What Makes APIs Hard to Learn? Answers from Developers

Martin P. Robillard, *McGill University*

A study of obstacles that professional Microsoft developers faced when learning to use APIs uncovered challenges and resulting implications for API users and designers.

Most software projects reuse components exposed through APIs. In fact, current-day software development technologies are becoming inseparable from the large APIs they provide. To name two prominent examples, both the Java Software Development Kit and the .NET framework ship with APIs comprising thousands of classes supporting tasks that range from reading files to managing complex process workflows.

An API is the interface to implemented functionality that developers can access to perform various tasks. APIs support code reuse, provide high-level abstractions that facilitate programming tasks, and help unify the programming experience (for example, by providing a uniform way to interact with list structures). However, APIs have grown very large and diverse, which has prompted some to question their usability.[1] It would be a pity if the difficulty of using APIs would nullify the productivity gains they offer. To ensure that this doesn't happen, we need to know what makes APIs hard to learn.

Common sense indicates that an API's structure can impact its usability (see the "API Usability" sidebar).[2] This intuition is reflected by efforts to flesh out sound design principles for APIs and empirical studies on the impact of design structure on API usability.[3–5] However, APIs don't exist in isolation, and other factors can also affect how developers experience them. So, what exactly does make an API hard to learn?

To answer this question, I investigated the obstacles professional developers at Microsoft faced when learning how to use APIs. As opposed to previous API usability studies that focused on specific design aspects, I used an approach completely grounded in developers' experience. By surveying and interviewing developers about the obstacles they faced learning APIs, I discovered many issues that complement those mentioned in API design textbooks and articles. In particular, I found that API learning resources are critically important when considering obstacles to learning the API, and as worthy of attention as the structural aspects of the API. I also elicited specific relationships between resources and API usage that API designers and documentation writers shouldn't overlook when designing API documentation. First, information about the high-level design of the API is necessary to help developers choose among alternative ways to use the API, to structure their code accordingly, and to use the API as efficiently as possible. Second, code examples can become more of a hindrance than a benefit when there's a mismatch between the tacit purpose of the example and the goal of the example user. Finally, some design decisions can influence the behavior of the API in subtle ways that confuse developers.

Survey Design

In February and March 2009, I conducted a survey to gather information about developers' experiences learning APIs. Specifically, I sought to identify areas of concern and themes worthy

API Usability

This article focuses on the obstacles to learning an API. Although *learnability* is only one dimension of usability, there's a clear relationship between the two, in that difficult-to-use APIs are likely to be difficult to learn as well. Many API usability studies focus on situations where developers are learning to use an API. www.apiusability.org provides an extensive list of resources on API usability, including additional references to studies not mentioned here due to space limitations.

of detailed investigation (as opposed to producing generalizable descriptive statistics). To ensure that I didn't bias the results with preconceptions about obstacles I thought developers would face, I left the main questions open-ended. The survey consisted of 13 questions, with the initial three assessing the respondent's professional experience. To focus the survey's answers and get the developers thinking about specifics, the remainder asked them to comment on their most recent learning experiences with a publicly released API.

The survey's core consisted of a three-part, open-ended question on the obstacles developers faced learning APIs:

> *What obstacles made it difficult for you to learn the API? Obstacles can have to do with the API itself, with your background, with learning resources, etc. List the three most important obstacles, in order of importance (1 being the biggest obstacle). Please be more specific than the general categories mentioned here.*

I formatted this question as three comment boxes. Respondents could fill in any number of boxes, from none to all three. In addition, the survey asked for information intended to help interpret the respondent's answers and discover his or her API learning strategies. I gathered context through seven questions on the specific API learned, the time frame, familiarity with the application domain, and so on. Three questions on learning strategies asked how developers approached learning the API and were formatted in the same style as the obstacle question.

The survey concluded by asking for additional comments and if respondents would be willing to participate in future research.

Population and Sample

The survey targeted software developers at Microsoft. Microsoft's software development staff consists of roughly 30,000 engineers, mostly developers, testers, and program managers. For the purpose of the survey, I considered all employees whose title implies software development as developers, but excluded testing engineers due to the specialized nature of their work.

Because the survey also served to recruit participants for in-person interviews, the sampling frame[6] I used was the list of all Microsoft developers working at Microsoft's Redmond, Wash., campus. This sampling frame includes many thousands of professional developers. From this pool, I randomly selected 1,000 and sent them a link to the survey. Targeted developers had two weeks to complete it.

