IBM

developerWorks.

XML and Java technologies: Data binding, Part 1: Code generation approaches -- JAXB and more

Generating data classes from DTDs or schemas

Skill Level: Intermediate

Dennis Sosnoski
President
Sosnoski Software Solutions, Inc.

14 Jan 2003

Enterprise Java expert Dennis Sosnoski looks at several XML data binding approaches using code generation from W3C XML Schema or DTD grammars for XML documents. He starts out with the long-awaited JAXB standard now nearing release through the Java Community Process (JCP), then summarizes some of the other frameworks that are currently available. Finally, he discusses how and when you can best apply code generation from a grammar in your applications.

Data binding provides a simple and direct way to use XML in your Java Platform applications. With data binding your application can largely ignore the actual structure of XML documents, instead working directly with the data content of those documents. This isn't suitable for all applications, but it is ideal for the common case of applications that use XML for data exchange.

Other articles in this series

- Part 2: Performance
- Part 3: JiBX architecture
- Part 4: JiBX Usage

Data binding can also provide other benefits beyond programming simplicity. Since it abstracts away many of the document details, data binding usually requires less memory than a document model approach (such as DOM or JDOM) for working with

© Copyright IBM Corporation 2003. All rights reserved.

documents in memory. You'll also find that the data binding approach gives you faster access to data within your program than you would get with a document model, since you don't need to go through the structure of the document to get at the data. Finally, special types of data such as numbers and dates can be converted to internal representations on input, rather than being left as text; this allows your application to work with the data values much more efficiently.

You might be wondering, if data binding is such great stuff, when would you want to use a document model approach instead? Basically there are two main cases:

- When your application is really concerned with the details of the document structure. If you're writing an XML document editor, for instance, you'll want to stick to a document model rather than using data binding.
- When the documents that you're processing don't necessarily follow fixed structures. For example, data binding wouldn't be a good approach for implementing a general XML document database.

Back to binding

Last year, I wrote an article showing how to use the Castor framework for mapped data binding of Java objects to XML documents. I promised a follow-up that would look at code generation approaches including coverage of JAXB, the standard API for data binding in the Java language under development through the Java Community Process (JCP). Right after that earlier article went to publication Sun announced a major change in direction on JAXB (see JAXB rearchitecture). Given this change, I felt it was better to wait until something closer to the final JAXB code was available before writing the follow-up, and I'm happy to finally be able to do so now!

Data binding dictionary

Here's a mini-dictionary of some terms I use in this article:

Grammar is a set of rules defining the structure of a family of XML documents. One type of grammar is the Document Type Definition (DTD) format defined by the XML specification. Another increasingly common type is the W3C XML Schema (Schema) format defined by the XML Schema specification. Grammars define which elements and attributes can be present in a document, and how elements can be nested within the document (often including the order and number of nested elements). Some types of grammars (such as Schema) also go much further, allowing specific data types and even regular expressions to be matched by character data content. In this article I'll often use the term *description* as an informal way to refer to the grammar for a family of documents.

Marshalling is the process of generating an XML representation for an object in memory. As with Java object serialization, the

© Copyright IBM Corporation 2003. All rights reserved.

representation needs to include all dependent objects: objects referenced by our main object, objects referenced by those objects, and so on.

Unmarshalling is the reverse process of marshalling, building an object (and potentially a graph of linked objects) in memory from an XML representation.

In this article I cover five XML data binding frameworks that generate Java language code from XML document grammars: JAXB, Castor, JBind, Quick, and Zeus. These are all freely available, and with the exception of JAXB, are usable in both open source and proprietary projects. The current JAXB reference implementation beta is not licensed for anything other than evaluation use, but that appears likely to change when it becomes a production release. JAXB, Castor, and JBind all provide code generation from Schema descriptions of XML documents, while Quick and Zeus implement code generation from DTD descriptions. Castor and Quick also support mapping existing classes to XML as an alternative to using generated code.

There are advantages and disadvantages to all of these frameworks, so I'll try to point out the best (and worst) features of each along the way. In Part 2, I'll go further by showing you how these frameworks perform on some sample documents, and I'll also explore how existing data binding frameworks may be lacking in features important to many types of applications.

