

Axiom Developer Guide

Axiom Developer Guide

1.2.16

Licensed to the Apache Software Foundation (ASF) under one or more contributor license agreements. See the NOTICE file distributed with this work for additional information regarding copyright ownership. The ASF licenses this file to you under the Apache License, Version 2.0 (the "License"); you may not use this file except in compliance with the License. You may obtain a copy of the License at

<http://www.apache.org/licenses/LICENSE-2.0>

Unless required by applicable law or agreed to in writing, software distributed under the License is distributed on an "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied. See the License for the specific language governing permissions and limitations under the License.

Table of Contents

1. Working with the Axiom source code	1
Importing the Axiom source code into Eclipse	1
Testing	1
Unit test organization	1
Testing Axiom with different StAX implementations	1
2. Design	3
General design principles and goals	3
OSGi integration and separation between API and implementation	3
Introduction	3
Requirements	3
Analysis of the Geronimo JAXB bundles	6
New abstract APIs	7
Common implementation classes	8
LifecycleManager design (Axiom 1.3)	9
Issues with the LifecycleManager API in Axiom 1.2.x	9
Cleanup strategy for temporary files	10
3. Release process	12
Release preparation	12
Prerequisites	14
Release	14
Post-release actions	17
References	17
A. Appendix	18
Installing IBM's JDK on Debian Linux	18

List of Figures

3.1. Package dependencies for r944680	12
3.2. Package dependencies for r939984	13

Chapter 1. Working with the Axiom source code

Importing the Axiom source code into Eclipse

In order to import the Axiom source code into Eclipse with the Maven Eclipse plugin, use the following command:

```
mvn -DskipTests=true -DdownloadSources=true install eclipse:eclipse
```

Testing

Unit test organization

Historically, all unit tests were placed in the `axiom-tests` project. One specific problem with this is that since all tests are in a common Maven module which depends on both `axiom-impl` and `axiom-dom`, it is not rare to see DOOM tests that accidentally use the LLOM implementation (which is the default). The project description in `axiom-tests/pom.xml` indicates that it was the intention to split the `axiom-tests` project into several parts and make them part of `axiom-api`, `axiom-impl` and `axiom-dom`. This reorganization is not complete yet¹. For new test cases (or when refactoring existing tests), the following guidelines should be applied:

1. Tests that validate the code in `axiom-api` and that do not require an Axiom implementation to execute should be placed in `axiom-api`. This primarily applies to tests that validate utility classes in `axiom-api`.
2. The code of unit tests that apply to all Axiom implementations and that check conformance to the specifications of the Axiom API should be added to `axiom-api` and executed in `axiom-impl` and `axiom-dom`. Currently, the recommended way is to create a base class in `axiom-api` (with suffix `TestBase`) and to create subclasses in `axiom-impl` and `axiom-dom`. This makes sure that the DOOM tests never accidentally use LLOM (because `axiom-impl` is not a dependency of `axiom-dom`).
3. Tests that check integration with other libraries should be placed in `axiom-integration`. Note that this is the only module that requires Java 1.5 (so that e.g. integration with JAXB2 can be tested).
4. Tests related to code in `axiom-api` and requiring an Axiom implementation to execute, but that don't fall into category 2 should stay in `axiom-tests`.

Testing Axiom with different StAX implementations

The following StAX implementations are available to test compatibility with Axiom:

Woodstox

This is the StAX implementation that Axiom uses by default.

Sun Java Streaming XML Parser (SJSXP)

This implementation is available as Maven artifact `com.sun.xml.stream:sjsxp:1.0.1`.

¹See AXIOM-311 [<https://issues.apache.org/jira/browse/AXIOM-311>].

StAX Reference Implementation

The reference implementation was written by BEA and is available as Maven artifact `stax:stax:1.2.0`. The homepage is <http://stax.codehaus.org/Home>. Note that the JAR doesn't contain the necessary files to enable service discovery. Geronimo's implementation of the StAX API library will not be able to locate the reference implementation unless the following system properties are set:

```
javax.xml.stream.XMLInputFactory=com.bea.xml.stream.MXParserFactory
javax.xml.stream.XMLOutputFactory=com.bea.xml.stream.XMLOutputFactoryBase
```

XL XP-J

“XL XML Processor for Java” is IBM's implementation of StAX 1.0 and is part of IBM's JRE/JDK v6. Note that due to an agreement between IBM and Sun, IBM's Java implementation for the Windows platform is not available as a separate download, but only bundled with another IBM product, e.g. WebSphere Application Server for Developers [<http://www.ibm.com/developerworks/downloads/ws/wasdevelopers/>].

On the other hand, the JDK for Linux can be downloaded as a separate package from the developerWorks site [<https://www.ibm.com/developerworks/java/jdk/linux/download.html>]. There are versions for 32-bit x86 (“xSeries”) and 64-bit AMD. They are available as RPMs and tarballs. To install the JDK properly on a Debian based system (including Ubuntu), follow the instructions given in the section called “Installing IBM's JDK on Debian Linux”.

Chapter 2. Design

General design principles and goals

Consistent serialization. Axiom supports multiple methods and APIs to serialize an object model to XML or to transform it to another (non Axiom) representation. This includes serialization to byte or character streams, transformation to StAX in push mode (i.e. writing to an `XMLStreamWriter`) or pull mode (i.e. reading from an `XMLStreamReader`), as well as transformation to SAX. The representations produced by these different methods should be consistent with each other. If a given use case can be implemented using more than one of these methods, then the end result should be the same, whichever method is chosen.

