XML Data Binding: Integrating XML and Object-Oriented Technologies

Neil **Chaudhuri**

Abstract

Data are the essence of business processes and technical applications, and managing data effectively is critical for success in any industry. To that end, XML has emerged as the dominant syntax for data management. The fundamental organizing principle of XML is hierarchy. Parent-child relationships among data are maintained to infinite depth through markup. Hierarchies also serve as a critical component of XML's validation capability. An XML Schema document defines the rules for structuring data within an XML instance by describing a finite set of hierarchy sequences and an explicit set of sequences of elements within them. Hierarchy, therefore, is the underlying principle of data management in XML.

While XML is a relatively recent arrival on the technology landscape, object-oriented (OO) programming has long been venerated as the dominant paradigm for developing complex, mission-critical software. From Smalltalk and C++ to Java and C#, OO's fundamental organizing principles for data management are encapsulation and inheritance. Encapsulation is the principle whereby objects hide their data and allow other objects to have access to them only through defined APIs (Application Program Interfaces). Inheritance is the principle whereby data and behavior are passed from a parent class to its children. Encapsulation and inheritance, therefore, are the underlying principles of data management in OO.

As powerful as XML and OO principles are, it is no surprise that these two threads have been interwoven so often into the fabric of application development. However, it is just as predictable that integrating the two is not without its challenges, for their approaches to data management are nearly incompatible. One utilizes a static hierarchy of data elements while the other utilizes dynamic data exchange among multiple entities through method calls and inheritance.

A solution to the problem of integrating these two approaches is XML data binding. This approach seeks to generate objects from XML Schema documents and populate them with data in instance documents validated against the schemas. Then, after interacting and perhaps evolving in such as way as to meet business requirements, the objects are converted back into XML instances valid against the original schemas. The promise offered by XML data binding is enormous, yet it remains unclear the extent to which this promise has been realized.

This paper addresses this question for the benefit of intermediate and advanced XML and OO developers who have sought to move data seamlessly between the two. The discussion begins with an overview of XML data binding and describes the potential benefits it offers to application developers. It then introduces the reader to two popular Java-XML binding frameworks, JAXB from Sun Microsystems and the open-source tool Castor. The core of this discussion is a comparative and painstaking evaluation of these tools against criteria deemed to be of greatest significance to XML analysts and Java developers in this space. Finally, conclusions are drawn regarding the relative effectiveness of the tools, and suggestions are made for further study.

Table of Contents

1. Introduction

With technology becoming ever more pervasive within the realm of business, organizations have witnessed the confluence of two powerful streams: object-oriented (OO) technologies and XML. Where the two meet is in data management. Object-oriented code is developed to retrieve, manipulate, and persist data in such a manner as to achieve the goals of the business. OO languages provide powerful application programming interfaces (APIs) for enabling behavior, but they are proprietary in that no worldwide standard exists. Meanwhile, XML has emerged as the dominant syntax for the organization and validation of data—and more recently in other areas such as the transmission and security of data that define web services. XML does not exhibit any behavior on its own, but it is a worldwide standard recognized by all major vendors in industry. It is not hard to see why OO technologies and XML complement each other so well and thus why organizations derive tremendous benefits from combining the two to meet vital business needs.

1.1. The Challenges of Integrated Data Management

As is invariably the case with any technology, there are obstacles to overcome no matter how powerful the benefits may be. A joint OO-XML solution has one key obstacle—fundamentally different approaches to data management. The fundamental organizing principle of XML is hierarchy. Parent-child relationships among data are maintained to infinite depth through markup. Hierarchies also serve as the basis for XML's validation capability. An XML Schema document defines the rules for structuring data within an XML instance by describing a finite set of hierarchy sequences and an explicit set of sequences of elements within them.

Yet from Smalltalk and C++ to Java and C#, OO's fundamental organizing principles for data management are encapsulation and inheritance. Encapsulation is the principle whereby objects hide their data along with the details of their manipulation and allow other objects to have access to them only through the method calls defined in their APIs. Inheritance is the principle whereby data and behavior are passed from a parent class to its children. The data management approach utilized by XML and that utilized by OO are very difficult to resolve, for one utilizes a static hierarchy of data elements while the other utilizes dynamic data exchange among multiple entities through method calls and inheritance.

1.2. The Case for XML Data Binding

The unquestioned power of XML and OO design—and even more so their union—has motivated the search for a solution to reconcile their approaches to data management. Traditional solutions like DOM and SAX have become quite pervasive, but they have proven wanting. DOM has an intuitive API, but it is far too costly in memory and performance to be a viable option for large documents. SAX solves the resource issues associated with DOM, but it demands sophisticated programming expertise in order to be utilized effectively. Yet in either case, developers are forced to devote significant effort towards bridging the gap between the XML realm and the OO realm—effort better spent addressing the core business requirements of the application. Moreover, the effort is all but futile. As a consequence of the power and complexity of XML Schema, all but the most trivial schema documents offer countless combinations of possibilities for valid instance documents. It is impossible for developers to write code that addresses all of these combinations. The tendency then is to write and test code to handle the common cases and hope that no gaps are exposed. Such code is certain to fail eventually, and the subsequent rework is sure to be costly.

A more robust alternative that has captivated both the XML and OO communities is XML data binding. This approach seeks first to validate XML Schema documents and then generate object representations of the constructs—element declarations, attribute declarations, type declarations, and more. Subsequently, in a process called *unmarshalling*, these objects are populated with data in instance documents validated against the schemas. In theory, the object model in memory is a faithful OO representation of the hierarchical data model defined in the XML instance. At this stage, business-logic code can interact with these objects in the manner familiar to OO developers in order to meet business requirements. Ultimately, in a process called *marshalling*, the objects are serialized into new XML instance documents faithfully representing the underlying object model. Again, in theory, these instances are valid against the original

XML Schema document because the objects enforced the constraints like cardinality and type restrictions defined therein.

The promise offered by XML data binding is enormous. A solution that automates efficient compatibility between the data management approaches of XML and OO would have profound repercussions for the future of E-business. However, the challenges are formidable. Although XML binding is automated unlike the DOM and SAX alternatives, it still stretches the bounds of credulity that an object model can enforce the constraints imposed by the original schema and can accurately encompass all of the combinations permitted by it. Moreover, validity must be preserved while the objects interact with other domain objects to accomplish business objectives. Numerous tools have emerged that claim to do all of these. It is the goal of this discussion to assess the extent to which this promise has been realized by two leading XML data binding tools.

1.3. XML Data Binding Tools: JAXB and Castor

The XML and OO communities—particularly the Java community—have witnessed the proliferation of several XML data binding tools claiming to solve the data integration quandary. Of these, JAXB and Castor are particularly compelling.

1.3.1. JAXB

JAXB (Java Architecture for XML Binding) is a Sun standard and is included as part of Sun’s Java Web Services Development Pack (JWSDP) [Java Technology Documentation]. As with all Sun specifications, JAXB grew out of the Java Community Process as Java Specification Request (JSR) 31 [JSR 31]. What is most compelling about JAXB is that it is a standard implementation with the full support of Sun. Moreover, JAXB 2.0, which is available for public review, will be a part of the next J2EE specification [JSR 222].

1.3.2. Castor

The venerable Castor is an open-source tool that is the oldest and most pervasive XML data binding tool. It is written in Java, but it is proprietary software developed by the ExoLab Project [The Castor Project]. Because it is not a standard, those who use Castor run the risk of vendor lock-in. However, it is widely viewed as the most powerful tool of its kind.

1.4. Course of Discussion

JAXB is the Java standard for XML data binding, but Castor is the most popular binding tool available. The remainder of this discussion will examine these tools in detail and provide a comparative evaluation against a set of criteria deemed to be of greatest significance to XML analysts and Java developers in this space.

2. Evaluation Methodology

The methodology for evaluating JAXB and Castor has at its foundation the interest of both the XML and Java communities. To that end, the XML Schema documents at the core of this discussion were composed of a variety of constructs, and the manner in which these constructs were reflected in the generated object models is a major point of emphasis for this discussion. Next, a single XML instance validated against both schemas was unmarshalled with the resulting objects manipulated and tested against several criteria. Finally, after undergoing several modifications, the code was marshalled into new XML instances, and the well-formedness and validity of the instances were scrutinized carefully. The following sections describe the entire evaluation process in detail.

2.1. XML Schema Documents and Corresponding Instance

Two W3C XML Schema documents were created to serve as the basis for the assessment [1]. The “business” to be conducted with these schemas is the management of data associated with Major League Baseball. Although it is an artifact of American popular culture, baseball is a sport that is very much data-driven. Moreover, the topic of baseball is a welcome respite from the trite purchase-order use cases found so often in the literature.

One schema, called *mlb_primary.xsd*, contains only global declarations and follows many of the schema-design best practices established in industry. It features a wide array of constructs including the following [2]:

- XML Schema data types
- *<xs:sequence>*
- *<xs:choice>*
- *<xs:simpleType>* with restrictions
- *<xs:complexType>* with extensions and with *abstract="true"*
- Cardinality for elements beyond the default value of 1
- Atypical data types
- *<xs:attributeGroup>*

The second schema is called *mlb_locals.xsd*. This schema is largely similar to the other in terms of constructs with the following exceptions:

- Replacement of *<xs:attributeGroup>* with duplicate local attribute declarations
- *<xs:sequence>* with restriction using *<xs:union>*

Where *mlb_locals.xsd* significantly differs from *mlb_primary.xsd* is in the proliferation of local declarations of elements, attributes, and types. The purpose of this schema is to enable analysis of the differences, if any, between the object models generated from the two schemas as a result of increased localization. However, for the sake of simplicity, the schemas were written in such a manner as to produce identical valid instances.

Finally, a sample instance was generated that was valid against both schemas, and the *xsi:schemaLocation* attribute was simply toggled to reflect the schema in question.[3] This instance, called *mlb.xml*, served as the basis for the unmarshalling effort.

2.2. Evaluation Criteria

With the XML Schemas created and valid sample instances derived from them, the next step was simply to use JAXB and Castor to derive object models from the schemas and then unmarshal the instance documents, manipulate the objects, and marshal the objects to create new instances. The tools accomplish these tasks in different ways. The following sections detail the criteria against which their approaches were evaluated.

[1]See Appendices 1 and 2
[2]The *xs* prefix maps to the XML Schema namespace: *http://www.w3.org/2001/XMLSchema.*
[3]The *xsi* prefix maps to the *XML Schema-instance* namespace: *http://www.w3.org/2001/XMLSchema-instance.*

2.2.1. Bidirectional Integrity

Among the many reasons XML has emerged as the dominant syntax for data management is its validation capability. An XML Schema document uses hierarchies of parent-child relationships to define the rules for structuring data within an XML instance, and theoretically these rules are enforced with zeal by validating parsers. For XML data binding to prove successful, the same vigilant validation must transpire in some fashion within the object model—either in the API calls or during the marshalling process—because the object model is mutable. Therefore, the criterion of Bidirectional integrity reflects the extent to which the validation capability inherent in XML is preserved throughout the XML data binding process.

The following categories of integrity are examined in this discussion:

- With regard to generation of the object model
  - Validation of original schema
  - Validation of source instance
  - Target namespace resolution in the generated object model
  - Preservation of relationships among elements, attributes and types within the ensuing object model—particularly with varying levels of localization
- With regard to unmarshalling and object manipulation
  - Preservation of the intent of XML Schema constructs
- With regard to marshalling
  - Well-formedness of the generated instance
  - Preservation of the changes made to the object model upon unmarshalling
  - Mapping of the *xsi* prefix to the *XMLSchema-instance* namespace and specification of the *xsi:schemaLocation* attribute
  - Validity of target instance

Bidirectional integrity is the most critical point of comparison between JAXB and Castor, for this issue is at the heart of the viability of XML data binding.

2.2.2. Usability

Usability is the most basic criterion for any technology, for robust sophistication is of little comfort to a frustrated user at wit's end. However, with XML binding tools, usability, or lack thereof, manifests itself in numerous forms. Therefore, it was necessary to subdivide the usability criterion into the following categories:

- *Generation of the Object Model*. This addresses the usability of the process whereby an object model is generated from an XML Schema document.
- *Unmarshalling*. This addresses the usability of the process whereby an XML instance document is validated against an XML Schema and subsequently represented within the established object model.

- *Object Manipulation*. This addresses the usability of the process whereby the object-model manifestation of an XML instance can be manipulated to perform the familiar CRUD operations upon the data in the original document.[4]
- *Marshalling*. This addresses the usability of the process whereby an object representation of an XML instance document (perhaps modified) is serialized into a valid XML instance.

The performance of each tool in the context of each of these categories contributes equally to the appraisal of its overall usability.

3. Evaluation Results: JAXB

This portion of the discussion examines the performance of JAXB as an XML data binding tool and its usability by client code.

3.1. Bidirectional Integrity

JAXB was observed for its faithfulness to the intent of the W3C XML Schema documents from the initial generation of the object model through unmarshalling of the XML instance and marshalling of new XML instances from the underlying object model.

