# XSP Internals (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. Index

This document presents Apache Cocoon's dynamic markup language framework and its use in implementing XSP:



## 2. Markup-to-code Transformation

XSP is based on a general-purpose markup-to-code transformation engine built around three key abstractions:

- **Dynamic Markup Language**. An namespace-qualified XML vocabulary providing *code embedding* directives. An associated *dynamic markup language processor* transforms static markup interspersed with code embedding directives into an equivalent *source program string* written in a *target programming language*. Upon execution, the generated program will rebuild the original XML document as augmented by dynamic content emitted by the embedded code.
- **Programming Language**. A procedural language in which the dynamic markup processor generates source code from an input XML document. Its associated *programming language processor* is responsible for compiling, loading and executing the generated code within the boundaries of its calling environment.
- **Program Generator**. A component that integrates markup and programming language processors to build and execute markup-generating programs derived from XML documents. Beyond this "glue" role, this component is responsible for persistently storing generated programs as well as automatically rebuilding them should their source XML documents change on disk after program generation.

Despite its particular usage for XSP, ProgramGenerator is not restricted to run in a server pages environment.

## 3. XSP and Cocoon Generators

As a rule, XSP pages are translated into Cocoon Generator's.

### 3.1. Server Pages Generator Proxy

Generator's created by XSP are invoked exclusively through ServerPagesGenerator, a proxy that uses Cocoon's ProgramGenerator component to load pages and subsequently delegates actual SAX event generation to them.

The terms Generator and ProgramGenerator are somewhat confusing. Here, Generator refers to a Cocoon org.apache.cocoon.generation.Generator instance responsible for the initial feeding of Cocoon's SAX pipeline. ProgramGenerator, on the other hand, refers to a Cocoon component responsible for building and executing programs derived from XML documents containing dynamic markup: org.apache.cocoon.components.language.generator.ProgramGenerator

ServerPagesGenerator attempts to cope with a not unlikely possibility: *premature* termination of proxied generator execution. "Premature" here means that the invoked generator may return after starting one or more SAX events but without properly ending them.

While this not an expected scenario in "manual" SAX programming, server pages may well need to terminate in the middle of document production:

```
<page> <title>For Your Eyes Only</title> <xsp:logic> if
(!request.getParameter("pet").equals("Cheetah")) { <p> Hey, you're not Tarzan! </p> /***
Unclosed SAX events here! ***/ return; } </xsp:logic> <!-- Multi-racial Jane affair description
follows --> . . . </page>
```

The server pages generator proxy is defined in the sitemap as follows:

```
. . . <map:generator name="serverpages"
src="org.apache.cocoon.generation.ServerPagesGenerator"/> . . . <map:pipelines>
<map:pipeline> . . . <map:match pattern="/samples/*.xsp"> <map:generate
type="serverpages" src="../samples/documents/{1}.xsp"> <!-- <parameter
name="markup-language" value="xsp"/> <parameter name="programming-language"
value="java"/> --> </map:generate> <map:transform type="xslt"
src="../samples/stalemates/simple-page.xsl"/> <map:serialize type="html"
mime-type="text/html"/> </map:match> . . . </map:pipeline> </map:pipelines>
```

Note that parameters markup-language and programming-language default to *xsp* and *java* respectively.

The complete XSP sitemap configuration is explained below.

## 3.2. XSP Generators and Compiled Languages

For the Java language (and other compiled languages like Rhino Javascript), XSP pages are translated into classes extending AbstractServerPage . This class, in turn, extends ComposerGenerator , which gives it access to commonly used components such as *parser* or *cocoon* itself (typically used as EntityResolver for request URI's).

AbstractServerPage implements org.apache.arch.Modifiable. This is tested by ProgramGenerator to assert whether the page has been invalidated as a result of files it depends on having changed on disk. These files are typically logicsheets and template files included by means of XInclude.