AXIOM-430 [<https://issues.apache.org/jira/browse/AXIOM-430>] provides an example where this principle was not respected.

It should be noted that this principle can obviously only be respected within the limits imposed by a given API. E.g. if a given API has limited support for DTDs, then a `DOCTYPE` declaration may be skipped when that API is used.

OSGi integration and separation between API and implementation

Introduction

This section addresses two related architectural questions:

- OSGi support was originally introduced in Axiom 1.2.9, but the implementation had a couple of flaws. This section discusses the rationale behind the new OSGi support introduced in Axiom 1.2.13.
- Axiom is designed as a set of abstract APIs for which two implementations are provided: LLOM and DOOM. It is important to make a clear distinction between what is part of the public API and what should be considered implementation classes that must not be used by application code directly. This also implies that Axiom must provide the necessary APIs to allow application code to access all features without the need to access implementation classes directly. This section in particular discusses the question how application code can request factories that support DOM without the need to refer directly to DOOM.

These two questions are closely related because OSGi allows to enforce the distinction between public API and implementation classes by carefully selecting the packages exported by the different bundles: only classes belonging to the public API should be exported, while implementation classes should be private to the bundles containing them. This in turn has implications for the packaging of these artifacts.

Requirements

Requirement 1. The Axiom artifacts **SHOULD** be usable both as normal JAR files and as OSGi bundles.



The alternative would be to produce two sets of artifacts during the build. This should be avoided in order to keep the build process as simple as possible. It should also be noted that the Geronimo Spec artifacts also meet this requirement.

Requirement 2. All APIs defined by the `axiom-api` module, and in particular the `OMAbstractFactory` API **MUST** continue to work as expected in an OSGi environment, so that code in downstream projects doesn't need to be rewritten.



This requirement was already satisfied by the OSGi support introduced in Axiom 1.2.9. It therefore also ensures that the transition to the new OSGi support in Axiom 1.2.13 is transparent for applications that already use Axiom in an OSGi container.

Requirement 3. `OMAbstractFactory` **MUST** select the same implementation regardless of the type of container (OSGi or non OSGi). The only exception is related to the usage of system properties to specify the default `OMMetaFactory` implementation: in an OSGi environment, selecting an implementation class using a system property is not meaningful.

Requirement 4. Only classes belonging to the public API should be exported by the OSGi bundles. Implementation classes should not be exported. In particular, the bundles for the LLOM and DOOM implementations **MUST NOT** export any packages. This is required to keep a clean separation between the public API and implementation specific classes and to make sure that the implementations can be modified without the risk of breaking existing code. An exception **MAY** be made for factory classes related to foreign APIs, such as the `DocumentBuilderFactory` implementation for an Axiom implementation supporting DOM.



When the Axiom artifacts are used as normal JAR files in a Maven build, this requirement implies that they should be used in scope `runtime`.

Although this requirement is easy to implement for the Axiom project, it requires changes to downstreams project to make this actually work:

- As explained in AXIS2-4902 [<https://issues.apache.org/jira/browse/AXIS2-4902>], there used to be many places in Axis2 that still referred directly to Axiom implementation classes. The same was true for Rampart and Sandesha2. This has now been fixed and all three projects use `axiom-impl` and `axiom-dom` as dependencies in scope `runtime`.
- Abdera extends the LLOM implementation. Probably, some `maven-shade-plugin` magic will be required here to create Abdera OSGi bundles that work properly with the Axiom bundles.
- For Spring Web Services this issue is addressed by SWS-822 [<https://jira.springsource.org/browse/SWS-822>].

Requirement 5. It **MUST** be possible to use a non standard (third party) Axiom implementation as a drop-in replacement for the standard LLOM and DOOM implementation, i.e. the `axiom-impl` and `axiom-dom` bundles. It **MUST** be possible to replace `axiom-impl` (resp. `axiom-dom`) by any Axiom implementation that supports the full Axiom API (resp. that supports DOM in addition to the Axiom API), without the need to change any application code.



This requirement has several important implications:

- It restricts the allowable exceptions to Requirement 4.
- It implies that there must be an API that allows application code to select an Axiom implementation based on its capabilities (e.g. DOM support) without introducing a hard dependency on a particular Axiom implementation.
- In accordance with Requirement 2 and Requirement 3 this requirement not only applies to an OSGi environment, but extends to non OSGi environments as well.

Requirement 6. The OSGi integration **SHOULD** remove the necessity for downstreams projects to produce their own custom OSGi bundles for Axiom. There **SHOULD** be one and only one set of OSGi bundles for Axiom, namely the ones released by the Axiom project.



Currently there are at least two projects that create their own modified Axiom bundles:

- Apache Geronimo has a custom Axiom bundle to support the Axis2 integration.
- ServiceMix also has a custom bundles for Axiom. However, this bundle only seem to exist to support their own custom Abdera bundle, which is basically an incorrect repackaging of the original Abdera code. See SMX4-877 [<https://issues.apache.org/jira/browse/SMX4-877>] for more details.

Note that this requirement can't be satisfied directly by Axiom. It requires that the above mentioned projects (Geronimo, Axis2 and Abdera) use Axiom in a way that is compatible with its design, and in particular with Requirement 4. Nevertheless, Axiom must provide the necessary APIs and features to meet the needs of these projects.

Requirement 7. The Axiom OSGi integration **SHOULD NOT** rely on any particular OSGi framework such as Felix SCR (Declarative Services). When deployed in an OSGi environment, Axiom should have the same runtime dependencies as in a non OSGi environment (i.e. StAX, Activation and JavaMail).



