When Not to Comment

Questions and Tradeoffs with API Documentation for C++ Projects

Andrew Head*
UC Berkeley
andrewhead@berkeley.edu

Caitlin Sadowski
Google, Inc.
supertri@google.com

Emerson Murphy-Hill*
NC State University
emerson@csc.ncsu.edu

Andrea Knight
Google, Inc.
aknight@google.com

ABSTRACT

Without usable and accurate documentation of how to use an API, developers can find themselves deterred from reusing relevant code. In C++, one place developers can find documentation is in a header file. When information is missing, they may look at the corresponding implementation code. To understand what's missing from C++ API documentation and the factors influencing whether it will be fixed, we conducted a mixed-methods study involving two experience sampling surveys with hundreds of developers at the moment they visited implementation code, interviews with 18 of those developers, and interviews with 8 API maintainers. In many cases, updating documentation may provide only limited value for developers, while requiring effort maintainers don't want to invest. We identify a set of questions maintainers and tool developers should consider when improving API-level documentation.

ACM Reference Format:
Andrew Head, Caitlin Sadowski, Emerson Murphy-Hill, and Andrea Knight. 2018. When Not to Comment. In *Proceedings of ICSE '18: 40th International Conference on Software Engineering , Gothenburg, Sweden, May 27-June 3, 2018 (ICSE '18),* 11 pages.
https://doi.org/10.1145/3180155.3180176

1 INTRODUCTION

Seeking information is a substantial part of day-to-day programming work. Both professional [32] and hobbyist [5] developers frequently search for code examples. For routine coding, debugging, and maintenance tasks, developers spend a large portion of the time navigating and searching existing code [14, 25]. Developers report that understanding existing code is one of the most time-consuming parts of software development, and that understanding the rationale behind code is a serious challenge [16].

Developers sometimes find that missing or insufficient documentation can block them from using an API [29, 30, 38]. At Google, 274 of 601 surveyed developers reported that they encountered "Missing/poor documentation for an API" their project depended on in the last 6 months. In the words of one of the interviewees in our study, missing information is a "standard fact of life." While the literature demonstrates documentation is often incorrect or missing, it remains unclear what information is missing, what it would take for maintainers to add or update this information, and whether improved documentation would help developers in typical situations where documentation is insufficient.

In this paper, we report the results of a mixed-methods study of what developers are looking for when they leave an API specification to look at implementation code, and maintainers' perspectives about updating documentation to answer these developers' questions. To make this study feasible, we focus on C++ APIs where developers could easily search both the API specification and implementation code, a typical context for professional and open source development. We instrumented Google's internal code search tool with an in-situ survey to find out what questions developers were asking when they navigated from a `.h` "header" file containing API declarations, to a corresponding `.cc` "implementation" file containing API definitions. We interviewed developers who made these transitions. With stories and questions from these developers, we interviewed maintainers for these and other APIs to see whether they thought the questions represented missing documentation, and whether they would update the documentation.

Concretely, the findings from this study were as follows. First, a minority (between around 5–25%) of visits to implementation files were to learn about API usage. Some visits were for questions that possibly should have been answered in the documentation (input values, return values). Other visits were for questions that aren't often answered in low-level documentation (hidden contracts, implementation details, side effects).

Second, respondents frequently reported that it would have been most convenient to find answers to these questions in header files, instead of implementation code or documents on our Markdown server. This was the case even for some questions typically left out of API-level documentation. However, developers we interviewed sometimes preferred to find answers in implementation code, which could be more accurate and quick enough to read.

Finally, maintainers were reluctant to answer searchers' questions for several reasons, falling into themes of it not being the right time to document, and keeping explanations minimal.

The main contributions of this paper are: 1) a set of questions that developers are seeking to answer about C++ APIs when viewing implementation files, and 2) trade-offs for maintainers to consider when updating documentation to answer searchers' questions. At Google, developers and technical writers invest effort in choosing and organizing content to document APIs and tools, and they need to decide how to prioritize that effort. Software engineering researchers are developing tools to mine API information and serve it in helpful places (e.g., [24, 37]). We believe our study helps inform what information should be surfaced, and the software development context that would determine the acceptance and value of tools and strategies to improve API documentation.

*Authors did this research while an Intern and Visiting Scientist at Google, repectively.

ICSE '18, May 27-June 3, 2018, Gothenburg, Sweden
© 2018 Copyright held by the owner/author(s).
ACM ISBN 978-1-4503-5638-1/18/05.
https://doi.org/10.1145/3180155.3180176

2 RELATED WORK

2.1 How Developers Reference Documentation

The literature suggests that developers face a number of challenges when looking for information in API documentation. From interviews with developers, Lethbridge et al. reported that documentation can be out-of-date, poorly written, and some systems may have too much of it [17]. In a survey of professional developers, Robillard found that inadequate learning resources were a common obstacle for learning about APIs [29]. Robillard and DeLine interviewed and surveyed developers about learning obstacles, recommending that common obstacles could be avoided if the intent of an API is documented, code examples cover non-trivial use cases and best practices, and documentation helps developers find API elements for their tasks, understand relevant parts of APIs' internal behavior, and avoids fragmentation [30]. In another survey, Uddin and Robillard found that developers were more likely to report incompleteness, ambiguity, and incorrectness of documentation as "blockers" or "severe" issues than other documentation issues [38].

