ptc

arbortext®

Content Pipeline Guide

8.3.0.0

Contents

About this Guide

This guide describes the content pipeline concepts and describes the components that make up a pipeline. It provides examples of using AOM and ACL to manipulate filters and pipelines, along with providing information on adding web publishing to a document type, switching XSL processing engines, and other topics.

Prerequisite Knowledge

This guide assumes the person who will be creating and implementing filters and pipelines is a programmer who has some experience with SAX, AOM, C++, and Java. Refer to Where to Get More Information on page 8 for resources.

Technical Support

The PTC eSupport portal provides resources and tools to support your Arbortext implementation:

www.ptc.com/support/

If you encounter problems using this product, contact PTC Technical Support. For complete details, see "Opening a Case" on the **Processes** tab of the PTC Customer Support Guide:

www.ptc.com/support/csguide/Processes

You must have a Service Contract Number (SCN) before you can receive technical support. If you do not know your SCN, see "Preparing to contact TS" on the **Processes** tab of the PTC Customer Support Guide for information about how to locate it.

Help for PTC Products

You can access PTC product help using the following resources:

- Online Help

    Click **Help** from the user interface for online help available for the product.

- **Reference Documents** website — Information assistance in PDF format:

    www.ptc.com/support/refdoc

    You must have a Service Contract Number (SCN) before you can access the **Reference Documents** site. If you do not know your SCN, see "Preparing to contact TS" on the **Processes** tab of the PTC Customer Support Guide for information about how to locate it:

    www.ptc.com/support/csguide

- Help Center

    Use the following link to access all PTC Help Centers:

    support.ptc.com/apps/help_center/help/

Global Services

PTC Global Services delivers the highest quality, most efficient and most comprehensive deployments of the PTC Product Development System including Creo, Windchill, Arbortext, and PTC Mathcad. PTC's Implementation and Expansion solutions integrate the process consulting, technology implementation, education and value management activities customers need to be successful. Customers are led through Solution Design, Solution Development and Solution Deployment phases with the continuous driving objective of maximizing value from their investment.

Contact your PTC sales representative for more information on Global Services.

Comments

PTC welcomes your suggestions and comments on its documentation:

- To send feedback about a topic you are viewing in the PTC Arbortext Help

    Center, click the send feedback icon in the top right corner of the topic. The title of the help topic you were viewing when you clicked the icon is automatically included with your feedback.

- You can also send an email to documentation@ptc.com.

    To help us more quickly address your concern, include the name of the PTC product and its release number with your comments. If your comments are about a specific help topic or book, include the title.

1

Content Pipeline Guide

The *Content Pipeline Guide* provides detailed information about the SAX filter pipeline that Arbortext Editor and Arbortext Publishing Engine use to transform documents. A pipeline, which is a sequence of filters that perform a task in several step, is configured by a content configuration file (CCF) and exposed as an AOM composer object.

A pipeline can perform many kinds of document manipulation, not just publishing. For example, Arbortext Editor uses a pipeline to display a profiled document in the Edit window and validate a document against a schema.

Conventions Used in this Guide

In addition to the conventions described earlier, this guide uses the following notational conventions:

- `Bold text` represents an exact reference, such as names of methods, classes, and attributes or paths and file names. For example:

  You can use the `hide` method to hide the dialog box.

  See the sample file `demo.xml`.

- File paths are typically given using backslashes. For example:
  `Arbortext-path\custom\doctypes`

- Square braces (**[ ]**) denote optional parameters which may be omitted. For example:

  insertBefore(newChild[, **refChild**])

- A vertical bar (**|**) separates parameters in a list from which one parameter must be chosen or used. For example:

  allowinvalidmarkup {**on** | **off**}

- `Arbortext-path` refers to the directory in which Arbortext Editor is installed.

Where to Get More Information

The files for Arbortext Editor and Arbortext Publishing Engine supporting documentation can be found in the Arbortext Help Center. You can access the Help Center from the Arbortext Editor **Help** menu.

If you're using Arbortext Publishing Engine, be sure to review *Installation Guide for Arbortext Publishing Engine* and *Configuration Guide for Arbortext Publishing Engine* for extensive information on Arbortext Publishing Engine installation, setup, and configuration.

Refer to the *Programmer's Reference* for more information on the Arbortext Object Model (AOM).

You can find training classes on the PTC web site at www.ptc.com

If you are looking for more general information on the Java programming language, you may want to consult the following resources:

- *Thinking in Java*, Third Edition, by Bruce Eckel. Published by Prentice Hall PTR. The full content of the book is available online at www.mindview.net.

- Sun has extensive Java information available at its web site java.sun.com. The tutorials are especially helpful to beginners.

2

Content Pipelines

Arbortext Editor and Arbortext Publishing Engine use a SAX filter pipeline, called a composer, to transform documents. A composer is a processor that can be configured to transform a document by passing it through one or more SAX filters in a filter pipeline. The result of this transformation may be one or more XML documents or external files. A content pipeline is defined in a CCF file, and exposed as an AOM composer object.

You can configure and run a composer object using an associative array of parameters. Users can create new composer objects or customize existing ones by subclassing existing filters in a pipeline or by adding or deleting filters in a CCF file to create new pipelines.

A composer is created by calling a `createComposer` method. When this method is called, the following process occurs:

- The composer's CCF file is parsed.
- For each filter in the pipeline, the composer:
    - Creates the adapter from the class specified in the `adapterClass` attribute.
    - Creates the filter.
    - Links the filter to its predecessor.
    - Maps the interface parameters to the filter's parameters.

Creating Filters and Filter Adapters

At a basic level, a filter in a content pipeline is a Java object that implements at least one of the seven SAX2 interfaces. You can use the publishing framework to write custom filters. These filters must conform to the SAX2 interfaces.

You can use filters in a pipeline to perform a variety of processes. The following list describes primitive transformations that filters can perform. You can also combine these filters to create more powerful transformations.

- **Suppression** or **profiling** — Filters out content that does not match certain criteria. The criteria may be imposed on elements, attribute values, or content. The output is a subset of the input.

- **Extraction** — Retains data that matches certain criteria. The criteria can be based on markup or content. The output is a subset of the input.

- **Insertion** or **embellishing** — Adds data to a document, and is driven by document content or external information. The output is a superset of the input.

- **Aggregation** — Assembles many documents into one document. Aggregation can be driven by a single document, called by a driver, or by external data.

- **Division** or **disassembly** — Splits one document into components. Division can be driven by document markup or by size constraints on the resulting chunks.

- **Reordering** — Moves data within a document from one location to another. You can reorder using markup to identify the data to move or content or attribute values to define target locations. Sorting is an example of reordering.

A filter needs an adapter object. An adapter is a Java object that implements the **FilterAdapter** interface. The **FilterAdapter** interface defines methods that allow the composer to query the SAX2 interfaces that the filter supports. It also defines methods that allow the composer to initialize and configure the filter.

An adapter is responsible for creating and destroying the filter. It knows which SAX2 interfaces the filter implements. Thus, when the composer queries the adapter for the filter's SAX2 interfaces, it returns the filter or null.

Default Java Classes

Arbortext Editor and Arbortext Publishing Engine provide two classes, `DefaultSAXFilter` and `DefaultAdapter`, for developing filters. These files help minimize the amount of code that needs to be written.

The `DefaultSAXFilter` class implements the **SAXFilter** interface, which is a convenience mix-in interface that extends the seven SAX2 interfaces and the **FilterControl** interface. The `DefaultSAXFilter` echoes the SAX events it receives to its outputs.

For example, if you want to write a simple filter that counts the number of characters in an XML document, you would subclass the `DefaultSAXFilter` class and override the `characters` method to keep a tally of the number of characters.

You can use the `DefaultAdapter` class with any filter that implements the **SAXFilter** interface. Therefore, any class that is a subclass of `DefaultSAXFilter` can be used in the pipeline with an instance of the `DefaultAdapter` class. Refer to the grep Filter Example on page 11 for an example of how to implement the `DefaultAdapter` class in a filter.

If an object exists that implements one or more of the SAX2 interfaces, you can write a filter adapter. This is helpful in situations in which the object is an off-the-shelf component that cannot be modified to implement Arbortext-specific interfaces.

Reusing Filters

To improve performance, the publishing framework caches pipelines. This allows you to reuse filters. Since you can't control a filter's lifetime, you should write the filter to obtain resources using the `initFilter` method and free them using the `destroyFilter` method. You should also use the `initFilter` method to reinitialize parameter values for parameters with default values. If you don't reinitialize parameter values, you could end up with a parameter value from a previous pipeline run, rather than the default value.

grep Filter Example

This example describes a filter that provides functionality similar to the `grep` command. The `grep` filter uses a regular expression to look for character matches in the input. Matches are reported to the error handler.

If you use this filter in a pipeline that runs in Arbortext Editor, matches are listed in the Event log window with links to the location within the document where the match occurred.

The filter has the following configuration:

- A regular expression.
- The number of characters reported (includes the matched string) upon a successful match.
- Spanning of non-character events such as start and end elements. For example, when you span non-character events, the regular expression "testcase" results in a match in the following situation:
  ```
  <element1>test</element1>
  ```

```
<element1>case</element1>
```

- Maximum number of characters that the filter tries to match against the regular expression.

This filter implements the `characters`, `startElement` and `endElement` methods in the **ContentHandler** interface. All SAX events are repeated since `grep` functionality does not alter content.

To implement this filter, you subclass the `DefaultSAXFilter` class, which overrides the `characters`, `startElement`, `endElement`, and `initFilter` methods. The `initFilter` method is called by the publishing framework to initialize filters.

```java
/*
 * grep.java Version 1.0
 *
 * Created 15-Aug-02
 *
 * 1000 Victor's Way, Ann Arbor, MI, 48108, U.S.A.
 */
package com.arbortext.epic.compose.examples;
import java.util.Map;
import org.xml.sax.Attributes;
import org.xml.sax.Locator;
import org.xml.sax.SAXException;
import org.xml.sax.SAXParseException;
import gnu.regexp.RE;
import gnu.regexp.REMatch;
import com.arbortext.epic.saxfilter.DefaultSAXFilter;
import com.arbortext.epic.saxfilter.utils.Logger;
/**
 * grep
 *
 * The grep filter performs an operation similar to the grep
 * command on characters that are present in the SAX event stream.
 * When a match is found, the filter creates a SAXParseException
 * and sends it to the ErrorHandler resource. The SAXParseException
 * contains the 'matched' string, along with some context as the
 * message. A locator object is also set in the constructor of the
 * exception.
 *
 * The filter can be configured with these parameters:
 * - regexp is the regular expression that is used to perform
 * the match.
 * - numContextChars is the number of characters that are reported
 * (includes the matched string) upon a successful match.
 * - crossNonCharEventBoundary indicates if the match crosses
 * non-character events such as start and end element
 * events.
 * - maxMatchSize is the maximum number of characters that the
 * filter tries to match against the regular expression.
 * NOTE: This filter does not affect the SAX events that
```

```java
 * pass through. Therefore, all methods that override the
 * base class methods call the respective super class methods.
 */
public class grep extends DefaultSAXFilter
{
 //Parameters
 public static final String PARAM_REGEXP = "regexp";
 public static final String PARAM_NUM_CONTEXT_CHARS =
 "numContextChars";
 public static final String
 PARAM_CROSS_NONCHAR_EVENT_BOUNDARIES =
 "crossNonCharEventBoundary";
 public static final String PARAM_MAX_MATCH_SIZE =
 "maxMatchSize";
 //constants used for default value for the parameters
 public static final int DEFAULT_NUM_CONTEXT_CHARS = 100;
 public static final int DEFAULT_MAX_MATCH_SIZE = 1000;
 //instance variables
 private boolean crossNonCharEventBoundary;
 private int numContextChars;
 private int maxMatchSize;
 private RE reObj; //regular expression object used to perform
 the actual match.
 private StringBuffer currString;
 private Logger logger;
 /**
 * Default constructor.
 */
 public grep()
 {
 resetParameters();
}
 /**
 * Helper function that resets the parameter values to the
 * default values.
 */
 private void resetParameters() {
 crossNonCharEventBoundary = false;
 numContextChars = DEFAULT_NUM_CONTEXT_CHARS;
 maxMatchSize = DEFAULT_NUM_CONTEXT_CHARS;
}
 /**
 * initFilter
 *
 * The initFilter method sets-up the parameters.
 *
 * @param parameters is a map containing the parameters.
 *
 * @throws exception if the regular expression string is
 * not present.
 */
```

```
public void initFilter(Map parameters) throws Exception
{
//The super class, DefaultSAXFilter, manages the output
//handler objects. Therefore, this class needs to call
//the super class' initFilter methods to set up the output
//handlers.
super.initFilter(parameters);
logger = Logger.getLogger(this.getClass());
//Uses the utility logger class
resetParameters();
Object paramValue;
//Get the regular expression string.
String regExpString = (String)parameters.get(PARAM_REGEXP);
if (regExpString == null || regExpString.equals(""))
throw new Exception("Grep filter requires a"
+ "valid regular expression.");
//How many context chars?
paramValue = parameters.get(PARAM_NUM_CONTEXT_CHARS);
if (paramValue != null) {
try {
numContextChars =
Integer.parseInt((String)paramValue);
}
catch (NumberFormatException nfe) {
logger.warn("Illegal value for "
+ PARAM_NUM_CONTEXT_CHARS);
numContextChars = DEFAULT_NUM_CONTEXT_CHARS;
}
}
//Can the match cross non char event boundaries?
if ("true".equalsIgnoreCase((String)parameters.get
(PARAM_CROSS_NONCHAR_EVENT_BOUNDARIES))) {
crossNonCharEventBoundary = true;
}
paramValue = parameters.get(PARAM_MAX_MATCH_SIZE);
if (paramValue != null) {
try {
maxMatchSize =
Integer.parseInt((String)paramValue);
}
catch (NumberFormatException nfe) {
logger.warn("Illegal value for "
+ PARAM_MAX_MATCH_SIZE);
maxMatchSize = DEFAULT_MAX_MATCH_SIZE;
}
}
reObj = new RE(regExpString);
currString = new StringBuffer(maxMatchSize);
} //initFilter()
/**
 * endElement resets the current string if the
```

```
 * match can't span non-character events.
 *
 */
public void endElement(String NameSpaceURI, String lName,
String qName) throws SAXException
{
super.endElement(NameSpaceURI, lName, qName);
//If the match can't cross non-character event
//boundaries, then the match string should be reset.
if (!crossNonCharEventBoundary)
currString.setLength(0);
} //endElement()
/**
 * startElement resets the current string if the match
 * can't span non-character events.
 */
public void startElement(String NameSpaceURI, String lName,
String qName, Attributes atts) throws SAXException
{
super.startElement(NameSpaceURI, lName, qName, atts);
if (!crossNonCharEventBoundary)
currString.setLength(0);
} // startElement()
/**
 * characters - This method tests the current string
 * against the regular expression. The method does
 * the following:
 * - Maintains the current string. The maximum size
 * of the string is controlled by the "maxMatchSize"
 * parameter.
 * - Tests the current string against the regular
 * expression.
 * - If a match is found, a SAXParseException is created
 * with the match string.
 *
 * The filter gets the document Locator using the
 * getDocumentLocator() method. The document locator
 * must have been set for the locator to be useful.
 * The publishing framework typically sets the document
 * locator.
 *
 * IMPLEMENTATION NOTE: If generateEpicDirectives is set to
 * true for the epicGenerator filter, then the locator is set
 * for all filters in the pipeline that follow the epicGenerator.
 */
public void characters(char[] ch, int start, int length)
throws SAXException
{
super.characters(new string cch, start, length);
currString.append(ch);
if (currString.length() > maxMatchSize) {
```

```
  currString.delete(0, currString.length() - maxMatchSize);
}
 //See if the current string matches the regular expression.
 //For now ignore multiple matches.
 REMatch match = reObj.getMatch(currString);
 if (match != null) {
 int matchStartIndex = match.getStartIndex();
 int matchEndIndex =
 Math.min(currString.length(),
 matchStartIndex + numContextChars);
 SAXParseException ex = new SAXParseException
 (currString.substring(matchStartIndex,
 matchEndIndex),
 getDocumentLocator());
 getErrorHandlerResource().warning(ex);
 //Reset the current string to avoid duplicate matches
 //in subsequent character method calls.
 currString.setLength(0);
}
} // characters(...)
}
```

