IEEE Software

How API Documentation Fails

Gias Uddin and Martin P. Robillard
McGill University

Volume 32, Issue 4
July/August 2015

DOI: 10.1109/MS.2014.80

© 2015 IEEE. Personal use of this material is permitted. Permission from IEEE must be obtained for all other uses, in any current or future media, including reprinting/republishing this material for advertising or promotional purposes, creating new collective works, for resale or redistribution to servers or lists, or reuse of any copyrighted component of this work in other works.

How API Documentation Fails

Gias Uddin and Martin P. Robillard, *McGill University*

Researchers investigated how 10 common documentation problems manifested themselves in practice. The three severest problems were ambiguity, incompleteness, and incorrectness of content. The surveyed practitioners considered six of the problems "blockers" that forced them to use another API.

Most software development employs libraries and frameworks whose functionality is made available through APIs. Like other sophisticated technical tools, APIs must be properly learned and correctly operated to be effective. To achieve this goal, APIs typically come with documentation.

A piece of API documentation is a product in itself, whose quality can vary. In previous research on API usability, we learned that API documentation can be critical for software developers.[1] Because good documentation can help developers work efficiently, it can even serve to promote the API.[2] In contrast, documentation that doesn't meet its readers' expectations can lead to frustration, major loss of time,[1] and even abandonment of the API.

Although providing high-quality API documentation is obviously desirable in absolute terms, creating and maintaining it is costly, and predicting the payoff is difficult. For this reason, it can sometimes be difficult to justify allocating resources to improve API documentation.

To provide an evidence-based foundation for planning API documentation efforts, we conducted two surveys of API documentation quality with a total of 323 IBM software professionals. The first survey let us catalog and understand how 10 common documentation problems manifested themselves in practice. The second survey assessed those problems' frequency and severity. We concluded that the most pressing problems were related to content, as opposed to presentation.

The Exploratory Survey

The first survey explored developers' experiences with documentation. We aimed to collect both good and bad examples of API documentation. The survey contained five questions: two for demographics and three related to API documentation (see Table 1).

Table 1. The exploratory survey questions.*

Question	Type of response
1. What is the last development task you completed that required you to consult API documentation?	The task description and URL
2. Give up to three examples of API documentation that you found useful. Explain why the documentation was useful.	The documentation unit's URL and an explanation of why the documentation unit was useful.
3. Give up to three examples of API documentation that you did not find useful. Explain why the documentation was not useful.	The documentation unit's URL and an explanation of why the unit wasn't useful

* We also asked two questions related to respondent demographics.

We selected the participants from the employees at IBM Canada's Ottawa and Toronto labs. We sent the survey invitation to 698 employees whose job role contained the word "software" and one or more of the words "engineering," "development," "architect," "testing," and "consultant." Sixty-nine employees responded (9.9 percent): 47 developers, 15 architects, three consultants, three managers, and one tester. The respondents' experience ranged from 1 to 40 years (average 13.1, standard deviation 10.8). The consultants were engaged in proof-of-concept prototype development. The tester consulted API documentation to write test cases. Two of the managers were product managers. In this article, we refer to each respondent using $R_{i:j}$, where *i* denotes the survey and *j* denotes the respondent.

Each of the 69 responses contained a reference to or description of a development task; 20 of them also contained a URL for the task in an issue-tracking repository. This let us study the responses in the context of the task, the related code changes (when we had an access to the repository), and the API documentation units (DUs) the developers consulted to complete the task. A DU is an explicitly designated block of information in the documentation that's related either to an API element (such as the documentation for a method) or to the API itself (such as an overview page).

The respondents provided 179 examples of good or bad documentation for 131 documentation units that documented API elements in a total of 72 distinct APIs from six types of programming languages (see Figure 1). There's no standard definition for an API: APIs can be software libraries, framework components, or even Web APIs. In our analysis, we considered an API to roughly correspond to a coherent set of elements that could be distributed together, such as JAR (Java archive) files in the Maven repository. Fourteen respondents provided examples from publicly available corporate APIs, three from private corporate APIs, and the rest from open source APIs. One of us had access to all the referenced corporate APIs in the context of an IBM research internship.