Prior studies indicate that the perceived and actual utility of comments varies based on where they appear and who is reading them. Roehm et al. observed that most professional developers in their study reported getting their main information from source code and inline comments rather than documentation, as documentation could be sparse or inaccurate [31]. Salviulo et al. found that young professional developers, compared to student programmers, were less likely to consult comments during in-lab code comprehension tasks [33]. Borstler et al. showed that although source code with "good" comments was reported as more readable, these comments didn't appear to impact actual comprehension [4].

In relation to these past findings about documentation, this study provides a perspective on what information may be missing from low-level API documentation, and an understanding of costs and benefits of improving missing documentation in header files.

2.2 Information Foraging in Code

Software engineering is an information intensive task, requiring developers to gather information from peers, code, design documentation, and more [16]. Recently, information foraging theory has been used to describe how software developers search for information, and tradeoffs in designing software engineering tools (see [11] for a primer). In information foraging, a developer searches to satisfy a *goal* (e.g., a piece of code with desired functionality). They locate information *patches* to inspect for features satisfying their goal, called *prey*. Developers make choices to maximize the value of information they find, and minimize the cost of navigation. To choose where to look, developers estimate the expected value of information within a patch, and the cost of finding it. When there is a mismatch between the expected and actual costs and values, a developer could benefit from a better strategy or tools. For some tasks, foraging consistently delivers less value than expected: in one study, as many as 50% of navigation choices yielded less value than expected, and 40% cost more than expected [25].

The software engineering research community has elicited many information goals as concrete questions developers ask as they write and maintain code. Sillito et al. describe 44 such questions arising during software change tasks, which belonged to four categories: finding initial focus points, building on these points, building a model connecting found information, and integrating an understanding across such models [34]. Sadowski et al. observed developers' queries to a code search tool, grouping them into questions of how to do something, what code does, why it is behaving the way it is, finding where something is, who did something, and when they did it [32]. Duala-Ekoko and Robillard describe 20 questions developers ask about APIs, based on talk-aloud data of developers performing in-lab coding tasks [10].

Developers adapt their foraging strategies to their goals. In one study, developers inspected code more when collecting details about types, and *searched code* more to find initial locations relevant to their debugging task [26]. From a foraging perspective, our study seeks to characterize a specific foraging strategy: looking for API usage information in implementation code. We report a set of information goals developers have when they look at implementation code for an API, and describe factors influencing the cost and value of finding answers to API usage information in code vs. documentation in a real-world software development setting.

2.3 Choosing What to Document

When considering how maintainers decide what to document, we build on prior studies of writing both unofficial and official documentation. Parnin et al. interviewed programming bloggers, finding that they face challenges keeping up with community contributions and preparing examples [23]. Dagenais and Robillard spoke with contributors to open source projects, finding that maintainers' motivation to write and maintain documentation could be low, though maintainers may update documentation in response to community contributions [8]. Our study can be seen as providing context about the choices maintainers make when writing and considering making updates to low-level documentation.

Maalej and Robillard described twelve types of knowledge in reference documentation, including purpose and usage examples [18]. Padioleau et al. observed what code comments describe by classifying hundreds of comments: 52.6% went beyond explaining the code, to describe types, relationships between code entities, aspects of code evolution, synchronization, and more [22]. In relation to this work, our study provides some examples of questions that were not answered in API-level comments.

Recent research has proposed tools for automatically improving documentation. Such tools synthesize code examples [6, 21], generate method [19, 35] and parameter descriptions [36], mine API usage patterns (e.g., [20, 40]), collect insightful sentences describing APIs [37], and identify improvable documentation [39, 41]. Researchers have extended development environments to reveal important usage information [9] and integrate online documentation [27] and web search history [12]. This study provides an understanding of developer questions and perspectives on documentation that we hope can help motivate the design of such tools.

3 METHODS

3.1 Overview of Mixed Methods

We used a mixed-methods approach to understand what developers are looking for in implementation code and whether more information should be added to documentation. Through Code Search logs analysis, we were able to interview and survey developers at moments where we suspected they had an unsatisfied information need. Our mixed methods approach includes:

- **Logs analysis**: We analyzed usage logs for the Code Search tool to understand documentation usage and identify transitions from headers to implementation code.
- **Searcher Interviews**: We conducted interviews with 18 developers who had recently visited implementation code in Code Search after looking at header files.
- **Code Search experience sampling**: We deployed two in-situ Code Search surveys, each with hundreds of searchers.
- **Maintainer Interviews**: We conducted interviews with 8 developers who maintain C++ APIs.

In all cases, the method was focused on the research questions. The Searcher Interviews were focused on when developers want to find answers to API usage questions with code versus documentation. The experience sampling responses were focused on identifying what developers were looking for in implementation code (and whether what they were looking for belonged there). In experience sampling [7], questions are asked at or close to the time a triggering event of interest occurs. The benefit of this method is that responses are fresh, and bias due to recalling long-past events is minimized. We used a novel application of experience sampling triggered by interactions with source code files. The Maintainer Interviews were designed to elicit the factors that impact maintainers' decisions about updating API documentation.