Axiom 1.2.12 relies on Felix SCR. Although there is no real issue with that, getting rid of this extra dependency is seen as a nice to have. One of the reasons for using Felix SCR was to avoid introducing OSGi specific code into Axiom. However, there is no issue with having such code, provided that Requirement 8 is satisfied.

Requirement 8. In a non OSGi environment, Axiom **MUST NOT** have any OSGi related dependencies. That means that the OSGi integration must be written in such a way that no OSGi specific classes are ever loaded in a non OSGi environment.

Requirement 9. The OSGi integration **MUST** follow established best practices. It **SHOULD** be inspired by what has been done to add OSGi integration to APIs that have a similar structure as Axiom.



Axiom is designed around an abstract API and allows for the existence of multiple independent implementations. A factory (`OMAbstractFactory`) is used to locate and instantiate the desired implementation. This is similar to APIs such as JAXP (`DocumentBuilderFactory`, etc.) and JAXB (`JAXBContext`). These APIs have been successfully "OSGi-fied" e.g. by the Apache Geronimo project. Instead of reinventing the wheel, we should leverage that work and adapt it to Axiom's specific requirements.

It should be noted that because of the way the Axiom API is designed and taking into account Requirement 2, it is not possible to make Axiom entirely compatible with OSGi paradigms (the same is true for JAXB). In an OSGi-only world, each Axiom implementation would simply expose itself as an OSGi service (of type `OMMetaFactory` e.g.) and code depending on Axiom would bind to one (or more) of these services depending on its needs. That is not possible because it would conflict with Requirement 2.

Non-Requirement 1. APIs such as JAXP and JAXB have been designed from the start for inclusion into the JRE. They need to support scenarios where an application bundles its own implementation (e.g. an application may package a version of Apache Xerces, which would then be instantiated by the `newInstance` method in `DocumentBuilderFactory`). That implies that the selected implementation depends on the thread context class loader. It is assumed that there is no such requirement for Axiom, which means that in a non OSGi environment, the Axiom implementations are always loaded from the same class loader as the `axiom-api` JAR.



This (non-)requirement is actually not directly relevant for the OSGi support, but it nevertheless has some importance because of Requirement 3 (which implies that the OSGi support needs to be designed in parallel with the implementation discovery strategy applicable in a non OSGi environment).

Analysis of the Geronimo JAXB bundles

As noted in Requirement 9 the Apache Geronimo has successfully added OSGi support to the JAXB API which has a structure similar to the Axiom API. This section briefly describes how this works. The analysis refers to the following Geronimo artifacts: `org.apache.geronimo.specs:geronimo-jaxb_2.2_spec:1.0.1` (called the "API bundle" hereafter), `org.apache.geronimo.bundles:jaxb-impl:2.2.3-1_1` (the "implementation bundle"), `org.apache.geronimo.specs:geronimo-osgi-locator:1.0` (the "locator bundle") and `org.apache.geronimo.specs:geronimo-osgi-registry:1.0` (the "registry bundle"):

- The implementation bundle retains the `META-INF/services/javax.xml.bind.JAXBContext` resource from the original artifact (`com.sun.xml.bind:jaxb-impl`). In a non OSGi environment, that resource will be used to discover the implementation, following the standard JDK 1.3 service discovery algorithm will (as required by the JAXB specification). This is the equivalent of our Requirement 1.
- The manifest of the implementation bundle has an attribute `SPI-Provider: true` that indicates that it contains provider implementations that are discovered using the JDK 1.3 service discovery.
- The registry bundle creates a `BundleTracker` that looks for the `SPI-Provider` attribute in active bundles. For each bundle that has this attribute set to `true`, it will scan the content of `META-INF/services` and add the discovered services to a registry (Note that the registry bundle supports other ways to declare SPI providers, but this is not really relevant for the present discussion).
- The `ContextFinder` class (the interface of which is defined by the JAXB specification and that is used by the `newInstance` method in `JAXBContext`) in the API bundle delegates the discovery of the SPI implementation to a static method of the `ProviderLocator` class defined by the locator bundle (which is not specific to JAXB and is used by other API bundles as well). This is true both in an OSGi environment and in a non OSGi environment.

The build is configured (using a `Private-Package` instruction) such that the classes of the locator bundle are actually included into the API bundle, thus avoiding an additional dependency.

- The `ProviderLocator` class and related code provided by the locator bundle is designed such that in a non OSGi environment, it will simply use JDK 1.3 service discovery to locate the SPI implementation, without ever loading any OSGi specific class. On the other hand, in an OSGi environment, it will query the registry maintained by the registry bundle to locate the provider. The reference to the registry is injected into the `ProviderLocator` class using a bundle activator.
- Finally, it should also be noted that the API bundle is configured with `singleton=true`. There is indeed no meaningful way how providers could be matched with different versions of the same API bundle.

This is an example of a particularly elegant way to satisfy Requirement 1, Requirement 2 and Requirement 3, especially because it relies on the same metadata (the `META-INF/services/javax.xml.bind.JAXBContext` resources) in OSGi and non OSGi environments.

Obviously, Axiom could reuse the registry and locator bundles developed by Geronimo. This however would contradict Requirement 7. In addition, for Axiom there is no requirement to strictly follow the JDK

1.3 service discovery algorithm. Therefore Axiom should reuse the pattern developed by Geronimo, but not the actual implementation.