As of this writing, XInclude support is still unimplemented but will be based on Donald Ball's (possibly extended) XIncludeTransformer.

AbstractServerPage implements Modifiable by means of two *static* variables: dateCreated and dependencies (a, possibly empty, array of File's pointing to logicsheets and other files included during the code generation stage).

AbstractServerPage also provides a boolean hasContentChanged() method that is tested by ServerPagesGenerator to assert whether dynamic content should not be regenerated for a given request. The default implementation unconditionally returns true, but can be overridden by XSP pages based on their interpretation of the Cocoon request object. This is an *experimental* feature that will

become meaningful only when a SAX-event caching mechanism is added to Cocoon.

Finally, AbstractServerPage also provides a number of utility methods used to shorten the generation of SAX events not requiring a namespace.

## 4. The Programming Language Processor

A Cocoon's ProgrammingLanguage processor exposes the following methods:

- load. Load a program from a file in a given directory, compiling it, if necessary, using a given encoding.
- instantiate Create a new instance of a previously loaded program
- unload Discard a previously loaded program performing any necessary cleanup
- getSourceExtension Return the canonical source file extension used by this programming language
- getCodeFormatter Return an (optional) instance of CodeFormatter used to beautify source code written in this programming language
- quoteString Escape a string constant according to the programming language rules

A default implementation ( AbstractProgrammingLanguage ) is provided that extends org.apache.arch.named.AbstractNamedComponent and retrieves language-related sitemap parameters.

### 4.1. Filenames and Encoding

load and unload are passed a file/directory pair used to locate the program.

The baseDirectory should be an absolute pathname pointing to the top-level directory (also known as *repository*) containing the program file.

The filename is a path, *relative to the baseDirectory*, pointing to the program file.

Source program filenames are built by concatenating the repository's baseDirectory name, the given filename, the dot extension separator and the language-specific source or object *extensions*. The cross-platform File.separator is used to ensure portability.

The filename must **not** contain any source or object extension. It may, though, contain subdirectories depending on its position within the repository tree. Also, programming languages **must** define a source extension even when their actual compilers/interpreters do not enforce this. This is also true of *object* extensions for compiled languages. Furthermore, the dot character is *always* used as the extension separator.

Finally, the (optional) encoding argument specifies the how the source program file contents are encoded. This argument can be null to specify the platform's default encoding.

### 4.2. Loading Programs

Currently, programs returned by the load operation are "plain" Java Object's and are not required to implement any interface or to extend any particular class.

This may change in the future so that the loaded program may be required to provide dependency information (for automatic reloading) as well as source code information (for debugging purposes).

Compiled programs attempt to locate the *object program* first. If found, it's loaded in a language-specific way and then returned to the calling environment. Failing that, the source file is located and the language-specific compiler is invoked prior to actual program loading.

Of course, it is an error for the source program file not to exist as a readable, regular operating system file.

### 4.3. Unloading Programs

When a previously loaded program is no longer needed (or becomes "outdated" as explained below) the language processor may need to perform cleanup actions, such as releasing memory or (in the case of Java-like compiled languages) reinstantiating the class loader.

Loaded programs may become outdated as a consequence of events external to the programming language processor. In a server pages environment, this is the result of the source XML document (or any of the files it depends on) having changed on disk.

The base class AbstractProgrammingLanguage implements this method *as final* to delete the unloaded *source* program file and delegate actual unloading to method doUnload.

Method doUnload is *not* defined as abstract in order to relieve interpreted subclasses from having to implement an empty method when no cleanup is required.

Currently, only the program object is being passed to unload. It may be possible for some interpreted languages to also require knowing what file the program was originally loaded from. In this case, instantiation should take place through the program object itself, rather than through the language processor (see *Program Instantiation* below)

### 4.4. Instantiating Programs

The program object returned by load must act as an factory capable of creating *program instance* objects on demand.

Currently, instantiation is performed by the language processor given a previously loaded program.

Compiled programs use a language-specified class loader to create a new program instance.