Generating Java language code from a Schema or DTD grammar offers some major advantages over the mapping binding approach I described in the earlier article. Using generated code you can be certain that your data objects are properly linked to the XML document, unlike the mapped binding approach where you need to specify the linkage directly and make sure you've properly covered all the structure variations. When a Schema is used, you can even make use of the type information supplied by the grammar to generate code with appropriate data types.

There are also a few drawbacks to the code generation approach. It creates a tight coupling between your application data structure and the XML document structure. It also may restrict you to working with simple data classes (passive data containers with no associated behavior) rather than true object classes, and may limit your flexibility to apply custom transformations of data in the process of marshalling and unmarshalling. I discuss these tradeoffs between code generation and the mapped binding approach in more depth later in this article (see Mapped versus generated).

Data and code

For the performance tests I'll cover in Part 2, I generated code using each of these data binding frameworks. The documents I used for performance tests contain mock airline flight timetable information. Here's a sample document so you can get a

© Copyright IBM Corporation 2003. All rights reserved.

feeling for the structure:

Listing 1. Sample document

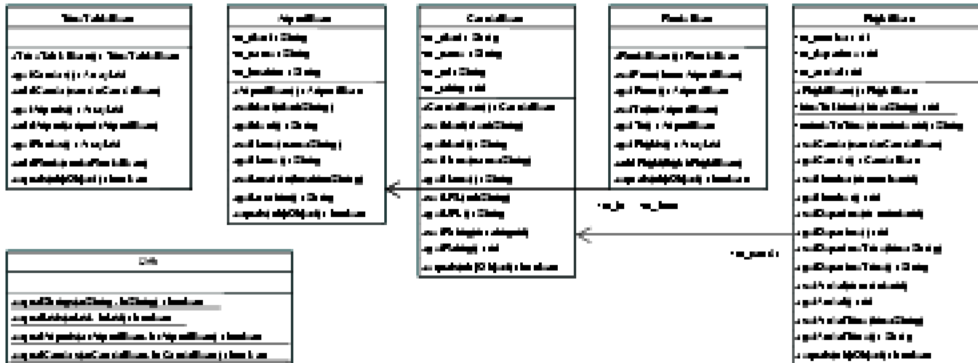
```
                <?xml version="1.0"?>
<timetable>
    <carrier ident="AR">
        <rating>9</rating>
        <URL>http://www.arcticairlines.com</URL>
        <name>Arctic Airlines</name>
    </carrier>
    <carrier ident="CA">
         <rating>7</rating>
       <URL>http://www.combinedlines.com</URL>
        <name>Combined Airlines</name>
    </carrier>
    <airport ident="SEA">
        <location>Seattle, WA</location>
        <name>Seattle-Tacoma International Airport</name>
    </airport>
    <airport ident="LAX">
        <location>Los Angeles, CA</location>
        <name>Los Angeles International Airport</name>
    </airport>
    <route from="SEA" to="LAX">
        <flight carrier="AR">
          <number>426</number>
          <depart>6:23a</depart>
          <arrive>8:42a</arrive>
        </flight>
        <flight carrier="CA">
          <number>833</number>
          <depart>8:10a</depart>
          <arrive>10:52a</arrive>
        </flight>
        <flight carrier="AR">
          <number>433</number>
          <depart>9:00a</depart>
          <arrive>11:36a</arrive>
        </flight>
    </route>
    <route from="LAX" to="SEA">
        <flight carrier="CA">
          <number>311</number>
          <depart>7:45a</depart>
          <arrive>10:20a</arrive>
        </flight>
        <flight carrier="AR">
          <number>593</number>
          <depart>9:27a</depart>
          <arrive>12:04p</arrive>
        </flight>
        <flight carrier="AR">
          <number>102</number>
          <depart>12:30p</depart>
          <arrive>3:07p</arrive>
        </flight>
    </route>
</timetable>
```

Figure 1 shows the class structure I used for mapped data binding to these documents. In the sections on each data binding framework I'll show the generated class structures for comparison purposes. The diagrams included inline are just

© Copyright IBM Corporation 2003. All rights reserved.

thumbnails in all cases; click on the small image if you'd like to view the full size graphic.

Figure 1. Mapped binding class diagram



(click to enlarge)

The long road to JAXB

Java API for XML Binding (JAXB) is the evolving standard for Java Platform data binding. This is being developed through the Java Community Process as "JSR-31 - XML Data Binding Specification". The project started in August 1999, with the goal of defining an approach for generating Java language code that would be linked to XML structures. Originally targeted for release in the second quarter of 2000, a preliminary Early Access (EA) version was announced at JavaOne 2001 and made publicly available in June 2001.

