Creating and Maintaining Large Families of Related Schemas

Anthony B. **Coates**

Abstract

Many XML and schema tools work sufficiently well when your task is to create a single XML schema. However, how do you create and manage large families of XML schemata that share definitions of simple types and/or complex structures? What are the advantages and disadvantages of sharing definitions? What technical problems arise in trying to share definitions? How does sharing of definitions impact the documentation of those definitions? This presentation discusses the real-life development of a set of a family of >450 inter-related W3C XML Schemas. Issues will be presented, and the advantages and disadvantages of the solution that has been implemented will be discussed.

Table of Contents

1. Introduction

When working on large-scale "enterprise" software projects, one of the key architectural concerns is whether the selected technologies will scale up to the size of the problem. In this paper, the practicalities and scalability of W3C XML Schema are discussed based on the real-life creation and maintenance of a large family (>450) of inter-related Schemas.

Many of the standards that are used in the Internet/Web/XML arena are developed on an essentially **volunteer** basis. While the people involved in these standards are generally very capable, there is rarely the time available to put together the kind of testing needed to fully validate the scalability of the standards they produce (I know this myself from first-hand experience). This *caveat* applies to W3C XML Schema, and some implications of that will be discussed.

2. The practical problem

I was brought in to work on the XML aspects of a major financial system (which cannot be named here) in the architectural (pre-development) phase. What had already been decided was that communication between components of the system would be done using XML messages. This meant that the number of XML messages would be significantly greater than for a system where XML was only used for the external interfaces.

Before the detailed interface design was started, I was asked to estimate how many messages would be required. Now, a common answer to such a question would be something like "you can't say until you have more of the details". However, when you are hired to provide **expert services**, what is usually wanted is that you can provide sufficiently useful answers to the majority of questions in spite of a lack of known detail.

In this case, I fell back on my training as a physicist. In my first year of university physics, in my first tutorial, the lecturer posed the question *"how long is a piece of string?"* This was a serious question, he wanted an answer. None of the students were able to come up with an answer, so the lecturer unveiled his answer: **the length of a piece of string is between 1 millimetre and 1 kilometre** (between 1/25 inch and 3/5 mile in imperial units). The reasoning was as follows: 1 millimetre is **too short** for something to be consider a *piece* of string, and 1 kilometre is **too long**. The point he was making was that even where you cannot give a precise answer, you can still often put **upper and lower limits** on what the answer can be. Sometimes, just having those upper and lower limits can be enough to provide a useful partial answer to a problem.

With 10–20 components in the system, there were going to be at least 10 messages in the system. My experience is that people can deal well with tens or hundreds of items, but that thousands can become problematic without special organisation. On that basis, I didn't see there being more than 1000 messages in the system, as I expected that smaller messages would be aggregated if necessary to keep the number down. So my answer was that the system would have 10–1000 messages.

Now, some people thought that was a curiously large range for an estimate, but people often take the (incorrect) view that an estimate is a single number to which you add a standard percentage error like 25%. In my experience, people do this kind of estimation badly, estimating a single number and then an error percentage. I find that you get far better results if you ask people to estimate the upper and lower limits separately. The range you get can be quite large in percentage terms, but it is better to be realistic about having a large range. The practical upshot of this was that my planning was always on the basis that **the solution had to work for tens of messages, but scale smoothly to hundreds or even a thousand messages**.

3. Solution issue #1: sharing

In a project involving a large number of Schemas, you need to have a policy towards how you share element and type definitions across those Schemas. An extreme approach, **albeit a very popular one**, is to have a single Schema that is the superset of all of the messages that you need. This is such a popular approach that many of its users don't think

of it as being a **sharing strategy**. However, if you use the same Schema to satisfy all of your message use cases, then you couple all of your use cases together by forcing them to share the same Schema. This means that changes to the Schema for one use case can adversely impact other use cases. In particular, a change to the Schema for one use case will force you to **re-test all use cases**, because all use cases are at risk when the Schema changes.

Part of the issue here is that modern IT methodologies revolve around sharing. Sharing of function libraries and sharing of object-oriented classes are encouraged, and software designs are typically *normalised* to maximise the sharing of code. This is done for two key reasons:

- To minimise the amount of code that needs to be written, by avoiding any duplication of functionality;
- To centralise functionality so that when a change needs to be made, it is isolated to the fewest possible locations in the code.

Make no mistake, these are worthy reasons for code sharing. However, what is often overlooked is that the more you share code, the more brittle and less robust your code becomes. Now, some people will immediately disagree with this assertion, and argue that by centralising functionality, so that a change needs only be applied in one place in the code, you improve robustness in the face of change. Empirically, I have never found this to work as the theory would suggest. Indeed, I'm sure everyone has seen shared libraries/classes that have become onerously complex to use as they try to provide a single solution that works for all use cases of all users. Equally, I'm sure most people have seen how a problem in a shared piece of code can cause an application to break everywhere, when what was intended was a change or fix just for a single aspect of the application.

