Empir Software Eng (2011) 16:703–732
DOI 10.1007/s10664-010-9150-8

A field study of API learning obstacles

Martin P. Robillard · Robert DeLine

Published online: 14 December 2010

Editor: Premkumar Thomas Devanbu

Abstract Large APIs can be hard to learn, and this can lead to decreased programmer productivity. But what makes APIs hard to learn? We conducted a mixed approach, multi-phased study of the obstacles faced by Microsoft developers learning a wide variety of new APIs. The study involved a combination of surveys and in-person interviews, and collected the opinions and experiences of over 440 professional developers. We found that some of the most severe obstacles faced by developers learning new APIs pertained to the documentation and other learning resources. We report on the obstacles developers face when learning new APIs, with a special focus on obstacles related to API documentation. Our qualitative analysis elicited five important factors to consider when designing API documentation: documentation of intent; code examples; matching APIs with scenarios; penetrability of the API; and format and presentation. We analyzed how these factors can be interpreted to prioritize API documentation development efforts

Keywords Application programming interfaces · Software libraries · Programming · Documentation

1 Introduction

Recently, much effort has been spent trying to improve the usability of Application Programming Interfaces (APIs), including frameworks for estimating API usability based on its structure (Clarke 2004), API design assessment studies, and empirical

M. P. Robillard (✉)
School of Computer Science, McGill University, 3480 University St., #318,
Montréal, QC, H3A 2A7, Canada
e-mail: martin@cs.mcgill.ca

R. DeLine
Microsoft Research, One Microsoft Way, Redmond, WA 98052, USA
e-mail: rob.deline@microsoft.com

studies of API documentation (see Section 2). Researchers are also continually proposing new tools to facilitate learning and using APIs, including advanced documentation systems (Berglund 2003), usage example finding tools (Holmes et al. 2006), and various types of library usage checkers (Feilkas and Ratiu 2008).

In this space of design and innovation for APIs (Stylos and Myers 2007), it can be difficult to judge where further research and development investments should be directed to achieve significant impact. There exists little data about the nature of the challenges associated with learning APIs, and the data that does exist is fragmented. A few studies targeted the types of questions developers ask when conducting maintenance tasks (Hou et al. 2005; Ko et al. 2007; Sillito et al. 2008), but only a small subset of the questions elicited had to do specifically with APIs. Industrial API usability studies have also been reported (Jeong et al. 2009; McLellan et al. 1998; Nykaza et al. 2002), but these all targeted a single, generally small API. In this context, we were interested in determining, more broadly: what are the major difficulties for developers learning new APIs?

To answer this question, we conducted a series of surveys and interviews with professional software developers to determine, both broadly and specifically, the nature of the obstacles they faced when learning new APIs. The series of studies consisted of:

1. An **exploratory survey** to determine the broad classes of obstacles, and insights to guide further research;
2. A set of **qualitative interviews** to collect informed opinions and detailed stories about API learning in the workplace;
3. A **follow-up** survey to confirm our general findings and collect additional data on the demographic factors that could impact API learning obstacles.

We know of no existing general theory about the obstacles professionals face when learning APIs, so we used a grounded approach to help move us towards a deeper understanding of the factors that impact the API learning experience of software developers. Our overarching methodological goal for all phases of the study was therefore to collect data that was *grounded in developers' experience*.

The preliminary results collected as part of the exploratory survey and the initial interviews identified API learning resources as a critical element in developers' API learning experience. As summarized by a participant in our study:

> The biggest hurdle when learning an API is the documentation. If the documentation for an API is good, it solves 99% of your problems [Participant 20, see Table 2].

These preliminary findings are described in a previous report (Robillard 2009). Based on these findings, we focused the remainder of the study (part of the interviews and the follow-up survey) on isolating the elements of API learning resources that influenced our participants' experience, and on recording the nature of this influence.

The contributions of this paper include our findings about the obstacles developers face when learning new APIs, an in-depth analysis of the *nature* of the obstacles faced by developers, and corresponding insights for prioritizing API documentation development efforts. Our analysis of documentation-related challenges revealed five important factors that impacted developers' API learning experience: (1) documentation of intent; (2) code examples; (3) cookbooks for mapping usage scenarios

to API elements; (4) penetrability of the API; (5) format and presentation of the documentation.

Although this study was conducted within a single organization (Microsoft), this corporate environment provides excellent conditions for a broad study of API usage. First, Microsoft's development staff consists of over 30,000 engineers, distributed worldwide, with varying levels of experience. As such it represents a non-trivial segment of the software development population at large. Second, Microsoft developers work on a large variety of APIs that are public and widely used outside the company. Finally, since most of the APIs developers reported on in this study were also developed at Microsoft, we were in a unique position to reason about the various explanations for the API learning obstacles we observed. With data collected from over 440 different professional developers, this study is, to the best of our knowledge, the first comprehensive field study of API learning obstacles. With a specific focus on API documentation, the implications of our study provide clear insights that can help prioritize API documentation and API tool support efforts in the future.

In the rest of this paper, we first provide an overview of previous empirical work related to API usability (Section 2). We then provide an overview of the research (Section 3) where we describe the methodology followed and intermediate results obtained in each phase of our study. In Section 4, we report on the quantitative findings that steered the research, and in Section 5 we present our principal qualitative results, which focus more specifically on documentation-related obstacles. We discuss the validity of the study in Section 6 and conclude in Section 7.

2 Related Work

Pioneering Work The earliest work related to API usability includes a number of lab studies conducted in the late 1990s. Rosson and Carroll (1996) investigated the reuse strategies of experts learning a graphical user interface (GUI) API in SmallTalk. A study by Shull et al. (2000) also investigated how subjects learned a GUI-framework in C++, but this study involved student programmers. A study by McLellan et al. (1998) focused on the usability of APIs for specialized hardware and involved professional programmers. Despite their widely varying experimental contexts and populations, all three studies report on extensive use of examples in the participant's strategies for learning the API, an observation that was consistently asserted in most subsequent studies.

Information Needs of Developers Many more recent studies on the topic have focused on the information needs of developers engaged in maintenance tasks. Based on two users studies involving, respectively, nine graduate students and 16 professional programmers, Sillito et al. (2008) provided a catalog of 44 questions programmers ask during software evolution tasks. In a later study, Ko et al. (2007) observed 17 professional developers during their work and produced a different catalogue of 21 information types developers seek during their work. Although both studies report questions representing different levels of abstractions, their catalogues show evidence of a marked concern for matching scenarios in the application

domain with the corresponding program behavior. Hou et al. (2005) also studied the questions of programmers, but as reflected through a sample of 300 messages posted on mailing lists for the Swing framework. Their study identified a list of API design and documentation aspects that were often reflected in newsgroup questions. In a separate study, Hou (2008) investigated the effect of framework design on reuse patterns through a detailed inspection of 11 student projects. Seaman (2002) investigated the information gathering strategies of software maintainers, reporting on extensive use of the source code to understand software. Ko et al. (2004) also studied learning barriers faced by non-programmers in their attempts to use a simple programming system. This study elicited a detailed description of six different types of barriers. Finally, a number of studies conducted in industry assessed the usability of specific APIs (Beaton et al. 2008; Bore and Bore 2005). The above studies yielded a valuable body of knowledge on programmers' information needs and learning strategies, but do not broadly address the challenges of learning APIs in the field: the studies either targeted students, a single API, or general change tasks with no specific focus on API usage.

API Usability Studies In addition to the exploratory studies described above, a number of research projects have specifically targeted the evaluation of the usability of an API's structure. Clarke (2004) proposed a framework of cognitive dimensions for assessing API usability. This framework captures important practical knowledge and has been used to reason about API usability by other researchers at Microsoft. Formal API usability studies include that of Stylos and Clarke (2007), who investigated the impact of various design guidelines on API usability, including the use of parameters in object constructors, the use of the Factory design pattern (Ellis et al. 2007), and method placement choices (Stylos and Myers 2008). This work, usually conducted in the form of lab experiments, was able to demonstrate that API design decisions have significant impact on API usability in the conditions studied. Stylos et al. (2008) also followed-up with a case study of user-centered API redesign for usability in an industrial context, which further demonstrated the impact of API structure on usability. As opposed to our field study, the above studies were deductive (hypothesis-testing), and focused exclusively on API structure, excluding other factors such as API documentation or the learning process.