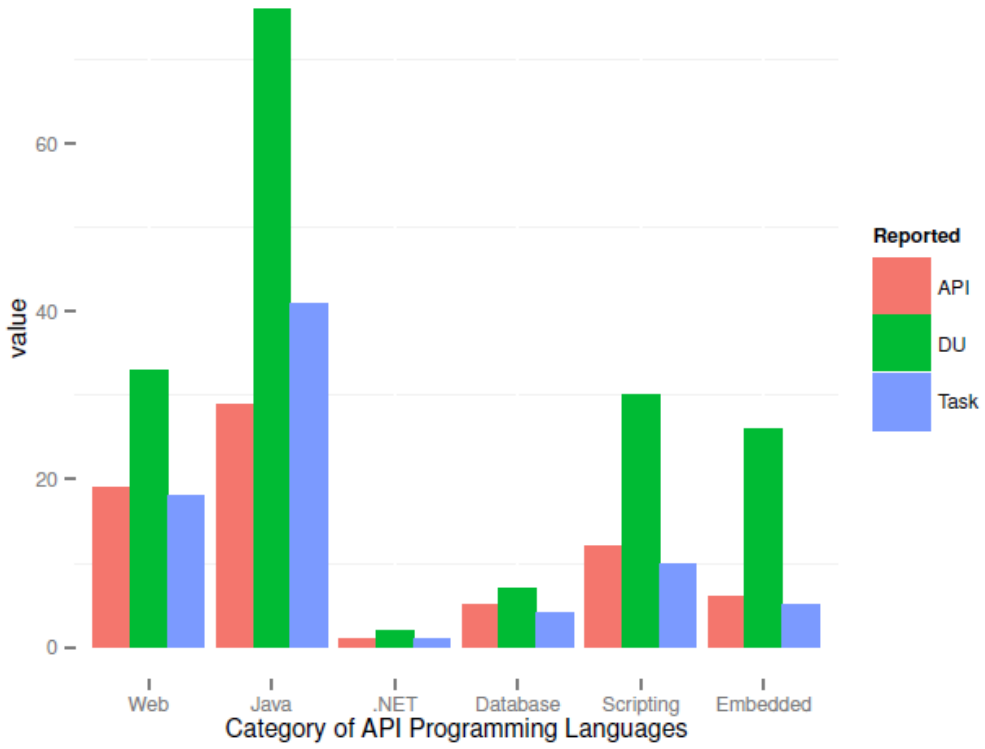

Figure 1. The APIs reported in the exploratory survey. A DU (documentation unit) is an explicitly designated block of information in the documentation that's related either to an API element (such as the documentation for a method)

or to the API itself (such as an overview page).

Figure 1 shows the number of APIs the respondents mentioned, organized by the type of programming language. Of the 131 API elements whose documentation was discussed, 44 percent (58) were Java-based and 29 percent were Web-based and were mostly for JavaScript. Embedded APIs were mostly for C/C++, and scripting APIs were mostly for Python.

Data Analysis

From the answers to questions 2 and 3 in Table 1, we collected 179 documentation examples: 90 examples of good documentation and 89 examples of bad documentation. After inspection, we realized we could learn little from the examples of good documentation. Most of these were justified as generally fulfilling the developer's information needs. So, we focused on learning about documentation quality by studying the documentation failures.

Employing a card-sorting approach, we grouped the examples of bad documentation according to the nature of the problem. We took into account the respondent's comment and the DU provided in the URL (when applicable). We discarded six examples as invalid because they contained comments too generic to qualify, such as "All API docs help more or less."

From the remaining 83 examples, we discarded another four because we couldn't confidently interpret the respondent's answer. This phase resulted in the mapping of the 10 common types of problems, illustrated by 79 comments describing concrete examples of inadequate documentation. Five examples were linked to two or more problems; the remaining 74 were linked to a single problem.

Documentation Problems

Table 2 lists the 10 types of problems, which fell into the categories of content and presentation.

Table 2. API documentation problems reported in the exploratory survey.

