AXT Programmers Guide

Version 2.0.0

Content

Introduction 3
Core interaction and parameter set-up 4
Writing a filter component 6
  Exceptions 7
  Filter parameters 7
  Plugging it in 8
Writing an error handler 9
Sharing and exposing global data across multiple filter instances (AXT v.2.0.0) 9
Advanced concepts 12
  Method invocation and passing of object definitions to filters 12

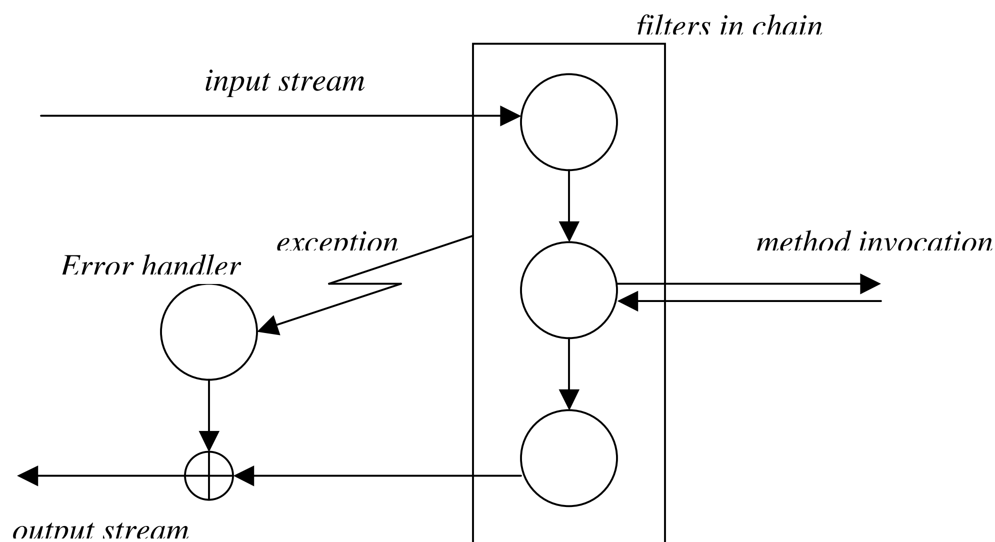
Introduction

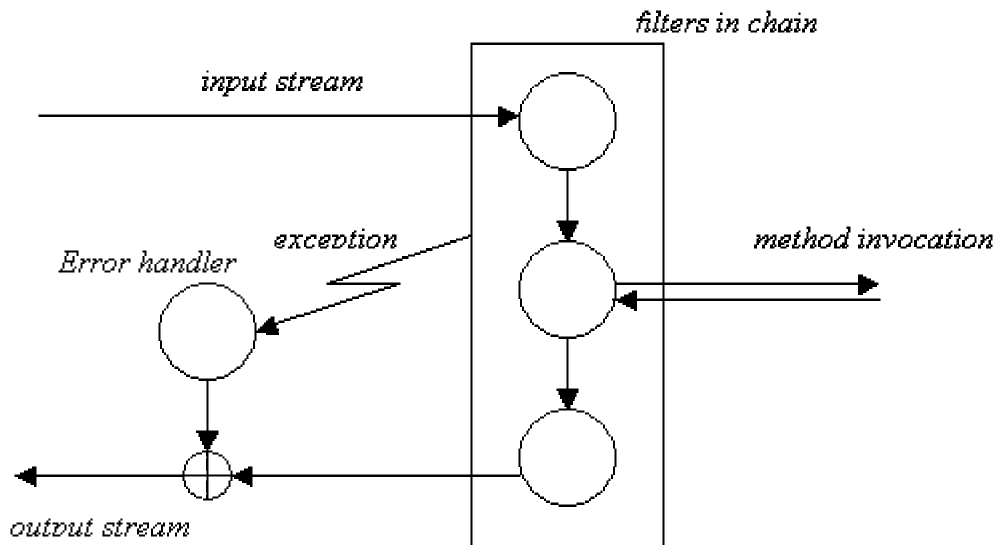
This guide is an introduction to the API for the Agetor XML Transformation (AXT) package. The reader is assumed familiar with the general concepts introduced in the AXT user guide. This guide should enable you to write you own plugable AXT components when your transformation needs goes beyond what is bundled with your AXT package.

When an input document reaches the AXT server a transformation process is selected. This transformation process is configurable and consists of a chain of filter that may each alter the data stream somehow. Each filter component uses the output of the previous filter in the chain as its input. A step in the filtering process may be a format conversion, addition or removal of data or a method invocation into a back-end system using the data in the input stream and the invocation result as output. The result of the filtering process is returned as the output document.