Documentation Studies In contrast to studies focusing on API structure, additional related work specifically targeted API documentation. Lutters and Seaman (2007) conducted a qualitative study of the use of documentation in an industrial setting through the elicitation of "war stories", a technique we have integrated as part of our study. In this study they report on the pivotal role of individuals as pointers, gatekeepers, or barriers to documentation. Nykaza et al. (2002) conducted a needs assessment study for a domain-specific API through surveys and interviews of developers, and Jeong et al. (2009) examined the use of user studies to improve the documentation for a specific Service Oriented Architecture system. Nykaza's et al.'s study identified, among other requirements, the importance of an overview section in API documentation. Jeong's study identified 18 guidelines they believe would lead to increased documentation quality for the system under study, including "explaining starting points" for using the API. Another recent study by Brandt et al. (2009) reports on the use of on-line resources by programmers involved in software

reuse. In this study, the authors observed consistent differences in query styles and durations that may be linked to the purpose of the query. Our study complements the work described above by focusing on documentation-related obstacles as only one facet to API learnability, and in the broader context of multiple APIs.

Documented Practical Experience The empirically-derived body of knowledge generated by the studies described in this section is complemented by the work of expert API designers, who have shared their accumulated knowledge through books (Cwalina and Abrams 2009; Tulach 2008) and other media, such as blogs and talks (Bloch 2006; des Rivières 2004). The advice offered in these contributions focuses heavily on techniques for structuring the API to eliminate accidental complexity (suggesting, for example, practical ways to distribute functionality among modules). Besides naming conventions, few specific insights on documentation concerns are typically provided.

3 Overview of the Research

We followed a multi-phase, mixed-method approach that involved successive refinements of our research questions to create knowledge grounded in the experience of professional developers. In a first phase, we conducted an exploratory survey to determine *what makes APIs hard to learn?* This survey identified inadequate learning resources as a critical obstacle for developers learning new APIs. In a second phase, we interviewed developers to understand API learning obstacles in detail, with a special focus on the role of learning resources in the API learning process. Finally, we conducted a follow-up survey to validate our hypothesis that *inadequate documentation is the most severe obstacle* and to study the relationships between API learning obstacles and demographic variables in the developer population.

This work focuses on the technical aspects of APIs and their documentation, and does not attempt to relate our findings to any of the numerous available theories of learning (Olson and Hergenhahn 2008). However, we assume as valid the six core principles of adult learning proposed by Knowles et al. (2005), namely, that adult learners (1) need to know why/what/how they learn; (2) are self-directed; (3) build on prior experience; (4) are motivated by subjects of immediate relevance; (5) are problem-centered, and (6) see learning for its intrinsic value. As such, we do not report results that are implied by these principles (e.g., participants stating that they learn by example).

3.1 Target Population

We drew participants from the worldwide population of Microsoft Software Developers (henceforth referred to as "developers"). The implications for generalizability of the findings are discussed in Section 6. Microsoft's software development staff consists of roughly 30,000 engineers, of which most are developers, testers or program managers. For the purpose of this research, we considered developers to be all employees holding jobs whose title implies engagement in software development,

but we excluded testing engineers due to the specialized nature of their work. Specifically, the software development staff in the population we considered covers four distinct groups:

- **Software Development Engineers (or SDEs)**, whose primary responsibility is software development;
- **Lead SDEs**. Although technically a management position, Lead SDEs are typically involved in active software development along with their team;
- **Architects**, a multi-disciplinary role involving both program management and software development;
- **Other** development staff, who typically specialize in one area, such as security, or user experience.

Within one category, the seniority of developers is distinguished according to four different ranks that correspond to increasing levels of recognition and responsibilities. For example, SDEs rank from SDE to SDE 2, Senior SDE, and Principal SDE (most senior). We note that at Microsoft, career ranks are not necessarily related to years of professional experience. Developers in our population can also be distinguished by the location of their workplace. Although the majority of developers work at Microsoft's main campus in Redmond, WA, a fraction of the total developer population works in other locations worldwide.

3.2 Phase I: Exploratory Survey

We surveyed developers to uncover the types of obstacles they face when learning new APIs. To ensure we did not bias the survey results with any preconceptions, we left the main questions open-ended. The survey asked respondents to comment on their most recent learning experience with a publicly-released API. Developers were asked to describe their three most important obstacles. The survey instrument is provided in Appendix A.

Because this survey also served to recruit participants for in-person interviews, we constrained the sampling frame to that of Microsoft developers located in Redmond, WA, USA (the final phase of the research targeted the worldwide developer population). We randomly selected 1,000 developers from this pool and sent them a link to the survey. For this survey no incentive was offered to participants.

Eighty developers (8%) answered the survey. The set of respondents constituted a representative sample of the target population, cutting across job titles in proportion to the target population, and reflecting on the use of a wide variety of APIs.[1] Across all job titles, respondents had on average 12.9 years of professional experience (self-reported). 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 libraries the respondents were learning included a library 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.

[1] Details about the population of respondents can be found in a separate publication (Robillard 2009).

Table 1 Obstacle categories (exploratory survey)

Obstacle related to...	# resp.
Learning resources (e.g., documentation, code examples)	50
API structure (e.g., design, names of API elements)	36
Developer background (e.g., prior knowledge, professional training)	17
Technical environment (e.g., build tools, testing infrastructure, hardware)	15
Process (e.g., lack of time, interruptions by colleagues, other priorities)	13

We collected 165 free-form descriptions of obstacles respondents faced when learning APIs. Each respondent described between one and three obstacles. We analyzed these descriptions through a process of open-coding (Creswell 2007, p. 239), which involves attaching short labels ("codes") to text segments that share some commonalities and iteratively merging and refining labels based on the themes that emerge from the data. From this process, we elicited five dominant categories of obstacles.

Table 1 shows the obstacle categories we elicited and the number of respondents who made at least one response that fell in that category. The resulting categories are clearly inter-related. For example, an API with a simple structure will require fewer learning resources to master. Similarly, developers with little experience would probably be expected to require more tutorial documentation than more experienced ones. The purpose of these categories was to provide an initial set of topics for our interviews in Phase II. To the extent that these categories are inter-related, the informants were free to discuss them together. In addition to motivating our decision to focus on documentation, our exploratory survey guided the subsequent research by providing a more detailed collection of important themes (such as the use of code examples in API learning) to explore further.

3.3 Phase II: Qualitative Interviews

The specific goals of the interviews were to get a detailed picture of the important obstacles developers faced when learning new APIs, to get the context in which these obstacles occurred, and to infer possible causes and explanations for these obstacles. For this reason, we chose an open-ended, loosely-structured style of qualitative interview (Weiss 1994), which consisted of asking participants to summarize their work with the API and explain the obstacles they faced. In this way, participants would naturally discuss their main concerns.

We interviewed 28 software developers recruited from respondents to the exploratory survey (24) and from personal contacts (4). Interviews lasted between 15 and 45 min and were audio-recorded (except one due to a malfunction of the recording equipment). In addition to these systematic interviews of software developers, we also interviewed other stakeholders in the API development process, including one API usability researcher, two program managers responsible for API design and reviews, and one lead content publisher responsible for producing API documentation. These interviews provided additional perspectives on what we heard from software developers. The interviews took place over a period of several months. During this time, we interleaved interviews with qualitative analysis of the data. As the main themes emerged, we focused the interviews on these themes. The data

Table 2 List of informants

#	Title	Exp.
1	SDE 2	20
2	SDE 2	10
3	Architect	17
4	SDE 2	5
5	SDE 2	18
6	Lead 2	15
7	SDE	10
8	Senior Lead	15
9	Senior Lead	24
10	SDE 2	4
11	SDE	7
12	Senior SDE	11
13	Senior SDE	15
14	SDE	6
15	SDE 2	13
16	Senior SDE	20
17	Senior Lead	15
18	Senior Lead	6
19	SDE	2
20	SDE 2	5
21	SDE 2	8
22	Principal SDE	33
23	SDE 2	10
24	Senior SDE	30
25	Principal Lead	23
26	SDE	3
27	Senior SDE	11
Overall: 1 Architect; 20 SDEs; 6 Team Leads		

we collected for this phase included over 12 hours of recordings of 30 software professionals, which we transcribed. The transcripts were then *coded* (Weiss 1994) several times by one author and discussed extensively among both authors, in an iterative process, to derive the main themes described in Section 5. Table 2 lists the job title and years of experience of all the developers who participated in the interviews. The identification number (left column) is used to link them to the sources of evidence in Section 5.

