Improving Software API Usability through Text Analysis: A Case Study

Robert B. Watson
Microsoft Corporation
bob.watson@microsoft.com

Abstract

Technical writers who want to be more involved in the earlier stages of software product development must often find ways to demonstrate how their skills can benefit the initial design process. This case study describes how the application of technical communication skills and tools helped improve the usability and clarity of a new application program interface (API) by performing a text analysis of the API elements. The case study presents the theory upon which this approach is grounded and describes how the theory was applied to analyze a specific API. The paper concludes with a review of how this analysis method could be transferred to other projects and how the tools used in this analysis can be applied to benefit the design, development, and documentation processes of APIs. Keywords: API usability, text analysis, document design, single-source, content management system.

Introduction

In 1996, Johndan Johnson-Eilola [1] observed that, "The support model of technical communication encourages corporations to view technical communication as something to be added on to a primary product." Today, 13 years later, technical communication is still viewed by many to be a support or peripheral role; however, the case study in this paper describes how I expanded my role as a technical communicator to contribute to the product design. This case study presents a tangible example of how to apply the tactics for rearticulating technical communication that Johnson-Eilola describes in [1].

In the worst-case, support-model writing scenario, the writing process often begins in the later phases of the product development cycle. In my experience as a technical writer, inconsistencies in the design can complicate the job of writing clear and easy-to-read documentation. Identifying and removing these inconsistencies earlier in the product development cycle benefits everyone—especially the customer. While technical writers can help, they are not always part of the design process and to get closer to the early stages of the product design process might require that technical writers demonstrate how their skills can contribute to the initial design. Finding the opportunity and the means by which to demonstrate their benefit can be difficult and doing so might require some experimentation, as Johnson-Eilola suggests in [1]. Unfortunately, a product development team that subscribes to the support model of technical communication might not recognize the benefit of involving a technical writer early in the design process, placing the burden of demonstration on the technical writer. Although Beck [2], for example, suggests having technical writers on extreme programming (XP) teams, if technical writers are not present during the early stages of design, it can be difficult for them to demonstrate how their skills can help.

The method described in this case can help technical writers who are already on design teams as well as those coming in to a project later in the development process. In this case study, I applied the method after the API had been designed but before it had been fully implemented. Whether early or late in the design process, the method used in this case study can be used by a technical writer to apply his or her role as a user advocate. It applies the core technical communication skills that [1] describes to help improve API design and enables the technical writer to present this benefit in a way that the product development team can appreciate.

Literature Review

This section describes the previous work on which the method used in this case study is based and starts by recognizing that technical writers can provide more than a just a support role to the product development process. This is followed by the usability, design, and documentation principles that form the theoretical foundation of this method.

Relocating Technical Writer Value

Johnson-Eilola [1] suggests that technical writers have many skills that can aid design and refers to Robert Reich's list of the four key areas to use for reinventing technical communication education [3]. While Reich's list was intended to improve education, I found that the areas he lists—collaboration, experimentation, abstraction, and system—could also be applied successfully in the workplace.

For experimentation, Johnson Eilola suggests in [1] that, "Technical communicators must continue to investigate broader forms of usability studies." He concludes the section on experimentation with, "Insisting on broader

978-1-4244-4358-1/09/$25.00 ©2009 IEEE

forms of usability serves a double purpose for technical communicators: it helps us produce documentation and assistance more attuned to a user's broader needs, and it also shifts the focus of value away from a discrete technology and toward communication and learning."

Usability Principles

Arnold [4] suggests that we consider programmers as humans whose interface to a computer is through an API. He suggests that it would be reasonable to apply human factors principles to the study of APIs. Henning [5] presents the consequences of poorly designed APIs in terms of reduced reliability of the final product and reduced productivity of the programmer using the API. Both of these consequences can be measured by technical support call volume. Clearly unreliable products will generate support calls from end-users, and APIs that are difficult to use or understand are likely to generate more technical support calls from programmers than those that are easily understood. Each support call represents a cost to the customer who cannot accomplish a task and a cost to the vendor who must pay the technical support engineer who answers the call.

API usability can be measured during the design and observed throughout the development of an API. Green [6] describes 12 dimensions for measuring API usability, one of which is *consistency*. Aligning the terminology used in the API to ensure it is consistent reduces distractions, which helps to reduce the cognitive load [7] required to learn a new API. Aligning the terminology also makes it possible for a programmer to feel more confident when they transfer what they learn in one part of the API to another. Clarke [8] describes how these 12 dimensions can be (and, in fact, were) applied in a practical setting and used to evaluate the usability of Microsoft's .NET Framework [9]. McLellan [10] also describes API usability factors that can be studied.

