

# Versioned XML Processing in a Java Environment

by Sean Barnett  
([Sean.Barnett@funkypeople.biz](mailto:Sean.Barnett@funkypeople.biz))

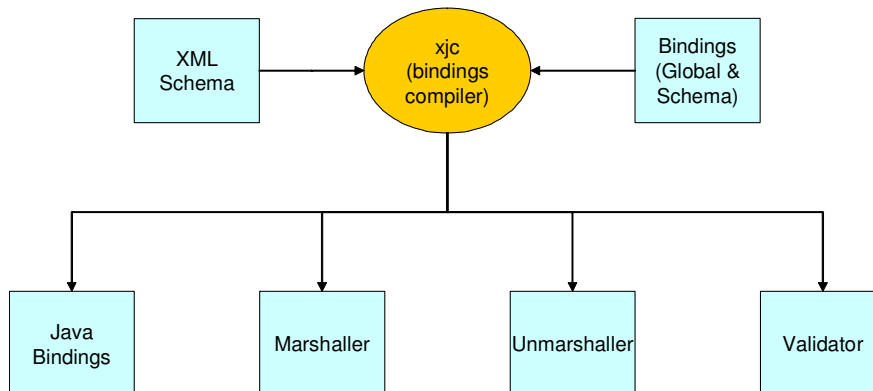
Application software that relies upon XML for persisting or transporting structured data will inevitably be affected by new or changing requirements that mandates the evolution of their XML document type (e.g., DTD or XML Schema). Such evolution of document types poses a number of questions to the application developer, for example:

- how should XML document types be versioned?
- is it feasible to migrate all document instances (e.g., all deployed clients or all saved files) to be compliant with the evolved document type, or is there a requirement to simultaneously support multiple document type versions?
- where simultaneous support for multiple document type versions is required, how should this be implemented in software?

This paper considers the case where it is necessary to simultaneously support multiple versions of an XML Schema in a Java software environment.

## Simple XML Processing

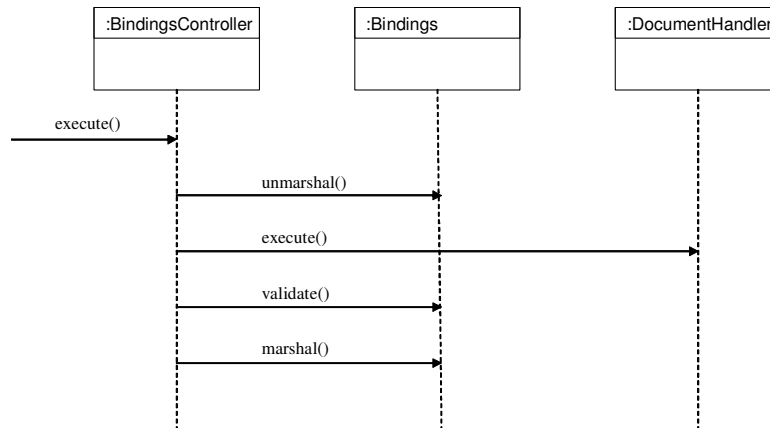
Before considering how a Java application might simultaneously support multiple versions of an XML Schema, it is useful to construct a model of how a Java application processes a single version of an XML Schema. While low level Java XML technologies – such as SAX and DOM parsers – are an option, the Java Architecture for XML Bindings (JAXB) has been developed to bridge the gap between XML and Java and is the obvious and natural choice.



As illustrated above, a central component of JAXB is the xjc compiler – this takes as input an XML Schema and (optionally) a set of bindings directives. The xjc output includes:

- *Java Bindings* these are classes that correspond to the elements and types declared in the XML Schema document. It is useful to consider Java bindings as “value objects”
- *Unmarshaller* responsible for translating between an XML document and the equivalent Java bindings objects. The unmarshaller can be configured as either validating or non-validating – if validating it checks conformance with the XML Schema during the unmarshal operation
- *Marshaller* responsible for translating between a Java bindings object tree and the equivalent XML document
- *Validator* responsible for traversing a Java bindings object tree and ensuring that it conforms to the XML Schema from which it was produced (i.e., would produce valid XML if marshalled)

The sequence diagram below illustrates how these artefacts might be combined using a simple controller class (BindingsController).