3.1.1. Generation of the Object Model

JAXB tested both schemas and the instance document for well-formedness and validity and reported errors by throwing standard Java exceptions. Subsequently, when the object model was generated from either *mlb_primary.xsd* or *mlb_locals.xsd*, the objects were placed by default in a Java package structure reflecting the target namespace of the schema. For example, the element called *AmericanLeagueTeam* found in *mlb_primary.xsd* with target namespace *http://www.xmlconference.org/xmldatabinding/jaxb/primary* produced a Java artifact (more on this shortly) called *org.xmlconference.xmldatabinding.jaxb.primary.AmericanLeagueTeam*. Such target namespace resolution was clever and intuitive while requiring no customization.

JAXB took an interesting approach with regard to the structure of the object model. A set of Java interfaces representing global elements and types and named identically was created in the *org.xmlconference.xmldatabinding.jaxb.primary* package. Then in a subpackage called *impl*, JAXB generated concrete Java classes implementing those interfaces. In the above example, the *AmericanLeagueTeam* element mapped to an interface with the same name implemented by a class called *AmericanLeagueTeamImpl* in the *impl* subpackage.

Also of note is the relationship of this element with its type. The *AmericanLeagueTeam* element is of type *AmericanLeagueTeamType*, which JAXB predictably represented as an interface called *AmericanLeagueTeamType* and an implementation class called *AmericanLeagueTeamTypeImpl*. Moreover, the type declaration for *AmericanLeagueTeamType* contains an occurrence of *AmericanLeagueRosterType*, and this relationship was represented in the object model through composition. The following UML diagram conveys the relationships among the *AmericanLeagueTeam*, *AmericanLeagueTeamType*, and *AmericanLeagueRosterType* interfaces and their respective implementations.

[4]CRUD is an acronym often used in the context of databases referring to the common operations of *Create*, *Read*, *Update*, and *Delete*.

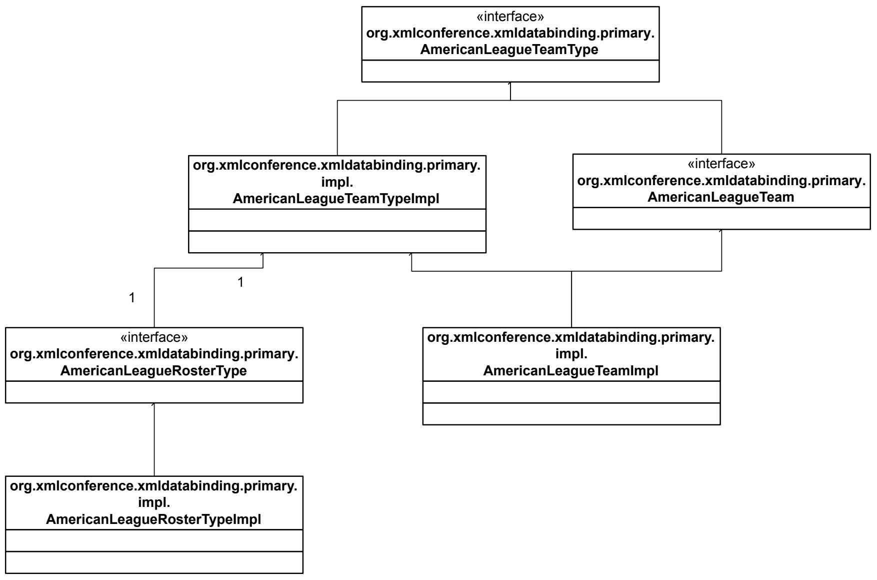
«interface»
org.xmlconference.xmldatabinding.primary.
AmericanLeagueTeamType
org.xmlconference.xmldatabinding.primary.
impl.
AmericanLeagueTeamTypeImpl
«interface»
org.xmlconference.xmldatabinding.primary.
AmericanLeagueTeam
1
1
«interface»
org.xmlconference.xmldatabinding.primary.
AmericanLeagueRosterType
org.xmlconference.xmldatabinding.primary.
impl.
AmericanLeagueTeamImpl
org.xmlconference.xmldatabinding.primary.
impl.
AmericanLeagueRosterTypeImpl

Figure 1. UML Diagram of Relationships among Element and Type Representations Generated by JAXB from *mlb_primary.xsd*

Several key conclusions can be drawn from the diagram. First, JAXB constructs an object model by building relationships among interface definitions rather than class definitions. Second, JAXB manifests the relationship of an element to its type in an XML instance through inheritance in the object model—where the element is a subclass of its type. Finally, and perhaps not so surprisingly, the hierarchical relationship between an XML element or type and elements contained therein are represented in the object model through composition.

In the other schema, *mlb_locals.xsd*, the object model derived by JAXB was largely similar but with a remarkable exception. Indeed, as before, relationships were among generated interface definitions, and elements were related to types through inheritance. However, local declarations were represented not through composition as is the case with their global counterparts but rather through *public inner classes and interfaces*. The *AmericanLeagueRosterType* interface was defined *within* the *AmericanLeagueTeamType* interface, and the *AmericanLeagueRosterTypeImpl* class was defined *within* the *AmericanLeagueTeamTypeImpl* class. The following UML diagram conveys the same relationships depicted above but among classes derived from *mlb_locals.xsd*.

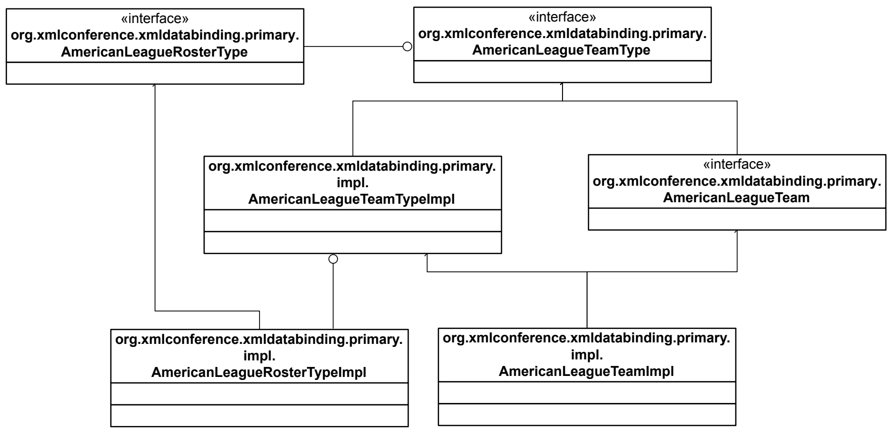
«interface»
org.xmlconference.xmldatabinding.primary.
AmericanLeagueRosterType
«interface»
org.xmlconference.xmldatabinding.primary.
AmericanLeagueTeamType
org.xmlconference.xmldatabinding.primary.
impl.
AmericanLeagueTeamTypeImpl
«interface»
org.xmlconference.xmldatabinding.primary.
AmericanLeagueTeam
org.xmlconference.xmldatabinding.primary.
impl.
AmericanLeagueRosterTypeImpl
org.xmlconference.xmldatabinding.primary.
impl.
AmericanLeagueTeamImpl

Figure 2. UML Diagram of Relationships among Element and Type Representations Generated by JAXB from *mlb_locals.xsd*

The proliferation of inner classes mapping to local declarations in the original schema is less than ideal. Aside from the inelegance and ambiguity of such an approach, public inner classes also lead to security vulnerabilities due to the vagaries of the Java compiler [Kolawa]. It is clear that the behavior of JAXB with increased localization of declarations should motivate XML analysts to produce schemas that minimize their occurrence.

3.1.2. Unmarshalling and Object Manipulation

The two schemas, however, showed no difference in the manner in which XML Schema constructs were represented in the object model. The following sections detail the manner in which JAXB unmarshalled these constructs as found in *mlb.xml*.

3.1.2.1. XML Schema Data Types

The schemas utilized for this discussion did not push the limits of XML Schema's data typing capability because it was not necessary for the business process. Only four data types were utilized, and the following table reflects the default mapping of those to Java data types by JAXB.

XML Schema Data Type	Java Data Type
xs:string	*java.lang.String*
xs:integer	*java.math.BigInteger*
xs:nonNegativeInteger	*java.math.BigInteger*
xs:decimal	*java.math.BigDecimal*

Table 1. JAXB Data Type Mappings

The first entry in the table comes as no surprise, but the other entries are less than intuitive. Both the *BigInteger* and *BigDecimal* classes are provided by the Java 2 Standard Edition (J2SE) API to represent numerical values to preclude the likelihood of overflow or loss of precision possible with Java primitive types like *int* and *long* or *float* and *double* [J2SE API]. It is for this reason that these classes are so useful in applications with requirements for rigorous mathematics. This capability, however, is not as useful in most cases for XML data binding. Java primitive types require fewer resources and are easier to manipulate in code, and they will likely serve the majority of use cases where XML is directly involved. Moreover, they can always be converted to the *BigInteger*, *BigDecimal*, or primitive wrapper classes as requirements dictate—either through unmarshalling via JAXB customization or through business logic code at some point after unmarshalling. Therefore, the default data-type mapping behavior of JAXB could be improved.

3.1.2.2. <*xs:sequence*>

As is commonly the case, all complex type declarations in the tested schemas contained <*xs:sequence*>. The element declarations contained therein were represented through composition in the ensuing Java code and manipulated through accessor and mutator methods (colloquially called *getters* and *setters* by OO developers). For example, consider the type declaration for *BattingStatisticsType*, which contains an element declaration called *BattingAverage* of type *xs:decimal* within a sequence. *BattingStatisticsType* was unmarshalled into a Java interface called *BattingStatisticsType* containing methods called *getBattingAverage()* and *setBattingAverage()*. This behavior is intuitive and consistent.

3.1.2.3. <*xs:choice*>

Both schemas feature two instances of <*xs:choice*>. The first is a trivial case where the first child of the root element *MajorLeagueBaseball* (of type *MLBType*) may be either the element *Year* of type *xs:integer* or the element *Designator* of type *xs:string*. The second is more elaborate. The aforementioned complex type *AmericanLeagueRosterType* contains a choice between either a sequence of one *DesignatedHitter* element and seven *BenchPlayer* elements *or* simply eight *BenchPlayer* elements.

JAXB failed to reflect either choice accurately in the generated object model. With the trivial choice, the Java code for *MLBTypeImpl* simply had mutators for *Year* and *Designator* where the code did not toggle between the current and former choice if a change was made. It were as if the two elements were in sequence rather than a binary choice.

The other instance of <*xs:choice*> in the schemas fared no better. The generated code for *AmericanLeagueRosterTypeImpl* did not demonstrate awareness of the choices, and therefore it certainly did not toggle between them no matter the measures taken by client code. Thus, JAXB rendered the existence of the <*xs:choice*> ultimately meaningless.

3.1.2.4. <*xs:simpleType*> with Restrictions

Both schemas feature instances of <*xs:simpleType*> with different restrictions. The following table summarizes these types as they are found in *mlb_primary.xsd*.[5]

Simple Type Name	**Base Type**	**Restriction Type**	**Value(s)**
YearType	*xs:integer*	*xs:minInclusive*	*2005*
ERAType	*xs:decimal*	*xs:fractionDigits*	*2*
PitcherRoleType	*xs:string*	*xs:enumeration*	*starter, reliever*
BatsType	*xs:string*	*xs:enumeration*	*left, right, switch*
ThrowsType	*xs:string*	*xs:enumeration*	*left, right*
PercentageType	*java.lang.String*	*xs:pattern*	*.[0-9]{3}|1.000*[6]

Table 2. Simple Type Declarations

[5]In the case of *mlb_locals.xsd*, some of these types are local to their respective elements and are therefore anonymous.
[6]Regular expression representing a decimal with no leading digit and three digits to the right of the decimal point or the literal value 1.000. Possible values include .331, .067, and 1.000. This type is used for such elements as *BattingAverage*.

JAXB did not enforce any of these restrictions within the Java implementations corresponding to these type declarations. Mutators allowed any values so long as they were of the correct Java data type (*e.g. java.math.BigDecimal* for *ERAType*). This phenomenon is further complicated in the case of restriction by enumeration, for JAXB did not create constants representing the possible values. Consequently, JAXB placed the burden on client code to avoid passing to the Java representation of *BatsType*, for example, a value of *both* rather than the valid *switch* [7].

The pattern associated with *PercentageType* is replicated in *mlb_locals.xsd* via the <*xs:union*> construct to represent a union of two regular expressions rather than the single one described above [8]. Unfortunately, JAXB ignored <*xs:union*> altogether as well.

3.1.2.5. <*xs:complexType*> with Extensions and with *abstract="true"*

As discussed, a global complex type has its own Java rendering subclassed by the Java rendering of the associated element declaration. Interestingly, the same is true for local complex type declarations as well. For example, in *mlb_locals.xsd*, the element *AmericanLeagueTeam* has its type locally declared, yet a Java interface and implementation combination was created for this type called *AmericanLeagueTeamType*—exactly as with *mlb_globals.xsd*. It would seem that JAXB generates Java artifacts for locally declared types and names them by appending *Type* to the associated element name. The same inheritance relationship was maintained as well.