Analyzing the transcripts revealed an important attribute of experiences described by developers: the *learning context* they were describing. The learning context captures the reasons and motivation for learning the API. Table 3 describes the five learning contexts we elicited.

Table 3 Learning contexts for participants

Context	Learning the API...
Owning	because the respondent joined the team developing it
Major	to use as major component of production code
Minor	to complete some minor task (e.g., bug fix)
Exp.	to experiment with a technology
Hobby	for a side project not critical to main work

The analysis of the qualitative data we collected during this phase forms the cornerstone of this study. As is typical in qualitative research, our completed report "includes the voices of the participants, the reflexivity of the researcher, and complex description and interpretation of the problem" (Creswell 2007, p. 37). In addition, our qualitative analysis and our quantitative data obtained through the next phase of our study together provide multiple, converging lines of evidence. The results of the qualitative analysis are presented separately, in Section 5.

3.4 Phase III: Follow-up Survey

Phases I and II provided us with detailed accounts of how different factors interact to facilitate or hinder developers learning new APIs. However, they did not provide information about the possible links between characteristics of the developers and their experience learning APIs. Additionally, the generalizability of the qualitative findings is difficult to establish. We conducted a follow-up survey to address these points. The survey instrument is provided in Appendix B.

We designed our survey taking into account what we had learned about API learning obstacles. The survey asked developers to focus on *the last API they had learned* and was designed in four parts, three of which are relevant to this research.

Demographics and API Learning Context We asked developers for their years of professional experience, the API that they had last learned, the programming language they used, and the learning context (Table 3).

Obstacle Severity We asked respondents to comment on the severity of five different types of API learning obstacles. We describe each obstacle with the text used in the survey.

- *Background*: Your background was not adapted to learning the new API. For example: you were not familiar with the programming language or application domain, your previous knowledge of a similar API (or a previous version of the API) made it confusing to learn the new API.
- *Structure*: The way the API was structured or designed made it difficult to understand. For example: it was not clear how to instantiate an object, there were too many abstract classes, the names did not make sense.
- *Technical Environment*: The technical environment made it difficult to use the API. By technical environment, we mean any technical aspects not directly related to the design of the API itself. For example, the tools did not work well, the API required extensive infrastructure to test, you could not get the builds to work.
- *Low-Level Documentation*: Specific member-level usage details were not documented. For example, description of parameters, error codes.
- *High-Level (or conceptual) Documentation*: You did not find conceptual-level information explaining how to use the API. Consider "conceptual-level" information to mean any type of information you need to use the API correctly that is not typically associated with particular API members (classes/methods/functions). For example, description of required concepts, the API's execution model, non-trivial code examples, usage patterns, best practices, mappings between scenarios and API members.

These categories map to the categories described in Table 1 with two exceptions. First, we eliminated the Process category as it was the least reported and offered limited insight into potential for improving APIs. Second, we split the documentation categories into two categories representing, respectively, the class-member-level reference documentation, and the other learning resources such as conceptual overviews. We introduced this distinction because the interviews revealed that our informants naturally distinguished between reference documentation of individual API members and more general documentation entries offered as learning resources. Because at Microsoft reference documentation for APIs is almost universally delivered in a specific format (not unlike the popular Javadoc-style for Java), the distinction between reference and conceptual documentation is objective. Unfortunately, among developers there is no standard vocabulary to distinguish reference documentation from conceptual documentation. For example, for reference documentation terms such as "class-level" or "member-level" documentation are also used. To paint over these differences we elected to use the very generic terms "Low-level" (reference) and "High-Level" (conceptual) documentation.

Finally, we also asked developers to comment on other obstacles they might have encountered. Respondents were asked to rate the severity of each obstacle according to a six-value ordinal scale: *Not an obstacle, trivial, moderate, severe, very severe, blocker*.

Solutions Respondents were also asked to rate 14 different potential solutions to documentation-related obstacles using a Likert-scale. The solutions were derived from proposals from the literature and from insights generated from the interviews. Unfortunately after review we found that there were too many different ways to interpret the descriptions of the solutions in the survey to provide reliable insights for future work. For this reason, we did not use this part of the survey. However, as part of this question respondents were also invited to comment on other potential solutions to their problem in a free-form comment box. This part of the question produced valuable data which we integrated in our qualitative analysis.

We sent this survey to 2,000 randomly-selected software developers from Microsoft's worldwide developer population, 1,936 of which were reachable. Participants in phases I and II of the research were explicitly excluded from the sampling pool. Targeted developers had one week to complete the survey. As an incentive to participate, all respondents who completed the survey were entered in a draw for a $250 (US) gift card for an on-line store.

A total of 334 developers (17.3%) answered the survey. 56.8% of the respondents were from the Redmond campus, and the rest from other locations. Table 4 shows the geographic distribution of the population under study, our random sample of 1,936 developers, and the 334 respondents. Overall, the geographic distribution of respondents very closely maps the distributions of both the sample and the total population (<1.5% difference in ratios for the geographical categories used). The maximum discrepancies are a 1.1% over-response by Redmond developers (with respect to the corresponding ratio in the overall population), and a 1.3% under-response by developers from Asian locations (including Australia). Respondents had on average 9.8 years of professional experience (self-reported). The median was 8.5 years, and 82% of the respondents reported four or more years of experience.

Table 4 Geographic distribution of respondents

Region	Population (%)	Sample (%)	Respondents (%)
Redmond (WA, USA)	55.7	53.6	56.8
Puget Sound (WA, USA)	8.5	9.7	9.0
America (other locations)	9.6	10.5	9.6
Europe	8.1	8.2	8.1
Asia and Australia	15.8	15.1	14.4
Other	2.3	2.9	2.1

4 Quantitative Results

Figure 1 shows the relative severity of each type of obstacle. For this figure, the categories *Blocker, Very Severe*, and *Severe* were aggregated as "Severe", and both *Not an obstacle* and *Trivial* were aggregated as "Trivial". Documentation-related obstacles are perceived as the most severe by developers, followed by structure, and other types of obstacles.

We analyzed the relationships between the severity variables of Fig. 1. A Pearson R test between obstacle severity variables revealed no strong correlation (values varied between 0.149 and 0.405 and were all statistically significant).

Observation 1 *The severity variables are only weakly correlated (*< 0.5*), which indicates that respondents perceive differences in the types of obstacles.*

In particular, the limited correlation of 0.377 between the two types of documentation-related obstacles reinforces our decision to treat these types of documentation as distinct in our research. Figure 1 provides evidence that the qualitative

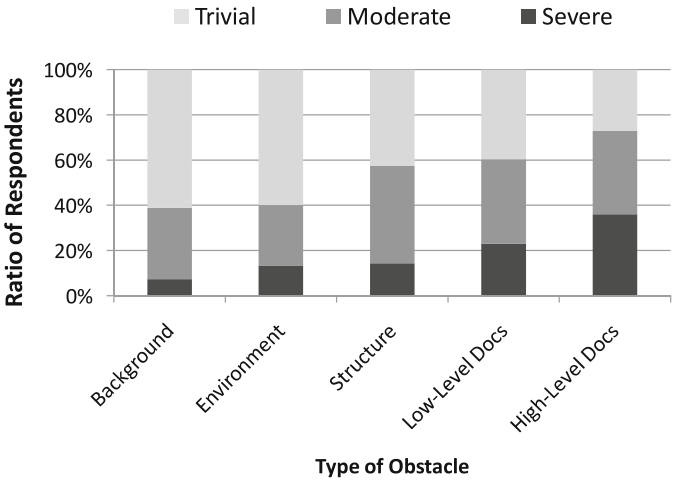

Fig. 1 Obstacle severity (follow-up survey)

Table 5 Variations between proportions in obstacle severity

Relation / Relative severity	<	>	=
Background ? Structure	151	56	124
Background ? Environment	116	91	124
Background ? Low-level docs	169	78	80
Background ? High-level docs	208	39	83
Structure ? Environment	76	143	112
Structure ? Low-level docs	131	87	109
Structure ? High-level docs	158	57	115
Environment ? Low-level docs	160	78	89
Environment ? High-level docs	195	55	80
Low-level Docs ? High-level docs	135	67	124

findings we made about documentation-related obstacles are likely to generalize to developers outside the group of interviewed participants, since surveyed developers also report documentation-related obstacles as the most severe.