3.2 Search and Documentation at Google

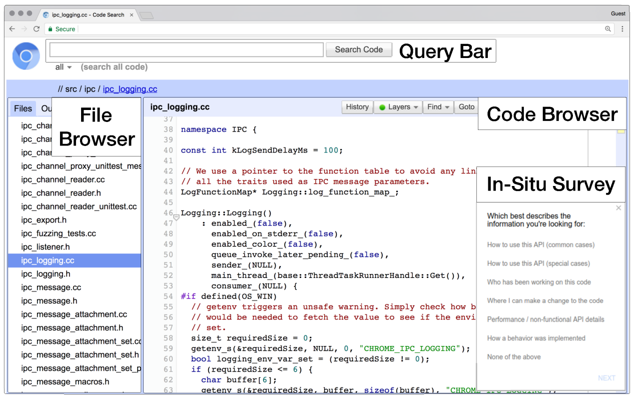
At Google, developers typically write code in a multi-billion line code base [28], using shared infrastructure that is frequently updated. Due to the codebase size, Google developers often browse and navigate source code using the Code Search tool (Figure 1). This tool indexes almost all of the source code at Google, and is used by more than 20,000 developers every workday. We obtained access to the web logs of developer interactions with Code Search. The logs define a sequence of search events for each developer, including queries, clicks on links to function and symbol definitions, file selections, and the application getting or losing focus.

We talked with a variety of stakeholders of API documentation at Google, including owners of core libraries in different languages, technical writers, and developers and project leads working on new tools for finding or viewing documentation. Informants reported that most API documentation for internal code is either embedded in code as comments, or accessible from g3doc, a web-based tool designed to make it easy to write or view Markdown-based documentation. g3doc stores project-wide documentation, typically higher-level than that found in code comments.

Code Search and g3doc are just two of several ways Google developers view API-level documentation. For external libraries, they may browse external sites (e.g., Javadoc, Stack Overflow [3]). Developers may also view documentation that is written as comments in their integrated development environment or text editor.

3.3 Instrumenting Implementation Code Visits

In C++, code is divided into two types of files. Class and method declarations are written in a `.h` "header" file. Documentation for



ipc_char
ipc_char
ipc_char
ipc_channel_proxy_unittest_mes
ipc_channel_reader.cc
ipc_channel_reader.h
ipc_channel_reader_unittest.cc
ipc_export.h
ipc_fuzzing_tests.cc
ipc_listener.h
ipc_logging.cc
ipc_logging.h
ipc_message.cc
ipc_message.h
ipc_message_attachment.cc
ipc_message_attachment.h
ipc_message_attachment_set.cc
ipc_message_attachment_set.h
ipc_message_attachment_set_p
ipc_message_macros.h

```

namespace IPC {

const int kLogSendDelayMs = 100;

// We use a pointer to the function table to avoid any lin
// all the traits used as IPC message parameters.
LogFunctionMap* Logging::log_function_map_;

Logging::Logging()
    : enabled_(false),
      enabled_on_stderr_(false),
      enabled_color_(false),
      queue_invoke_later_pending_(false),
      sender_(NULL),
      main_thread_(base::ThreadTaskRunnerHandle::Get()),
      consumer_(NULL) {
#if defined(OS_WIN)
  // getenv triggers an unsafe warning. Simply check how b
  // would be needed to fetch the value to see if the envi
  // set.
  size_t requiredSize = 0;
  getenv_s(&requiredSize, NULL, 0, "CHROME_IPC_LOGGING");
  bool logging_env_var_set = (requiredSize != 0);
  if (requiredSize <= 6) {
    char buffer[6];
    getenv_s(&requiredSize, buffer, sizeof(buffer), "CHROME
```

Which best describes the information you're looking for:

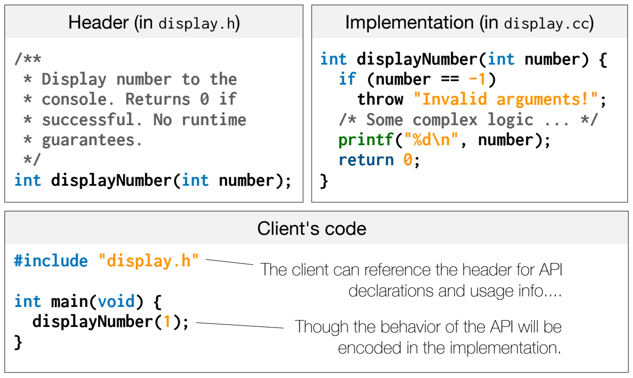
How to use this API (common cases)
How to use this API (special cases)
Who has been working on this code
Where I can make a change to the code
Performance / non-functional API details
How a behavior was implemented
None of the above

Figure 1: The Code Search tool. Google developers frequently browse source code using Code Search. In our study, developers were asked to describe the questions they had about APIs via an in-situ survey, pictured in the bottom-right corner.

these API members (classes, functions, member variables) often appears in the header too. The implementation of a class or method (i.e. actual procedures) are written in a corresponding `.cc` or "implementation" file, typically sharing the same base name (Figure 2). This declaration-definition split is due to the design of the C++ compiler, though the division of declarations and documentation in `.h` files and implementation code in `.cc` files is typical at Google, according to the style guide and developers we interviewed. It also seems to be standard practice for C++ projects outside our company, as implied by classic books on software design (e.g., [15]).

We instrumented Code Search to detect when developers left a header file to visit a corresponding implementation file. We focused on this pattern for several reasons. First, it is straightforward to detect from both real-time activity and logs, comprising only two navigation events from the same application. Second, we expected it would be easy for developers to recall and comment on this transition during interviews. Finally, C++ developers we talked to found this pattern compelling to investigate.