The filter does not affect the SAX events that pass through the filter. This is
because the super class methods are called in all the methods that the filter
overrides.

```
super.initFilter(parameters);
...
super.characters(ch, start, length);
...
```

You can put this `grep` filter between an `epicGenerator` and an
`xslTransformer` with no data loss. The filter uses the
`com.arbortext.epic.saxfilter.utils.Logger` class to log
warnings. We recommend using this class to report messages that are not related
to SAX events.

```
logger = Logger.getLogger(this.getClass());
...
logger.warn("Illegal value for " + PARAM_MAX_MATCH_SIZE);
```

The filter resets its parameters in the `initFilter` method so that subsequent
calls (the filter is reused by the publishing framework) do not result in stale values
for non-required parameters (for example, *PARAM_MAX_MATCH_SIZE*).

Composer Configuration Files

A composer configuration file (CCF) is an XML document that defines a filter
pipeline. CCF files are named *type*`.ccf`, where *type* is the type of composer
(for example, `htmlfile.ccf`). A composer reads a CCF file, and builds and
configures the pipeline based on the information in the CCF file.

Content Pipeline Guide

Composer Document Type Definition

A CCF file is created based on the composer document type definition (DTD). The DTD specifies three main elements to define a pipeline:

- `Interface`
- `Resource`
- `Pipeline`

📝 **Note**

For more detailed information on the composer DTD, refer to the documentation included in the `composer.dtd` file distributed with Arbortext Editor and Arbortext Publishing Engine. This file is located at *Arbortext-path*`\doctypes\composer`.

Creating a Simple CCF File

The following CCF file defines a two-filter content pipeline that reads a Arbortext Editor XML document and then serializes it to a file on the file system.

```xml
<?xml version="1.0" encoding="utf-8"?>
<!DOCTYPE Composer PUBLIC "-//Arbortext//DTD Composer 1.0//EN"
 "composer.dtd">
<Composer>
<Interface>
 <Parameter name="document" idref="epicGenerator.docId"
 required="yes"></Parameter>
 <Parameter name="output_filename" idref="fileSerializer.outputFile"
 required="yes"></Parameter>
</Interface>
<Resource>
 <FilterDef id="epicGenerator" type="source"
 adapterClass="com.arbortext.epic.saxfilter.DefaultFilterAdapter"
 filterClass="com.arbortext.epic.saxfilter.EpicGenerator">
 <Parameter id="epicGenerator.docId" name="docId"
 required="yes"></Parameter>
 </FilterDef>
 <FilterDef id="fileSerializer" type="sink"
 adapterClass="com.arbortext.epic.saxfilter.DefaultFilterAdapter"
 filterClass="com.arbortext.epic.saxfilter.FileSerializer">
 <Parameter id="fileSerializer.outputFile"
name="outputFile"></Parameter>
 </FilterDef>
</Resource>
<Pipeline startFilters="epicParser">
 <Filter id="epicParser" filterDefRef="epicGenerator">
 <FilterParameter name="docId"><ComposerParameter
```

```
name ="document"/></FilterParameter>
</Filter>
<Filter id="serializer" filterDefRef="fileSerializer">
<FilterParameter name="outputFile">
<ComposerParameter name="output_filename"/>
</FilterParameter>
<FilterParameter name="method">
<Value>xml</Value>
</FilterParameter>
<Input filterRef="epicParser"/>
</Filter>
</Pipeline>
</Composer>
```

Interface element

The `Interface` element exposes pipeline parameters to the composer API. An `Interface` element can incorporate `Label`, `Documentation`, and `Parameter` elements.

```
<Interface>
 <Parameter name="document" idref="epicGenerator.docId"
 required="yes"></Parameter>
 <Parameter name="output_filename"
 idref="fileSerializer.outputFile"
 required="yes"></Parameter>
</Interface>
```

The composer in this example has two parameters, the document ID and the name of the serialized file. The `Parameter` names are exported as composer parameter names. These composer parameters get mapped to the filter parameters.

Resource element

The `Resource` element defines filters and their properties. This includes filter and adapter class names, and all parameters that the filter expects. Filter definitions contain parameter definitions for every parameter that the filter adapter accepts. After you define a filter using the `FilterDef` element, the `Pipeline` element can reference the filter.

```
<Resource>
 <FilterDef id="epicGenerator" type="source"
 adapterClass=
 "com.arbortext.epic.saxfilter.DefaultFilterAdapter"
 filterClass=
 "com.arbortext.epic.saxfilter.EpicGenerator">
 <Parameter id="epicGenerator.docId" name="docId"
 required="yes"></Parameter>
 </FilterDef>
```

An associated adapter constructs, initializes, and controls the filter. In the following example, the `adapterClass` attribute specifies `com.arbortext.epic.saxfilter.DefaultFilterAdapter` as the adapter's qualified class. The optional `filterClass` attribute specifies `com.arbortext.epic.saxfilter.FileSerializer` as the filter's package qualified class. The adapter's constructor uses this value as its argument. If you omit the `filterClass` attribute, the adapter constructor has no arguments.

```
<FilterDef id="fileSerializer" type="sink"
 adapterClass="com.arbortext.epic.saxfilter.DefaultFilterAdapter"
 filterClass="com.arbortext.epic.saxfilter.FileSerializer">
 <Parameter id="fileSerializer.outputFile"
 name="outputFile"></Parameter>
</FilterDef>
```

A filter can implement the adapter interface. In this situation, the adapter class is the same as the filter, so you can safely omit the filter class.

The `type` attribute defines the type of filter: source, transformer, or sink.

- A sink cannot have any `Output` elements.
- A source does not have to implement the SAX2 interfaces, but it must implement the `runFilter` method.
- A transformer must have `Input` and `Output` elements.

Pipes connect the filters in a pipeline. A filter that generates SAX events may direct these events to one or more pipes using the `Output` element. Each `Output` element within a `FilterDef` must have a distinct `pipeName` attribute. If you omit an `Output` element, the default pipe is used. If a filter doesn't declare an output, then it is a sink filter, and cannot come before any other filter in the pipeline.

Pipeline element

The `Pipeline` element defines the pipeline structure by specifying the filters and their inputs.

The `Pipeline` calls a filter's `runFilter` method, which is identified by the `startFilters` attribute.

In the following example, `epicParser` is the pipeline's source filter. The composer runs this filter when it receives the specified signal.

```
<Pipeline startFilters="epicParser">
```

📝 Note

If the `startFilters` attribute lists multiple source filters, their `runFilter` methods are called in the order listed.

The `Filter` element defines the filters that make up the pipeline. In the following example, the first filter in the pipeline (`epicParser`) is connected to the second filter (`serializer`) by the `Input` element.

```
<Filter id="epicParser" filterDefRef="epicGenerator">
<FilterParameter name="docId"><ComposerParameter
name="document"/></FilterParameter>
</Filter>
<Filter id="serializer" filterDefRef="fileSerializer">
<FilterParameter name="outputFile">
<ComposerParameter name="output_filename"/>
</FilterParameter>
<FilterParameter name="method">
<Value>xml</Value>
</FilterParameter>
<Input filterRef="epicParser"/>
</Filter>
```

Filters must receive certain parameters at run-time. These parameters can come from the composer parameters provided at run-time or they can be specified in the CCF file. For example, the `serializer` needs to know the name of the file to which it outputs (`output_filename`). The value for this parameter is provided at run-time from the composer parameter named `output_filename`. As a result, the value provided when the `run_composer` command is issued is passed to the filter.

The method parameter specifies the output format. The `fileSerializer` filter (*Arbortext-path*`\composer\fileSerializer.ent`) defines the valid values for the method parameter (`xml`, `html`, `text`, or `runTime`). In the following example, `xml` is the output method.

```
<FilterParameter name="method">
 <Value>xml</Value>
</FilterParameter>
```

Creating a Complex CCF File

The following CCF file defines a pipeline that appends a copyright statement to the input document and writes the output to a file or back to a Arbortext Editor document.

```
<?xml version="1.0" encoding="utf-8"?>
<!--ArborText, Inc., 1988-2002, v.4002-->
<!DOCTYPE Composer PUBLIC "-//Arbortext//DTD Composer 1.0//EN"
 "composer.dtd" [
<!ENTITY % stock PUBLIC "-//Arbortext//DTD Fragment -
 ATI Stock filter list//EN" "">
%stock;
]>
<Composer>
 <Label>Copyright.ccf</Label>
 <Documentation>This CCF file defines the pipeline that
```

```
appends a copyright statement to an HTML file that is
generated by the XSL transformation.
</Documentation>
<Interface>
<Label>API Parameters</Label>
<Documentation>The composer exposes the following parameters
to the API.
</Documentation>
<Parameter idref="epicGenerator.docId" name="document"
required="yes"></Parameter>
<Parameter idref="switch.outputPipe" name="destination"
required="yes"></Parameter>
<Parameter idref="fileSerializer.outputFile" name="filename"
required="no"></Parameter>
<Parameter idref="epicSerializer.docId" name="output_docid"
required="no"></Parameter>
<Parameter idref="xslTransformer.stylesheet"
name="stylesheet" required="yes"></Parameter>
<Parameter idref="fileSerializer.html.entSubFname"
name="html.entSubFname"></Parameter>
</Interface>
<Resource>
<Label>Filter resources.</Label>
<Documentation>The composer uses the following filter resources
</Documentation>
&epicGenerator;
&fileSerializer;
&epicSerializer;
&switch;
&xslTransformer;
&namespaceFixer;
<FilterDef id="htmlTailAppender"
adapterClass=
"com.arbortext.epic.saxfilter.DefaultFilterAdapter"
filterClass=
"com.arbortext.epic.compose.examples.HTMLTailAppender"
type="transformer">
<Label>HTML Tail Appender</Label>
<Documentation>Definition for filter that appends a given
HTML stream to the first one.
</Documentation>
</FilterDef>
</Resource>
<Pipeline startFilters="epic_generator copyright_statement">
<Label>Pipeline geometry.</Label>
<Documentation>The following filters implement this composer.
</Documentation>
<Filter id="epic_generator"
filterDefRef="epicGenerator">
<FilterParameter name="docId">
<ComposerParameter name="document"/>
```

```
</FilterParameter>
</Filter>
<Boilerplate id="copyright_statement">
<Content>
<html:h6 xmlns:html="http://www.w3.org/TR/REC-html40">
Copyright2002.Arbortext Inc.
</html:h6>
</Content>
</Boilerplate>
<Filter id="xsl_transformer"
filterDefRef="xslTransformer">
<FilterParameter name="stylesheet">
<ComposerParameter name="stylesheet"/>
</FilterParameter>
<Input filterRef="epic_generator"/>
</Filter>
<Filter id="copyright_ns_fixer"
filterDefRef="namespaceFixer">
<FilterParameter name="originNamespace">
<Value>http://www.w3.org/TR/REC-html40</Value>
</FilterParameter>
<FilterParameter name="prefix">
<Value> </Value>
</FilterParameter>
<Input filterRef="copyright_statement"/>
</Filter>
<Filter id="copyright_appender"
filterDefRef="copyrightAppender">
<Input filterRef="xsl_transformer"/>
<Input filterRef="copyright_ns_fixer"/>
</Filter>
<Filter id="output_selector" filterDefRef="switch">
<FilterParameter name="outputPipe">
<ComposerParameter name="destination"/>
</FilterParameter>
<FilterParameter name="pipeName1">
<Value>File</Value>
</FilterParameter>
<FilterParameter name="pipeName2">
<Value>Epic</Value>
</FilterParameter>
<Input filterRef="copyright_appender"/>
</Filter>
<Filter id="file_output" filterDefRef="fileSerializer">
<FilterParameter name="outputFile">
<ComposerParameter name="filename"/>
</FilterParameter>
<FilterParameter name="html.entSubFname">
<ComposerParameter name="html.entSubFname"/>
</FilterParameter>
<Input filterRef="xsl_transformer"
```

```
pipeName="outputProps"/>
<Input filterRef="output_selector" pipeName="pipe1"/>
</Filter>
<Filter id="epic_output" filterDefRef="epicSerializer">
<FilterParameter name="docId">
<ComposerParameter name="output_docid"/>
</FilterParameter>
<Input filterRef="output_selector" pipeName="pipe2"/>
</Filter>
</Pipeline>
</Composer>
```

The pipeline has two start filters, the `epicgenerator`, which reads the document from Arbortext Editor or the Arbortext Publishing Engine, and the `Boilerplate`, which sends the copyright statement as SAX events.

```
<Pipeline startFilters="epic_generator copyright_statement">
```

A `Boilerplate` is a source filter that can contain CDATA or namespace elements. The `Content` element specifies the SAX events to be generated by the `Boilerplate`.