We also analyzed the within-subject obstacle severity relationships through pairwise McNemar tests of variation between proportions.

For example, 46% of respondents rated Background types of obstacles as less severe than Structure types of obstacles, 17% of respondents rated Structural obstacles as less severe, and 37% rated the two obstacle types as equal in severity. This variation in proportion is determined to be highly unlikely to appear randomly ($p < 0.0001$), from which we infer that the fact that respondents rate Background obstacles as less severe than Structure ones is statistically significant. Table 5 shows the variations in proportions between all obstacle types. We performed the analysis for all pairs of obstacle types and made the following observation:

Observation 2 *The order of bars shown in Fig. 1 represents a statistically significant progression in the perceived severity of obstacles, except for the relationship between Background and Environment related obstacles (McNemar test of variation between proportions, $p = 0.01$).*

Using a chi-squared test of independence, we also explored the relationship between characteristics of the population and the responses to the obstacle severity variables. We did so by partitioning the respondents according to thresholds on different demographic variables, and using the 3-valued partition on severity shown in Fig. 1. Specifically, we explored the relationship between responses to severity of all obstacles and the following variables:

- **Overall experience**. We partitioned the population into the 50% less and more experienced developers.
- **Extreme experience**. We selected the groups 25% more experienced developers, and everyone else.
- **API learned**. We compared all developers who had learned a legacy type of API (Win32) against all developers who had learned an API of the more recent .NET framework.
- **Language Used**. We compared all developers who were using C/C++ to access the API with those who were using C#.

– **Learning Context**. We compared the responses of developers who had declared different learning contexts (see Table 3).[2]

Of the five partitions × five obstacles = 25 potential relationships between demographics and obstacle severity, only two were statistically significant at the 0.01 level:[3] (1) Structure and (2) high-level documentation-related obstacles faced by developers learning Win32 APIs are significantly perceived as more severe that those faced by learners of .NET APIs. At the 0.05 level, two additional relations can be detected: low-level documentation-related obstactles are also perceived as more severe by learners of Win32, and high-level documentation-related obstacles are perceived as more severe by users of C/C++ (as opposed to C#). We thus note that experience level and learning context were not significant enough factors to register impact at the level of measure that we studied.

Overall, the survey-derived obstacle ranking of Fig. 1 can thus be considered robust with respect to both developer experience and learning context because neither factor is a predictor of severity.

5 Qualitative Analysis of Documentation-Related Obstacles

Our study identified inadequate API documentation as the most severe obstacle facing developers learning a new API, and we decided to concentrate on this critical type of obstacle. Indeed, "documentation-related obstacles" is a very broad (and vague) area of concern, and we needed to determine what aspects of the API documentation impacts how well developers learn it. Through a qualitative analysis of the interview and survey data (see Section 3.3), we elicited five dimensions of analysis and, from these, isolated the factors that influenced developers' perception of API documentation-related obstacles. In this section, we describe each dimension of API documentation, and link it with our quantitative results when appropriate. Based on this analysis, we provide recommendations for prioritizing API documentation efforts for each dimension.

Link to the Evidence We support our findings with qualitative evidence from the interviews and the free-form responses from the survey respondents,[4] and evidence from the literature. When appropriate, we also link the findings to some of our quantitative findings as described in Section 4.

We link all our findings to the qualitative data that supports it by labeling all quotes with the participant responsible for the quote, and observations with the list of participants whose quotes helped inform the observation. References to participants are provided in brackets, where the participant number corresponds to those in Table 2. For ease of interpretation, we also include the general professional titles

[2]To test with sufficiently large partitions, we aggregated the *Owning* and *Major* categories, and the *Experimental* and *Hobby* respondents.

[3]After correcting the p-values for the simultaneous testing of multiple hypotheses using Holm's procedure (Westfall et al. 1999).

[4]Drawn only from the follow-up survey, since the interview participants were selected from the exploratory survey.

with the participant numbers (Dev for SDEs of all levels, Arch for architects, and Lead for development leads of all levels). When observations were also informed by survey responses, we include the label "Survey". For example, the label [Dev 10,11; Lead 25; Survey] means that the corresponding observation was based on quotes from informants # 10 and 11 (who are developers), informant # 25 (who is a team lead) and one or more survey responses.

It is important to note that the number of informants discussing a given theme is not a reliable measure of how often this theme was encountered by developers in general because open-ended interview techniques do not involve systematically covering all areas of interest. The goal of this section is not to describe the frequency of a phenomenon but to explain some of the human processes involved in this phenomenon as experienced by developers. For a general picture of the obstacles encountered by developers when learning APIs, we rely on the survey data as described in Section 4.

5.1 Intent Documentation

> A lot of the time when you're writing software there's an intent, you expect something to work a certain way. You code it that way. And you hope that the [documentation] team figures out what your intent was. [Arch 3]

The documentation of *intent* in an API provides information about why certain API design decisions were made, and how the API is intended to be used. Capturing and documenting intent incurs a cost. Why should intent be documented? How do API users take advantage of intent documentation?

The interviews revealed the different reasons why developers wanted or needed to learn about the intent behind the design of various APIs. At least eight participants shed light on the benefits of providing design intent in API documentation. These benefits can be separated into three classes.

Reasoning About the Correct and Efficient Way to Use the API Participants mentioned that knowledge of the intent behind the design of an API generally helped "code efficiently without much friction" [Arch 3], but also helps developers avoid misuse of the API through information about what "it's designed for and what it's not designed for" [Lead 25], and avoid pitfalls:

> Because another problem [...] is that there is documentation, it's good, and then you go ahead and take advantage of the features of the API, but you end up using it in a way that the writers of the API didn't quite mean for you to do it. And at that point you can actually shoot yourself in the foot. [Dev 20]

Determining How to Implement Advanced API Usage Code This task was repeatedly described as one of the most central challenges when learning to use an API. Intent documentation can help with this aspect through insights about why and how a certain class should be used [Arch 3], or if there are many apparent ways to implement a solution, which would be the best (i.e., which would be "intended" by the API designers) [Dev 10,20; Survey].

Reasoning About Performance Characteristics The documentation of intent also relates to the documentation of performance characteristics. Some participants had

experienced difficulties where unexpected performance was thought to be caused by going against the intent of the API [Lead 6; Dev 10]

> Nowhere in there does it say, "and we intended to be used for a few graphics of small size because the memory footprint is going to be this" [Arch 3]

Notwithstanding the benefits, documenting intent involves a tradeoff. Besides the obvious cost of capturing and documenting intent during the API design process, and the fact that additional information can bloat the documentation, not all participants universally considered design intent to be of significant help. One respondent even provided this word of caution:

> Often developers use actual behavior, not intent. If we document the intent (too much), then it becomes a must-fix bug to match the intent. [Survey]

The above quote corroborates the findings of Nykaza et al., that their subjects "felt that the learning content should focus on how to do things, not necessarily why" (Nykaza et al. 2002, p. 136).

Implication 1 *Intent documentation should only be provided on an as-needed basis. Sections of the API where correct usage is not self-evident, documentation to support advance uses of the API, and performance aspects are areas likely to benefit from intent documentation.*

5.2 Code Examples

> It's tough to know the context of the example and yet it has to be very small, and only highlight exactly what the concept in the API is that you're looking for. They have to work, too! [Lead 9]

Code examples are listings, of various length, that show an API being used. We distinguish four categories of code examples: short *code snippets* intended to demonstrate a specific aspects of the API; sequences of small code examples that, together implement a small but non-trivial piece of functionality (*tutorials*); small but complete and self-contained *sample* applications; *production code* which, when available, can also be searched for examples of API usage. In most cases, code examples are provided with a certain amount of context (such as natural language explaining the concepts illustrated by the example).[5] Although the related context is likely to impact the effectiveness of the example for learning how to use the API, informants were surprisingly quiet about the relation between code examples and their context. One explanation could be that developers eagerly seek examples in documentation pages, skipping over explanatory text. In any case, our following discussion assumes that code examples are provided in their appropriate context.