Besides this primary dataflow a number of secondary filter processes may be defined. Each of these are filter chains in themselves as described above. These filter chains differ from the main flow though, by not providing any resulting output (the output of the final component in such a filter stream is consumed and discarded by the system). These secondary filter streams are termed *routers* since their rationale is to route data (probably somehow altered) to a secondary destination. For example a router might format the stream as a text mail and send it to a configured recipient.

The filters working on the data stream execute in parallel as the data stream runs through them. In the event of a fatal error a filters may throw an exception. Such fatal errors must be reported to the client that sent the input document. Since the client may rely on a specific data format the error must be formatted accordingly. An error-handling component (*errorhandler*) may be specified for each transformation defined, allowing for the appropriate error formatting. The default error handler will format the error as XML.





The basic AXT package contains the framework for configuring basic transformation components such as filters and error handlers into transformation processes as well as selecting and executing them based on input document properties. Some basic filters and error handlers are provided as well.

When the processing needs goes beyond that provided by the AXT package, the plugable component oriented architecture of AXT allows for the necessary customisation. Useful, but not provided filters, could be filters mapping between the proprietary data format of a business partner and XML. Once converted to XML the full range of data manipulation with XSL scripts as well as automated method invocation is at hand in AXT.

In AXT a plugable component is a Java class implementing a predefined interface. The type of interface depends on the type of the component. To ease the development of an AXT component, abstract implementations of the interfaces provides the bulk of the necessary functionality, and new components should extend them. The table below list the corresponding component type, interface and abstract implementations.

Component type	Interface	Abstract or extendable implementation
Filter	`AXTFilter`	`AbstractAXTFilter`
Error handler	`AXTErrorHandler`	*None*

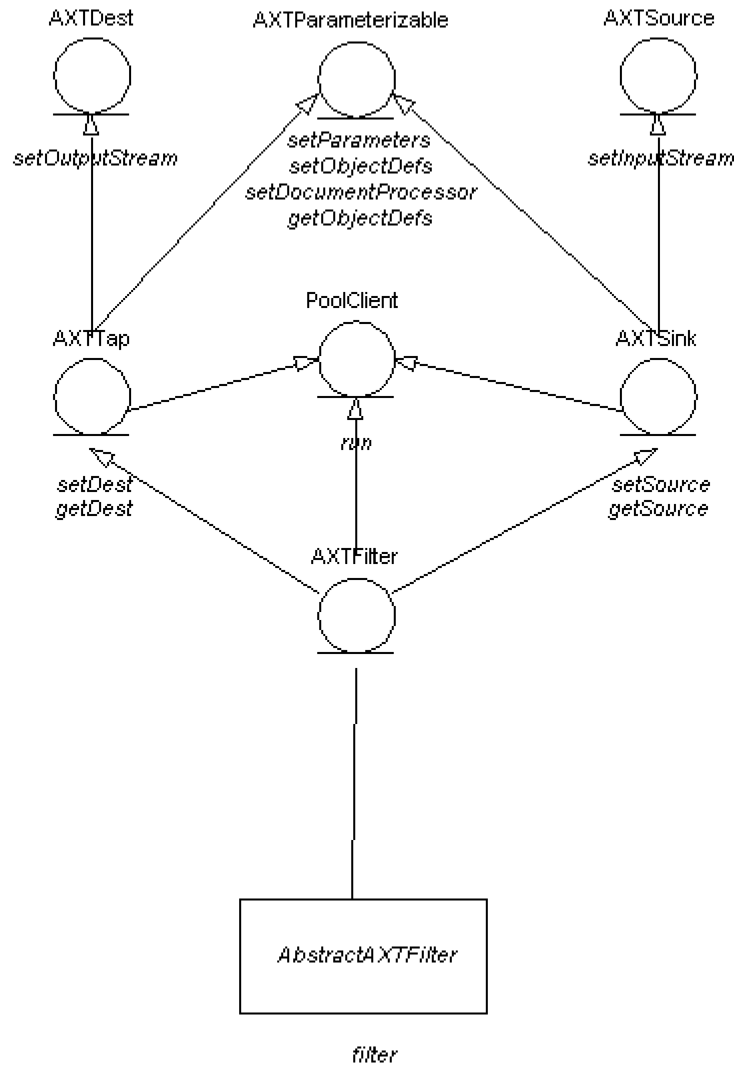
Core interaction and parameter set-up

The illustration below shows the full interface hierarchy of a filter in AXT. For the AXT core to instantiate and run a plugable filter the filter must implement the `AXTfilter` interface. Using the set methods of the interface classes the core can link different filter instances using input and output streams. The set methods also allows the core to pass references to parameter lists set up in the AXT configuration file, as well as references to a few useful core classes.