Survey Respondents

A total of 83 developers answered the survey. However, three respondents didn't provide answers to the six open questions on strategies and obstacles, so I discarded their responses. Despite the response rate of 8 percent, the set of respondents constituted an excellent representation of the target population, cutting across job titles, seniority levels, and technology use.

Figure 1 shows the distribution of respondents across job titles and the corresponding distribution of job titles across the population (Redmond developers). The four leftmost bar groups represent four seniority levels for software development engineers (SDEs). Each bar group represents a distinct job title with different associated responsibilities. The following four bar groups represent job titles in development management (lead SDEs). Although technically a management position, lead SDEs are typically involved in active software development along with their team. The next two bar groups represent seniority levels for architects, a multidisciplinary role involving both program management and software development. Finally, the population also included a small fraction of developers with other titles (typically specialists in areas such as security and user experience), but I didn't get respondents from this pool. Among general job categories, titles run left to right, from junior to senior.

As the figure shows, respondents' distribution across job titles in the sample closely maps that of the population. Across all job titles, respondents had on average 12.9 years of professional experience (self-reported). The median was 10 years, and 90 percent reported four of more years of professional experience. The respondents also reported on their aggregated experience learning 54 distinct APIs covering a wide span of technologies, abstraction levels, and application domains. Examples of

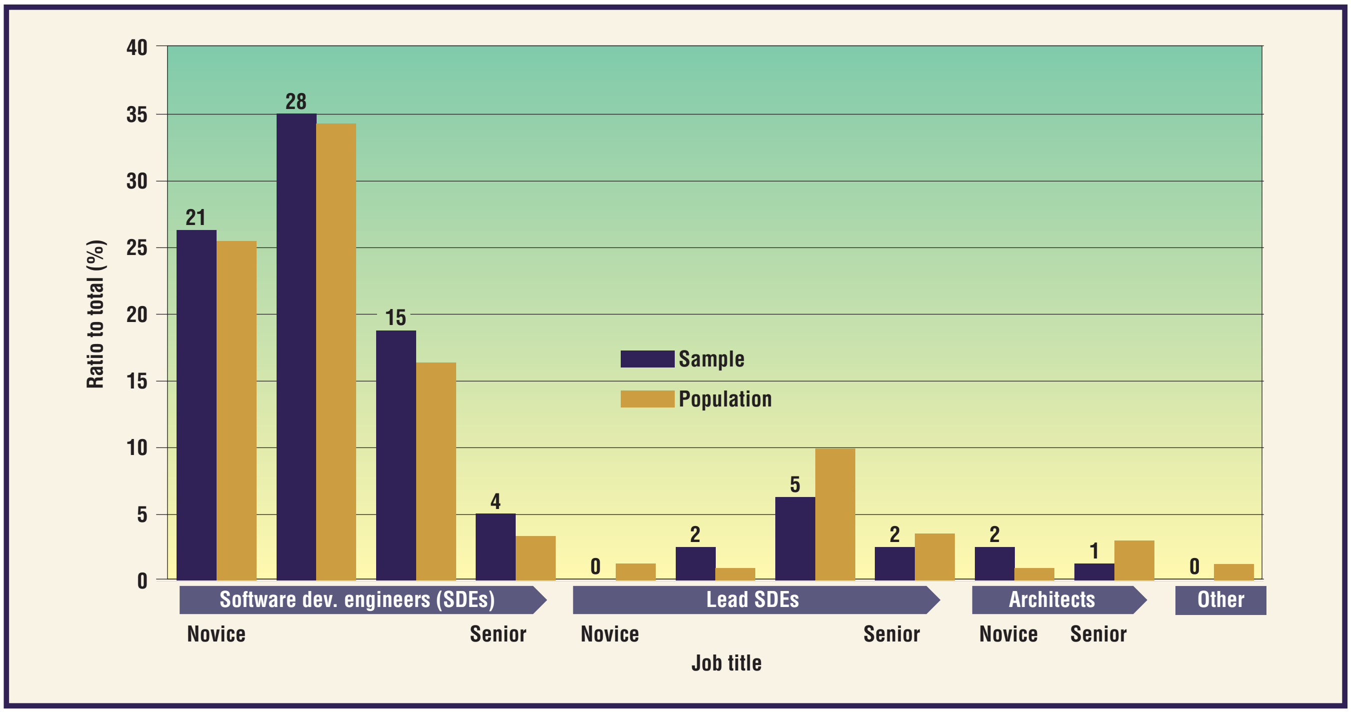

Figure 1. Distribution of respondents across job titles. Most respondents (49 percent) were developers with junior or intermediate job titles. The distribution of respondents across job titles closely maps that of the population.