Previous researchers have repeatedly observed that code examples are an essential element of API learning (Brandt et al. 2009; McLellan et al. 1998; Nykaza et al. 2002; Shull et al. 2000), and are used by developers for a whole range of purposes, including "understanding the purposes of the library, its usage protocols, and its usage contexts" (McLellan et al. 1998). Our data directly corroborates these findings,

[5] A notable exception would be examples returned as the result of queries to code search engines.

but also allows to further contribute a detailed description of the attributes that make examples valuable. Over 25 participants commented on the role of examples in their API learning experience.

Example Complexity How complex[6] should code examples be? Although examples of different complexity will serve different purposes, informants provided many insights about their desired level of complexity for code examples. A first observation is that the pedagogical power of examples is perceived to decrease as the size of the example grows, not only because it is longer, but also because it embodies a tangled web of different functional concerns [Dev 2; Arch 3; Lead 8, 25; Survey].

> [Longer examples have] really been informative in getting us going. But it doesn't mean we understand them, it just means we can copy them and get them to work. [Arch 3]

However, examples can easily become too simple. Participants indicated that code snippets exercising only one API method were of little value [Dev 1, 2, 26; Lead 9; Survey]. The ideal size for examples appears to be examples that demonstrate single programming patterns, namely, ways to use a number of related API functions together [Dev 2, 27].

> So just looking at one method call, it didn't really show the flow of everything that you needed to do to tie it together. So it would have been really useful there to have: "this is the sample code that shows, these are the steps to do it, and you can kind of take that as a starting point". [Dev 2]

Proposing a solution, one survey respondent indicated:

> Don't focus on documenting individual methods/classes so much as how they are used together to achieve specific goals. [Survey]

In terms of how specific the examples are, one informant commented "I guess examples are better if they are fairly general, not in what they're trying to do, but in showing you how to use the interfaces without a specific purpose." [Dev 4]

Finally, one type of example that seemed to be consistently missing were examples showing code to integrate multiple APIs [Dev 1].

Implication 2 *Small examples that nevertheless demonstrate API usage patterns involving more than one method call will be more useful than single-call examples.*

Recommendations Participants perceived code examples not only as demonstrations ("this is how you can use the API"), but as recommendations ("this is how you *should* use the API"). [Dev 5]

> Definitely, samples[7] are the thing that helps the most, because you get some guidance on how to use the API. [Dev 12]

[6]Complexity is informally defined as a combination of length of the example and amount of interaction with the API.

[7]Participants routinely used the terms "sample" and "example" interchangeably. In the quotes, the term "sample" does not necessarily refer to code samples as defined above.

In the eyes of informants, good examples provide a guide to the "best practices" for using the API [Dev 20, 21]

> One of the difficulties of writing a sample, I think, is [...] that you want to convey enough to the end-user, the application developer, how to use the API. [Lead 25]

In fact, examples were also seen by some as the *best* way to convey this type of knowledge [Dev 2; Lead 25].

Because they are perceived as recommendations, code examples are assessed in terms of how authoritative and credible they are. The two main factors that impact this quality measure are knowledge of, and respect for, the author of the code, and evidence that the example is up-to-date [Dev 4, 10, 11, 22, 24, 26; Lead 18; Survey]. For this reason, developers with access to production code generally chose this option [Dev 10, 18, 26; Lead 8]. Informants who retrieved examples from the Internet were at times suspicious of their quality [Dev 4, 11, 22, 24].

> Those examples would still run but I don't think they have the most recent way of doing stuff. [Dev 11]

Implication 3 *Examples should be developed to demonstrate "best practices" for using an API.*

In this context we consider best practices to mean the most effective way to combine API elements, as opposed to general best practices of programming (such as extensive input validation and error checking, consistent naming, etc.). We realize that in certain cases developing examples that follow *all* best practices for an API would go against the goal of brevity described above.

An additional observation is that annotating examples with an explicit mention of "authority" would save developers the trouble of continually having to figure it out.

5.3 Matching APIs with Scenarios

> If it's not clear how I match APIs with their scenarios, if I need to draw a circle on the screen, and I don't see something that clearly says, "this is how you draw", I will say that's complex. [Dev 12]

A central challenge when learning APIs is discovering how to match "scenarios" (a desired chunk of functionality, such as "drawing a circle on the screen"), with the API elements that support this scenario. Although information about intent (see above) can help meet the conceptual gap challenge [Survey], we discovered that informants had high expectations for the API documentation to help them meet this challenge. Over 24 participants commented on this challenge.

> What I would really love to see would have been something that says "if you're trying to do something like this, this is the type of pattern you should follow...". [the documentation library] has great documentation on "here's all the methods on this control", but what I always have trouble finding is "why would I want to call any one of them over any one of the others?" [Dev 13]

In addition to the fact that information on how to match scenarios to APIs might be missing [Arch 3; Dev 4, 10; Survey] or present but misleading [Dev 19] this challenge is complicated by the fact that the ability to map scenarios to APIs can be made worse by the structure of the API [Dev 23; Lead 8, 25; Survey] and or the background of the developers [Dev 1, 7, 10, 12, 13, 21, 22, 27; Lead 17; Survey].

The elements of API structure that impact an informant's ability to map to scenarios are the API's abstraction level and work-step units (Clarke 2004). For example, the practice of heavily overloading constructors for an object makes it more difficult for developers to select the proper way to use the object of their need [Lead 8]. Extensive webs of dependencies between objects also create problems for developers:

> The type was dependent on many other types, which made it more difficult for me to know how things work in the big picture. [Survey]

Regarding the background, although knowledge of a previous API in a similar domain should help developers learn new ones, in some cases this knowledge caused problems:

> Maybe the fact that I already knew a lot of other technologies and was always trying to map what I knew to what I was seeing interfered with my learning. [Dev 7]

We found that our informants had high expectations that APIs for the same application domain (e.g., user interface components) would exhibit a similar design, and were surprised and confused when they were not able to map scenarios they knew how to implement with one API onto a new API or did not find explicit information to that effect [Survey].

Implication 4 *Matching scenarios to API elements is an area where documentation support is perceived as particularly helpful, and can compensate for hard-to-understand API structure. Scenario-matching documentation should take into account previous generations of APIs in the same domain (for example by including "bridge" documentation catering to experts with other APIs).*

5.4 Penetrability

> So we thought we have terrible leaks, it's really bad. Lots of debugging, we actually went through the source code of [the API] to look at it and understand. No, no, no, it's normal. [Arch 3]

Penetrability is one of Clarke's 12 cognitive dimensions for evaluating API usability. It is defined as "how the API facilitates exploration, analysis, and understanding of its components" (Clarke 2004, p. S7). When building APIs, designers must walk a fine line between an over-exposure of the APIs internal elements (which would violate the principle of information hiding (Parnas 1972)), and a design that makes the behavior of the API completely impenetrable, and thus hard to learn. API designers have discussed this challenge on many occasions (Fowler 2002; Larman 2001). Many participants in our study felt the need to understand or visualize how the API works internally (while recognizing that some parts must remain hidden).

Over 18 participants commented on penetrability issues. Our data sheds light on the distinction between what must be transparent and what must remain opaque in API design.

Participants have referred to their difficulty to reason about *performance aspects of the API* that had direct consequences on the behavior of their client application [Arch 3; Lead 9; Dev 10, 27; Survey].

> I agree that encapsulation and hiding the internals is a good design choice for the API. [...] But if you want to know how to use an API, understanding that making a method call is actually going to create five threads, it's going to start these processes... So you do need to know the internals. [Lead 9]

Other prominent aspects mentioned by informants were a desire to mentally predict the *visual rendering of UI or graphics operations* [Dev 7, 13; Lead 25] and clearer error-handling behavior [Dev 7, 10, 12, 13, 22, 23; Lead 17].

A special case of penetrability issue occurs when methods transparently perform multiple high-level tasks (e.g., load *and* parse a file in one operation). This type of API design can lead to misinterpretations [Dev 20] and should be explicitly documented.