For compiled languages, it is possible to guarantee that a generated program implements a given interface or extends a given class. For interpreted languages, though, it may be necessary to pass an additional prototype object to load as to ensure that created instances conform to a given Java type expected behavior.

### 4.5. Source Extensions

All languages are required to return a *source extension*. This extension is used to locate source files for subsequent interpretation or compilation.

### 4.6. Code Formatting

Programming languages may provide a CodeFormatter instance used by code generators to beautify source code.

Interface CodeFormatter exposes a single method: formatCode. formatCode takes as arguments a String containing the source code to be beautified and an encoding to be preserved during string conversions.

Code formatters can be associated with a programming language by specifying a code-formatter parameter in its sitemap configuration:

<parameter name="code-formatter"

value="org.apache.cocoon.components.language.programming.java.JstyleFormatter"/>
Currently, Jstyle is being used for Java source formatting. This open source project appears to be stagnated and lacks advanced formatting options present in other (unfortunately, not open-sourced) products like Jindent.

### 4.7. String Quoting

Method quoteString applies the programming language string constant escaping rules to its input argument.

This method exists to assist markup language code generators in escaping Text XML nodes.

## 5. Compiled Languages

Compiled languages extend the ProgrammingLanguage abstraction by introducing the notions of *compilation* and *object extension*.

A base implementation (CompiledProgrammingLanguage) is provided that adds the following protected variables and abstract/overridable methods:

- Variable compilerClass. Used to create instances of the language's compiler.
- Variable deleteSources. Used to state whether intermediate source files should be deleted after successful compilation
- Method getObjectExtension. Used to build object filenames
- Method loadProgram. Used to perform actual program load after source and (possibly) object files have been located
- Method doUnload. Used to perform cleanup after program unloading

Object files are not required to be *Java class files*. It's up the the compiled programming language processor to handle object files.

Compiled programming languages must specify their preferred compiler as a sitemap parameter:

```
<component-instance name="java"
class="org.apache.cocoon.components.language.programming.java.JavaLanguage"> . . .
<parameter name="compiler"
value="org.apache.cocoon.components.language.programming.java.Jikes"/> . . .
</component-instance>
```

### 5.1. Object Extensions

All compiled languages are required to return a *source extension*. This extension is used to locate object files for subsequent loading.

### 5.2. Object Program Loading

Concrete compiled programming languages must implement the abstract method loadProgram to actually load an *object* program resulting from compilation.

### 5.3. Program Compilation

Compilation is delegated to a sitemap-specified LanguageCompiler instance, as explained below.

### 5.4. Compilers

Interface LanguageCompiler defines the initialization and behavior for all compilers.

Methods exposed by this interface are:

- setFile. Used to specify the source file to be compiled. This should be an absolute filename
- setSource. Used to specify the directory where dependent source files (if any) are stored
- setDestination. Used to specify the directory where the generated object files should be placed
- setClasspath. Used to specify the class loading path used by the compiler. While this option is named after Java's *classpath* system variable, its semantics are language-independent
- setEncoding. Used to specify the encoding used by the input source file
- compile. The compiler's workhorse (boolean)
- getErrors. Used to retrieve a list of compilation error messages should compilation fail

### 5.4.1. Compiler Errors

Error message producer by the compiler must be collected and massaged by the LanguageCompiler in order to wrap each of them as a CompilerError instance.

Class CompilerError exposes the following methods:

- getFile. Returns the program filename originating the error
- isError. Asserts whether the error is a server error or simply a warning
- getStartLine. Returns the starting line of the offending code
- getStartColumn. Returns the starting column (within the starting line) of the offending code
- getEndLine. Returns the ending line of the offending code
- getEndColumn. Returns the ending column (within the ending line) of the offending code
- getMessage. Returns the actual error message text

### 5.4.2. Java Compilers

For the Java language, 2 pluggable compilers are available:

- *Javac*. A wrapper to Sun's builtin compiler
- *Jikes*. A wrapper to IBM's *Jikes* compiler

Both of these compilers are based on AbstractJavaCompiler.