Category	Problem	Description	*E**	*D**
Content	Incompleteness	The description of an API element or topic wasn't where it was expected to be.	20	20
	Ambiguity	The description of an API element was mostly complete but unclear.	16	15
	Unexplained examples	A code example was insufficiently explained.	10	8
	Obsoleteness	The documentation on a topic referred to a previous version of the API.	6	6
	Inconsistency	The documentation of elements meant to be combined didn't agree.	5	4
	Incorrectness	Some information was incorrect.	4	4
Total			61	57
Presentation	Bloat	The description of an API element or topic was verbose or excessively extensive.	12	11
	Fragmentation	The information related to an element or topic was fragmented or scattered over too many pages or sections.	5	5
	Excess structural information	The description of an element contained redundant information about the element's syntax or structure, which could be easily obtained through modern IDEs.	4	3
	Tangled information	The description of an API element or topic was tangled with information the respondent didn't need.	4	3
Total			25	22

* *E* is the number of examples that mentioned a problem; *D* is the number of developers who reported a problem.

Content. The first type of content problem was incompleteness. In the trivial case, documentation can be incomplete owing to the lack of effort invested in it:

> *I have had to deal with lots of auto-generated documentation, with no documentation at all for many classes or methods. (respondent $R_{2:3}$)*

However, even when earnest effort is invested in documenting an API, it can be difficult to anticipate all the ways it can be used. When an API element might be involved in complex interactions with other API elements, understanding how to use it might require more than the description of its functionality.

For example, the Java Ehcache API offers functionality to set up distributed caching across nodes. The interaction among distributed nodes must be established on the basis of the Java RMI (remote method invocation) API. Respondent $R_{1:29}$ experimented with the Ehcache API to set up a Java-based distributed-caching system. Although he found the summary overview of the API useful, he was frustrated when the documentation didn't adequately discuss how the API communicates with the RMI API:

> *The experiment involved extending the existing RMI replication mechanism, and the javadoc didn't go into sufficient detail as to when or how the various methods would be called.*

The second content problem was ambiguity. In this case, the documentation appears to cover a topic of interest but leaves out important details so that multiple interpretations are possible. In a way, ambiguity is a form of incompleteness, but where the missing information is of a specific, clarifying nature. Nevertheless, the respondents reacted differently to each type of problem. In the case of incompleteness, they mentioned missing information; in the case of ambiguity, they mentioned confusion or incomprehension.

For example, the ASM 3.2 API is used to manipulate Java bytecode. It contains the method getItem(int item) in the class ClassReader. The getItem javadoc states, "Returns the start index of the constant pool item in b, plus one." In Java, a constant pool is a table of structures storing the identifiers of types, variables, and string constants. So, the start index can point to a type (for example, String) or its constants or identifiers. $R_{1:1}$ commented on an ambiguity related to this API:

> *Documentation for ClassReader did not adequately describe the getItem() method used to index into a constant pool. Specifically, it wasn't clear whether the return value was the index of the constant-pool entry type tag or the start of the data for the constant-pool entry.*

The third content problem was unexplained examples. Although developers generally appreciate code examples, some respondents were frustrated when an example didn't have an adequate explanation.

The Dojo JavaScript APIs are used to develop Ajax-based applications. A respondent liked that many methods in the APIs were documented with code examples but got frustrated when having trouble determining how to configure an input in the example. Dojo, being a JavaScript API, needs documentation that explains the code examples in terms of how the code can be deployed in a browser environment. This requires the configuration of input objects for each documented code example in a method. The Dojo documentation doesn't always discuss the configuration objects before a code example:

> *Dojo's APIs ... should specify the configuration object that each method takes in detail (for example, the methods or strings that the object can contain), because simply saying that it takes an object is not usable. ($R_{1:22}$)*

Indeed, as Seyed Nasehi and his colleagues reported in their study of the code examples posted on the Stack Overflow website (http://stackoverflow.com), "explanations accompanying an example are as important as the examples themselves."[3]

The fourth content problem was obsoleteness. When APIs go through rapid development, their documentation can quickly become outdated,[4] so the current documentation often doesn't reflect recent changes.

For example, the Apache HBase API is used to host distributed Hadoop databases for big data. The API goes through frequent version changes, often changing or removing elements. For instance, the ClusterStatus class had a method Collection<ServerName> getServers() in version 0.90x, which changed to int getServers() in 0.92. Also, Collection<ServerName> getServerInfo() changed to Collection<HServerInfo> getServerInfo(). $R_{1:5}$ commented:

Even though the method names are similar, they have different functionality in 0.92x.