If you include namespace elements in the `Boilerplate` filter, you must specify a namespace to conform to the composer DTD. In the following example, an HTML `h6` tag is used as markup, so the element must specify the HTML namespace.

📋 **Note**

    The namespace is subsequently stripped using the `namespace_fixer` filter.

```
<Boilerplate id="copyright_statement">
<Content>
<html:h6 xmlns:html="http://www.w3.org/TR/REC-html40">
Copyright2002.Arbortext Inc.
</html:h6>
</Content>
</Boilerplate>
```

📋 **Note**

    You could also store this `Boilerplate` in a file and include it as a file entity. This would localize the copyright message, allowing for translations.

Two SAX event streams are merged in the `copyright_appender` filter using two `Input` elements. The filter intercepts the end element events for `body` and `html`, and appends the copyright statement.

```
<Filter id="copyright_appender" filterDefRef="copyrightAppender">
 <Input filterRef="xsl_transformer"/>
 <Input filterRef="copyright_ns_fixer"/>
 </Filter>
```

A switch filter controls the output destination. The filter directs its input to the
pipe specified by the `outputPipe` parameter. There are two pipes in this
example, a file and Arbortext Editor.

```
 <Filter id="output_selector" filterDefRef="switch">
 <FilterParameter name="outputPipe">
 <ComposerParameter name="destination"/>
 </FilterParameter>
 <FilterParameter name="pipeName1">
 <Value>File</Value>
 </FilterParameter>
 <FilterParameter name="pipeName2">
 <Value>Epic</Value>
 </FilterParameter>
 <Input filterRef="copyright_appender"/>
 </Filter>
```

Common CCF Elements

The following table lists the three common elements in a CCF file.

Element	Description
Label	Identifies the components of a composer in a user interface.
Documentation	Documents the components in a pipeline. 📝 **Note** All elements in the content must have a namespace.
Value	Contains the actual or default value of a filter parameter or composer parameter. If used to define a parameter's enumeration values, the `default` attribute indicates the value to be treated as the default.

Distributed CCF Files

Arbortext Editor and Arbortext Publishing Engine distribute a set of CCF files for performing web, PDF, and HTML publishing using user-provided stylesheets. These CCF files use entities to modularize filter definitions so they can be reused.

For example, most CCF files use `EpicGenerator` as the start filter. This filter takes a Arbortext Editor document and generates SAX events that become inputs in the pipeline. The `EpicGenerator FilterDef` element is an entity, `epicGenerator.ent`, and is included in the CCF files that require it.

The entity definitions are stored in the `stock.ent` file, which is located at *Arbortext-path*\composer. The entity files are found using normal catalog resolution.

The following example illustrates how the `epicGenerator.ent` and `fileSerializer.ent` entities are defined in the `stock.ent` entity:

```
<ENTITY epicGenerator PUBLIC
 "-//Arbortext//ENTITIES Epic Generator//EN" "epicGenerator.ent">
<ENTITY fileSerializer PUBLIC
 mposer"-//Arbortext//ENTITIES File Serializer//EN" "fileSerializer.ent">
```

Publishing Types and Distributed CCF Files

The following table lists the Arbortext Editor publishing types and their corresponding CCF files.

Publishing types and CCF files

Publishing Type	CCF file
View profiled document	`profile.ccf`
Print Preview	`xslfo.ccf`
Web	`web.ccf`
HTML Help	`htmlhelp.ccf`
HTML File	`htmlfile.ccf`
PDF File	`pdf.ccf`
Using XSL	`xsl.ccf`

CCF Files and Profiling

Arbortext Editor publishing only handles three profiling attributes by default. If you want to use more than three profiling attributes, you can modify the corresponding CCF files.

📝 Note

You must configure profiling attributes in the `.dcf` file.

If you want to pass an XML document and a profile through a pipeline and return the profiled version of the same document, you can create a composer for XSL with the desired profile as the input profile and use `identity.xsl` as the stylesheet. The stylesheet is located in `lib\xsl\debug`.

The `compose_using_xsl` batch function and other ACL publishing functions are described in Interactive and Batch Publishing Functions on page 70 .

To Add a Profile Filter to a CCF File:

1. Determine the publishing types to which you want to add profiling attributes, and add a profile filter and its parameters to the associated CCF files on page 25. For example, if you want to add a fourth filter to the pipeline in the `profile.ccf`, include the following code after the `profiler3` filter definition.

```
<Filter id="profiler4" filterDefRef="profiler">
<FilterParameter name="targetAttribute">
<ComposerParameter name="prof4.targetAttribute"/>
</FilterParameter>
<FilterParameter name="targetValue">
<ComposerParameter name="prof4.targetValue"/>
</FilterParameter>
<FilterParameter name="alias">
<ComposerParameter name="prof4.alias"/>
</FilterParameter>
<FilterParameter name="client">
<ComposerParameter name="prof4.client"/>
</FilterParameter>
<FilterParameter name="server">
<ComposerParameter name="prof4.server"/>
</FilterParameter>
<FilterParameter name="separator">
<ComposerParameter name="prof4.separator"/>
</FilterParameter>
<Input filterRef="profiler3"/>
</Filter>
<Filter id="epic_serializer" filterDefRef="epicSerializer">
...
<Input filterRef="profiler4"/>
</Filter>
</Pipeline>
```

The `profiler4` filter is now the input for the last filter in the pipeline (`epic_serializer`).

2. Add the filter parameters to the `Interface` section of the CCF file to which you want to add the profiling attributes. All CCF files include similar profile filter settings.For example, if you want to use the fourth profiling attribute with the **Edit ▸ View Profiled Document** menu option, you would add the following code to the `profile.ccf` file after the `&profilerParameters;` entity.

```
<Interface>
...
 <Parameter name="prof4.targetAttribute"
 idref="profiler.targetAttribute"/>
 <Parameter name="prof4.targetValue"
 idref="profiler.targetValue"/>
 <Parameter name="prof4.alias"
 idref="profiler.alias"/>
 <Parameter name="prof4.client"
 idref="profiler.client"/>
 <Parameter name="prof4.server"
 idref="profiler.server"/>
 <Parameter name="prof4.separator"
 idref="profiler.separator"/>
 </Interface>
```

3

Using AOM with Pipelines and Filters

The AOM provides object-oriented programming access to Arbortext Editor and Arbortext Publishing Engine. The AOM supports the W3C DOM (Document Object Model) interfaces with extensions, and provides many additional interfaces for Arbortext-specific features that are not part of the DOM.

The Arbortext extensions to the DOM use a naming convention where A (for Arbortext) is prefixed to the DOM interface name; for example, the Arbortext extension for the DOM `Node` interface is `ANode`.

The AOM supports bindings to Java, COM (Component Object Model), and C++. The AOM also provides scripting access to its interfaces using JavaScript, JScript, VBScript, and ACL (Arbortext Command Language).

Running a Composer using AOM

You can use the AOM to run a content pipeline. This allows you to write composer calls in any AOM-supported language.

The `Application.createComposer on page 63` method creates a composer object. You need to pass this method the path for the CCF file.

In the following example, the CCF file (*Arbortext-path*\composer\ `htmlfile.ccf`) creates a composer used by **File ▸ Publish ▸ HTML File** for publishing an HTML file.

```
Composer comp = Application.createComposer(APT_PATH +
 "/composer/htmlfile.ccf");
```

AOM Publishing using Java

The following example shows how to create a pipeline using Java. `HTMLFilecomposer` has two public methods that take XML input (as an in-memory Arbortext Editor document or a file) and transforms it to an HTML file using the specified XSL stylesheet.

```
package com.arbortext.epic.compose.examples;
/*
 * HTMLFileComposer is an example of calling the content
 * pipeline using the AOM composer. In this example, an XML
 * document is published to an HTML file. The source document
 * can exist in one of two places, in Arbortext or a file.
 * The composer uses the htmlfile pipeline defined in
 * htmlfile.ccf in the composer directory of the Arbortext
 * installation tree.
 */
import com.arbortext.epic.*;
import org.w3c.dom.*;
import java.io.File;
public class HTMLFileComposer {
 /**
 * Used internally to access the composer configuration file.
 */
 private static final String HTMLFILE_CCF =
 File.separator + "composer" + File.separator +
 "htmlfile.ccf";
 /**
 * Used internally to access the entity substitution file.
 */
 private static final String HTMLENTSUBFILE =
 File.separator + "composer" + File.separator +
 "htmlEntSub.xml";
 /**
 * Produces HTML from an in-memory XML file and an
 * XSL stylesheet.
 *@param docId is the ID of document to process.
```

```
*@param stylesheet is a fully-pathed XSL stylesheet.
*@param outputFile is a fully-pathed HTML output filename.
*/
public static void composeToHtmlFromDoc
(int docId, String stylesheet,String outputFile) {
ComposerLog log = new ComposerLog();
try {
String installPath = Acl.eval("main::aptpath");
//Create the Composer object for the HTML publishing process.
Composer composer = Application.createComposer(installPath
+ HTMLFILE_CCF);
PropertyMap params = Application.createPropertyMap();

//Set up the parameters.
params.putString("stylesheet", stylesheet);
params.putString("document", Integer.toString(docId));
//The entity substitution file for HTML
params.putString("html.entSubFname", installPath
+ HTMLENTSUBFILE);
params.putString("outputFile", outputFile);
//The following code creates the directory in which
//graphics would be placed and the associated href in
//the HTML document.
params.putString("graphicsHref",
(new File(outputFile)).getName() + ".graphics/");
params.putString("graphicsPath", outputFile + ".graphics/");
//Let the composer know we are using an XSL stylesheet.
params.putString("stylesheetType", "xsl");
//Start the composer log with level at info.
log.startJob("HTMLFileComposer", ComposerLog.SEVERITY_INFO);
//runPipeline returns a boolean indicating success or failure.
if (composer.runPipeline(params)) {
log.logMessage("Success", ComposerLog.SEVERITY_INFO);
}
else {
//Error information will have been placed into the
//Event Log.
log.logMessage("Failure", ComposerLog.SEVERITY_INFO);
}
}
catch (AclException ex) {
//Unexpected.
System.err.println("ACLException in
composeToHtmlFromDoc: " + ex);
ex.printStackTrace(System.err);
}
catch (AOMException aomex) {
//Unexpected.
System.err.println("AOMException in
composeToHtmlFromDoc: " + aomex);
aomex.printStackTrace(System.err);
```

```
}
 finally {
 log.endJob();
}
}
 /**
 * Produces HTML from an on-disk XML file and an XSL stylesheet.
 * @param inputFile is a fully-pathed XML filename.
 * @param stylesheet is a fully-pathed XSL stylesheet.
 * @param outputFile is a fully-pathed HTML output filename.
 */
 public static void composeToHtmlFromFile
 (String inputFile, String stylesheet,
 String outputFile) {
 ADocument doc = null;
 try {
 doc = (ADocument) Application.openDocument
 (inputFile, Application.OPEN_RDONLY |
 Application.OPEN_NOSTYLE |
 Application.OPEN_NOCC |
 Application.OPEN_NOMSGS |
 Application.OPEN_NODTPROMPT);
 composeToHtmlFromDoc(doc.getAclId(), stylesheet,
 outputFile);
}
 catch (AOMException aomex) {
 System.err.println
 ("AOMException in composeToHtmlFromFile: " +
 aomex);
 aomex.printStackTrace(System.err);
}
 finally {
 if (doc != null) {
 doc.close();
}
}
}
}
```

The first step in this example is creating the composer object from the CCF file
(*Arbortext-path*\composer\htmlfile.ccf):

```
Composer composer = Application.createComposer(installPath
 + HTMLFILE_CCF);
```

The composer expects to receive parameters as a **PropertyMap** AOM object. Refer
to the createPropertyMap method in the *Programmer's Reference* for more
information.

```
PropertyMap params = Application.createPropertyMap();
params.putString("stylesheet", stylesheet);
...
```

The Event Log must be started before a publishing process is run. A **ComposerLog** object wraps the ACL calls. The source can be found in the `examples.jar` file distributed with the *Content Pipeline Guide*. (included in the `examples.zip` file).

```
log.startJob("HTMLFileComposer", ComposerLog.SEVERITY_INFO);
```

The pipeline is then run using the parameters provided by the **PropertyMap** AOM object. If the pipeline runs successfully, a true value is returned. If there are errors, a false value is returned.

```
if (composer.runPipeline(params)) ...
```

The Event Log must have an end signal to conclude logging. Place the end signal in the `finally` clause to ensure that this signal is called even if an exception occurs.

```
log.endJob();
```

4

Using ACL with Pipelines and Filters

ACL functions are often used to post-process composer output. You may also want to use ACL to call third party tools, such as HTML Help Workshop.