### 5.4.3. Other Compilers

Since Rhino Javascript provides its own, only compiler (*jsc*), class JavascriptLanguage doesn't use the compiler class initialized by CompiledProgrammingLanguage.

## 5.5. Object Program Unloading

CompiledProgrammingLanguage extends the default implementation provided by AbstractProgrammingLanguage by deleting the *object* program file and delegating actual unloading to the doUnload method.

Method doUnload provides an empty default implementation that can be overridden by derived compiled languages should unloading cleanup be actually required.

For Java-based compiled languages (i.e., those using *class files* as their object format, unloading implies reinstantiating their class loader such that it "forgets" about previously loaded classes thus becoming able to refresh class files updates since their last load.

This is a commonly-used workaround for the (somewhat buggy) standard Java class loader, which doesn't provide for an explicit method for reloading class files.

### 5.6. The Cocoon Class Loader

To circumvent standard Java class loaders limitation, Cocoon provides a simple customized class loader (RepositoryClassLoader) that features:

- A directory-based extensible classpath that can grow at execution time
- Class reloading by means of reinstantiation

RepositoryClassLoader extends java.lang.ClassLoader adding an addDirectory method that adds the directory pointed to by its String argument to its local classpath.

Access to *protected* RepositoryClassLoader class is proxied through interface ClassLoaderManager. This interface exposes the following methods:

- addDirectory. Passed to the proxied RepositoryClassLoader
- loadClass. Passed to the proxied RepositortyClassLoader
- reinstantiate. Used to discard the previous class loader and create a new one

Class ClassLoaderManagerImpl implements ClassLoaderManager in a singleton-like fashion that ensures that only one instance of this class loader exists, thus ensuring the reinstantiation mechanism works properly.

The class loader can be specified in the sitemap on a per-language basis:

```
<component-instance name="java"
class="org.apache.cocoon.components.language.programming.java.JavaLanguage"> . . .
<parameter name="class-loader"
value="org.apache.cocoon.components.classloader.ClassLoaderManagerImpl"/>
</component-instance>
```

Alternatively, the class loader can be specified in the sitemap as a global component:

```
<component role="class-loader"
class="org.apache.cocoon.components.classloader.ClassLoaderManagerImpl"/>
```

## 6. Interpreted Languages

Interpreted languages for which a Java-based interpreter exists are supported by means of IBM's outstanding Bean Scripting Framework (BSF).

Currently, BSF supports:

- Mozilla Rhino
- NetRexx
- Jacl
- JPython
- VBScript (Win32 only)
- JScript (Win32 only)
- PerlScript (Win32 only)
- BML (Not applicable to server pages)
- LotusXSL (Not applicable to server pages)

Interpreted language support is still unimplemented!
While BSF is extremely easy to use and very stable, there's still a challenge in writing

code-generation logicsheets for each of this languages; this task requires familiarity with XSP internals, XSLT and, above all, the programming language at hand...Despite being supported by BSF, Rhino Javascript is separately supported by Cocoon as a compiled language in order to take advantage of automatic class reloading and persistent class file storage.Since ProgramGenerator clients will typically require that program instances implement a given interface or extend a given class, method instantiate in interface ProgrammingLanguage may need to be augmented with a prototype interface that can be used by each language processor to ensure that the program instance can act as a Java object of the given type.