By focusing experience sampling on the transition from header files to implementation code, we inevitably missed developers' transitions between many other types of documents. This was an intentional choice to prioritize an easy-to-interpret pattern over other noisy transitions. Most projects do not use g3doc for API-level documentation, and many transitions to code from g3doc may not indicate missing details. We also ruled out transitions between implementation files, as the only way for developers to jump directly between implementation files is by knowing what file has the implementation. Any click on the use of an API member jumps to its declaration in a header file; another click is needed to reach its implementation from the header file. We also did not track visits to external documentation sites (e.g., Stack Overflow [3]). We mitigate this by collecting most survey responses for internal APIs that have no external documentation; only 3 of 54 paths collected in the API Usage Survey were for external projects.


Header (in display.h)
Implementation (in display.cc)
Client's code
The client can reference the header for API declarations and usage info....
Though the behavior of the API will be encoded in the implementation.

```
/**
 * Display number to the
 * console. Returns 0 if
 * successful. No runtime
 * guarantees.
 */
int displayNumber(int number);
```

```
int displayNumber(int number) {
  if (number == -1)
    throw "Invalid arguments!";
  /* Some complex logic ... */
  printf("%d\n", number);
  return 0;
}
```

```
#include "display.h"

int main(void) {
  displayNumber(1);
}
```

Figure 2: Writing docs in .h and implementation in .cc files. For C++ APIs, the member declarations and actual implementation are often split between two distinct files, as shown in this toy example. Clients may have to reference both to understand both an API's intended usage *and* its actual behavior.

3.4 Searcher Interviews

We conducted interviews with 18 developers, whom we refer to as "searchers" (S1...S18), to learn what developers look for when they leave header files to visit implementation files. We invited subscribers of an internal C++ developers mailing list to agree to participate in a 15-minute interview if warranted based on log analysis. 62 developers opted in.

Twice a day, we ran a script to identify developers in our opt-in list who had made a transition from a .h file to the corresponding .cc file. To increase the likelihood of revealing missing API usability information, we only considered transitions from files that had visits from 10 or more distinct developers in the last six months.[1] We then reached out to the developer to arrange an interview. Most interviews took place on the same day as the searcher had made the transition from the .h file to the .cc file.[2]

We held a semi-structured interviews with each developer (API Searcher Questions in the online appendix [1]). One author conducted all interviews. First, we reminded the developer what .h file they left to visit implementation code, and asked them to describe what they were looking for in the .h file, and why they visited the .cc file. Developers then described their process of looking for information in the implementation code, including files they visited, methods they inspected, and the answer they found.

If the developer was looking for information in a .cc file for how to use an API, we asked them where it would be most convenient to find this information. With any time remaining, we asked the developer about one of two topics. The first topic was to recall recent experiences of information missing from .h files, or of reading well-maintained .h files. The second topic was having them describe their team's process for deciding how to document their code, so we could gather context about how API documentation was written and maintained at Google.

[1]With one exception: for one interview (S12), the header had no views in the past.
[2]One interview was held 2 days after the searcher had made the transition from header to implementation, and four were held 1 day after the transition.

Survey	N	Questions
API Usage Survey	1,147	• Q1: What best describes the information you are looking for? • Q2: What would be the most convenient location for this information? • Q3: What question are you trying to answer about this API? e.g., how this API behaves when passing in dates that are in the past • Q4: What ".cc" files are you looking at?
Implemented Behavior Survey	778	Q1 from "API Usage Survey" and: • Why are you looking into how a behavior was implemented? • You selected "None of the above". Please describe what you are trying to find out by looking at the implementation.

Table 1: Experience sampling surveys to collect developers' questions for implementation code. We ran two main surveys to ask developers to describe the questions they had about code as they left header files to inspect implementation files.

The interviewer did live transcription while interviewing each participant. To validate these transcripts, we recorded audio for all but three participants.[3] For each of the interviews where audio was recorded, the notes were replaced with a transcript of the full session audio. We observed that the difference between the live transcription and the audio was minimal.

Threats to validity: Developers who opted into the Searcher Interviews may not be representative of all developers, at Google or other institutions. Many interviewees had strong opinions, which represented a diversity of viewpoints; there's a chance that some perspectives on the experience of finding answers in code vs. documentation were not represented in this sample. Furthermore, the interviews were aimed at finding anti-patterns in documentation; there are many benefits of getting information from documentation that are not reported in our results.

3.5 Code Search Experience Sampling

We instrumented the web-based Code Search tool so that an experience sampling survey would pop-up whenever a developer navigated from a .h file to a .cc file in the same directory with the same name. Either one of our surveys appeared in the lower-right corner of the Code Search app (Figure 1), and could be closed if a developer did not want to answer the questions.

The aim of our surveys was to understand what questions developers ask when they visit implementation code to learn about APIs, at the moment they left a header file. We launched two variants of the same survey, each designed to collect information about a different type of API question: the *API Usage* and *Implemented Behavior* Surveys. Each survey had hundreds of respondents. The design of the two surveys is summarized in Table 1.

[3]Due to technical issues, audio for S1, S2, and S9 is missing.

