Writing a Cocoon 2 generator (2.1 legacy document)

Table of contents

Table of contents

Warning:
This document was copied as is from the Cocoon 2.1 documentation, but has not yet been fully reviewed or moved to its new home.

1. Preface

This document is written in context of the thesis of Carolien Coenen and Erwin Hermans at the department of Computer Science at the Katholieke Universiteit Leuven (Department of Computer Science).

At some point in our thesis the need for extending the functionality of Cocoon 2 became imminent. At that moment, we worked with an application that generated XML documents from Java source files (a sort of JavaDoc tool). We wrote some XSL stylesheets for transforming these documents into HTML, so they could easily be served by Cocoon 2. But **every time** the documentation comments changed in the Java source files, these XML documents **required (re)generation**. The ultimate goal of this project was to be able to do all the intermediate steps of the document generation in memory. This way the HTML documents would change whenever the Java source files were modified, **without the need to regenerate the XML documents**.

To reach this goal, we made some modifications in our program. Our output documents were built in memory using the JDOM API (JDOM.org). At the time of writing this API supported 3 output methods, which are **DOM** (Document Object Model (DOM) Level 1 Specification, Version 1.0, W3C Recommendation 1 October 1998), **SAX** (The official SAX website) and **XML** (Extensible Markup Language (XML) 1.0 (Second Edition), W3C Recommendation 6 October 2000). The DOM output method outputs a DOM tree from an existing JDOM tree. The XML output method outputs an XML file to an OutputStream or a Writer. But the most interesting of these three outputmethods is SAX. The generators that come with Cocoon 2 don't supply a method (or at least we haven't found any) to take the SAX output of an arbitrary Java program and feed it into the transformers (in our case). That's why we wanted to write a generator that would be able to supply SAX events that were outputted by our (or an arbitrary) Java program and to feed this SAX events into the transformer pipeline. This generator would be responsible for starting the Java program and delegating the received SAX events to the Cocoon 2 transformation pipeline in some way.

To accomplish this task we decided to write our own generator and this documentation in parallel, as there was no documentation available, except for the following phrase on Apache's Cocoon 2 website (Cocoon2 van The Apache XML project): "A Generator generates XML content as SAX events and initializes the pipeline processing." Apache's Cocoon 2 website also enumerates the available Generators, which package contains them, which interfaces and helper classes there are in that package and which of those should be implemented/extended. But at that point, the documentation stopped. So the only way to actually be able to write a generator ourselves, was by studying the Cocoon 2 code (more specific the Generators) and figure out for ourselves how to write a generator. Because we want to make a contribution to the Cocoon 2 community, we have written this document. We think our experiences may come in handy for developers who are also thinking about extending Cocoon 2 with their own generators, .

The writing of this generator and all the testing of our code were performed with the following configuration:

- Compaq/Digital Personal Workstation 500a (Alpha 500MHz processor)
- Redhat Linux 7.1
- Compaq JDK 1.3.1-1 (Classic VM (build 1.3.1-1, native threads, jit))
- Jakarta Tomcat 3.2.3
- Cocoon 2.0.2-dev

2. Planning

Here you'll find a list of consequent steps that we expect will be necessary to write our own Generator. It is of course possible that in this first draft of our planning we have forgotten a few steps or that some steps actually form one step.

- Find out which classes should be extended and which interfaces implemented.
- Examine these superclasses and interfaces and find which methods should be actually implemented and what is excepted from these methods.
- Write a first Generator as a **proof of concept** to see if our conlusions in relation to the methods are correct, and let this Generator generate some SAX events (hard coded in the Generator) to 'feed' to Cocoon2.
- Find out how to feed the SAXOutput of a JDOM document (hardcoded in the Generator) to Cocoon2.
- Modify our program so it can generate SAXOutput when this is requested.
- Feed the SAXOutput from our program to the Generator (we think it shall require running our program from the Generator). This SAXOutput shall then be passed to Cocoon 2. Again, every call to our program and the parameters will be hardcoded in our Generator.
- Find out how we can use the sitemap to pass parameter values to our Generator (= examing how the sitemap is transformed into Java code and how we can read the parameter values from this code). This will no longer be hardcoded then.
- Examine how we can define in the most general way the syntax from the sitemap, so that it is possible to define which class should be called to generate SAXOutput and with which values and methods this class should be called. This also implies that we must study if this is possible in Java.
- This will be tested with our program and our Generator (hopefully we'll get this far) will become heavily parameterized.
- Modify our program once again, so that it satisfies our final needs.
- Submit our generator, and this document to the Cocoon 2 project.

3. Our planning, step by step

3.1. Classes and interfaces to be extended/implemented

In this section, we'll discuss which classes should/can be extended to implement our own Generator. Also, we'll take a closer look at these classes to see which functionality they provide and (try to) discuss their functionality. Let it be clear that it is **not required** to extend one of the existing (abstract) generator classes. But by doing so, it'll make your work a great deal easier.

The second part of this section will discuss the interface(s) that have to be implemented to write our own generator. We'll look at what the methods that are defined in the interface should do. There is one interface that has to be implemented if you write your own generator: the **org.apache.cocoon.generation.Generator** interface.

3.1.1. Classes

According to the Cocoon 2 website at the time of writing (21st november 2001) there are four helper classes in the org.apache.cocoon.generation package that can be extended. These four are (they will be discussed later):

- AbstractGenerator
- AbstractServerPage
- ServiceableGenerator

- ServletGenerator

Java only supports single inheritance, so you'll have to choose one of these for your Generator. We want to use the AbstractGenerator (in our first attempt), but to help the reader of this document in making a well motivated choice, we'll discuss each of these options briefly as to what specific functionality they provide.

There is a hierarchy between these classes, namely:

- AbstractGenerator
- ServiceableGenerator extends AbstractGenerator
- ServletGenerator extends ServiceableGenerator
- AbstractServerPage extends ServletGenerator

So the choice of which class to extend will depends mostly on which is the level of abstraction required by your generator.