Extensions of complex types worked very well in JAXB. The base complex type was manifested as Java artifacts, which then were subclassed by the complex-type representations that derived from them. For example, in *mlb_globals.xsd*, *BaseRosterType* is extended by *AmericanLeagueRosterType* (or its local anonymous equivalent in *mlb_locals.xsd*), and the same is exactly true of their corresponding Java representations. As a result, the data and behavior associated with the *BaseRosterType* implementation are completely inherited by the *AmericanLeagueRosterType* implementation.

It should be noted that both *BaseRosterType* and *PlayerType* have *abstract="true"* in both schemas. However, neither the Java class *BaseRosterTypeImpl* nor *PlayerTypeImpl* is declared an *abstract* class. Therefore, because each class is concrete and declared to have *public* access, it is possible for client Java code to violate the spirit of the schema from whence it came. This is hardly ideal however unlikely it may be that client code will be instantiating these classes directly.

3.1.2.6. Cardinality for Elements Beyond the Default Value of 1

In both schemas, there are three cases of explicit cardinality. One was mentioned before—the choice involving seven or eight *BenchPlayer* elements within the *AmericanLeagueRosterType* element. In addition, the *AmericanLeagueTeam* element must appear fourteen times in a valid instance, and the *NationalLeagueTeam* element which must appear sixteen times in a valid instance. Both are children of the root element *MajorLeagueBaseball*.

JAXB failed to enforce cardinality effectively in both cases. In the case of the *MajorLeagueBaseball* element, the corresponding Java class maintained instances of *java.util.ArrayList* to store instances of the *AmericanLeagueTeamImpl* and *NationalLeagueTeamImpl* classes. Yet the *MajorLeagueBaseballImpl* class performed no validation checks to ensure that the number of instances in the respective lists remained within the upper bound. As a result, client code was able to add items to the respective lists without any regard for the constraints defined by the schema.

3.1.2.7. Atypical Data Types

The only atypical XML Schema data type found in either schema is <*xs:nonNegativeInteger*>, which is used for such elements as *HomeRuns* and *RBI*. JAXB simply mapped this to *java.math.BigInteger* with no checks on inputs to mutators to verify that the values are indeed non-negative. Therefore, it was possible to set a value for *HomeRuns* to be -25, for example. This behavior also violated the spirit of the schema and placed the production of a valid XML instance in the trust of the client.

[7] It is common in the vernacular of baseball to list a switch hitter as one who bats “both.”
[8] The two expressions are *.[0-9]{3}* and the literal *1.000*, which are combined in the regular expression found in *mlb_primary.xsd*.

3.1.2.8. Attributes and <*xs:attributeGroup*>

Attribute declarations were represented as private data members within those Java classes representing the elements containing them, and these members were manipulated through accessors and mutators—albeit with the flaws in enforcing data type restrictions described previously. For example, the *commissioner* attribute of type *xs:string* belonging to the *MajorLeagueBaseball* element could be accessed through the *getCommissioner()* method in the *MajorLeagueBaseball* interface.

With regard to <*xs:attributeGroup*>, JAXB simply handled the attributes contained therein as if they were declared separately within the containing element. The grouping of the *name* and *manager* attributes within the *AmericanLeagueTeam* and *NationalLeagueTeam* elements in *mlb_primary.xsd* was handled exactly as if each were defined individually therein.

3.1.3. Marshalling

JAXB marshalled the Java object model into a well-formed XML instance document that preserved the hierarchies and sequences of elements found in the original document. Moreover, changes made to the Java object model representation of the original XML instance were in fact reflected in the new instance. The declaration of the *XMLSchema-instance* namespace and associated mapping to the conventional *xsi* prefix did not happen by default. A setting was applied in code in order to map *xsi* to the *XMLSchema-instance* namespace and specify the value of the *xsi:schemaLocation* attribute.

Given its conspicuous lack of rigor in validating changes made to the object model, it should come as no surprise that JAXB did not produce a valid XML instance. If, for example, a change to the alternate was made within even the trivial <*xsd:choice*> construct, JAXB did not toggle the choices effectively and did not yield a valid instance.

Furthermore, JAXB-generated code did not validate data against faceted restrictions on simple types. Invalid data were permitted into the object model and ultimately into the derived instance. This negligence had even greater significance in combination with the use of the *BigDecimal* class. Consider the *BattingAverage* element and associated JAXB class. Upon unmarshalling, all the values for *BattingAverage* were denoted by *java.lang.String* representations of the corresponding *BigDecimal* instances. The *pattern* facet restricting the appearance of these decimal values was never enforced. Therefore, upon marshalling, all of the *BattingAverage* data—even those unchanged from the original instance—were corrupted. Thus, an original *BattingAverage* value of *.287* remaining otherwise unchanged during object manipulation became *0.287* in the generated instance, which is invalid according to the *pattern* facet. A much more gross violation occurred when setting the value of a *BattingAverage* element with a *BigDecimal* object instantiated with a Java double primitive as permitted by the *BigDecimal* API. An attempt to set a *BattingAverage* value to *.299* with a *double* produced the following value in the new XML instance: *0.298999999999999988009591334048309363424777984619140625*. Although it may be argued that using the *pattern* facet with *xs:decimal* is rare, such a glaring violation is intolerable and one that could easily be resolved.

Finally, the problems JAXB had with cardinality provided yet another basis for invalid instances. It was possible to add *AmericanLeagueTeam* and *NationalLeagueTeam* objects in code well beyond the constraints set by the cardinalities defined in the schemas—as if the upper bounds were both defined to be *unbounded*. These overflows were permitted in the final result which left *MajorLeagueBaseball* invalid.

3.2. Usability

JAXB was observed for its usability by developers seeking to leverage the power of XML data binding. In particular, usability was assessed in four areas: generation of the object model, unmarshalling, code manipulation, and marshalling.

3.2.1. Generation of the Object Model

JAXB[9] generates a Java object model from an XML Schema document only through a command-line interface. The command is the following:

```
xjc [-options ...] <schema>
```

where the list of options is available in the Java Web Services Tutorial available from Sun [The Java Web Services Tutorial]. The JAXB distribution includes scripts for executing this command (*xjc.bat* for Windows and *xjc.sh* for Unix), but such an approach is not portable. Far preferable would be the distribution of an Ant task that executes an equivalent of the above command [10]. Though it is not readily apparent from *The Java Web Services Tutorial*, JAXB does indeed ship with an Ant task mapping to the class *com.sun.tools.xjc.XJCTask* [Using XJC with Ant].

There is also no direct programmatic interface to JAXB's code generation capability. It was possible for the purposes of this discussion to create one, but such an effort demands at least intermediate experience with Java programming and Java artifacts like JAR files [11]. It is a significant mark against the usability of JAXB that developers do not have an obvious choice between command-line and programmatic interfaces.

JAXB allows customization of its default behavior through two means [The Java Web Services Tutorial]. One is by means of XML Schema annotations, which contain proprietary JAXB XML tags defining customization of default binding behavior. This is a very poor approach because it couples the schema to JAXB. The better approach is to create an XML file, with a *.xjb* extension by convention, that specifies how default behavior is to be overridden. The degree of customizability is reasonable, but it lacks the real fine-grained control developers crave.

3.2.2. Unmarshalling

Unmarshalling an XML instance was reasonably straightforward with JAXB, but it did require multiple steps. Two classes were required from the *javax.xml.bind* package: *JAXBContext* and *Unmarshaller*. A context was required for handling classes in a certain package root (*e.g. org.xmlconference.xmldatabinding.jaxb.primary*), and the *Unmarshaller* instance was obtained from the context. The *Unmarshaller* then read the XML file and returned an *Object* reference to the root element interface—in this case *org.xmlconference.xmldatabinding.jaxb.primary.MajorLeagueBaseball*. Thus, a cast was required to the appropriate type. Awkward and costly in terms of performance, casts should be all but nonexistent in elegant OO code. Moreover, *javax.xml.bind.JAXBException* and *java.io.FileNotFoundException* demanded handling. The proliferation of checked exceptions is less than desirable as well, but the effect was mitigated by their being standard Java exceptions.

3.2.3. Object Manipulation

CRUD operations with the JAXB object model were rather intuitive. Creation of new objects to add to the model and ultimately to the marshalled document was accomplished through the *org.xmlconference.xmldatabinding.jaxb.primary.ObjectFactory* class produced by the code generation process. Despite the name, the *ObjectFactory* was most certainly not an implementation of any of the Factory Patterns articulated by the Gang of Four [12]. It did, however, have the similar aim of decoupling client code from implementation details. *ObjectFactory* contained *create* methods for all of the interface types—representing element and type declarations—defined in the object model. For example, *ObjectFactory* contained such methods as *createAmericanLeagueTeam()* and *createAmericanLeagueTeamType()* returning instances of the *AmericanLeagueTeam* and the *AmericanLeagueTeamType* interfaces respectively. As a result, the client remained blissfully unaware of the actual implementation classes—*AmericanLeagueTeamImpl* and *AmericanLeagueTeamTypeImpl*.

[9]See Appendix 3 for code samples.
[10]Ant [http://ant.apache.org/] is an open-source build tool that enjoys tremendous popularity in the Java community.
[11]Java ARchive files are libraries of executable Java code.
[12]The Gang of Four is the colloquial name for Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. They are the authors of *Design Patterns: Elements of Reusable Object-Oriented Software*, the seminal text on OO design and development. The creational patterns described therein are Abstract Factory, Builder, Factory Method, Prototype, and Singleton.

From a code quality perspective, the creation paradigm administered by *ObjectFactory* left much to be desired. Because all the classes in the *impl* package were declared *public*, JAXB allowed client code the option to bypass *ObjectFactory* altogether and create concrete instances of its own accord. To do so would produce the very dependencies that *ObjectFactory* is in place to avoid. As a result, JAXB demonstrated the potential to allow inexperienced developers to write code that will prove rigid over time [13].

Greater potential for rigidity arose from a curious violation of the venerated OO principle of encapsulation. Sequences of elements with maximum cardinality greater than one were represented, as mentioned before, as *java.util.List* objects. That is an implementation detail, yet it was exposed through the public API. For example, *AmericanLeagueRoster* had a method called *getAmericanLeaguePitcher()*, which returned a *List* of *AmericanLeaguePitcher* objects. Absence of cardinality validation aside, this was an unacceptable level of coupling between client code and the *AmericanLeagueRoster* interface. Should the implementation change, countless client modules will be forced to change as well. JAXB should have gone farther toward greater abstraction through recognition of the benefits of encapsulation.

Read and update functions were accomplished through the interface representing the root element. In this case, that interface was *MajorLeagueBaseball*. Beginning from the root element interface, intuitive accessor methods enabled client code to retrieve information from the unmarshalled XML document. Mutator methods existed for updating data as well, although they did so in a manner that belied XML's rich validation capability.

Deletes are not generally applicable with XML data binding. The only exception arises with collections whose contents may be altered—implemented by JAXB with the aforementioned *List* objects. The public API for *List* provides a method called *removeAll()*, which purges the collection of items matching those in another collection, and overloaded *remove()* methods, which purge individual items in the collection. Aside from the danger of exposing the implementation details to the client, JAXB also did not validate the results of these method calls upon marshalling.

3.2.4. Marshalling

Marshalling an object model into an XML document is accomplished through the *marshal()* method of the *javax.xml.bind.Marshaller* class, an instance of which was obtained from the *JAXBContext*. Once the *MajorLeagueBaseball* instance was modified to the satisfaction of the business requirements, it was passed to the *Marshaller* instance along with a *java.io.FileInputStream* object. The result was a well-formed XML instance with the proper sequencing of elements, but as discussed at length previously, the document could not be trusted to be valid.

By default, *Marshaller* produces an XML instance that lacks human-readable formatting and any mention of the associated schema. To that end, it was necessary to set two properties of the *Marshaller* instance—*Marshaller.JAXB_FORMATTED_OUTPUT* to *java.lang.Boolean.TRUE* and *Marshaller.JAXB_SCHEMA_LOCATION* to a *java.lang.String* value representing the desired value for the *xsi:schemaLocation* attribute [14]. The performance cost for creating indentation in the output was found to be minimal and well worth the gain in readability.

Altogether, the marshalling process required only six lines of code including the setting of the properties. It is good that *Marshaller* does not apply indentation by default and allows the developer to accept the performance loss, but it would seem that applying the same value for *xsi:schemaLocation* or *xsi:noNamespaceSchemaLocation* (should either be present) as the original XML instance would make for more intuitive default behavior.

3.3. Next Steps

With the evaluation of Sun's JAXB XML data binding tool complete for the purposes of this discussion, it is now time to perform the same rigorous analysis on its leading open-source counterpart, Castor.