It isn't reasonable to expect an API to have documentation of every change. However, significant functionality changes related to an element require feedback in the form of documentation or something to overcome the information mismatch between versions. This is especially true when the version change is expected to be backward compatible.[5]

The fifth content problem was inconsistency. Software development can be incremental, with multiple products often combining into a new product. These intermediate products can be developed by more than one development team, leading to inconsistent documents with insufficient explanation on how to support the interoperability between the intermediate products. $R_{1:45}$ commented:

I'm looking at ways to improve integration between different products. … One approach would be to use the different product APIs to construct new "glue" type applications as part of our portfolio. … The problem is that there is no consistency across the different products, … not in documentation …. Obviously that does not promote interoperability between products.

The final content problem was incorrectness. An API element's description can be incorrect. For example, the input parameters and output type of an API method as implemented in the source code can differ from those in the documentation. To do a routine update, a reported API offered scheduling functionality using a set of Unix-like triggers:

There is a Cron trigger with wrong information: It says the "schedule" parameter must be a string with the same format as a Unix crontab file; however, it does not implement everything that the Unix crontab format provides. … Inaccurate documentation (it says prev_trigered_time is available, but it was not) …. ($R_{1:36}$)

Presentation. The first type of presentation problem was bloat. The risk with associating large chunks of text to a specific element or topic is that readers won't readily be able to determine whether the text provides the information they seek, especially when the title or header is general. For example, $R_{1:31}$ was trying to understand some of the advantages of using a cluster installation:

This introductory section is so bloated that it was hard to understand why it has so much text at all, when the introduction gives neither an overview nor the purpose of clustering.

The second presentation problem was fragmentation. When the respondents had to click through multiple pages of an API document to learn the functionality and use of an API element, they found the separation of the descriptions at such a micro level to be unnecessary (as Martin Robillard and Robert DeLine also observed[1]). For example, unlike other APIs, Dojo APIs adhere to a two-layer description format of each function. The first page (called the "API document") provides an overview of the functionality with a code example. The second page (called the "usage document") describes the related parameters—for example, the underlying environment or server. $R_{1:22}$ commented:

This information (functionality description and usage configuration) should be part of the API, instead of separated into two documents: API and usage.

The respondents had difficulty navigating through the myriad pages in an API document to find information:

Fragmented documentation I find really difficult to use, where you have to have 10s of clicks through links to find the information you need, and page after page to read. ($R_{2:69}$)

The third presentation problem was excessive structural information. The description of a type (class or interface) in object-oriented APIs normally includes structural relations with other types of the API. For example, all the classes in the Java SE (Standard Edition) APIs are subclasses of the class java.lang.Object. Respondents indicated that excessive structural information, which often can be readily obtained on demand from the IDE, was an obstacle to focusing on the important content:

Again too much information on how the class is related to other classes; I don't need docs for this. ($R_{1:11}$)

The final presentation problem was tangled information. Respondents considered the explanation of APIs with specific usage scenarios helpful, but not when multiple usage scenarios were tangled with each other in one description. $R_{1:9}$ was unhappy with the documentation of the JSONObject class of the json.org API:

> *The class says that for get and put, you can cast to a type, which is fine, but the next line dives to a different topic of type coercion that I did not need.*

The Validation Survey

We conducted the second survey with a different group of participants at IBM. The questionnaire had seven questions (see Table 3). For questions 3, 4, and 6, the respondents were to provide a response for each of the 10 types of problems.

Table 3. The validation survey questions.*

Question	Type of response
1. Do you consult API documentation as part of your development tasks?	Yes/No
2. Approximately how many hours on average per week do you spend consulting API documentation?	Number (decimals okay)
3. Based on your experience of the last three months, how frequently did you observe the documentation problems?*	Never / Once or twice / Occasionally / Frequently / No Opinion
4. How severe was the documentation problem to complete your development task, when you last observed it?*	Not a Problem / Moderate (kind of irritating) / Severe (I wasted a lot of time on this but figured it out) / Blocker (I could not get past it and picked another API) / No Opinion
5. Based on your experience of the last three months, list the three APIs whose documentation caused you the most severe problems, as reported in the survey.	The API's name and the documentation problem
6. Given a strict budget to solve the 10 documentation problems, which three should be prioritized?*	A numeric value (3 = Top, 2 = Fair, 1 = Low, 0 = No Priority)
7. Any comment about your experience with the problems?	Comments

* For questions 3, 4, and 6, the respondents were to provide a response for each of the 10 types of problems.

