Axiom Developer Guide

Axiom Developer Guide

1.2.9

Table of Contents

List of Figures

Chapter 1. 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 yet[1]. 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`.

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.MXPa`
	`javax.xml.stream.XMLOutputFactory=com.bea.xml.stream.XML`

[1] See WSCOMMONS-419 [https://issues.apache.org/jira/browse/WSCOMMONS-419].

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. 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. Release process

Release preparation

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

- Check for the latest Apache parent POM version (artifact `org.apache:apache`) and if necessary, change the parent of the Axiom root POM.

- 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 2.1. 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 2.2. Package dependencies for r939984

- Check that the generated Javadoc contains the appropriate set of packages. In particular, unit test related classes should be excluded.
- 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.
- 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 -Dmaven.test.skip=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
  ```

- Prepare the release note. This should include a description of the major changes in the release as well as a list of resolved JIRA issues. Note that both `index.apt` and `RELEASE-NOTE.txt` need to be updated.
- Check the `download.xml` file for releases that are no longer available from the mirrors and change the corresponding entries so that they point to the archives.
- Preview and validate the changes that will be done by the release plugin to the POM files. In order to do this, execute the following command:

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

  Next, compare the `pom.xml.tag` files to the corresponding `pom.xml` files:

  ```
  for pom in $(find . -name "pom.xml"); do diff $pom $pom.tag; done
  ```

  The differences should be limited to `version` and `scm` tags. If necessary, change the original POM files to avoid spurious changes. After that, clean up using:

  ```
  mvn release:clean
  ```

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> <!-- YOUR KEY PASSPHRASE --> </gpg.passphrase>
        </properties>
      </profile>
    </profiles>
    ...
  </settings>
```

Release

1. Add an entry for the release to the `download.xml` file.
2. Start the release process with the following command:

   ```
   mvn release:prepare
   ```

   When asked for the "SCM release tag or label", override the default value (`axiom-x.y.z`) by entering a tag in the form `x.y.z`, which is compatible with the tag names used for previous releases.

   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.
3. Perform the release using the following command:

   ```
   mvn release:perform
   ```

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

To be continued.

Post-release actions

- Update the DOAP file (see `etc/axiom.rdf`) and add a new entry for the release.

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]

Chapter 3. The StAX specification

The StAX specification comprises two parts: a specification document titled “Streaming API For XML JSR-173 Specification” and a Javadoc describing the API. Both can be downloaded from the JSR-173 page [http://jcp.org/en/jsr/detail?id=173]. Since StAX is part of Java 6, the Javadocs can also be viewed online [http://java.sun.com/javase/6/docs/api/javax/xml/stream/package-summary.html].

Semantics of the `setPrefix` method

Probably one of the more obscure parts of the StAX specifications is the meaning of the `setPrefix`[1] method defined by `XMLStreamWriter`. To understand how this method works, it is necessary to look at different parts of the specification:

- The Javadoc of the `setPrefix` method.
- The table shown in the Javadoc of the `XMLStreamWriter` class in Java 6[2].
- Section 5.2.2, “Binding Prefixes” of the specification.
- The example shown in section 5.3.2, “XMLStreamWriter” of the specification.

In addition, it is important to note the following facts:

- The terms *defaulting prefixes* used in section 5.2.2 of the specification and *namespace repairing* used in the Javadocs of `XMLStreamWriter` are synonyms.
- The methods writing namespace qualified information items, i.e. `writeStartElement`, `writeEmptyElement` and `writeAttribute` all come in two variants: one that takes a namespace URI and a prefix as arguments and one that only takes a namespace URI, but no prefix.

The purpose of the `setPrefix` method is simply to define the prefixes that will be used by the variants of the `writeStartElement`, `writeEmptyElement` and `writeAttribute` methods that only take a namespace URI (and the local name). This becomes clear by looking at the table in the `XMLStreamWriter` Javadoc. Note that a call to `setPrefix` doesn't cause any output and it is still necessary to use `writeNamespace` to actually write the necessary namespace declarations. Otherwise the produced document will not be well formed with respect to namespaces.

The Javadoc of the `setPrefix` method also clearly defines the scope of the prefix bindings defined using that method: a prefix bound using `setPrefix` remains valid till the invocation of `writeEndElement` corresponding to the last invocation of `writeStartElement`. While not explicitly mentioned in the specifications, it is clear that a prefix binding may be masked by another binding for the same prefix defined in a nested element.

An aspect that may cause confusion is the fact that in the example shown in section 5.3.2 of the specifications, the calls to `setPrefix` (and `setDefaultNamespace`) all appear immediately before a call to `writeStartElement` or `writeEmptyElement`. This may lead people to incorrectly believe that a prefix binding defined using `setPrefix` only applies to the next element written[3]. This interpretation is clearly in contradiction with the `setPrefix` Javadoc, unless one assumes that “the current START_ELEMENT / END_ELEMENT pair” means the element opened by a call to

[1] For simplicity, we only discuss `setPrefix` here. The same remarks also apply to `setDefaultNamespace`.