3.1.1.1. AbstractGenerator

Extend this one for easier building of your own Generator

This **abstract class** extends the class **org.apache.cocoon.xml.AbstractXMLProducer** and implements the interface **org.apache.cocoon.generation.Generator**.

The **Generator** interface extends the interfaces **org.apache.cocoon.xml.XMLProducer** and **org.apache.cocoon.sitemap.SitemapModelComponent**.

The interface **XMLProducer** is a top-level interface.

The interface **SitemapModelComponent** extends the interface **org.apache.avalon.framework.component.Component**, which in turn is a top-level interface.

The abstract class **AbstractXMLProducer** extends the abstract class **org.apache.avalon.framework.logger.AbstractLoggable** and implements the interfaces **org.apache.cocoon.xml.XMLProducer** and **org.apache.avalon.excalibur.pool.Recyclable**.

AbstractLoggable is a top-level abstract class to provide logging capabilities. This is deprecated and **AbstractLogEnabled** should be used instead. AbstractLoggable implements the interface **org.apache.avalon.framework.logger.Loggable**, but as mentioned in the API docs **org.apache.avalon.framework.logger.LogEnabled** should be used instead.

The interface **Recyclable** extends the interface **org.apache.avalon.excalibur.pool.Poolable**, which is a top-level interface.

The following methods are defined for **AbstractGenerator**, some of which already have an implementation:

- From **org.apache.avalon.excalibur.pool.Poolable**:
  - None

  This interface is implemented bij components if it is reasonable to Pool the object. It marks the component Poolable.
- From **org.apache.avalon.excalibur.pool.Recyclable**:
  - public void recycle(): this method should be implemented to remove all costly resources in the object. These resources can be object references, database connections, threads, etc. What is categorised as "costly" resources is determined on a case by case analysis.
- From **org.apache.avalon.framework.logger.Loggable** (Deprecated):

    - public void setLogger (org.apache.log.Logger logger): provide component with a logger.

  Interface can be implemented by Components that need to log.
- From **org.apache.avalon.framework.logger.AbstractLoggable** (Deprecated):
  - protected org.apache.log.Logger getLogger(): Helper method to allow sub-classes to acquire logger (implemented).
  - protected void setupLogger(java.lang.Object component): Helper method to setup other components with the same logger (implemented).
  - protected void setupLogger(java.lang.Object component, org.apache.log.Logger logger): Helper method to setup other components with logger (implemented).
  - protected void setupLogger(java.lang.Object component, java.lang.String subCategory): Helper method to setup other components with logger (implemented).

  This is a utility class to allow the construction of easy components that will perform logging.
- From **org.apache.cocoon.xml.XMLProducer**
  - void setConsumer(XMLConsumer xmlconsumer): set the XMLConsumer that will receive XML data. The XMLConsumer interface extends **org.xml.sax.ContentHandler** and **org.xml.sax.ext.LexicalHandler**.

  This interface identifies classes that produce XML data, sending SAX events to the configured XMLConsumer.
- From **org.apache.cocoon.xml.AbstractXMLProducer**:
  - void setConsumer(XMLConsumer xmlconsumer): set the XMLConsumer that will receive XML data (implemented).
  - public void setContentHandler(ContentHandler consumer): Set the ContentHandler that will receive XML data (implemented).
  - public void setLexicalHandler(LexicalHandler handler): Set the LexicalHandler that will receive XML data (implemented).
  - public void recycle(): Recycle the producer by removing references (implemented).
- From **org.apache.cocoon.generation.Generator**:
  - void generate(): generate the SAX events to initialize a pipeline.
- From **org.apache.cocoon.sitemap.SitemapModelComponent**:
  - void setup(SourceResolver resolver, Map objectmodel, String src, Parameters par): set the SourceResolver, objectmodel Map, the source and sitemap Parameters used to process the request.
- From **AbstractGenerator** itself:
  - void setup(SourceResolver resolver, Map objectmodel, String src, Parameters par): set the SourceResolver, objectmodel Map, the source and sitemap Parameters used to process the request (implemented).
  - public void recycle(): Recycle the generator by removing references (override implementation).

If we carefully analyse this list, we see that the only method left unimplemented is the **generate()** method. So if we extend the **AbstractGenerator** class to make our own generator, the only method we'll have to implement is the **generate()** method.

The following variables are defined in the different interfaces and classes:

- From **org.apache.avalon.excalibur.pool.Poolable**:
  - None
- From **org.apache.avalon.excalibur.pool.Recyclable**:
  - None

- From **org.apache.avalon.framework.logger.AbstractLoggable** (Deprecated):
  - None
- From **org.apache.avalon.framework.logger.AbstractLoggable** (Deprecated):
  - private org.apache.log.Logger m_logger: the base logger instance. Provides component with a logger. The **getLogger()** method should be used to aquire the logger.
- From **org.apache.cocoon.xml.XMLProducer**
  - None
- From **org.apache.cocoon.xml.AbstractXMLProducer**:
  - protected XMLConsumer xmlConsumer: The XMLConsumer receiving SAX events. Can be accessed via **super.xmlConsumer**.
  - protected ContentHandler contentHandler: The ContentHandler receiving SAX events. Can be accessed via **super.ContentHandler**.
  - protected LexicalHandler lexicalHandler: The LexicalHandler receiving SAX events. Can be accessed via **super.LexicalHandler**.

  We here access these variables via the **super.** qualifier, this is only done for clarity. They can be equally well accessed via using the **this.** qualifier (or omitting this). For reasons of clarity, if we access such a variable in our code, we will use the **super.** qualifier, although when summing up all the variables we will use **this.**.
- From **org.apache.cocoon.generation.Generator**:
  - String ROLE = "org.apache.cocoon.generation.Generator"
- From **org.apache.cocoon.sitemap.SitemapModelComponent**:
  - None
- From **org.apache.avalon.framework.component.Component**:
  - None
- From **AbstractGenerator** itself:
  - protected SourceResolver resolver = null: The current SourceResolver. Can be accessed via **this.resolver**.
  - protected Map objectModel = null: The current Map objectModel. Can be accessed via **this.objectModel**.
  - protected Parameters parameters = null: The current Parameters. Can be accessed via **this.parameters**.
  - protected String source = null: The source URI associated with the request or **null**. Can be accessed via **this.source**.