We sent the survey invitation to the software developers and architects at IBM Canada and Great Britain. We knew that 65 of the 69 respondents in the first survey were software developers or architects. So, we sent the survey invitation only if the employee's job role contained the phrase "software developer," "software engineer," or "software architect." The target population thus included 1,064 people (610 from Canada and 454 from Great Britain). We received 254 responses (23.8 percent). Among the respondents, 203 (80 percent) were developers (114 from Canada and 89 from Great Britain). Among the 51 architects (20 percent), 28 were from Canada and 23 from Great Britain. The respondents' experience ranged from one to 45 years (average 13.17, standard deviation 9.9).

We analyzed three aspects of the responses: the problems' frequency, their severity, and the necessity to solve them (questions 3, 4, and 6 in Table 3). Figure 2 plots the analysis results.

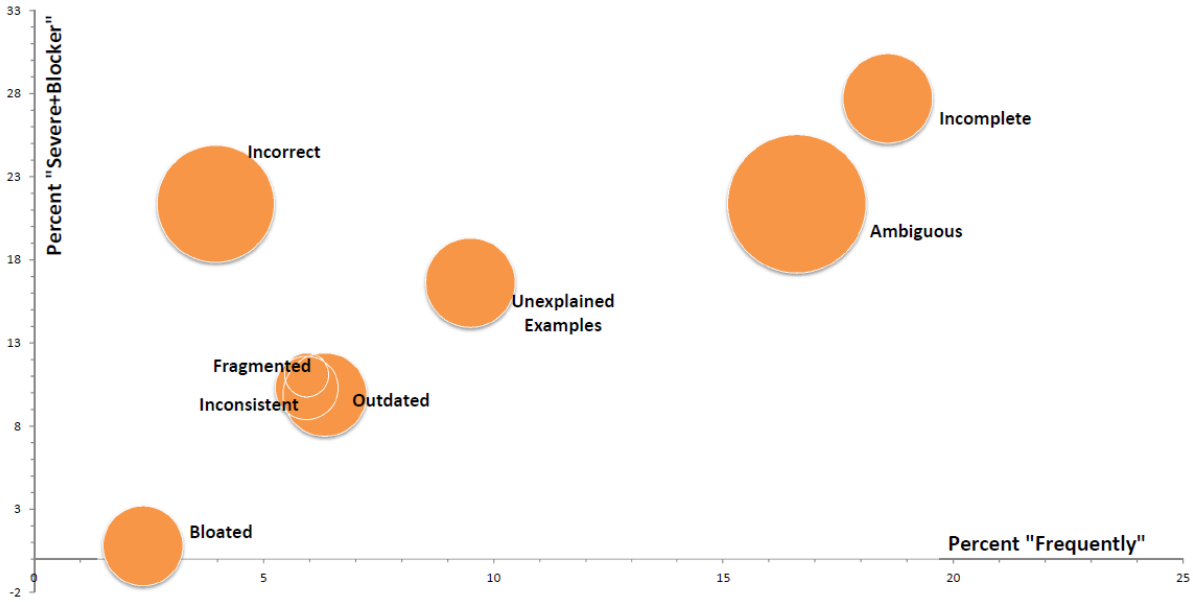

Figure 2. The results of our analysis of the problems' frequency, their severity, and the necessity to solve them. A circle's size indicates the percentage of respondents who gave that problem top priority. "Tangled information" and "Excessive structural information" are absent because no one selected the two problems as the top problems.

For the frequency scale, for each documentation problem, we computed the ratio of the response "Frequently" to all responses (the *x*-axis in Figure 2). For example, regarding whether the respondents had observed ambiguity during the past three months, 42 answered "Frequently," 68 answered "Occasionally," 38 answered "Once or Twice," and 10 answered "Never." Ninety-six respondents answered "No Opinion." Following Christian Bird and Thomas Zimmermann's strategy,[6] we didn't include the number of times respondents answered "No Opinion" when we calculated the frequency ratio, leading to a 26.5 percent ratio.