APIs the respondents reported learning included one that provides access to personal information manager data on Windows mobile-based devices, classic windowing APIs, and Microsoft's most recent Web application development platform.

Survey Results

I analyzed the data by identifying major categories of responses and labeling each response by category and subcategory. Because no clear trend emerged from the ranking of response data for both strategies and obstacles, I ignored this facet of the survey.

Responses on learning strategies yielded few surprises. Of the 80 respondents, 78 percent indicated they learned APIs by reading documentation, 55 percent used code examples, 34 percent experimented with APIs, 30 percent read articles, and 29 percent asked colleagues. Lower-frequency items included a wide variety of other strategies, such as reading books or tracing code through a debugger. In addition to providing a general understanding of how developers approach APIs,[7] this part of the survey, along with the responses to the general "comments" question, yielded few opportunities for deeper analysis. The rest of this article is therefore concerned with the obstacles developers faced when learning APIs.

I derived five major categories from the responses to the "obstacles" question (see Table 1). For each category, the table provides a description and the number of associated respondents. A total of 74 respondents mentioned at least one obstacle. I associated respondents with categories when they indicated at least one obstacle in that category. For example, 50 (out of 74) respondents mentioned at least one obstacle relating to API resources. Some responses pertained to multiple categories.

A major result of the survey is that resources topped the list of obstacles to learning APIs. This is a good reminder that efforts to improve the usability of an API's structure[2–4] need to be complemented by efforts to improve the resources available to learn them.

The refined categorization of the resource-related responses elicited six categories with at least eight associated respondents (listed under "Resources" in Table 1).

Except for general gripes about the official documentation released with the API ("General"), this classification reveals the variety of challenges facing the personnel in charge of producing resources for APIs. Namely, to mitigate obstacles, API documentation must

- include good examples,
- be complete,
- support many complex usage scenarios,
- be conveniently organized, and
- include relevant design elements.

A refined categorization of the structure-related responses elicited two subcategories with at least eight associated respondents (listed

Table 1

Response categories for API learning obstacles

Main category	Subcategories/descriptions		Associated respondents
Resources	Obstacles caused by inadequate or absent resources for learning the API (for example, documentation)		50
	Examples	Insufficient or inadequate examples	20
	General	Unspecified issues with the documentation	14
	Content	A specific piece of content is missing or inadequately presented in the documentation (for example, information about all exceptions raised)	12
	Task	No reference on how to use the API to accomplish a specific task	9
	Format	Resources aren't available in the desired format	8
	Design	Insufficient or inadequate documentation on the high-level aspects of the API such as design or rationale	8
Structure	Obstacles related to the structure or design of the API		36
	Design	Issues with the API's structural design	20
	Testing and debugging	Issues related to the API's testing, debugging, and runtime behavior	10
Background	Obstacles caused by the respondent's background and prior experience		17
Technical environment	Obstacles caused by the technical environment in which the API is used (for example, heterogeneous system, hardware)		15
Process	Obstacles related to process issues (for example, time, interruptions)		13

under "Structure" in Table 1). These subcategories confirm the generally held view that an API's basic design does impact its users. However, the responses also bring to light that an API's testability and the ease of reasoning about its runtime behavior also have an important impact. For example, a respondent indicated being hindered due to a "subtle difference in behavior of APIs depending on context."

Hearing from Respondents

To understand more deeply how developers deal with new APIs in their daily work, I conducted a series of 12 interviews with Microsoft software developers recruited from survey respondents and personal contacts. The goal was to get a detailed picture of important obstacles developers faced when learning new APIs, the context in which these obstacles occurred, and infer possible causes for these obstacles. For this reason, I again chose an open-ended, loosely structured style of *qualitative interview*,[8] which consisted of asking participants to summarize their work with the API and explain the obstacles they faced. Interviews lasted between 15 and 45 minutes and were audio-recorded.

Although I conducted this study with Microsoft developers who might not be representative of all API users, the results should be usable by others working with APIs. Indeed, the main lessons derived from the survey and interviews don't concern the frequency or predominance of specific trends, but a detailed interpretation of how different situations played out in practice and the lessons we can derive from them.

