

2009

PACIFIC NW SOFTWARE
QUALITY
CONFERENCE



MOVING
QUALITY
FORWARD

OCTOBER 27-28, 2009

Conference Paper Excerpt

From the

CONFERENCE
PROCEEDINGS

Permission to copy, without fee, all or part of this material, except copyrighted material as noted, is granted provided that the copies are not made or distributed for commercial use.

Developing Requirements for Legacy Systems: A Case Study

Bill Baker and Todd Gentry

Abstract

Many legacy systems were created without documented requirements. Over the years changes have been made, often without adequate documentation. Software quality suffers as the system becomes more and more complex. This paper describes a case study of bringing requirements management and other related process improvements to a software product, which has been successful for over twenty years.

Eight years ago, we realized that we needed drastic improvements in requirements. It became obvious that just documenting changes to the system by each development team was not adequate. We had cases where one team counteracted changes made by another team in a previous release, causing bugs to reappear.

The product utilizes rules based on compliance and regulation from multiple sources; federal, state, and local. The company offers a significant guarantee of compliance and meeting of the regulations. Quality problems represent a significant business liability.

A cross functional team of management undertook a project to significantly improve processes related to requirements. These included centralizing requirements using a newly purchased requirements management tool and developing guidelines for writing requirements. Much of this paper describes the guidelines that were found useful, as well as some failures. We will show how the implemented process improvements have eliminated the quality problems that were the original impetus for undertaking the project and have improved the overall quality of the product.

We will discuss the following, among other things:

- The benefits we found from having a centralized repository and how this helped us discover problems in our processes.
- How we organized requirements to fit with the existing application.
- Prioritization of requirements work and where to focus efforts.
- Style. We found that the style appropriate for capturing changes in legal language is different than what worked well to capture enhancements to software behavior and workflow.
- How we successfully propagated change to multiple functional teams.

Biography

Bill Baker is a software development manager at Sage in Beaverton, Oregon. He has been involved in software development, project management and process improvement for a number of years. While at Harland Financial Solutions, Bill led the improvements outlined in this paper.

Bill has a Ph.D. in Electrical Engineering from Washington State University and an excessive collection of other degrees from Washington State University and Michigan State University. Bill can be reached at bill.baker@sage.com.

Todd Gentry has been a software developer for 25 years. He is currently a Senior Manager in Product Development for Harland Financial Solutions, supervising a software development team working in the .NET framework upgrading the technology of a legacy system.

Todd is a Certified Scrum Master. He holds a B.S. in Computer Science and Mathematics from the University of Oregon. Todd can be reached at todd.gentry@harlandfs.com.

Introduction

This case study focuses on process improvements around a legacy application that generates loan documentation for all of the different locales in the U.S. Loan documentation consists of a pool of hundreds of forms. The details of the specific transaction determine which forms apply. The regulatory environment surrounding these forms is diverse and includes federal, state, and local laws and regulations. The location of the transaction determines the specific regulatory context that is applied when generating the language and formatting for the forms for the transaction. The application was first created just after HP introduced the laser printer as an alternative to pre-printed forms. As with many legacy applications there was no real specification for this system. We used the source code as the specification when defending legal challenges against the compliance warrantee that backed the software.

This study describes the challenges we faced and the changes we made while implementing a requirements management system. Several years prior to this, the development organization had addressed delivery problems by setting fixed release dates, reorganizing into a matrix style development organization and breaking work into manageable size projects as compared to one massive project for the whole release.

The paper starts with a description of the situation at the beginning of the study. This provides the context upon which the process improvements were made. In the next section we describe the implementation and discuss the challenges we faced. The challenges are described using a narrative style. Each section is concluded with a take away lesson that we believe can be applied by any organization in a similar situation. We conclude with a section describing areas that need improvement, or where we think we could have done a better job, and a summary.