The DocumentHandler represents application-specific code that takes as input the Java bindings object tree representing the input XML, and outputs a Java bindings object tree that gets encoded as the XML output. The Bindings interface is used to gather the unmarshal, validate and marshal operations together so they can be implemented by different strategies – as illustrated later in the paper.

It is arguable that the `validate()` call should be omitted from production configurations of the BindingsController as correctly implemented DocumentHandler code should only return valid Java bindings object trees.

## XML Schema Versioning

A further pre-requisite to devising strategies for simultaneous support of multiple XML Schema versions is to consider how an XML Schema should be versioned in the first place. This paper is premised upon the following three strategies being engaged, with the latter having the most significant impact upon implementation:

- when designing XML Schemas some level of flexibility or extensibility might be included, thereby accommodating some level of new and changing requirements without any schema modifications
- where possible the XML Schema should be modified in a manner that is backwardly compatible, for example by the addition of optional attributes and elements
- where a change is required that breaks backwards compatibility, the revised XML schema should have a new target namespace – technically a new XML schema is created, rather than a new version of the old XML schema. Typically the target namespace of the new and old XML Schemas should differ only by a version number

The web site [www.xfront.com](http://www.xfront.com) has a number of articles and white papers on XML Schema best practices, including *Creating Extensible Content Models* and *What's the Best Way to Version XML Schemas?*

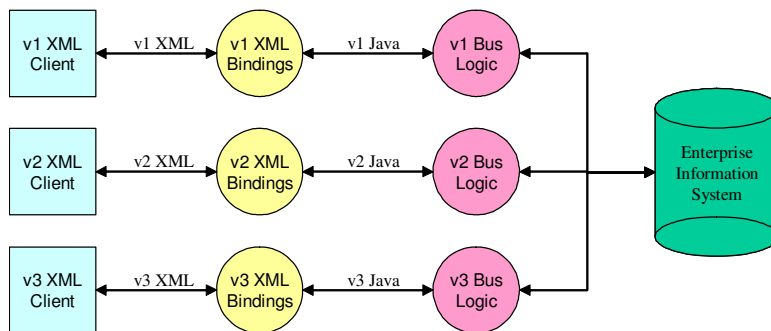
## Versioned XML Processing Design Options

Having determined that the principal strategy for versioning XML Schemas is to use separate namespaces (i.e., declaring a separate XML Schema for each version), consideration can be given to the design options for concurrently supporting multiple XML Schema versions.

### Separate Streams

The most obvious approach to supporting multiple XML Schema versions is to generate separate Java bindings from each version using JAXB. As each XML Schema version has a different namespace, it will result in a discrete set of Java bindings classes and packages that are different and independent from those for the other XML Schema versions. Having multiple, discrete sets of bindings classes in turn requires that there are multiple sets of Java

business logic – while the Java bindings classes for different XML Schema versions will look similar, they will have unrelated Java class and package names. This is illustrated below.



The advantages of this approach are that:

- it is simple to understand and implement
- performance should be optimal and consistent as there is a dedicated pipeline for each XML Schema version
- each XML Schema version can have dedicated Java business logic code that meets any specific business requirements relating to that version (e.g., some functionality is only available when using a specific version of the input XML Schema)

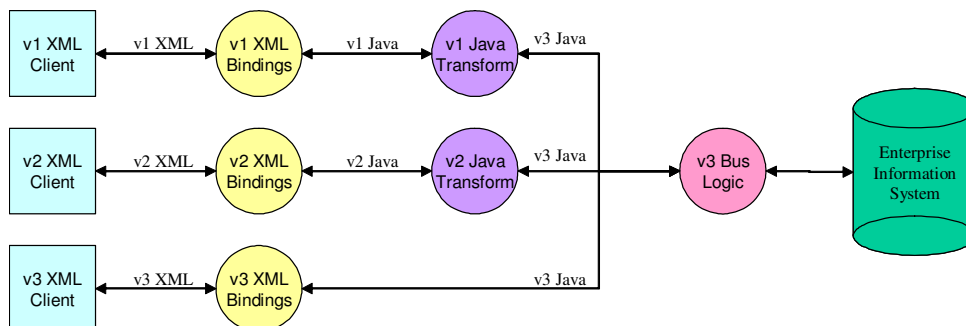
The disadvantages of this approach are that:

- as Java bindings classes and Java business logic classes are required for each XML Schema version, the application will quickly grow in size as it matures
- business logic that is common to many XML Schema versions must be duplicated across multiple sets of Java business logic, and therefore represents a testing and maintenance overhead

The latter disadvantage is considered significant, and therefore this is not the preferred option.