When you are looking for a robustness strategy, nature provides a good example. DNA works pretty well as a way of encoding the structure of life, but it isn't centralised, nature does a *copy-and-paste* from one organism to the next. This means that if an arbitrary animal suddenly evolves an extra leg, humans don't all suddenly grow an extra leg in sympathy. While our DNA will be largely similar to that animal's, we have our own copy, we aren't directly sharing the same instance of DNA with that animal.

So, in deciding on a sharing policy, I sought a workable **balance** between the robustness of having completely independent messages for every use case, with no sharing (beyond an initial copy-and-paste), and the efficiency of centralising everything (a single Schema solution counts as complete centralisation).

To this, I also had to add my own bad experiences with trying to implement element and type sharing in W3C XML Schemas using only the import and include mechanisms that W3C XML Schema (WXS) provides. Now, I don't mean this as a criticism of the WXS specification *per se*. The Schema Working Group was tasked to write a specification for a control language for a type a standardised XML validator, and nothing more. The import/include facilities that they provided mirror the kind of facilities that programming languages commonly provide, and were really the only kind of sharing mechanism that you could put in such a specification within the scope of the work.

What is the problem with the import/include as implemented? The problem is that it is hard to import/include the elements and types that you need into a new Schema in a simple way. You can't just import/include a single element or type and then expect the WXS validator to automatically add all of the elements and types on which your chosen one depends. Instead, you have to make sure that you import all or include all of the dependencies yourself. This kind of thing is very easy to get wrong, and leads to a lot of lost developer time. It causes development groups to (attempt to) stop using common enterprise WXS definitions and just write their own private versions.

There are also problems when the import/include paths stretch across multiple WXS files. For example, suppose Schema *A* depends on `typeB` in Schema *B*, and `typeB` depends on `typeC` in Schema *C*. So Schema *A* imports/includes Schema *B*, and Schema *B* imports/includes Schema *C*. That will work OK. Now suppose you want to use `typeC` in Schema *A* directly, so you import/include Schema *C* into Schema *A*. The WXS validator now sees the definition of `typeC` twice; once from the Schema *A* importing/including Schema *C*, and once from Schema *A* importing/including Schema *B* which imports/includes Schema *C*.

A WXS validator will throw an error if the same type is defined in two different places. Now, here `typeC` is only defined in one place, but my experience is that when you have import/includes with their own imports/includes, etc., the validator may not be able to resolve the relative paths completely, so that the validator **isn't sure whether it is seeing the same definition twice, or two separate definitions of `typeC`**. Not being sure, the validator throws an error. I spent a lot of time on an earlier project writing a series of XSLT scripts to process imports/includes in Schemas and resolve relative paths appropriately to work around this problem. I definitely didn't want a repeat on a much larger project where the problems could only be worse.

What I needed was a solution that went beyond what the WXS specification gives you. What I needed was a **element/type repository** that would allow me to select different elements and types for different messages and do two important things:

- Automatically locate and import/include all dependencies of any element or type that I selected for my Schema;
- Generate separate Schemas for each message without using WXS import/include mechanisms.

