "RestoreFile---External_Calls_Sample.fm");

//and let's remove the coloring
F_ApiSimpleImportElementDefs(docId, docId,
    FF_IED_REMOVE_OVERRIDES | FF_IED_REMOVE_BOOK_INFO);

//Just for kicks, let's do some element-level functions.

//Let's get the ID of the first Section element,
//the one tagged for Product A.
flowId = F_ApiGetId(FV_SessionId, docId, FP_MainFlowInDoc);
elemId = F_ApiGetId(docId, flowId, FP_HighestLevelElement);
elemId = F_ApiGetId(docId, elemId, FP_FirstChildElement);
for(i = 0; i < 5; i++)
    elemId = F_ApiGetId(docId, elemId, FP_NextSiblingElement);

//Prompt
F_ApiAlert("The code has retrieved the ID of the first Section element.\
Click OK to color it.", FF_ALERT_CONTINUE_WARN);
```

```
//Color the first Section element
F_Sprintf(msg, "ColorElement---%d---%d---True---False", docId, elemId);
returnVal = F_ApiCallClient("ABCM", msg);

//Prompt
F_ApiAlert("The element has been colored. Click OK to filter check it.",
    FF_ALERT_CONTINUE_WARN);

//Filter-check the first Section element. It should return a value of
//zero, meaning that the element should not be filtered out. In other
//words, the active coloring scheme allows it to stay.
F_Sprintf(msg, "FilterCheckElement---%d---%d", docId, elemId);
returnVal = F_ApiCallClient("ABCM", msg);

F_Sprintf(msg, "The filter check returned: %d.\n\n\
0 = Keep\n1 = Filter out", returnVal);
F_ApiAlert(msg, FF_ALERT_CONTINUE_WARN);

//Do the same thing for the second Section element. This one should
//return a flag to filter it out, like it was filtered out when we
//ran the full document filter earlier.

//Get the ID of the second Section element
elemId = F_ApiGetId(docId, elemId, FP_NextSiblingElement);

//Prompt
F_ApiAlert("The code has retrieved the ID of the second Section element.\
Click OK to color it.", FF_ALERT_CONTINUE_WARN);

//Color the second Section element
F_Sprintf(msg, "ColorElement---%d---%d---True---False", docId, elemId);
returnVal = F_ApiCallClient("ABCM", msg);

//Prompt
F_ApiAlert("The element has been colored. Click OK to filter check it.",
    FF_ALERT_CONTINUE_WARN);

//Filter-check the second Section element. It should return a value of
//one, meaning that the element should be filtered out. In other words,
//it is content to be hidden.
F_Sprintf(msg, "FilterCheckElement---%d---%d", docId, elemId);
returnVal = F_ApiCallClient("ABCM", msg);

F_Sprintf(msg, "The filter check returned: %d.\n\n\
0 = Keep\n1 = Filter out", returnVal);
F_ApiAlert(msg, FF_ALERT_CONTINUE_WARN);

F_ApiAlert("All done!", FF_ALERT_CONTINUE_WARN);
}
```