### **Single Stream via Java Transform**

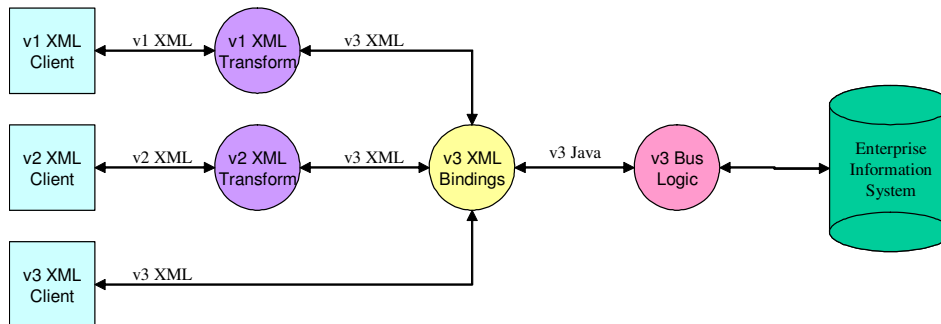
A first, naïve attempt to reduce the duplication of Java business object code is to transform the Java bindings classes from earlier XML Schema versions into those for the current XML Schema version. This is illustrated below.



This strategy is not preferred, nor indeed particularly viable – writing transforms for Java bindings classes is most likely to be a very manual process, and therefore likely to incur a significant maintenance and testing overhead. Also, the application will still grow in size as XML Schema versions are added to the application as multiple sets of Java bindings classes are still required.

## Single Stream via XML Transform

A refinement to the above strategy is to transform the incoming and outgoing XML, rather than the Java bindings classes. This is illustrated below.



The advantages of this approach are:

- as with the previous strategy, there is only one set of Java business logic to be maintained. Further, by using appropriate JAXB schema binding directives all versions of the XML Schema will have the same Java package and class names despite their different target namespaces
- there is only one set of Java bindings classes required, keeping the size of the application constant (at least from a JAXB perspective)
- transformation of XML is supported by robust and mature technology (XSLT), with a convenient scripting language for defining transformations (i.e., no Java coding required)

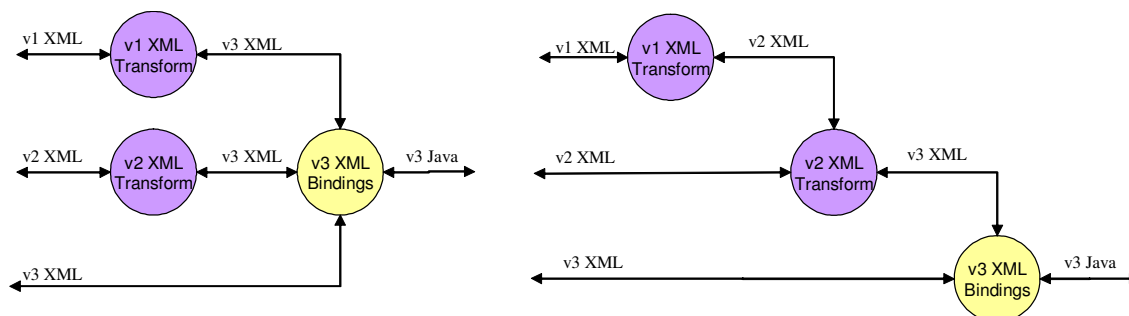
The principal disadvantages of this approach are that:

- all XML input streams (excluding the current version) do not have a dedicated, direct path to the business logic – they must incur the performance overhead of an XML transform
- a number of XML transforms must be maintained, and this number will grow as more versions of the XML Schema are defined. This is explored under the heading below

This is the preferred option, and the one that is implemented in the sample source code.

## Cumulative versus Cascading XML Transforms

Assuming that the Java business logic is coded against the Java bindings objects for the most recent XML Schema version, the preferred option of *Single Stream via XML Transform* requires that an XML transform exists to translate between each previous XML Schema version and the most recent version (e.g., supporting 20 XML Schema versions requires 19 XML transforms). When a new XML Schema version is defined, each of the existing XML translations must be modified and a new XML transform written. Quite a bit of work!