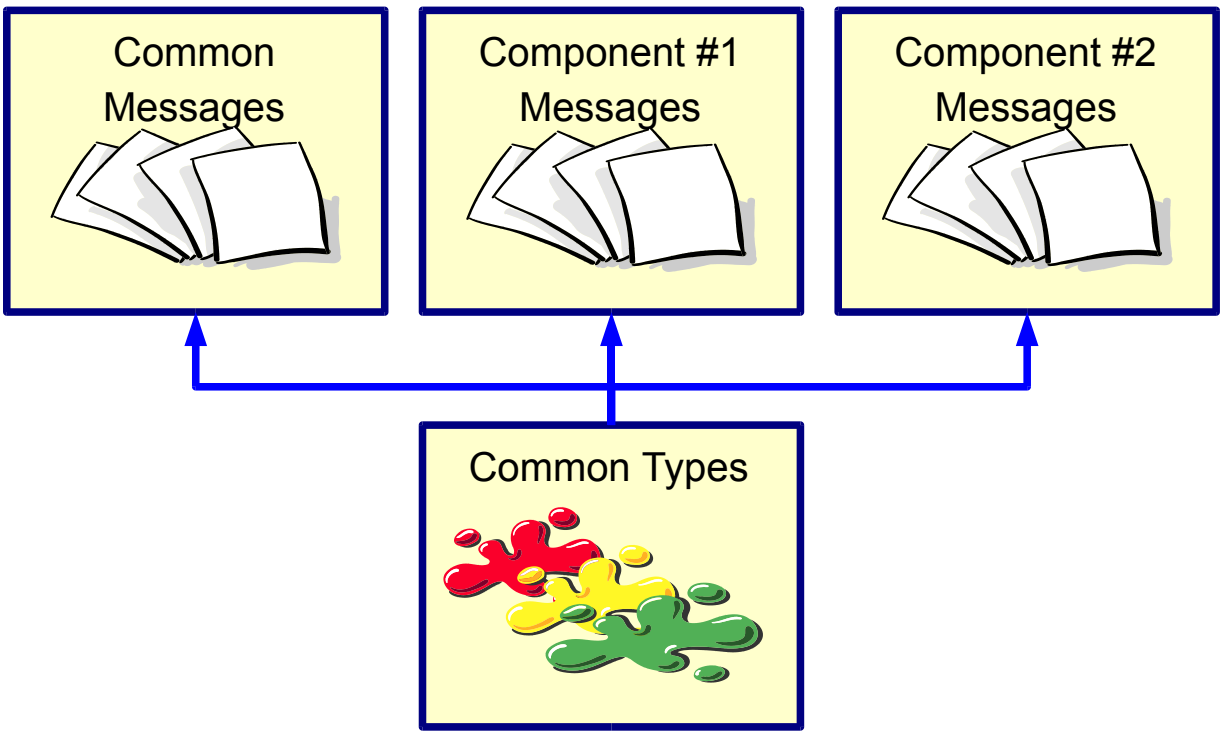
So the sharing in my solution is done at the repository level, and not at the Schema level. This turns out to be a much easier way to manage large sets of re-usable definitions. It is a pity that this kind of repository was out of scope of the work of the Schema WG. It would still be useful as a future work item. Lacking a standard for such a repository, I simply went out and found a company with the right kind of repository software. At that stage the software didn't handle WXS (it was dealing with ASN.1 messages), but the company was keen to move into XML, so I helped them map their repository approach onto WXS; I like to take an active approach to vendor management.

There are different approaches a tool can take to how to store the definitions for a repository. You can use a database, or you can store the definitions in files in the filesystem. The solution I chose used the latter; it had its own *model files* that could include other model files, although without the import/include issues that bedevil WXS. The software also did something else I needed for the project; it generated Java classes to represent the message contents, and Java classes to do the transformation to/from XML. This is the kind of added value you can get from a repository, especially one that abstracts the WXS specifics out of the message model design, as this tool did. **I can't recommend this repository approach to you too highly; it really works in practice**.

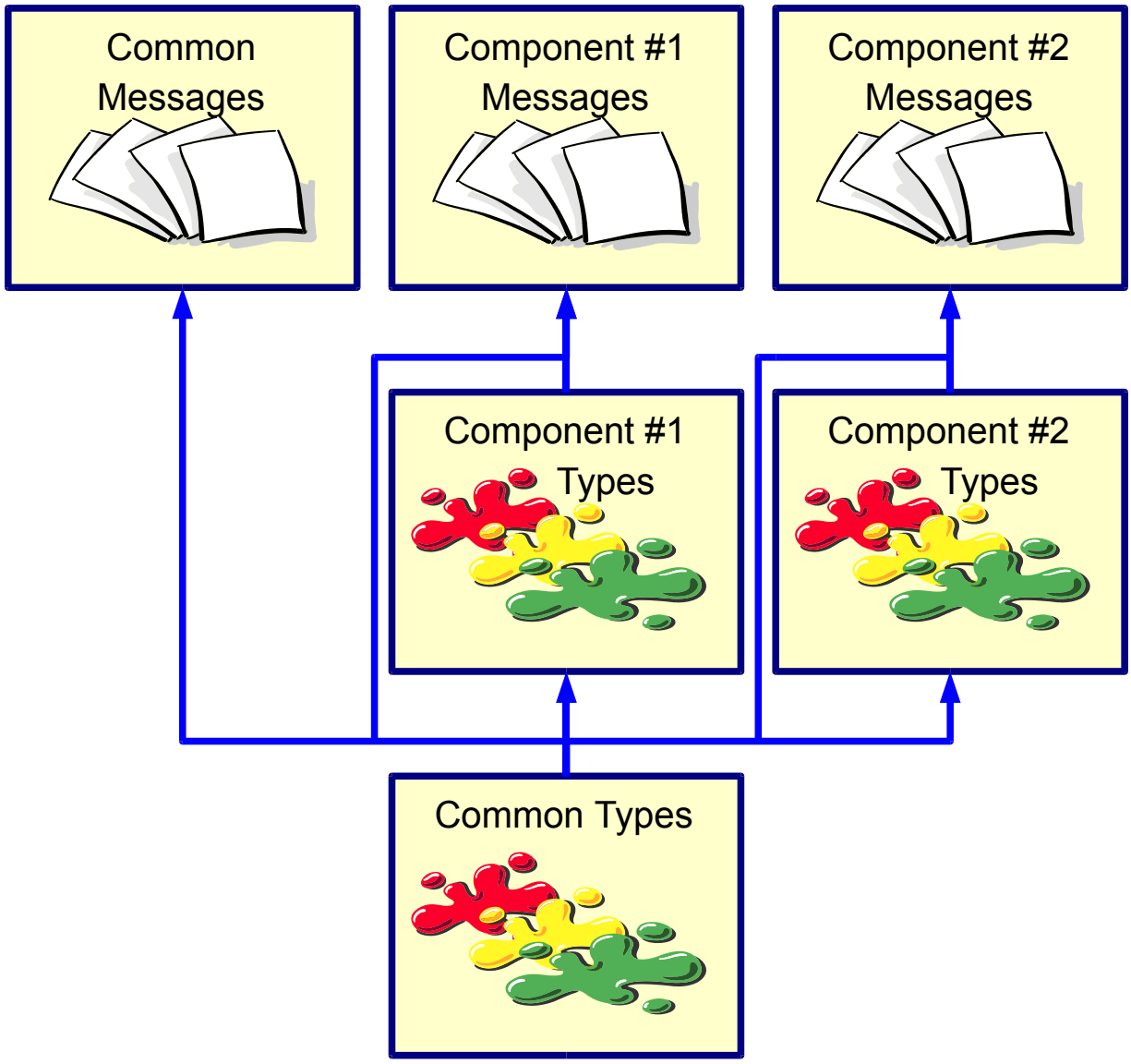
Having chosen to use a repository that could produce independent Schemas based on shared definitions in the repository, the basic sharing approach was as follows:

- Each use case would have a separately named message. There was to be no sharing of messages across use cases unless there was a strong argument for doing so. This meant that, as necessary, it would be straightforward to convert two messages with shared definitions into two messages with separate definitions, if the definitions needed to be allowed to diverge as the messages changed.
- Definitions would be shared where they were identical across messages, but would be converted into separate message-specific definitions rather than generalised into forms that required optional parts in order satisfy the different need of different messages. Suppose Schema *A* and Schema *B* share `typeA`, which contains `<elementA>` and `<elementB>`. Now suppose you need to append `<elementC>` for Schema *A*, and `<elementD>` for Schema *B*. You could continue to share `typeA` by appending a choice of `<elementC>` or `<elementD>` to the shared `typeA`, but that would just weaken the validation of both Schemas. Instead, it is better to give Schema A and Schema B their own separate definitions of `typeA`, based on the original shared `typeA`.

This approach led to the following structure for the file-based repository:



Although this was workable, what was found, especially during heavy periods of message development, was that this led to too much contention for the file containing the common types. Also, many of the common types were found to be re-used in multiple messages, but only in messages relating to a single component. On this basis, the repository structure was changed to:



This **reduced the contention issues** sufficiently. Had there been more contention, we would then have looked at splitting up the main set of common types into several repository files, being careful to **avoid circular dependencies** between the definitions in those files.

4. Solution issue #2: message structure

W3C XML Schema (WXS) supports a variety of different approaches to creating Schemas. For any particular project, you need to settle on just one approach, to maintain consistency. For example, for projects where the XML more-or-less represents a human readable document (e.g. DocBook), people tend to use a rule-of-thumb that element content should be used only for the human readable content, and attribute content should be used for any technical content (identifiers, codes, etc.) that is not intended for display for humans.

For the project described in this paper, the XML was completely data-oriented, and not intended to contain human readable information. On that basis, I followed a common rule for such XML (certainly common in financial applications), i.e. that element content was preferred over attribute content, and attributes would only be used in special circumstances. The main reasons for taking this approach are:

- Element content is easily extended by adding child elements, whereas attributes are restricted to **simple** (textual) content only;
- Where there isn't a **natural rule** for deciding when to use element content versus attribute content, you need to impose a rule which dictates the choice, or a lot of design and meeting time will be wasted debating elements versus attributes, for no significant advantage to the project;
- If you choose to use element content or attribute content arbitrarily, users will be left wondering whether the choice of element content versus attribute content **carries any important meaning or not**, and that just becomes confusing for your users.

So in general, element content was used exclusively. However, some exceptions to this were allowed, again following the lead of some financial XML specifications:

- Attributes could be used where a qualifying term needed to be tightly attached to some element content, e.g. if an element contained the numeric value of an amount of currency, an attribute on that element would be used to indicate which currency. If the currency was in a separate element, there would be too much risk of the wrong currency being associated with the wrong value;
- Attributes could be used if they had a global character, i.e. the same attribute with the same values could be applied to many different elements with the same semantic meaning.

Another consequence of the Schemas being data-oriented rather than document-oriented is that a **no mixed content** policy could be enforced. So elements could contain simple textual values, or child elements, but not a mixture of text and elements. This makes processing simpler, and is definitely a good design policy where there is no absolute requirement for mixed content (mixed content typically is only a requirement where the XML corresponds to marked-up text that will be displayed for humans to read).