This gives us a list of variables that we can use throughout our own generator.

3.1.1.2. ServiceableGenerator

Can be used as base class if you want your Generator to be an Avalon Serviceable

This **abstract class** extends **org.apache.cocoon.generation.AbstractGenerator** and extends the interfaces **org.apache.avalon.framework.service.Serviceable** and **org.apache.avalon.framework.activity.Disposable**.

In addition to all the methods introduced in the **AbstractGenerator** class, these two interfaces introduce som new methods:

- From **org.apache.avalon.framework.service.Serviceable**:
  - public void service(ServiceManager serviceManager): Pass the ServiceManager to the

Serviceable. The Serviceable implementation should use the specified ServiceManager to acquire the components it needs for execution.

- From **org.apache.avalon.framework.activity.Disposable**:
  - public void dispose(): The dispose operation is called at the end of a components lifecycle. Components use this method to release and destroy any resources that the Component owns.

  The Disposable interface is used when components need to deallocate and dispose resources prior to their destruction.
- From **ServiceableGenerator** itself:
  - public void service(ServiceManager serviceManager): Pass the ServiceManager to the Serviceable. The Serviceable implementation should use the specified ServiceManager to acquire the components it needs for execution. (implemented)
  - public void dispose(): The dispose operation is called at the end of a components lifecycle. Components use this method to release and destroy any resources that the Component owns. (implemented - implementation sets the ServiceManager to null)

We see that this class provides a default implementation of the methods introduced by the two new interfaces. The only method that needs to be implemented remains the **generate()** method, if we are satisfied with the default implementations.

Besides these methods, also a new variable is introduced:

- From **ServiceableGenerator** itself:
  - protected ServiceManager manager = null: the service manager instance, can be accessed via **this.manager**.

3.1.1.3. ServletGenerator

If you want to generate servlets. This is the base class for the ServerPagesGenerator

The **ServletGenerator** extends **ServiceableGenerator**.

We are not giving a more elaborate description of this component at this time. We have not experimented with this component and we would not like to risk making any wrong assumptions.

3.1.1.4. AbstractServerPage

[FIXME: This seems to be intended as basis for the ServerPagesGenerator, but it seems to be obsolete now?]

The **AbstractServerPage** extends **ServletGenerator** and implements the **org.apache.cocoon.caching.Cacheable** and **org.apache.cocoon.components.language.generator.CompiledComponent** interfaces.

We are not giving a more elaborate description of this component at this time. We have not experimented with this component and we would not like to risk making any wrong assumptions.

3.1.2. Interface(s)

Following the somewhat little pointers on develop-part of the Cocoon-website Extending Apache Cocoon van The Apache XML Project), we find that the only interface that should be implemented is the Generator interface in the **org.apache.cocoon.generation** package, so we will try to give an overview as complete as possible for now.

3.1.2.1. Generator

As mentioned earlier this public interface is situated in the **org.apache.cocoon.generation** package. This interface in its turn extends the **org.apache.cocoon.xml.XMLProducer** and the **org.apache.cocoon.sitemap.SitemapModelComponent** interfaces.

The interface **XMLProducer** is a top-level interface.

The interface **SitemapModelComponent** extends the interface **org.apache.avalon.framework.component.Component**, which in turn is a top-level interface.

Analyzing these interfaces tells us that the following methods should be implemented when implementing the **Generator** interface:

- From **org.apache.cocoon.xml.XMLProducer**
  - void setConsumer(XMLConsumer xmlconsumer): set the XMLConsumer that will receive XML data. The XMLConsumer interface extends **org.xml.sax.ContentHandler** and **org.xml.sax.ext.LexicalHandler**.

  This interface identifies classes that produce XML data, sending SAX events to the configured XMLConsumer.
- From **org.apache.cocoon.sitemap.SitemapModelComponent**:
  - void setup(SourceResolver resolver, Map objectmodel, String src, Parameters par): set the SourceResolver, objectmodel Map, the source and sitemap Parameters used to process the request.
- From **org.apache.cocoon.generation.Generator** itself:
  - void generate(): generate the SAX events to initialize a pipeline.

We decided that the easiest way of writing a custom generator was to extend the **AbstractGenerator** class. The only method required to implement was the **generate** method, for now, we would settle with the provided default implementations of the other methods.

3.2. Writing a test generator

After making these decisions, and looking at the implementations of the classes, we could begin the implementation keeping in mind the following:

- We have to provide SAX events to the XMLConsumer, that is set via the **setConsumer** method.
- We can access the XMLConsumer via **super.xmlConsumer** (analysis of code of **FileGenerator** and definition of the **xmlConsumer** variable as **protected** in the **AbstractXMLProducer** class). The **super.** modifier is only used for clarity, since it can also be accessed via **this.xmlConsumer.**
- We will extend the **org.apache.cocoon.generation.AbstractGenerator** class.
- We have to implement the **generate** method which purpose is to produce SAX events and feed them to the XMLConsumer.

3.2.1. The code of our first generator

As a first test we decided to parse a string containing the following XML content and feed the SAX events to the XMLConsumer:

```
<doc>My first Cocoon 2 generator!</doc>
```

First, we will give our code and then we will explain what it does and why we made these choices.

```
package test; import java.io.IOException; import java.io.StringReader; import
```

```
org.xml.sax.XMLReader; import org.xml.sax.InputSource; import org.xml.sax.SAXException;
import org.xml.sax.XMLReaderFactory; import org.apache.cocoon.ProcessingException;
import org.apache.cocoon.generation.AbstractGenerator; public class MyGenerator extends
AbstractGenerator { public void generate () throws IOException, SAXException,
ProcessingException { String message = "<doc>My first Cocoon 2 generator!</doc>";
XMLReader xmlreader = XMLReaderFactory.createXMLReader();
xmlreader.setContentHandler(super.xmlConsumer); InputSource source = new
InputSource(new StringReader(message)); xmlreader.parse(source); } }
```

First of all, in our working directory (may be any directory) we made a directory "test" and in that directory we created the Java source file **MyGenerator.java**. We also decided to put this class in a package and named that package **test**. This can be easily changed afterwards.

