Beyond Theory:
Making Single-Sourcing Actually Work

Liz Fraley
Juniper Networks, Inc.
1194 N. Mathilda Avenue
Sunnyvale, CA 94089
408-745-2000

liz@juniper.net,
acm@caltonia.com

liz.fraley@single-sourcing.com

ABSTRACT

In this paper, I discuss how we made single-sourcing work at Juniper Networks. This is a practical discussion of issues, problems, and successes.

Categories and Subject Descriptors

D.3.2 [**Language Classifications**]: Extensible Languages—*Using XML/XSLT for documentation production, single sourcing, implementation issues and successes*

H.4.1 [**Office Automation**]: Desktop publishing—*publishing XML to HTML, publishing XML to PDF, converting FrameMaker to XML*

H.5.3 [**Group and Organization Interfaces**]: Collaborative Computing—*using branched repository for collaborative authoring*

I.7.1 [**Document and Text Editing**]: Document Management, Version Control—*branching documentation, best practices, when to branch*

I.7.2 [**Document Preparation**]: Desktop publishing, Languages and systems, Markup languages, Multi/mixed media— *publishing XML to HTML, publishing XML to PDF, converting FrameMaker to XML*

General Terms

Documentation, Performance, Management, Design

Keywords

XML, XSLT, Java, case study, single source, single-sourcing, modular writing, chunking, documentation, publishing, branching, Interwoven, TeamSite, Arbortext, Epic, E3, FrameMaker, WebWorks, Juniper Networks, commit

1. INTRODUCTION

This paper provides a case study in single-sourcing, examining how the Technical Publications group at Juniper Networks has been implementing single-sourcing.

In the past three years, there has been much theory about single-sourcing, but not enough practice. The literature is full of information about single sourcing from a theoretical perspective. So, this paper is more about what isn't covered in the literature than what is.

For example, it's not about how to choose a tool or evaluate a product, how to code XML, how to get cost savings through single-sourcing, how to write modularly, or how to structure your documentation.

And it isn't about amazing product features, the Juniper Document Definition, the Juniper-specific applications that we developed.

Most importantly, this paper is not a set of generalized rules for making single-sourcing work. It is one long a concrete example because, in the end, that is what the developer of a single-sourcing system needs to see. And interestingly enough, it's what the users of that single-sourcing system need to see, too.

1.1 The Single Sourcing Literature

For all of the topics that I am not covering, the existing authorities—Hackos, Rockley, Ament—have everything the beginning single-sourcer could need. Their books (and conferences) are extremely useful. They are full of detailed information to teach managers, writers, and document designers how to think about single sourcing.

In fact, these authors were influential during our initial document design. Our original documents, style guide, and templates were all designed for eventual conversion to a single sourcing system from the very start. The document design included the methodology that defined the way documents would be converted to online html documents. And although we were using FrameMaker and WebWorks at the time, the definition behind the

methodology carried over into our single source publishing implementation.

The only problem with the existing literature is that nearly all of it is theoretical in nature. Most books on single sourcing contain advice about planning, managing, and creating modular projects and documentation. At this, they are very good. What they're all missing is the bridge between theory and practice. And they're not alone.

I was hired at Juniper Networks specifically to architect (and then implement) the single-sourcing project. When I started, I had no practical background in XML; I spent my first six months reading all the single sourcing literature I could find. I also attended JoAnn Hackos's Single Source conference almost before I started researching.

As an engineer, I found that most of the single-sourcing literature was aimed at writers or managers.

I was hired to design and implement not to manage. I wouldn't be selling upwards: my director was championing this project throughout the company. She knew that the current documentation methods (FrameMaker and WebWorks) would not scale. The company would continue to grow; the demand for documentation would increase while staffing and resource challenges persisted. She would be the one determining return on investment (ROI) and measuring success.

Although I would help determine which tools we eventually chose, I was not the project manager. My direct manager would be managing the project and its resources.