## 7. The Markup Language Processor

A Cocoon's MarkupLanguage processor exposes the following methods:

- getEncoding. Return the encoding to be used in program generation and compilation or null to use the platform's default encoding
- generateCode. Given a DOM Document written in a given *markup language*, generate an equivalent program in a given *programming language*)

A base markup language processor implementation is provided in class AbstractMarkupLanguage. This class extends org.apache.arch.named.AbstractNamedComponent to set the markup language's associated namespace using the following required parameters:

- prefix. The markup language's namespace prefix
- uri. The markup language's namespace URI

<component-instance name="xsp"
class="org.apache.cocoon.components.language.markup.xsp.XSPMarkupLanguage">
<parameter name="prefix" value="xsp"/> <parameter name="uri"
value="http://apache.org/xsp"/> </component-instance>

AbstractMarkupLanguage adds a number of abstract/overridable methods that must be implemented by concrete markup language processors:

- preprocessDocument. Augment the input DOM Document to prepare it for simpler, faster logicsheet-based code generation
- getLogicsheets. Return the list of logicsheets declared in the input document according to the syntax of the markup language at hand
- addDependency. Add a dependency on an external file. This is used to inform the concrete markup language processor about XML documents included by means of XInclude as well as any intervening logicsheet

AbstractMarkupLanguage is currently tied to logicsheets as the *only* means of generating source code. While logicsheets provide a very powerful means for code generation, good design dictates that the actual code generation mechanism should be decoupled from the dynamic markup language abstraction.The current code generation strategy is DOM-based. In principle, this is adequate because document preprocessing may need random access to document nodes. Code generation is being reconsidered, however, to overcome this and make it possible to reuse Cocoon's SAX-based filtering pipeline.

### 7.1. Markup Encoding

All markup languages must provide a way to declare the XML document's encoding so that it is preserved during code generation, beautifying and compilation.

This is required for proper i18n support, where the default encoding usually replaces "exotic" characters with question marks.

Ideally, it should be possible to determine the source XML document's encoding from its declaring <?xml?> processing instruction. Unfortunately, XML parsers (both DOM and SAX) don't seem to provide access to it, thus forcing server pages authors to redundantly specify it.

### 7.2. The Logicsheet class

A *logicsheet* is an XML filter used to translate user-defined dynamic markup into equivalent code embedding directives for a given markup language.

Logicsheets lie at the core of XSP's promise to separate logic from content and presentation: they make dynamic content generation capabilities available to content authors not familiar with (and not interested in) programming.

For a detailed description of logicsheets, see Logicsheet Concepts.

Logicsheets are represented in class Logicsheet. This class exposes the following methods:

- setInputSource. Set the InputSource pointing to the XSLT stylesheet to be used for dynamic tag transformation
- apply. Apply the stylesheet to a given document

Logicsheet takes care of preserving all namespaces defined in the input document. This is necessary when multiple logicsheets are applied and multiple namespaces are used in the input document.

Currently, Logicsheet is a concrete class. It should be redefined as an interface in order to decouple it from the use of XSLT stylesheets. Again, while stylesheets are the "obvious" way to implement logicsheets, a user-supplied XML filter may also be used in some cases. The current implementation uses an ugly hack where a Xalan stylesheet processor is used to perform the transformation without an intervening stylesheet processor wrapping abstraction.

### 7.3. Named Logicsheets

As explained in Logicsheet Concepts, logicsheets are typically associated with a single object type whose methods it wraps to make them available as *markup commands*.

Markup commands related to a given object type are grouped under a single namespace.

Class NamedLogicsheet extends Logicsheet to associate it with a namespace. This class exposes the following additional methods:

- setPrefix. To set the logicsheet's namespace prefix
- getPrefix. To retrieve the logicsheet's namespace prefix
- setUri. To set the logicsheet's namespace URI
- getUri. To retrieve the logicsheet's namespace URI

Named logicsheets are used as builtin logicsheets by AbstractMarkupLanguage to preload logicsheets and make them accessible to dynamic XML documents without explicit declaration.

This feature relieves page authors from the need to explicitly declare commonly used logicsheets in their documents. Builtin logicsheets are automatically applied if the document declares their same namespace URI.

The current AbstractMarkupLanguage implementation wrongly binds named logicsheets based on their namespace *prefix* instead of their URI!

