

# The SOA Magazine

## Feature Article



### The Importance of Schema Design in SOA

by Priscilla Walmsley

Published: March 9, 2010 (SOA Magazine Issue XXXVII: March 2010)

[Download PDF](#)

[Digg This](#) • [De.licio.us](#) • [Slashdot](#) • [Technorati](#) • [StumbleUpon](#) • [Google Bookmark](#)

### Introduction

XML Schemas are a fundamental part of any XML-based service oriented architecture. They define the structure and content of the messages that are passed among services. Enterprise architects, DBAs and software developers often devote a lot of time to carefully designing data: they create enterprise data models, data dictionaries with strict naming and documentation standards, and carefully designed and optimized relational databases. Unfortunately, service designers and implementers often do not pay as much attention to good design when it comes to XML messages.

There are several reasons for this. Some people feel that since the XML messages are transitory, it is not important how they are structured. Some decide that it is easier to use whatever schema is generated for them by a toolkit. Others decide to use an industry standard XML vocabulary, but fail to figure out how their data really fits into that standard or come up with a strategy for how to customize it for their needs.

As with any data design, there are many ways to structure XML messages. For example, decisions must be made about how many levels of structural elements to include, whether the elements should represent generic or more specific concepts, how to represent relationships, and how far to break down data into separate elements. In addition, there are multiple ways to express the XML structure in XML Schema. Decisions must be made about whether to use global vs. local declarations, whether to use named vs. anonymous types, whether to achieve reuse through type extension or through named model groups, and how schemas should be broken down into separate schema documents.

The choices you make when designing a schema can have a significant impact on the ease of implementation, ease of maintenance, and even the ongoing relevance of the service itself. Failure to take into account design goals such as reuse, graceful versioning, flexibility and tool support can have serious financial impacts on service implementation projects.

This article explains the role of schemas in a Web services environment, describes the qualities of a well-designed schema, and provides some pointers for achieving that good design. It also provides guidelines a creating a Naming and Design Rules (NDR) document that forces the consideration of design issues at an enterprise level before service implementation begins.

### Uses for Schemas

When designing an XML Schema, it is first important to understand what it will be used for. Schemas actually play several roles:

- *Validation.* Validation is the purpose that is most often associated with schemas. Given an XML message, you can use a schema to automatically determine whether that message is valid or invalid. Are all of the required elements there, in the right order? Do they contain valid values according to their data types? Schema validation does a good job of checking the basic structure and content of

elements.

- *A service contract.* A schema serves as part of the understanding between two parties. The service provider and the service consumer can both use the schema as a machine-enforceable set of rules describing the interface.
- *Documentation.* Schemas are used to document the service contract for the developers and end users that will be implementing or using the service. Narrative human-readable annotations can be added to schema components to further document them. Although XML Schemas themselves are not particularly human-readable, they can be viewed by less technical users in a graphical XML editor tool. In addition, there are a number of tools that will generate HTML documentation from schemas, making them more easily understood.
- *Code generation.* Schemas are also commonly used by Web services software products to generate classes and interfaces that are used to read and write the XML message payloads. When a service contract is designed first, classes can be generated automatically from the schema definitions, ensuring that they match. The schema provides information not only about what elements (objects) will appear in XML messages, but also about their data types.

As you can see, schemas are an important part of a service-oriented environment. Although it is certainly possible to implement Web services without schemas, valuable functionality would be lost. You would be forced to use a non-standard method to validate your messages, document your interface, and generate your service classes. You also would not be able to take advantage of the many schema-based tools that implement this functionality at low cost.

The various roles of XML schemas should be taken into account when designing them. For example, use of obscure schema features can make code generation difficult, or not adequately documenting schemas can impact the usefulness of generated documentation.

## XML Schema Design Goals

Designing schemas well is a matter of paying attention to certain important design considerations: flexibility and extensibility, reusability, clarity and simplicity, support for versioning, interoperability and tool support [REF-2]. The rest of this article takes a closer look at each of these design goals.

### Flexibility and Extensibility

Schema design often requires a balancing act between flexibility on the one hand, versus rigidity on the other. For example, suppose I am selling digital cameras that have a variety of features, such as resolution, battery type, and screen size. Each camera model has a different set of features, and the types of features change over time as new technology is developed. When designing a message that incorporates these camera descriptions, I want enough flexibility to handle variations in feature types, without having to redesign my message every time a new feature comes along. On the other hand, I want to be able to accurately and precisely specify these features.

To allow for total flexibility in the camera features I could declare a features element whose type contains an `xsd:any` wildcard, which means that any well-formed XML is allowed. This would have the advantage of being extremely versatile and adaptable to change. The disadvantage is that the message structure is very poorly defined. A developer trying to write an application to process the message would have no idea what features to expect, and what format they might have.