What was not aimed at managers was aimed at writers: guidelines for writing and designing modular documentation. This is something I would not be part of and should not be. The writers who would be using the single sourcing system would be planning their documentation, just as they always did.

This sort of information was valuable as a look at the point of view of the user, but it wasn't what I was looking for as an implementer. But I knew that these books would be essential for training the writers to write and think modularly.

1.2 The Programming Literature

The way that modular writing works is very similar to methods for code reuse found in Object-Oriented programming literature. Code reuse is the assertion that if you build generic objects they can be used and reused. It is the idea that you can isolate functionality into a module (function) and then use that module rather than rewriting the code. The ideas are the same.

Unfortunately, the programming literature faces the same implementation gap, from the other side. The XML programming books, which don't describe its implementation as a language, describe the multitude of ways you can use XML. They tell you how to write the XML and how to process it: They do not tell you how to make XML work in a single sourcing environment.

In addition, these books are not aimed at either of the groups that the single sourcing documentation targets. XML authors assume their readers have a programming background and already understand programming concepts.

1.3 The Bridge between Single-Sourcing and XML

Ament says it best: "Single sourcing is a methodology, not a technology" [1]. XML is a technology, not a methodology. Bringing the two together is not obvious or well-defined.

Many companies try to sell systems that bring it together. But in the end, "to ensure success, develop local, project-based standards for modular writing. Base your standards on what actually works in your own projects" [1].

We did not manage to find one system that worked for us. And we did not find one book that describes how to put it all together.

We made choices—good and bad—along the way that influenced the way we implemented particular pieces. We chose a set of tools. We did very little customization of those tools but built a custom HTML generation program, so authors would not have to worry about pagination issues.

My goal is to provide specific examples like these that can serve spark ideas to solve someone else's real problems. That is the best any case study can do: Give you an idea about what you can try.

In this paper, I will describe what we developed and the tools we are using as a context for the lessons we learned and the choices we made in our pursuit of single sourcing.

2. THE CONTEXT

The single-sourcing project at Juniper Networks is a major technical publications initiative designed to convert existing documentation and author new documentation in a single-sourcing environment. Our goals were to increase the efficiency of our entire staff as the demand for documentation increased while staffing and resources did not.

2.1 Our Department

The Juniper Networks technical publications department handles 116 books that have between 100 and 800 pages each. Of these books, 20 exist in 4 different physical incarnations to match the 4 active software releases (for a total of 80 active documents). These same 20 books are owned by a total of 10 writers (5 books per writer). Two writers manage the remaining 36 hardware books.

We have 2 editors who perform at least two edit passes for 25 books every 3 months; the editors also perform at least one pass for as many other books, which require updating, as they can squeeze into the release schedule. Editors work with change bars and whole documents during each edit cycle.

2.2 Our Team

We divided the work into several roles, but not necessarily several individuals. These roles include:

- Project lead—oversees all aspects of the project, sets schedule, manages resources

- Tools lead—interacts with IT, does any product customization to improve user experience, evaluates new versions and new technology [This person should be an engineer].

- Single-source architect—architects operation of the overall system and the individual pieces

- XML expert—provides direction in initial planning, develops initial DTD and FOSIs

- DTD lead—maintains and enhances DTD(s)

- Mapping developer—develops mapping file to convert FrameMaker files to XML files

- FOSI lead—maintains and enhances print and screen FOSIs, resolves FOSI limitations

- HTML lead—implements HTML generation

- Graphics lead—develops naming conventions, resolves format issues for print and online

- Process guide authors—document tools and how to use them, document new processes

- IT—installs, configures and troubleshoots tools

3. THE IMPLEMENTATION

We expected a single sourcing environment to address several key points. The new authoring environment had to be user-friendly and easy to learn. The look and feel of the published documents had to be comparable to the existing published documentation. Most importantly, it had to be able to scale easily, so we could make enhancements and changes without major infrastructure changes going forward.