The EA version of JAXB was based on an innovative pull parser design that allowed validation to easily be built into generated unmarshalling code. It used DTDs as the basis for code generation, building classes that automatically performed structure -- but not data -- validation for XML documents as they were parsed. This approach showed great promise as a fast and efficient way of handling conversion between XML and Java language objects, but the EA code was only a partial implementation and clearly needed a lot more work before it could be considered complete.

The expert group soon began receiving feedback on the EA release. Partially in response to this feedback they looked into restructuring JAXB, and later updated the Web site with a note that JAXB was being enhanced in several respects. The site also noted that the next version would not be API-level compatible with the earlier version -- but kept the EA version available for download.

JAXB rearchitecture

No details on the new architecture were made public until March 2002, when at JavaOne, Sun announced that the EA code was being effectively discarded as a basis for further work on JAXB. It was to be replaced by a new design that shared a

© Copyright IBM Corporation 2003. All rights reserved.

few common features but used an incompatible API and internal architecture. The dramatic change of direction caught me by surprise, along with others who'd been interested in the EA code.

Joseph Fialli, the Sun JSR lead for JAXB, attributes the major changes to several factors. The main issue was the complexity of extending the older codebase to support W3C XML Schema. This is a notoriously complex specification, to the extent that more than two years after approval there are still only a handfull of parsers on any platform that come close to full compliance. The original JAXB code required validation to be controlled by the actual objects, and extending this approach to Schema turned out to be too much of an effort to be accomplished in a reasonable timeframe.

Along with the changes to accommodate Schema, the expert group also decided to rethink the way validation was handled. The original JAXB code unconditionally verified the structure of documents, throwing an exception and aborting the processing if an error was found. Fialli said that public comments complained that this approach was both too limited and too restrictive -- users in some cases wanted to be able to check for multiple validation errors at once, and in other cases wanted to disable validation completely (either for performance reasons, or to marshal documents that did not precisely match the grammar). The new JAXB architecture allows for both these alternatives.

Finally, the expert group decided to abandon the idea of a single binding framework runtime like that found in the original EA release. Instead they went with an interface approach where different data binding frameworks could be used behind the scenes. This allows user code to be ported between frameworks, but not the generated classes -- those are specific to a particular data binding framework and will only operate with that framework. The pull parser approach used in the EA runtime has been replaced by mandatory SAX2 2.0 parser support with optional framework-specific support for other alternatives (possibly including the new pull-parser based Streaming API for XML that's being defined by JSR 173), and the data structures themselves are changed to JavaBean-like data objects that can be easily manipulated by an external framework.

JAXB beta

Since March, the JAXB project has been moving forward in this new direction. The first public result of this effort was the publication in late summer of a new (but still preliminary) draft of the specification. Then in October, Sun provided a new beta version of the JAXB reference implementation, finally replacing the long outdated EA version. This recent beta is the version I used for my evaluation and performance tests in these articles. It works directly from Schema document descriptions, generating a hierarchy of classes that matches the element types and usages defined for the documents. This hierarchy includes four general types of classes: interfaces for the defined types, interfaces for the actual elements, and

© Copyright IBM Corporation 2003. All rights reserved.

implementations for both sets of interfaces.

Figure 2. JAXB interfaces class diagram

(click to enlarge)

Figure 3. JAXB implementation class diagram

(click to enlarge)

The type interfaces are the most interesting of these classes from the application program standpoint. These are JavaBean-style collections of get and set methods for the data within the type. The methods included in the type interfaces follow rules set down in the JAXB specification, so your application code can safely use the interfaces for access to all data while retaining portability between JAXB implementations. These interfaces make JAXB-generated code fairly easy to use with existing documents. However, it is a little awkward for constructing and modifying data structures. Since interfaces are used, instances cannot be constructed directly; instead, instances must be created using factory methods, then populated with data values using the JavaBean-like accessor methods.

Generating code from a Schema description is very simple with JAXB. A supplied binding compiler, **xjc**, operates as a command-line tool. It takes a Schema document as input, generating files to a specified output package and destination directory. Options also allow the user to control whether generated code files are read-only, and whether the Schema description is strictly validated.