Situation

This study starts in late 2002. The product we are working on is a Windows application that generates loan documentation using a proprietary rules based system. The text and format in the documents must be compliant with all applicable federal, state, and local banking laws and regulations. At that time, the development group used a matrix organization made up of the following silos:

- Product Management group
- Product Legal group
- Project Management group
- Requirements/Design group
- Software Engineering group
- QA group

Release dates were set, up to a year in advance. We had four releases a year that added enhancements, software upgrades and bug fixes. Two releases focused on major enhancements and two focused just on “compliance” to address minor changes in wording to keep documents in compliance with current laws and regulations. Work was organized into relatively small projects of 2 to 6 months. All projects that were completed by a specified cutoff date were merged, regression tested and released in the next release.

Product managers, in consultation with the product legal group, were responsible for defining the scope of each project. They attempted to group like work together into a single project. For example, a project might be formed around a specific enhancement. The scope would include the enhancement and fixes for bugs that had been reported in that area of functionality. While, in concept, the project was localized around a specific area, this did not always translate well into source code. One problem the development group had was multiple projects in the same release were touching the same area of source code. It was not always clear how to merge the code, resulting in quality problems.

Generally there were four to six projects active at any given time. At the beginning of each project, project managers formed a team from available resources consisting of someone from the requirements/design group, several developers and a QA person. The designer, working with product managers and the legal group, developed a specification for the project. The “requirements” document developed was really a change specification. Design documents were updated to further specify the changes to be made by software engineers.

There was no organized communication between project teams. Specifically, there was no central repository for requirements documentation. There was no mechanism to query what had been done in past projects in some particular area of functionality. Project documentation resided in a project specific folder on a shared file system. Because of this and because we were capturing changes rather than a more general description of desired behavior, we had quality problems which are illustrated in the following scenario.

There is a paragraph of legal language that is shared between the privacy disclosure and the promissory note. The Privacy project had the requirement to make the paragraph all caps. The product was released. Customer calls pointed out that the paragraph on the promissory note should be mixed case, not all caps. A subsequent project changed things to make the paragraph mixed case. The product was released. Customer calls complained that the paragraph on the privacy disclosure was now mixed case and not all caps. At this point a manager who had knowledge of both projects made the connection and corrected the problem so that the paragraph is all caps on the privacy disclosure and mixed case on the promissory note.

The teams generally focused on changes. Because of this narrow focus they did not often step back and see the bigger picture. Team members were aware from the design documentation and the source code that the paragraph was used by multiple documents, but they focused on the requested change, not thinking to ask if the change applied to all documents. This was compounded because of the lack of a central requirements repository where the designer might have seen previous requirements about the case of the paragraph. There was no way to look at a complete set of requirements for any particular design element where the conflict above might have easily been detected. We also did not have any tool support for traceability, so analyzing the impact of making a particular change was generally not carried out because of the difficulty of doing so. There was no way for anyone to assess impact other than to review source code.

The problem in this scenario did not result in breaking the compliance guarantees of our product, but it should be easy to see how something similar could happen that would result in a significant compliance problem.

At this point in time we saw that the solution to this problem was a centralized requirements repository and decided to implement this using a requirements management tool. We put out an RFP and evaluated the responses. We performed an in-house evaluation of the top two contenders which resulted in the final selection and purchase of a tool. The implementation of the tool is where this case study really begins.

Implementation

A fair amount of work needed to be done before we rolled out the new tool. An initial tool configuration had to be designed and processes and procedures developed. A committee of stakeholders was formed to oversee this process. We wanted to create a standard for all products in the group. Besides the product that is the focus of this case study, there are other loan and banking products that could potentially benefit. The product legal group has input to all products, and regulatory developments often impact more than one product. The stakeholders were management representatives from the different matrix disciplines as well as representatives from different product teams. We were aware that a successful implementation depended on buy in from all concerned and made an effort to organize a complete set of stakeholders.

Training and consulting was included in the purchase agreement. A three day implementation session was organized. The first part of the session was training for the stakeholders in the use of the tool. The second part was the initial organization of the data, based on our needs. This discussion was facilitated by the consultant. The implementation session was instrumental in getting buy in from all the stakeholders. The consulting services we purchased were a key to our success.

Much of the discussion during the facilitated implementation session focused on how to leverage the work of the product legal group for use in multiple products. Discussion also focused on different levels of requirements and traceability between them. We came up with a high-level structure that we thought would meet our needs, with the understanding that it would be refined and changed based on experience. This structure, shown in Fig. 1, represents different levels of requirements that we thought would be useful. We initially focused on the product requirements which are equivalent to what Wiegers (Wiegers 1999) calls Software Requirements Specification or SRS, because that is where we thought we would get the most immediate benefit. We left the development of the higher level requirements, on the left, to later.