[3] Another factor that contributes to the confusion is that in SAX, prefix mappings are always generated before the corresponding `startElement` event and that their scope ends with the corresponding `endElement` event. This is so because the `ContentHandler` interface specifies that “all `startPrefixMapping` events will occur immediately before the corresponding `startElement` event, and all `endPrefixMapping` events will occur immediately after the corresponding `endElement` event”.

`writeStartElement` immediately following the call to `setPrefix`. This however would be a very arbitrary interpretation of the Javadoc[4].

The correctness of the comments in the previous paragraph can be checked using the following code snippet:

```
XMLOutputFactory f = XMLOutputFactory.newInstance();
XMLStreamWriter writer = f.createXMLStreamWriter(System.out);
writer.writeStartElement("root");
writer.setPrefix("p", "urn:ns1");
writer.writeEmptyElement("urn:ns1", "element1");
writer.writeEmptyElement("urn:ns1", "element2");
writer.writeEndElement();
writer.flush();
writer.close();
```

This produces the following output[5]:

```
<root><p:element1/><p:element2/></root>
```

Since the code doesn't call `writeNamespace`, the output is obviously not well formed with respect to namespaces, but it also clearly shows that the scope of the prefix binding for `p` extends to the end of the `root` element and is not limited to `element1`.

To avoid unexpected results and keep the code maintainable, it is in general advisable to keep the calls to `setPrefix` and `writeNamespace` aligned, i.e. to make sure that the scope (in `XMLStreamWriter`) of the prefix binding defined by `setPrefix` is compatible with the scope (in the produced document) of the namespace declaration written by the corresponding call to `writeNamespace`. This makes it necessary to write code like this:

```
writer.writeStartElement("p", "element1", "urn:ns1");
writer.setPrefix("p", "urn:ns1");
writer.writeNamespace("p", "urn:ns1");
```

As can be seen from this code snippet, keeping the two scopes in sync makes it necessary to use the `writeStartElement` variant which takes an explicit prefix. Note that this somewhat conflicts with the purpose of the `setPrefix` method; one may consider this as a flaw in the design of the StAX API.

The three `XMLStreamWriter` usage patterns

Drawing the conclusions from the previous section and taking into account that `XMLStreamWriter` also has a “namespace repairing” mode, one can see that there are in fact three different ways to use `XMLStreamWriter`. These usage patterns correspond to the three bullets in section 5.2.2 of the StAX specification[6]:

1. In the “namespace repairing” mode (enabled by the `javax.xml.stream.isRepairingNamespaces` property), the writer takes care of all

[4]Early versions of XL XP-J were based on this interpretation of the specifications, but this has been corrected. Versions conforming to the specifications support a special property called `javax.xml.stream.XMLStreamWriter.isSetPrefixBeforeStartElement`, which always returns `Boolean.FALSE`. This allows to easily distinguish the non conforming versions from the newer versions. Note that in contrast to what the usage of the reserved `javax.xml.stream` prefix suggests, this is a vendor specific property that is not supported by other implementations.

[5]This has been tested with Woodstox 3.2.9, SJSXP 1.0.1 and version 1.2.0 of the reference implementation.

[6]The content of this section is largely based on a reply posted by Tatu Saloranta on the Axiom mailing list [http://markmail.org/message/olsdl3p3gciqqeob]. Tatu is the main developer of the Woodstox project.

namespace bindings and declarations, with minimal help from the calling code. This will always produce output that is well-formed with respect to namespaces. On the other hand, this adds some overhead and the result may depend on the particular StAX implementation (though the result produced by different implementations will be equivalent).

In repairing mode the calling code should avoid writing namespaces explicitly and leave that job to the writer. There is also no need to call `setPrefix`, except to suggest a preferred prefix for a namespace URI. All variants of `writeStartElement`, `writeEmptyElement` and `writeAttribute` may be used in this mode, but the implementation can choose whatever prefix mapping it wants, as long as the output results in proper URI mapping for elements and attributes.

2. Only use the variants of the writer methods that take an explicit prefix together with the namespace URI. In this usage pattern, `setPrefix` is not used at all and it is the responsibility of the calling code to keep track of prefix bindings.

   Note that this approach is difficult to implement when different parts of the output document will be produced by different components (or even different libraries). Indeed, when passing the `XMLStreamWriter` from one method or component to the other, it will also be necessary to pass additional information about the prefix mappings in scope at that moment, unless the it is acceptable to let the called method write (potentially redundant) namespace declarations for all namespaces it uses.

3. Use `setPrefix` to keep track of prefix bindings and make sure that the bindings are in sync with the namespace declarations that have been written, i.e. always use `setPrefix` immediately before or immediately after each call to `writeNamespace`. Note that the code is still free to use all variants of `writeStartElement`, `writeEmptyElement` and `writeAttribute`; it only needs to make sure that the usage it makes of these methods is consistent with the prefix bindings in scope.

   The advantage of this approach is that it allows to write modular code: when a method receives an `XMLStreamWriter` object (to write part of the document), it can use the namespace context of that writer (i.e. `getPrefix` and `getNamespaceContext`) to determine which namespace declarations are currently in scope in the output document and to avoid redundant or conflicting namespace declarations. Note that in order to do so, such code will have to check for an existing prefix binding before starting to use a namespace.

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.