The JAXB specification defines ways to customize several aspects of the generated

© Copyright IBM Corporation 2003. All rights reserved.

data binding by using binding declarations. These include:

- Options for controlling the names of generated classes and properties
- A way to specify an existing implementation class to be used by the binding
- Options that allow (limited) control over the validation handling and the serializers/deserializers used for marshalling and unmarshalling

You can either embed the customizations as annotations within the actual Schema document, or supply them separately through the use of a separate external binding declaration document. The current beta version of the reference implementation only supports the first approach, but the use of an external binding declaration document will be supported in a future release.

Overall, JAXB is shaping up as a powerful and flexible tool for binding Java language code to documents defined by W3C XML Schema grammars. Since it's likely to be approved as a Java Platform standard it will be widely supported, and moving binding applications between implementations should be about as easy as moving Web applications between servlet engines (generally very simple, but with occasional quirks).

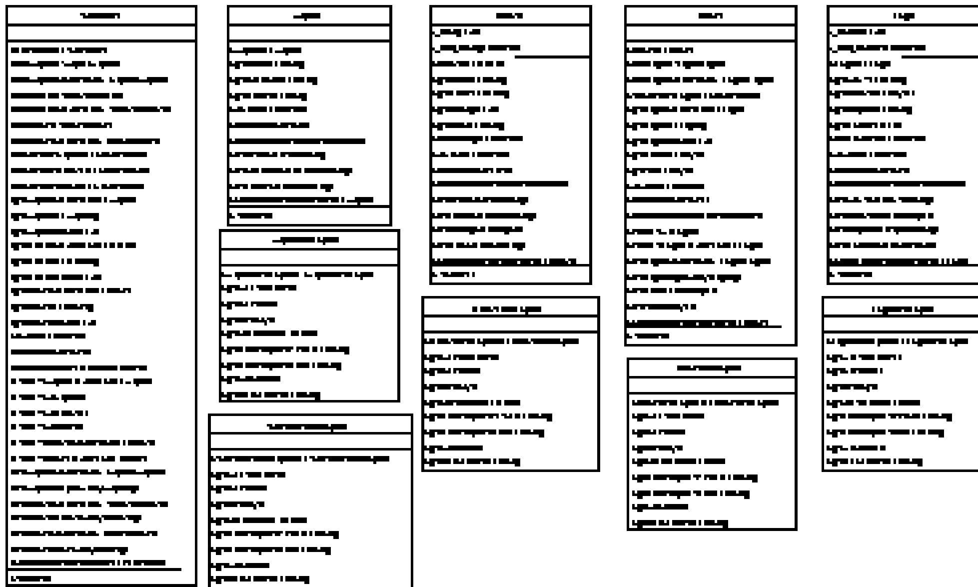
JAXB does have some weaknesses, however. The biggest current limitation is that it's simply not licensed for anything other than evaluation use. Until the production release (currently planned for this quarter), JAXB is not a practical alternative for use in real projects. It is also limited in the degree of customization that can be applied to the generated code. In many cases, you can define your own implementation classes for the interfaces defined by JAXB, but the interfaces themselves are always tied to the Schema description with little possibility for modification.

Castor

The Castor framework for XML data binding supports both mapped and generated bindings. In my last article, I covered some of the features of the Castor mapping binding approach. For this one I'm only discussing the code generation from Schema, though in Part 2 I'll look at performance for both approaches. Refer back to the earlier article (see Resources) to find out more about how mapped data binding works in Castor.

Figure 4. Castor generated class diagram

© Copyright IBM Corporation 2003. All rights reserved.



(click to enlarge)

Castor's support for code generation differs in details from the JAXB approach, but is very similar in intent. As with JAXB, the data model is provided to the application with JavaBean-like structures. The main difference is that Castor avoids the use of interfaces in favor of going directly to generated implementation classes. Along with each implementation class, Castor also generates a descriptor class that contains the binding and validation code. Since Castor uses concrete classes rather than interfaces, it's a little simpler than JAXB for constructing or modifying document data structures. You can just use a constructor for the appropriate class directly, without going through a factory class.

The current beta release of Castor (0.9.4.1 as of my writing this) doesn't support any substantial customizations in the code generation, but that's changing. The next beta release is expected to support the use of a mapping file to control various aspects of code generation. Initially these aspects will be limited to class and package names, but it's planned to add support for user-supplied implementation classes in the longer term. The Castor developers also plan to support JAXB in Castor, probably using some sort of compatibility layer.