Arbortext Editor and Arbortext Publishing Engine provide a set of ACL functions that query CCF files, and configure and start pipeline processes. Refer to ACL Reference on page 67 for detailed information on the ACL publishing functions.

ACL Pipeline Example

You can use ACL functions to start publishing when the user interface is unavailable (such as in the Arbortext Publishing Engine) or when running a custom pipeline.

The following example illustrates two ACL functions for HTML publishing, one from an in-memory XML document and one from an on-disk XML file. The functions use the pipeline defined in *Arbortext-path*\composer\ htmlfile.ccf.

```
#HTMLFileComposer.acl
require _composerlog;
#Produces HTML from an in-memory XML file and an
#XSL stylesheet.
#docId is the ID of the document to process.
#stylesheet is a fully-pathed XSL stylesheet.
#outputFile is a fully-pathed HTML output filename.
#Returns 1 if publishing was successful, 0 otherwise.
function composeToHtmlFromDoc(docId, stylesheet,
 outputFile) {
 local installPath = main::aptpath;
 local ccfParameters[];
 local composeStatus;

 #Set the catalog path.
 compose::composer_set_catalog_path(option("catalogpath"));
 #Get the composer for htmlfile.ccf (present in
 #the composerpath).
 local composer = get_composer(ccfParameters, "htmlfile", docId);

 #Set up the parameters.
 ccfParameters["stylesheet"] = stylesheet; #stylesheet
 #Use only XSL stylesheets in this example.
 ccfParameters["stylesheetType"] = "xsl";
 ccfParameters["outputFile"] = outputFile;

 #the entity substitution file for HTML
 ccfParameters["html.entSubFname"] = installPath .
 "\\composer\\htmlEntSub.xml";

 #The following code creates the directory in which graphics
 #are placed and the associated href in the HTML document.
 ccfParameters["graphicsHref"] = basename(outputFile) .
 ".graphics/";
 ccfParameters["graphicsPath"] = outputFile .
 ".graphics\\";

 #Start logging.
 _composerlog::start_job("HTMLFileComposer");
 composeStatus = run_composer(composer, ccfParameters);
```

Content Pipeline Guide

```
 if (composeStatus) {
 compose::info("Success.");
}
 else {
 compose::info("Failure.");
}

 _composerlog::end_job();
 return composeStatus;
}
#Produces HTML from an on-disk XML file and an
#XSL stylesheet.
#inputFile is a fully-pathed XML filename.
#stylesheet is a fully-pathed XSL stylesheet.
#outputFile is a fully-pathed HTML output filename.
#Returns 1 if publishing was successful, 0 otherwise.
function composeToHtmlFromFile(inputFile, stylesheet,
 outputFile) {
 #Load the file into a Arbortext Editor document. Then,
 #call composeToHtmlFromDoc.
 local doc = doc_open(inputFile, 0x01 | 0x10 | 0x20 |
 0x200 | 0x400);
 local composerStatus;
 if (!doc_valid(doc)) {
 message "Unable to open file $inputFile";
 return 0;
}
 composerStatus = composeToHtmlFromDoc(doc, stylesheet,
 outputFile);
 doc_close(doc);
 return composerStatus;
}
```

Set up the catalog path, so all entity files not in the standard install location can be resolved.

```
compose::composer_set_catalog_path(option("catalogpath"));
```

For example, a new catalog file is created when you move the `xsltransformer.ent` entity file to the `custom\composer` directory. You must call `compose_set_catalog_path` so the composer will see this catalog file.

The location of the CCF file is not specified. Instead, the `get_composer` method calls with the name of the composer:

```
local composer = get_composer(ccfParameters, "htmlfile", docId);
```

The `get_composer` method looks for the `htmlfile.ccf` file in the path specified by the `set composerpath` command. The default `composerpath` includes the *Arbortext-path*\composer directory and the *Arbortext-path*\custom\composer directory. (Refer to for more information.)

After the composer object is obtained, the pipeline can be run using the `run_composer` function. This function takes the composer object and an array of parameters. This array can be the array from the `get_composer` function, with the parameters filled in.

```
composeStatus = run_composer(composer, ccfParameters);
```

To ensure proper error logging, you must inform the Event Log of the start and end of the publishing process using the `_composerlog::start_job` and `_composerlog::end_job` methods.

```
 #Start logging.
 _composerlog::start_job("HTMLFileComposer");
 composeStatus = run_composer(composer, ccfParameters);

 if (composeStatus) {
 compose::info("Success.");
}
 else {
 compose::info("Failure.");
}

 _composerlog::end_job();
 return composeStatus;
}
```

The following example illustrates how to call the ACL functions:

```
#publish to HTML from Arbortext Editor document
composeToHtmlFromDoc(current_doc(), \
"c:\\arbortext\\editor\\doctypes\\axdocbook\\axdocbook-html.xsl",
\
"c:\\document.htm")
#publish to HTML from file
composeToHtmlFromFile("c:\\temp\\file.xml", \
"c:\\arbortext\\editor\\doctypes\\axdocbook\\axdocbook-html.xsl",\
"c:\\file.htm");
```

Running Standard Publishing Processes with ACL

The Arbortext Editor **Publish** menu relies on ACL functions to perform publishing processes. These processes can be run interactively or in batch mode. The interactive functions open dialog boxes in which you can enter publishing information and then run the publishing process.

These interactive functions are in the form `compose_type`, where *type* is one of the following publishing types:

htmlfile	wap
htmlhelp	web

pda	xsl
pdf	xslfo

The following example illustrates how to use ACL to run the web publishing process on the current document:

```
compose_web(current_doc());
```

The batch mode equivalents to the interactive functions are in the form `compose_for_type`, where *type* is one of the publishing types listed above. These commands accept the document ID and an array of parameters.

```
local parameters[];
#gather parameters
...
compose_for_pdf(current_doc(), parameters);
```

📋 **Note**

The **Publish ▸ Using XSL** process uses the `compose_using_xsl` batch function.

Refer to Interactive and Batch Publishing Functions on page 70 for more information on ACL publishing functions.

Using Core Functions

Arbortext Editor and the Arbortext Publishing Engine provide a set of core publishing functions. We recommend that you only use these functions for debugging purposes as they may change in future releases.

Refer to Core Functions on page 79 for more information on these functions.

5

Error Handling

The content pipeline uses Apache log4j Ver 1.13 (`http://jakarta.apache.org/log4j/docs/`) as its error logging mechanism. The `Log4jErrorHandler` is the default message reporting class. Messages generated by individual filters can be reported by calling log4j methods or using calls to the error handler embedded in every filter.

The logj4 logging mechanism specifies the amount of information logged (verbosity), destination (file, console, or screen), and format (for example, plain text or xml). The only parameters to specify in the filter for error logging are message content and severity levels.

You can control attributes of the Event Log by calling the `composer_log` ACL function. Refer to the Arbortext Editor online help for detailed information on the `composer_log` function.

Using ErrorHandler Interface Methods

A default error handler is embedded when a filter is constructed. You can retrieve the error handler using a `getErrorHandlerResource` call in the **FilterControl** interface.

In the `grep` filter example, the `getErrorHandlerResource` call returns the embedded error handler.

```
SAXParseException ex =
new SAXParseException
(currString.substring(matchStartIndex,
matchEndIndex),
getDocumentLocator());
getErrorHandlerResource().warning(ex);
//Reset the current string to avoid duplicate
//matches in subsequent character method calls.
currString.setLength(0);
}
} // characters(...)
```

The `warning` method takes **org.xml.sax.SAXParseException** as its argument. `SAXParseException` contains location information for the current document. The location information lets users link from the event log to the string in the document.

Refer to ErrorHandler Interface on page 60 for more information on the ErrorHandler interface, and grep Filter Example on page 11 for the entire `grep` filter example.

Using log4j Methods

If you do not need to know the error location, you can call `log4j` class methods to log errors.

The following example illustrates a filter that uses log4j to report messages. This filter counts the number of elements in the document instance and reports the number to the log. The log4j `Category` class is imported and an instance of `Category` logger is created using the `getInstance` static method.

```
package com.arbortext.epic.saxfilter.custom;
import java.util.Map;
import org.xml.sax.SAXException;
import org.xml.sax.Attributes;
// log4j
import org.apache.log4j.Category;
import com.arbortext.epic.saxfilter.DefaultSAXFilter;
/**
 * CountTags.java
 *
 * <p> This filter counts how many elements are in the
 * current document.
```

```
 *
 * <pre>
 * <FilterDef id="tagCounter"
 * adapterClass=
"com.arbortext.epic.saxfilter.DefaultFilterAdapter"
 * filterClass=
"com.arbortext.epic.saxfilter.custom.CountTags"
 * type="transformer">
 * <Label>Count Tags Filter</Label>
 * <Documentation>
 * This filter counts how many elements are there in
 * the current document.
 * </Documentation>
 * </FilterDef>
 * </pre>
 *
 *
 * Created: Wed Sep 11 09:53:38 2002
 *
 */
public class CountTags extends DefaultSAXFilter {
 protected int count = 0;
 private Category logger = Category.getInstance
 (CountTags.class);
 public void initFilter (Map params) throws
 Exception {
 super.initFilter(params);
 //Reset counter for each run.
 count = 0;
}
 public void startElement (String NameSpaceURI,
 String lName, String qName,
 Attributes atts) throws SAXException {
 count++; // increment the counter
 super.startElement(NameSpaceURI, lName, qName, atts);
}
 public void endDocument () throws SAXException {
 //Report the total number of elements using log4j
 //"info" method.
 logger.info("There are total of " + count +
 " elements in this document.");
 super.endDocument();
}

}//CountTags
```

The `debug`, `info`, `warn`, `error` and `fatal` methods are the most frequently used logging methods in the `Category` class.

> **📝 Note**
>
> Refer to the log4j web site at `http://jakarta.apache.org/log4j/docs/` for more information on the `Category` class.

Customizing Error Handling

If the default logging mechanism does not satisfy your needs, you can customize an error handler. To do this, you need to:

* Create a filter that implements the **ErrorHandler** interface methods.

* Specify the new error handler in the appropriate CCF file.

Creating an Error Handling Filter

The following example shows a simple implementation of an error handler. This error handler reports error messages to the Java console. The `SimpleErrorHandler` extends the `SAXFilterImpl` and overrides the methods inherited from the **ErrorHandler** interface.

```
package com.arbortext.epic.saxfilter;
import java.util.Map;
import java.util.HashMap;
import org.xml.sax.SAXException;
import org.xml.sax.SAXParseException;
// log4j
import org.apache.log4j.Category;
/**
 *
 * <pre>
 * <FilterDef id="simpleErrorHandler"
 * adapterClass=
"com.arbortext.epic.saxfilter.DefaultFilterAdapter"
 * filterClass=
"com.arbortext.epic.saxfilter.SimpleErrorHandler"
 * type="sink">
 * <Label>Simple Error Handler</Label>
 * <Documentation> This filter reports error message
to the console.
 * </Documentation>
 * </FilterDef>
 * </pre>
 */
public class SimpleErrorHandler extends SAXFilterImpl {
 /**
 * Creates a new SimpleErrorHandler instance.
 */
```

```
 public SimpleErrorHandler () {
}
 public void warning (SAXParseException e) throws
 SAXException {
 System.out.println("[WARNING]: " + e.getMessage());
}
 public void error (SAXParseException e) throws
 SAXException {
 System.out.println("[ERROR]: " + e.getMessage());
}
 public void fatalError (SAXParseException e) throws
 SAXException {
 System.out.println("[FATAL]: " + e.getMessage());
 throw e;
}
}
```

Adding an Error Handler to a CCF File

You must add the error handler filter definition to the CCF's `Resource` element, and specify the error handler as an attribute of the `Pipeline` element.

The following example illustrates how to replace the default error handler with your custom error handler. The `Resource` element defines the `simpleErrorHandler` filter.

```
<Resource>
 ....
 <FilterDef id="simpleErrorHandler"
 adapterClass=
 "com.arbortext.epic.saxfilter.DefaultFilterAdapter"
 filterClass=
 "com.arbortext.epic.saxfilter.SimpleErrorHandler"
 type="sink">
 <Label>Simple Error Handler</Label>
 <Documentation> This filter reports error message to
 the console.
 </Documentation>
 </FilterDef>
....
</Resource>
```

The `Pipeline` element specifies the `errorHandler` attribute, which refers to the `simple_error_handler`. The `simple_error_handler` is then associated with the `simpleErrorHandler` defined by the `Resource` element.

```
<Pipeline startFilters="epic_generator"
 errorHandler="simple_error_handler">
...
<Filter id="simple_error_handler"
 filterDefRef="simpleErrorHandler"/>
```

```
...
</Pipeline>
```

This error handler serves as the embedded error handler and reports messages for publishing processes defined by the CCF file. Messages reported by calling log4j methods directly are not affected by the new error handler.