### 7.4. Logicsheet Code Generators

Logicsheets translate dynamic tags to equivalent code-embedding directives expressed in the markup language at hand. They do not, however, actually emit the final source code program.

Code generation as such (i.e., the final production of a string containing a source program written in a programming language) is the responsibility of class LogicsheetCodeGenerator.

Class LogicsheetCodeGenerator exposes the following methods:

- addLogicsheet. Add a logicsheet to the generator's logicsheet list. Logicsheets are applied in the order of their addition.
- generateCode. Return a string containing a source program resulting from successively applying added logicsheets.

Though "regular" logicsheets as such do not emit source code, LogicsheetCodeGenerator expects its *last* stylesheet to produce *a single element* containing only a text node.

This final, programming language-specific logicsheet is responsible for actually expanding code-embedding directives into source code.

For each supported target programming language, markup languages must provide a *core* logicsheet.

LogicsheetCodeGenerator is currently implemented as a class. It should be defined as an interface in order to the decouple the code generator abstraction from its logicsheet-based implementation. This would allow for alternative code-generation strategies to be plugged.

### 7.5. Markup Language Definition

Markup languages are defined in the sitemap as follows:

```
<component-type name="markup-language"> <component-instance name="xsp"
class="org.apache.cocoon.components.language.markup.xsp.XSPMarkupLanguage">
<parameter name="prefix" value="xsp"/> <parameter name="uri"
value="http://apache.org/xsp"/> <target-language name="java"> <parameter
name="core-logicsheet"
value="resource://org/apache/cocoon/components/language/markup/xsp/java/xsp.xsl"/>
<builtin-logicsheet> <parameter name="prefix" value="xsp-request"/> <parameter name="uri"
value="http://apache.org/xsp/request/2.0"/> <parameter name="href"
value="resource://org/apache/cocoon/components/language/markup/xsp/java/request.xsl"/>
</builtin-logicsheet> <builtin-logicsheet> <parameter name="prefix" value="xsp-response"/>
<parameter name="uri" value="http://apache.org/xsp/response/2.0"/> <parameter
name="href"
value="resource://org/apache/cocoon/components/language/markup/xsp/java/request.xsl"/>
</builtin-logicsheet> </target-language> </component-instance> </component-type>
```

Here, the markup language *prefix* and *uri* are defined together with one or more *supported programming languages*.

For each supported programming language, a corresponding *core logicsheet* is defined as a URL pointing to its code-generation stylesheet.

Optionally, each supported programming language may define one or more namespace-mapped *builtin logicsheets*.

## 8. The XSP Markup Language

So far, programming and markup languages have been described in general, without explicitly

referring to the XSP language.

This section describes how the above described framework is used to implement XSP in particular. For a description of logicsheet authoring requirements for XSP in Java, see XSLT Logicsheets and XSP for Java.

The XSP syntax is being revised to allow for the omission of the root <xsp:page> element. This is convenient for the (typical) case in which all logic has been conveniently placed in logicsheets so that XSP pages do not need to embed any code. In this case, there should be no need for the <xsp:page> element.

### 8.1. Markup Encoding

Method getEncoding is implemented by class XSPMarkupLanguage by retrieving the attribute named encoding in the root <xsp:page> element.

In absence of a <xsp:page> root element, the encoding will be retrieved from an attribute named xsp:encoding present in the "user" root element.

### 8.2. Document Preprocessing

XSPMarkupLanguage preprocesses its input document by:

- Setting the root element file-name attribute to the *base* filename of its input source.
- Setting the root element file-path attribute to the *base* directory name of its input source.
- Setting the root element creation-date attribute to the current system time
- Escaping text nodes according to the rules dictated by the target programming language. This excludes text nodes enclosed in <xsp:logic> and <xsp:expr> elements, as they are to be output as code.