An alternative approach, illustrated above (right hand diagram), is to use cascading transforms. Under this approach each XML transform translates only from one XML Schema version to its successor: to get from any XML Schema

version to the current version involves applying a number of transforms in sequence, each one moving the XML forward one XML Schema version. This approach greatly reduces the overheads of maintaining multiple XML Schemas, with each new XML Schema version requiring only one new XML transform (i.e., all existing transforms remain unaltered). The downside of using cascading transforms is that instances of “very old” XML Schema versions must pass through many transforms before being processed – this will not only increase roundtrip processing times, but also increase the loading on the CPU and therefore decrease processing capacity.

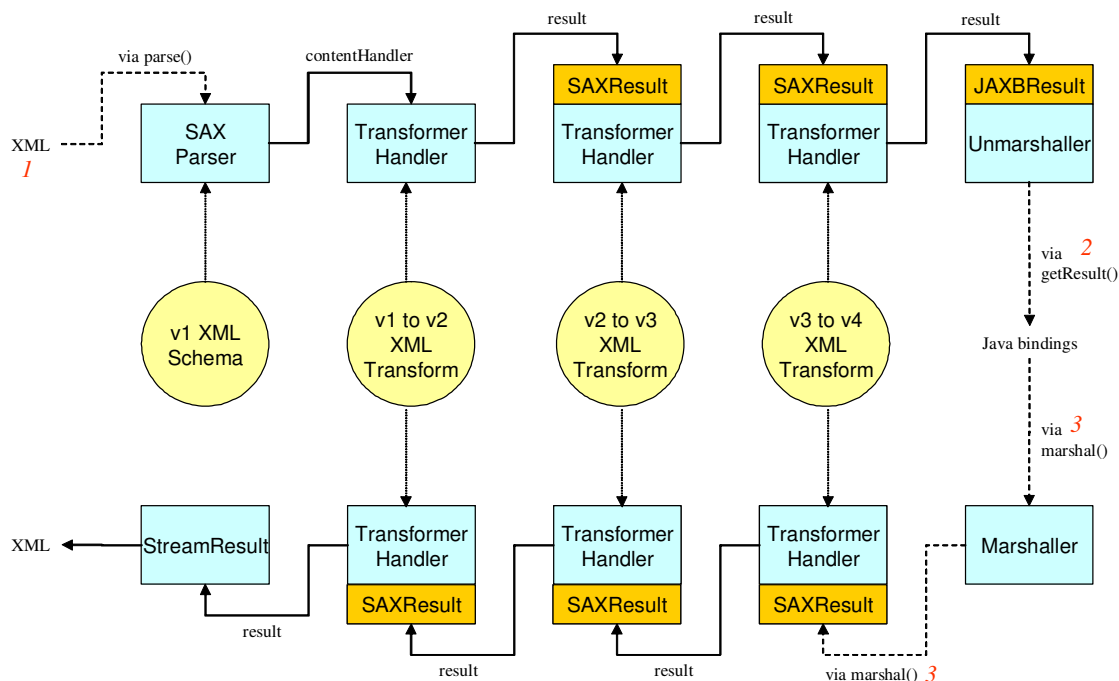
In reality there is no imperative to use either cumulative or cascading transform strategies exclusively - a hybrid strategy will typically be employed involving an intermediate number of transforms. For example, it may be possible to use a single transform for multiple translations (i.e., multiple input XML Schema versions).

Whether using either cascading XML transforms or a hybrid approach, it is necessary for the Java processing strategy selected to accommodate multiple transforms being performed for any given incoming XML Schema version.

## Versioned XML Processing Implementation Options

Having selected the *Single Stream via XML Transform* design strategy for supporting multiple XML Schema versions, thought can be given how this can be implemented in Java, and in a way that is consistent with the JAXB-based *Simple XML Processing* model described previously. In particular, an implementation is sought that provides an implementation of the `unmarshal()` and `marshal()` methods of the Bindings interface, each of which apply one or more XML transforms to an arbitrary version of the input XML Schema.

The various Java XML APIs provide a plethora of tools and techniques for tackling this problem, and attempting to select the most appropriate is a somewhat confusing exercise. Indeed, during the writing of this paper a number of candidate solutions were implemented and assessed. In the final analysis, the arrangement illustrated below proved to be the most elegant, simple and direct identified.



The principal feature of this design is that all components process a stream of SAX events – there is no concrete document created other than during the final stage of marshalling and unmarshalling. In contrast, any alternative design that relied upon intermediate document structures (e.g., DOM trees) would potentially consume greater CPU and memory resources.