[13]The term *rigid* comes from Robert C. Martin's excellent text *Agile Software Development: Principles, Patterns, and Practices*. According to Martin, rigidity is the tendency for changes in one software module to force changes in its client modules, which themselves force changes in their client modules, and so on until the refactoring effort becomes far more arduous than originally expected [Martin].
[14]The *Marshaller* class also has a *Marshaller.JAXB_NO_NAMESPACE_SCHEMA_LOCATION* property.

4. Evaluation Results: Castor

This portion of the discussion examines the performance of Castor as an XML data binding tool and its usability by client code. Comparisons to JAXB are made throughout, and the relative strengths and weaknesses of both are assessed.

4.1. Bidirectional Integrity

As with JAXB, Castor was observed for its faithfulness to the intent of the W3C XML Schema documents from the initial generation of the object model through unmarshalling of the XML instance and marshalling of new XML instances from the underlying object model. Integrity comparisons of the two tools pervade the discussion.

4.1.1. Generation of the Object Model

Like JAXB, Castor tested both schemas and the instance document for well-formedness and validity and reported errors with both by throwing Java exceptions. This was as expected. Where Castor differed from JAXB is the manner in which package names for generated classes were defined, for the burden was entirely the developer’s. Developers would be better served if Castor took the JAXB approach—declaring package names according to the target namespace of the schema by default and allowing developers to override as necessary.

Castor also produced a far different structure for its generated object model. For every global element and type declaration, two classes were created. One was a Java representation of the element or type analogous to that produced by JAXB. The other was a *Descriptor* class containing the metadata for binding the XML representation to the Java one—including validation. For example, the aforementioned *AmericanLeagueTeam* element in *mlb_primary.xsd* was rendered into both an *org.xmlconference.xmldatabinding.castor.primary.AmericanLeagueTeam* and *org.xmlconference.xmldatabinding.castor.primary.AmericanLeagueTeamDescriptor* class. Developers are unlikely to interact with a *Descriptor* class directly, so it is less than ideal that they were publicly accessible. Moreover, while there were two public concrete classes for each global declaration, Castor, unlike JAXB, did not generate any interfaces. This constrained the object model and limited the testability of the generated classes.

With regard to the relationship of this element with its type as manifested in the object model, *AmericanLeagueTeam* subclassed *AmericanLeagueTeamType*, which had a compositional relationship with *AmericanLeagueRoster*. This in turn subclassed *AmericanLeagueRosterType*. The following UML diagram conveys the relationships among the *AmericanLeagueTeam*, *AmericanLeagueTeamType*, and *AmericanLeagueRosterType* classes:

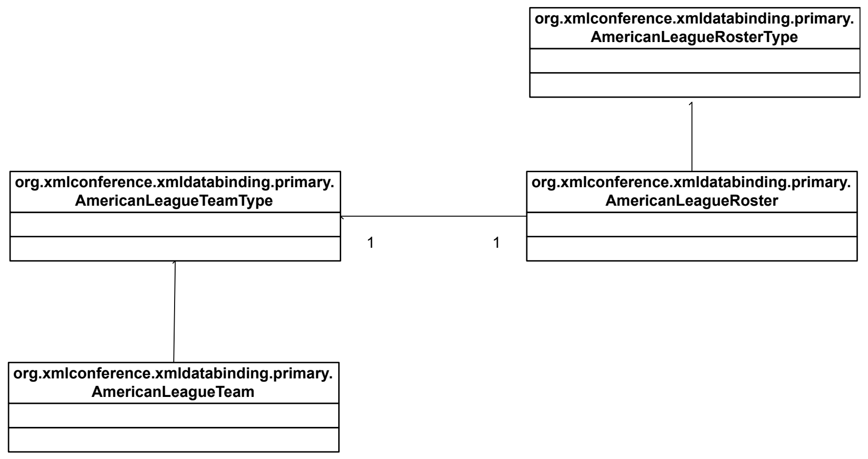
org.xmlconference.xmldatabinding.primary.
AmericanLeagueRosterType
org.xmlconference.xmldatabinding.primary.
AmericanLeagueTeamType
org.xmlconference.xmldatabinding.primary.
AmericanLeagueRoster
1
1
org.xmlconference.xmldatabinding.primary.
AmericanLeagueTeam

Figure 3. UML Diagram of Relationships among Element and Type Representations Generated by Castor from *mlb_primary.xsd*

While there is large similarity to JAXB, the distinction is clear. The simplicity of the UML diagram derives from the lack of abstractions. In fact it is interesting to note that Castor built relationships among concrete classes--and more specifically, among subclasses rather than superclasses. This is hardly a good example of OO-design best practices.

In the other schema, *mlb_locals.xsd*, the object model was smaller because there are fewer global declarations to render into Java objects. Castor avoided the use of inner classes to represent the local declarations; rather, the code associated with them was folded into the container class. For example, the code found in *org.xmlconference.xmldatabinding.castor.primary.AmericanLeagueTeamType* was found instead in the element class *org.xmlconference.xmldatabinding.castor.locals.AmericanLeagueTeam* when generating code from *mlb_locals.xsd*. As the following UML diagram illustrates, this simplifies matters greatly.

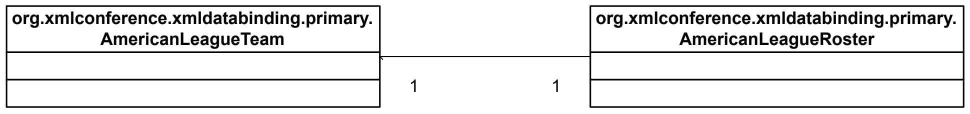
org.xmlconference.xmldatabinding.primary.AmericanLeagueTeam
org.xmlconference.xmldatabinding.primary.AmericanLeagueRoster
1
1

Figure 4. UML Diagram of Relationships among Element and Type Representations Generated by Castor from *mlb_locals.xsd*

Despite the continued insistence upon details over abstractions, this strategy was far superior to JAXB's, for it avoided the awkwardness and security vulnerabilities associated with inner classes.

Finally, it is important to recognize a critical distinguishing feature of Castor from JAXB. Beneath the top-level package, Castor created a package called *types*. Contained therein were classes representing all simple-type declarations restricted with *xs:enumeration*. Consequently, for both schemas, the *types* subpackage contained classes called *BatsType*, *RoleType*, and *ThrowsType* (each paired with a *Descriptor* class as described above) [15]. It became clear that the purpose of these classes was to enforce the validation constraints imposed by the respective enumeration constructs in the parent schema. While the function of these classes will be detailed shortly, this was the first evidence of Castor's superior validation capability.

4.1.2. Unmarshalling and Object Manipulation

Analysis of the unmarshalling strategy of Castor revealed similarities to JAXB but numerous key differences as well. The following sections detail the manner in which Castor unmarshalled these constructs as found in *mlb.xml*.

4.1.2.1. XML Schema Data Types

The mapping of XML Schema data types to Java data types was largely the same but with an important difference.

[15]In the case of *mlb_locals.xsd*, the *BatsType* and *RoleType* classes were created from anonymous simple-type declarations for the elements *Bats* and *Role* respectively. Apparently, Castor named the classes by appending *Type* to the containing element name.

XML Schema Data Type	**Java Data Type**
xs:string	*java.lang.String*
xs:integer	*int* primitive
xs:nonNegativeInteger	*int* primitive
xs:decimal	*java.math.BigDecimal*

Table 3. Castor Data Type Mappings

Where Castor differed was in the use of the Java primitive type *int* to represent the *xs:integer* and *xs:nonNegativeInteger* types. This default behavior is more effective because primitives require fewer resources than Java objects and are easier to manipulate in code, and they will likely serve the majority of use cases where XML is directly involved. Moreover, an *int* can be converted into a Java object as requirements dictate—either through unmarshalling via Castor customization or through business logic code at some point after unmarshalling. Therefore, this is preferred to the JAXB approach. Of course, the mapping of *xs:decimal* to *BigDecimal* renders Castor vulnerable to the same problems as JAXB.

4.1.2.2. <*xs:sequence*>

The behavior of Castor is identical to that of JAXB with regard to the modeling of <*xs:sequence*>. The element declarations contained within a sequence were represented through composition in the ensuing Java code and manipulated through accessor and mutator methods.

4.1.2.3. <*xs:choice*>

As mentioned previously, the schemas feature two instances of <*xs:choice*>. Unlike JAXB, Castor took significant measures to address this construct—albeit incompletely. The simpler choice, where the first child of the root element *MajorLeagueBaseball* (of type *MLBType*) may be either *Year* (of type *xs:integer*) or *Designator* (of type *xs:string*), was represented by Castor through a Java class of its own called *MLBTypeChoice*. An instance of this class was obtained through the *getMLBTypeChoice()* method of the *MajorLeagueBaseball* class, and it managed the choice between the two elements with two mutators. *MLBTypeChoice* had a method called *deleteYear()*, which informed the class of the desire to toggle from rendering the *Year* element to rendering the *Designator* element. It was already clumsy to have client code perform the toggle, but if that was to be the *modus operandi*, it is inexplicable that there was no comparable *deleteDesignator()* method. As a result, Castor could shift the choice from *Year* to *Designator*, but it was incapable of effecting the reverse.

With the more complex choice, where the complex type *AmericanLeagueRosterType* contains a choice between either eight *BenchPlayer* elements *or* a sequence of one *DesignatedHitter* element and seven *BenchPlayer* elements, Castor created two classes to address these cases. One was *AmericanLeagueRosterTypeChoice*, which had a method called *addBenchPlayer()* that performed validation to ensure the size of the container of *BenchPlayer* elements did not exceed eight. The second class was *AmericanLeagueRosterTypeChoiceSequence*. It also had an *addBenchPlayer()* method, which performed validation to ensure instead that the size of the container of *BenchPlayer* elements did not exceed seven, and a *setDesignatedHitter()* method. Though the mechanisms were in place to validate the choice between the two sets of elements, there was nothing analogous to the *deleteYear()* method to explicitly toggle between the two choices. As a result, the Castor machinery was unable to make sense of the developer's desired choice. The typical result was report of a validation error—too many *BenchPlayer* instances in either case—when there should have been none.

4.1.2.4. <*xs:simpleType*> with Restrictions

As a reminder, the following table summarizes the instances of <*xs:simpleType*> with different restrictions as they are found in *mlb_primary.xsd* [16].

[16]In the case of *mlb_locals.xsd*, some of these types are local to their respective elements and are therefore anonymous.

Simple Type Name	**Base Type**	**Restriction Type**	**Value(s)**
YearType	*xs:integer*	*xs:minInclusive*	*2005*
ERAType	*xs:decimal*	*xs:fractionDigits*	*2*
PitcherRoleType	*xs:string*	*xs:enumeration*	*starter, reliever*
BatsType	*xs:string*	*xs:enumeration*	*left, right, switch*
ThrowsType	*xs:string*	*xs:enumeration*	*left, right*
PercentageType	*java.lang.String*	*xs:pattern*	*.[0-9]{3}|1.000*[17]

Table 4. Simple Type Declarations

Overall, Castor achieved far greater success in validating these constructs than JAXB, but there remained a delta between Castor's performance and the rigorous validation endemic to XML. Castor did indeed enforce the minimum value for *YearType*. Also, Castor went a great deal farther than JAXB at handling enumerations. Classes devoted to enumeration processing were maintained in the aforementioned *types* subpackage, and these classes contained identically typed Java constants representing the enumeration values. For example, the *BatsType* simple type was represented as a class called *BatsType* with static constants called *LEFT*, *RIGHT*, and *SWITCH*—all of type *BatsType* as well. Where this had tremendous significance was with regard to the mutators whose signatures demanded *BatsType* objects. In order to set the value of *Bats* (the element whose type is *BatsType*) for such an element as *FirstBaseman*, the argument to the method had to be of type *BatsType*. This shifted the validation burden from client code to Castor in that it guaranteed that only one of the enumerated values would be passed. Castor used the same approach for the other enumerations as well.

Despite its success with the validation of those restrictions, Castor disappointed with regards to the *xs:fractionDigits* and the *xs:pattern* restrictions. It ignored them altogether and thus performed no better than JAXB. Moreover, Castor was just as lethargic in enforcing the <*xs:union*> construct associated with *PercentageType* replicated in *mlb_locals.xsd*.

4.1.2.5. <*xs:complexType*> with Extensions and with *abstract="true"*

Much has been discussed already regarding the rendering of global complex types into Java code by Castor. In a manner similar to JAXB, each global complex type had its own Java representation that served as the superclass of the Java representation of the associated element declaration. The key difference was the absence of abstraction through interfaces. In contrast, Castor injected the code related to anonymous local complex types inside the code related to the associated element declaration.