6

Customizing Publishing

Adding Web Publishing to a Document Type

Adding an existing composer to a document type is easy, provided that a suitable stylesheet exists for the document type.

To Add Web Publishing to a Document Type:

1. Add a web stylesheet ID processing instruction to the stylesheet you're going to use for web publishing. For example,

   ```
   <?APT StylesheetID Title="DocBook Web"
    PublishingTypes="web,xsl"?>
   ```

   This allows the stylesheet to be used for web publishing.

   > **Note**
   >
   > Refer to the Stylesheet identification processing instruction topic in the Arbortext Editor online help for more information.

2. Edit the DCF file for the document type as follows:

   - Set the `allowComposeStylesheetList` attribute in the `Options` element to `yes`.

     ```
     <Options allowApplicationToolbar="no"
      allowComposeStylesheetList="yes"
      allowFosiMod="yes" allowTouchup="yes"
      protected="no"/>
     ```

     > **Note**
     >
     > Refer to the Document type configuration files topic in the Arbortext Editor online help for more information.

   - Specify `web` as the publishing type for the `Compose` element in the DCF file.

     ```
     <Composition><Compose type="web"></Compose></Composition>
     ```

     This adds the **Publish ▸ For Web** menu option to the **File** menu.

   - Add a frameset definition to the DCF file:
     ```
     <Framesets>
     <Frameset description="default" location="default"/>
     </Framesets>
     ```

Switching from Saxon to the Xalan Processing Engine

Arbortext Editor and Arbortext Publishing Engine now use the Saxon XSLT processing engine to publish XML and SGML documents using XSL stylesheets. Previously, Arbortext Editor and the Arbortext Publishing Engine used the Xalan XSLT processing engine. If you use document types distributed with Arbortext Editor or the Arbortext Publishing Engine, there should be no visible changes to your publishing output as the result of this change.

If you use stylesheets that incorporate Xalan extensions, you can switch your publishing engine from Saxon to Xalan.

📑 **Note**

The information on switching to the Xalan processing engine is provided as a convenience. If you experience problems using the Xalan processing engine, we suggest you switch back to using the Saxon processing engine.

To Switch from Saxon to Xalan for XSL Publishing (Other than HTML Help):

1. Locate the `xslTransformer.ent` file in the *Arbortext-path*\ `composer` directory. Copy this file into the *Arbortext-path*\custom\ `composer` directory.

2. Edit the copied file with a text editor and locate the `<Parameter name= "transformerClass"` section. You will see a nested `<Value>` element similar to the following example:
   `<Value>com.icl.saxon.TransformerFactoryImpl</Value>`

3. Remove the contents of the `<Value>` element, and specify the value of the Xalan processor:
   `<Value>org.apache.xalan.processor.TransformerFactoryImpl</Value>`

4. In the *Arbortext-path*\custom\composer directory, create a text file called `catalog` (if one is not already present).

5. Open the *Arbortext-path*\composer\catalog file and copy the following line into the new `catalog` file (ignore the line break):
   ```
   PUBLIC "-//Arbortext//ENTITIES XSL Transformer//EN"
    "xslTransformer.ent"
   ```

6. Restart Arbortext Editor for the change to take effect.

To Switch from Saxon to Xalan for HTML Help Publishing:

1. Locate the `htmlHelpAdapter.ent` file in the *Arbortext-path*\
    `composer` directory. Copy this file into the *Arbortext-path*\`custom`\
    `composer` directory.

2. Edit the copied file with a text editor and locate the `<Parameter name=`
    `"transformerClass"` section. You will see a nested `<Value>` element
    similar to the following example:
    `<Value>com.icl.saxon.TransformerFactoryImpl</Value>`

3. Remove the contents of the `<Value>` element, and specify the value of the
    Xalan processor:
    `<Value>org.apache.xalan.processor.TransformerFactoryImpl</Value>`

4. In the *Arbortext-path*\`custom`\`composer` directory, create a text file
    called `catalog` (if one is not already present).

5. Open the *Arbortext-path*\`composer`\`catalog` file and copy the
    following line into the new catalog file:
    `PUBLIC "-//Arbortext//ENTITIES HtmlHelp Adapter//EN"`
    `"htmlHelpAdapter.ent"`

6. Restart Arbortext Editor for the change to take effect.

Configuring Character Entity Substitution Files for HTML Publishing

Entity substitution solves various issues with web browsers and HTML. The
HTML 4.0 standard provides definitions for some characters, such as the em dash
(Unicode character 8212), and the publishing process will use this definition.
However, some browsers may not fully support the HTML 4.0 standard, resulting
in dropped or malformed characters. You can usually solve this by substituting a
numeric character reference for the problem character.

The HTML Help navigation pane is a proprietary format that is not based on
HTML. For example, if the em dash character (Unicode character 8212) was
included in a title that displayed in the HTML Help navigation pane, it would not
display properly because the navigation pane cannot display Unicode characters.
In this example, you could substitute a hyphen character to achieve acceptable
results.

Entity Substitution Files

The following entity substitution files are located in the *Arbortext-path*\
`composer` directory. Each file controls a different aspect of entity substitution,
and uses the same XML format for specifying substitutions.

- `htmlEntSub.xml` — Controls entity substitution for HTML (not XHTML) outputs.

- `htmlHelpNavEntSub.xml` — Controls entity substitution for the HTML Help navigation pane.
- `xmlEntSub.xml` — Controls entity substitution for XML outputs (includes XHTML).
- `textEntSub.xml` — Controls entity substitution for XSL stylesheets that include a text output method.

For entity substitution when publishing using **File ▸ Publish ▸ For Web**, use the XSLT 2.0 character map feature describe at www.w3.org/TR/xslt20/#character-maps.

Entity Substitution File Format

This section defines the format for the entity substitution files. These files are free-form XML with the following simple element structure.

`<character-mapping>` — Top-level element; required. No attributes.

`<unicode/>` — The only allowed child element. It has the following attributes:

- `char` — Required. Represents the Unicode character to be substituted.
- `entity` — Optional. Represents the entity to substitute. The entity may be a regular named entity like "mdash" or a numeric character reference like "#8212". The value of this attribute would be the markup with the leading ampersand and trailing semicolon removed.
- `string` — Optional. Represents a string value to substitute, used for HTML Help Navigation and text because these outputs are not represented by markup. However, you can use this attribute with the other output formats. If both `entity` and `string` are present, `entity` is used.

Substitutions Involving Entities

Following are examples of entity substitutions:
```
<character-mapping>
<unicode char="34" entity="quot"/>
<unicode char="38" entity="amp"/>
<unicode char="39" entity="#8217"/>
<unicode char="60" entity="lt"/>
<unicode char="62" entity="gt"/>
<unicode char="160" entity="nbsp"/>
<unicode char="161" entity="#161"/>
<unicode char="8212" entity="#8212"/>
</character-mapping>
```

Substitutions Involving Strings

Following are examples of string substitutions:
```
<character-mapping>
<unicode char="8658" string="=&gt;"/>
<unicode char="8656" string="&lt;="/>
<unicode char="8594" string="-&gt;"/>
<unicode char="8592" string="&lt;-"/>
<unicode char="402" string="f"/>
<unicode char="8230" string="..."/>
<unicode char="352" string="S"/>
<unicode char="8249" string="&lt;"/>
<unicode char="338" string="OE"/>
<unicode char="8216" string="'"/>
<unicode char="8217" string="'"/>
<unicode char="8220" string="&amp;quot;"/>
<unicode char="8221" string="&amp;quot;"/>
<unicode char="8226" string="."/>
<unicode char="8211" string="-"/>
<unicode char="8212" string="--"/>
<unicode char="732" string="~"/>
<unicode char="8482" string="(TM)"/>
<unicode char="353" string="s"/>
<unicode char="8250" string="&gt;"/>
<unicode char="339" string="oe"/>
<unicode char="376" string="Y"/>
<unicode char="8194" string=" "/>
<unicode char="8195" string=" "/>
</character-mapping>
```

The `string` attribute value is read by an XML parser, so entity references beginning with an ampersand (&) are resolved to form a character. For example "=>" is parsed and substituted as "=>", and "&quot;" is parsed and substituted as """.

Customizing Entity Substitution Files

Following is the procedure for customizing entity substitution files.

To Customize Entity Substitution Files:

1. Determine which file in the *Arbortext-path*\composer directory you need to modify.
2. Copy the appropriate file to *Arbortext-path*\custom\composer.
3. Edit the files as you would any XML file.
4. Save and close the file.
5. Exit and restart Arbortext Editor.

Controlling Graphics Conversion for HTML-Based Publishing

When publishing documents to HTML (**Publish ▸ For Web** and **Publish ▸ For HTML File**), Arbortext Editor and Arbortext Publishing Engine copy graphics with GIF, JPEG, and PNG file formats. Other graphic file formats are converted to GIF during the publishing process.

If you do not want Arbortext Editor and Arbortext Publishing Engine to convert your graphics, you can use the *APTCOPYGRAPHICEXTS* environment variable to override the list of graphic file formats that Arbortext Editor and Arbortext Publishing Engine automatically convert.

The format for this is a comma-delimited list of extensions (case-insensitive) that Arbortext Editor or the Arbortext Publishing Engine will copy instead of converting. In the following DOS example,
`set APTCOPYGRAPHICEXTS=gif,jpg,jpeg,png,tif`

the GIF, JPG, PNG, and TIF file formats will be copied instead of converted.

Configuring Publishing Processes to Output Change Tracking Markup

You can configure Arbortext Editor publishing processes to output change tracking markup. This allows you to specify formatting in an XSL stylesheet to output additions and deletions in a document, similar to the change tracking markup displayed in the Edit pane.

changetracking Variable

Change tracking markup is output based on the setting of the *changetracking* parameter in the PTC Arbortext Generator pipeline filter (`Arbortext-path/composer/epicGenerator.ent`). The *changetracking* parameter accepts the following values:

- *nothighlighted* — Outputs the view of the document specified by the `set viewchangetracking` command. However, if `set viewchangetracking` is set to `changeshighlighted`, then *nothighlighted* outputs the changes applied view of the document. This is the default setting.
- *original* — Outputs the original view of the document.
- *changesapplied* — Outputs the changes applied view of the document.
- *changeshighlighted* — Outputs the changes highlighted view of the document.
- *active* — Outputs the view of the document specified by the `set viewchangetracking` option

Change Tracking and Publishing Example

The following example describes how to enable change tracking markup.

In this example, you don't have to modify the Arbortext XML DocBook stylesheet to view the change tracking markup because the stylesheet outputs the unrecognized markup in red (the stylesheet doesn't output all unrecognized markup; in some instances, it ignores unrecognized markup).

To Enable Change Tracking Markup for HTML Publishing:

1. Open Arbortext Editor, and choose **File ▸ New**.
2. In the **New Document** dialog box, choose **DocBook** in the **Category** list and **ATI XML DocBook V4.0** in the **Type** list, select **Sample**, and then click **OK**.
3. Choose **Tools ▸ Change Tracking ▸ Track Changes**, and then delete the first paragraph in the document. The `para` element will display as red, with red strikethrough to indicate the paragraph has been deleted.
4. Enter the following command on the Arbortext Editor command line:
   `require compose compose.acl`
5. Type the following command on the Arbortext Editor command line:
   `$compose::changetracking="changeshighlighted"`
6. Choose **File ▸ Publish ▸ HTML File** and select the `axdocbook-html.xsl` stylesheet. The change tracking markup for the deleted paragraph displays in red in the resulting HTML file.

Internationalization Considerations

To support different encodings in web output, the `SAXSerializer` filter supports an encoding parameter. Following is a list of valid encodings for this parameter:

- ISO-8859-1
- ISO-8859-2
- ISO-8859-5
- ISO-8859-7
- ISO-8859-9
- Windows-1252
- UTF-8

7

SAX2 Filter Interfaces

This appendix describes the SAX2 filter interfaces. A pipeline consists of a collection of filters that are driven by the SAX2 API.

If you need to extend a filter or implement a new one, you must be familiar with the following SAX2 interfaces:

- ContentHandler

- DTDHandler

- LexicalHandler

For more complete information, refer to the Javadoc for these interfaces at SAX: Simple API for XML: Overview (http://www.saxproject.org/apidoc/ overviewsummary.html)

ContentHandler

ContentHandler is the interface that most SAX2 filters implement. Filters that need to be informed of basic parsing events implement the **ContentHandler** interface.

The pipeline ensures that the filter receives method calls defined by the **ContentHandler** interface from the preceding filter in the pipeline. The preceding filter uses **ContentHandler** to report basic document-related events, such as the start and end of elements and character data.

The pipeline guarantees that the first event seen by a filter is `startDocument` and the last event seen is `endDocument` unless an error terminates the pipeline process.

Since filters allow multiple inputs, the pipeline buffers the SAX events for all inputs except the currently active one. When the active input pipe receives the `endDocument` event, the pipeline releases buffered SAX events for the next pipe. If the next pipe receives the `endDocument` event, then the pipeline moves on to the next pipe, until the SAX events from all the inputs have been processed.