Generating code from a Schema description is as easy with Castor as with JAXB, and uses the same basic options. Castor does use a few additional command line parameter options, and supplies even more options through property file settings. These are mainly useful in special circumstances, and do not include as much control over class names and validation as that provided through Schema document annotations with JAXB.

The main weakness of source code generation with Castor right now is the limited support for customization. That's starting to change, and given the substantial customization that's possible when using mapped data bindings with Castor (as

© Copyright IBM Corporation 2003. All rights reserved.

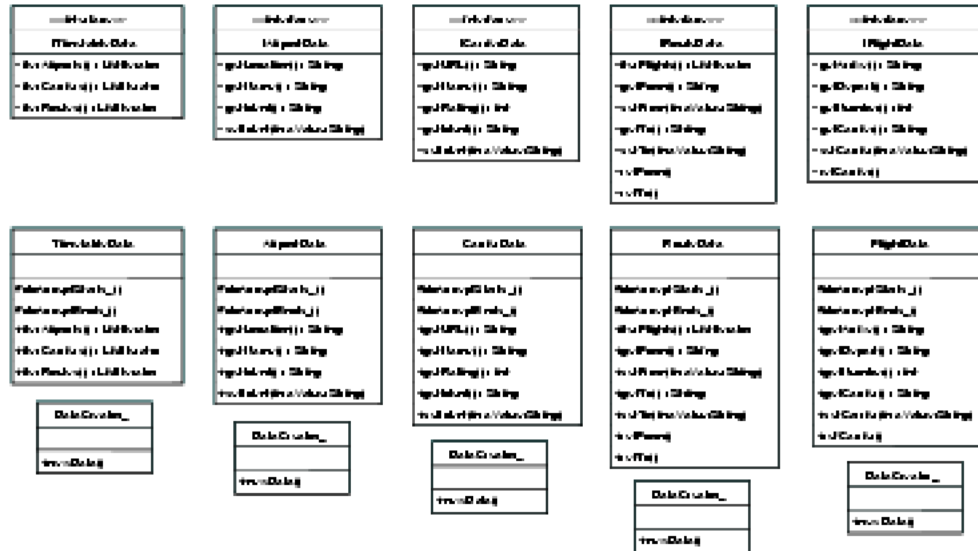
described in the prior article -- see Resources), I'd expect that it will eventually allow at least as much flexibility in source code generation. In the long run, this should make it more adaptable than JAXB.

Castor has been released with a BSD-style license that allows for full commercial use without significant restriction. It appears to be reasonably stable, but you'll need to update to the latest development code (or wait for a new beta release) any time you run into a bug that you need fixed.

JBind

Like JAXB and Castor, JBind generates binding code based on Schema descriptions of XML documents. Despite this commonality, the main focus of JBind is actually very different from the other two. The main author (Stefan Wachter) calls this focus "XML Code", describing it as the combination of XML data described by a Schema with behavior realized by Java language code. Whereas JAXB and Castor focus more on allowing Java language applications to easily work with XML, JBind builds application code frameworks around the XML. Once the framework has been built by JBind, you can extend it with your own code to add functionality.

Figure 5. JBind generated class diagram



(click to enlarge)

JBind *can* also be used for conventional data binding, and I use it in this manner for the performance tests reported in Part 2. Doing this is somewhat awkward, though, partially because JBind always needs to process the Schema for a document at runtime. If your instance documents don't reference the corresponding Schema directly, you need to either use a special mapping file or manually load the proper Schema into your own code prior to reading an instance document. The current

© Copyright IBM Corporation 2003. All rights reserved.

documentation doesn't really show you how this works. JBind's handling of changes to the bound document structure is also rigid compared to the other data binding frameworks. Existing element objects can be deleted through the use of a `ListIterator`, but new ones can only be created using generated create methods, which automatically append them following existing content.