The obvious **import** statements are those to import the **AbstractGenerator** class and those to import the exceptions thrown by the **generate** method. The other import statements serve to parsing our string and generating SAX events.

The code itself is pretty straightforward. We have our class definition containing one method definition. First of all, in the **generate** method, we define the variable **message** containing the XML content we want to generate SAX events for.

```
XMLReader xmlreader = XMLReaderFactory.createXMLReader();
```

Here we make a new **XMLReader** via the **XMLReaderFactory**. Since XMLReader is an interface, the XMLReaderFactory has to provide us with a class that implements the XMLReader interface, commonly known as a **SAXParser**. Therefore the XMLReaderFactory uses the system variable **org.xml.sax.driver** to determine which class to instantiate to provide us with an XMLReader. An example of how this is done is provided after we have discussed the rest of the code.

```
xmlreader.setContentHandler(super.xmlConsumer);
```

With this line of code, we tell the XMLReader which object will receive the SAX events that will be generated when parsing. You can see that we use **super.xmlConsumer** to receive the SAX events.

```
InputSource source = new InputSource(new StringReader(message));
xmlreader.parse(source);
```

With the second line we tell the XMLReader to start parsing the source, provided as argument of the **parse** method. This parse method can only be supplied with an **org.xml.sax.InputSource** argument or a **String** that represents a system identifier or URI. To parse our string we must encapsulate it in an InputSource object. Since the InputSource class can not be passed an XML document that is contained in a string, we first must encapsulate our string into another object, which we then pass to an InputSource object. In this example we have chosen for a **StringReader**. A StringReader can be given as argument when constructing an InputSource object and a StringReader can be given a String object as argument for its construction. This way we succeed in parsing our string.

The next step is compiling our newly written class. We give here an overview of the our work environment and show how we compiled this Java-file. All the commands from this example were carried out on a PC running Linux, but with respect to a few minor modifications, these commands will also work on a PC running Windows. The commands were carried out in the directory "/home/erwin/cocoon2/generator/". This directory has three subdirectories:

- "test/": directory containing the source files
  - **MyGenerator.java**: source file for our generator
- "jar/": directory containing the necessary jar (Java Archive) files
  - **xerces.jar**: Xerces has an implementation for the XMLReader interface which we use
  - **cocoon.jar**: contains the classes from Cocoon 2.0.2-dev, needed to extend AbstractGenerator.

This is in fact a symbolic link to $TOMCAT_HOME/webapps/cocoon/WEB-INF/lib/cocoon-2.0.2.jar. Under Windows you will have to copy this file or point directly to this file.

- "compiled/": outputdirectory for javac. The compiled files will end up in this directory
  - **test/MyGenerator.class**: after compiling, we will have this file here

```
javac -classpath .:jar/cocoon.jar:jar/xerces.jar \ -d compiled test/MyGenerator.java
```

Now we have our compiled class, we can make the big step of putting it to work. To make sure there were no errors in our code, we tested our code by using another class as the ContentHandler of our generator. After these tests were completed (without errors), we tried to deploy our generator from within Cocoon 2.

3.2.2. Deploying MyGenerator

The next step is deploying our custom written generator. First of all we stopped the Tomcat engine (and thus Cocoon 2). We also emptied the **work** directory, located at "$TOMCAT_HOME/work/". Experience learned that this is something you have to do every time you want to try something like this with Cocoon 2.

For the next step, we changed the main sitemap to be able to use or generator in the following way:

Under the **map:generators** element, we added the following:

```
<map:generator name="mygenerator" src="test.MyGenerator"/>
```

Under the **map:pipelines** element, we added the following:

```
<map:pipeline> <map:match pattern="mygenerator.xml"> <map:generate
type="mygenerator"/> <map:serialize type="xml"/> </map:match> <map:handle-errors>
<map:transform src="stylesheets/system/error2html.xsl"/> <map:serialize
status-code="500"/> </map:handle-errors> </map:pipeline>
```

If the page **mygenerator.xml** is requested, we tell Cocoon 2 to use our generator, which we have named **mygenerator**. We do not define the **src** attribuut, since we do not use it in our generator. Once we get the content, we serialize it as xml, so we can check if the input matches the output. In the event that an error occurs, we use one of the stylesheets of Cocoon 2 or another pipeline to signal the error to the user.

After the changes to the sitemap, we added the directory "/home/erwin/cocoon2/generator/" to the classpath. After these changes, we restarted Tomcat and tried to access the page "http://localhost:8080/cocoon/mygenerator.xml". After waiting a while, we received a fatal error. Inspection of the log-files (which is something you should always do when receiving an error that is not so clear) showed that the following exception was the cause of that fatal error:

```
ERROR (2002-03-27) 23:21.40:190 [sitemap] (/cocoon/) Thread-23/Handler: Error compiling
sitemap java.lang.NoClassDefFoundError: org/apache/cocoon/generation/AbstractGenerator
at java.lang.ClassLoader.defineClass0(Native Method) at java.lang.ClassLoader.defineClass
(ClassLoader.java, Compiled Code) at java.security.SecureClassLoader.defineClass
(SecureClassLoader.java, Compiled Code) at java.net.URLClassLoader.defineClass
(URLClassLoader.java, Compiled Code) at java.net.URLClassLoader.access$100
(URLClassLoader.java, Compiled Code) at java.net.URLClassLoader$1.run
(URLClassLoader.java, Compiled Code) at java.security.AccessController.doPrivileged
(Native Method) at java.net.URLClassLoader.findClass (URLClassLoader.java, Compiled
Code) at java.lang.ClassLoader.loadClass (ClassLoader.java, Compiled Code) at
sun.misc.Launcher$AppClassLoader.loadClass (Launcher.java, Compiled Code) at
```