Emerging Questions

From considering the aggregated survey results, reading the individual responses, and studying the interview transcripts, several important themes emerged. In choosing material for this article, I favored themes that went beyond well-known issues, as opposed to a systematic coverage of the concerns mentioned by respondents (survey) and participants (interviews). In doing so, I left out interactions that described valid but well-known issues. For example, one participant described the difficulty of choosing the right function among alternatives in the Win32 API. The participant referred to the practice of adding functionality to an API without breaking backward compatibility by introducing "extension methods" with the "Ex" suffix:

> *There is a function CreateWindow, and a function CreateWindowEx. Ex creates some new types of windows, which weren't created in the earlier functions of the API. So they are growing the [set of] functions, but sometimes*

you have to read through the documentation to find out which function you should call.

By now, API designers have explicitly recognized this practice as problematic.[3]

In the end, the three themes that offered the best insights were the necessity to understand the API's design aspects and rationale on an as-needed basis, obstacles related to using code examples, and the challenges of dealing with an API's seemingly inexplicable behavior.

Understanding Design Aspects and Rationale

Many survey respondents expressed the feeling that a lack of knowledge about the API's high-level design hindered their progress:

I don't understand the design intents behind the API, the overall architecture, why certain functions are designed as such.

But why do users need to know about an API's high-level design and the intent behind it? General API design guidelines include the principle of "low barrier to entry." According to this principle, developers should design APIs so that their users only need to "take the time to fully understand the whole architecture if they find a particular feature interesting or if they need to move beyond simple scenarios."[3] But is the need to learn more of the design completely explained by the transition from basic to advanced usage scenarios? As it turns out, moving beyond trivial usage involves many types of decisions that can be informed by high-level design. Seven participants specifically said that to properly understand an API, they needed to understand its high-level design. The major insight resulting from this data is that knowledge of an API's high-level design (and its rationale) can help developers choose among alternative ways to use it, structure their code accordingly, and employ it as efficiently as possible.

One participant explained the issue directly:

One of the things that I struggle with is trying to figure out when there're multiple ways of doing something. Which one is more appropriate? I'm not entirely sure … Sometimes it's difficult to know without going and asking. But you don't necessarily have time to go and ask the experts, if you can find them.

A different participant independently echoed these comments on the value of experts for providing the API's design context:

I had the luck of actually working with the main guys that designed it, so I could go and ask people: 'Hey! Why is this like that?' And they would give me the background.

Choosing among alternative usages of an API is a decision that well illustrates the underlying challenge of understanding relevant parts of an API's high-level design on a need-to-know basis, as opposed to systematic study (as Janet Nykaza and her colleagues similarly observed[7]).

Many participants also indicated, explicitly or tacitly, a desire to understand the design and rationale of the API to use it as efficiently as possible. Phrases such as "it would help you make your code better" and "use it the most efficiently" weren't uncommon when participants spoke about the value of design knowledge. One participant explicitly linked knowledge of design intent with a smooth API usage experience.

When you're building a framework, there's an intent … if you can understand what the intent was, you can often code efficiently, without much friction. If you don't know what the intent is, you fight the system.

Seven participants said that to properly understand an API, they needed to understand its high-level design.

Working with Code Examples

In studies of developers, examples often emerge as a key learning resource.[1,7,9] As Samuel McLellan and his colleagues summarize, "The code examples supported several different learning activities, such as understanding the purpose of the library, its usage protocols, and its usage context."[1] It's no surprise that both survey respondents and interview participants repeatedly mentioned code examples. In fact, more than one-quarter of all respondents identified the absence of API usage examples tailored to their needs as an obstacle to learning the API. My detailed analysis of the data largely confirmed McLellan and his colleagues' observations but also explained in more detail how examples support API learning. In fact, studying how examples fail to support developers provided the richest insights about the role of examples in API learning.

We can divide code examples, very roughly, into three categories. In the first category (*snippets*), we find small code snippets intended to demonstrate how to access the basic API functionality. At Microsoft, technical writers author snippets provided in the Microsoft Developer Network Library (MSDN). A typical snippet on MSDN is the 30-line function showing how to read from and write to files. Tutorial examples form the second category (*tutorials*). Tutorials

Design decisions that can impact API usage should be traceable from the point-of-entry documentation pages.

are typically longer, consist of multiple code segments, and form a more or less complete application. They can be embedded in prose and are intended to teach developers a specific aspect of the API. Tutorials can be found on MSDN as well as in blogs and books. The third category of examples consists of code segments from complete applications (*applications*). Applications include both the demonstration samples sometimes distributed with an API and open source projects that developers can download from various source code repositories.