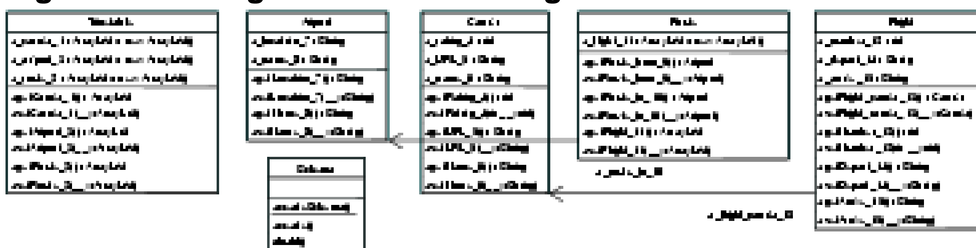
Behind the scenes, JBind uses a very different approach to handling document data. Rather than generating JavaBean-style data classes (as with JAXB and Castor), JBind stores everything in a document model (currently a DOM level 2 implementation) and builds the binding code as a facade to access the data stored in the document model. This is a very interesting approach, and seems to have the potential for some nice paradigm-crossing benefits if carried out fully. The only advantage provided at present is support for XPath-based constraints and access methods in the generated code. Since the storage mechanism is incidental to the main intent of JBind, this may also change in the future.

JBind has the advantage of offering the most complete Schema support of any of the data binding frameworks considered, as well as the XPath extensions mentioned above. If the core of your application is processing an XML document, the XML Code framework constructed by JBind may make it very easy to use. For general data binding usage where XML documents are involved but aren't the main focus of the application, one of the other data binding approaches would probably be simpler. JBind also suffers a substantial performance disadvantage compared to other frameworks due to required validation when unmarshalling and to the document model back-end storage mechanism (I cover this in more detail in Part 2). JBind is distributed with an Apache-style license that allows full commercial use.

Quick

The Quick documentation describes it not as a tool for processing XML, but rather an *extension* to the Java language that uses XML. It is based on a series of development efforts that predate both the Java Platform and XML, with many restructurings along the way. It certainly provides a very flexible framework for working with XML on the Java Platform -- far more than I've been able to learn and use for this article.

Figure 6. Quick generated class diagram



(click to enlarge)

© Copyright IBM Corporation 2003. All rights reserved.

Quick's flexibility does come at a price. It uses a rather complex series of steps to move from a DTD document description to generated code, with three separate binding schema (not to be confused with W3C XML Schema) documents as intermediate steps in the process:

- The QDML document provides a document description roughly equivalent to a DTD with the addition of types and inheritance.
- The QJML document defines a binding of XML into Java language objects.
- The QIML file is basically a compiled form of the QJML which you can use to generate the actual binding code.

In my performance tests with Quick for Part 2 I kept the customizations of these files to a minimum, but still needed to do some manual edits to get the desired end result. After generating the QDML file from a DTD grammar I had to edit the file to define the document root element and add type information for non-`String` values (a couple of `int`s, in this case). I then ran the program to generate a QJML file from the QDML, and edited the resulting QJML to add type information to references. This step was not really necessary, but allowed me to generate code with specific types for object references (a feature not supported by Castor or JAXB code generation). Finally I ran the tool to generate a QIML file from the QJML, then finished by running code generation tools to get both the JavaBean-like object classes and the actual binding classes that convert to and from XML.

With more manual editing of the files I could have avoided generating new code for the object classes, instead linking directly to the existing classes used by the Castor mapped binding. This ability to work with existing classes is a very powerful feature. Its utility is somewhat diminished in Quick by the complexity of the schema files and the amount of changes necessary to make use of the feature, but it does show how flexible Quick can be.

This flexibility is the most powerful feature of Quick. The main drawbacks to using Quick are the complexity of the various schema files and the lack of support for Schema grammars. It also appears difficult to get help in using Quick: Questions often do not receive any responses on the forum and mailing list. Quick is licensed under the GNU Library or Lessor General Public License (LGPL), which generally allows software to be included in both free and commercial projects.