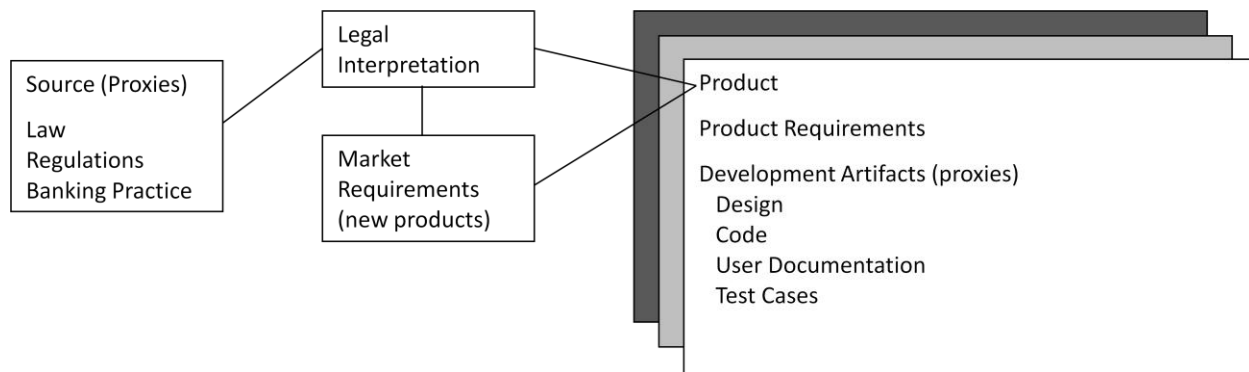


Figure 1. This diagram shows that we intended to have a single set of legal and market requirements that could be used by multiple products. We wanted the legal interpretation to link to laws and regulations.

Even the relatively simple relationships shown in this diagram weren't clear at the start of our discussions. Having someone from outside the organization facilitate discussions helped bring

out internalized knowledge that hadn't been explicitly documented until this point. This greatly helped in organizing our knowledge and was a key in a successful start to this project.

Lesson 1: Outside consulting services are key to a good start

We did the initial database configuration based on the structure that was developed. This was used by a pilot project. The main goal of the pilot project was to define and document how each discipline should use the new tool; in other words we were developing and documenting processes and procedures. The pilot was focused on the department's primary product with the assumption that the processes and procedures developed could be generalized to other product lines.

The first thing that the pilot team needed to do was to work out how the application should be represented in the requirements system. One of the main challenges was that there was no universal view of how the product should be represented. Our challenge was to develop consensus from all the interested parties. There were pros and cons to each alternative, but we did settle on one that was acceptable to all. Our top level structure was general and wasn't much different from what might be used for a brand new application. It is similar to the Software Requirements Specification (SRS) template by Wiegers (Wiegers 1999). The outline is listed below:

1. General Requirements
2. Definitions and Terms
3. Data Requirements
4. Data
5. User Interface Requirements
6. Product Features
7. Form Requirements
8. Documentation Requirements
9. Integration Requirements
10. Testing Requirements

Where things became interesting was when we started putting actual requirements into this structure. The people who were going to write the requirements, as well as the audience who was going to consume them, have deep experience with the existing product. In order for people to find information, the structure needed to easily map to their product knowledge. This differentiates legacy systems from new systems, where everyone is developing domain knowledge as they go and the product is still being defined.

The structure for the requirements that we developed reflects the structure of the product itself. The application builds documents or forms from paragraphs. Paragraphs are architectural elements in the application and can be reused in multiple forms. Rules are also reusable elements. A rule, for example, might be used to determine whether an element should appear in a particular location based on the transaction occurring in a particular geographic state. The same rule could be reused for anything dependent upon that state. We reflected this architecture in our structuring of requirements modules. Modules are high level elements in the tool we purchased and are equivalent to documents. The forms module listed forms and the paragraphs used in each by name. The paragraph names were linked to the section in the paragraphs module that

contained the requirements for that specific paragraph. We linked to rules defined in the rules module in a similar manner. This is illustrated in Fig. 2.