Design Principles

Computer programming and API design have been around long enough to establish design principles that can be used for heuristic evaluation. One such reference is [11]. Chapter 2 in [11] is titled "Framework Design Fundamentals," for example, and contains many guidelines for software developers to use when designing new APIs for Microsoft's .NET Framework. While the guidelines presented in [11] are written for the .NET Framework, many can be applied directly or, with some adaptation, to other programming languages and environments. To varying degrees, other languages have similar guidelines documents such as [12] for the C language and [13] for Java. Many organizations have also established coding conventions and standards.

One section in chapter 2 of [11] is titled "The Principle of Self-Documenting Object Models." It describes how API documentation starts with the API itself when it says, "it is very important to design APIs that do not require that developers read documentation every time they want to perform a simple task." While the documentation of the API might start with the API, it does not end there. Cwalina adds in [11], "The documentation remains critically important for many users who do take the time to understand the overall design of the framework up front. For those users, informative, concise, and complete documentation is as crucial as self-explanatory object models." Chapter 2 of [11] continues to describe how to name the methods and parameters, and follows that in Chapter 3, which is dedicated completely to the subject of names and naming.

When studying design principles to apply to this analysis, it is important to distinguish between design principles for software reuse and design principles of API usability [11]. Software reuse principles are more appropriate for designing the internal implementation than the exposed API. The principles and practices described for each have different goals and they can appear to conflict with one another if applied in the wrong context. In this study, only the external interface of the API is studied, not the internal implementation.

Documentation Principles

There seems to be no disagreement that programmers like (or expect) to learn an API by using it [11][14]. Paul Vick in [11] says, "Documentation can help give the initial idea of what's supposed to happen, but we all know that you never really learn how something works until you get down into it and start fiddling around, trying to do something useful." My own experience as a software engineer is echoed in [11] and [14], which describe how programmers first try to learn how an API should work from the API or from similar APIs they already understand. They check the documentation to correct their assumptions only when the method or function does not work as they expected. This is also consistent with [11], which says that the programmers expect the API to tell them what they need to know so they do not have to look it up. Thinking of the API as a form of documentation is also considered in [11], which suggests making "discussion about identifier naming choices a significant part of specification reviews" and that user education—that is, the technical writers—be involved early in the API design process.

While the API itself is the first source of documentation that a programmer encounters, [14] reinforces the need for effective reference documentation when it says that by the time a programmer asks for help, they have already been unsuccessful with many other sources of documentation. When programmers are unfortunate enough to get to that point, according to the study con-

ducted in [14], "99% are mad if they have to ask for help."

Looking further into the documentation process, improving consistency also makes the API more usable for programmers who speak other languages. International programmers might lack the cultural or linguistic context to associate several terms to a single context even when such an association might appear obvious to a native speaker. Using different terms for the same concept also increases localization costs, and the resulting documentation might not be as clear if the translator cannot understand why different terms are used for a single concept.

Case Study

The analysis I applied in this study started with the goal of learning the API by evaluating its usability from a documentation perspective. The API design and documentation principles informed the heuristic I used for the study. While the design principles and usability dimensions describe more than how an API looks and behaves, in this study, I limited the scope to only what the programmers see when they use the API.

Starting with the 12 dimensions for measuring usability, *consistency* was the only one I was in a position to apply in this study. While all 12 dimensions are worthwhile to consider when designing a new API, much of the design had already been accomplished by the time I became involved. Consistency, however, is certainly a core skill of a technical communicator, even if it is a skill [1] attributes to the support model.

Figure 1 illustrates the analysis process I developed for this study. The steps in the process are described in the following text and represent the specific analysis steps I derived from the principles described in the literature review.

Data Preparation

The goal of this step is to identify and extract all the API elements of interest for study; however, the source code files of the API contained over 15,000 lines of program code and comments. To analyze the API systematically, I had to convert the content from the source code to a more manageable format. To do this, I broke the API into the elements listed in Table 1 because they are the first API components that programmers see when they use the API in their programming.

This API had about 800 interfaces, methods, structures, and enumerators and over 900 individual parameters to document and study. To document the API, I had to enter the API components into our XML-based content management system. Once entered, extracting them from the content management system required only a small additional effort.