java.lang.ClassLoader.loadClass (ClassLoader.java, Compiled Code) at org.apache.tomcat.loader.AdaptiveClassLoader.loadClass (AdaptiveClassLoader.java, Compiled Code) at java.lang.ClassLoader.loadClass (ClassLoader.java, Compiled Code) at java.lang.ClassLoader.loadClass (ClassLoader.java, Compiled Code) at org.apache.cocoon.util.ClassUtils.loadClass (ClassUtils.java, Compiled Code) at org.apache.cocoon.sitemap.AbstractSitemap.load_component (AbstractSitemap.java, Compiled Code) at org.apache.cocoon.www.sitemap_xmap $Configurer.configGenerators(sitemap_xmap.java, Compiled Code) at org.apache.cocoon.www.sitemap_xmap.configure (sitemap_xmap.java, Compiled Code) at org.apache.avalon.excalibur.component. DefaultComponentFactory.newInstance (DefaultComponentFactory.java, Compiled Code) at org.apache.avalon.excalibur.component. ThreadSafeComponentHandler.initialize (ThreadSafeComponentHandler.java, Compiled Code) at org.apache.cocoon.components.language.generator. GeneratorSelector.addGenerator (GeneratorSelector.java, Compiled Code) at org.apache.cocoon.components.language.generator. ProgramGeneratorImpl.addCompiledComponent (ProgramGeneratorImpl.java, Compiled Code) at org.apache.cocoon.components.language.generator. ProgramGeneratorImpl.generateResource (ProgramGeneratorImpl.java, Compiled Code) at org.apache.cocoon.components.language.generator. ProgramGeneratorImpl.createResource (ProgramGeneratorImpl.java, Compiled Code) at org.apache.cocoon.components.language.generator. ProgramGeneratorImpl.load (ProgramGeneratorImpl.java, Compiled Code) at org.apache.cocoon.sitemap.Handler.run (Handler.java, Compiled Code) at java.lang.Thread.run(Thread.java:484)

Puzzled by this error, we mailed to the cocoon-users mailinglist (users@cocoon.apache.org) and explained our situation. The answer we received was to put our generator in the "$TOMCAT_HOME/webapps/cocoon/WEB-INF/classes/". We stopped Tomcat, emptied the work-directory, removed the directory "/home/erwin/cocoon2/generator/" from the classpath and made a directory "test/" under the "$TOMCAT_HOME/webapps/cocoon/WEB-INF/classes/" and placed **MyGenerator.class** in that directory. We then restarted Tomcat and once again tried to access "http://localhost:8080/cocoon/mygenerator.xml". But after making that request in our browser, we got a message from the browser saying that the server could not be reached. Looking at the xterm from which we started Tomcat, we saw the following error:

IGSEGV 11* segmentation violation si_signo [11]: SIGSEGV 11* segmentation violation si_errno [0]: Success si_code [128]: unknown siginfo sc_pc: 0x20010164f08, r26: 0x200001e19e0 thread pid: 26157 stackpointer=0x3fffc5f69b8 Full thread dump Classic VM (1.3.1-1, native threads): ... ...

Removing our class (and commenting out our changes in the sitemap for safety) would resolve the problem, but then we can't use our generator.

Somewhere on the Web we had read a mail from someone who was also having **NoClassDefFoundError**s that he was able to solve by unzipping all the jar-files (a jar is basically a zip file containing the compiled classes) from "$TOMCAT_HOME/webapps/cocoon/WEB-INF/lib/" into the "$TOMCAT_HOME/webapps/cocoon/WEB-INF/classes/" directory. We stopped Tomcat, emptied the work-directory and started Tomcat again.

After restarting Tomcat we had our hopes up that this time it would work. We also started our browser and tried to access "http://localhost:8080/cocoon/mygenerator.xml", again. After waiting a while (Cocoon 2 had to recompile its sitemap and some other components) we got the see our XML file. Cocoon 2 produced the following XML document:

<?xml version="1.0" encoding="UTF-8"?> <doc>My first Cocoon 2 generator!</doc>

So, after a bit of struggling, we finally succeeded in deploying our own generator.

3.2.3. Considerations afterwards

After seeing our example and having some experience with Cocoon 2 one might ask why we reinvented the wheel by instantiating a parser and not using the one provided by Cocoon 2. It is evident that a start of a pipeline is a generator that fires SAX events, there must be a SAXParser available throughout Cocoon 2 that can be easily accessed. This is in fact the case. There are a number of reasons why we had not chosen that approach the first time around:

- Limited knowledge of the whole underlying architecture, not really enhanced by the documentation.
- We wanted to keep the time-to-test as short as possible, so we didn't spend time finding this information in the source in the first phase.
- We didn't see any other possibility of testing our code before we tried to integrate it with the Cocoon 2 project.

We would still like to point the reader to an alternative solution, i.e. the solution that is used throughout Cocoon 2. We will give the code fragments here and we will then explain what it does.

```
... import org.apache.excalibur.xml.sax.SAXParser; ... SAXParser parser = null; try { parser =
(SAXParser)this.manager.lookup(SAXParser.ROLE);
parser.parse(this.getInputSource(),handler); } catch (SAXException e) { // Preserve original
exception throw e; } catch (Exception e) { throw new ProcessingException("Exception during
processing of " + this.getSystemId(),e); } finally { if (parser != null) {
this.manager.release(parser); } } ...
```

An extra **import** statement is added. The **SAXParser** interface of the Avalon/Excalibur project (Avalon/Excalibur van The Jakarta Project) defines the following method:

- void parse(InputSource in, ContentHandler consumer): the implementation of this method should parse the **InputSource** and send the SAX events to the **consumer**. The consumer can be an **XMLConsumer** or an object that implements **LexicalHandler** as well.

This interface defines a variable **ROLE** of the type String that is given the value **org.apache.excalibur.xml.sax.SAXParser**. This variable is used to ask the **ServiceManager**, which is accessed by **this.manager**, to **lookup** a **Component** that has that role. The returned Component is then casted to a **SAXParser** type. We can then apply the parse method to any **org.xml.sax.InputSource** object and to an object that implements the **ContentHandler** interface. Finally, we have to tell the ServiceManager that we are finished using the parser. This allows the ServiceManager to handle the End-Of-Life Lifecycle events associated with this Component.