Examples can become more of a hindrance than a resource when there's a clear mismatch between the example's purpose and the user's goal. Most issues with examples were related to participants wanting to use snippets for purposes that went beyond basic interaction with the API. Specifically, participants referred to the following usages when discussing examples:

- providing "best practices" of an API's use;
- informing the design of code that uses the API;
- providing rationale about an API's design; and
- confirming developers' hypotheses about how things work.

Among these, the most prevalent frustration was that snippets didn't provide any support for thinking about "how to put things together:"

The problem is always, when I feel I can't make progress … when there's multiple functions or methods or objects together that, individually they makes sense but sort of the whole picture isn't always clear, from just the docs.

In cases where snippets didn't support the developer's goal, there was sometimes a progression to the use of tutorials and applications. For instance, a participant explained how a scenario involving multiple function calls wasn't supported by snippets in MSDN:

So they have an example showing: "this is how you create a connection, this is how you issue a command with SQL." It wasn't clear from there what to do if I want to do two commands. […] So later I looked into a book and I learned that you have to close the connection, you have to close the command before you can issue another command. I was trying to issue two commands and I was just getting an exception.

In addition to the increased effort of using books or downloading examples from the Internet, we can explain the progression from snippets to applications on an as-needed basis by other trade-offs, including the need to validate examples found on the Internet.

There are examples there [on the Internet] but some of them seem to be quite old. […] Those examples would still run but I don't think they have the most recent way of doing stuff.

Unchecked obsolescence of available resources is only one reason why the collection of user-created Internet resources ("the cloud") isn't the ultimate solution for obtaining API learning resources. Another issue is the credibility of the examples' source. Examples that are more strongly tied to the API's creators seem more attractive because they seem to validate the rationale for using the API in a specific way:

[The example] at least gets you thinking that they are doing it in this particular way so there must be a reason why they chose this particular model […]. Even if I don't completely understand why, I'll do that anyway, just because that's what they say the best practice is, and I assume that the people who design the framework have a pretty good idea of what the best practice ought to be.

However reasonable this assumption, it should be made with care because the people who write API documentation at Microsoft aren't the people who develop the API.

Dealing with Magic

Respondents often mentioned being puzzled by an API's behavior and wanting to access its implementation to solve the puzzle. In his API design guidelines, Bloch encourages developers to "obey the principle of least astonishment" because surprised users introduce bugs.[5] Survey responses show that this general principle can be hard to put in practice. The interviews helped reveal specific factors that astonish developers.

Studying multiple cases of puzzling API behavior elicited a common trend: the API's high-level design explained puzzling behavior to a large extent and wasn't clearly reflected in the low-level documentation. Consequently, the API behaved as documented, but developers had difficulty discovering and interpreting the explanation for its behavior.

In one case, a participant using an activities workflow API couldn't understand why the different activities he created didn't run in separate threads. In this case, the API's design model was counterintuitive, and the participant had to read a book by the API's developers to use it properly.

In another case, a participant who had learned one API was working on a different API derived from the first one. Incompatibilities between the two were unsettling:

If you have a code base here [with one API] and you try to do the same thing [with the other API] and all of a sudden it doesn't work, and you don't know why. So, how do you tackle that?

For a third participant, one specific type of user interface component didn't seem to detect clicks from mouse events, whereas all other similar components did. A nonstandard default value explained by the component's role in the component hierarchy apparently caused this behavior.

So, that's an example of how you can dig it out of the docs post mortem, after you know what happened, but you couldn't predict that behavior ahead of time from the class-level documentation.

What these cases have in common are that some aspects of the API's design have observable consequences on the API's behavior but aren't explained clearly in its low-level documentation. As a result, they didn't surface when developers first tried to use the API, leading to inexplicable behavior.

One coping strategy for such situations was a desire to inspect the API's implementation. For example, one respondent indicated that an obstacle was

no view of the implementation. Good API design shouldn't require this, but in reality, understanding the internals can make a big difference in how well an API is used.

This trend is interesting because it goes against received knowledge about the principle of information hiding. As another example, in responding to a question about what led him to look at the API source code, a participant answered,

Binding, for example, has a lot of magic. A lot of "if your class is this class then we have a special behavior for it, if it's not, it doesn't." These things are hinted at in the documentation, but it's not clear what the rules are. That's where looking at the source could help.