The pipeline ensures that a filter only sees one `startDocument` and one `endDocument`. The pipeline sends the `startDocument` event when it receives the `startDocument` event for the first input pipe. The pipeline sends the `endDocument` event after it receives the `endDocument` event for the last input pipe. The pipeline suppresses all other `startDocument` and `endDocument` events.

```
package org.xml.sax;
public interface ContentHandler
{
 public void setDocumentLocator (Locator locator);
 public void startDocument ()
 throws SAXException;
 public void endDocument()
 throws SAXException;
 public void startPrefixMapping (String prefix, String uri)
 throws SAXException;
 public void endPrefixMapping (String prefix)
 throws SAXException;
 public void startElement (String namespaceURI,
 String localName, String qName,
 Attributes atts)
 throws SAXException;
 public void endElement (String namespaceURI, String localName,
 String qName)
 throws SAXException;
 public void characters (char ch[], int start, int length)
 throws SAXException;
 public void ignorableWhitespace (char ch[], int start,
 int length)
 throws SAXException;
```

```
public void processingInstruction (String target, String data)
throws SAXException;
public void skippedEntity (String name)
throws SAXException;
}
```

DTDHandler

The **DTDHandler** interface is notified of basic DTD-related events. If a SAX filter needs information about notations and unparsed entities or just needs to pass them along, then a filter implements the **DTDHandler** interface. The source filter may use the instance to report notation and unparsed entity declarations to the application.

```
package org.xml.sax;
public interface DTDHandler {
 public abstract void notationDecl
 (String name, String publicId,
 String systemId)
 throws SAXException;
 public abstract void unparsedEntityDecl
 (String name, String publicId,
 String systemId, String notationName)
 throws SAXException;
}
```

LexicalHandler Interface

The **LexicalHandler** is an optional extension handler for SAX2 that provides lexical information about an XML document, such as comments and CDATA section boundaries. XML readers are not required to recognize this handler, and it is not part of core-only SAX2 distributions.

Events in the **LexicalHandler** interface apply to an entire document, not just to the document element. All **LexicalHandler** events must appear between the content handler's `startDocument` and `endDocument` events.

```
package org.xml.sax.ext;
import org.xml.sax.SAXException;
public interface LexicalHandler
{
 public abstract void startDTD
 (String name, String publicId,
 String systemId)
 throws SAXException;
 public abstract void endDTD ()
 throws SAXException;
 public abstract void startEntity (String name)
 throws SAXException;
 public abstract void endEntity (String name)
```

```
 throws SAXException;
 public abstract void startCDATA ()
 throws SAXException;
 public abstract void endCDATA ()
 throws SAXException;
 public abstract void comment
 (char ch[], int start, int length)
 throws SAXException;
}
```

ErrorHandler Interface

The **ErrorHandler** is a basic interface for SAX error handlers. If a pipeline needs to implement customized error handling, some filters must implement this interface. The `Pipeline` element's `errorhandler` attribute must name the filter if it is to act as an `ErrorHandler` resource for the pipeline.

```
package org.xml.sax;
public interface ErrorHandler {
 public abstract void warning
 (SAXParseException exception)
 throws SAXException;
 public abstract void error
 (SAXParseException exception)
 throws SAXException;
 public abstract void fatalError
 (SAXParseException exception)
 throws SAXException;
}
```

DeclHandler

The **DeclHandler** is an optional extension handler for SAX2 that provides information about DTD declarations in an XML document. XML readers are not required to recognize this handler, and it is not part of core-only SAX2 distributions.

📋 **Note**

> Data-related DTD declarations (unparsed entities and notations) are reported using the **DTDHandler** interface.

If you are using the **DeclHandler** interface with the **LexicalHandler** interface, all events will occur between the `startDTD` and `endDTD` events.

```
package org.xml.sax.ext;
import org.xml.sax.SAXException;
```

```
public interface DeclHandler
{
 public abstract void elementDecl (String name, String model)
 throws SAXException;
 public abstract void attributeDecl
 (String eName, String aName, String type,
 String valueDefault, String value)
 throws SAXException;
 public abstract void internalEntityDecl
 (String name, String value)
 throws SAXException;
 public abstract void externalEntityDecl
 (String name, String publicId,
 String systemId)
 throws SAXException;
}
```

EntityResolver

The **EntityResolver** is a basic interface for resolving entities. If a SAX application needs to implement customized handling for external entities, some filters must implement this interface.

The `Pipeline` element's `entityresolver` attribute must name this filter if it is to act as an `EntityResolver` resource for the pipeline.

The **EntityResolver** interface is only called by parsers that need application-specific directions for resolving entities. The value of the entity is returned to the parser by the `resolveEntity` method as an **org.xml.sax.InputSource** object that wraps an `java.io.InputStream`, `java.io.Reader` or system file.

```
package org.xml.sax;
import java.io.IOException;
public interface EntityResolver {
 public abstract InputSource resolveEntity
 (String publicId, String systemId)
 throws SAXException, IOException;
}
```

EntityResolver2

The **EntityResolver2** is an extended interface for mapping external entity references to input sources and for providing missing external subsets.

If a SAX application needs to implement customized handling for external entities, some filters must implement the **EntityResolver2** interface. The `Pipeline` element's `entityresolver2` attribute must name this filter if it is to act as an `EntityResolver2` resource for the pipeline.

```
package org.xml.sax;
import java.io.IOException;
```

```
public interface EntityResolver2 {
 public InputSource getExternalSubset
 (java.lang.String name,
 java.lang.String baseURI)
 throws SAXException,
 java.io.IOException;
 public InputSource resolveEntity
 (java.lang.String name,
 java.lang.String publicId,
 java.lang.String baseURI,
 java.lang.String systemId)
 throws SAXException,
 java.io.IOException;
}
```

8

AOM Reference

This appendix describes the AOM interfaces and methods associated with pipelines and filters. Refer to the *Programmer's Reference* for a complete AOM reference. You can obtain the latest version of the guide from the Reference Documents section of www.ptc.com/en/support.

Application Interface

createComposer Method

createComposer (String *ccfPath*)

The `createComposer` method returns a **Composer** object. The *ccfpath* argument must be a string that is the path to the CCF file.

A content pipeline does not need to process a Arbortext Editor DOM document.

Composer Interface

The **Composer** interface implements a composer configuration. An object that implements this interface is obtained by calling the `createComposer` method.

getDefaultParameters Method

getDefaultParameters

The `getDefaultParameters` method returns a **PropertyMap** of all composer parameters in the pipeline definition. Each key is the name of a *Parameter* element, and its value is a string. If a parameter has no default value, its key value is null.

> ### Note
>
> Refer to the Composer DTD (*Arbortext-path*\doctypes\ composer\composer.dtd) for more information about composer parameters.

runPipeline Method

runPipeline ([*parameters*] PropertyMap)

The `runPipeline` method runs a pipeline associated with a composer object. It returns the status associated with the composer object from the running pipeline. If *parameters* is not specified, the pipeline runs with the default parameters specified in the `Pipeline` element in the CCF file.

getParameterLabel Method

getParameterLabel (String *parameter*)

The `getParameterLabel` method returns the label for the pipeline parameter specified by *parameter*. If *parameter* is not found, this function returns a null value.

The label is text that can display in a user interface.

> **Note**
>
> Refer to the Composer DTD (*Arbortext-path*\doctypes\ composer\composer.dtd) for more information about composer parameters.

getParamDocumentation Method

getParamDocumentation(String *parameter*)

The `getParamDocumentation` method returns the documentation for the pipeline parameter specified by *parameter*. If *parameter* is not found, the function returns a null value.

The documentation describes the *parameter*.

> **Note**
>
> Refer to the Composer DTD (*Arbortext-path*\doctypes\ composer\composer.dtd) for more information about composer parameters.

getParamType Method

getParamType (String *parameter*)

The `getParamType` method returns the type of the pipeline parameter specified by *parameter*. If *parameter* is not found, this function returns a null value.

The type specifies filter type: sink, source, or transformer.

> **Note**
>
> Refer to the Composer DTD (*Arbortext-path*\doctypes\ composer\composer.dtd) for more information about composer parameters.

getParamEnumerationValue Method

getParamEnumerationalues(*parameter*)

The `getParamEnumerationValue` method returns all possible values for the enumeration as a `StringList` if the value of *parameter* is "enumeration". If *parameter* does not exist or is not an enumeration, this function returns a null value.

isParamRequired Method

isParamRequired (*parameter*)

The `isParamRequired method` method determines if the given *parameter* is required. This function returns a false value if *parameter* is missing.

9

ACL Reference

This appendix describes the ACL functions associated with pipelines and filters.

ACL Publishing Functions

This section lists the ACL functions used with content pipelines.

append_composer_path

`append_composer_path` (*dir*[, *prepend*])

The `append_composer_path` function appends *dir* to the directories specified by the `set composerpath` command. If the optional *prepend* argument is specified and non-zero, *dir* is added to the beginning of the path list, removing any later occurrence of the directory. If *prepend* is zero or omitted and if *dir* is already present in the path list, the path list is returned.

For example, if you want to append your composer directory stored in the `composer` subdirectory of the `company` directory, so your CCF files are searched last:
```
append_composer_path("/company/composer")
```

If you want to prepend your composer directory stored in the `company` subdirectory, so your CCF files are searched first:
```
append_composer_path('/company/composer',1)
```

If there is an *Arbortext-path*`\custom\composer` subdirectory at startup, the `\custom\composer` path is automatically prepended to the path for CCF files. Putting your `CCF` files in the `\custom\composer` subdirectory makes them automatically available, avoiding manual steps to add them to the path.

get_composer

`get_composer` (*arr*, *type*[, *doc*])

The `get_composer` function gets a composer handle that corresponds to the composer type from the DCF file of the document to be published. It then populates an array with the composer parameters.

The *arr* parameter specifies the array to populate. The *type* parameter specifies the type of composer. The types are the *type* attribute values specified in the `Compose` element in the DCF file for *doc*.

The *doc* parameter specifies the document for which the composer is being created. If *doc* is `0` or not specified, the current document is used.

The associative array *arr* is populated with the values of the composer parameters. Composer parameters are defined by `Parameter` elements in the `CCF` file's `Interface` element. A `Parameter` element in the DCF file can override the composer parameters specified in the CCF file. A defined parameter that does not have a default value specified in the DCF or CCF file returns an empty string as its value.

The function returns a composer handle or an empty string if an error occurs.

> 📝 **Note**
>
> Calling this function requires parsing the CCF file, unless the file has been cached. Subsequent changes to the CCF file will not be seen unless you flush the composer using the `flush_composer` function.

run_composer

`run_composer` (*composer, arr*)

The `run_composer` function runs the publishing process associated with *composer*, using the parameters in the associative array *arr*.

The *composer* parameter is a composer handle generated by the `get_composer` function. The *arr* parameter specifies an associative array of parameter values. Both parameters are defined in a composer's CCF file.

The function returns `0` if an error occurs, or `1` if the operation was successful.

> 📝 **Note**
>
> Calling this function requires parsing the CCF file, unless the file has been cached. Subsequent changes to the CCF file will not be seen unless you flush the composer using the `flush_composer` function.

flush_composer

`flush_composer` ([*doc*[, *type*]])

The `flush_composer` function removes the cached composer configuration for the publishing *type*.

If *type* is not specified, all composers associated with *doc* are flushed. If *doc* is not specified, then all cached composers are flushed.

list_stylesheets

`list_stylesheets` ()

The `list_stylesheets` function returns the paths to all stylesheets in the compiled stylesheet cache. Arbortext Editor and Arbortext Publishing Engine store XSL stylesheets in this cache after they've been compiled during a

publishing process. By storing these stylesheets, Arbortext Editor and Arbortext Publishing Engine avoid having to recompile them during subsequent publishing runs.

You can use a path returned by this function as an argument to the `clear_stylesheet` function.

clear_stylesheet

`clear_stylesheet` (*`path`*)

The `clear_stylesheet` function removes the stylesheet indicated by *path* from the compiled stylesheet cache. Arbortext Editor and Arbortext Publishing Engine store XSL stylesheets in this cache after they've been compiled during a publishing process. By storing these stylesheets, they don't need to be recompiled during subsequent publishing runs.

You can obtain a list of paths using the `list_stylesheets` function. If *path* is not specified or is an empty string, all stylesheets are cleared from the cache.

For example, if you edited an XSL stylesheet that had previously been used in a publishing process, you would use this function to clear out the stylesheet cache. The updated stylesheet would then be used during the next publishing process.

Interactive and Batch Publishing Functions

You can run publishing processes using ACL functions. These processes can be run interactively or in batch mode.

Interactive Functions

The interactive functions bring up dialog boxes that gather publishing information and then run the publishing process.

compose_htmlfile

`compose_htmlfile` ([*`doc`*])

The `compose_htmlfile` interactive function publishes a document as HTML output. This function assumes the DCF file defines an `htmlfile` composer.

The *doc* parameter specifies the document to process. If *doc* is not specified, the current document is used.

This function returns the following values:

Value	Description
-1	The publishing process was cancelled.
0	An error occurred.
1	The publishing process completed successfully.

📝 **Note**

The **File ▸ Publish ▸ HTML File** menu option uses this function. Publishing parameters are entered in the **Publish HTML File** dialog box.

compose_htmlhelp

compose_htmlhelp (*[doc]*)

The `compose_htmlhelp` interactive function publishes a document as HTML Help output (`.chm`). This function assumes the DCF file defines an `htmlhelp` composer.

The *doc* parameter specifies the document to process. If *doc* is not specified, the current document is used.

This function returns the following values:

Value	Description
-1	The publishing process was cancelled.
0	An error occurred.
1	The publishing process completed successfully.

📝 **Note**

The **File ▸ Publish ▸ For HTML Help** menu option uses this function. Publishing parameters are entered in the **Publish for HTML Help** dialog box.

compose_pdf

compose_pdf (*[doc]*)

The `compose_pdf` interactive function publishes a document as PDF output. This function assumes the DCF file defines a `pdf` composer.

The *doc* parameter specifies the document to process. If *doc* is not specified, the current document is used.