New abstract APIs

Application code rarely uses DOOM as the default Axiom implementation. Several downstream projects (e.g. the Axis2/Rampart combination) use both the default (LLOM) implementation and DOOM. They select the implementation based on the particular context. As of Axiom 1.2.12, the only way to create an object model instance with the DOOM implementation is to use the `DOOMAbstractFactory` API or to instantiate one of the factory classes (`OMDOMMetaFactory`, `OMDOMFactory` or one of the subclasses of `OMSOAPFactory`). All these classes are part of the `axiom-dom` artifact. This is clearly in contradiction with Requirement 4 and Requirement 5.

To overcome this problem the Axiom API must be enhanced to make it possible to select an Axiom implementation based on capabilities/features requested by the application code. E.g. in the case of DOOM, the application code would request a factory that implements the DOM API. It is then up to the Axiom API classes to locate an appropriate implementation, which may be DOOM or another drop-in replacement, as per Requirement 5.

If multiple Axiom implementations are available (on the class path in non OSGi environment or deployed as bundles in an OSGi environment), then the Axiom API must also be able to select an appropriate default implementation if no specific feature is requested by the application code. This can be easily implemented by defining a special feature called "default" that would be declared by any Axiom implementation that is suitable as a default implementation.



DOOM is generally not considered suitable as a default implementation because it doesn't implement the complete Axiom API (e.g. it doesn't support `OMSourcedElement`). In addition, in earlier versions of Axiom, the factory classes for DOOM were not stateless (see AXIOM-412 [<https://issues.apache.org/jira/browse/AXIOM-412>]).

Finally, to make the selection algorithm deterministic, there should also be a concept of priority: if multiple Axiom implementations are found for the same feature, then the Axiom API would select the one with the highest priority.

This leads to the following design:

1. Every Axiom implementation declares a set of features that it supports. A feature is simply identified by a string. Two features are predefined by the Axiom API:
 - `default`: indicates that the implementation is a complete implementation of the Axiom API and may be used as a default implementation.
 - `dom`: indicates that the implementation supports DOM in addition to the Axiom API.

For every feature it declares, the Axiom implementation specifies a priority, which is a positive integer.

2. The relevant Axiom APIs are enhanced so that they take an optional argument specifying the feature requested by the application code. If no explicit feature is requested, then Axiom will use the `default` feature.
3. To determine the `OMMetaFactory` to be used, Axiom locates the implementations declaring the requested feature and selects the one that has the highest priority for that feature.

A remaining question is how the implementation declares the feature/priority information. There are two options:

- Add a method to `OMMetaFactory` that allows the Axiom API to query the feature/priority information from the implementation (i.e. the features and priorities are hardcoded in the implementation).
- Let the implementation provide this information declaratively in its metadata (either in the manifest or in a separate resource with a well defined name). Note that in a non OSGi environment, such a metadata resource must be used anyway to enable the Axiom API to locate the `OMMetaFactory` implementations. Therefore this would be a natural place to declare the features as well.

The second option has the advantage to make it easier for users to debug and tweak the implementation discovery process (e.g. there may be a need to customize the features and priorities declared by the different implementations to ensure that the right implementation is chosen in a particular use case).

This leads to the following design decision: the features and priorities (together with the class name of the `OMMetaFactory` implementation) will be defined in an XML descriptor with resource name `META-INF/axiom.xml`. The format of that descriptor must take into account that a single JAR may contain several Axiom implementations (e.g. if the JAR is an uber-JAR repackaged from the standard Axiom JARs).

Common implementation classes

Obviously the LLOM and DOOM implementations share some amount of common code. Historically, implementation classes reusable between LLOM and DOOM were placed in `axiom-api`. This however tends to blur the distinction between the public API and implementation classes. Starting with Axiom 1.2.13 such classes are placed into a separate module called `axiom-common-impl`. However, `axiom-common-impl` cannot simply be a dependency of `axiom-impl` and `axiom-dom`. The reason is that in an OSGi environment, the `axiom-common-impl` bundle would have to export these shared classes, which is in contradiction with Requirement 4. Therefore the code from `axiom-common-impl` needs to be packaged into `axiom-impl` and `axiom-dom` by the build process so that the `axiom-common-impl` artifact is not required at runtime. Requirement 1 forbids using embedded JARs to achieve this. Instead `maven-shade-plugin` is used to include the classes from `axiom-common-impl` into `axiom-impl` and `axiom-dom` (and to modify the POMs to remove the dependencies on `axiom-common-impl`).

This raises the question whether `maven-shade-plugin` should be configured to simply copy the classes or to relocate them (i.e. to change their package names). There are a couple of arguments in favor of relocating them:

- According to Requirement 1, the Axiom artifacts should be usable both as normal JARs and as OSGi bundles. Obviously the expectation is that from the point of view of application code, they should work in the same in OSGi and non OSGi environments. Relocation is required if one wants to strictly satisfy this requirement even if different versions of `axiom-impl` and `axiom-dom` are mixed. Since the container creates separate class loaders for the `axiom-impl` and `axiom-dom` bundles, it is always possible to do that in an OSGi environment: even if the shared classes included in `axiom-impl` and `axiom-dom` are not relocated, but have the same names, this will not result in conflicts. The situation is different in a non OSGi environment where the classes in `axiom-impl` and `axiom-dom` are loaded by the same class loader. If the shared classes are not relocated, then there may be a conflict if the versions don't match.

However, in practice it is unlikely that there are valid use case where one would use `axiom-impl` and `axiom-dom` artifacts from different Axiom versions.