We wanted an environment that allowed writers and editors to be more productive. Our goals were to promote efficiency and productivity by reducing maintenance and overhead. We looked for tools that supported scaling and productivity. Also, we decided to tune skills, workflow, and processes to new tools so that the department and company achieve success.

By leveraging XML technology and single sourcing methodology, we would be able to write something once and use it multiple times within a single book as well as between books. We could edit once. We could revise once and have the change populated everywhere.

The production requirements were perhaps the most critical to the project. We had to be able to easily publish to multiple media: print, PDF, HTML, and PDA (Palm, Microsoft eBook, CD-ROM).

Moreover, the new process had to scale: it needed to be able to handle a growing documentation suite with a reduced production staff.

3.1 Our Original Environment

Our original environment was a traditional publishing environment. Source control was minimal. The FrameMaker model of lock-modify-unlock did not allow collaborative document authoring. Authors routinely clobbered their old documents on the network drive. Modifications could not be safely done by multiple authors unless they were made serially. Production was time consuming. Any change to the source required all output formats to be completely regenerated.

These programs and systems made up our original environment:

- Adobe's FrameMaker 6.0

- WebWorks Publisher 2000

- Network drive for storing documents

- FrameMaker's locking mechanism for document source control

3.2 Our Single Source Environment

The tools we chose were a compromise that addressed the goals we defined at the beginning of the project. In our new single-source environment, we are using the following tools:

- Authoring: Arbortext's Epic Editor

- Editing: Arbortext's Epic Editor, paper

- Content Management: Interwoven's TeamSite (server and web client)

- Publishing PDF: Arbortext's E3

- Publishing HTML: Java/XSLT

All content, including all entities (modular chunks of information), is created in Epic Editor. TeamSite acts as the repository for all content and support tools; it also provides individual work areas—sandboxes—for all authors, editors, and developers. Epic interacts directly with the TeamSite server; the TeamSite web client invokes the E3 for document production.

The E3 uses FOSI stylesheets for PDF production. The Epic Editor client uses a different FOSI stylesheet for on-screen viewing. There is both a screen FOSI and print FOSI for each distinct DTD.

We have a java program, htmlBuilder, which produces HTML output and which is not yet integrated into the TeamSite workflow. This program uses a set of XSLT stylesheets to determine the production of HTML pages for each DTD.

Additionally, we use TeamSite for publishing documents directly to the web.

3.3 Our Content Model

Our information is multidimensional: Information is divided across multiple books at one time. Several books also exist in multiple versions at the same time.

Reuse happens at the chapter, section, and even paragraph level. Both text entities and file entities are used to define modular chunks. At times they are used together to create conditional text (see Section 4.5.2).

Reusable content is stored in branch-specific libraries (including canimages). Branches identify versions at specific points in time. Content is shared across documents and across branches.

4. THE SUCCESSES

Within a year of beginning full-time development on the single-sourcing project, we were able to publish our first set of eight books fully within the single-source environment. This included creating the DTD, identifying information for reuse, creating the file entities, creating the print FOSI, integrating the E3 engine with our Interwoven environment, and developing some of the support tools (XSLT, HTMLBuilder, clean-up scripts, and so on). More importantly, we were able to author these guides with literally one quarter of the work required prior to the migration to a single-sourcing environment. The re-use of information across multiple books was essential to being able to deliver these books on time amid staffing changes. Being able to edit a single chunk of information rather than the same chunk multiple times was crucial to this success.

Since that time, we have been able to successfully convert 14 more books, create 13 new books, update the first 8 books, and push all 35 books all through production.

By the time of the conference, we will have converted 3 more books, created 6 new books, updated 14 existing books, and pushed all 23 through production. In addition, we will have 33 books in XML, and, including updates and new authoring, we will have published a total of 68 books using single sourcing.

4.1 Changing Look and Feel

Our editing team has been extremely responsive to the limitations of XML authoring. For example, they have managed to reduce the number of cross-references and homogenize the look and feel. We now have much fewer and more consistent cross-references that have well-defined usage rules.