WXS gives you the choice of both global and local definitions for both elements and types, so you need to have a policy on global versus local definitions. As will be discussed in Section 5, “Solution issue #3: naming”, local element names were chosen because they help in avoiding naming collisions. Local type definitions are interesting, in particularly because they are anonymous (unnamed). That means you can reduce your naming task significantly. However, for this project it was necessary to create Java classes to represent the contents of each Schema, and these anonymous types would have led to Java classes for those types with arbitrary, machine-generated names. Such arbitrary names would be confusing for developers to have to work with, so global type definitions were chosen. This does mean that naming conflicts are possible, but only a subset of the definitions were to be shared globally, with others confined to a single message. On that basis, global type definitions were acceptable, and in practice there were no problems from this approach. The use of global types and local elements for constructing Schemas is sometimes referred to as the *Venetian Blind* style of Schema design.

5. Solution issue #3: naming

It is often said that naming is the most difficult problem in information technology. The larger the system, the larger the number of names required, and the more difficult the problem becomes. When choosing a naming strategy for a large problem, it is important to avoid getting into **denial about the non-uniqueness of names**.

Humans typically have a vocabulary of 5–10 thousand words. The number of semantically distinct concepts that we express with these words is vastly more than 10 thousand. This means that we do not have unique names for everything.

Instead, we re-use names for different things, and we rely on the **context** of particular conversations or texts to disambiguate **which** usage of each name is the one that the speaker or author intends.

In technical areas, we commonly resort to both acronyms and jargon. Jargon in particular is not new, and is just a contextual re-use of existing words. When Sir Isaac Newton published his *"Principia"* in 1687 and described *force*, he took an existing word and gave it a precise definition in the context of physics[1]. This definition was similar to, but not necessarily the same as, the existing usage(s) of the day.

The reason for dwelling on this issue is that a common design paradigm for XML vocabularies (in the sense of element and attribute names) is to mandate globally unique names. This paradigm is inherited from DTDs (and hence from SGML), where element names are always global. If you have a `<name>` element in one part of a document, every `<name>` element in the document is the same `<name>` element with the same content model.

Now, many people will argue that this global uniqueness is a good thing. After all, if you have to deal with the names of both people and companies in a single document, isn't it better just to have unambiguous `<PersonName>` and `<CompanyName>` elements? In fact, where the problem is small enough and well understood enough from the outset to do this, it is an appropriate thing to do. It makes it easy to write matching rules for XSLT stylesheet templates, and it makes it easy to identify (in XML processing code) which part of your code processes which element.

However, in a large vocabulary that can and will evolve over time, the global naming approach can become problematic. Suppose, for example, that in version #1 of the schema[2] the only "name" that is involved is the name of a person. It is reasonable to use a `<Name>` element at this stage. You might be tempted to call it `<PersonName>` at this stage, to allow for other names in future, but that assumes that you can guess the future. Indeed, you might find in the future that instead of `<PersonName>` you need `<EmployeeName>`, `<EmployerName>`, `<BuyerName>`, and `<SellerName>`. As a general rule, you cannot assume that you know which names will become ambiguous in future, and which will not. It is worth remembering that **gold plating** of program code (adding functionality which isn't required at present, but which a developer feels **might possibly** be required in future) is frowned upon because it wastes development time and complicates the maintenance of code by obscuring the code that is used with code that is unused. The same principle is no less valid when it comes to the naming of XML vocabularies. You can do as much bad as good by making assumptions about which names will become ambiguous in future.

The only scalable approach to XML vocabulary naming is to allow the same name to be used with **different semantics in different contexts**. W3C XML Schema supports two mechanisms to support such re-use of names. One is XML namespaces[3], the other is local definition of elements (where an element name is defined to be unique and meaningful only within the context of a particular parent element or complex type). I use both in my schema designs. In the project in question, namespaces were used to disambiguate different messages, and **local element naming was used by default** so that element names were automatically disambiguated within individual messages. This is a common approach for data-oriented XML messages, and is used by the ISO 20022 standard [http://www.iso20022.org/] for financial XML messages.

Note that you need to have one global element, to serve as the root element of the document. Actually, with WXS, having only one global element is the **only way** to enforce a unique top-level element for your Schema; WXS allows any global element to be used a document root.