The *Java Web Services Tutorial* suggests chaining XML transforms by using the SAXTransformerFactory method `newXMLFilter()`, which returns an instance of `XMLFilter`. The design described above does not use this approach, but rather the method `newTransformerHandler()` is used to obtain a `TransformerHandler` implementation. This was found to be a superior solution:

- `TransformerHandler` provides direct access to the underlying transformer object (via the `getTransformer()` method), `XMLFilter` does not. This permits the transformer to be explicitly configured (e.g., to set its URI resolver or error listener)
- `TransformerHandler` is loosely coupled to the incoming SAX event source via the `ContentHandler` interface that it implements. In contrast, the `XMLFilter` implementation takes direct control of its parent (i.e., the upstream SAX event source), modifying the parent's entity resolver, error handler and content handler – values supplied by the programmer are discarded, being replaced by hidden objects that are not obviously available to the programmer.

The solution uses an explicitly created SAX parser, rather than an implicitly created parser (e.g., hidden within an XML transformer using a `StreamSource`), and this is configured as validating. This allows for the XML transform to introduce invalid values for any field introduced in later versions of the XML Schema; these can act as markers for Java business logic code that the field was not populated at the document's source.

### **Performance and Threading Considerations**

While no scientific benchmarking of the solution has been completed, ad hoc testing has revealed “round trip” (i.e., unmarshalling, processing, validating and marshalling) of 25-40 milliseconds – the lower end of the range was obtained when the SAXON XML transformation engine was substituted for the default engine distributed with the Java Web Services Developer Pack (i.e., Apache's XALAN). This timing is not considered “fast” for a middleware marshalling solution (e.g., compared to CORBA's IIOP), and reinforces that XML is not an appropriate technology for use between tightly coupled systems (e.g., the tiers of an n-tier application), particularly where performance is an important consideration.

Most components of the solution are not thread safe, although they are able to be used multiple times serially (i.e., one use after another). In multi-threaded environments it may not be worth investing significant effort in caching the components (e.g., `BindingsController`) for re-use as performance testing indicates the overhead of creating new components per-input document is reasonable low; the pre-compilation of XML transforms into templates no doubt contributes to this low overhead.

### **Sample Code**

The source code implementing this solution is available at <http://www.funkypeople.biz>.

## **Technical Tricks and Tips**

During the development of this solution, the following tricks and tips were discovered and thought worthy of recording.

### **JAXP**

#### **How do you switch to SAXON rather than XALAN?**

The SAXON transformation engine is significantly faster than XALAN, the default solution packaged with the Java Web Services Developer Pack. To switch to SAXON the distribution should be downloaded from <http://saxon.sourceforge.net>, the JAR file `saxon-7.jar` included in the classpath, and the following system property set:

```
System.setProperty("javax.xml.transform.TransformerFactory",  
    "net.sf.saxon.TransformerFactoryImpl");
```

Note, if using JAXBSource as input to SAXON, an attempt will be made to set the XML namespace features on the JAXBSource's XMLReader. While it is mandatory that all XMLReader implementations support these features, the JAXBSource does not.

### How do you configure a SAX parser to handle XML Schema?

By default SAXParsers are configured to validate against Document Type Definitions (DTDs). To switch the parser to validating against XML Schema use the following code:

```
static final String JAXP_SCHEMA_LANGUAGE =
    "http://java.sun.com/xml/jaxp/properties/schemaLanguage";
static final String W3C_XML_SCHEMA =
    "http://www.w3.org/2001/XMLSchema";
parser.setProperty(JAXP_SCHEMA_LANGUAGE, W3C_XML_SCHEMA);
```

See also question below.

### How do you point the SAX parser at the correct XML Schema?

The initial schema implemented by an XML document instance must be configured as a property of the parser:

```
parser.setProperty(JAXP_SCHEMA_SOURCE, new InputSource(
    getClass().getClassLoader().getResourceAsStream("Old.xsd")));
```

However, all imported XML Schemas are resolved using an EntityResolver, set as follows:

```
parser.getXMLReader().setEntityResolver(this);
```

The EntityResolver tends to have only the second parameter (SystemId) set on a call, and this is the schemaLocation URL referenced in the import element of the XML Schema. Any relative schemaLocation will be converted to a URL by prefixing the protocol "file:" and also the current working directory.