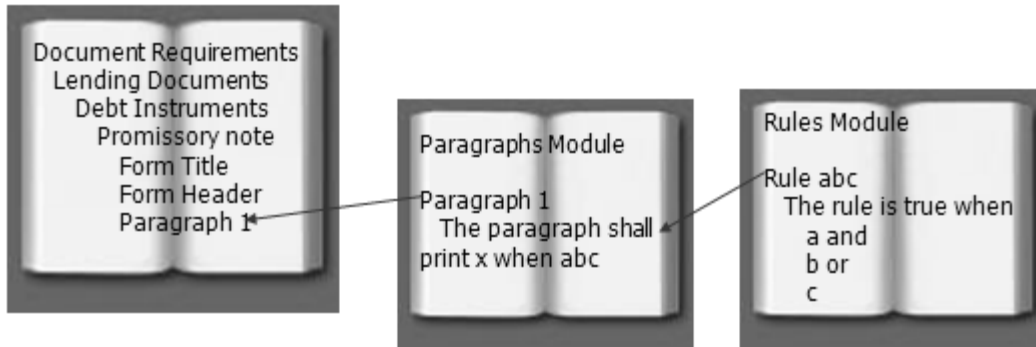


Figure 2. Diagram showing use of trace links to reflect architecture of application in requirements modules.

We made use of trace links in a unique way to model the relationships between architectural elements of the program familiar to everyone who worked with this product. The tool we purchased could display the requirements side-by-side which allowed the user to easily move from a higher level view to a detailed view. This is shown in Fig. 3.

7 Document Requirements	Links to Paragraphs	Links Paragraphs to Rules	Links to Rules
1.1.2.1.5 Iowa Consumer Notice Paragraph			
1.1.2.1.6 Vermont demand language			
1.1.2.1.7 Cosigner Notices Paragraph - Vermont Notice to Cosigner	<p>Paragraphs</p> <p>The notice shall print substantially as follows, once in each available Debt Instrument when the Vermont Cosigner Rule is TRUE.</p> <p>NOTICE TO COSIGNER: YOUR SIGNATURE ON THIS <NOTE WORD> MEANS THAT YOU ARE EQUALLY LIABLE FOR REPAYMENT OF THIS LOAN. IF THE BORROWER DOES NOT PAY, THE LENDER HAS A LEGAL RIGHT TO COLLECT FROM YOU.</p> <p>NOTE: This "notice" is meant to be appear COMPLETELY independent of any other type of cosigner notice.</p>	<p>Rules</p> <p>The Vermont Cosigner Rule shall be TRUE when all of the following conditions have been met:</p> <ul style="list-style-type: none"> The <i>GSL State is Vermont</i> EITHER of the following are TRUE: <ul style="list-style-type: none"> The <i>All Borrowers Will Receive Money, Property or Services from Loan control</i> has NOT been selected and there is more than one <i>Individual Borrower</i> in the transaction. The transaction contains at least one <i>Individual entity whose Capacity is Cosigner</i> 	
1.1.2.1.8 Fed Box for Disclosure Paragraph			
1.1.2.1.9 Lien Granted Pursuant to this Agreement Paragraph			

Figure 3. The left column shows the contents of the Document Requirements module. This module shows the structure of a document, in this case the promissory note, in terms of a list of paragraph names. The paragraph name is linked to a section in the Paragraphs module containing the requirements for the paragraph. The paragraph may link to rules in a similar fashion.

While the structure we developed for our existing application is probably not what we would have developed if we were creating a new product, the structure worked well because it was easily understood by the development staff, who were the primary consumers of this information. Another area where there is a direct mapping between the requirements structure

and the product is in the user interface requirements. The requirements hierarchy followed screens and workflow in the product itself.

Lesson 2: Product requirements reflecting product and software architecture make for easy initial use

There are many definitions of requirements. One definition (Sommerville and Swayer 1997) closely represents our working definition:

Requirements are...a specification of what should be implemented. They are descriptions of how the system should behave, or of a system property or attribute. They may be a constraint on the development process of the system.