- Relocation allows to preserve compatibility when duplicate code from `axiom-impl` and `axiom-dom` is merged and moved to `axiom-common-impl`. The `OMNamespaceImpl`, `OMNavigator` and `OMStAXWrapper` classes from `axiom-impl` and the `NamespaceImpl`, `DOMNavigator` and `DOMStAXWrapper` classes from `axiom-dom` that existed in earlier versions of Axiom are examples

of this. The classes in `axiom-dom` were almost identical to the corresponding classes in `axiom-impl`. These classes have been merged and moved to `axiom-common-impl`. Relocation then allows them to retain their original name (including the original package name) in the `axiom-impl` and `axiom-dom` artifacts.

However, this is only a concern if one wants to preserve compatibility with existing code that directly uses these implementation specific classes (which is something that is strongly discouraged). One example where this was relevant was the SAAJ implementation in Axis2 which used to be very strongly coupled to the DOOM implementation. This however has been fixed now.

Using relocation also has some serious disadvantages:

- Stack traces may contain class names that don't match class names in the Axiom source code, making debugging harder.
- Axiom now uses JaCoCo to produce code coverage reports. However these reports are incomplete if relocation is used. This doesn't affect test cases executed in the `axiom-impl` and `axiom-dom` modules (because they are executed with the original classes), but tests in separate modules (such as integration tests). There are actually two issues:
 - For the relocated classes, JaCoCo is unable to find the corresponding source code. This means that the reported code coverage is inaccurate for classes in `axiom-common-impl`.
 - Relocation not only modifies the classes in `axiom-common-impl`, but also the classes in `axiom-impl` and `axiom-dom` that use them. JaCoCo detects this [<https://github.com/jacoco/jacoco/issues/51>] and excludes the data from the coverage analysis. This means that the reported code coverage will also be inaccurate for classes in `axiom-impl` and `axiom-dom`.

In Axiom 1.2.14 relocation was used, but this has been changed in Axiom 1.2.15 because the disadvantages outweigh the advantages.

LifecycleManager design (Axiom 1.3)

The `LifecycleManager` API is used by the MIME handling code in Axiom to manage the temporary files that are used to buffer the content of attachment parts. The `LifecycleManager` implementation is responsible to track the temporary files that have been created and to ensure that they are deleted when they are no longer used. In Axiom 1.2.x, this API has multiple issues and a redesign is required for Axiom 1.3.

Issues with the LifecycleManager API in Axiom 1.2.x

1. Temporary files that are not cleaned up explicitly by application code will only be removed when the JVM stops (`LifecycleManagerImpl` registers a shutdown hook and maintains a list of files that need to be deleted when the JVM exits). This means that temporary files may pile up, causing the file system to fill.
2. `LifecycleManager` also has a method `deleteOnTimeInterval` that deletes a file after some specified time interval. However, the implementation creates a new thread for each invocation of that method, which is generally not acceptable in high performance use cases.
3. One of the stated design goals (see AXIOM-192 [<https://issues.apache.org/jira/browse/AXIOM-192>]) of the `LifecycleManager` API was to wrap the files in `FileAccessor` objects to “keep track of activity that occurs on the files”. However, as pointed out in AXIOM-185 [<https://issues.apache.org/jira/browse/AXIOM-185>], since `FileAccessor` has a method that returns the corresponding `File` object, this goal has not been reached.

4. As noted in AXIOM-382 [<https://issues.apache.org/jira/browse/AXIOM-382>], the fact that `LifecycleManagerImpl` registers a shutdown hook which is never unregistered causes a class loader leak in J2EE environments.
 5. In an attempt to work around the issues related to `LifecycleManager` (in particular the first item above), AXIOM-185 [<https://issues.apache.org/jira/browse/AXIOM-185>] introduced another class called `AttachmentCacheMonitor` that implements a timer based mechanism to clean up temporary files. However, this change causes other issues:
 - The existence of this API has a negative impact on Axiom's architectural integrity because it has functionality that overlaps with `LifecycleManager`. This means that we now have two completely separate APIs that are expected to serve the same purpose, but none of them addresses the problem properly.
 - `AttachmentCacheMonitor` automatically creates a timer, but there is no way to stop that timer. This means that this API can only be used if Axiom is integrated into the container, but not when it is deployed with an application.
- Fortunately, that change was only meant as a workaround to solve a particular issue in WebSphere (see APAR PK91497 [<http://www-01.ibm.com/support/docview.wss?rs=180&uid=swglPK91497>]), and once the `LifecycleManager` API is redesigned to solve that issue, `AttachmentCacheMonitor` no longer has a reason to exist.
6. `LifecycleManager` is an abstract API (interface), but refers to `FileAccessor` which is placed in an impl package.
 7. `FileAccessor` uses the `MessagingException` class from JavaMail, although Axiom no longer relies on this API to parse or create MIME messages.

Cleanup strategy for temporary files

As pointed out in the previous section, one of the primary problems with the `LifecycleManager` API in Axiom 1.2.x is that temporary files that are not cleaned up explicitly by application code (e.g. using the `purgeDataSource` method defined by `DataHandlerExt`) are only removed when the JVM exits. A timer based strategy that deletes temporary file after a given time interval (as proposed by `AttachmentCacheMonitor`) is not reliable because in some use cases, application code may keep a reference to the attachment part for a long time before accessing it again.