A final, more subtle factor making API behavior impenetrable is when design decisions result in API behavior that varies depending on classes of inputs. This type of behavior was commonly referred to as "magic" [Arch 3; Dev 12, 20, 26].

> 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" [Dev 12].

This type of behavior was consistently perceived as an obstacle:

> And what I mean by awkward is, certain behavior, you would only understand, if you know a bit about how the implementation behind them actually works. So, that's one of the things I didn't quite like. [...] Usually when I use an API, the main reason you use an API is to abstract something so you don't need to know the implementation details in the back. And for the [...] APIs there's a few things that, unless you really know how the whole thing works, it's hard to predict what the outcome is. And unpredictable APIs to me are the biggest problem. If you don't know exactly how what you're calling is going to react, or what to expect, this is basically the biggest problem. [Dev 20]

As a coping strategy, most informants who talked about penetrability issues mentioned that the usual workaround for impenetrable API behavior was inspecting the source code [Arch 3; Dev 10, 12, 22, 27; Lead 9, 18] and micro-experimentation with the API [Arch 3; Dev 1, 13; Lead 18]. The first strategy can be problematic because it can be very ineffective to do so, and in some cases developers may not have access to the source code of the API. The second strategy can lead to decreased productivity if the experimentation has less to do with learning the API and more with reverse-engineering its behavior.

Implication 5 *Explicit documentation on the performance consequences of API elements, better descriptions of error handling, and, for methods triggering significant chunks of behavior, explanations of the abstract operations of the API, all have*

potential to decrease the amount of source code inspection and iterative prototyping needed to learn the API.

5.5 Documentation Format

> Some [documentation] pages talk about one method or have very little additional information and waste your time navigating through them. [Survey]

Many informants discussed the frustration of encountering boilerplate documentation that merely rehashes the name of an API method, bloats the presentation with derived information such as inherited members [Survey] or provides overly trivial examples that simply show a single method call [Dev 1, 5, 12, 14, 19, 21, 22, 27; Lead 8, 9, 18, 25; Survey].

API documentation guidelines (such as those of Bloch 2006) and coding standards generally emphasize the thoroughness of member-level API documentation. Our study reveals that focusing on completeness of member-level documentation can backfire not only because it takes away effort from the production of conceptual-level documentation, but also because inappropriate member-level documentation can have a negative effect on developer productivity. When, for example, developers choose to navigate to a method documentation page and find nothing, the time they took to navigate to the documentation and assess it is wasted. Specifically, since most modern IDEs provide in-line descriptions of all methods in a type and the corresponding documentation, the act of seeking additional information is generally conscious:

> But when I go to the documentation, I'm obviously saying "oh, that was not enough, I need a little more information, explain to me with an example, or, just tell me what it does...". Something more than an expansion of the [method signature]. [Lead 18]

Implication 6 *Navigating to a documentation page is an implicit query. Boilerplate member-level documentation will often not answer the query, and waste developer's time.*

> It should be possible to browse API documentation like a book. When trying to learn an API it is nice to browse through it to get a more concrete feel as to what it does. Basically I want to be able to repeatedly hit a "next" button. [Survey].

Many of our previous recommendations stress the importance of conceptual-level documents to facilitate API learnability. But what is the best way to present API documentation? Electronic documentation, such as that for APIs, can come in a wide spectrum of fragmentation, from a long, single document to a web of tiny, ultra-focused articles.

On this question, our participants came out strongly in favor of a relatively continuous, unfragmented presentation. We inferred the following justifications for this position:

A Continuous Narrative Provides a Single Place to Look and an Obvious Place to Begin Many respondents were overwhelmed by the documentation on some APIs

and could not easily figure out how to approach it [Survey]. Books (which are by default continuous), were seen as especially useful for the first approach [Dev 1, 2, 5, 7, 12, 14, 21, 23; Lead 9].

> It wasn't a reference book it was almost like a novel. You could read from the beginning to the end and it just described in a story-like fashion how to build [the application]. So I thought that was really useful for getting the concepts. [Dev 7]

Fragmentation Makes the Information Less Discoverable Other participants described cases where they wasted a lot of time because they had not found the information they needed (although it existed) [Arch 3; Dev 12, 19, 27; Survey].

> So, that's an example of, you know, 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]. [Arch 3]

The Hyperlink Structure can be Disconcerting Many participants mentioned being confused by the navigation structure. One provided the clue that misalignment with the code structure may be problematic:

> There's a structure to the code and when documentation doesn't follow that structure it's annoying. [Dev 4]

Some Developers are Used to Learning from Continuous Documents Another interesting trend we noticed is that books were favored by older and more experienced developers [Dev 5, 14; Lead 6, 9, 25].[8] This finding may point to different learning styles between different generations of developers.

> But one of the things [that is] lacking, especially now that there is so much information on the web, is: In the old days people would read a book, and you could kind of go from Chapter 1 to the end, and it would take you through the whole way of thinking. [Lead 6]

Implication 7 *Some developers will look for a coherent, linear presentation of the documentation (in particular for point-of-entry overviews); Fragmented collections of hyperlinked articles can be overwhelming.*

Comments on the practical implications of this recommendations are best left to a survey respondent:

> Improve locality of information [...] People who need to read API documentation are used to searching in large bodies of text, and navigation in a long page is not new to them [...] It is a lot easier and faster to scroll back and forth on the same long web page and use the browser's search function than to navigate among a dozen related pages. [Survey]

[8] Developer 14 was relatively new to the profession, but older than average in age.

6 Experimental Critique

Our use of multiple sources of evidence helps paint a reliable picture of our phenomenon of interest: API learning obstacles. Nevertheless, each of our three experimental phases is subject to inherent threats that must be considered when interpreting the results.

Our **exploratory survey** is based on a true random sample of a well-known population (Microsoft-Redmond developers). Although we obtained a low response rate, we found that the distribution of developer across job titles in the sample very closely matched that of the population (Robillard 2009). For this survey the main risk is non-response bias. In our case this risk is not significant because we were not attempting to infer a property of a population (e.g., the percentage of developers who had learned an API), but to explore the variety of API learning obstacles. If fact, our questionnaire was naturally irrelevant to all developers who had not recently learned an API. The actual experimental threat for this survey is therefore that obstacles faced by non-respondents somehow were not represented in Table 1. Given that our follow-up survey did not uncover any significant additional types of obstacles, we judge this risk to be small.

Our **follow-up survey** is also based on a true random sample of a well-known population (in this case of the entire population of Microsoft developers). Although our response rate was twice as high, non-response bias also needs to be considered. Although the same arguments as above can be made about the relevance of the survey, in this case the risk is that non-respondents with API learning experience may have had experiences where API learning was easy (and thus not worth reporting). To mitigate this risk, we did not base our analysis on direct interpretations of the severity ratios values in Fig. 1. Instead, all of our interpretations from this survey are based on relations between obstacle types.

For the **interviews** (and the free-form survey responses), we followed a qualitative approach. In qualitative research, one important concern is the role of the researcher. Because results are based on an interpretation of detailed, context-rich data, it is understood that this interpretation will be informed by the researcher's experience. In qualitative research, investigator bias is not a threat, but a desired attribute: the investigators are the ones selecting the main themes for analysis, identifying the relevance of data, and developing the hypotheses. This research was conducted while both authors were full-time employees at Microsoft. We were thus able to understand and interpret the survey responses and interviews in the exact corporate environment where they applied.

Nevertheless, it remains important to consider threats to the quality and credibility of the results. Corbin and Strauss (2007) propose a number of criteria to evaluate the quality of grounded theory research, which are generally applicable to qualitative analysis. Prominent amongst these criteria is the question of "fit". In other words, how can we ensure that the results resonate with the professionals: both our participants, and other stakeholders for which the research was intended. To meet this criterion, we linked every observation with the related qualitative evidence (Section 5). Our follow-up survey also included two open-ended questions that would have detected a mismatch between our interpretation of the data and developers' experience. For both the categories of learning contexts (Table 3) and the types of obstacles (Section 3.4), our survey asked respondents whether an additional

context/type of obstacle applied to them. We recorded only 3 responses for other contexts (two of which were simply alternative ways to state proposed contexts). In the case of other obstacles, we recorded 68 responses, but a detailed analysis showed that almost all of them were simply detailed explanations that unequivocally fell within a proposed obstacle type. Overall, our follow-up survey did not reveal any significant new context or obstacle type. To confirm these conclusions with a different source of evidence, we also procured advanced reports of our analysis to important stakeholders within the organization, who commented positively and in some cases vigorously acknowledged having had similar experiences.