We struggled with what should be in the requirements and what should be deferred to a design specification. Wiegers discusses this issue in general (Wiegers 2006). Our solution was to focus on how much we needed to constrain the rest of the development process. For example in some cases product legal wanted specific language under specified conditions. This did not leave the designer any leeway and we deemed this a requirement even though, to some, it seemed this was more appropriate for the design documentation. In other cases product legal would specify that a statement, such as a warning, was necessary and leave the wording up to the designer. In this case the requirement is for a warning, but not for specific wording. For this product, there seems to be a broad gray area between a requirement and design that is determined by the specific context.

This concept extended to the user interface as well. Certain features in the interface turned specific portions of language on or off and needed to be tightly specified in the requirements. In other areas the designer had more flexibility in what they could do. Since our requirements writers came with a background of tightly constraining things, one of the big problems for them was recognizing when this was not needed.

Lesson 3: Use requirements to constrain design only as much as is needed

Starting from no requirements documentation at all for a complex application is daunting. From the beginning there was a question of how much effort to put into filling in context around the requirements we needed to write for a specific element. Wiegers addresses this issue (Wiegers 2006). We strove to write the minimum number of requirements needed to understand what we wanted to do. This really is a question of how much context is needed. We wanted to provide enough context so issues such as the scenario cited above with the security paragraph wouldn't recur. This needed to be balanced against the business need of getting maintenance and new feature work completed.

We used several criteria to determine how much work to do. Bugs are often small and narrowly focused. We didn't see much benefit in creating requirements to capture changes needed to fix just bugs, and decided not to reflect these in our requirements. We did create trace links between any design and code changes and a bug tracking number to have complete traceability back to the reason for a change. On the other hand we found benefit in defining the context around larger changes or enhancements. Spending some effort on defining the context clarified the use case and resulted in a higher quality product because development had a better understanding of the desired product behavior.

Another criterion was the volatility of a particular area or document. We put in significant effort to capture current behavior as context for changes in volatile areas of documents with the anticipation that this work would pay off in the future by providing a base for future requirements work.

Lesson 4: Be pragmatic when filling in descriptions of current behavior

We structured the requirements so that they could easily be navigated by anyone with knowledge of the product. Developers are one of the main audiences for requirements and we addressed some of their needs by developing a style that effectively communicated what they needed to know. Much of the development work in any legacy application is maintenance. Developers are really looking for something that tells them what they need to change. This desire was particularly strong in our case because previous to this project, developers were given change specifications rather than requirements that described desired behavior. We developed a requirements style that communicated this change information.

To start we captured a complete set of requirements for the current product for a given feature. The version was labeled after these requirements had been reviewed and approved as accurately describing the current system. Next the requirements writer would modify the requirements to reflect the new desired behavior. We used a “change type” property to record the type of change to the requirements (modified, new, delete, or no change to existing behavior). After being reviewed and approved this version was also labeled. We developed some custom scripts that would compare the two labeled versions and create a redlined version that showed the differences. The redlined version and the change type designations proved very effective in communicating the desired changes to development staff.

The precursor to this process was the capture of a fairly complete description of the existing system. This provided a good contextual base for everyone. Modifying this base to describe the new desired behavior was effective in communicating the requirements to product management, the other main audience for our requirements. Having a complete description in a single location made review much easier than inferring behavior from partial requirements and other sources such as bug descriptions or the product itself.

Lesson 5: Find ways to effectively communicate to particular audiences

The product legal team has a big part to play in determining behavior for the product. But their training is in evaluating and writing legal documents, not in writing requirements. Requirements written in a standard style don't provide the information in a form that is easy to review by this audience. Yet, because of the regulatory nature of the product we need legal input, especially for areas of some documents that depend upon a wide range of law and regulations. Writing requirements based on first principles would have had a high cost but would have provided little real business benefit. One technique we worked out was to write a small program that produced multiple versions of the same paragraph or document. This program iterated over the parameters that involved the legal or regulatory changes for a particular paragraph and printed a version for each different set of parameters. Pseudo code for this program is shown below.

```

FOR each value of parameter P1
  FOR each value of parameter P2
    ...
    FOR each value of parameter Pn
      Generate paragraph (P1, P2, ... Pn)
      Print paragraph
    END FOR
  ...
END FOR
END FOR

```

The number of unique paragraph versions could potentially range over a hundred depending on the number of parameters of interest and the number of values for each parameter. The output was passed to the legal team, who redlined the text. The redlined text became the requirements covering the particular changes in law or regulations for the paragraph. Although this is unconventional, it provided both an efficient and effective means of specifying language changes for the legal team, the developers who implemented the changes, and for the testers.