Extensions of complex types worked in Castor exactly as in JAXB with regard to global type declarations. The base complex type was manifested as a Java class, which was subsequently subclassed by the complex-type representations that derived from them. Localization, however, was a factor in how this was achieved. For example, in *mlb_globals.xsd*, the type declaration *BaseRosterType* is extended by the type declaration *AmericanLeagueRosterType*, and the relationship between the corresponding Java representations was exactly parallel. In *mlb_locals.xsd*, the type declaration *BaseRosterType* is extended by the anonymous local type declaration for the *AmericanLeagueTeam* element. In this case, it was the *AmericanLeagueTeam* class that subclassed the *BaseRosterType* class. In both cases, the data and behavior associated with the *BaseRosterType* implementation were completely and accurately inherited. It was interesting to find the Java manifestation of an XML Schema element subclassing the Java manifestation of an XML Schema super type.

Castor did a much better job than JAXB at recognizing that *BaseRosterType* and *PlayerType* are declared *abstract* in both schemas. Both implementations were declared to be abstract Java classes, and consequently, it was impossible for client code to instantiate these directly. This was very much in the spirit of the schemas from whence these objects came.

[17]Regular expression representing a decimal with no leading digit and three digits to the right of the decimal point or the literal value 1.000. Possible values include .331, .067, and 1.000. This type is used for such elements as *BattingAverage*.

4.1.2.6. Cardinality for Elements Beyond the Default Value of 1

As mentioned in the JAXB discussion, each schema features two cases of explicit cardinality. One is the choice within *BaseRosterType* between seven or eight *BenchPlayer* elements as described previously. The other applies to the *AmericanLeagueTeam* element, which must appear fourteen times in a valid instance, and the *NationalLeagueTeam element*, which must appear sixteen times in a valid instance. Both of these are children of the root element *MajorLeagueBaseball*.

Castor enforced these cardinal values at two points. The first of these was during interaction with the object model. Each element with multiple cardinality was maintained in a *java.util.Vector* container within the parent Java representation, and checks were in place to ensure the size of the *Vector* never exceeded the maximum cardinality. Consider once again the *MajorLeagueBaseball* and *AmericanLeagueTeam* elements. The corresponding *MajorLeagueBaseball* class held a *Vector* of *AmericanLeagueTeam* elements. Moreover, the API for *MajorLeagueBaseball* had an *addAmericanLeagueTeam()* method, which was written by Castor to throw a *java.lang.IndexOutOfBoundsException* if an attempt was made to call the method when the *Vector* was at the maximum capacity of fourteen.

Interestingly, the *MajorLeagueBaseball* API also contained methods called *removeAllAmericanLeagueTeam()*, which emptied the *Vector* of *AmericanLeagueTeam* elements, and another method called *removeAmericanLeagueTeam()*, which removed an *AmericanLeagueTeam* instance at a given index within the *Vector*. While such methods are necessary for the flexibility of modifying the unmarshalled XML instance, they also expose the model to the risk of violating the rules for *minimum* cardinality of the elements. The *MajorLeagueBaseball* class could not oblige client code to meet the minimum for *AmericanLeagueTeam* instances. Indeed, client code should be free to purge the container entirely and replace the contents therein at its leisure. Furthermore, there is technically no violation until an attempt is made to marshal *MajorLeagueBaseball* with an insufficient number of *AmericanLeagueTeam* instances. Hence, the challenge of enforcing minimum cardinality is not trivial.

To understand how Castor tackled this challenge, recall that for each Java class representing an element or type in the schema, Castor generated a companion *Descriptor* class charged with, among other things, enforcing validation upon marshalling. This was precisely where Castor enforced minimum cardinality—albeit with some academic differences related to localization. In *mlb_globals.xsd*, the *MajorLeagueBaseball* element is defined to be of type *MLBType*, which includes the *AmericanLeagueTeam* element and is represented with the Java classes *MLBType* and *MLBTypeDescriptor*. In *mlb_locals.xsd*, *MajorLeagueBaseball* is defined to be of an anonymous type which includes the *AmericanLeagueTeam* element and is represented within the Java classes *MajorLeagueBaseball* and *MajorLeagueBaseballDescriptor*. In either case, the *Descriptor* classes contained code checking for the minimum cardinality of elements. When a violation occurred, Castor threw a proprietary exception to the caller. Although the proliferation of checked exceptions—particularly proprietary ones—is problematic, the rigor of validation was impressive.

4.1.2.7. Atypical Data Types

As a reminder, the only atypical XML Schema data type found in either schema is *xs:nonNegativeInteger*, which is used for such elements as *HomeRuns* and *RBI*. Castor simply mapped this to an *int* primitive with no checks on inputs to mutators to verify that the values were indeed non-negative. Therefore, while the default data-type mapping strategy was different from JAXB's, the validation outcome was no different.

4.1.2.8. Attributes and <*xs:attributeGroup*>

Castor also performed exactly as JAXB with regard to attributes and attribute groups. Attribute declarations were represented as private data members within those Java classes representing the elements containing them, and these members were manipulated through accessors and mutators—albeit with the flaws in enforcing some data type restrictions as described previously.

With regard to <*xs:attributeGroup*>, JAXB simply handled the attributes contained therein as if they were declared separately within the containing element. The grouping of the *name* and *manager* attributes within the *American-*

LeagueTeam and *NationalLeagueTeam* elements in *mlb_primary.xsd* was handled exactly as if each were defined individually therein.

4.1.3. Marshalling

Like JAXB, Castor marshalled the Java object model into a well-formed XML instance document that preserved the hierarchies and sequences of elements found in the original document. Moreover, changes made to the Java object model representation of the original XML instance were in fact reflected in the new instance. However, the declaration of the *XMLSchema-instance* namespace and associated mapping to the conventional *xsi* prefix did not happen by default. A setting must be applied in code in order to map *xsi* to the *XMLSchema-instance* namespace and specify the value of the *xsi:schemaLocation* attribute.

Although it proved far superior in many ways to JAXB in its rigor in validating changes made to the object model, Castor could not always be counted upon to produce a valid XML instance. Castor did indeed excel in the enforcement of cardinality and in managing enumerations and boundary values with the *minInclusive* restriction. To the extent to which a schema contains only these sorts of constraints, a marshalled instance will likely be valid. Recall, however, that Castor's performance in handling the *<xs:choice>* construct was suspect at best. Moreover, while Castor avoided the vagaries of Java's *BigInteger* class, it still fell prey to those of the *BigDecimal* class as JAXB did. The failure to resolve the *pattern* facet in concert with the *BigDecimal* class did not escape Castor either. Yet for all its shortcomings, it is quite clear that Castor far surpassed JAXB in its faithfulness to the intent of XML Schema constructs.

4.2. Usability

Like JAXB, Castor also was observed for its usability by developers seeking to leverage the power of XML data binding. In particular, usability was assessed in four areas: generation of the object model, unmarshalling, code manipulation, and marshalling. Usability comparisons of the tools in each category pervade the discussion.

4.2.1. Generation of the Object Model

Castor[18] generates a Java object model from an XML Schema document through the execution of a particular Java class called *org.exolab.castor.builder.SourceGenerator*. Therefore, code can be generated either at the command-line interface or in code. The command is the following:

```
java org.exolab.castor.builder.SourceGenerator [-options ...]
```

where the list of options is available in the Castor documentation [Castor API]. Because it is really a Java runtime command, the command-line interface demands an understanding of how Java bytecode is executed. Even Java developers, who are almost always spared the details of Java execution by their development environments, may require a refresher on how to do this. A JAR command like the one provided by JAXB, or an Ant script with a built-in target to accomplish the same, would be preferable.

Still, *SourceGenerator* does provide an intuitive programmatic interface with which developers may generate the object model. A simple call to one of several overloaded *generateSource()* methods produced an object model from a schema. The API for the class also has numerous methods for configuring the code-generation process. Among the most notable of these are *setNamespacePackageMapping()*, which allows client code to define a mapping of XML namespaces to Java packages, and *setEqualsMethod()*, which compels Castor to override the *equals()* method inherited by each generated class from *Object*. Doing so is recommended for all Java classes, regardless of whether they are created by developers or tools. Still, advanced Java developers are aware that overriding the *equals()* method should be coupled with overriding the *hashCode()* method also inherited from *Object*, yet Castor does not provide a comparable means for doing so. This oversight is really just academic since it is a rare use case that demands the storage of XML binding objects in hashtables, but it would appear an oversight nonetheless.

[18]See Appendix 4 for code samples.

Castor customization is also available through two other means: a Java *properties* file and a proprietary XML file similar to the binding files utilized by JAXB [19]. Both allow fine-grained customization down to the class level.

4.2.2. Unmarshalling

Unmarshalling an XML instance was very simple with Castor. Client code has two choices, each with its own advantages and disadvantages. One approach is to call *unmarshal()*, a *static* method which Castor generated for every Java class representation of an element or type declaration, on the class representing the root element. Thus simply calling *unmarshal()* on the *MajorLeagueBaseball* class unmarshalled the entire XML instance. It required only a single line of code but left no room for configuration.

The other approach is to call *unmarshal()* on an instance of the *Unmarshaller* class. This instance can be configured through various API calls and is far better suited to advanced users, who may be willing to accept more verbose and esoteric code in exchange for a greater degree of sophistication [Castor API]. Among the configuration possibilities are whitespace preservation and validation.

Both approaches, however, share a common disadvantage. They throw proprietary exceptions upon error. This is a very subtle but all too common violation of the principle of encapsulation, for client code forced to address these exceptions becomes coupled to Castor. The burden is on developers to make a critical design decision—either to provide a layer of abstraction between Castor and client code or to do nothing with the hope that the increased coupling does not make change more arduous than it need be should the circumstance arise.

4.2.3. Object Manipulation

Castor takes a patently different approach to CRUD operations with its object model than JAXB. Creation of new objects to add to the model and ultimately to the marshalled document is accomplished primarily through creating instances with the *new* operator in a manner familiar even to the most novice Java developer, and this new object is populated with data through API calls on the newly created instance [20].

The manner in which Castor managed collections of objects is particularly noteworthy for its advantages and disadvantages. Unlike JAXB, which exposed its internal use of *java.util.List* to its clients, Castor exposed only generic collections and offered a distinct advantage over its counterpart. For example, the *MajorLeagueBaseball* class had one method called *getAmericanLeagueTeam()*, which returned an array of *AmericanLeagueTeam* objects, and another method called *enumerateAmericanLeagueTeam()*, which returned a *java.util.Enumeration* of *AmericanLeagueTeam* objects. Moreover, because arrays and *Enumerations* are immutable, additions were made to collections through *add* methods like the *addAmericanLeagueTeam()* method on *MajorLeagueBaseball*. Therefore, Castor kept the internal details of the collection (which is a *java.util.Vector* as mentioned before) hidden. This strategy decoupled the implementation from the client and thus insulated the client from changes therein.

Where Castor fell short of JAXB was in the collection mechanism itself, for the use of *Vector* is a poor choice. According to its API, each *Vector* method is declared to be *synchronized*, a Java keyword mandating that only a single thread execute the method at a time [J2SE API]. Synchronization incurs huge runtime overhead and is often unnecessary. It is for this reason that experienced Java developers often use *java.util.ArrayList*, which runs unsynchronized, when they require a flexible, linear collection mechanism. Recall that JAXB utilizes *ArrayList* for its collections, and these are certain to perform with far greater performance. It is true that Castor can be customized to utilize other Java collections like *ArrayList*, but its use of *Vector* is a weak default choice.

Similar to JAXB, Castor enabled read and update functions through the class representing the root element—in this case *MajorLeagueBaseball*. Beginning from the root element interface, intuitive accessor methods enabled client code to retrieve information from the unmarshalled XML document. Mutator methods existed for updating data as well,

[19] Java *properties* files simply contain a collection of key-value pairs such as *org.exolab.castor.builder.equalsmethod=true*, which is equivalent to calling *setEqualsMethod(true)* on *SourceGenerator* in code.
[20] Castor also has an *ObjectFactory* class similar to JAXB's, but it is of little use to all but the most sophisticated users.

and these were capable of performing some validation on such constructs as cardinality and enumerations while failing on others like patterns.

Deletes are not generally applicable with XML data binding. The only exception arises with collections whose contents may be altered. Castor provided *remove* methods to that end. For example, the *MajorLeagueBaseball* class had methods called *removeAllAmericanLeagueTeam()*, which purged the entire collection, and *removeAmericanLeagueTeam()*, which purged a single *AmericanLeagueTeam* instance at a specified point in the collection. These operations were ultimately validated upon marshalling as specified earlier.

4.2.4. Marshalling

Just as there were two alternative mechanisms for unmarshalling an XML instance, Castor offered two parallel alternatives for marshalling an object model into an XML instance, and each had its own advantages and disadvantages. One approach was to call the *marshal()* method, which Castor generated for every class representation of an element or type declaration, on the class representing the root element. Thus simply calling *marshal()* on an instance of the *MajorLeagueBaseball* class with an instance of *java.io.FileWriter* marshalled the entire XML instance. It required only a few lines of code but left no room for configuration.