NOTE: if you want to use this method to obtain a parser, it would be better to extend the **ServiceableGenerator** class, instead of the **AbstractGenerator** class. The ServiceableGenerator is defined to make use of a **ServiceManager**, while this is not the case for the **AbstractGenerator** class. You should envisage the given code as part of a class that extends the **ServiceableGenerator** class or one of its children.

3.3. Going the distance

We have succeeded in implementing a first test to find out how everything works, but a generator that only sends a fixed string to Cocoon 2 is not that interesting. Since we have written an application that can serve XML documents contained in a String object (using JDOM (JDOM.org)), we want to be able to retrieve these documents through our browser, which sends this request to Cocoon 2. Cocoon 2 then

fires up our generator to retrieve the requested XML document and can start the pipeline for processing that document.

Since we had experimented with Java RMI in one of our courses, we decided to try a setup where our generator was a client for the document server and the communication would happen via RMI. For this section, we will first look at setting up the server, next we will look at accessing the server from within MyGenerator and finally we will put it all together. If we get this to work, we then can ponder about looking up parameters defined in the sitemap to use in MyGenerator. We used (Getting Started Using RMI) as a basis for getting started with RMI. If you have never used RMI, we recommend that you read this document to develop a basic understanding of working with RMI.

3.3.1. Setting up a RMI server

After reading the document (Getting Started Using RMI) and having deployed the example, we started writing our own interface, called **Serverfunctions** that defines the methods that should be implemented by a program that wishes to serve as a server for **MyGenerator**. This interface looks like this:

```
package test; import java.rmi.Remote; import java.rmi.RemoteException; public interface ServerFunctions extends Remote { /** This method returns a String, containing a well-formed XML fragment/document. This String contains information about the application implementing this interface. Choosing what information is put into this String is left to the application designer. */ String sayHello () throws RemoteException; /** This method returns a String, containing a well-formed XML fragment/document. To determine the information that should be returned, a systemId is passed to this method. */ String getResource (String systemId) throws RemoteException; }
```

This interface defines two methods that should be implemented. Since these methods can be invoked via RMI we must declare that these methods can throw a RemoteException. These methods should return well-formed XML, as specified.

With interfaces alone we cannot build an application. We also must have a class that implements this interface. The following example demonstrates how this can be implemented. We used JDOM (JDOM.org) for reading in a XML document and converting it to a String.

```
package test; import java.rmi.Naming; import java.rmi.RemoteException; import java.rmi.RMISecurityManager; import java.rmi.server.UnicastRemoteObject; import org.jdom.Document; import org.jdom.JDOMException; import org.jdom.input.SAXBuilder; import org.jdom.output.XMLOutputter; import test.ServerFunctions; public class Server extends UnicastRemoteObject implements ServerFunctions { public Server () throws RemoteException { super(); } public String sayHello () { return "<doc>My First RMI Server!</doc>"; } public String getResource (String systemId) { try { SAXBuilder sb = new SAXBuilder(); Document newdoc = sb.build(systemId); return (new XMLOutputter()).outputString(newdoc); } catch (JDOMException jde) { System.out.println("JDOM error: " + jde.getMessage()); jde.printStackTrace(); // Rethrow the exception so the other // side knows something is wrong throw new RemoteException("JDOMException while processing " + systemId,jde); } } public void main (String args[]) { // Create and install a security manager // For testing purposes only, set this to null System.setSecurityManager(null); try { Server obj = new Server(); // Bind this object instance to the name "MyServer" Naming.rebind("MyServer",obj); System.out.println("MyServer bound in registry"); } catch (Exception e) { System.out.println("Server error: " + e.getMessage()); e.printStackTrace(); } } }
```

We first have the necessary import-statements. This class implements the **ServerFunctions** interface

we defined before. We also extend the **UnicastRemoteObject**. The Java API docs ((Java 2 Platform, SE v1.3 API documentation)) tell us the following about UnicastRemoteObject: "The UnicastRemoteObject class defines a non-replicated remote object whose references are valid only while the server process is alive. Objects that require remote behavior should extend RemoteObject, typically via UnicastRemoteObject." This allows us, by calling the constructor of this superclass, to use the behavior of the UnicastRemoteObject for our RMIServer. This is typically done by calling the **super()** constructor in the constructor of our class.

Next, we have the implementation of the two methods defined in our interface. The **sayHello** method just returns a string representing the following XML fragment:

<doc>My First RMI Server!

We then also implement the **getResource** method. In the body of the try-block we first build a JDOM Document using the given systemId. This means that an XML file, at the given location, is read and a JDOM Document object is created. Next, we use the method **outputString(Document doc)** of the **XMLOutputter** class to convert the JDOM Document to a string. It is this string that is returned to the client. In the event that there may be an error building the document, a **JDOMException** is thrown. If this is the case, we print the info to stdout and rethrow the exception, encapsulated in a RemoteException.

We then only need a **main** method to have a Java application at hand. The first thing we do is disabling the **SecurityManager**. For security reasons, this should only be done only for testing purposes on an isolated system and in production environments. We did this so we could bind this server in the rmiregistry without rewriting any Java policy files. Next, we make a new **Server** object and bind this in the rmiregistry, where it is associated with the name **MyServer**. We end with printing out a line that we have bound this object in the rmiregistry.

3.3.2. Setting up a RMI client

The next step in the process is to implement a Java application that can connect to our RMI server and invoke its methods. Once again, we will first give our code and then explain what it does.

```
package test; import java.rmi.Naming; import java.rmi.RemoteException; import test.ServerFunctions; public class Client { public void main (String args[]) { String message = "blank"; try { // "obj" is the identifier that we'll use to refer // to the remote object that implements the // "ServerFunctions" interface ServerFunctions obj = (ServerFunctions)Naming.lookup("//myhost.com/MyServer"); message = obj.sayHello(); System.out.println(message); message = obj.getResource("index.xml"); System.out.println(message); } catch (Exception e) { System.out.println("Server exception: " + e.getMessage()); e.printStackTrace(); } } }
```

Our client only defines a **main** method. We first initialize the variable, to which we will assign the return value of the **sayHello** method. Next, we try to **lookup** an object that is bound to "//myhost.com/MyServer" (note that myhost.com is a random chosen example). The lookup method returns an object, that is casted to the **ServerFunctions** type. We then invoke the sayHello method on the object and we print this message out. We also invoke the **getResource** method and print the result out. If this succeeds, we know everything works correctly. If an exception occurs, we print out the message from this exception plus its stack trace.