We altered the look and feel our unconverted documentation to match the XML-generated documentation. Any items we could not produce with the single sourcing system (such as the graphic down the side of the page) were removed from the FrameMaker templates, so the traditionally produced documentation would look the same as the XML-produced documentation during the transition period.

No one has been afraid to compromise or to find alternative suitable look and feel substitutions.

4.2 Small Steps

We didn't convert all the documents at once. We prioritized the books and set out a plan for converting books over the course of a year.

4.2.1 Network Card Books First

Our first books were the network card guides. These books are small books (about 50 pages) that share nearly 90 percent of their content. These books had their own DTD that contained a subset of the elements in the other DTDs. The DTD, FOSIs (PDF generating stylesheets), htmlBuilder, and the necessary XSLT stylesheets required less time to develop because they only include a subset of the total elements. Creating tools to manage a subset gave us the time to polish the way everything functioned and the way production produced output. Because these books update four times per year, the tools team had plenty of time to flush out bugs.

In addition, the author assigned to these books was not senior but was enthusiastic about having her books converted. Since so much of the content of these books was shared, she looked forward to reducing the amount of time she spent updating them.

4.2.2 Hardware Books Second

After the network card books, we chose to convert the hardware books. These books average 250 pages and include a greater subset of total elements. These books also only update when necessary. We had two quarters to get these books converted and the tools development completed.

The hardware book was sufficiently different from the PIC Guides, so we developed a separate hardware DTD, FOSI, and XSL stylesheets. By this time, we had also implemented TeamSite, the content repository.

The author assigned to the hardware books was a senior writer who knew the entire set of books very well. He performed the book analyses and decided which chunks of text to turn into entities and where to place them in the library structure.

4.2.3 Software Books Last

The software books were left to last because they have a very unusual requirement: These books are authored, edited, and published in the span of two months. Once these books go to conversion, the tools must be ready and polished, so conversion and publication can go smoothly and without several rounds of repeated effort.

4.2.3.1 Some Books Have Software-Generated Text

At least three of the software books will go before the rest. These three books all have content that is generated by the software engineering teams. In one case, the content is generated directly by the software build process. In another case, the content is generated directly from the bug database. The Engineering department and the Software Tools group have both been extremely supportive of the single sourcing effort. Currently, both groups produce MIF files that the authors then reapply the FrameMaker templates to. Once the software tools are ready, they will generate XML according to the software DTD, so the author will only need to create a file entity and insert it into the book.

4.3 Tagging Determines Style

One of our requirements was to minimize the number of ways that style could be altered by the author.

We chose to let tagging and context determine style. We specifically decided to omit elements like <emphasis> that have attributes like "italic" and "bold." Italic and bold are determined through the use of other elements when used in a particular context. We reasoned that if some text is important enough to mark up for style, it is highly likely that same text should be marked up for content.

For example, the text inside the <citation> element is always italicized; because of the citation markup, it is also always identifiable as a "citation."

We also created a tag to identify variables. Because we document command line software and show lots of examples, variables need to be appear visually different. The text inside <variable> markup is always italicized and, here also, has the added benefit of being identified as a "variable" by the markup as well.

If both variable and citation were simply marked up with the <emphasis> element with the italics attribute, an author searching for variables would get citations as well as variables as results of the search. However, searching for <variable> brings up only variables and the italicizing is left to the stylesheet.

Remember: if it's not marked, you can't search for it.

4.4 Chunks Are Entities

Reusable content that is shared between files is always stored in a file entity. Reusable content that is shared within a single document is stored in a text entity. Complicated chunks—chunks with a large amount of markup—are stored in file entities instead of text entities for easy revision.

Every book has a top-level "book" file that contains data that pertains only to the book as well as pointers to any file entities required to complete it.