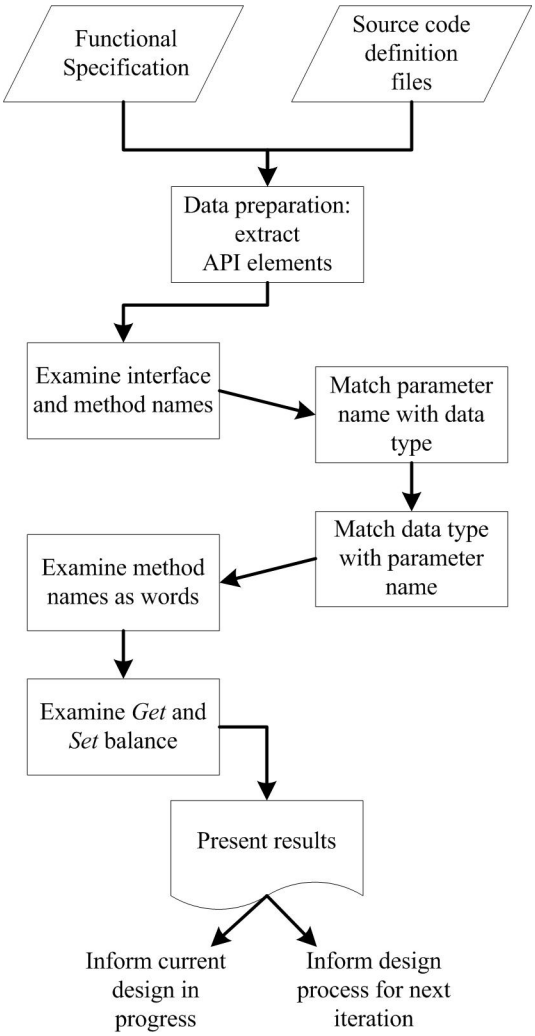

Figure 1. API analysis process

I formatted the API elements as the XML data elements listed in Table 1. The resulting XML data file contained one record for each parameter and one additional record for each method. To analyze the API, I imported the XML data that contained the API elements as a spreadsheet into Microsoft Excel 2007. Microsoft Excel 2007 provides the ability to sort and filter the data in an ad hoc fashion making it well suited for my analysis.

In some sense, this project was a "best-case" scenario to study because the API was defined in an interface definition language (IDL) file, which contains a structured description of the interfaces, methods, and parameters of the API. API elements are easier to parse and extract from a structured file than from an unstructured file.

Many of the methods in this API are Get/Set methods, which get or set the value of an object's property. To place those methods together in an alphabetic sort, I created a field that contains a normalized version of the method name. In this field, the "Get" and "Set" substrings of the method name are moved from the beginning of the

method name to the end. For example, a method named *GetThisProperty* would be stored in the normalized field as *ThisProperty (Get)*. When the data are sorted on this field, *SetThisProperty* will follow *GetThisProperty* immediately as *ThisProperty (Get)* and *ThisProperty (Set)* for easy visual comparison.

Table 1. API analysis data elements.

Element	Element Description
Interface name	Name of the interface.
Method name	Name of method in the interface.
Normalized method name	For Set/Get methods, the name is reformatted to move the Set/Get substring to the end of the string. All other names are unchanged.
Parameter number	Index of parameter: 1 for first parameter, 2 for second, etc. 0 is used to identify method name records.
Parameter name	Name of parameter. One parameter per data record. Methods with more than one parameter have more than one record in file.
Parameter data type	Data type of parameter.

Tool support can also make the process of extracting the elements easier. In this case, I used a proprietary tool to parse the API components of the IDL file into an XML file defined by a proprietary schema. Doxygen [15] is a publicly available tool that can also be used to extract the API elements from definition and header files to an XML file. Once the elements are in an XML file, an XSL transform can transform the elements into another schema.

API Analysis

The following sections describe the different analyses that I performed on the API element data after it was collected in the format described in the previous section.

Interface and Method Name Analysis

I used an ad hoc heuristic based on our programming standard and the naming and design guidelines described in [6] and [11] to examine the interface and method names used in the API. Because I studied the API after the design was essentially complete, I was not in a position to recommend any major changes to the organization of the API; however, I could use my findings to inform how I would later document the API.

In this study, I observed that some interfaces were logically related by how they were described in the specification, but this relationship was not apparent from the interface names. For example, some interfaces were subordinate, hierarchically, to others, but the interface names did not reflect that relationship as [11] suggests. The names, however, were consistent with what they represented so they would make sense to programmers once they understood the nature of the API. By recognizing this early, I would be sure to describe this relationship when I wrote the documentation.