The other approach was to call *marshal()* on an instance of the *Marshaller* class. This instance could be configured through various API calls and is far better suited to advanced users, who may prefer more verbose and esoteric code in exchange for a greater degree of sophistication. Among the configuration possibilities are insertion of the XML declaration, insertion of the *schemaLocation* attribute (or *noNamespaceSchemaLocation* attribute), and validation.

Also parallel with the unmarshalling mechanisms was the proclivity for proprietary checked exceptions in the marshalling code. Once more Castor betrays the principle of encapsulation and compels developers to choose between providing their own layers of abstraction or doing nothing and hoping that nothing changes.

4.3. Next Steps

With the evaluation of JAXB and Castor complete for the purposes of this discussion, a summary comparison of the tools and suggestions for further research are in order.

5. Summary and Concluding Remarks

XML data binding offers the possibility of seamlessly integrating XML hierarchies with OO encapsulation and inheritance. The promise for the future of E-business is immeasurable, but the challenge of harmonizing these fundamentally different data management strategies is daunting at best.

With the promise too great to ignore, several XML data binding tools have emerged claiming to realize it. Over the course of this discussion, two leading tools written in Java, JAXB and Castor, were evaluated with painstaking scrutiny to compare their performance in this space.

The evaluation criteria were selected because they were deemed to be of greatest interest to both XML analysts and Java developers. From a client usability standpoint, both tools were, despite certain differences, largely comparable in their performance of the basic XML data binding operations: validation of the W3C XML Schema, object-model generation from the schema, validation and unmarshalling of the XML instance, manipulation of the marshalled instance, and finally unmarshalling into a new XML instance. However, one tool distinguished itself simply by being more aligned with the intent of XML, and that was Castor.

Castor managed to capture the spirit of many of the key constructs of XML Schema within the object model it generated. While handling constructs like sequences, attributes and attribute groups, and complex types with extensions, Castor also incorporated key validation concepts like cardinality, enumeration restrictions, and boundary restrictions in its object model. It also had rich customization capability. However, Castor was not perfect by any means. It failed to manage restrictions of simple types on *xs:pattern* and on <*xs:union*>. It did not quite handle the <*xsd:choice*> construct despite a valiant effort. The generated code, by virtue of its insistence upon relationships among concrete classes rather than abstractions and upon the proliferation of proprietary checked exceptions, might not necessarily be ideal from the perspective of a Java developer. Castor's default reliance on the *Vector* class for its collections rather than the far more efficient *ArrayList* was also a poor choice. Yet whatever its flaws, Castor was clearly superior to JAXB, which failed to account for any but the most basic XML Schema constructs. Although JAXB is a standard and presents some distinct advantages, its inability to capture the essence of XML Schema was disappointing. It is clear that Castor has earned its place at the forefront of XML data binding tools, but it still has far to go.

Whatever the relative quality of XML binding tools may be to date, there is no doubt that the possibilities offered by XML data binding are immense. Therefore, existing tools will evolve while new tools emerge, and each new generation of binding tools is likely to narrow the gap that much further between the data management philosophies of XML and OO technologies. Whether the gap is ever closed remains to be seen, but the technology holds exciting possibilities for the future of E-business.

6. Suggestions for Further Research

Although every attempt was made to provide a thorough analysis of JAXB and Castor, the power and complexity associated with XML Schema and OO technologies render virtually any analysis of XML data binding wanting. It is thus incumbent upon those with interest in this space to remain vigilant.

The following are suggestions for further research in the realm of XML data binding:

- Other tools besides JAXB and Castor like XMLBeans, XSDObjectGen, and *xsd.exe* from Microsoft .Net [netxml]
- Execution with countless other XML Schema constructs like <*xs:substitutionGroup*> and <*xs:unique*> (with its XPath components), other built-in data types like *xs:date* (and faceted restrictions thereupon), restrictions of complex types, *etc.*
- Execution with schema content that is included and imported—and namespace resolution in the latter case
- Resolution of collisions between declarations with the same name or between declarations named identically to reserved words in the OO programming language[21]
- Mapping of URN namespaces to Java package names
- Performance in terms of resource consumption and time
- The lessons XML Schema data binding tools may hold for WSDL data binding tools as web services continue to proliferate
- Database integration
- Detailed customization capabilities
- Quality of generated code in terms of testability, security, and the extent to which "Design Smells" are absent[22]

[21]As an example of the latter case, consider the type declaration *PlayerType*, which in both schemas contains a child element called *Throws*. Suppose instead that *PlayerType* contained an *attribute* called *throws*. This would be problematic because Java has a reserved word called *throws* as well, and the collision would have to be addressed through some sort of customization. Moreover, collisions like this can happen with any language. An attribute called *throw* (rather than *throws*) would face collisions with keywords in both Java and C#.

[22]Martin describes several "Design Smells" that are symptoms of poor code—or more specifically, code that is resistant to change in the face of changing requirements. These are Rigidity, Needless Complexity, Needless Repetition, Opacity, Fragility, Immobility, and Viscosity [Martin].

These are but a sample of the innumerable areas of study in XML data binding. It behooves both the XML and OO communities to explore all possible areas of study with as much zeal as possible, for binding tools could go far towards revolutionizing the development of enterprise applications at the core of E-business.

A. *mlb_globals.xsd*

```
<?xml version="1.0" encoding="UTF-8"?>
<!--
Baseline case with no particular idiosyncrasies.  General schema design best
practices are followed including 1) upper camel case for elements
and types, 2) lower camel case for attributes, 3) global
declarations of elements and types amd 4) local declarations of
attributes.
-->
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
           xmlns="http://www.xmlconference.org/xmldatabinding/primary"
         targetNamespace="http://www.xmlconference.org/xmldatabinding/primary"

           elementFormDefault="qualified"
           attributeFormDefault="unqualified">
   <xs:element name="MajorLeagueBaseball" type="MLBType"/>
   <xs:complexType name="MLBType">
      <xs:sequence>
         <!-- Using choice -->
         <xs:choice>
            <xs:element ref="Year" />
            <xs:element ref="Designator" />
         </xs:choice>
         <!-- Enforcing cardinality -->
         <xs:element ref="AmericanLeagueTeam" minOccurs="14" maxOccurs="14"/>

         <!-- Enforcing cardinality -->
         <xs:element ref="NationalLeagueTeam" minOccurs="16" maxOccurs="16"/>

      </xs:sequence>
      <xs:attribute name="commissioner" type="xs:string" use="optional"/>
   </xs:complexType>
   <xs:element name="Year" type="YearType" />
   <xs:element name="Designator" type="xs:string" />
   <xs:simpleType name="YearType">
      <xs:restriction base="xs:integer">
         <xs:minInclusive value="2005"/>
      </xs:restriction>
   </xs:simpleType>
   <!-- Pitcher -->
   <xs:element name="AmericanLeaguePitcher" type="PitcherType"/>
  <xs:element name="NationalLeaguePitcher" type="NationalLeaguePitcherType"/>

   <!-- Extension of abstract complex type -->
```

```
    <xs:complexType name="PitcherType">
       <xs:complexContent>
          <xs:extension base="PlayerType">
             <xs:sequence>
                <xs:element ref="ERA"/>
                <xs:element ref="Innings"/>
                <xs:element ref="Role"/>
             </xs:sequence>
          </xs:extension>
       </xs:complexContent>
    </xs:complexType>
    <!-- Extension of abstract complex type -->
    <xs:complexType name="NationalLeaguePitcherType">
       <xs:complexContent>
          <xs:extension base="PitcherType">
             <xs:sequence>
                <xs:element ref="BattingStatistics"/>
             </xs:sequence>
          </xs:extension>
       </xs:complexContent>
    </xs:complexType>
    <xs:element name="Innings" type="xs:nonNegativeInteger"/>
    <xs:element name="ERA" type="ERAType"/>
    <xs:element name="Role" type="PitcherRoleType"/>
    <!-- Restriction of simple type -->
    <xs:simpleType name="ERAType">
       <xs:restriction base="xs:decimal">
          <xs:fractionDigits value="2"/>
       </xs:restriction>
    </xs:simpleType>
    <!-- Restriction of simple type -->
    <xs:simpleType name="PitcherRoleType">
       <xs:restriction base="xs:string">
          <xs:enumeration value="starter"/>
          <xs:enumeration value="reliever"/>
       </xs:restriction>
    </xs:simpleType>
    <!-- Batter -->
    <!-- Extension of abstract complex type -->
    <xs:complexType name="BatterType">
       <xs:complexContent>
          <xs:extension base="PlayerType">
             <xs:sequence>
                <xs:element ref="BattingStatistics"/>
             </xs:sequence>
          </xs:extension>
       </xs:complexContent>
    </xs:complexType>
    <xs:complexType name="BattingStatisticsType">
       <xs:sequence>
          <xs:element ref="BattingAverage"/>
          <xs:element ref="HomeRuns"/>
          <xs:element ref="RBI"/>
```

```
         <xs:element ref="Runs"/>
         <xs:element ref="Bats"/>
      </xs:sequence>
   </xs:complexType>
   <xs:element name="BattingStatistics" type="BattingStatisticsType"/>
   <xs:element name="BattingAverage" type="PercentageType"/>
   <xs:element name="HomeRuns" type="xs:nonNegativeInteger"/>
   <xs:element name="RBI" type="xs:nonNegativeInteger"/>
   <xs:element name="Runs" type="xs:nonNegativeInteger"/>
   <xs:element name="Bats" type="BatsType"/>
   <!-- Restriction of simple type -->
   <xs:simpleType name="BatsType">
      <xs:restriction base="xs:string">
         <xs:enumeration value="left"/>
         <xs:enumeration value="right"/>
         <xs:enumeration value="switch"/>
      </xs:restriction>
   </xs:simpleType>
   <!-- Roster -->
   <xs:element name="Catcher" type="BatterType"/>
   <xs:element name="FirstBaseman" type="BatterType"/>
   <xs:element name="SecondBaseman" type="BatterType"/>
   <xs:element name="ThirdBaseman" type="BatterType"/>
   <xs:element name="Shortstop" type="BatterType"/>
   <xs:element name="LeftFielder" type="BatterType"/>
   <xs:element name="CenterFielder" type="BatterType"/>
   <xs:element name="RightFielder" type="BatterType"/>
   <xs:element name="DesignatedHitter" type="BatterType"/>
   <xs:element name="BenchPlayer" type="BatterType"/>
   <xs:complexType name="BaseRosterType" abstract="true">
      <xs:sequence>
         <xs:element ref="Catcher"/>
         <xs:element ref="FirstBaseman"/>
         <xs:element ref="SecondBaseman"/>
         <xs:element ref="ThirdBaseman"/>
         <xs:element ref="Shortstop"/>
         <xs:element ref="LeftFielder"/>
         <xs:element ref="CenterFielder"/>
         <xs:element ref="RightFielder"/>
      </xs:sequence>
   </xs:complexType>
   <!-- Extension of abstract complex type -->
   <xs:complexType name="AmericanLeagueRosterType">
      <xs:complexContent>
         <xs:extension base="BaseRosterType">
            <xs:sequence>
               <!-- Using choice -->
               <xs:choice>
                  <xs:sequence>
                     <xs:element ref="DesignatedHitter"/>
                     <!-- Enforcing cardinality -->
                     <xs:element ref="BenchPlayer" minOccurs="7"
                                 maxOccurs="7"/>
                  </xs:sequence>
                  <!-- Enforcing cardinality -->
                  <xs:element ref="BenchPlayer" minOccurs="8" maxOccurs="8"/>

               </xs:choice>
               <xs:element ref="AmericanLeaguePitcher" minOccurs="10"
                           maxOccurs="10"/>
            </xs:sequence>
         </xs:extension>
      </xs:complexContent>
   </xs:complexType>
   <!-- Extension of abstract complex type -->
   <xs:complexType name="NationalLeagueRosterType">
      <xs:complexContent>
         <xs:extension base="BaseRosterType">
            <xs:sequence>
               <!-- Enforcing cardinality -->
               <xs:element ref="BenchPlayer" minOccurs="8" maxOccurs="8"/>
               <!-- Enforcing cardinality -->
               <xs:element ref="NationalLeaguePitcher" minOccurs="10"
                           maxOccurs="10"/>
            </xs:sequence>
         </xs:extension>
      </xs:complexContent>
   </xs:complexType>
   <xs:element name="AmericanLeagueRoster" type="AmericanLeagueRosterType"/>
   <xs:element name="NationalLeagueRoster" type="NationalLeagueRosterType"/>
   <!-- Teams -->
   <xs:attribute name="manager" type="xs:string"/>
   <xs:complexType name="AmericanLeagueTeamType">
      <xs:sequence>
         <xs:element ref="AmericanLeagueRoster"/>
      </xs:sequence>
      <xs:attributeGroup ref="TeamAttributes" />
   </xs:complexType>
   <xs:complexType name="NationalLeagueTeamType">
      <xs:sequence>
         <xs:element ref="NationalLeagueRoster"/>
      </xs:sequence>
      <xs:attributeGroup ref="TeamAttributes" />
   </xs:complexType>
   <xs:element name="AmericanLeagueTeam" type="AmericanLeagueTeamType"/>
   <xs:element name="NationalLeagueTeam" type="NationalLeagueTeamType"/>
   <!-- Attribute group -->
   <xs:attributeGroup name="TeamAttributes">
      <xs:attribute name="name" use="required"/>
      <xs:attribute name="manager" use="required"/>
   </xs:attributeGroup>
   <!-- Misc -->
   <!-- Restriction of simple type -->
   <xs:simpleType name="ThrowsType">
      <xs:restriction base="xs:string">
         <xs:enumeration value="left"/>
         <xs:enumeration value="right"/>
      </xs:restriction>
   </xs:simpleType>
   <!-- Restriction of simple type -->
   <xs:simpleType name="PercentageType">
      <xs:restriction base="xs:decimal">
         <xs:pattern value=".[0-9]{3}|1.000"/>
      </xs:restriction>
   </xs:simpleType>
   <!-- Abstract type -->
   <xs:complexType name="PlayerType" abstract="true">
      <xs:sequence>
         <xs:element ref="Name"/>
         <xs:element ref="FieldingPercentage"/>
         <xs:element ref="Throws"/>
      </xs:sequence>
   </xs:complexType>
   <xs:element name="Name" type="xs:string"/>
   <xs:element name="FieldingPercentage" type="PercentageType"/>
   <xs:element name="Throws" type="ThrowsType"/>
</xs:schema>
```