The only reliable strategy is to take advantage of finalization, i.e. to rely on the garbage collector to trigger the deletion of temporary files that are no longer used. For this to work the design of the API (and its default implementation) must satisfy the following two conditions:

1. All access to the underlying file must be strictly encapsulated, so that the file is only accessible as long as there is a strong reference to the object that encapsulates the file access. This is necessary to ensure that the file can be safely deleted once there is no longer a strong reference and the object is garbage collected.
2. Java guarantees that the finalizer is invoked before the instance is garbage collected. However, instances are not necessarily garbage collected before the JVM exits, and in that case the finalizer is never invoked. Therefore, the implementation must delete all existing temporary files when the JVM exits. The API design should also take into account that some implementations of the `LifecycleManager` API may want to trigger this cleanup before the JVM exits, e.g. when the J2EE application in which Axiom is deployed is stopped.

The first condition can be satisfied by redesigning the `FileAccessor` such that it never leaks the name of the file it represents (neither as a `String` nor a `File` object). This in turn

means that the `CachedFileDataSource` class must be removed from the Axiom API. In addition, the `getInputStream` method defined by `FileAccessor` must no longer return a simple `FileInputStream` instance, but must use a wrapper that keeps a strong reference to the `FileAccessor`, so that the `FileAccessor` can't be garbage collected while the input stream is still in use.

To satisfy the second condition, one may want to use `File#deleteOnExit`. However, this method causes a native memory leak, especially when used with temporary files, which are expected to have unique names (see bug 4513817 [http://bugs.sun.com/bugdatabase/view_bug.do?bug_id=4513817]). Therefore this can only be implemented using a shutdown hook. However, a shutdown hook will cause a class loader leak if it is used improperly, e.g. if it is registered by an application deployed into a J2EE container and not unregistered when that application is stopped. For this particular case, it is possible to create a special `LifecycleManager` implementation, but for this to work, the lifecycle of this type of `LifecycleManager` must be bound to the lifecycle of the application, e.g. using a `ServletContextListener`. This is not always possible and this approach is therefore not suitable for the default `LifecycleManager` implementation.

To avoid the class loader leak, the default `LifecycleManager` implementation should register the shutdown hook when the first temporary file is registered and automatically unregister the shutdown hook again when there are no more temporary files. This implies that the shutdown hook is repeatedly registered and unregistered. However, since these are relatively cheap operations¹, this should not be a concern.

An additional complication is that when the shutdown hook is executed, the temporary files may still be in use. This contrasts with the finalizer case where encapsulation guarantees that the file is no longer in use. This situation doesn't cause an issue on Unix platforms (where it is possible to delete a file while it is still open), but needs to be handled properly on Windows. This can only be achieved if the `FileAccessor` keeps track of created streams, so that it can forcibly close the underlying `FileInputStream` objects.

¹Since the JRE typically uses an `IdentityHashMap` to store shutdown hooks, the only overhead is caused by Java 2 security checks and synchronization.