The *API Usage Survey* was designed to elicit the types of questions developers asked when visiting implementation code. We asked developers to choose one of seven reasons that best matched why they opened the implementation code. These reasons were chosen based on prior work [10, 32] and the Searcher Interviews:

(1) What best describes the information you are looking for?
  - How to use this API (common cases)
  - How to use this API (special cases)
  - Who has been working on this code
  - Where I can make a change to the code
  - Performance / non-functional API details
  - How a behavior was implemented
  - None of the above

If a respondent selected one of the first two answers, indicating that they were interested in learning about how to use the API, we asked them where it would have been most convenient to find an answer to their question. Participants could choose from one of three options—a `.h` file, a `.cc` file, and the project's g3doc. We also asked what question developers were trying to answer in this case.

(2) What would be the most convenient location for this information?
(3) What question are you trying to answer about this API? e.g. how this API behaves when passing in dates that are in the past
(4) What `.cc` file are you looking at?

The survey also asked respondents to enter the path of the '`.cc`' file, since the survey tool could not collect this information.

We piloted these questions with 246 responses. In the pilot, we asked Q2 regardless of which options were selected for Q1. In addition, the pilot run of the API Survey included the question, "Where would you expect to find this type of information?" with the same options. However, we dropped this question since the responses were essentially the same as answers to Q2. When piloting we used different wording for Q3: "What information are you trying to gather by viewing this '`.cc`' file?", but received too many responses reporting "implementation details".

The *API Usage Survey* was deployed over a period of 3 workdays. After a developer completed the survey once, it would not be shown to them again until at least 5 hours had passed. The survey received a total of 1,147 responses (out of what we expect was about 8,000 total prompts). 60 respondents were looking for information about API usage. 54 of these 60 completed all four questions in the survey; all 54 were looking at different source code files.

The *Implemented Behavior Survey* was designed to answer questions raised internally about what exactly respondents meant by "How a behavior was implemented", a response that a majority of respondents selected in the API Usage Survey. Using the first question from the API Usage Survey, we screened respondents to just those asking a question about implemented behavior. Then we asked two additional questions:

(2) Why are you looking into how a behavior was implemented?
  - Understanding unexpected code behavior
  - Finding code or logic to reuse
  - Planning a code refactoring
  - Checking specific values (e.g. path name)
  - Checking style or best practices
  - None of the above

(3) (If "None of the above" selected). You selected "None of the above". Please describe what you are trying to find out by looking at the implementation:

The *Implemented Behavior Survey* was deployed over a period of 2 workdays. After a developer completed the survey once, it would not be shown to them again until at least 36 hours had passed. The survey received 625 responses. 325 respondents were looking for how a behavior was implemented.

Threats to validity: In the API Usage Survey, some respondents may not have been looking for information about the API defined in the `.h` file, but rather to understand how to use an API called from the implementation code. This possibility is more likely for some types of API questions than others — for example, questions about input types were likely about the API in the `.h file.` As we discuss in the results, question order could have also biased the responses so that we underestimate the percentage of questions that are about API usage patterns in the API Usage Survey.

3.6 Maintainer Interviews

We interviewed the maintainers for a handful of internal APIs (M1...M8) to understand the maintainers' process and rationale for writing and updating API documentation.

M1–M5 were maintainers of header files participants in the Searcher Interviews reported as missing API usage information. It wasn't always straightforward to find an active and relevant maintainer for a file. In two cases, the person we initially contacted was a recent contributor to the file, but made only a handful of contributions, and the document's original authors or main contributors had left the company or were out of the office.

M6 and M7 contributed to files containing API functions that Code Search users click extremely frequently. We interpreted a large number of clicks on a function as an indication that some information about the method was missing. Before contacting the maintainers, we inspected the headers and verified that some important information was likely missing from the comments.

M8 was a maintainer of a widely-used internal API, updating the documentation as part of an open sourcing effort.

One interviewer conducted all eight interviews (see "Questions: API Maintainer" [1]). For M1–M5, we described questions that searchers had about the API, asked whether the question was answered in the header, and if it was not, whether it should be, and where the answer should appear. We also asked if they were surprised that a developer was using their API in this way. All maintainers (M1–M8) were asked to describe their process of deciding what goes into the project's documentation. The interviewer took notes during each interview and transcribed the audio recording from each one. Interviews typically lasted 20–30 minutes.

Threats to validity: As with the Searcher Interviews, maintainers who opted to answer questions about their process could have systematically stronger or different opinions about what belongs in documentation than those who did not opt in; these results should be seen as representing an important but perhaps not comprehensive set of maintainer perspectives.

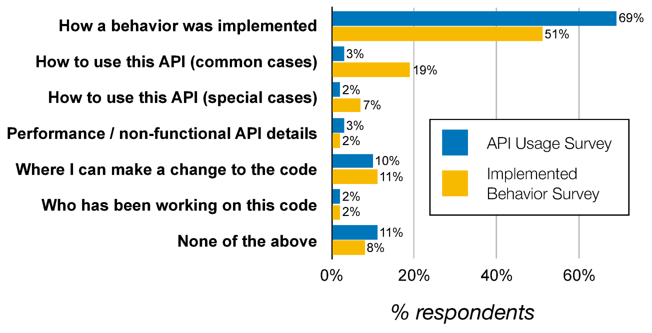


Figure 3: When visiting implementation code, the majority of questions were to find details about how a behavior was implemented, and 5–25% were about API usage.