B. *mlb_locals.xsd*

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- Some, but not all, declarations from mlb_primary.xsd are localized -->
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
           xmlns="http://www.xmlconference.org/xmldatabinding/locals"
          targetNamespace="http://www.xmlconference.org/xmldatabinding/locals"

           elementFormDefault="qualified"
           attributeFormDefault="unqualified">
   <xs:element name="MajorLeagueBaseball">
      <!-- Local type declaration -->
      <xs:complexType>
         <xs:sequence>
            <!-- Using choice -->
            <xs:choice>
               <!-- Local element declaration -->
               <xs:element name="Year">
                  <!-- Local type declaration -->
                  <xs:simpleType>
                     <xs:restriction base="xs:integer">
                        <xs:minInclusive value="2005"/>
                     </xs:restriction>
                  </xs:simpleType>
               </xs:element>
```

```
                <!-- Local element declaration -->
                <xs:element name="Designator" type="xs:string"/>
             </xs:choice>
             <!-- Enforcing cardinality -->
             <xs:element ref="AmericanLeagueTeam" minOccurs="14"
                         maxOccurs="14"/>
             <!-- Enforcing cardinality -->
             <xs:element ref="NationalLeagueTeam" minOccurs="16"
                         maxOccurs="16"/>
          </xs:sequence>
          <xs:attribute name="commissioner" type="xs:string" use="optional"/>
       </xs:complexType>
    </xs:element>
    <!-- Pitcher -->
    <xs:element name="AmericanLeaguePitcher" type="PitcherType"/>
    <xs:element name="NationalLeaguePitcher" type="NationalLeaguePitcherType"/>

    <xs:complexType name="PitcherType">
       <xs:complexContent>
          <xs:extension base="PlayerType">
             <xs:sequence>
                <!-- Local element declaration -->
                <xs:element name="ERA">
                   <!-- Local type declaration -->
                   <xs:simpleType>
                      <xs:restriction base="xs:decimal">
                         <xs:fractionDigits value="2"/>
                      </xs:restriction>
                   </xs:simpleType>
                </xs:element>
                <xs:element ref="Innings"/>
                <xs:element ref="Role"/>
             </xs:sequence>
          </xs:extension>
       </xs:complexContent>
    </xs:complexType>
    <xs:complexType name="NationalLeaguePitcherType">
       <xs:complexContent>
          <xs:extension base="PitcherType">
             <xs:sequence>
                <xs:element ref="BattingStatistics"/>
             </xs:sequence>
          </xs:extension>
       </xs:complexContent>
    </xs:complexType>
    <xs:element name="Innings" type="xs:nonNegativeInteger"/>
    <xs:element name="Role">
       <!-- Local type declaration -->
       <xs:simpleType>
          <xs:restriction base="xs:string">
             <xs:enumeration value="starter"/>
             <xs:enumeration value="reliever"/>
          </xs:restriction>
```

```
      </xs:simpleType>
   </xs:element>
   <!-- Batter -->
   <xs:complexType name="BatterType">
      <xs:complexContent>
         <xs:extension base="PlayerType">
            <xs:sequence>
               <xs:element ref="BattingStatistics"/>
            </xs:sequence>
         </xs:extension>
      </xs:complexContent>
   </xs:complexType>
   <xs:complexType name="BattingStatisticsType">
      <xs:sequence>
         <xs:element ref="BattingAverage"/>
         <xs:element ref="HomeRuns"/>
         <xs:element ref="RBI"/>
         <xs:element ref="Runs"/>
         <xs:element ref="Bats"/>
      </xs:sequence>
   </xs:complexType>
   <xs:element name="BattingStatistics" type="BattingStatisticsType"/>
   <xs:element name="BattingAverage" type="PercentageType"/>
   <xs:element name="HomeRuns" type="xs:nonNegativeInteger"/>
   <xs:element name="RBI" type="xs:nonNegativeInteger"/>
   <xs:element name="Runs" type="xs:nonNegativeInteger"/>
   <xs:element name="Bats">
      <!-- Local type declaration -->
      <xs:simpleType>
         <xs:restriction base="xs:string">
            <xs:enumeration value="left"/>
            <xs:enumeration value="right"/>
            <xs:enumeration value="switch"/>
         </xs:restriction>
      </xs:simpleType>
   </xs:element>
   <!-- Roster -->
   <xs:element name="Catcher" type="BatterType"/>
   <xs:element name="FirstBaseman" type="BatterType"/>
   <xs:element name="SecondBaseman" type="BatterType"/>
   <xs:element name="ThirdBaseman" type="BatterType"/>
   <xs:element name="Shortstop" type="BatterType"/>
   <xs:element name="LeftFielder" type="BatterType"/>
   <xs:element name="CenterFielder" type="BatterType"/>
   <xs:element name="RightFielder" type="BatterType"/>
   <xs:element name="DesignatedHitter" type="BatterType"/>
   <xs:element name="BenchPlayer" type="BatterType"/>
   <xs:complexType name="BaseRosterType" abstract="true">
      <xs:sequence>
         <xs:element ref="Catcher"/>
         <xs:element ref="FirstBaseman"/>
         <xs:element ref="SecondBaseman"/>
         <xs:element ref="ThirdBaseman"/>
         <xs:element ref="Shortstop"/>
         <xs:element ref="LeftFielder"/>
         <xs:element ref="CenterFielder"/>
         <xs:element ref="RightFielder"/>
      </xs:sequence>
   </xs:complexType>
   <!-- Teams -->
   <xs:complexType name="NationalLeagueTeamType">
      <xs:sequence>
         <!-- Local element declaration -->
         <xs:element name="NationalLeagueRoster">
            <!-- Local type declaration -->
            <xs:complexType>
               <xs:complexContent>
                  <xs:extension base="BaseRosterType">
                     <xs:sequence>
                        <xs:element ref="BenchPlayer" minOccurs="8"
                                    maxOccurs="8"/>
                       <xs:element ref="NationalLeaguePitcher" minOccurs="10"

                                    maxOccurs="10"/>
                     </xs:sequence>
                  </xs:extension>
               </xs:complexContent>
            </xs:complexType>
         </xs:element>
      </xs:sequence>
      <!-- Local attribute declarations -->
      <xs:attribute name="name" type="xs:string" use="required"/>
      <xs:attribute name="manager" type="xs:string" use="required"/>
   </xs:complexType>
   <xs:element name="AmericanLeagueTeam">
      <!-- Local type declaration -->
      <xs:complexType>
         <xs:sequence>
            <!-- Local element declaration -->
            <xs:element name="AmericanLeagueRoster">
               <!-- Local type declaration -->
               <xs:complexType>
                  <xs:complexContent>
                     <xs:extension base="BaseRosterType">
                        <xs:sequence>
                           <xs:choice>
                              <xs:sequence>
                                 <xs:element ref="DesignatedHitter"/>
                                 <xs:element ref="BenchPlayer" minOccurs="7"
                                             maxOccurs="7"/>
                              </xs:sequence>
                              <xs:element ref="BenchPlayer" minOccurs="8"
                                          maxOccurs="8"/>
                           </xs:choice>
                           <xs:element ref="AmericanLeaguePitcher"
                                       minOccurs="10"
```

```
                                        maxOccurs="10"/>
                         </xs:sequence>
                      </xs:extension>
                   </xs:complexContent>
                </xs:complexType>
             </xs:element>
          </xs:sequence>
          <!-- Local attribute declarations -->
          <xs:attribute name="name" type="xs:string" use="required"/>
          <xs:attribute name="manager" type="xs:string" use="required"/>
       </xs:complexType>
    </xs:element>
    <xs:element name="NationalLeagueTeam" type="NationalLeagueTeamType"/>
    <!-- Misc -->
    <xs:simpleType name="ThrowsType">
       <xs:restriction base="xs:string">
          <xs:enumeration value="left"/>
          <xs:enumeration value="right"/>
       </xs:restriction>
    </xs:simpleType>
    <xs:simpleType name="PercentageType">
       <!-- Using union -->
       <xs:union>
          <!-- Local type declaration -->
          <xs:simpleType>
             <xs:restriction base="xs:decimal">
                <xs:pattern value=".[0-9]{3}"/>
             </xs:restriction>
          </xs:simpleType>
          <!-- Local type declaration -->
          <xs:simpleType>
             <xs:restriction base="xs:decimal">
                <xs:pattern value="1.000"/>
             </xs:restriction>
          </xs:simpleType>
       </xs:union>
    </xs:simpleType>
    <xs:complexType name="PlayerType" abstract="true">
       <xs:sequence>
          <xs:element ref="Name"/>
          <xs:element ref="FieldingPercentage"/>
          <xs:element ref="Throws"/>
       </xs:sequence>
    </xs:complexType>
    <xs:element name="Name" type="xs:string"/>
    <xs:element name="FieldingPercentage" type="PercentageType"/>
    <xs:element name="Throws" type="ThrowsType"/>
</xs:schema>
```

C. JAXB Code Sample

```
import com.sun.tools.xjc.Driver;
import org.xmlconference.xmldatabinding.jaxb.primary.AmericanLeagueRosterType;
import org.xmlconference.xmldatabinding.jaxb.primary.AmericanLeagueTeam;
import org.xmlconference.xmldatabinding.jaxb.primary.BatterType;
import org.xmlconference.xmldatabinding.jaxb.primary.BattingStatistics;
import org.xmlconference.xmldatabinding.jaxb.primary.BattingStatisticsType;
import org.xmlconference.xmldatabinding.jaxb.primary.LeftFielder;
import org.xmlconference.xmldatabinding.jaxb.primary.MajorLeagueBaseball;
import org.xmlconference.xmldatabinding.jaxb.primary.ObjectFactory;

import javax.xml.bind.JAXBContext;
import javax.xml.bind.JAXBException;
import javax.xml.bind.Marshaller;
import javax.xml.bind.Unmarshaller;
import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.io.FileOutputStream;
import java.math.BigDecimal;
import java.math.BigInteger;
import java.util.List;

public final class JAXBSample extends Object {
   private static final String[] args_primary = {
           "C:\\xml2005\\mlb_primary.xsd",
           "-d",
           "C:\\xml2005\\code\\src"
   };

   public JAXBSample() {
      JAXBContext context = null;
      Unmarshaller unmarshaller = null;
      Marshaller marshaller = null;
      MajorLeagueBaseball mlb = null;
      List teams = null;
      AmericanLeagueTeam alt = null;
      FileInputStream fis = null;
      FileOutputStream fos = null;
      AmericanLeagueRosterType roster = null;
      BatterType firstBaseman = null;
      BattingStatisticsType stats = null;
      ObjectFactory factory = new ObjectFactory();

      try {
         Driver.main(args_primary);
         context =

JAXBContext.newInstance("org.xmlconference.xmldatabinding.jaxb.primary");
         unmarshaller = context.createUnmarshaller();
         fis = new FileInputStream("C:\\xml2005\\mlb_primary.xml");
         mlb = (MajorLeagueBaseball) unmarshaller.unmarshal(fis);
      }
      catch (JAXBException e) {
         e.printStackTrace();
      }
      catch (FileNotFoundException e) {
         e.printStackTrace();
      }
      catch (Exception e) {
         e.printStackTrace();
      }

      System.out.println("The commissioner of Major League Baseball is " +
                         mlb.getCommissioner());

      teams = mlb.getAmericanLeagueTeam();

      for (Object obj : teams) {
         alt = (AmericanLeagueTeam) obj;

         if (alt.getName().equals("Baltimore Orioles")) {
            roster = alt.getAmericanLeagueRoster();
            firstBaseman = roster.getFirstBaseman();
            stats = firstBaseman.getBattingStatistics();