This function returns the following values:

Value	Description
-1	The publishing process was cancelled.
0	An error occurred.
1	The publishing process completed successfully.

📋 **Note**

The **File ▸ Publish ▸ PDF File** menu option uses this function. Publishing parameters are entered in the **Publish PDF File** dialog box.

compose_web

compose_web (*[doc]*)

The `compose_web` interactive function publishes a document as web output. This function assumes the DCF file defines a web composer.

The *doc* parameter specifies the document to process. If *doc* is not specified, the current document is used.

This function returns the following values:

Value	Description
-1	The publishing process was cancelled.
0	An error occurred.
1	The publishing process completed successfully.

📋 **Note**

The **File ▸ Publish ▸ For Web** menu option uses this function. Publishing parameters are entered in the **Publish For Web** dialog box.

compose_xsl

compose_xsl (*[doc]*)

The `compose_xsl` interactive function publishes a document as XSL output. This function assumes the DCF file defines an XSL composer.

The *doc* parameter specifies the document to process. If *doc* is not specified, the current document is used.

This function returns the following values:

Value	Description
-1	The publishing process was cancelled.
0	An error occurred.
1	The publishing process completed successfully.

📝 **Note**

> The **File ▸ Publish ▸ Using XSL** menu option uses this function. Publishing parameters are entered in the **Publish Using XSL** dialog box.

Batch Functions

The publishing batch functions accept the document ID and an array of parameters. They are part of the ACL `compose.acl` package, so you'll need to specify it with the function, for example `compose::compose_for_htmlfile`.

compose_for_htmlfile

compose_for_htmlfile (*doc*, *arr*)

The `compose_for_htmlfile` batch function publishes a document as HTML output. This function assumes the DCF file defines an `htmlfile` composer.

The *doc* parameter specifies the document to process. If *doc* is not specified, the current document is used.

The *arr* parameter is an associative array of properties to pass to the composer. If the array is empty, the publishing process proceeds interactively. Otherwise, the *arr* parameter overrides any entries in the array populated by the `get_composer` function.

This function returns the following values:

Value	Description
-1	The publishing process was cancelled.
0	An error occurred.
1	The publishing process completed successfully.

compose_for_htmlhelp

compose_for_htmlhelp (*doc*, *arr*)

The `compose_for_htmlhelp` batch function composes a document as HTML Help output. This function assumes the DCF file defines an `htmlhelp` composer.

The *doc* parameter specifies the document to process. If *doc* is not specified, the current document is used.

The *arr* parameter is an associative array of properties to pass to the composer. If the array is empty, the publishing process proceeds interactively. Otherwise, the *arr* parameter overrides any entries in the array populated by the `get_composer` function.

This function returns the following values:

Value	Description
-1	The publishing process was cancelled.
0	An error occurred.
1	The publishing process completed successfully.

compose_for_pdf

compose_for_pdf (*doc*, *arr*)

The `compose_for_pdf` batch function publishes a document as PDF output. This function assumes the DCF file defines a `pdf` composer.

The *doc* parameter specifies the document to process. If *doc* is not specified, the current document is used.

The *arr* parameter is an associative array of properties to pass to the composer. If the array is empty, the publishing process proceeds interactively. Otherwise, the *arr* parameter overrides any entries in the array populated by the `get_composer` function.

This function returns the following values:

Value	Description
-1	The publishing process was cancelled.
0	An error occurred.
1	The publishing process completed successfully.

compose_for_wap

compose_for_wap (*doc*, *arr*)

The `compose_for_wap` batch function publishes a document as WAP output. This function assumes the DCF file defines a `wap` composer.

The *doc* parameter specifies the document to process. If *doc* is not specified, the current document is used.

The *arr* parameter is an associative array of properties to pass to the composer. If the array is empty, the publishing process proceeds interactively. Otherwise, the *arr* parameter overrides any entries in the array populated by the `get_ composer` function.

This function returns the following values:

Value	Description
-1	The publishing process was cancelled.
0	An error occurred.
1	The publishing process completed successfully.

compose_for_web

compose_for_web (*doc*, *arr*)

The `compose_for_web` batch function publishes a document as web output. This function assumes the DCF file defines a `web` composer.

The *doc* parameter specifies the document to process. If *doc* is not specified, the current document is used.

The *arr* parameter is an associative array of properties to pass to the composer. If the array is empty, the publishing process proceeds interactively. Otherwise, the *arr* parameter overrides any entries in the array populated by the `get_ composer` function.

This function returns the following values:

Value	Description
`-1`	The publishing process was cancelled.
`0`	An error occurred.
`1`	The publishing process completed successfully.

compose_using_xsl

`compose_using_xsl` (*doc*, *arr*)

The `compose_using_xsl` batch function publishes a document as XSL output. This function assumes the DCF file defines an XSL composer.

The *doc* parameter specifies the document to process. If *doc* is not specified, the current document is used.

The *arr* parameter is an associative array of properties to pass to the composer. If the array is empty, the publishing process proceeds interactively. Otherwise, the *arr* parameter overrides any entries in the array populated by the `get_composer` function.

This function returns the following values:

Value	Description
`-1`	The publishing process was cancelled.
`0`	An error occurred.
`1`	The publishing process completed successfully.

compose_for_xslfo

`compose_for_xslfo` (*doc*, *arr*)

The `compose_for_xslfo` batch function publishes a document as XSL-FO output. This function assumes the DCF file defines an `xslfo` composer.

The *doc* parameter specifies the document to process. If *doc* is not specified, the current document is used.

The *arr* parameter is an associative array of properties to pass to the composer. The *arr* parameter overrides any entries in the array populated by the `get_composer` function.

This function returns the following values:

Value	Description
-1	The publishing process was cancelled.
0	An error occurred.
1	The publishing process completed successfully.

Publish Menu Options

The following table lists the **File ▸ Publish** menu options that use the pipeline architecture.

Publish menu option	Command (`main::`)	CCF file
For Web	compose_for_ webcompose_web	`web.ccf`
For HTML Help	compose_for_ htmlhelpcompose_ htmlhelp	`htmlhelp.ccf`
HTML file	compose_for_ htmlfilecompose_htmlfile	`htmlfile.ccf`
PDF	compose_for_ pdfcompose_pdf	`pdf.ccf`
Using XSL	compose_using_ xslcompose_xsl	`xsl.ccf`

Event Log Functions

get_composer_log_contents

`get_composer_log_contents`([*html*])

The `get_composer_log_contents` function returns the contents of the log file associated with the last publishing operation as a formatted multiline string. If the *html* parameter is present and non-zero, the string is HTML wrapped in a `pre` element.

📝 **Note**

This function doesn't return anything if the Event Log window is closed after a composer operation.

---

get_composer_log_doc

`get_composer_log_doc` ()

The `get_composer_log_doc` function returns the ID of the read-only document containing the log output from the last publishing operation.

composer_log

`composer_log` (*destination*, *severity*, *verbosity*)

The `composer_log` function sets the preferences for the Event Log error file. Use this log file to troubleshoot publishing problems.

destination specifies the output destination for composer events:

Setting	Description
0	Outputs to the Java Console.
1	Outputs to the Event Log window.
string	Outputs a text file to the specified path.

severity specifies the minimum severity of output events to generate:

Setting	Description
$eventlog::SEVERITY_INFO	Includes all diagnostic events, including errors and warnings.
$eventlog::SEVERITY_WARNING	Includes errors and warnings, but no informational events. This is the default setting.
$eventlog::SEVERITY_ERROR	Includes errors only.

verbosity indicates the amount of contextual information to generate in the log:

Setting	Description
0	No contextual information.
1	Exception message.
2	Exception message and traceback. This is the default setting.

For example,
```
# Trace all possible composer events to a file.
composer_log("c:\\temp\\composer.log", $eventlog::SEVERITY_INFO, 2)
# Trace warnings and errors (but not informational events) to
# the java console and don't include traceback information.
composer_log(0, $eventlog::SEVERITY_WARNING, 1);
# Trace only errors (but not warnings and informational events)
```

```
# to the event log.
composer_log(1, $eventlog::SEVERITY_ERROR);
```

show_composer_log

show_composer_log ()

The `show_composer_log` function displays the Event Log window. If there is no Event Log window, this function creates it.

Core Functions

Arbortext Editor and Arbortext Publishing Engine provide a set of core publishing functions. We recommend that you only use these functions for debugging purposes as they may change in future releases.

composer_types

composer_types (*arr*[, *doc*])

The `composer_types` function populates an array *arr* with the available composer types for a document.

The *doc* parameter specifies the document for which the composer type information is being requested. If *doc* is not specified, the current document is used.

This function returns the size of the populated array or `0` if an error occurs.

composer_sysid

composer_sysid (*type*[, *doc*])

The `composer_sysid` function returns the composer handle for the specified composer type associated with this document. The system ID acts as a handle.

The *type* parameter specifies the type of composer. The type is specified by the `Compose` element's `type` attribute in the DCF file for *doc*.

The *doc* parameter specifies the document for which the composer handle is being created. If *doc* is not specified, the current document is used.

The associated composer name is defined by the `Compose` element's `config` attribute in the DCF file. This attribute may be a path to a CCF file or the CCF file name. If the file name does not specify the `.ccf` extension, the file name is concatenated to the type name to construct the composer name.

If the composer name is not an absolute path, then the `doctype` directory is searched for the specified file. If the file is not found, the `set composerpath` directories are searched in order. If a matching file is found, its full cannonicalized path becomes the composer handle.

This function returns an empty string if the composer type was not defined or if the search path does not contain an associated composer.

composer_get_ccf_parameters

composer_get_ccf_parameters(*arr, comp*)

The `composer_get_ccf_parameters` function populates an array *arr* with the parameters associated with a composer. The *comp* parameter specifies a handle for the composer. The handle should be the handle returned by the `composer_sysid` function.

The resulting array is an associative array in which each index is the name of a composer interface parameter, as defined by the *Parameter* element in the composer's CCF file. The entry at each index is the parameter's default value. The entry is an empty string if a parameter has no default value.

This function returns the size of the populated array or `0` if an error occurs.

> ### Note
>
> Calling this function requires parsing the CCF file unless the CCF file has been cached. Subsequent changes to the CCF file will not be seen unless the composer is flushed by a call to the `composer_flush` function.

---

composer_get_dcf_parameters

composer_get_dcf_parameters(*arr, type*[, *doc*])

The `composer_get_dcf_parameters` function populates the array *arr* associated with a publishing type for *doc*. The *type* parameter specifies the composer type. The type is specified by the `Compose` element's `type` attribute in the DCF file for *doc*.

The *doc* parameter specifies the document for which the composer is being created. If *doc* is not specified, the current document is used.

The resulting array is an associative array where each index is a name of a composer interface parameter, as defined by the *Parameter* element in the DCF for the specified publishing type. The entry at each index is the parameter's default value. The entry is an empty string if a parameter has no default value.

This function returns the size of the populated array or `0` if an error occurs.

Content Pipeline Guide

composer_get_all_parameters

composer_get_all_parameters(*arr*, *type*[, *doc*])

The `composer_get_all_parameters` function populates an array *arr* with all the parameters associated with the publishing *type*. The *type* parameter specifies a composer type. The type is specified by the `Compose` element's `type` attribute in the DCF file for *doc*.

The *doc* parameter specifies the document for which the composer is being created. If *doc* is not specified, the current document is used.

The resulting array is an associative array where each index is a name of a composer interface parameter, as defined by the `Parameter` element in the DCF file for the specified publishing type or by the `Parameter` element in the associated CCF file.

The entry at each index is the parameter's default value, with the default values in the DCF file taking precedence. If a parameter has no default value in the DCF or CCF file, the entry is an empty string.

As a special case, the `document` parameter is added to the array and given the value of *doc*.

The return value is the size of the populated array, or zero on failure.

📝 Note

Calling this function requires parsing the CCF file, unless the CCF file has been cached. Subsequent changes to the CCF will not be seen unless the composer is flushed by a call to the `composer_flush` function.

---

composer_get_parameter_info

composer_get_parameter_info(*arr*, *comp*, *name*)

The `composer_get_parameter_info` function populates an array *arr* with information about a parameter in a CCF file. The *comp* parameter specifies a handle for the composer. The handle specified should be the handle returned by the `composer_sysid` function.

The *name* parameter specifies a parameter in the CCF file, an index of the array populated by the `composer_get_ccf_parameters` function.

The resulting array is an associative array where each index is a property of the named parameter. Valid properties are:

Property	Description
default	A parameter's default value.
label	The name of a parameter, to be displayed in the user interface.
documentation	The documentation for a parameter.
enumeration	A list of all possible values for a parameter.
type	The type of a parameter's value.

The *type* property can be boolean, byte, short, int, long, char, float, double, or enumeration. If *type* is enumeration, the enumeration property should contain a list of all possible values. These are concatenated into one string with a pipe (`|`) character as a delimiter. Use the `split` function to convert this string into an array.

This function returns the size of the populated array or `0` if an error occurs.

Note

Calling this function requires parsing the CCF file unless the CCF file has been cached. Subsequent changes to the CCF file will not be seen unless the composer is flushed by a call to the `composer_flush` function.

composer_check

`composer_check` (*comp*)

The `composer_check` function checks the CCF file against the composer DTD for context errors. The *comp* parameter specifies a handle for the composer. The handle should be the handle returned by the `composer_sysid` function.

This function returns a `0` if an error occurs or the document is malformed. It returns a `1` if the CCF file is well-formed.