3.7 Qualitative Analysis

One of the authors performed open and axial coding on transcripts from the Searcher and Maintainer Interviews; this coding was cross-checked by a second author. Both open and axial coding were iterative: we started coding with data from the first several interviews, and refined the codes as we conducted additional interviews. This allowed us to refine the interview scripts to focus on yet-unanswered questions and themes that arose from past interviews.

To analyze free text responses to the survey, two authors performed card sorting. They printed out the 54 responses describing API usage information survey respondents were looking for, and sorted these responses into groups together, discussing the rationale behind the groups of responses. The two authors together gave each group a descriptive name and chose key examples representing the main theme or variations within each group.

4 RESULTS

4.1 Frequency of Implementation Visits

According to our log analysis, Google developers cumulatively spent an average of 234 days of time viewing Code Search pages per single workday in the month of February 2017. Over 4,000 developers leave from a header file to look at the corresponding implementation files every workday. When this transition occurs, the median time from opening a header file to opening the implementation file is about 8 seconds. The median likelihood of someone leaving any given header file for implementation code is 12%, though for some header files the likelihood is 50% or above.

4.2 Questions About Implementation Code

Between 5–25% of visits to implementation code from header files were to learn about API usage, focused on nine types of questions about APIs. Some of these questions are traditionally answered in low-level documentation, and others are not. Most visits that were not about API usage were made to understand how behaviors were implemented, and where to change code.

4.2.1 High-Level Question Distribution. In the API Usage Survey, 5.2% of searchers were looking for information about how to use an API, for common cases (3.4%) or special cases (1.8%). In the Implemented Behavior Survey, 26.5% of searchers were looking up how to use an API, for common cases (19.3%) or special cases (7.2%) (Figure 3). The 26.5% of searchers from the Implemented Behavior Survey aligns with results from related work, which suggested that around 22% of developer queries to Google's Code Search tool were made to learn more about APIs [32].

We are not sure why we observed a difference in the proportion of respondents looking for API usage information in the two surveys. One explanation is the variation in how options were ordered. In the API Usage Survey, options were randomly shown either forwards or backwards; in the Implemented Behavior Survey, options were randomly shuffled. This could have biased respondents in the API Usage Survey to select "How a behavior was implemented" as it was either the first or last option. The takeaway is that in both surveys, a minor yet non-negligible share of searchers visiting implementation code were looking for API usage information.

API usage questions also made up a minority of questions in the Searcher Interviews: about one third of interviewees reported looking for information that was related to API usage. The other 12 interviewees were looking for code to reuse, trying to understand unexpected code behavior, or maintaining or refactoring code.

In the API Usage Survey, searchers who weren't looking for API usage information were looking performance or non-functional API details (3.3%), for how a behavior was implemented (68.9%), who had been working on the code (2.2%), a place to modify the code (9.8%), or something else (10.6%). In the Implemented Behavior Survey, similar percentages of respondents were looking for performance or non-functional API details (1.6%), how a behavior was implemented (51.2%), who had been working on the code (2.4%), a place to modify the code (10.8%), or something else (7.7%).

4.2.2 Questions about API Usage. Of the 60 participants looking for information about API usage in the API Usage Survey, 54 wrote a response to the question, "What question are you trying to answer about this API?" Card-sorting yielded nine types of questions about APIs (summarized with sample responses in Table 2). The relative proportion of question counts should be interpreted with caution as the number of responses to this survey question was small.

It's common for documentation tools and style guides (e.g., Javadoc [2]) to recommend that API designers document the parameters, return values, and errors of each method. However, we observed that developers weren't always finding the right answers to all such questions in the headers alone. Two of the largest categories of questions were for input values ("how a given argument behaves when it's empty") and return values ("what does the return value mean and how can this method fail").

We also observed that some developers were asking questions that are typically answered in high-level documentation, when looking for relevant functionality ("what method to use to convert the current timestamp into a string") and recommended usage of an API ("sample use cases of this API").

Finally, we observed a set of implementation-specific questions that aren't typically considered appropriate for method-level API documentation. These questions were about hidden contracts ("if I need to do special tear-down in order not to leak memory"), implementation details ("how this API passes data to TensorFlow session

API Usage Question	Sample Responses	.h	.cc	g3doc	N
Input Values	"How a given argument behaves when it's empty" "What does the arguments mean exactly" "I'm trying to figure out how the flags are used"	9	2	2	13
How Do I…?	"What method to use to convert the current timestamp into a string"	6	2	1	9
Return Values	"What does the return value mean and how can this method fail"	7	1	0	8
Recommended Use	"Sample use cases of this API" "How to properly update deprecated functions to use this API"	3	2	2	7
Hidden Contracts	"If I need to do a special tear-down in order not to leak memory"	3	1	2	6
Implementation Details	"How this API passes data to TensorFlow session run calls in C"	3	2	0	5
Side Effects	"What logs it writes or status messages it returns when it finishes reading the file"	2	1	0	3
Extension Points	"Whether I should need to override this method in my subclass"	1	1	0	2
Verify Inconsistency	"Why the service in the proto says one thing but the code does something else, and if I can file a fix to correct that"	0	1	0	1
Total		34	13	7	54

Table 2: Nine API usage questions developers asked when looking up implementation code, and where they wanted to find the answers. For each question, we report how many respondent had that question (*N*), and how many thought it would be most convenient to find an answer in a header file (*.h*), implementation code (*.cc*), or the projects g3doc (*g3doc*). Respondents often reported it would be most convenient to find answers in `.h` files, even for implementation-specific questions like those about hidden contracts and side effects.

run calls in C"), and side effects ("what logs it writes or status messages it returns when it finishes reading the files"). Respondents still sometimes thought it would be most convenient to find answers to such questions in headers.