3.3.3. Testing the RMI components

We will first test if the RMI communication works. If it doesn't work there is no point in trying to integrate RMI communication in MyGenerator.Located in the directory

"/home/erwin/cocoon2/generator/", which has the subdirectory "test/" containing our files, we execute the following commands:

```
javac -classpath .:jar/jdom.jar:jar/xerces.jar -d compiled/ test/*.java rmic -classpath
.:jar/jdom.jar:jar/xerces.jar -d compiled/ test.Server rmiregistry &amp;amp; java -classpath
compiled/:jar/jdom.jar:jar/xerces.jar \
-Djava.rmi.server.codebase=http://myhost.com/~erwin/cocoon2/generator/compiled/ \
test.Server MyServer bound in registry
```

If you forget to define the **java.rmi.server.codebase** system property or give it a wrong value, you are most likely to get the following exception:

```
HelloImpl err: RemoteException occurred in server thread; nested exception is:
java.rmi.UnmarshalException: error unmarshalling arguments; nested exception is:
java.lang.ClassNotFoundException: test.Server_Stub java.rmi.ServerException:
RemoteException occurred in server thread; nested exception is:
java.rmi.UnmarshalException: error unmarshalling arguments; nested exception is:
java.lang.ClassNotFoundException: test.Server_Stub java.rmi.UnmarshalException: error
unmarshalling arguments; nested exception is: java.lang.ClassNotFoundException:
test.Server_Stub java.lang.ClassNotFoundException: test.Server_Stub at
sun.rmi.transport.StreamRemoteCall.exceptionReceivedFromServer
(StreamRemoteCall.java, Compiled Code) at
sun.rmi.transport.StreamRemoteCall.executeCall (StreamRemoteCall.java, Compiled Code)
at sun.rmi.server.UnicastRef.invoke(UnicastRef.java, Compiled Code) at
sun.rmi.registry.RegistryImpl_Stub.rebind(Unknown Source) at
java.rmi.Naming.rebind(Naming.java, Compiled Code) at test.Server.main(Server.java,
Compiled Code)
```

We now can start the client to test if everything works. Notice that the resource requested in the code is in fact a relative URI. It is relative to the path from where we started the server application. The file index.xml contains the following information:

```
<?xml version="1.0"?> <document> <title>This is a document</title> <para>This is the first
paragraph.</para> </document>
```

The client is started with the following command:

```
[erwin generator]$ java -classpath compiled/ test.Client
```

This resulted in the following output:

```
<doc>My First RMI Server!</doc> <?xml version="1.0" encoding="UTF-8"?> <document>
<title>This is a document</title> <para>This is the first paragraph.</para> </document>
```

This is exactly the output we expected, except for the encoding attribute. But this is something that is added by JDOM.

NOTE: we would like to conclude this section with a final note about the RMI server application. If you wish to deploy an RMI server application in the real world, you may wish to delete the code that disables the SecurityManager. If no other settings are changed, you may get the following error when starting your server application (depending on the configuration in your **java.policy** file):

```
HelloImpl err: access denied (java.net.SocketPermission 127.0.0.1:1099 connect,resolve)
java.security.AccessControlException: access denied (java.net.SocketPermission
127.0.0.1:1099 connect,resolve) at java.security.AccessControlContext.checkPermission
(AccessControlContext.java, Compiled Code) ... at test.Server.main(Server.java, Compiled
Code)
```

The most likely reason is that the default policy does not permit your server to bind its name in the rmiregistry. You have to change the security policy specified in the "$JAVA_HOME/jre/lib/security/java.policy" file. Since we are no experts in security we cannot give you any advice in this matter, but a general advice in security related matters is that you are better safe then sorry.

3.3.4. Putting the pieces together

We now have been able to setup a generator and use RMI communication, now it is time to integrate these two pieces so we have a fully blown RMIGenerator for Cocoon 2. But before we do that, we will look how we can access the parameters and source that are passed from the sitemap to MyGenerator.

We have seen that the method **setup** is implemented in the **AbstractGenerator** class. One of the arguments of this method is String src. The value of the **src** attribute in the sitemap is passed via this argument and the variable **source** will be assigned this value. If for instance the following is a small part of the sitemap:

```
<map:match pattern="mygenerator.xml"> <map:generate type="mygenerator"
src="example.xml"/> <map:serialize type="xml"/> </map:match>
```

If we request "$FULL_URL_PATH/mygenerator.xml", the value of the **src** attribute will be passed to **MyGenerator** using the setup method. This value, **example.xml** can then be accessed via the **this.source** variable in our code.

As for now, we still have hardcoded in MyGenerator to which RMI server our generator should connect and also which bindname should be looked up. This is not desirable, we wish to have a configurable generator. "Compile once, run many" is maybe the way you could describe this. We wish to pass these values as parameters to the generator. Clearly, these values should be specified in the sitemap. Amongst the elements allowed in the sitemap there is a **parameter** element. If we want to use this element to pass parameters to our generator this element has to appear as a child of the **generate** element. Our sitemap fragment will then look like this:

```
<map:match pattern="mygenerator.xml"> <map:generate type="mygenerator"
src="example.xml"> <map:parameter name="host" value="myhost.com"/> <map:parameter
name="port" value="1099"/> <map:parameter name="bindname" value="MyServer"/>
</map:generate> <map:serialize type="xml"/> </map:match>
```

We define three parameters:

- **host**: tells the generator at which host the RMI server application is running. **REQUIRED**.
- **port**: tells the generator at which port at the remote host the rmiregistry process is running. If no value is specified Java uses the default port (1099). **OPTIONAL**.
- **bindname**: tells the generator which name should be looked up in the remote registry to obtain access to the RMI server object. **REQUIRED**.