We constructed the severity scale by discussing it with three respondents from the first survey. For that scale, we computed the ratio of the combined number of "Severe" and "Blocker" responses to all responses (the *y*-axis in Figure 2), again excluding "No Opinion"

Regarding the necessity of solving the problems, the respondents chose the three problems that most needed attention, using an ordinal scale of "Top", "Fair", and "Low". // Martin: I don't think we refer to these numbers anywhere, so we can omit them// Each respondent chose at least one problem as worth solving. In Figure 2, a circle's size indicates the percentage of respondents who ranked that problem as "Top." "Tangled information" and "Excessive structural information" are absent because no one selected the two problems as the top problems.) For example, 54 respondents ranked ambiguity as "Top," 18 ranked it as "Fair," 28 ranked it as "Low," and four ranked it as "No Priority." Thus, respondents chose ambiguity as the top priority 51.9 percent of the time.

Respondents ranked six problems as "Blocker" at least once: incompleteness, ambiguity, obsoleteness, incorrectness, inconsistency, and unexplained examples. Ambiguity and incompleteness, identified as the most critical problems in Figure 4, were two of the top three priorities for improving documentation.

An interesting observation from the budget priority question (question 6 in Figure 3) is the perceived importance of fixing incorrect content. The respondents infrequently observed incorrect content, but they judged instances as "Severe" or "Blocker." This shows that for some API projects, it might make more sense to address problems according to their severity rather than their combined frequency and severity. In any case, the analysis clearly confirmed that incompleteness, incorrectness, and ambiguity were most in need of attention.

Comparing the results for the 50 percent least and most experienced respondents in the second survey, we found no statistically significant evidence that novices experienced documentation problems differently.

The major finding of our surveys is that the respondents cared most about quality content. The most frequent and common problems had to do with content, and the respondents prioritized addressing five content-related problems over any presentation problem. Perhaps unsurprisingly, the biggest problems with API documentation were also the ones requiring the most technical expertise to solve. Completing, clarifying, and correcting documentation require deep, authoritative knowledge of the API's implementation. This makes accomplishing these tasks difficult for nondevelopers or recent contributors to a project.

So, how can we improve API documentation if the only people who can accomplish this task are too busy to do it or are working on tasks that have been given a higher priority? One potential way forward is to develop recommendation systems that can reduce as much of the administrative overhead of documentation writing as possible, letting experts focus exclusively on the value-producing part of the task. As Barthélémy Dagenais and Martin Robillard discovered, a main challenge for evolving API documentation is identifying where a document needs to be updated.[2] We're exploring ways to automatically provide adaptation recommendations based on an analysis of the source code linked to a manual. Other avenues could include leveraging the crowd with tools such as StackDoc (https://github.com/alnorth/stackdoc), which links Stack Overflow posts with related reference documentation. Although such solutions aren't authoritative, they might constitute an acceptable expedient in cases of rapid evolution or depleted development resources.

Acknowledgments

We thank Barthélémy Dagenais for comments on this article. IBM and NECSIS (Network for the Engineering of Complex Software-Intensive Systems for Automotive Systems) funded this research.

References

1. M.P. Robillard and R. DeLine, "A Field Study of API Learning Obstacles," *Empirical Software Eng.*, vol. 16, no. 6, 2011, pp. 703–732.
2. B. Dagenais and M.P. Robillard, "Creating and Evolving Developer Documentation: Understanding the Decisions of Open Source Contributors," *Proc. 18th Int'l Symp. Foundations of Software Eng.*, 2010, pp. 127–136.
3. S.M. Nasehi et al., "What Makes a Good Code Example? A Study of Programming Q&A in StackOverflow," *Proc. 28th IEEE Int'l Conf. Software Maintenance*, 2012, pp. 25–34.
4. T.C. Lethbridge, J. Singer, and A. Forward, "How Software Engineers Use Documentation: The State of the Practice," *IEEE Software*, vol. 20, no. 6, 2003, pp. 35–39.
5. "APR's Version Numbering," Apache, 2013; http://apr.apache.org/versioning.html.
6. C. Bird and T. Zimmermann, "Assessing the Value of Branches with What-If Analysis," *Proc. 20th Int'l Symp. Foundations of Software Eng.*, 2012, article 45.

Gias Uddin *is a PhD student at McGill University's School of Computer Science. His research focuses on analysis of API artifacts and support forums to facilitate the improvement of API formal documentation. Uddin received a master's in computer engineering from Queen's University. Contact him at giasu@cs.mcgill.ca.*

Martin P. Robillard *is an associate professor at McGill University's School of Computer Science. His research focuses on API usability, information discovery, and knowledge management in software engineering. Robillard received a PhD in computer science from the University of British Columbia. Contact him at martin@cs.mcgill.ca.*