Parameter Names and Data Types

In the next analysis steps, I studied the parameter names and data types. The API in this study was designed for C++, which is a statically typed language so the data type tells the programmer a lot about a parameter's nature and use. In a statically typed language, a parameter's data type can be as informative as the name, if not more so. If there is a conflict between the parameter name and the data type, the data type is more credible because the parameter name is only significant to the programmer whereas the data type is significant to both the programmer and the compiler.

For this part of the analysis, I performed several tests and asked, "Did the usage implied by the data type agree with that implied by the name," and "did the usage implied by the parameter name agree with that implied by the data type?" Any apparent conflicts might confuse the programmer, be difficult to explain, or both.

Parameter Name Matches Data Type

I sorted the API elements by the parameter data type field and then, for each parameter, I compared the parameter data type to the parameter name, checking to see if the usage described by the parameter name agreed with the usage implied by the data type. For generic data types, such as LONG or DWORD, I checked to see if a more specific data type might better inform the usage. There are, however, many times when a generic data type is expected. For example, when the parameter refers to a count or an index value, a more specific data type would be less usable in a programming context. I noted the cases where the usage was confusing or not obvious so I could clarify them with the development team.

With the API elements sorted by parameter data type, it was easy to spot inconsistencies that ultimately might confuse, if only momentarily, any programmers using the API. For example, if a data type is used by a parameter in 10 different methods and in 9, the parameters have the same name, a programmer is likely to assume that the reason the tenth parameter has a different name is deliberate and significant. The programmer is likely to try to find the reason for the difference, even of no reason exists. In my role as an advocate for the API user, I investigated such cases with the development team to either correct the different name or to explain why it was different in the documentation. In a large API that is being developed by many software developers, it is easy for minor naming variations to creep in. Applying my tools creatively gave

me a unique view of the API definition—one which made it easy to catch these inconsistencies quickly and early enough to correct.

Data Type Matches Parameter Name

This analysis step is very similar to the previous one and sorts the API elements by the parameter name field to compare the data types of similarly named parameters. In the same way that similarly typed parameters should be checked to see if they are similarly named, similarly named parameters should most likely, but not always, be similarly typed. As in the previous step, the naming inconsistencies that were found were resolved with the development team.

Method Name as a Word

Chapter 3 of [11] describes characteristics for method and parameter name construction that can easily be checked without an in-depth knowledge of the programming language. For example, [11] suggests that methods should describe the task they perform, not how they are implemented, method names that return Boolean values should be in active voice instead of passive voice, and parameter names should reflect their meaning or use and not their data type. Chapter 3 of [11] also suggests that methods that retrieve or return Boolean parameters should work grammatically with "if" in front of it. Likewise, the method should also sound reasonable when negated. For example, consider the following program code examples:

```
if (Valid(value)) ...
if !(Valid(value)) ...
```

These would be read as, "if valid value" and "if not valid value," which sound reasonable when read aloud. The following code examples are functionally the same as the previous examples but sound more awkward when read aloud.

```
if (IsValid(value)) ...
if !(IsValid(value)) ...
```

The first example in this set would be read as, "if is valid value," which, while not as good as the previous set, is not too bad. The negated version, "if not is valid value," is much more awkward. How this analysis is applied to a specific API depends on the programming and the spoken languages being analyzed.

Get and Set Balance

Get and Set (or get and put) methods are another group that can cause confusion when inconsistently named. For properties that can be written as well as read, the method names should match after the first three letters. For example, if the API has a *GetPrompt* method and the Prompt property can also be set, the set method for this property should be *SetPrompt*.

The parameter name and data type of related set and get methods should match within the constraints of the language. This balance and consistency make it easier for a programmer to discover both methods when both exist and not waste time searching for another method should the property support only one of the two operations (that is, only Get or only Set).

Presentation of Results

I started this project as a way to become familiar with a new API and to develop my documentation plan. Sorting and filtering the API elements made it easy to spot the inconsistencies that I would probably find later as I was writing the documentation (or maybe not), and I could find them much earlier in the process. By identifying problems early, I was able to work with the development team to either correct them in the design or schedule time for a more detailed explanation in the help. Fortunately, I found the inconsistencies early enough in the process for the development team to fix them.