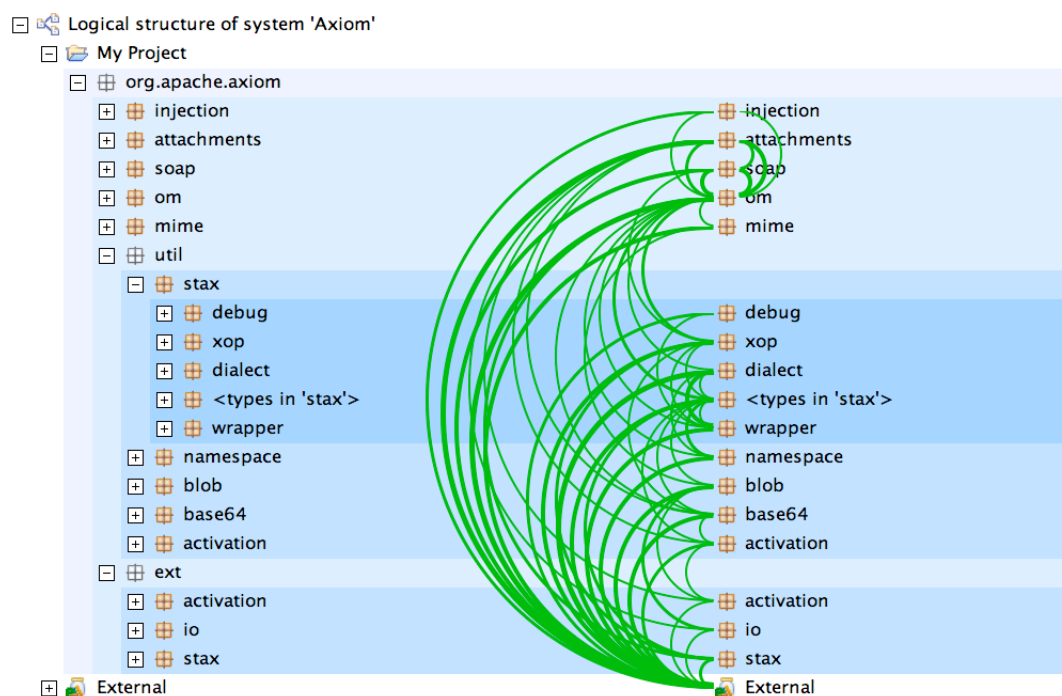
Chapter 3. Release process

Release preparation

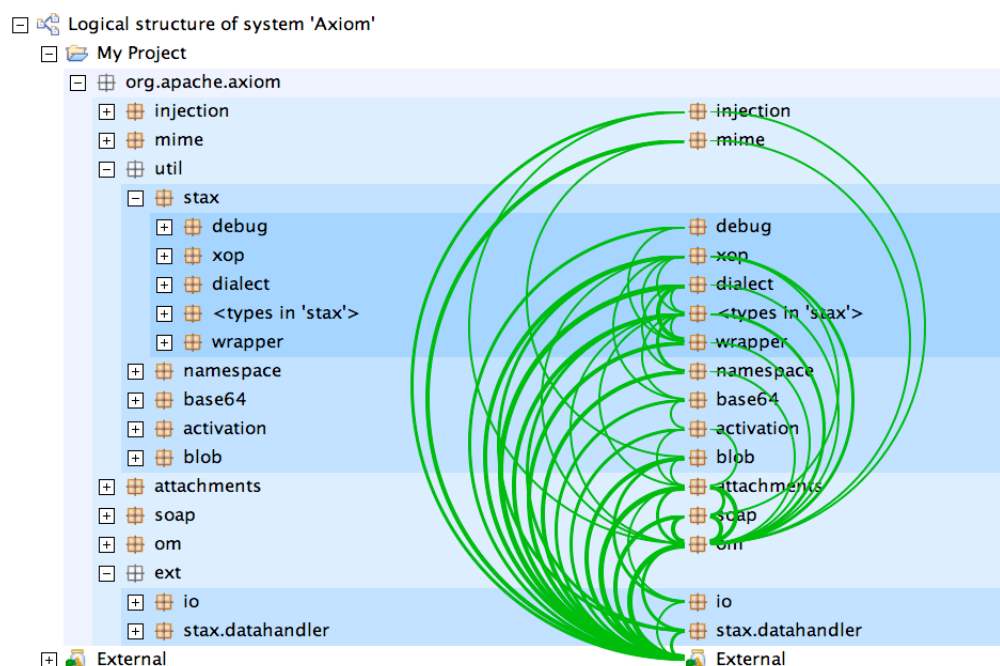
The following items should be checked before starting the release process:

- Check the dependencies between Java packages in the `axiom-api` module. The `org.apache.axiom.util` package (including its subpackages) is specified to contain utility classes that don't depend on higher level APIs. More precisely, `org.apache.axiom.util` should only have dependencies on `org.apache.axiom.ext`, but not e.g. on `org.apache.axiom.om`. SonarJ [<http://www.hello2morrow.com/products/sonarj>] can be used to check these dependencies. The following figure shows the expected structure:

Figure 3.1. Package dependencies for r944680



In contrast, the following figure shows an earlier trunk version of `axiom-api` with incorrect layering and cyclic dependencies involving `org.apache.axiom.util`:

Figure 3.2. Package dependencies for r939984

The check can also be done using `jdepend-maven-plugin` [<http://mojo.codehaus.org/jdepend-maven-plugin/>]. To do this, execute the following command in the `axiom-api` module:

```
mvn jdepend:generate
```

Then open `target/site/jdepend-report.html` and go to the "Cycles" section. The report should not show any package cycles involving `org.apache.axiom.mime`, `org.apache.axiom.util` and `org.apache.axiom.ext`.

- Check that the generated Javadoc contains the appropriate set of packages, i.e. only the public API. This excludes classes from `axiom-impl` and `axiom-dom` as well as classes related to unit tests.
- Check that all dependencies and plugins are available from standard repositories. To do this, clean the local repository and execute **`mvn clean install`** followed by **`mvn site`**.
- Check that the set of license files in the `legal` directory is complete and accurate (by checking that in the binary distribution, there is a license file for every third party JAR in the `lib` folder).
- Check that the Maven site conforms to the latest version of the Apache Project Branding Guidelines [<http://apache.org/foundation/marks/pmcs>].
- Check that the `apache-release` profile can be executed properly. To do this, issue the following command:

```
mvn clean install -Papache-release -DskipTests=true
```

You may also execute a dry run of the release process:

```
mvn release:prepare -DdryRun=true
```

After this, you need to clean up using the following command:

```
mvn release:clean
```

- Check that the Maven site can be generated and deployed successfully, and that it has the expected content.
- Complete the release note (`src/site/markdown/release-notes/version.md`). It should include a description of the major changes in the release as well as a list of resolved JIRA issues.

Prerequisites

The following things are required to perform the actual release:

- A PGP key that conforms to the requirement for Apache release signing [<http://www.apache.org/dev/release-signing.html>]. To make the release process easier, the passphrase for the code signing key should be configured in `${user.home}/.m2/settings.xml`:

```
<settings>
...
<profiles>
  <profile>
    <id>apache-release</id>
    <properties>
      <gpg.passphrase><!-- KEY PASSPHRASE --></gpg.passphrase>
    </properties>
  </profile>
</profiles>
...
</settings>
```

- The release process uses a Nexus staging repository. Every committer should have access to the corresponding staging profile in Nexus. To validate this, login to `repository.apache.org` and check that you can see the `org.apache.ws` staging profile. The credentials used to deploy to Nexus should be added to `settings.xml`:

```
<servers>
...
<server>
  <id>apache.releases.https</id>
  <username><!-- ASF username --></username>
  <password><!-- ASF LDAP password --></password>
</server>
...
</servers>
```

Release

In order to prepare the release artifacts for vote, execute the following steps:

1. Start the release process with the following command:

```
mvn release:prepare
```

When asked for the "SCM release tag or label", keep the default value (`x.y.z`).

The above command will create a tag in Subversion and increment the version number of the trunk to the next development version. It will also create a `release.properties` file that will be used in the next step.