When the core realizes that in AXT transformation is to be performed, it does the following things:

- Constructs an instance of each filter to be used in the transformation
- Constructs input and output streams and links the filter instances using these
- Invokes the `set`-methods of the filters so parameters and stream references are available to the filter
- Invokes the `run` method of each filter in a separate thread so they can execute in parallel
- Awaits the successful termination of each thread *or* a thrown exception
- In the event of a thrown exception any error-router matching thrown exception are executed
- If no exception was thrown the success-routers (if any) are executed

To alleviate the AXT application programmer from implementing the methods in the interfaces an abstract implementation is provided with AXT. This implementation stores the references handed to the `set`-methods and returns them through `get`-methods. The implementation also contains an implementation of the `run`-method. This run method invokes an abstract method (filter) that the application programmer must provide. The filter method receives an input and output stream as well as the set of parameters.



Figur 1AXT interfaces

Writing a filter component

Writing a filter for AXT is very simple. Use the `AbstractAXTFilter` class as a starting point and you only need to implement one method - `filter`.

The code snippets below shows the most basic AXT filter that will compile.

```
package dk.bording.axt.examples;

import java.io.InputStream;
import java.io.OutputStream;
```

```
import dk.bording.AXT.tc.ext.AbstractAXTFilter;
import dk.bording.AXT.core.conf.AXTParameters;

public class ExampleFilter extends AbstractAXTFilter
{
  public void filter(InputStream in, OutputStream out, AXTParameters p)
         throws Throwable {
  }
}
```

As can be seen the filter method is handed three parameters. The first two are ordinary Java input and output streams. The filter should read data from the input stream, modify it as needed and finally write the result of the output stream. The third parameters is an object containing simple name-value parameters.

The above filter class isn't very useful since it doesn't perform any filtering of the input and provides no output. Usually the task of filter will be to provide an output that is the result of a transformation or modification of the input read. Thus the filter should read from the input stream and write to the output stream. The next example copies the input to the output converting all characters to uppercase representation on the fly.

```
public void filter(InputStream in, OutputStream out, AXTParameters p) throws
       IOException {
  int ch;

  while((ch=in.read())!=-1) {
    out.write(Char.toUpperCase(ch));
  }
}
```

Exceptions

When your filter needs to signal an error it must throw an exception. You may use one of the predefined extending exceptions or write your own. The predefined exceptions extending `AXTException` are:

```
dk.bording.AXT.AXTInputException
dk.bording.AXT.AXTOutputException
dk.bording.AXT.AXTProcessException
dk.bording.AXT.AXTSecurityException
dk.bording.AXT.AXTMissingParameterException
dk.bording.AXT.AXTIllegalParameterException
```

These exception types have constructors that accepts an exception message as a string. You may however throw any exception types you see fit. Whatever exception types your filter throws you should document these by explicitly throwing these in the filter method signature. This allows the person that configures your filter to initiate actions based on particular exceptions.

Filter parameters

The set of simple name-value parameters given to the filter is set up in the AXT configuration file. They are filter specific and may be used to control that behaviour of the filter. The object holding the name value parameters is an instance of the `AXTParameters` class. This class extends the Java `Properties` class. Use the methods provided by this class or any super class (like `Hashtable`) to access the parameters.

To extract a value using a key you may use one of two methods. `getParameter(String key)` will return the simple string value of the key parameters or `null` if the key was not provided. A more restrictive method `getRequiredParameter(String key)` will throw the `AXTMissingParameterException` if the named key is absent.

To enable the passing of multiple values for one key the method `getStrings` is provided by the `AXTParameters` class. For example if a filter needs to receive multiple script-names for one parameters name these values may be separated with an unambiguous delimiter character like semicolon. Below is shown the configuration of a filter receiving to script names for the key `xsl-script`.

```
<filter class="dk.bording.axt.tc.xsl.XSLProcess">
  <param name="xsl-script" value="xmlorder.xsl;xmlpostproc.xsl" />
</filter>
```

To extract these values to a string array use to `getStrings` method in your filter as shown below. Another delimiter character may be used but require you to specify that delimiter as a second parameters to `getStrings` method.

```
String []ss = p.getStrings("xsl-script");
if(ss.length==0) {
  throw new AXTMissingParameterException(key);
}
```