4.4.1 Create Entities for Shared Content

We created text entities for trademarked names and small pieces of boilerplate information that are always shared. Updating any of these entities automatically updates that text in every book that includes those entities. This behavior has been extremely useful for maintaining the copyrights and the preface.

4.4.2 Use Scope to Create Conditional Text

By using entities, in the following two ways, we have managed to achieve the behavior of conditional text:

- Make entities out of shared content across multiple files if most of the content is not shared.

- Make entities out of unshared content in a single file if most of the content in that file is shared.

It feels like scope works in reverse. Every XML document is an independent object. There is no concept of encapsulation in XML. Saying that File A includes File B means that File A defines an entity that points to File B, and File A uses that entity somewhere. A does not literally include File B: it defines an entity that points to file B:

<!ENTITY file "fileB.xml">

A could just as easily define the "pointer-to-file" entity this way:

<!ENTITY file "fileC.xml">

So what happens when you chain files? File A includes File B, and File B includes File C.

Saying that File A includes File B and File B includes File C is the same as saying that File A includes Files B and C and File B includes File C. Both File A and File B list all files they include.

In the simplest case, the definitions for File A would be:

<!ENTITY file-1 "fileB.xml">

<!ENTITY file-2 "fileC.xml">

The definitions for File B would only include the pointer to C:

<!ENTITY file-2 "fileC.xml">

So what happens when the definitions do not agree in the two files?

What if the definitions look like this?

File A:

<!ENTITY file-1 "fileB.xml">

<!ENTITY file-2 "fileD.xml">

File B:

<!ENTITY file-2 "fileC.xml">

If you process only file B, you see the contents of FileC where the entity is used. However, if you process File A, you will see

the contents of FileD where the "file2" entity is used, because A redefines the "file2" entity. It doesn't matter how "file2" is defined within B as long as the variable is also defined in A.

The top-most entity has the last word on entity definitions; we can make small entities out of unique content if most of the content in the larger entity is shared. In the book file, the small entity can be redefined to the appropriate book-specific content.

4.4.3 Chapters Are Always File Entities

We made this decision based primarily on the method we are using for initial conversion. Our existing FrameMaker-based documents have a book file and several chapter files. Rather than appending all of the files together into one large file and converting that large file to XML, we decided to convert the individual chapter files.

This leaves us with several initial file entities that belong in the book-level library. However, we have found that this makes speeds up the response time of Epic Editor. Epic caches information (see Section 5.2), and loads the entire document into memory. Writers can author in the smaller chapter files rather than in the full book file. By working in smaller files, the client response time is much quicker and easier to use.

4.5 Content Libraries

All reusable content exists in the libraries; only reusable content exists in the libraries. Libraries exist at multiple levels in the directory structure. The TeamSite repository is structured like a tree. At every level in the tree, a library exists to hold files (modular chunks) common to the files at that level and below.

Table 1. Libraries exist at multiple levels

Level	Purpose
Corporate	Shared across all books, corporate-wide
Document Type	Shared across all hardware books or all Software books
Series Type	Shared across all M-series hardware books or all T-series hardware books
Book level	Shared across files in a single book

The image library is the only exception: all images live in the same library. We use a parameter file entity, declared in the DTD, to define the declarations for all images. In this way, graphics only need to be declared in this one file to be available to all authors and all books.

4.6 Branches not a Database

The software books exist in four (and sometimes five) different versions at the same time. The versions differ by at most 5 percent; the "shared" material is 95 percent from version to version.

TeamSite is a file system, not a database. Three factors contributed to our choosing Interwoven's TeamSite over the typical database implementation:

- We decided that the documentation—including the library—branched rather existed in a three metadata-controlled dimensions.

- We inherited Interwoven in a merger. Our IT department had been looking for a content management system for the company but had not made a final decision when the merger took place. The other company had Interwoven. We got it.

- Resource restrictions lead us to choosing a branch-based implementation over a database-based implementation. We did not have the staffing resources to implement a full database solution. We had only one full-time programmer devoted to the project's implementation.