Sorting the parameters by data type, for example, makes naming inconsistencies very obvious. To the developers, the inconsistent names might be physically or functionally distant in their view of the API, such as in the 10,000-line header file, the specification document, or the many different source code files in the source code library. In a sorted list of parameter names, however, the names line up one after another and the discrepancies speak for themselves. The development team's desire for quality and usability was evidenced by the relatively few errors I uncovered by this study and the team's responsiveness to my findings. When I presented the development team with the relatively few inconsistencies that I found, they were very concerned and quickly corrected or explained them.

Discussion

One could argue that it might not be worth the effort to analyze an API in such an extensive manner if only a few inconsistencies were found. In fact, the high degree of overall consistency makes it even more important to perform this analysis because programmers will assume, in a generally consistent API, that any inconsistencies they find are intentional and meaningful. The programmers will then waste time trying to find and understand the (nonexistent) meaning behind the inconsistency. The technical writer, and advocate for the programmer who will be using the API, can use the methods presented in this study to find and eliminate this type of confusion. The alternative is for the programmer to find them and have their programming interrupted while they spend time

and trying to find and understand an explanation for the apparent inconsistency.

This analysis provides a unique view of the API that makes it easy to find inconsistencies quickly and early enough in the design process to correct (if unintentional) or better understand (if intentional) them. The tools and skills of the technical communicator are uniquely suited to this type of analysis and can provide direct and tangible benefit to the design as well as to the documentation.

Reflecting on the process, it followed closely the tactics described in [1]: collaboration, experimentation, abstraction, and system. This was a new process, informed by the usability, design, and documentation principles outlined in the literature review and expands the boundaries of usability as [1] described under *experimentation*. The case study was the *collaboration* between the development team and the technical writer, which was facilitated by the results of the analysis. The analysis relied on an *abstraction* of the API elements by taking them out of their normal context as programming elements and considering them as text elements. Finally, the analysis takes place in a system perspective that considers the impact an API as a source of documentation as opposed to simply programming elements that exist apart from their documentation.

While it would have been beneficial to be involved earlier in the design process, this case study occurred under very favorable conditions. I had early access to the specifications and the IDL files of the source code, a supportive product development team to work with, and reasonable tools support. The absence of any one of these would have complicated the process.

The content management system in which I author API documentation stores the content according to an XML schema. As part of the normal documentation process, I must code the API elements as elements in the XML schema. Some of this process can be automated, which lowers the initial cost of data entry, but I have to enter the content into the format required by the content management system whether I want to perform this analysis or not. Once the data has been entered to produce the documentation, converting the API elements from a proprietary XML schema to the format described in the Data Preparation section above is a matter of writing and applying an XSL transform on the data in the content management system. If our help content was based on a different storage model—for example, if the help content was stored as HTML elements—extracting the API elements might be more difficult. However, even if it requires additional work at the beginning—for example in the worst-case scenario where the API elements must be entered into a spreadsheet by hand—it might still be worth the effort because of the unique view of the API it affords. Clearly, an automated solution that is integrated with the source code and documentation systems would make the formatting and analysis process more efficient.

With less than 1,000 parameters, this process was manageable as a single spreadsheet. I am not sure how far this process would scale up before becoming unwieldy. At some point, however, the analysis might need to be subdivided or automated.

Future Work

The heuristic I applied to the text analysis was informed by the literature but was still ad hoc in this case. The analysis process could benefit from additional research into formalizing the heuristic and then adopting it as part of the coding and design standards for an API. With that in place, software developers and technical communicators could share the same rules for the design and evaluation of an API.

Going forward, I'd like to continue relocating the value of technical communication closer to the beginning of the design; realizing, of course, that the value of each step must be demonstrated and proven along the way. Along those lines, I am currently working with the development team on ways to repurpose our API reference documentation further so that it can be used as source code comments and as the foundation of the final functional specification. This can save the development team time in that I have to keep the reference documentation up-to-date in the normal course of my job as a technical writer. When the development team spends time updating the specifications during development or after the product ships, they use time they could spend developing and debugging the current version or designing the next version. Repurposing single-sourced content can eliminate duplicated effort.

It follows that if I can produce the content for a functional specification from the help content after the product is complete; it is a short step to using the help content to produce some of the original specification at the beginning of the product development cycle. Having single-sourced reference content means the basic reference topics are finished when the functional specification is finished. Single sourcing the reference topics in this manner also minimizes overall documentation overhead and developer workload.