Plugging it in

When your filter compiles you may start testing it using AXT. Perhaps you may want to set your filter up as the only filter in a transformation until you have verified the correct behaviour. The transformation below examplifies the set-up of one filter transforming text input to XML. The error handler ensures that output is formatted as text in the event of an error during processing. Consult the AXT user guide for details on AXT configuration.

```
<doc name="comma2xml">

  <key name="ip" value="*"/>
  <key name="content-type" value="text/plain"/>
  <key name="url" value="comma2xml"/>

  <errorhandler class="dk.bording.axt.tc.err.TextErrorHandler" />

  <!-- map input to basic xml -->
  <filter class="dk.bording.axt.tc.txtxml.Text2XML">
    <param name="confname" value="orders" />
  </filter>
```

```
</doc>
```

Writing an error handler

Writing an error handler implies implementing the simple `AXTErrorHandler` interface. The interface only has one method `print`. The method is handed the output stream to which it must the write an appropriate error message, as well as the exceptions thrown and the parameters that the AXT core received from the requesting client. Also an object `AXTDocument` representing the full description of the transformation chosen is provided.

```
public void print(OutputStream out, AXTDocument doc, Throwable e , AXTParameters
p) throws IOException ;
```

Below is a simple implementation of a text error handler. It will return the text of the thrown exception with the prefix “ERROR:”. In the example below the thrown exception is an `AXTIllegalParameterException`.

```
package dk.bording.axt.tc.err;
import java.io.OutputStream;
import java.io.IOException;
import dk.bording.axt.om.AXTErrorhandler;
import dk.bording.axt.core.conf.AXTParameters;
import dk.bording.axt.core.conf.AXTDocument;

/**
 * Formats exception message as plain text.
 */
public class TextErrorHandler implements AXTErrorHandler {

/**
 * Formats exception message as plain text with the prefix "ERROR: "
 */
  public void print(OutputStream out,
                     AXTDocument doc,
                    Throwable e , AXTParameters p) throws IOException {
    out.write(("ERROR: "+e.getMessage()).getBytes());
  };
}
```

The output from the above error handler might look like this:

```
ERROR: Illegal value ‘ordr.xsl’ for the parameter: xsl-script message: no such script
```

Sharing and exposing global data across multiple filter instances (AXT v.2.0.0)

AXT provides a simple global in-memory database abstraction that may be used to collect statistical information or share information across multiple filter instances. Since objects of the database must adhere to a well-defined interface, the AXT core can automatically expose the data in the AGETOR control center.

When a filter is used in AXT, the actual instance is taken from an object pool. This means that local variable doesn't survive across multiple invocations. One way to maintain such persistent information would be through the use of static variables. This is a valid approach, but it does not support external exposure of the values. When this is a wish you should use a global singleton variable instance in your filter. To do this you may call the `getInfoObjectInstance(key)` method of your filter. This method will request an object from the AXT Information database known under the given key. If no object exists (the case on the very first invocation), the core will make a call-back to your filter for the method `createInfoObjectInstance(scope, key)`. Thus you filter must provide this method, which is responsible of creating the information object. All information objects must implement the `InfoObject` interface. This interface simply requires your information object to implement a method that returns relevant object data as an `Info` (ADK utility class) object. Once the core has obtained an instance for the given key, this instance will be held as a singleton instance and returned on all subsequent requests for the key.