Finally, our research was conducted within a single organization, and there are a few notable ways in which our population may not reflect the population of software developers at large. First, most the participants had access to either the teams who created the APIs they were learning (e.g., through email lists) or to the source code implementing the API, or both. This is often the case in large corporate settings and in the open source community, but not in companies that rely on components from outside suppliers, where developers may use strategies not represented in this study. Second, the sampled population is fairly experienced (average of 9.8 years of professional experience), and there may be types of obstacles faced by less experienced developers (in particular, students and new hires) that may not be represented in this study.

7 Conclusions

When professional developers like the ones in our studies learn a new API, they struggle not so much in the mechanics of using the API, but in understanding how the API relates *upwards* towards its problem domain and *downwards* towards its implementation. In the *upwards* direction, the study found that developers need help mapping desired scenarios in the problem domain to the content of the API, and in understanding what scenarios or usage patterns the API provider intends and does not intend to support. In the *downwards* direction, developers want to understand how the API's implementation consumes resources, reports errors and has side effects (e.g., rendering to the screen). Given the high-level, conceptual nature of these issues, it is not surprising that developers prefer centralized, narrative presentations to narrow, hyperlinked ones. This applies not only to text, but to code examples, where showing a pattern of related calls is preferred to illustrations of individual methods.

Based on our qualitative findings, we derived seven implications for prioritizing API documentation efforts across five dimensions related to many different aspects of an API's life-cycle (from capturing intent during the early phases to choosing the best way to format the documentation). An immediate corollary is that responsibility for documenting an API cannot be cleanly separated from the responsibility for designing the API, even though different skills are involved. In particular, only the team designing the API controls issues like supported scenarios, design intent, and implicit implementation behavior, which are critical to how easily the API can be learned. Ultimately, our findings could be used to help coordinate the work of API design and documentation teams, so that resources to help developers learn APIs can be produced as cost-effectively as possible.

Acknowledgements We thank the anonymous participants in our study for generously contributing their time to this project, and the members of the Human Interactions in Programming group at Microsoft Research for feedback and discussions on the work reported here. Additional thanks to E. Duala-Ekoko, U. Farooq, G. Murphy, A. Ying, T. Ratchford, and the anonymous reviewers for feedback on the paper. We are also grateful to Jose Correa of the McGill University Statistical Consulting Service for his assistance with a statistical test.

Appendix A: Exploratory Survey

Exact text of the survey described in Section 3.2 for the questions used in the research. This instrument also included questions whose answers were not included in the research reported in this article. These questions have been removed.

MSR API Learning Survey

The Human Interactions in Programming (HIP) group in Microsoft Research is conducting this survey to gather some initial data on Microsoft developers' experiences with learning APIs. This data will be used in part to design tools and resources to help developers learn new APIs effectively. Your experiences and opinions would make a big difference in this research. Any personally-identifying information that you provide will be accessible only to the research team. For more information about this survey or the corresponding project, contact [the investigator].

This survey will take about 5 min to complete. Please refresh your memory about the API you learned the most recently. Published results will be anonymous. For your input to be useful, you must complete the survey by [deadline].

After taking the survey click "Submit" to save your changes. This survey is not anonymous

1. What is your primary job area?
2. How many total years of professional experience related to your job do you have?
3. How many of these years were at Microsoft?

This part of the survey will concern only your most recent experience with an API that you had to learn in your professional capacity. For this survey, consider that an API is a reusable set of program elements (classes, methods, functions, etc.) that are distributed and used together to provide higher-level functionality.

4. Name the last publicly-released API you learned:
5. What particular area of this API did you learn?
6. How many months ago did you start learning the API?
7. How many months ago did you last use the API?
8. How many hours in total would you estimate you've spent learning this API?
9. Were you also learning the programming language used to access the API? **[Yes, No]**
10. Were you familiar with the application domain of the API you learned? (For example, if you were learning WPF, were you already familiar with the creation of GUIs?) **[Yes, No]**

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.

14. Obstacle 1:
15. Obstacle 2:
16. Obstacle 3:
17. Do you have any additional comments about learning APIs?
18. Would you be willing to participate in future research about this topic? **[Yes, No]**

Appendix B: Follow-up Survey

Exact text of the survey described in Section 3.4 for the questions used in the research. This instrument also included questions whose answers were not included in the research reported in this article. These questions have been removed.

API Learning Survey

The goal of this survey is to learn about your experience interacting with public APIs and to collect your opinions on how to improve API documentation. You can only complete this survey if you have had experience learning a public API that you remember enough to comment on. By "learning an API", we mean using an API for the first time and engaging in activities that increase your knowledge of how this API works.

The survey should take less than 10 minutes to complete. As our appreciation for your time, participants who complete the survey will be entered in a draw for one $250 [on-line store] gift certificate.

The contest rules can be found at [internal URL]. After taking the survey click "Submit" to save your changes. This survey is not anonymous

1. How many years of professional experience do you have (decimals ok):

API Learning Obstacles

This part of the survey applies only to the last public API you have learned.

2. What is the last public API you have learned? Pick the best match or choose "Other" (please specify below).

   [Azure; LINQ; Entity Framework; WPF; WCF; WF; CardSpace; Winforms; ASP.NET; ADO.NET; .NET Base Classes; Silverlight; XNA; Win32 (Admin); Win32 (Diagnostics); Win32 (Graphics); Win32 (Networking); Win32 (Security); Win32 (System Services); Win32 (UI); Other].

3. If you chose "Other", please specify:
4. What programming language did you primarily use to access this API? **[C/C++; C#; Visual Basic; JScript, JavaScript, or other web language; Other]**
5. If you chose "Other", please specify:

6. What was the main reason for learning this API? **[Your team started owning it; You needed to use it extensively as part of your work; You needed to use it to complete some specific tasks, but these were not the main part of your work; You learned the API to experiment with new technology related to your job; You learned the API for a hobby project, or for a side project not critical to your work; Other (please specify)]**
7. Other:

For each type of obstacle described below, please rate how severe this type of obstacle was in your experience learning the API you mentioned above. [For questions 8–12 and 14, possible answers were as follows:]

[Blocker (Led to work being abandoned or a different APIs used, could not be realistically overcome); Very Severe (Led to significant delays and frustration, very difficult to overcome); Severe (Led to delays and frustration, difficult to overcome); Moderate (Led to delays and/or frustration, but could be overcome without excessive difficulty); Trivial (Was easy to overcome); This was not an obstacle at all]

8. When learning the API you mentioned above, how severe was this obstacle: **Your background was not adapted to learning the new API.** For example: you were not familiar with the programming language or application domain, your previous knowledge of a similar API (or a previous version of the API) made it confusing to learn the new API.
9. When learning the API you mentioned above, how severe was this obstacle: **The way the API was structured or designed made it difficult to understand.** For example: it was not clear how to instantiate an object, there were too many abstract classes, the names did not make sense.
10. When learning the API you mentioned above, how severe was this obstacle: **The technical environment made it difficult to use the API.** By technical environment, we mean any technical aspects not directly related to the design of the API itself. For example, the tools did not work well, the API required extensive infrastructure to test, you could not get the builds to work.
11. When learning the API you mentioned above, how severe was this obstacle: **Specific member-level usage details were not documented.** For example, description of parameters, error codes.
12. When learning the API you mentioned above, how severe was this obstacle: **You did not find conceptual-level information explaining how to use the API.** Consider "conceptual-level" information to mean any type of information you need to use the API correctly that is not typically associated with particular API members (classes/methods/functions). For example, description of required concepts, the API's execution model, non-trivial code examples, usage patterns, best practices, mappings between scenarios and API members.
13. If you encountered a type of obstacle not included in the above list, please describe it here:
14. If you described an additional obstacle, how severe was this obstacle?
29. Do you have any additional ideas for improving high-level API documentation that could help developers learn how to use APIs more efficiently? If so please describe it here.