The completed reference topics could also be used to conduct usability tests on the API earlier in the product design cycle by using the API reference topics as a type of paper prototype of a new API [11]. Having the reference documentation available early in the design process would make it possible to conduct usability tests sooner, possibly before any code has been written. This would make it possible to find usability problems while there was still time to address them in the design. Having API reference topics finished before the code also leaves more time for the technical writers to work on the program-code examples and sample programs, which are what customers really want to see in the help [14].

Acknowledgments

I want to thank Professors Mark Zachry and Jennifer Turns of the University of Washington's Human Centered Design and Engineering department for their support in this project and the development of this paper. I would also like to recognize the product development team with whom I worked on this project. Without their ongoing support, this project would not have been possible. Finally, I would like to thank my immediate manager whose encouragement provides a supportive environment in which to explore innovative approaches to advancing technical communication.

References

[1] Johnson-Eilola, J., Relocating the Value of Work: Technical Communication in a Post-Industrial Age, *Technical Communication Quarterly*, Summer 1996, 245-270.

[2] Beck, Kent (2007). *Extreme Programming Explained, Second Edition.* Addison Wesley, Upper Saddle River, NJ.

[3] Reich, Robert B. (1991) *The Work of Nations: Preparing Ourselves for 21st-century Capitalism*. Alfred A. Knopf. New York, NY. Cited in [1].

[4] Arnold, K. (2005). Programmers are people, too. *Queue*, 3(5), 54-59. doi: 10.1145/1071713.1071731.

[5] Henning, M. (2007). API design matters. *Queue*, 5(4), 24-36. doi: 10.1145/1255421.1255422. Also published in (2009) *Communications of the ACM*, Vol. 52 No. 5, Pages 46-56. doi: 10.1145/1506409.1506424

[6] Green, T., Blandford, A., Church, L., Roast, C., & Clarke, S. (2006). Cognitive dimensions: Achievements, new directions, and open questions. *Journal of Visual Languages & Computing*, 17(4), 328-365. doi: 10.1016/j.jvlc.2006.04.004.

[7] Mayer, Richard E. & Moreno, Roxana (2003). Nine Ways to Reduce Cognitive Load in Multimedia Learning. *Educational Psychologist*, 38 (1), 43-52. Retrieved March 21, 2009, from http://www.informaworld.com.offcampus.lib.washington.edu/10.1207/S15326985EP3801_6

[8] Clarke, S. (2004). Measuring API usability [Electronic Version]. *Dr. Dobb's Journal Windows/.NET Supplement*, May 2004, S6-S9. Retrieved April 26, 2008 from http://www.ddj.com/windows/184405654

[9] Clarke, S (2005, March 29). *HOWTO: Design and run an API usability study*. Retrieved April 26, 2008 from http://blogs.msdn.com/stevencl/archive/2005/03/29/403436.aspx

[10] McLellan, S., Roesler, A., Tempest, J., & Spinuzzi, C. (1998). Building more usable APIs. *Software, IEEE*, 15(3), 78-86. doi: 10.1109/52.676963.

[11] Cwalina, K., Abrams, B. (2008) *Framework Design Guidelines: Conventions, Idioms, and Patterns for Reusable .NET Libraries (Second Edition)*. Addison-Wesley Professional, Indianapolis, IN.

[12] Maguire, S. (1993) *Writing Solid Code*. Microsoft Press, Redmond, WA.

[13] Tulach, J. (2008). *How to design a (module) API*. NetBeans, Retrieved April 26, 2008, from http://openide.netbeans.org/tutorial/apidesign.html

[14] Nykaza, J., Messinger, R., Boehme, F., Norman, C. L., Mace, M., & Gordon, M. (2002). What programmers really want: results of a needs assessment for SDK documentation. In *Proceedings of the 20th annual international conference on computer documentation* (pp. 133-141). Toronto, Ontario, Canada: ACM. Retrieved March 21, 2009, from http://portal.acm.org.offcampus.lib.washington.edu/citation.cfm?id=584955.584976&coll=ACM&dl=ACM&CFID=27598684&CFTOKEN=34478726.

[15] Heesh, Dimitri van (2009). *Doxygen source code documentation generator tool*, Retrieved March 26, 2009, from http://www.stack.nl/~dimitri/doxygen/

About the Author

I began writing documentation for software developers as a programmer/writer after 17 years of writing software for software developers as a software engineer. After a few years of writing, I returned to the University of Washington and graduated with a Master of Science in Human Centered Design and Engineering with a focus on User-Centered Design. Attending the university while working as a programmer/writer allowed me to build a mutually beneficial bridge between industry and academia—one that I hope to continue to develop.