We only need these three parameters to define the remote server object. We do not need to specify which methods should be invoked since we demand that a remote server implements the **ServerFunctions** interface. This is something that may be considered in the future.

We now have defined the host, port and bindname parameters, but how can we access the value of these parameters in our code? The setup method has an argument Parameters par. It is via this argument that the parameters defined in the sitemap will be passed to the generator. This argument will be assigned to the **parameters** variable defined in AbstractGenerator. To obtain the value of each parameter we can invoke the following method on the parameters variable: public java.lang.String getParameter(java.lang.String name). This method returns the value of the specified parameter, or

throws an exception if there is no such parameter.

With all this in mind, we can finally build our configurable RMIGenerator. Also, this time we are going to extend the **ServiceableGenerator** instead of the **AbstractGenerator** class. This way, we can make use of the **ServiceManager** to obtain a SAXParser.

At this moment we decide that if there is no value given to the src attribute in the sitemap (**source is null**), we will invoke the **sayHello** method and otherwise the getResource with the appropriate parameter. When the value of the src attribute is the **empty string**, the **getResource** method is invoked, so this should be **handled by the RMI server application**. After a little bit of thinking about how to code all this, we eventually wrote the following generator:

```
package test; // import the necessary classes from the java.io package import
java.io.IOException; import java.io.StringReader; // import the necessary classes from the
java.rmi package import java.rmi.Naming; import java.rmi.RemoteException; import
java.rmi.NotBoundException; // import the necessary SAX classes import
org.xml.sax.InputSource; import org.xml.sax.SAXException; // import of the classes used
from Cocoon import org.apache.cocoon.ProcessingException; import
org.apache.cocoon.generation.ServiceableGenerator; // Avalon Framework import
org.apache.avalon.framework.parameters.Parameters; import
org.apache.avalon.framework.parameters.ParameterException; // needed for obtaining
parser in Cocoon import org.apache.excalibur.xml.sax.SAXParser; import
test.ServerFunctions; public class MyGenerator extends ServiceableGenerator { public void
generate () throws IOException, SAXException, ProcessingException { String host; // lookup
parameter 'host' try { host = parameters.getParameter("host"); // test if host is not the empty
string if (host == "") { throw new ParameterException( "The parameter 'host' may not be the
empty string"); } } catch (ParameterException pe) { // rethrow as a ProcessingException throw
new ProcessingException( "Parameter 'host' not specified",pe); } String bindname; // lookup
parameter 'bindname' try { bindname = parameters.getParameter("bindname"); // test if
bindname is not the empty string if (bindname == "") { throw new ParameterException( "The
parameter 'bindname' may not be the empty string"); } } catch (ParameterException pe) { //
rethrow as a ProcessingException throw new ProcessingException( "Parameter 'bindname'
not specified",pe); } String port = ""; // lookup parameter 'port' try { port =
parameters.getParameter("port"); port = ":" + port; } catch (ParameterException pe) { // reset
port to the empty string // port is not required port = ""; } try { ServerFunctions obj =
(ServerFunctions)Naming.lookup("//" + host + port + "/" + bindname); String message = ""; //
determine the method to invoke // depending on value of source if (this.source == null) {
message = obj.sayHello(); } else { message = obj.getResource(source); } SAXParser parser
= null; parser = (SAXParser)this.manager.lookup(SAXParser.ROLE); InputSource
inputSource = new InputSource( new StringReader(message));
parser.parse(inputSource,super.xmlConsumer); } catch (NotBoundException nbe) { throw
new ProcessingException(" Error looking up the RMI server application",nbe); } catch
(ServiceException ce) { throw new ProcessingException(" Error obtaining a SAXParser",ce); }
} }
```

Since we have already explained every step that happens in this generator, we are confident that everyone will understand the code. We are now ready to deploy this generator.

3.3.5. The final step: deployment

We can now compile our classes and put the generator, along with the ServerFunctions interface, in the right place. For compiling, we used the following command:

```
javac -classpath .:jar/xerces.jar:jar/cocoon.jar:jar/framework.jar: \
jar/excalibur.jar:jar/exc-scratchpad.jar \ -d compiled/ test/ServerFunctions.java
test/MyGenerator.java
```

where xerces.jar is a symbolic link to "$TOMCAT_HOME/webapps/cocoon/WEB-INF/lib/xercesImpl-2.0.0.jar", framework.jar to "$TOMCAT_HOME/webapps/cocoon/WEB-INF/lib/avalon-framework-4.1.2.jar", excalibur.jar to "$TOMCAT_HOME/webapps/cocoon/WEB-INF/lib/avalon-excalibur-4.1.jar" and exc-scratchpad.jar to "$TOMCAT_HOME/webapps/cocoon/WEB-INF/lib/avalon-excalibur-scratchpad-20020212.jar". This is valid for Cocoon 2.0.2-dev. If you use another version of Cocoon 2, you might have to change some of these names. If your platform does not allow the use of symbolic links, you should use the complete path to the corresponding jar-files.

Now that these classes are compiled we can place them in the $TOMCAT_HOME/webapps/cocoon/WEB-INF/classes/" directory as before. Now all that is left is shutting down Tomcat/Cocoon, emptying the work directory, modifying the sitemap, setting up a RMI server application and starting Tomcat/Cocoon.

4. Future plans

The first version of this generator was written as a proof-of-concept. The latest version (as given here, extending the ServiceableGenerator) only foresees in the **generate** method. There are a number of plans we still have to extend the functionality and thus usability of this generator:

- allow passing of a (J)DOM document instance as a resource to our generator. JDOM does require an additional entry in the classpath.
- supply a possibility for caching documents
- if the RMI server application can generate SAX events, try to pass the xmlConsumer to the server application as the ContentHandler

These are some of the extensions we have in mind for this generator. Our goal is to complete these steps within a few weeks (it will probably be a bit longer since the deadline of our thesis is only three weeks a way at the time of writing).

1. Comments

add your comments