            System.out.println("The Orioles manager is " + alt.getManager());

            System.out.println("The Orioles designated hitter is " +
                               roster.getDesignatedHitter().getName());
            System.out.println("The Orioles first baseman throws " +
                               firstBaseman.getThrows());
            System.out.println("He is batting (float) " +
                               stats.getBattingAverage().floatValue());
            System.out.println("He is batting (plain String) " +
                               stats.getBattingAverage().toPlainString());
            System.out.println("He has hit this many home runs: " +
                               stats.getHomeRuns().intValue());

            stats.setBattingAverage(new BigDecimal(.299));
            firstBaseman.setThrows("right");
            roster.setLeftFielder(getNewLeftFielder(factory));
         }
      }

      try {
         fos = new FileOutputStream("C:\\xml2005\\mlb_primary_modified.xml");

         marshaller.setProperty(Marshaller.JAXB_FORMATTED_OUTPUT,
                                Boolean.TRUE );
         marshaller.setProperty(Marshaller.JAXB_SCHEMA_LOCATION,
                                "http://www.xmlconference.org/" +
                                "xmldatabinding/primary mlb_primary.xsd");
         marshaller.marshal(mlb, fos);
      }
      catch (FileNotFoundException e) {
         e.printStackTrace();
      }
      catch (JAXBException e) {
         e.printStackTrace();
      }


   }

   private LeftFielder getNewLeftFielder(ObjectFactory factory) {
      LeftFielder lf = null;

      try {
         lf = factory.createLeftFielder();
         lf.setName("David Newhan");
         lf.setThrows("right");
         lf.setFieldingPercentage(new BigDecimal(.991));
         lf.setBattingStatistics(getNewStats(factory));
      }
      catch (JAXBException e) {
         e.printStackTrace();
      }

      return lf;
   }

   private BattingStatistics getNewStats(ObjectFactory factory) {
      BattingStatistics stats = null;

      try {
         stats = factory.createBattingStatistics();
         stats.setBats("lefty");
         stats.setBattingAverage(new BigDecimal(".300"));
         stats.setHomeRuns(new BigInteger("24"));
         stats.setRBI(new BigInteger("80"));

      }
      catch (JAXBException e) {
         e.printStackTrace();
      }

      return stats;
   }
}
```

D. Castor Code Sample

```
import org.exolab.castor.builder.SourceGenerator;
import org.exolab.castor.xml.MarshalException;
import org.exolab.castor.xml.ValidationException;
import org.xmlconference.xmldatabinding.castor.primary.AmericanLeagueTeam;
import org.xmlconference.xmldatabinding.castor.primary.BattingStatistics;
import org.xmlconference.xmldatabinding.castor.primary.LeftFielder;
import org.xmlconference.xmldatabinding.castor.primary.MajorLeagueBaseball;
import org.xmlconference.xmldatabinding.castor.primary.FirstBaseman;
import org.xmlconference.xmldatabinding.castor.primary.AmericanLeagueRoster;
import
org.xmlconference.xmldatabinding.castor.primary.AmericanLeagueRosterTypeChoice;
import
org.xmlconference.xmldatabinding.castor.primary.AmericanLeagueRosterTypeChoiceSequence;
import org.xmlconference.xmldatabinding.castor.primary.types.BatsType;
import org.xmlconference.xmldatabinding.castor.primary.types.ThrowsType;

import java.io.FileNotFoundException;
import java.io.FileReader;
import java.io.FileWriter;
import java.io.IOException;
import java.math.BigDecimal;

public final class CastorSample extends Object {
   public CastorSample() {
      SourceGenerator sg = new SourceGenerator();
      MajorLeagueBaseball mlb = null;
      AmericanLeagueTeam[] teams = null;
      FileWriter fw = null;
      FileReader reader = null;
      AmericanLeagueRoster roster = null;
      FirstBaseman firstBaseman = null;
      BattingStatistics stats = null;
      AmericanLeagueRosterTypeChoice choice = null;
      AmericanLeagueRosterTypeChoiceSequence choiceSequence = null;

      sg.setEqualsMethod(true);

      try {
         sg.generateSource("C:\\xml2005\\mlb_primary.xsd",
                          "org.xmlconference.xmldatabinding.castor.primary");

      }
      catch (IOException e) {
         e.printStackTrace();
      }

      try {
         reader = new FileReader("C:\\xml2005\\mlb_primary.xml");
```

```
            mlb = (MajorLeagueBaseball) MajorLeagueBaseball.unmarshal(reader);
         }
         catch (MarshalException e) {
            e.printStackTrace();
         }
         catch (ValidationException e) {
            e.printStackTrace();
         }
         catch (FileNotFoundException e) {
            e.printStackTrace();
         }

         System.out.println("The commissioner of Major League Baseball is " +
                            mlb.getCommissioner());

         teams = mlb.getAmericanLeagueTeam();

         for (AmericanLeagueTeam alt : teams) {
            if (alt.getName().equals("Baltimore Orioles")) {
               roster = alt.getAmericanLeagueRoster();
               firstBaseman = roster.getFirstBaseman();
               stats = firstBaseman.getBattingStatistics();
               choice = roster.getAmericanLeagueRosterTypeChoice();
             choiceSequence = choice.getAmericanLeagueRosterTypeChoiceSequence();


               System.out.println("The Orioles manager is " + alt.getManager());

               System.out.println("The Orioles designated hitter is " +
                                choiceSequence.getDesignatedHitter().getName());

               System.out.println("The Orioles first baseman throws " +
                                  firstBaseman.getThrows());
               System.out.println("He is batting (as a float) " +
                                  stats.getBattingAverage().floatValue());
               System.out.println("He is batting (as a plain String) " +
                                  stats.getBattingAverage().toPlainString());
               System.out.println("He has hit this many home runs: " +
                                  stats.getHomeRuns());

               stats.setBattingAverage(new BigDecimal(.299));
               firstBaseman.setThrows(ThrowsType.RIGHT);
              alt.getAmericanLeagueRoster().setLeftFielder(getNewLeftFielder());

            }
         }

         try {
            fw = new FileWriter("C:\\xml2005\\mlb_primary_modified.xml");
            mlb.marshal(fw);
            fw.close();
         }
         catch (IOException e) {
```

```
         e.printStackTrace();
      }
      catch (MarshalException e) {
         e.printStackTrace();
      }
      catch (ValidationException e) {
         e.printStackTrace();
      }
   }

   private LeftFielder getNewLeftFielder() {
      LeftFielder lf = new LeftFielder();

      lf.setName("David Newhan");
      lf.setThrows(ThrowsType.RIGHT);
      lf.setFieldingPercentage(new BigDecimal(.991));
      lf.setBattingStatistics(getNewStats());

      return lf;
   }

   private BattingStatistics getNewStats() {
      BattingStatistics stats = new BattingStatistics();

      stats.setBats(BatsType.LEFT);
      stats.setBattingAverage(new BigDecimal(".300"));
      stats.setHomeRuns(24);
      stats.setRBI(80);

      return stats;
   }

}
```

Bibliography

[The Java Web Services Tutorial] *The Java Web Services Tutorial*, June 14, 2005. Available at http://java.sun.com/webservices/docs/1.6/tutorial/doc/index.html

[The Castor Project] *The Castor Project*. Available at http://www.castor.org/index.html

[Using the Source Code Generator] *Using the Source Code Generator*, Keith Visco and Arnaud Blandin. Available at http://www.castor.org/sourcegen.html

[Castor API] *Castor API*. Available at http://www.castor.org/api/overview-summary.html

[javaworld] *Use XML Data Binding to Do Your Laundry: Explore JAXB and Castor from the Ground Up*, Sam Brodkin, December 28, 2001. Available at http://www.javaworld.com/javaworld/jw-12-2001/jw-1228-jaxb.html [onjava] *XML Data Binding with Castor*, Dion Almaer, October 24, 2001. Available at http://www.onjava.com/pub/a/onjava/2001/10/24/xmldatabind.html

[xmlcom] *Comparing Java Data Binding Tools*, Mette Hedin, September 03, 2003. Available at http://www.xml.com/pub/a/2003/09/03/binding.html

[Martin] *Agile Software Development: Principles, Patterns, and Practices*, Robert C. Martin et al., Prentice Hall, October 15, 2002.

[Martin UML] *UML for Java Programmers*, Robert C. Martin, Prentice Hall, May 27, 2003.

[javaboutique] *Converting XML documents to Java objects with Castor XML*, Keld H. Hansen. Available at http://javaboutique.internet.com/tutorials/CastorXML/index.html

[Kolawa] *Java Code Security Rules*, Adam Kolawa et al. Available at http://www.ftponline.com/resources/spcollections/jcsr/

[JSR 31] *JSR 31: XML Data Binding Specification*. Available at http://www.jcp.org/en/jsr/detail?id=31

[JSR 222] *JSR 222: Java Architecture for XML Binding (JAXB) 2.0*. Available at http://www.jcp.org/en/jsr/detail?id=222

[serverside] *Opinion: What Tool for Xml Binding?*, Joseph Fouad, December 17, 2004. Available at http://www.theserverside.com/news/thread.tss?thread_id=30658

[usingschema] *Using W3C XML Schema*, Eric van der Vlist, October 17, 2001. Available at http://www.xml.com/pub/a/2000/11/29/schemas/part1.html?page=1

[roseindia] *Beginning JAXB (Jav Architecture for XML Binding)*, R.S. Ramaswamy, May 2005. Available at http://www.roseindia.net/jaxb/r/jaxb.shtml

[zdnetasia] *Java-XML Data Binding Offers the Best of Both Worlds*, Harshad Oak, September 18, 2002. Available at http://www.zdnetasia.com/builder/architect/system/0,39045489,39081953,00.htm

[netxml] *.NET and XML*, Niel M. Bornstein, O'Reilly, July 2003.

[J2SE API] *Java 2 Platform Standard Edition 5.0 API Specification*. Available at http://java.sun.com/j2se/1.5.0/docs/api/

[Java Technology Documentation] *Java Technology and Web Services - Documentation*. Available at http://java.sun.com/webservices/docs.html

[Gamma *et al.*] *Design Patterns*, Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides, Addison-Wesley Professional, January 15, 1995.

[C# Reserved Words] *Getting Started with C#*, September 2, 2001.. Available at http://www.informit.com/articles/article.asp?p=23211&seqNum=4&rl=1

[Using XJC with Ant] *Java Architecture for XML Binding: Using XJC with Ant*. Available at http://java.sun.com/webservices/docs/1.6/jaxb/ant.html

Acknowledgements

The author would like to recognize the following individuals who helped to improve the quality of this discussion: Research Fellows Jessica Glace and Jonathan Leete of LMI Government Consulting, James Villani of ASSETT, Inc., Chadwick Adebiyi of EMC Legato, and Geoffrey H. Simpson, J.D. The author would also like to express his heartfelt gratitude to LMI officers Vice-President Dr. Susan Marquis, Executive Director Robert Hutchinson, Program Director Joseph Zurlo, and Program Manager Debra Dennie, whose continuous encouragement and support were invaluable. Finally, the author wishes to express a special note of thanks to Gregory Wilson of LMI Government Consulting, whose knowledge of XML is matched only by his kindness and generosity and without whom this paper would not have been possible.

Biography

Neil **Chaudhuri**
Software Engineer
LMI Government Consulting [http://www.lmi.org]
McLean
Virginia
United States of America

A Sun Certified Java Programmer, Mr. Chaudhuri is currently a Software Engineer with LMI Government Consulting in McLean, VA. His work focuses on object-oriented design principles and best practices, and he leads development teams building applications utilizing such technologies as Java/J2EE, Microsoft .Net, leading database technologies like Oracle, and XML and related technologies like W3C XML Schema, XSLT, ebXML Core Components, and web services. Mr. Chaudhuri's experience has led him to become a champion of agile methodologies for software development. He also teaches introductory XML and JavaScript courses in Virginia for Fairfax County Public Schools Adult and Community Education Program.

Mr. Chaudhuri is the author of *J2EE or .Net: A Managerial Perspective*, published in 2003 by the International Workshop on Evolution of Large-scale Industrial Software Applications, and *Very Large Software Systems: A Service-Oriented Approach*, published by the 2005 World Multi-Conference on Systemics, Cybernetics and Informatics.

Mr. Chaudhuri recently received a Master of Science degree in Information Systems Technology with a concentration in Management from The George Washington University. He also has a Bachelor of Science degree in Computer Science from the University of Maryland as well as a Bachelor of Science degree in Biological Sciences and a Bachelor of Arts degree in Political Science from the University of Pittsburgh. Mr. Chaudhuri is a member of *Phi Beta Kappa* and the the *Zeta* Chapter of *Alpha Iota Mu*, the National Honor Society for Information Systems.