Lesson 6: Be creative in how requirements are generated and expressed.

We knew going into this that we needed to get everyone involved. The stakeholders group was formed with this idea in mind. We knew that without buy in by everyone in this group the implementation would not go well. As with any organization, there were organizational changes. In our case this occurred during our pilot. A new person became the manager for the requirement/design group, a key group in the adoption of the new tool. This person had not been part of the original stakeholder team. Specifically, this person had not been part of the tool evaluation and early planning. He was initially skeptical of the procedures we were developing. The pilot project timeline was extended so that we could address his issues and bring him onboard. The result was that we got buy in from this person, which greatly helped with implementation in the requirements writing team.

Lesson 7: Take time to bring people up to speed.

During the initial facilitated implementation sessions, we spent a fair amount of time defining traceability relationships between requirements modules and other development artifacts such as design and code. Creating the individual trace links fell to the development staff and became part of the updated development process. At the end of each project an audit is done to make sure that all requirements were completed in terms of updated design, source code, and tests. We also check to make sure that all source code changes can be traced back to a requirement or a bug report. This change in software development process has increased quality because we are now sure we have tested all changes. It also makes sure that we get everything we need to into release notes sent to customers. Prior to this project, developers would sometimes make undocumented changes, i.e. “fix bugs”, that they discovered during development. These changes didn’t always get properly documented and resulted in support calls from customers wondering about some unexplained change they had observed.

Lesson 8: Use traceability to improve software development processes

While we could have made a number of the changes without using a requirements management tool, purchasing and implementing a new tool did have one big benefit. It helped make change palatable to the staff. We’ve already discussed how we centralized and structured our

requirements. We had a lot of discussions around style and whether we should express something as a requirement or part of the design.

One of the biggest changes was moving from writing change specifications to expressing desired behavior. Not only did this improve our requirements writing for new features and enhancements, but it unlocked creativity from the whole team. It encouraged everyone to think about the best way to do something as compared to the mindset we had before that was narrowly focused on the change to be made.

Everyone recognized that they would have to change how they worked. This provided an opportunity to make more sweeping changes than we might otherwise have done without the advent of a new tool. Implementing traceability is one example. We will cite two more examples.

The first was an organization change away from a matrix structure to fixed teams. Fixed development teams were formed that were made up of several software engineers and a QA person. We also formed a separate release team that was dedicated to merging projects and producing final releases. One benefit was that team members developed better working relationships because they were together for much longer periods of time. The churn at the beginning of a new project was significantly reduced because team members already knew each other and had established working relationships.

A second example was a move to writing high-level test cases early in the development cycle, concurrent with the writing of requirements. This second view of desired behavior has become an effective means of communicating desired changes. It also removes the burden on the requirements of being the sole means of communicating the context of the changes.

The changes to processes and procedures we made as part of the implementation of the requirements management tool were significant. As a result, the organization as a whole became more accepting of change in general. One of the more significant changes that occurred well after the adoption of the requirements management system was a change to a more agile development method where we have a single backlog list from which the teams pull. This has resulted in greater development productivity. The adoption of Agile techniques was part of ongoing change that started with the implementation of the requirements management tool rather than as a separate initiative.

Lesson 9: Leverage tool adoption to make other changes

Areas Needing Improvement

While our requirements management and software engineering processes have improved greatly, there are areas that still need improvement. The first area we will focus on is impact and legal analysis. The second area is use by the legal team and teams for other product lines.

Going into this project one of the things we wanted to accomplish was the ability to easily do an impact analysis. We wanted the ability to quickly assess how big an impact a given change might require. We recognized that this required a critical mass of requirements which would take some time to amass. We have achieved only limited success with this. We have achieved a critical mass of requirements in volatile areas of the program, but these are fairly limited compared to the entire scope of the product.