When choosing how similar or dissimilar names should be, I generally follow the rules suggested for naming in software libraries by Bertrand Meyer [Meyer1994], particularly for the names of individual messages. Meyer's argument was that things that are semantically similar should have similar names, and that things that are semantically different should have dissimilar names. At first reading, this may sound like nothing but a declaration of the obvious. However, when you are creating a large software or message library, consistency in naming is of extreme importance if your

[1] Pedants will note that Newton wrote the text in Latin, not English, so he didn't use the actual word 'force'. That distinction isn't important here.
[2] Within this paper, 'schema' with a lower-case 's' indicates an XML format in the most general sense, independent of any particular schema language.
[3] In the strictest sense, an XML element name is a combination of the namespace and the "local" (unprefixed) name of the element, so you don't actually have the "same name" in different namespaces. From a human perspective, it is the local name that is the "name", and this is the view that is taken in this paper.

users are going to be able to find the functions/methods/classes/messages quickly and efficiently. The longer it takes to find what you want in a library of any sort, the more likely you are to stop using the library.

In addition, I generally follow the approach of Winston Churchill, who wrote his speeches using **Anglo-Saxon/Germanic** words rather than **Latin/French** words. When you create vocabularies using English, you continually have to choose between these two vocabularies. Churchill chose the former because they are shorter and punchier, and I think that also makes for good software names. So I would prefer to call a message "`GetAccountDetails`" rather than "`RetrieveAccountDetails`".

I would also try to avoid having both names in the message library. Given these two names, it is simply difficult to know which one is the one that you want. Sometimes people are tempted to try and add semantics to the difference in naming, i.e. "`GetXXX`" for quick operations, "`RetrieveXXX`" for slow operations. However, such distinctions are often not robust, and can change over time (what happens if, after optimisation, "`RetrieveXXX`" becomes the faster operation?).

In spite of my earlier assertion that you should **not** expend effort trying to predict which XML element/attribute names will become ambiguous in future, I take a different approach to the names of complete messages. These names are critical to fast and efficient use of the message library, so to help avoid naming collisions as new messages are added, the names were chosen to be as specific as possible. For example, for someone who is writing an algorithmic trading system, whose behaviour is controlled by a set of parameters, it is fairly natural to request a message called "`GetParameters`". In the context of that system/component, these are the parameters. However, the full system is likely to have more than one message containing parameters, so in the bigger picture the message is better called "`GetTradingParameters`". You won't be able to avoid all possible clashes, but by keeping the names as specific as possible at this highest level, you can avoid some of the lost time that comes from having to rename messages and propagate those name changes through all of the affected systems.

Further, I also discourage message naming that implies a fixed RPC-style (Remote Procedure Call) message flow. For example, many developers will request a pair of message names like "`AccountDetailsRequest`" and "`AccountDetailsResponse`". The problem with this kind of naming is that it suggests that there is only the one response for the request, and only the one request for the response. Sometimes, you need the flexibility to have multiple requests for the one response, or multiple responses for the one request.

On that basis, I try to restrict names to either "noun" or "verb-noun" forms. The "noun" forms are for messages that carry data about one or more items. The "verb-noun" forms are for messages that request data or cause an action. So instead of "`AccountDetailsRequest`" and "`AccountDetailsResponse`", I would assign the names "`GetAccountDetails`" and "`AccountDetails`".

Finally, if your names are in a language with upper- and lower-case letters, like English, you need to think about having a case convention for names. The W3C doesn't give us any guidance on this (different W3C specifications using different conventions). In the financial XML world, the camel case conventions are dominant: **"lowerCamelCase"** and **"UpperCamelCase"**. These seem to be easier for business people to read than are names separated by hyphens, periods, etc. In my case, I chose UpperCamelCase with acronyms (like "URL") treated as words, i.e. if "URL" occurred in a message/element/attribute name, it would be written as "Url". The simple reason for choosing this convention is that is gives a very regular structure to names: repeating groups of an upper-case letter followed by one or more lower-case letters (all of our words/acronyms were two letters or longer). This allows you to introduce a level of automated consistency checking on the names. When you have a large library to maintain, any checking that you can do automatically is worth doing[4].

If all of these naming rules seem difficult, it is because naming isn't easy. Keeping the names consistent is easy when you only have a few names, but increasingly difficult as the number of names increases. The more rules you have, the

[4]Note, however, that UN/CEFACT's CCTS (Core Components Technical Specification) has rejected camel case in favour of using periods and underscores as separators (in a specific fashion), so that automated spell checking is easier. The terms between the separators start with an upper-case letter (for languages that have them), so that the readability is not so compromised. This is an approach that is worth considering when you make your own choice. I wasn't aware of the CCTS approach at the time the case convention was decided for the project described in this paper.

less room you leave for people to make arbitrary naming choices, and the **more consistent the result**. Never underestimate the importance of good, consistent naming to the usability of a large library of messages.

In practice, people's first reaction to these naming rules was to complain about how **draconian** they were. However, most people eventually came to appreciate the value of the consistency. You cannot expect people working on just one part of a system to see the big picture when it comes to good naming, so centralised review and oversight are required to make good naming rules work.