On the other hand, I can declare highly constrained elements for each feature, with no opportunity for variation. This has the benefit of making the features well defined, easy to validate and much more predictable. Validation is more effective because certain features can be required, and their values can be constrained by specific data types. However, the schema is brittle because it must be changed every time a new feature is introduced. When the schema changes, the applications that process the documents must also often change.

The ideal design is usually somewhere in the middle. A balanced approach in the case of the camera

features might be to create a repeating feature element that contains the name of the feature as an attribute, and the value of the feature as its content. This eliminates the brittleness while still providing a predictable structure for implementers.

## Reusability

Reuse is an important goal in the design of any software. Service contracts that reuse XML components across multiple messages are easier for developers and users to learn, are more consistent, and save development and maintenance time that would be spent writing redundant software components.

Reuse can be achieved in XML design in a number of ways in XML Schema:

- *Reusing types.* It is highly desirable to reuse complex and simple types in multiple element and attribute declarations. For example, you can define a complex type named `AddressType` that represents a mailing address, and then use it for both `BillingAddress` and `ShippingAddress` elements. Only named, global types can be reused; so types in XML Schema, particularly in a service-oriented environment, should always be named.
- *Type inheritance.* XML Schema complex types can be specialized from other types using the `xsd:extension` element. For example, I can create a more generic type `ProductType` and derive types named `CameraType` and `LensType` from it. This is a form of reuse because `CameraType` and `LensType` inherit a shared set of properties from `ProductType`.
- *Named model groups and attribute groups.* Through the use of named model groups (the `xsd:group` element), it is possible to define reusable pieces of content models. This is useful alternative to type inheritance, for types that are semantically different but just happen to share some properties with other types.
- *Reusing schema documents.* Entire schema documents can be reused by taking advantage of the `xsd:include` and `xsd:import` mechanisms of XML Schema. This is useful for defining components that might be used in several different contexts or services. In order to plan for reuse, schema documents should be broken down into logical components by subject area. Having schema documents that are too large and all-encompassing tends to inhibit reuse because it forces other schema documents to take all or nothing when importing them. It is also good practice to create a "core components" schema that has low-level building blocks, such as types for `Address` and `Quantity`, that are imported by all other schema documents.

## Clarity and Simplicity

Although most XML documents in a Web services environment are both written and read by software applications, they still should be designed in a way that they are easy to conceptualize and process. Implementers on both sides of the service contract writing and maintaining applications to process these messages, and they must understand them. Overly complex message designs lead to overly complex applications that create and process them, and both are hard to learn and maintain.

### *Naming and Documentation*

Properly and consistently naming schema components (elements, attributes, types, groups) can go a long way toward making the documents comprehensible. Using a common set of terms rather than multiple synonymous terms is good practice, as is the avoidance of obscure acronyms. In XML Schema, it is helpful to identify the kind of component in its name, for example using the word "Type" at the end of type names. Namespaces should also be consistently and meaningfully named.

Of course, good documentation is very important to achieving clarity. XML Schema allows components to be documented using `xsd:annotation` elements. While you probably have other documentation that describes your service contract, having human-readable definitions of the components in your schema is very useful to people who maintain that schema. It also allows you to use tools that automatically generate schema documentation more effectively.

### *Clarity of Structure*

Consistent structure can also help improve clarity. For example, if many different types have child elements Identifier and Name, put them first and always in the same order. Reuse of components helps to ensure consistent structure.

It is often difficult to determine how many levels of elements to put in a message. Having structural elements that group together related properties can help with understanding. For example, embedding all address-related elements (street, city, etc.) inside an Address child element rather than directly inside a Customer element is an obvious choice. It makes the components of the address clearly contained and allows you to make the entire address optional or repeating.

It is also often useful to use structural elements to contain lists of like elements. For example, it is a good idea to embed a repeating sequence of OrderedItem elements inside an OrderedItems (plural) container rather than directly inside a PurchaseOrder element. These container elements can make messages easier to process and often work better with code generation tools.

However, there is such a thing as excessive use of structural elements. XML messages that are a dozen levels deep can become unwieldy and difficult to process.

### **Support for Graceful Versioning**

Service contracts will change over time. Schemas should be designed in accordance with a plan for how to handle changes in a way that causes minimum impact on service consumers.