In brief, a main observation from this analysis is that the reason for the puzzling behavior existed but wasn't easily found because it related to high-level design concerns that weren't referred to in point-of-entry API documentation. API documentation guidelines generally focus on the thoroughness of the low-level documentation,[5] but a complement to this guideline is that low-level documentation should address design decisions that can impact the API's behavior.

One overarching result of this study is that the resources available to learn an API are important and that shortcomings in this area hinder the API learning progress. When learning APIs, developers interact with resources for many purposes, such as discovering key information about the API's high-level design. Developers in the study tried to understand part of the high-level design for many reasons, including finding out how to most efficiently use the API and understanding subtle aspects of its behavior. Studying examples is an important strategy for learning about design, but this approach led to frustration when the examples weren't well adapted to the task. Finally, some participants perceived API behavior that seemed inexplicable at first to be a major obstacle.

These observations have implications for API users, designers, documentation writers, and development tool builders. Developers stuck while learning an API should consciously try to match their information needs with the type of resource most apt to provide the required knowledge. For instance, if the API's behavior seems inexplicable, the answer might have as much to do with its design as with its low-level structure. Developers looking for ways to interact with different API methods along complex protocols might be more likely to find good examples in advanced tutorials and applications, rather than snippets.

For API designers and documentation writers, these observations complement existing API documentation guidelines by emphasizing that design decisions that can impact API usage should be traceable from the point-of-entry documentation pages and that user expectations about what they can get (and not get) out of a specific type of

resource should be explicitly stated. For instance, it might be worthwhile to explicitly state the range of usages illustrated by code examples provided in the documentation and point to other resources for additional usages.

Software tools can also assist developers in their quest for a better grasp of APIs. For now, search tools are promising in this area because they help bridge the gap between API users' information needs and the corresponding resources (such as code examples). As APIs keep growing larger, developers will need to learn a proportionally smaller fraction of the whole. In such situations, the way to foster more efficient API learning experiences is to include more sophisticated means for developers to identify the information and the resources they need—even for well-designed and documented APIs.

Acknowledgments

I thank the members of the Human Interactions in Programming group at Microsoft Research for a welcoming environment, and the anonymous participants who made this research possible by generously sharing their experience. Additional thanks to Rob DeLine, Andrew Begel, Barthélémy Dagenais, Ekwa Duala-Ekoko, Reid Holmes, Tom Zimmermann, and the anonymous reviewers for comments on this article.

References

1. S.G. McLellan et al., "Building More Usable APIs," *IEEE Software*, May/June 1998, pp. 78–86.
2. J. Stylos and S. Clarke, "Usability Implications of Requiring Parameters in Objects' Constructors," *Proc. 29th Int'l Conf. Software Engineering* (ICSE 07), IEEE CS Press, 2007, pp. 529–539.
3. K. Cwalina and B. Abrams, *Framework Design Guidelines: Conventions, Idioms, and Patterns for Reusable .NET Libraries*, 2nd ed., Addison-Wesley, 2009.
4. S. Clarke, "Measuring API Usability," *Dr. Dobb's J. Special Windows/.NET Supplement*, May 2004.
5. J. Bloch, "How to Design a Good API and Why It Matters," *Companion to the 21st ACM SIGPLAN Symp. Object-Oriented Programming Systems, Languages, and Applications* (OOPSLA 06), ACM Press, 2006, pp. 506–507.
6. B.A. Kitchenham and S.L. Pfleeger, "Chapter 3: Personal Opinion Surveys," *Guide to Advanced Empirical Software Engineering*, F. Shull, J. Singer, and D.I.K. Sjøberg, eds., Springer, 2008, pp. 63–92.
7. J. Nykaza et al., "What Programmers Really Want: Results of a Needs Assessment for SDK Documentation," *Proc. 20th ACM SIGDOC Ann. Int'l Conf. Computer Documentation*, IEEE CS Press, 2002, pp. 133–141.
8. R.S. Weiss, *Learning from Strangers: The Art and Method of Qualitative Interview Studies*, Free Press, 1994.
9. F. Shull, F. Lanubile, and V.R. Basili, "Investigating Reading Techniques for Object-Oriented Framework Learning," *IEEE Trans. Software Eng.*, vol. 26, no. 11, 2000, pp. 1101–1118.

About the Author

Martin P. Robillard is an associate professor in the School of Computer Science at McGill University. His research focuses on software evolution and maintenance. Robillard has a PhD in computer science from the University of British Columbia. Contact him at martin@cs.mcgill.ca.

Top articles, podcasts, and more.
cn
computingnow.computer.org