The programmer in me would really rather see this as a database application, but I'm not convinced that the database implementation is the right one for this document set. On the other hand, the file system seems to be an appropriate choice for our problem domain so far.

First, lots of dimensions are hard to visualize. It would be even harder to make sure that the metadata required to support a database implementation could be maintained. Certainly, a complex metadata scheme would difficult for a completely inexperienced user base. It may be that eventually a database implementation may make a nice second generation system after the users are comfortable with both XML and the source-control process. Small steps.

Second, we end of life (EOL) our documentation on a regular basis. Every three months any metadata or chunking related to a release becomes completely obsolete and useless. I believe that it will be easier to use a simple XML patch program to propagate bug fixes across branches than to over chunk—or worse, rechunk—content.

By choosing branching over database metadata, we can let the software books can branch with the software release to which they apply. Information is modified from release to release. And although information is 95% the same, the book analysis showed that most of the modifications were in existing feature descriptions rather than in new information. Chunking changes in the middle of a section is more difficult than patching changes with an intelligent, tag-aware, XML patching program.

4.6.1 Why Do You Branch?

Branching is a behavior more commonly seen in software configuration management Tools (SCM). Specifically, branching is "the creation of variant codelines from other codelines" and "the most problematic area of SCM. Different SCM tools support branching in markedly different ways, and different policies require that branching be used in still more different ways" [2].

When do you branch? Only when necessary.

Branches are used to provide single-purpose individual work areas that are not shared between developers. A workspace is the place "where engineers edit source files, build the software components they're working on, and test and debug what they've built…sometimes they are called 'sandboxes'" [2]. Sharing files creates confusion just like sharing a desk.

Branches are created when files contain features with incompatible policy. For example, when one development group does not wish to see another development group's changes, that is a form of incompatible policy. Each release has its own policy; changes to different releases cannot be seen across branches unless intentionally applied.

Like incompatible policies, incompatible processes can also be applied to on a per-branch basis. And branching systems typically support audit trails out of the box. Authors can have work areas, on each branch, that are subject to the different policy definitions.

Branches also allow the production of nightly builds. The E3 is capable of performing batch builds. Later in the implementation, we intend to build all the books nightly and to deliver a PDF of each book to the editing team twice a month. We expect this to help minimize the amount of incorrect tagging, invalid tagging and context errors. We also expect that this will help to preserve the integrity of the repository.

Remember: catch XML errors early. Production time is not the time to do debugging.

4.7 Printing Available Only From Staging

For each branch, TeamSite provides a central repository (called "staging") and a work area (sandbox) for each user. Work areas contain local copies of the documents contained in the staging area. Authors commit changed files to staging for sharing between work areas.

TeamSite manages merging and version control through the copy-modify-merge model of document collaboration. Authors can get a copy, edit freely, and commit their changes back to the repository.

When initially integrating TeamSite with E3, we found that we could not connect it to an individual author's work area easily. As a result we made a policy decision: in order to create a PDF of a document, that document (and all its required entities) must be committed to Staging. This means that authors cannot publish documents that exist only in their work area.

This decision seems like the wrong choice: authors want to create PDF documents without publishing their files. However, if a document is ready for printing, then it has, by definition, reached a level of maturity. And, if a document has reached a publishable maturity, it must be checked into the repository and submitted to staging.

4.8 Our DTD with CALS Tables and Docbook Indexes

We decided to create our own DTD that reflects the book structure of our original document template. However, the implementation includes two standard element structures that take advantage of well-tested code written by more-qualified authorities.

We chose the CALS table model as our table model. There is a great deal of support for this model within the XML community. Epic came with embedded table handling for the CALS model which made the FOSI code easier to implement. It also came with sample XSLT code for converting CALS tables to HTML tables.

We also chose to use the Docbook Index element model. This allowed us to use well-tested, index-generation code in the FOSI. Generating the index for print was the only time we used Arbortext Consulting during our implementation. Because we used the standard Docbook index definition, Arbortext was able to deliver the index-generation code to us in a single day.