6. Solution issue #4: versioning

The W3C Technical Architecture Group (TAG) is currently investigating versioning requirements, strategies, and solutions for XML (in particular in relation to versions of W3C XML Schemas). That tells you that versioning for Schemas is far from having a universal best practice. The original W3C advice on Schemas, namespaces, and versioning can be summarised as follows:

- Namespaces identify a set of element names uniquely, and allow a specific context and hence specific semantics to be assigned to those elements. Namespaces should not change version unless the semantics change;
- Schemas describe how a set of elements are structured for particular documents. As the same element names and element semantics can be used with different possible Schema structures, Schemas versioning should be independent of namespace versioning.

On the face of things, this are reasonable suggestions. However, there are practical failings:

- The only information that an XML document can provide directly about its Schema is the `schemaLocation`. However, this is formally only a hint to the Schema validator, and validators are **free to ignore it**. Indeed, as a matter of business security you would normally ignore Schema locations in XML documents that come from external sources, because there are security and processing implications to accepting anybody's version of any Schema.
- When no `schemaLocation` is provided, Schema validators check the namespace URI to see if it corresponds to a valid Schema. This means that the W3C XML Schema specification has produced, intentionally or not, a default relationship between namespace URIs and Schema versions, and hence a **default coupling** between namespace versions and Schema versions.
- The W3C arguments for namespaces being independent of Schema versions were based on the example of XHTML, which has one namespace but three different Schemas. Unfortunately, HTML is a poor example because it is **not representative** of most XML. HTML has a huge global audience, and browsers producers have been able to justify building in expensive and robust support, like support for invalid content. In addition, HTML browsers have well-known rules for what to do if they encounter an element or attribute that they don't know about. In practice, most XML is for comparatively small audiences, and there isn't the time or money available to make applications robust to arbitrary changes to the XML. Applications need to be able to identify the specific Schema version that applies to a document, so they can make assumptions about the XML in that document which simplify the processing. You can't expect people to write applications that don't need the Schema validation and instead do **robust validation of their own** during processing. In general, it isn't cost-effective.

I should note that the UN/CEFACT Applied Technology Group [http://webster.disa.org/cefact-groups/atg/] (ATG) is planning to recommend an approach where **major versions** of Schemas are identified by the the namespace URI (which changes as required), but **minor versions** are identified by changes to the `schemaLocation` of the Schema. It will be interesting to see how well this works, given the *hint* status of `schemaLocation`.

For the project described in this paper, I chose the most common approach, to change the namespace for **each version** of the Schema, major or minor. This can have adverse affects on XSLT stylesheets, but those effects are avoidable if you write or generate your stylesheets appropriately.

In particular, my approach is to use a separate namespace URI for **each version of each message Schema**. This allows the message and version to be uniquely identified from a knowledge of the namespace URI, which works well in practice. That said, I have found that namespaces are sometimes lost from XML documents during processing, due to poor handling of namespaces by some software. On that basis, I mandated that every message Schema must include two mandatory top-level attributes, `MessageName` (e.g. "`GetAccountDetails`") and `MessageVersion` (e.g. "`1.0`"). I find this to be the most robust way to do Schema version identification. It is unfortunate that there is no standard for using such attribute information to **drive the choice** of Schema version for a Schema validator; personally I would welcome such a standard.

The messages had major and minor versions (e.g. "`1.0`"). The versioning rule was the most widely used approach, that **minor versions must be backwards compatible**, while major versions need not be. Backwards compatibility here means that if an XML instance is valid with respect to a previous minor version of the Schema (same major version), it will also be valid with respect to the current minor version[5]. Backwards compatibility means that you can

- add new optional elements or attributes;
- extend the cardinality (multiplicity) of an element or attribute (e.g. make a required attribute optional, or increase an element's cardinality from `1..3` to `0..unbounded`).

Backward compatibility means that you cannot

- delete elements nor attributes;
- reduce the cardinality of an element or attribute (e.g. make an optional element or attribute into a mandatory one).

However, for the project described in this paper, an XML to Java mapping layer was generated for each message. This meant that the more important backwards compatibility requirement was at the **Java interface level**, not the WXS level. This has to be managed just as you manage WXS versioning, but the backwards compatibility criteria are different. For example:

- Making a mandatory value optional does not change the Java API if the value corresponds to a Java object, since a `null` can be returned to indicate a **non-existent value**, without the Java API changing. Note that this is not possible if the value corresponds to a Java primitive value, e.g. `int`, `double`, `boolean`.
- Increasing the cardinality of a non-repeating value (`0..1` or `1..1`) so that the maximum number of occurrences is `>=2` is not backwards compatible, since it typically changes the return value from an arbitrary Java object or primitive value into an **array or vector** of object values (unless your API uses arrays/vectors for all return values, which becomes tedious if most elements are not repeated).

For the project described in this paper, it was necessary that both the Java API and the Schemas be backwards compatible for minor versions, so the **most restrictive interpretation** of "backwards compatible" had to be used.