Appendix C: Quantitative Survey Results

2. What is the last public API you have learned? Pick the best match or choose "Other" (please specify below).

API	Respondents
Other	54
LINQ	36
WPF	34
WCF	32
.NET Base Classes	32
Silverlight	24
Win32 (System Services)	17
Win32 (Networking)	16
ASP.NET	14
Win32 (Security)	14
Winforms	10
Azure	10
Win32 (UI)	8
Win32 (Graphics)	8
XNA	8
WF	5
Entity Framework	4
ADO.NET	4
Win32 (Diagnostics)	3
Win32 (Admin)	1

3. If you chose "Other", please specify [Coded by categories]:

API category	Respondents
Operating System	16
Development Tools	12
Web	7
Graphics/media	7
Networking	5
User Interface	3
Unknown	3
Business Applications	1

4. What programming language did you primarily use to access this API?

Language	Respondents
C#	229
C/C++	97
JScript, JavaScript, or other web language	2
Other	5
Visual Basic	1

6. What was the main reason for learning this API?

API learning context	Respondents (%)
You needed to use it extensively as part of your work	47
You needed to use it to complete some specific tasks, but these were not the main part of your work	22
You learned the API to experiment with new technology related to your job	17
You learned the API for a hobby project, or for a side project not critical to your work	10
Your team started owning it	3
Other (please specify)	1

8–12. API Learning Obstacle Severity

	Blocker	Very severe	Severe	Moderate	Trivial	Not an obstacle
High-level documentation	1	9	27	37	17	10
Low-level documentation	2	9	13	37	21	18
Structure	0	4	11	43	27	15
Technical environment	0	3	10	27	32	27
Background	0	1	6	32	29	32

References

Beaton J, Jeong SY, Xie Y, Stylos J, Myers BA (2008) Usability challenges for enterprise service-oriented architecture APIs. In: Proc. IEEE symp. visual languages and human-centric computing, pp 193–196

Berglund E (2003) Designing electronic reference documentation for software component libraries. J Syst Softw 68(1):65–75

Bloch J (2006) How to design a good API and why it matters. In: Companion to the 21st ACM SIGPLAN symposium on object-oriented programming systems, languages, and applications, pp 505–506

Bore C, Bore S (2005) Profiling software API usability for consumer electronics. In: Digest of int'l conf. on consumer electronics, pp 155–156

Brandt J, Guo PJ, Lewenstein J, Dontcheva M, Klemmer SR (2009) Two studies of opportunistic programming: interleaving web foraging, learning, and writing code. In: Proc. 27th int'l conf. on human factors in computing systems, pp 1589–1598

Clarke S (2004) Measuring API usability. Dr Dobb's Journal Special Windows/NET Supplement

Corbin J, Strauss A (2007) Basics of qualitative research: techniques and procedures for developing grounded theory. Sage Publications

Creswell JW (2007) Qualitative inquiry and research design: choosing among five approaches. Sage Publications

Cwalina K, Abrams B (2009) Framework design guidelines: conventions, idioms, and patterns for reusable .NET Libraries, 2nd edn. Addison-Wesley

des Rivières J (2004) Eclipse APIs: lines in the sand. EclipseCon Technical Talk. http://www.eclipse.org/eclipse/development/apis/Eclipse-APIs-Lines-in-the-Sand.pdf
Ellis B, Stylos J, Myers B (2007) The factory pattern in API design: a usability evaluation. In: Proc. 29th int'l conf. on software engineering, pp 302–312. doi:10.1109/ICSE.2007.85
Feilkas M, Ratiu D (2008) Ensuring well-behaved usage of APIs through syntactic constraints. In: Proc. 16th int'l conf. on program comprehension, pp 248–253
Fowler M (2002) Public versus published interfaces. IEEE Softw 19(2):18–19
Holmes R, Walker RJ, Murphy GC (2006) Approximate structural context matching: an approach to recommend relevant examples. IEEE Trans Softw Eng 32(12):952–970
Hou D (2008) Investigating the effects of framework design knowledge in example-based framework learning. In: Proc. 24th int'l conf. on software maintenance, pp 37–46
Hou D, Wong K, Hoover JH (2005) What can programmer questions tell us about frameworks? In: Proc. 13th int'l workshop on program comprehension, pp 87–96
Jeong SY, Xie Y, Beaton J, Myers BA, Stylos J, Ehret R, Karstens J, Efeoglu A, Busse DK (2009) Improving documentation for eSOA APIs through user studies. In: Proc. 2nd int'l symp. on end-user development. LNCS, vol 5435. Springer, pp 86–105
Knowles MS, Holton III EF, Swanson RA (2005) The adult learner, 6th edn. Butterworth-Heinemann
Ko AJ, Myers BA, Aung HH (2004) Six learning barriers in end-user programming systems. In: Proc. symp. on visual languages and human centric computing, pp 199–206
Ko AJ, DeLine R, Venolia G (2007) Information needs in collocated software development teams. In: Proc. 29th int'l conf. on software engineering, pp 344–353
Larman C (2001) Protected variation: the importance of being closed. IEEE Softw 18(3):89–91
Lutters WG, Seaman CB (2007) Revealing actual documentation usage in software maintenance through war stories. Inf Softw Technol 49:576–587
McLellan SG, Roesler AW, Tempest JT, Spinuzzi CI (1998) Building more usable APIs. IEEE Softw 15(3):78–86
Nykaza J, Messinger R, Boehme F, Norman CL, Mace M, Gordon M (2002) What programmers really want: results of a needs assessment for SDK documentation. In: Proc. 20th annual ACM SIGDOC int'l conf. on computer documentation, pp 133–141
Olson M, Hergenhahn B (2008) Introduction to the theories of learning, 8th edn. Prentice Hall
Parnas DL (1972) On the criteria to be used in decomposing systems into modules. Commun ACM 15(12):1053–1058
Robillard MP (2009) What makes APIs hard to learn? The answers of developers. IEEE Softw (November/December):27–34
Rosson MB, Carroll JM (1996) The reuse of uses in Smalltalk programming. ACM Trans Comput-Hum Interact 3(3):219–253. doi:10.1145/234526.234530
Seaman CB (2002) The information gathering strategies of software maintainers. In: Proc. int'l conf. on software maintenance, pp 141–149
Shull F, Lanubile F, Basili VR (2000) Investigating reading techniques for object-oriented framework learning. IEEE Trans Softw Eng 26(11):1101–1118
Sillito J, Murphy GC, Volder KD (2008) Asking and answering questions during a programming change task. IEEE Trans Softw Eng 34(4):434–451
Stylos J, Clarke S (2007) Usability implications of requiring parameters in objects' constructors. In: Proc. 29th int'l conf. on software engineering, pp 529–539
Stylos J, Myers BA (2007) Mapping the space of API design decisions. In: Proc. symp. on visual languages and human-centric computing, pp 50–60
Stylos J, Myers BA (2008) Implications of method placement on api learnability. In: Proc. 16th ACM SIGSOFT int'l symp. on the foundations of software engineering, pp 105–112
Stylos J, Graf B, Busse DK, Ziegler C, Karstens REJ (2008) A case study of API redesign for improved usability. In: Proc. symp. on visual languages and human-centric computing, pp 189–192
Tulach J (2008) Practical API Design: confessions of a Java framework architect. APress
Weiss RS (1994) Learning from strangers: the art and method of qualitative interview studies. The Free Press
Westfall PH, Tobias RD, Rom D, Wolfinger RD, Hochberg Y (1999) Multiple comparisons and multiple tests using the SAS system. SAS Institute Inc., Cary, NC

Martin P. Robillard is an Associate Professor of Computer Science at McGill University, where he heads the Software Evolution Research Group (SWEVO). His research focuses on the automated analysis of software development artifacts to support software evolution and maintenance. He received his Ph.D. and M.Sc. in Computer Science from the University of British Columbia and a B.Eng. from École Polytechnique de Montréal. http://www.cs.mcgill.ca/~martin.

Robert DeLine is a principal researcher at Microsoft Research, working at the intersection of software engineering and human-computer interaction. His research group designs development tools in a user-centered fashion: they conduct studies of development teams to understand their work practice and prototype tools to improve that practice. He received his PhD from Carnegie Mellon University in 1999 and his BS/MS from the University of Virginia in 1993. http://research.microsoft.com/~rdeline.