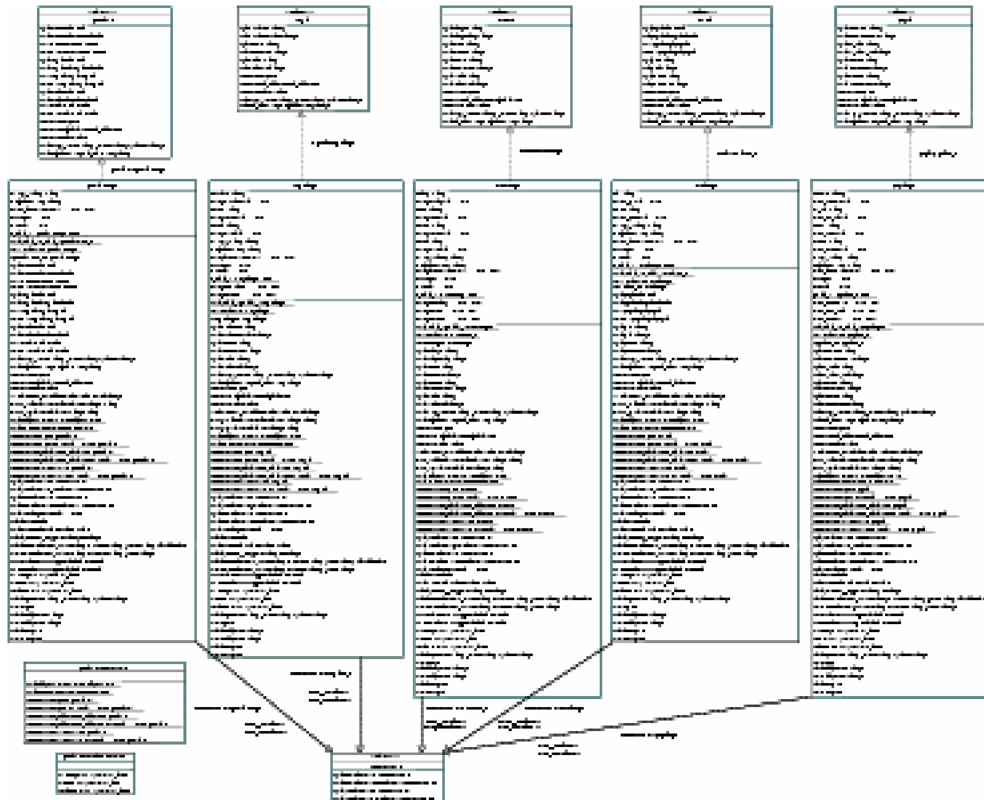
Zeus

Like Quick, Zeus also generates code based on DTD descriptions of XML documents. (Schema support is being worked on, but is currently at a pre-alpha stage of development.) That's about the only similarity between the two frameworks.

© Copyright IBM Corporation 2003. All rights reserved.

While Quick is complex and powerful, Zeus is simple to use -- but very limited.

Figure 7. Zeus generated class diagram



(click to enlarge)

Zeus code generation is similar in usage to JAXB or Castor, with a command line tool provided to construct the necessary classes. As with JAXB, the binding uses interfaces. Whereas JAXB uses a factory to construct new instances of an object class, Zeus uses a generated implementation class with prototyping. With Zeus, you subclass the implementation class and use your subclass instead of the generated class when unmarshalling a document.

Unlike any of the other frameworks discussed, Zeus supports only `String`s rather than typed values such as `int` or `Date`. It also lacks supports for references, so that graph structures cannot be directly unmarshalled or marshalled. These are major limitations -- much of the utility of data binding comes from the convenience of working with typed data values and transparently handling linkages between objects. Without support for these features, Zeus feels more like a trimmed-down document model than a full data binding framework.

If you're only working with `String` values, Zeus may still be a good choice to consider. The main drawbacks to using Zeus are the limited form of binding provided, and an overall slow rate of progress on the project. As with Quick, you may find it difficult to get answers to your questions. Zeus is distributed under the

© Copyright IBM Corporation 2003. All rights reserved.

Enhydra public license version 1.1, which is derived from the Mozilla public license.

Mapped versus generated

In this article, I've discussed several different frameworks for Java language code generation from an XML document grammar. This is just one way of handling XML data binding for Java language applications. The main alternative is to use some form of mapped binding approach, where you build your own classes (or have them initially built from a grammar, then modify them to meet your needs) and specify to the binding framework how these classes are associated with the XML documents. Each approach has advantages and disadvantages, and there are probably situations where each can most appropriately be applied.

Code generation automatically builds classes that reflect the XML document structure (in other words, the grammar in the form of a DTD or Schema), which allows you to start working with documents very quickly. When the code generation is based on Schema descriptions, the constructed classes can include full data typing information (though there are some problems with this; many of the Schema datatypes do not have direct Java language equivalents). With code generation, you can also build validation into the constructed classes, either automatically checking values when they are set or checking validity on demand. Thus, you can ensure that the documents you generate through marshalling always match the expected structure.