7. Solution issue #5: documentation

A large software library of any sort needs good documentation, if users to use it and continue to use it. The same applies to large Schemas and large families of Schemas. Where you are re-using types with a Schema or between Schemas, it is easy to overlook the fact that **re-usable types require re-usable documentation**. What makes documentation re-usable or not? I usually express it in terms of **documenting upwards versus documenting downwards**.

To explain what these concepts mean, imagine that you are creating a Schema to represent the information for a sale of something. Within that Schema, you need to represent both the buyer and the seller. Now, if the buyer and seller

[5] You can also talk of *forwards compatibility*, where new XML instances are valid with respect to all previous minor versions of the Schema. This was not a requirement for the project described in this paper.

have the same information (which is common), it makes sense to use a common `Party` type to represent both. So how do you document `Party`?

A common mistake, from my perspective, is to write something like "*A buyer or seller in a sale*". This is **documenting upwards**, where a type describes where it will be used. This makes the type hard to re-use. Suppose you need to add some new party to the sale, e.g. an agent or other intermediary. You can't re-use the `Party` type without editing the documentation. You shouldn't have to edit something whenever you want to re-use it; the cost of that quickly adds up and makes your library too expensive to use.

A better approach is to document only what the type represents independent of any usage contexts, and of what its content is. This is **documenting downwards**. So a more appropriate description would be "*Details of a party, including name and contact details*". This kind of description doesn't try to describe where the type will be used, and makes re-use quicker and simpler.

For *Venetian Blind* Schemas, where types are global and elements are local, elements are what provide context to particular usages of types, so only element documentation should include context-specific information, such as for `<Buyer>` or `<Seller>`.

8. Project notes

The message Schema development has been a successful part of the project, and the architecture and rules described in this paper have allowed the development and maintenance of hundreds of Schemas without cost or time overruns. Although the team at one stage peaked at 5 people, only 1–2 are required for ongoing development and maintenance (<1 in resource terms, but you need at least two people trained in the work to allow for sickness, holidays, or someone being run over by a bus). Turn-around of message changes is fast as compared to the time taken to turn-around changes to the application code that uses those messages, which is an important metric.

The system is expected to go live in 2006. Performance testing to date indicates no XML performance issues, i.e. the time taken to generate, transport, parse, and validate XML messages is not a significant part of the time taken for any major end-to-end process in the system. Gratuitous XML performance testing, where performance optimisations are made before knowing if they will have any significant impact on end-to-end process times, has been avoided since it can lead to design decisions that make the system more difficult to maintain but return **no extra value for that extra cost**.

An issue where improved tools are needed is **regression testing**, where new versions of Schemas are tested to check that the required changes have been correctly added, and that no new bugs have been introduced. In principle, you can regression test Schemas against a large body of test examples, but you may need tens or hundreds of test examples for each of hundreds of messages to test each Schema fully, and that is a major undertaking to create and maintain. Tools like Schema Unit Test [http://sut.sourceforge.net/] (SUT) are a help (SUT allows you to create a single marked-up XML document with indications of how to add/remove/change content to effectively generate many XML test documents), but we still need better tools, and tools that integrate with standard software build environments (e.g. Ant [http://ant.apache.org/] or Maven [http://maven.apache.org/] for Java).

9. Conclusion

In this paper, strategies used to successfully implement a large family of related W3C XML Schemas have been described. Key issues are managing shared definitions (a repository approach is recommended), managing naming, and managing unambiguous identification of Schema versions against which messages are valid.

The author's e-mail address is abcoates@londonmarketsystems.com [mailto:abcoates@londonmarketsystems.com], and the author's weblog [http://kontrawize.blogs.com/kontrawize/] regularly discusses the key topics from this paper and related topics.

This paper was written in XML using Syntext Serna [http://www.syntext.com/products/serna/index.htm]. Thanks to Syntext for providing Serna support for the XML 2005 conference schema.

Bibliography

[Meyer1994] Bertrand Meyer Reusable Software: The Base Object-Oriented Component Libraries Prentice-Hall 1994

Biography

Anthony B. **Coates**
Information and Software Architect
London Market Systems Ltd
33 Throgmorton St
London
EC2N 2BR
United Kingdom

Anthony B. Coates (Tony) specialises in information management and integration solutions for financial and corporate clients. Tony is the Information and Software Architect for London Market Systems, and previously was Leader of XML Architecture and Design in Reuters Chief Technology Office in London. Tony works on a number of financial XML initiatives, including ISO 19312 (Securities Data Model), MDDL (Market Data Definition Language) and FpML (Financial Products Markup Language). Tony is also a member of the OASIS UBL TC and the UN/CEFACT TMG (Techniques and Methodologies Group). He has worked with XML since 1998, and Java since 1996.