A feature to be added is collecting all text nodes under the document's root element and replacing them by references to their relative index position. This will allow for the generation of contentHandler.characters method calls that reference char arrays instead of constant String's. In addition to saving execution time, this will result in decreased program size because common substrings can be output by "reusing" their containing character arrays along with their corresponding offsets and lengths.

### 8.3. Dependency Tracking

File dependencies passed to XSPMarkupLanguage by its AbstractMarkupLanguage superclass are stored in top-level <xsp:dependency> elements.

These elements are used by XSP code-generation logicsheets to populate the File array defined by the generated classes' AbstractServerPage superclass.

### 8.4. XSP Builtin Logicsheets

XSP for Java currently provides only 2 builtin logicsheets: request and response, associated with their corresponding Cocoon counterparts.

A mechanism is needed for Cocoon to pass additional objects to XSP pages. In particular, for the servlet execution environment, access to servlet objects is a must.

## 9. The DOM-XSP Markup Language

The new, SAX-based XSP version for Cocoon is not backwards compatible with its DOM-based

former self.

In order to protect the existing DOM-based XSP code base, a "new" markup language will be added that simply wraps existing XSP version 1 pages, postprocessing their generated documents to convert them into SAX events.

While this solution implies additional overhead, it provides a simple path for migrating existing XSP pages.

In addition to this straight-forward mechanism, the new, SAX-based XSP version will overload the xspExpr method to accept as argument a Node expression and transform it to equivalent SAX events.

For the long run, though, developers are strongly encouraged to replace their "legacy" DOM pages and classes with equivalent, faster SAX counterparts.

## 10. The Program Generator

The ProgramGenerator interface exposes a single load method that takes as arguments a File pointing to a source XML document, as well as a *markup* and *programming* language name pair.

This method is responsible for locating, loading and instantiating a program derived from the given source document. Failing this, the program is generated and stored in an external, persistent *repository*.

Once instantiated, the program is kept in an in-memory cache for speeding up subsequent requests.

For each request, the source XML document is checked for changes and the program instance is queried for dependency changes so that the program can be automatically regenerated and reloaded if needed. This default behavior can be disabled by means of a sitemap parameter.

Currently, the program *instance* (as opposed to the program object itself) is queried for invalidating changes. This should change as a consequence of defining a separate Program abstraction as part of the upcoming addition of debugging support.

A default implementation of ProgramGenerator is provided that uses a FilesystemStore as repository: ProgramGeneratorImpl.

### 10.1. Program Repository

FilesystemStore is an implementation of the Store interface that uses a filesystem, hierarchical *directory* as its persistence mechanism.

FilesystemStore implements Store directly. A higher-level interface (PersistentStore) should be defined to accommodate other sensible persistent storage mechanisms such as relational databases or object databases like Ozone.

FilesystemStore expects the String representation of its key's to be *filenames* relative to its directory root.

Objects returned by FilesystemStore's get method are File's pointing to their corresponding entries (or null if their associated file doesn't exit).

FilesystemStore stores Java objects according to the following rules:

- null values generate empty directories
- String values are dumped to text files
- All other Object's are serialized

### 10.2. Program Reloading

Unless the auto-reload sitemap option is in effect, ProgramGeneratorImpl will check whether program instances implement interface Modifiable in order to assert whether they should be regenerated and reloaded.

Method load uses its markupLanguageName and programmingLanguage arguments to retrieve the corresponding NamedComponent instances.

In server pages mode, these parameters are set by the calling ServerPagesGenerator from parameters passed via the sitemap <process> section.

The appropriate MarkupLanguage and ProgrammingLanguage instances are used to generate and load a program for which an instance is created and then returned to the calling environment.

## 11. Named Components

In order to support pluggable markup and programming languages, a new abstraction was added to Cocoon's arch core interfaces: org.apache.arch.named.NamedComponent.

Interface NamedComponent is simply an extension to org.apache.arch.Component that exposes a getName() method.

NamedComponent's belong to a collection of components sharing the same Java type and are individually identified by a name unique within each collection.