2. Perform the release using the following command:

```
mvn release:perform
```

This will upload the release artifacts to the Nexus staging repository.

3. Log in to the Nexus repository (<https://repository.apache.org/>) and close the staging repository. The name of the staging profile is `org.apache.ws`. See <http://maven.apache.org/developers/release/apache-release.html> for a more thorough description of this step.
4. Execute the `target/checkout/etc/dist.py` script to upload the source and binary distributions to the development area of the <https://dist.apache.org/repos/dist/> repository.

If not yet done, export your public key and append it to <https://dist.apache.org/repos/dist/release/ws/axiom/KEYS>. The command to export a public key is as follows:

```
gpg --armor --export key_id
```

5. Delete <https://svn.apache.org/repos/asf/webservices/website/axiom-staging/> if it exists. Create a new staging area for the site:

```
svn copy \  
https://svn.apache.org/repos/asf/webservices/website/axiom \  
https://svn.apache.org/repos/asf/webservices/website/axiom-staging
```



This step can be skipped if the staging area already exists and is in a state where it can cleanly be merged.

6. Stage the site as described here [http://ws.apache.org/dev.html#Republishing_the_site_for_a_subproject]. Note that the commands must be executed in the `target/checkout` directory.
7. Start the release vote by sending a mail to `dev@ws.apache.org`. The mail should mention the following things:
 - The list of issues solved in the release (by linking to the relevant JIRA view).
 - The location of the Nexus staging repository.
 - The link to the source and binary distributions: `https://dist.apache.org/repos/dist/dev/ws/axiom/version`.
 - A link to the preview of the Maven site: `http://ws.apache.org/axiom-staging/`.

If the vote passes, execute the following steps:

1. Promote the artifacts in the staging repository. See <http://maven.apache.org/developers/release/apache-release.html> for detailed instructions for this step.
2. Publish the distributions:

```
svn mv https://dist.apache.org/repos/dist/dev/ws/axiom/version \
      https://dist.apache.org/repos/dist/release/ws/axiom/
```

version is the release version, e.g. 1.2.9.

3. Publish the site:

```
svn co https://svn.apache.org/repos/asf/webservices/website/axiom axiom-site
cd axiom-site/
svn merge https://svn.apache.org/repos/asf/webservices/website/axiom-staging
svn commit
```

It may take several hours before all the updates have been synchronized to the relevant ASF systems. Before proceeding, check that

- the Maven artifacts for the release are available from the Maven central repository;
- the Maven site has been synchronized to <http://ws.apache.org/axiom/>;
- the binary and source distributions can be downloaded from <http://ws.apache.org/axiom/download.cgi>.

Once everything is in place, send announcements to users@ws.apache.org and announce@apache.org. Since the two lists have different conventions, audiences and moderation policies, to send the announcement separately to the two lists.

Sample announcement:

Apache Axiom Team is pleased to announce the release of Axiom x.y.z. The release is available for download at:

<http://ws.apache.org/axiom/download.cgi>

Apache Axiom is a StAX-based, XML Infoset compliant object model which supports on-demand building of the object tree. It supports a novel "pull-through" model which allows one to turn off the tree building and directly access the underlying pull event stream. It also has built in support for XML Optimized Packaging (XOP) and MTOM, the combination of which allows XML to carry binary data efficiently and in a transparent manner. The combination of these is an easy to use API with a very high performant architecture!

Developed as part of Apache Axis2, Apache Axiom is the core of Apache Axis2. However, it is a pure standalone XML Infoset model with novel features and can be used independently of Apache Axis2.

Highlights in this release:

- ...
- ...

Resolved JIRA issues:

- [WSCOMMONS-513] Behavior of `insertSiblingAfter` and `insertSiblingBefore` is not well defined for orphan nodes
- [WSCOMMONS-488] The sequence of events produced by `OMStAXWrapper` with `inlineMTOM=false` is inconsistent

For `users@ws.apache.org`, the subject (“Axiom x.y.z released”) should be prefixed with “[ANN][Axiom]”, while for `announce@apache.org` “[ANN]” is enough. Note that mail to `announce@apache.org` must be sent from an `apache.org` address.

Post-release actions

- Update the DOAP file (see `etc/axiom.rdf`) and add a new entry for the release.
- Update the status of the release version in the AXIOM project in JIRA.
- Remove archived releases from <https://dist.apache.org/repos/dist/release/ws/axiom/>.
- Delete <https://svn.apache.org/repos/asf/webservices/website/axiom-staging/>.



This step is optional. The staging area may be reused during the next release. It may also be used to publish a snapshot version of the site.

References

The following documents are useful when preparing and executing the release:

- ASF Source Header and Copyright Notice Policy [<http://www.apache.org/legal/src-headers.html>]
- Apache Project Branding Guidelines [<http://apache.org/foundation/marks/pmcs>]
- DOAP Files [<http://projects.apache.org/doap.html>]
- Publishing Releases [<http://www.apache.org/dev/release-publishing.html>]

Appendix A. Appendix

Installing IBM's JDK on Debian Linux

1. Make sure that `fakeroot` and `java-package` are installed:

```
# apt-get install fakeroot java-package
```

2. Download the `.tgz` version of the JDK from <http://www.ibm.com/developerworks/java/jdk/linux/download.html>.
3. Edit `/usr/share/java-package/ibm-j2sdk.sh` and (if necessary) add an entry for the particular version of the IBM JDK downloaded in the previous step.
4. Build a Debian package from the tarball:

```
$ fakeroot make-jpkg xxxx.tgz
```

5. Install the Debian package.