4.2.3 Questions about How a Behavior Is Implemented. In the Implemented Behavior Survey, when respondents were looking for information about how a behavior was implemented, the majority of respondents were "finding code or logic to reuse" (30.4%) or "understanding unexpected code behavior" (31.7%). While it is expected that developers need to consult implementation code when looking for code to reuse, perhaps some unexpected code behaviors should have been documented in these headers.

4.3 Seeking Answers in Code vs. Documentation

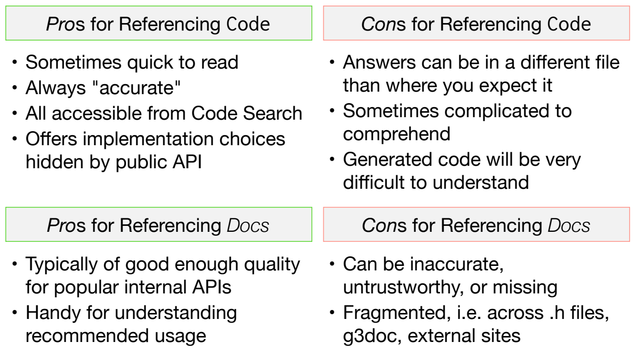
Survey respondents frequently reported it would be most convenient to find answers questions about API usage in headers. However, developers we interviewed in the Searcher Interviews indicated they sometimes preferred to find answers in implementation code, which could be more accurate and quick enough to read.

4.3.1 Convenient Locations to Find Answers. In the API Usage Survey, 61.7% of the 60 respondents looking for API usage information believed that the information they were looking for would have been most convenient to find in a `.h file`. This included 66.7% of respondents looking for information about common usage, and 52.4% looking for information about special case usage. Of the six interviewees in the Searcher Interviews looking for API usage information, four reported that it would have been convenient to find that information in a header file. Survey respondents were more likely to want to find answers in `.h` files for API-related questions than other questions: when piloting the API Usage Survey we asked all respondents (not just respondents looking for API usage information) where they wanted to find the answer to their question. Only 14.2% of the 183 pilot respondents that weren't looking for API usage information thought the header file would be the most convenient place to find the answer to their question.

However, header files weren't always reported as the most convenient place to find answers about APIs. For each type of question, at least one developer always thought it would be most convenient to find an answer in implementation code or g3doc. The proportion of respondents who preferred each location varied somewhat by question (see the rightmost columns of Table 2). For example, in line with our expectations that questions about discovering functionality and recommended usage belong in high-level documentation, g3doc was a preferred medium for some of these questions.

4.3.2 Rationale for Seeking Answers in Code vs. Documentation. The potential benefits developers could have reaped from improved comments varied based on the code that the developer was looking at and the question they had. We distilled a set of themes from the Searcher Interviews describing why some developers expected or preferred to find information about APIs in implementation code as compared with comments in a header file (Figure 4).

Correctness and Completeness. Because it is "what the computer will execute" (S4), developers could count on source code as an accurate representation of that code's behavior. Some interviewees distrusted code comments in general. In the words of S2, "to a first order approximation, I have stopped reading comments, because



Pros for Referencing Code
- Sometimes quick to read
- Always "accurate"
- All accessible from Code Search
- Offers implementation choices hidden by public API

Cons for Referencing Code
- Answers can be in a different file than where you expect it
- Sometimes complicated to comprehend
- Generated code will be very difficult to understand

Pros for Referencing *Docs*
- Typically of good enough quality for popular internal APIs
- Handy for understanding recommended usage

Cons for Referencing *Docs*
- Can be inaccurate, untrustworthy, or missing
- Fragmented, i.e. across .h files, g3doc, external sites

Figure 4: Why look at implementation code vs. header comments? Developers in the Searcher Interviews described when they would look for answers to API usage questions in API documentation or implementation code.

Theme	Examples
Minimal explanations	• No need to explain readable signatures • Readers may have sufficient prior knowledge • Adding details could clutter the docs
Not the right time	• Never maintained, won't be maintained • Concentrating on evolving or fixing the code • Good enough documentation already exists for similar external projects
Preservation	• Should preserve existing comment style • Writing comments that are unlikely to rot

Table 3: Why (not) update documentation? Themes and samples of these themes of why and how maintainers might choose to update the documentation for their APIs.

the comments are just lies. My eyes have just learned to skip them." It was clear why some developers' experiences could lead them to distrust documentation: one of the respondents to the API Usage Survey reported that they were visiting the implementation to clarify an inconsistency between documentation and behavior.

However, incorrect or incomplete documentation wasn't universal. Several interviewees made a distinction between widely-used internal APIs, and projects written by teammates and collaborators:

> "So there's those sorts of [general utilities], and those tend to be very well documented. And then there's the team-specific internal code, which is all very horribly documented. And it's relatively rare for me to find something where I don't personally know the person who wrote it but it's also missing documentation" — S16

Distrust of documentation could lead to unnecessary searches: S6 visited the implementation to check for unexpected behavior when "it was actually documented properly, but I didn't believe it."