A typical service versioning strategy differentiates between major versions and minor versions. Major versions, e.g. 1.0, 2.0 and 3.0, are by definition disruptive and not backward compatible; at times this is an unavoidable part of software evolution. On the other hand, minor versions, e.g. 1.1, 1.2 and 1.3, are backward compatible. They involve changes to schemas that will still allow old message instances to be valid according to the new schema. For example, a version 1.2 message can be valid according to a version 1.3 schema if the version 1.3 limits itself to backward compatible changes, such as:

- adding optional elements
- loosening occurrence restrictions, for example making a required element optional
- making types less restrictive, for example adding enumerations to a code list

Some service designers take their versioning strategy a step further; they make schemas forward compatible. That is, a version 1.3 instance is valid according to the version 1.2 schema. This requires some careful planning when developing the version 1.2 schema. An area needs to be set aside for the (optional) elements added in version 1.3. This area needs to be allowed to contain any content in the version 1.2 schema, but be more specifically defined (to add new element declarations) in the version 1.3 schema.

Namespace names often play a role in versioning because changing a namespace is necessarily backward incompatible. Often, namespace names reflect the major version of an XML schema but not the minor version.

A complete strategy for versioning is outside the scope of this article, but is covered more fully in the book *Web Service Contract Design and Versioning for SOA* [REF-1].

### **Interoperability and Tool Compatibility**

Schemas are used heavily by tools, not just for validation but also for the generation of code and documentation. In an ideal world, all Web services toolkits would support the exact same schema language, and all schemas would be interoperable. The unfortunate reality is that tools, especially code generation tools, vary in their support for XML Schema, for several reasons:

- Some toolkits incorrectly implement features of XML Schema because the recommendation is highly

complex and in some cases even ambiguous.

- Some Web services toolkits deliberately do not support certain features of XML Schema 1.0 because they do not find them to be relevant or useful.
- Some XML Schema concepts do not map cleanly onto object-oriented concepts. Even if a toolkit attempts to support these features, it may do so in a less than useful way.

In general, it is best to stick to a subset of the XML Schema language that is well supported by major toolkits. Features of XML Schema to avoid in a Web services environment include:

- mixed content (elements that allow text content as well as children)
- xsd:choice and xsd:all model groups
- complex content models with nested model groups
- substitution groups
- dynamic type substitution using the xsi:type attribute
- default and fixed values for elements or attributes
- redefinition of schema documents using xsd:redefine

It is advisable to test your schemas against a variety of toolkits to be sure that they can handle them gracefully.

### **Developing a Strategy, or NDR**

Many organizations that are implementing medium- to large-scale service-oriented architectures develop enterprise-wide guidelines for schema design, taking into account the considerations described in this article. Sometimes these documents are referred to as Naming and Design Rules (NDR) documents.

Using an NDR has a number of benefits:

- It promotes a standard approach to schema development that improves consistency and therefore clarity.
- It ensures that certain strategies, such as how to approach versioning, are well thought out before too much investment in development has been made.
- It allows the proposed approach to be checked with toolkits in use in the organization to see if they generate manageable code.
- It serves as a basis for design reviews, which are a useful way for centralized data architects to guide or even enforce design standards within an organization.

A schema design strategy should include the following components:

- *Naming standards*: what characters to use as separators between words, upper vs. lower case names, special considerations for naming types and groups
- *Documentation standards*: the types of documentation required for schema components, where they are to be documented
- *Namespaces*: what they should be named, how many to have, how many schema documents to use per namespace
- *XML Schema design*: a list of allowed (or prohibited) XML Schema features, limited to promote simplicity and interoperability

- *Versioning strategy*: whether to require forward compatibility (and if so how to accomplish it), list of allowed schema changes for backward-compatible minor versions
- *Schema reuse strategy*: how many schema documents to have (one per subject area? one per business division?), recommended folder structure, and an approach for core components
- *Incorporation of external standards*: which external standards are approved for use, description of the correct way to incorporate or extend them
- *Incorporation of enterprise standards*: for organizations that have an enterprise data model and/or enterprise architecture, how the schemas map to that model

## Conclusion

XML schemas are an important part a service-oriented environment in that they describe the content of XML messages, are used to automatically ensure that messages are correct, and are often used to generate application code. As such, they should be carefully designed and developed, taking into account a number of factors such as clarity, extensibility, versioning and tool support. In an ideal environment, a strategy in the form of an NDR is developed to address these concerns before a large-scale investment in service development.

## References

- [REF-1] "Web Service Contract Design and Versioning for SOA". Erl, Karmarkar, Walmsley, et. al., Prentice Hall, 2009.
- [REF-2] "Definitive XML Schema", Priscilla Walmsley, Prentice Hall, 2001.

THE PRENTICE HALL SERVICE-ORIENTED COMPUTING SERIES FROM THOMAS ERL