Note

Calling this function requires parsing the CCF file unless the CCF file has been cached. Subsequent changes to the CCF file will not be seen unless the composer is flushed by a call to the `composer_flush` function.

composer_flush

composer_flush ([*comp*])

The `composer_flush` function removes the cached composer or all cached composers from memory. The *comp* parameter specifies a handle for the composer. The handle should be the handle returned by the `composer_sysid` function. If *comp* is not specified, all cached composers are removed.

Once a composer CCF file is parsed, the results are stored in memory. Composers can be written so that they retain static information about a previous run. If this static information can interfere with the composer operation, you should flush the composer after each use.

This function returns a `0` if the composer was not cached. It returns a `1` if the composers were removed.

10

Java Reference

This appendix describes the Java interfaces and classes associated with pipelines and filters.

Interfaces

The publishing framework defines a set of interfaces that Java classes need to implement to enable filters in the pipeline.

SAXInterfaceProvider

The **SAXInterfaceProvider** interface provides methods that get the various SAX2 interfaces. Since a filter may implement a subset of the seven SAX2 interfaces, this interface allows the composer to query the object that implements this interface for the individual SAX2 interfaces.

```
public interface SAXInterfaceProvider {
 /** Returns the ContentHandler or null. */
 ContentHandler getContentHandler();
 /** Returns the DTDHandler or null. */
 DTDHandler getDTDHandler();
 /** Returns the ErrorHandler or null. */
 ErrorHandler getErrorHandler();
 /** Returns the EntityResolver or null. */
 EntityResolver getEntityResolver();
 /** Returns the LexicalHandler or null. */
 LexicalHandler getLexicalHandler();
 /** Returns the DeclHandler or null. */
 DeclHandler getDeclHandler();
 /** Returns the EntityResolver2 or null. */
 EntityResolver2 getEntityResolver2();
}
```

SAXInterfaceRecipient

SAXInterfaceRecipient is a complimentary interface to **SAXInterfaceProvider**. This interface allows you to set interfaces.

```
public interface SAXInterfaceRecipient {
 /**
 * Set the ContentHandler.
 * @param ch is the ContentHandler to be set
 */
 void setContentHandler(ContentHandler ch);
 /**
 * Set the DTDHandler.
 * @param dh is the DTDHandler to be set.
 */
 void setDTDHandler(DTDHandler dh);
 /**
 * Set the ErrorHandler
 * @param eh is the ErrorHandler to be set.
 */
 void setErrorHandler(ErrorHandler eh);
 /**
```

Content Pipeline Guide

```
 * Set the EntityResolver.
 * @param er is the EntityResolver to be set.
 */
void setEntityResolver(EntityResolver er);
/**
 * Sets the LexicalHandler.
 * @param lh is the LexicalHandler to be set.
 */
void setLexicalHandler(LexicalHandler lh);
/**
 * Sets the DeclHandler.
 * @param dh is the DeclHandler to be set.
 */
void setDeclHandler(DeclHandler dh);
/**
 * Sets the EntityResolver2.
 * @param er2 is the EntityResolver2 to be set.
 */
void setEntityResolver2(EntityResolver2 er2);
}
```

FilterControl

FilterControl specifies the filter control interface used by the publishing framework to configure and control the filter. The control interface uses methods to initialize and run filters (if it is a source), as well as to set output and resource handlers.

```
public interface FilterControl {
 /**
 * Sets the object that handles the filter's SAX events.
 * The output goes to the default (or unnamed) pipe.
 * @param outputHandler is a SAXInterfaceProvider object
 * that can be used to get the handles to the desired
 * SAX interfaces that handle the filter's SAX events.
 */
 void setOutputHandler(SAXInterfaceProvider outputHandler);
 /**
 * Sets the object that handles the filter SAX events for
 * the given pipename.
 *
 * @param pipeName is the name of the output pipe.
 * @param outputHandler is a SAXInterfaceProvider that can
 * be used to get the handles to the desired SAX interfaces
 * that handle the filter's SAX events.
 */
 void setOutputHandler(String pipeName, SAXInterfaceProvider
 outputHandler);
 /**
 * Gets the output handler object for the unnamed
 * (or default) pipe.
```

```
* @returns is a SAXInterfaceProvider that is the output
* handler.
*/
SAXInterfaceProvider getOutputHandler();
/**
* Gets the output handler object for the named pipe.
* @returns is a SAXInterfaceProvider that is the output
* handler.
*/
SAXInterfaceProvider getOutputHandler(String pipeName);

/**
* Sets the ErrorHandler object. If the filter implements the
* ErrorHandler interface, this method can ignore the object
* passed in.
*/
void setErrorHandlerResource(ErrorHandler theErrorHandler);

/**
* Initializes the filter with the given set of parameters.
*
* @param parameters is an object implementing the Map
* interface that contains name value pairs representing the
* filter's parameters.
* @throws is an exception if there is any problem with the
* initialization such as missing or illegal parameters.
*/
void initFilter(java.util.Map parameters) throws Exception;
/**
* Releases all resources used by the filter. The adapter
* can be destroyed after this method is called with no
* danger of resource leakage.
*/
void destroyFilter();
/**
* Runs the filter. This method only applies for filters
* that can generate SAX events. It is recommended that the
* implementing class throw an UnsupportedOperationException
* if the runFilter method does not apply.
* @throws is an exception if there is a problem running
* the filter.
*/
void runFilter() throws Exception;
/**
* Sets the EntityResolver object that the adapter can pass
* to the filter for the filter's entity resolution needs.
* The adapter can ignore the resolver if its filter
* implements the EntityResolver interface.
*/
void setEntityResolverResource(EntityResolver theResolver);
/**
```

```
* Sets the EntityResolver2 object that the adapter can pass
* to the filter for the filter's entity resolution needs.
* The adapter can ignore the resolver if its filter
* implements the EntityResolver2 interface.
*/
void setEntityResolver2Resource(EntityResolver2 theResolver);
/**
* Informs the adapter that the filter's parameter has
* changed to the new value. The adapter passes the
* information on to the filter if the filter's class
* matches the filterClass passed in.
* @param filterClass is the name of the class of the filter
* instance to which this parameter is intended.
* @param key is the name of the parameter
* @param value is the new value.
*/
void parameterChanged(String filterClass, String key, String
newValue);
/**
* Indicates the beginning of a named region. The name of the
* region is specified in the name parameter. The data parameter
* is a string that will be associated with the region.
* @param name is the name of the region.
* @param data is any data associated with the region. The
* format of the data is user defined.
*/
void regionBegin(String name, String data);
/**
* Indicates the end of the region. The regionBegin method is
* guaranteed to have been called with the same name.
* @param name is the name of the region.
*/
void regionEnd(String name);
/**
* Command to set the current location from the file being
* processed.
*
*/
void setFileLocation(String systemid, int line, int column);
/**
* Command to set the current location where the source of the
* document is Arbortext.
* @param location is the location as a string. The format of the
* string is described in Arbortext Editor help.
*/
void setEpicLocation(String location);
}// FilterControl
```

FilterAdapter

The **FilterAdapter** interface combines the **SAXInterfaceProvider** and **FilterControl** interfaces. Objects that act as filter adapters must implement this interface. The `adapaterClass` attribute for a `FilterDef` element in a CCF file must implement this interface.

```
public interface FilterAdapter extends SAXInterfaceProvider,
FilterControl {
}
```

SAXFilter

SAXFilter is a convenience interface that extends the seven SAX2 interfaces and the **FilterControl** interface. You can use objects that implement this interface in a pipeline. You can also use these objects with the `DefaultFilterAdapter` so that you do not have to write a custom adapter class.

```
public interface SAXFilter extends
 ContentHandler,
 DTDHandler,
 LexicalHandler,
 DeclHandler,
 ErrorHandler,
 EntityResolver,
 EntityResolver2,
 FilterControl
{
}
```

Helper Classes

The publishing framework provides subclasses to help you quickly write filters and adapters.

DefaultSAXFilter

The `DefaultSAXFilter` filter implements the **SAXFilter** interface. This filter repeats its input SAX events to its output. You can use this filter and its subclasses with the `DefaultFilterAdapter`.

📝 Note

We recommend that all non-buffering filters extend this class.

DefaultFilterAdapter

The `DefaultFilterAdapter` is a default implementation of an adapter. It specifies the output handlers for the filter. This implementation is adequate for filters that extend the `DefaultSAXFilter` class.

11

Distributed Files

This appendix describes the publishing files that are distributed with Arbortext Editor and Arbortext Publishing Engine. It also describes the sample files distributed for the *Content Pipeline Guide*.

File Types

Arbortext Editor and Arbortext Publishing Engine provide a set of CCF files and Java classes that implement the filters. You can use these for reference or for extending publishing functionality.

CCF Files

A set of CCF files defining several content pipelines are included in the `Arbortext-path\composer` directory:

- `htmlfile.ccf` — Transforms documents to HTML (output as a file if using an XSL stylesheet or as a Arbortext Editor document if using a FOSI stylesheet).
- `htmlhelp.ccf` — Transforms documents to HTML Help.
- `pdf.ccf` — Transforms XML to an XSL-FO document in Arbortext Editor. This document can then be used to create PDF files.
- `profile.ccf`— Profiles documents to a Arbortext Editor document.
- `schemavalidator.ccf` — Validates documents against a specified schema.
- `web.ccf`— Publishes to web. This involves chunking sections of the document into separate HTML files.
- `xsl.ccf` — Outputs the transformed document to a file.
- `xslfo.ccf` — Outputs the transformed XSL-FO document to a Arbortext Editor document.

Filters

The distributed filters are saved as entities with the name *filtername*`.ent`. By saving these filters as entities, they can be reused across CCF files. These files are included in the `Arbortext-path\composer` directory

Following is a list of the distributed filters:

- `charSubFilter.ent` — Substitutes characters according to user-specified substitution file in user-specified elements.
- `chunker.ent` — Sink filter that splits (or chunks) a stream of SAX events into multiple files.
- `debugFilter.ent` — Monitors SAX events that pass through the pipeline. Users can control which methods and handlers are monitored.
- `entityResolver.ent` — Resolves entities used in the pipeline. For example, you could use this filter to resolve entities contained in the document to be processed or resolve the file location for the entity substitution file used in HTML output.

- `epicGenerator.ent` — Source filter that produces SAX events by parsing a Arbortext Editor document.
- `epicSerializer.ent` — Sink filter that generates a Arbortext Editor document from SAX events.
- `errorHandler.ent` — Parses error handler calls (warning, error, fatalError), emits error messages as SAX events, and sends them to the output handler connected to this handler.
- `fileGenerator.ent` — Source filter that produces SAX events by parsing an XML file.
- `fileSerializer.ent` — Sink filter that serializes SAX events to a system file.
- `graphicConverter.ent` — Copies graphic references to the appropriate destination, and then modifies attributes to point to the new location. In future releases, this filter will convert graphic file formats from one file format to another (because most browsers support them, the GIF, JPEG, and PNG file formats will continue to be copied rather than converted).
- `htmlHelpAdapter.ent` — Sink filter that takes HTML Help parameters and converts them to XSLTransformer parameters.
- `log4jErrorHandler.ent`— Converts error handling calls (warning, error, fatalError) into new SAX events and passes them to its output handler using the log4j logging mechanism. The format and output destination depend on the appender and layout associated with the current filter class. For more information, refer to the log4j documentation at `http://jakarta.apache.org/log4j/docs/index.html`.
- `namespaceFixer.ent` — Transformer filter that changes namespace prefixes. You can use this filter to strip the namespaced content in the `Boilerplate` filter.
- `piToElementConverter.ent` — Converts processing instructions (PIs) into elements. During serialization, Arbortext-specific PIs (for example, `_font` or `_cellfont`) are transformed into text nodes containing XML markup and surrounded by `javax.xml.transform.disable-output-escaping` and `javax.xml.transform.enable-output-escaping` to retain escaping characters. When read back from the file system, the text nodes are parsed into elements.

📮 **Note**

In this release, the intermediate serialization is eliminated and each text node is emitted as a character event instead of a set of elements. To solve the problem, the PIs are first converted into elements. New stylesheet templates are provided to handle the converted elements.

- `profiler.ent` — Transformer filter that suppresses elements that don't match certain criteria.
- `switch.ent` — Determines the flow of SAX events across multiple forks in the pipeline. The SAX events are routed to the appropriate pipe.
- `xslTransformer.ent` — Compiles an XSL stylesheet and uses it to transform the input stream to the output stream.

Content Pipeline Guide Files

The *Content Pipeline Guide* comes with a set of examples. The examples include CCF files, ACL source files, and Java source and class files. These files are distributed in your installation:

```
Arbortext-path\samples\pipeline\content-pipeline-
examples.zip
```

CCF Files

The following CCF files are used in examples in the *Content Pipeline Guide*. These CCF files should be placed in `Arbortext-path\custom\composer` or a directory that is in the `set composerpath` path.

- `simpleccf.ccf` — Defines a two-filter content pipeline that reads a Arbortext Editor XML document and then serializes it to a file on the file system
- `copyright.ccf` — Defines a pipeline that appends a copyright statement to the input document and writes the output to a file or a Arbortext Editor document.
- `grep.ccf` — Defines a pipeline that provides functionality similar to the `grep` command.

ACL Files

The ACL files should be placed in the `Arbortext-path\custom\init` folder. These files define functions for running the CCF files distributed with Arbortext Editor and Arbortext Publishing Engine. Documentation regarding the functions can be found within the files.

Java Files

The Java source and class files are located in `examples.jar`. This file should be placed in `Arbortext-path\custom\classes` directory. All the examples are in the `com.arbortext.epic.compose.examples` package.

Index

S

show_composer_log function, 79
Stylesheets
   clearing from cache, 70
   determining paths to, 69