A org.apache.arch.named.NamedComponentManager is a component responsible for storing and locating NamedComponent instances. This interface exposes the following methods:

- getComponent. Retrieve a NamedComponent instance given its type and name.
- getTypes. Return an Enumeration of all known NamedComponent types.
- getComponents. Return an Enumeration of all NamedComponents within a given type.

A default implementation is provided for this interface:
org.apache.arch.named.NamedComponentManagerImpl.

Class org.apache.arch.named.AbstractNamedComponent provides a base implementation for NamedComponent that extends org.apache.arch.Configurable. This class exposes the following methods:

- setConfiguration. Retrieve named-component sitemap configuration values converting parameter name/value pairs into Parameters passed to subclasses for easier initialization
- setParameters. An empty method to be overridden by subclasses for parameter-based initialization
- setAdditionalConfiguration. An empty method to be overridden by subclasses when parameter-based initialization is not sufficient because there are nested configuration elements in the corresponding sitemap entry
- getRequiredParameter. A static convenience method that returns a named parameter as a String throwing an IllegalArgumentException if the parameter was not specified in the sitemap configuration

## 12. XSP Sitemap Configuration

The sitemap configuration shown here is likely to change in the near future.

A (rather verbose) sitemap definition for XSP follows:

<component role="factory" class="org.apache.avalon.NamedComponentManagerImpl">
<component-type name="programming-language"> <component-instance name="java"
class="org.apache.cocoon.components.language.programming.java.JavaLanguage">
<parameter name="compiler"
value="org.apache.cocoon.components.language.programming.java.Javac"/> <parameter
name="code-formatter"
value="org.apache.cocoon.components.language.programming.java.JstyleFormatter"/>
<parameter name="class-loader"
value="org.apache.cocoon.components.classloader.ClassLoaderManagerImpl"/> <parameter
name="delete-sources" value="false"/> </component-instance> </component-type>
<component-type name="markup-language"> <component-instance name="xsp"
class="org.apache.cocoon.components.language.markup.xsp.XSPMarkupLanguage">
<parameter name="prefix" value="xsp"/> <parameter name="uri"
value="http://apache.org/xsp"/> <target-language name="java"> <parameter
name="core-logicsheet"
value="resource://org/apache/cocoon/components/language/markup/xsp/java/xsp.xsl"/>
<builtin-logicsheet> <parameter name="prefix" value="xsp-request"/> <parameter name="uri"
value="http://apache.org/xsp/request/2.0"/> <parameter name="href"
value="resource://org/apache/cocoon/components/language/markup/xsp/java/request.xsl"/>
</builtin-logicsheet> <builtin-logicsheet> <parameter name="prefix" value="xsp-response"/>
<parameter name="uri" value="http://apache.org/xsp/response/2.0"/> <parameter
name="href"
value="resource://org/apache/cocoon/components/language/markup/xsp/java/request.xsl"/>
</builtin-logicsheet> </target-language> </component-instance> </component-type>
</component> <component role="program-generator"
class="org.apache.cocoon.components.language.generator.ProgramGeneratorImpl">
<parameter name="repository" value="/tmp/repository"/> <parameter name="auto-reload"
value="true"/> </component> <generator name="serverpages"
class="org.apache.cocoon.generation.ServerPagesGenerator"/> <!-- <component
role="class-loader"
class="org.apache.cocoon.components.classloader.ClassLoaderManagerImpl" /> -->
<sitemap> <partition> <process uri="simple-page.xsp"
source="../samples/documents/simple-page.xsp"> <generator name="serverpages"> <!--
<parameter name="markup-language" value="xsp"/> <parameter
name="programming-language" value="java"/> --> </generator> <filter name="xslt">
<parameter name="stylesheet" value="../samples/documents/simple-page.xsl"/> </filter>
<serializer name="html"> <parameter name="contentType" value="text/html"/> </serializer>
</process> </partition> </sitemap>

## 1. Comments

add your comments