5. THE PROBLEMS

We continue to encounter problems based on the decisions we made. However, we also continue to learn from them. This section lists several issues we faced and the lessons we learned as a result.

5.1 Epic Caches Information

One of our biggest issues is that Epic Editor caches information which is only refreshed at load time. This is both an advantage and a disadvantage.

Caching the DTD definition lets Epic do real-time verification during authoring. This is a real advantage. Authors can edit freely, without being required to know the DTD inside and out.

It is also a disadvantage: entities are not always updated when you change it outside of the editor. This is a problem for work areas: users can update a file in their work areas, but not see the change in the document that is open in their editor.

We have found other various inconsistent behaviors here and there. Unfortunately, we have not been able to isolate the causes well enough to submit bug reports back to Arbortext.

5.2 Tool Choice Timing

We spent entirely too long choosing our content management system. In part, this effort was a calamity of errors. We were trying to accomplish too much and to accommodate the needs of too many different departments; instead, we should have focused on the best tool for our needs.

I should mention that we had the authoring tool (Epic) chosen at least a year before the content management system was decided.

Originally, a committee drove the selection of the content management system. Web publishing, software tools, technical publications, and several other groups were all trying to find a solution that would fit everyone's needs. Part of this was budget restrictions: the systems are expensive enough that a company can really only afford only one. As a result, that one must suit a variety of needs.

We also spent too much time going back and forth on the decision between a file management system and a database system. I continue to think about the choice we did not make and know that we do not have the resources to pursue that option.

5.3 Scheduling

Failing to create a reasonable and well-planned schedule was one of the greatest issues we faced. We did not have any one who had project management experience on the team, and no one had enough real experience to create realistic schedules.

5.3.1 Unrealistic Goals

Our goals were defined by MBOs (Management Business Objectives). We often worked toward those goals rather than spending the time required to make the project work correctly. We regularly ended up spending the last three weeks of the quarter kludging bits together so we could claim that our MBO had been achieved. We would then spend the first three weeks of the next quarter fixing the bugs that we had cobbled together the quarter before. Each subsequent quarter of badly stated goals only served to compound the mistake; the amount of work not done right the first time carried over from the previous quarter but was not accounted for in the current quarter's MBOs.

5.3.2 Inexperienced Staff

Early on, we hired a contractor to get us started. While I continue to believe this was the right choice, we did not go about it the right way.

Hiring an expert early on allowed our completely inexperienced team the time to learn the technology (XML and Epic). It also provided us with a much-needed running start and some concrete examples of how to make it actually work (we did this again later on when IT began implementing Interwoven).

Unfortunately, we did not have large enough budget to employ the XML expert full-time. Instead, she worked only 40 hours per month, or one-quarter time. To her credit, she put in more hours than she billed us for; on the other hand, she was not the expert we thought she was. Also, neither of us knew how to estimate effort for a single sourcing project. Her estimates were regularly off. She would estimate effort that made sense for one-quarter time, but she would be estimating for full-time.

That said, she completed the initial development in six months. We would have done much better to have hired her full-time (but

that would not have spread out the budgetary dollars). We could have developed the application much more quickly from the start.

5.3.3 Limited Staffing

For the most part, we have had one engineer working on the project development. For six months we had the XML expert, who had a technical publications background, not a programming one; for another six months we had an engineering intern who did remarkable work for us in the short time she was with us.

Unfortunately, when we had the engineering intern, we had a limited problem space and limited sample data. Much of the work she completed had to be dramatically revised to accommodate the more complete DTDs (the superset of elements). This failure is as much my own fault as anyone else's. All our resources were so buried with unrealistic development goals; no one really had time to do testing and proper development.

I would not attempt a project of this scale with only one engineer again. We had IT support for the two major applications (TeamSite and E3) but that support ended where the application did. Any single-sourcing project is bigger than the applications it uses. It requires a significant amount of tools development that is entirely independent of the applications. The tools development must take up where the application (and the company that develops it believes that it) leaves off.