When customers challenge our legal content, we have to research the legal reason for a particular piece of language or why a particular change was made. The people who do this still use the same tools as before; they look up the piece of language in the source code and use comments there to find related bug reports or call records. Traceability in the requirements management tool could be used to do this, but it hasn't been adopted. The ability to do this kind of research has improved because of the use of traceability implemented using the requirements management tool though. It is much more likely that code comments will include the appropriate reference ids now, than in the past, because of the use of trace links by the software engineers.

Lesson 10: Don't have high expectations for achieving impact analysis.

The pilot and initial implementation focused on the product requirements. A lot of effort was put into working out all the details of how different roles within the development organization would use the system. Initially we focused on the requirements/design team. The issues here were detailed organization, style, how much context to fill in, etc. Another big area had to do with the details of traceability and how to trace between various pieces of information. All this effort resulted in a system that was usable by the development group.

Much less effort was put into bringing the product and legal teams into the system. After the initial implementation effort, Bill, the lead architect, moved on to another project. This resulted in both of these teams not having full time requirements and tool support. They essentially had to fend for themselves in terms of figuring out how to do requirements in their group and how to best use the new tool. This was the case for other product lines as well. The result was that they continued to do things as they had in the past and never made the jump to effectively using the new tool or any possible improvements it might have provided.

Using the tool was particularly problematical for the legal team. The tool we selected represents requirements in a hierarchy much like you would in a regular written document. This didn't fit well with how the legal team worked with things. They work in three dimensions: legal concept such as indebtedness, legal form such as the note or collateral disclosure, and jurisdiction, such as a state. In a hierarchy, one of the dimensions is chosen as the top level or primary, and the other dimensions become sub-hierarchies under each section of the primary. The product requirements were structured with documents as the primary dimension. Jurisdiction is used to turn on or off specific paragraphs or other units of language. Given this organization it is almost impossible to determine whether some definition of indebtedness is applied consistently across the other dimensions.

We really have two problems here. One is that the tool does not support a multi-dimensional view where each dimension has equal weighting, or at a minimum can easily be queried with parameters from each dimension. The other issue is that the legal team really needs a different structure from what was used for the product requirements. For example, they would prefer to create a general definition for indebtedness, then describe jurisdictional level differences to the main definition and finally apply those to specific documents. This requires a mapping between the two structures.

Both of these are significant problems and would require time and effort to work through. Unfortunately a resource knowledgeable about requirements writing and use of the new tool was not available. The result was that the hoped for synergies between legal requirements and

product requirements expressed in figure 1 have not occurred. We suggest that a permanent position that can help teams with adoption and use would have been very useful in this case.

Adoption by several other legacy product lines was mostly unsuccessful for similar reasons. One new product did make successful use of it though. This was primarily because there was a member of that team who was experienced in developing requirements. The longer term implementation across the whole business unit would have benefited from someone who was experienced in requirements development and had a vision for the overall structure of all the requirements and how they should be represented in the tool, as well as detailed knowledge of the tool itself.

Lesson 11: Need “requirements architect” position

Summary

We have described the process improvements made during the adoption of a new requirements management tool. Specifically we have focused on the challenges of writing requirements for a legacy system that didn't previously have them. The lessons we have generalized from this experience are restated below:

- Outside consulting services are key to a good start
- Reflecting product and software architecture in the requirements makes for easy initial use.
- Use requirements to constrain design only as much as needed
- Be pragmatic when filling in descriptions of current behavior
- Find ways to effectively communicate to particular audiences
- Be creative in how requirements are generated and expressed
- Take time to bring new people up to speed
- Use traceability to improve software development processes
- Leverage tool adoption to make other changes
- Don't have high expectations for doing impact analysis
- Staff a “requirements architect” position

As we look back over the changes that have occurred we see that we are a much improved software development organization.

References

Karl Wiegers 1999, *Software Requirements*, Microsoft Press.

Karl Wiegers 2006, *More About Software Requirements: Thorny Issues and Practical Advice*, Microsoft Press.

Ian Sommerville and Pete Sawyer 1997, *Requirements Engineering: A Good Practice Guide*. John Wiley & Sons.