The main drawbacks of the code generation approach are the flip side of its advantages. By so closely reflecting the document structure, it creates a tight coupling between your application code and that structure. If the document structure changes, you need to rerun the code generation and modify your application code to work with the resulting modified data classes.

You also generally need to work with the entire document structure, and cannot easily subset it for code generation purposes. This can be a concern if you're using a complex structure with many optional components (perhaps defined as an industry standard) but your application works with documents that only use a subset of the components. With most of these frameworks, your generated classes will always match the entire structure. Depending on the framework, you may also need to include all these generated classes at runtime. This results in both code bloat and an overly complex data model for your application. You can, of course, avoid this by editing the DTD or Schema to eliminate the components you don't need -- but then you need to maintain your modifications across any changes to the base grammar, which creates a new set of problems.

Mapped bindings (such as those implemented by Castor or Quick) offer more flexibility than code generation. You work with true object classes combining data and behavior. You can also decouple your object classes from the actual XML to a

© Copyright IBM Corporation 2003. All rights reserved.

certain extent. Modifying the mapping definition, rather than requiring changes in application code, often handles minor changes in the XML document structure. You can even define separate input and output mappings to unmarshal documents with one format and marshal them with another. On the downside, mapped bindings do require more effort to set up than the generated code approach.

In summary, neither approach is going to be best for all applications. If you're working with a stable document structure defined by a Schema or DTD, and the structure suits your application's needs, code generation is probably the best approach. If you're working from existing Java language classes, or you want to use a class structure that reflects your application's usage of the data rather than the XML document structure, mapping is probably best. Unfortunately, the main focus of most current development efforts appears to be on code generation rather than mapping. This limits the currently available mapping options to just Castor and Quick.

Conclusions

Here in Part 1, I've reviewed the main XML data binding frameworks that support Java language code generation from an XML document description. These frameworks vary widely in capabilities (and also in performance, as I'll show in Part 2). Of the approaches based on W3C XML Schema definitions, Castor offers the best current support for general-purpose data binding applications. It's already available for use, and is being extended to provide better customization of the generated code and more complete Schema support. Castor is also expected to support the coming JAXB standard in the future.

JAXB looks like it will be a very attractive option once it becomes available as a production release (the current beta license only allows evaluation use). It currently appears that Castor will support more customization than JAXB, but JAXB will offer the advantage of portability between implementations. JAXB is also likely to be used within other Java Platform standards (such as the JAX-RPC standard for Web services), so writing your application to JAXB may provide plug-in compatibility with these standards in the future.

JBind offers the best Schema support. It may be a good option if your application fits the XML Code model and does not have stringent performance requirements, but JBind seems relatively cumbersome for general XML data binding usage. Quick offers a very high degree of flexibility and power, but only supports DTD grammars and is fairly complex to use. Zeus is simple and easy, but (as with Quick) supports only DTDs and additionally only allows data to be accessed as `String` values.

These last three all seem more appropriate for applications with special requirements than for general usage. If you only have DTD descriptions of your documents rather than schemas, you may want to try Quick or Zeus for this reason.

© Copyright IBM Corporation 2003. All rights reserved.

This should not be a major concern for most applications since there are a number of programs available to convert DTDs into schemas (though some hand-tuning may be required). One such program is included in the Castor distribution (`org.exolab.castor.xml.dtd.Converter`, as mentioned in the Castor XML FAQ).

In Part 2 I'll show the performance results from testing these data binding frameworks on some sample documents. These results cover both generated code and mapped binding approaches, with document models included for comparison. I was really surprised to see that... but wait, I can't give *everything* away now. Add a reminder to your calendar to check back next week and catch the full performance story -- I guarantee you'll find the results worth the read!

About the author

Dennis Sosnoski
Dennis Sosnoski (dms@sosnoski.com) is the founder and lead consultant of Seattle-area Java technology consulting company Sosnoski Software Solutions, Inc., specialists in J2EE, XML, and Web services support. Dennis's professional software development experience spans over 30 years, with the last several years focused on server-side Java technologies. He's a frequent speaker on XML in Java and J2EE technologies at conferences nationwide, and chairs the Seattle Java-XML SIG.

© Copyright IBM Corporation 2003. All rights reserved.