5.4 Selling Upwards

Our project continues to face the threat of being cancelled by upper management. This project was originally sold to a management team in the early start-up days. At that time, the company had a pile of money and knew that it would last forever.

Since then, times and the executive team members have changed. When the management team changed, the project should have been re-sold to them.

To be successful, this kind of project really requires 100% buy-in from management. If not, the development team is continually asked to justify its existence and required to prove its worth. The problem discussed in section 5.3.1 is a direct result of the failure to re-sell this project to management.

5.5 Author Issues

We are facing the same author issues that the single-sourcing authorities are described in every one of their books. However, some of the choices we made have resulted in at least one issue I have not seen discussed previously.

5.5.1 No Offline Authoring

As of the time of this writing, authors cannot work offline. Authors must be connected to the network to author documents in our single sourcing system.

Three things contributed to this decision:

[1] To work offline, authors would be required to be diligent in keeping their DTDs, stylesheets, and libraries up to date.

[2] Epic has at least one environment variable that is set in the user's operating system. Authors would need to change this variable each time they start working. It would need to point to one place when they are online and a different point when they are offline.

[3] Interwoven provides individual work areas on the server. It has not been smart at noticing that a freshly copied document matches an old version rather than appearing to be a completely new version. To make offline authoring work, Interwoven would need to look at the timestamp second rather than first.

5.5.2 Book Ownership

Even our most senior writer has issues of book ownership. As the tools team fixed bugs, we changed the content markup in the hardware books to match our changes. Learning to merge changes and to let other people make changes in "his" books continues to be the most difficult challenge we face. It affects not only the author's productivity but the author's ability to get comfortable with the content repository and to use it for documentation.

6. CONCLUSIONS

You make decisions; you live with them. Hopefully, you also learn to make better decisions and learn how to improve the situation created by the worst of the ones you made.

When I started this project, I went looking for the information in the middle: the information that joined the single-source theory to the XML implementation. In the end, I learned how to create that information from source code developed by the XML expert that got us started and from the ramifications created by the choices we made.

Much of what I learned will be useful to me if I ever do this again somewhere else. I definitely enjoyed doing it. I love seeing old technologies applied to new domains: Technical documentation discovers object-oriented concepts in a real, practical way. It's great. This is what I got into programming for in the first place.

8. RESOURCES

I wanted to include a list of resources. In addition to the references list, these are the books that I found most useful while implementing single sourcing. The topics are varied but each contributed in some measurable way to the project's success.

[1] Carlis, J. and J. Maguire. Mastering Data Modeling: A User-Driven Approach. Addison-Wesley, Boston MA, 2001.

[2] Fogel. K, and M. Bar. Open Source Development with CVS. Coriolis, Scottsdale AZ, 2001.

[3] Holman, G.K. Definitive XSLT and XPATH. Prentice Hall PTR, Upper Saddle River NJ, 2002.

[4] Holman, G.K. Definitive XSL-FO. Prentice Hall PTR, Upper Saddle River NJ, 2003.

[5] Garshol, L.M. Definitive XML Application Development. Prentice Hall PTR, Upper Saddle River NJ, 2002.

[6] Maler, E., and J. El Andaloussi. Developing SGML DTDs: From Text to Model to Markup. Prentice Hall PTR, Upper Saddle River NJ, 1996.

[7] McConnell, S. Rapid Development. Microsoft Press, Redmond WA, 1996.

[8] McConnell, S. Software Project Survival Guide. Microsoft Press, Redmond WA, 1998.

9. REFERENCES

[1] Ament, K. Single Sourcing: Building Modular Documentation. William Andrew Publishing, Norwich NY, 2003.

[2] Wingard L, and C. Seiwald. "High-level Best Practices in Software Configuration Management." http://www.perforce.com/perforce/bestpractices.html. Viewed: 2003.