The example below shows excerpts from an `HTTPFilter` providing a global `HTTPStatistics` information object. When the filter has performed an HTTP request, timing information is submitted to the information object, which holds a map of summed data for each requested URL. To submit the information the filter must first obtain the instance from the database using the getInfoObjectInstance method. Since the InfoObject must provide an Info interface to the held data, a number of formatting routines are available to serialize the data.

```
// this is the singleton variable key.
private static final String STAT_KEY = "HTTPFilter.stat";


// somewhere in filter execution:
.
.
// the hasInfoDB() method returns true if the filter has a valid reference to
// the global information database. This may not be the case if the filter is run
// under an older version of the AXT core. Thus the check is to avoid run time errors.
// Since getInfoObjectInstance returns an InfoObject, we must typecast it to the
// type that we know it has (because we created it under the STAT_KEY).
if (hasInfoDB()) {
             HTTPStatistics hs = (HTTPStatistics) getInfoObjectInstance(STAT_KEY);
             hs.submitInvocationInfo(target, i);
             System.out.print(hs.asInfo().print(TextMessage.STATUS_TEXT).getText());
}
.
.

/**
** Method that must be implemented when requesting an information object of the type.
* Normally the scope and key values may be ignored when the filter only refers one object
* instance. If more instances are used with different keys, the filter must also be able to
* create the objects for the keys. The default scope is defined as global. The scope
* simply serves to discriminate objects with same key in different scope. This could be
* useful when inheritance is used for filters. When the extended version of the filter is
* used another scope is specified (implementing the getVariableScope() method) resulting in
* different object instances for child/parent filter invocations.
*/
public InfoObject createInfoObjectInstance(String scope, String key) throws InfoDBException {
             return new HTTPStatistics();
}

/**
** InfoObject implementation.
```

```
**
*/
private class HTTPStatistics extends AbstractInfoObject {

        private Map stats = Collections.synchronizedMap(new HashMap());

        /**
         * submits a sub-sum of invocations for a target key. The data are
         * added to the info held for the target key.
         * @param targetKey
         * @param i
         */
        public void submitInvocationInfo(String targetKey, InvocationInfo i) {
                InvocationInfo ii;
                ii = getInvocationInfo(targetKey);
                if (ii != null) {
                        ii.add(i);
                } else {
                        i.target = targetKey;
                        stats.put(targetKey, i);
                }
        }

        public InvocationInfo getInvocationInfo(String targetKey) {
                InvocationInfo i;
                i = (InvocationInfo) stats.get(targetKey);
                return i;
        }

        public Info asInfo() {
                InfoTable t = new InfoTable("HTTPFilterStatistics", new String[] {
        "url", "invocations", "avg.ms.", "min.", "max", "prexsl", "snd", "rcvxsl" });
                Iterator i = stats.values().iterator();
                synchronized (stats) {
                        while (i.hasNext()) {
                                InvocationInfo    ii    =    (InvocationInfo)
i.next();
                                int row = t.addRow();
                                t.putString(row, 0, ii.target);
                                t.putLong(row, 1, ii.req);
                                t.putFloat(row,   2,   ((float)   ii.tot)   /
ii.req);
                                t.putLong(row, 3, ii.min);
                                t.putLong(row, 4, ii.max);
                                t.putLong(row, 5, ii.ixsl);
                                t.putLong(row, 6, ii.totrcv / ii.req);
                                t.putLong(row, 7, ii.totsnd / ii.req);
                        }
                }
                return t;
        }
}
```

NOTE: Note that access to the singleton information object should be serialized and that this is the responsibility of the implementer.

The AXT core has access to the information objects defined by filters. The information may be browsed through the AGETOR Control Center (ACC). The screen dump below shows how the HTTPFilter information automatically appears in the ACC under the Status/Other tap:

Advanced concepts

Method invocation and passing of object definitions to filters

Some filters may have a method invocation as their primary function. If such a filter is to be general it needs to know the name of the method to invoke, parameter names and parameter values. The input stream may contain this information to a greater or lesser extend. In the lack of complete information the filter may obtain the lacking information in other ways. Some information could be passed as simple key-value pairs as described earlier, but this approach is often too simple to describe parameter attributes such as type, name etc. Alternatively the filter may lookup the information in a filter specific definition file. If so a reference to the file may be given as a parameter. The AXT core supports a third solution, namely to describe the objects expected in the input stream directly in the configuration file using XML-tags. The information provided here is made available to the executing filter through the `getObjectDefs` method in the `AbstractAXTFilter` class. This information merely describes

- the name of the parameter
- the type(class) of the parameter
- if the parameter is an in-parameter
- if the parameter is an out-parameter

The tags below illustrates the configuration syntax. Besides the known parameter tags, object tags are used. The filter below is a filter that expect an XML input stream with tags corresponding to the fields of the in-objects. The in-objects are instantiated with the values marked up in the XML input and passed to the method which is named by a parameter. What is made available to the filter is simply the four attributes of the object tag.

```
<filter class="dk.bording.axt.tc.invocation.InsideMethodInvocationFilter" >
  <object name="order" class="dk.bording.idl.xmlsample.OrderExt" in="true" out="false"
/>
  <object name="orderNo" class="dk.bording.inside.orb.IntHolder" in="false" out="true"
/>
  <param name="_return" value="true" />
  <param name="deserializer" value="dk.bording.axt.tc.invocation.AXTXml2ObjectParser" />
  <param name="serializer" value="dk.bording.axt.tc.invocation.AXTObject2XmlParser" />
  <param name="interface" value="dk.bording.idl.xmlsample.XmlSample" />
  <param name="broker" value="127.0.0.1" />
  <param name="port" value="20002" />
  <param name="env" value="axt" />
  <param name="method" value="createDemoOrder" />
</filter>
```