Intended vs. actual functionality. Source code can reveal undocumented behavior that can be useful for prototyping. S10 studied how code "actually works" while developing initial "messy code". Once they figured out "how to make it work", then they tried "to make it clean before I send it off for review" by adjusting the code to respect how the API was documented to work.

Cost of finding information. Sometimes, a developer could get an answer to a question about an API pretty quickly by just reading code. For S1 and S14, this only involved looking through a few dozen lines of code over a few methods. Tooling also plays a role in reducing the cost of code navigation: with the Code Search tool, much of the code is indexed in one place, with clickable cross-references between types and functions. Because of this, it may be more straightforward to navigate code than looking for documentation, which could be fragmented across multiple web locations and break one's "flow of thought" (S4). However, sometimes it was infeasible to glean an answer to an API usage question from the code. S9 and S11 searched through multiple functions in multiple files. S9 eventually gave up because the code became too complicated. In these cases, a well-written comment in the right place could have saved time. There are also some types of code, like generated code, which will always be difficult to read (S5).

4.4 Factors Impacting Whether Maintainers Will Update Comments

When presented with questions searchers asked about their APIs, most maintainers weren't surprised that searchers were asking such questions. However, maintainers were sometimes reluctant to update API comments to answer these questions. Reluctance to create and update documentation has been observed in the literature [8]. These interviews provide context behind why proposing and incorporating updates to API-level comments could be difficult. We developed three themes (Table 3) from analyzing Maintainer interview transcripts, and also report on the factors influencing past choices to change and add to documentation.

4.4.1 Keeping Explanations Minimal. This fundamental tension was described by M1 who told us, when asked if they should add a comment to answer S9's question, "How often do you want to go into details, which can be easily too much?" M8 was just as interested in finding out if their API had *too much documentation as whether the comments had the right content.*

Maintainers assumed that sometimes, API clients can infer usage protocol from an API's declaration. M8 described how the semantics of a function could be inferred for an API for formatting time:

> "So just from looking at that [signature] right there, I believe most callers would infer that it takes three arguments, a time and a time zone, and it takes some format that tells it how to format the time, and it returns the value as a string." — M8

However, it is clear is that in practice not all methods in the Google code base are self-explanatory. M8 also described a popular API method, written as a complex template in C++ to replace dozens of other methods with related functionality but distinct signatures. M8 stressed the importance of comments for this method, suggesting that it could take someone half an hour to understand the code without documentation.

For one maintainer, explicit functionality marked the boundary between what deserved to be described in comments and what didn't. M2 quoted Knuth's programming aphorism, "Premature optimization is the root of all evil" [13], to describe how a developer's main concern when using an API will be with writing code that does the right thing, rather than non-functional aspects like performance characteristics. For such non-functional aspects, clients should already have the requisite background knowledge to successfully select and use the API. M2 expected clients of a concurrency API to understand in what contexts they should use threads vs. futures. M5 did not believe it was relevant to describe the runtime of standard algorithms or data structures like maps.

Taken to an extreme, the contents of comments could indicate who should be using the API. M2 worried that if high-level guidance like performance characteristics of threads and futures was available, some developers might misinterpret this information, and be compelled to use it incorrectly or not at all:

> "It's still going to have people that are not experts trying to follow and if you say something is slow, you'll get people writing alternatives first of all, or not using it, or, arbitrarily saying, 'Oh no, you shouldn't use that', just because they, if they're not an expert, all they know is what they've read in that brief comment."
>
> — M2

4.4.2 It's Not the Right Time to Document. In a code base with millions of lines of code, developers may find APIs that were never be intended to be widely used, and continue to use use them long after the authors of that API have moved on to other projects. Logistically, this made it difficult for us to get in contact with maintainers of APIs mentioned in the Searcher Interviews. Contributions from core contributors could be years old, and some contributors had even moved on to other companies.

M3 told us they weren't surprised that searchers like S14 were finding and using an API they had once contributed to. However, the API wasn't for use by other teams, and the team had moved on to other projects. M3 had no intention, and believed no one else had any intention, to further update the code or comments:

> "It's unlikely this will ever get changed again. They're going to delete the underlying data and then, hopefully someone will clean up this utility, I guess, ostensibly it's my team that's responsible for it, but...if you didn't schedule this meeting I would have forgotten this file existed."
>
> — M3

Even for an API with a lot of development attention and an active clientele, maintainers could have good reason not to add or update comments. When informed of a searcher's question, M4 and M5 told us that it wasn't the right time to document. Their project was a re-implementation of a standard C++ library, with "quite a few early adopters." M4 told us current development effort was going to go into improving compiler error messages and fixing performance bugs. Another complication was that, as a re-implementation of an externally available library, documentation already existed—however, this documentation was not accessible within Code Search. For M5, writing documentation in the header files would not only be redundant, it could cause maintenance issues later on:

> "...recapitulating the entire documentation for [our API] here is just not a good choice. It duplicates a lot of things, it lets them fall out of date weirdly... Also, [the online reference] is better cross-linked than the documentation in the header."
>
> — M5

4.4.3 Preference and Preservation. Most searchers and maintainers we interviewed had opinions about what *did belong in documentation, at both the level of headers and in-line comments.* Maintainers and searchers mentioned the importance of describing how a file relates to other files in the project (S17), the state of the world when a method is called (S8), executable examples (M5, M8), implementation comments for future maintainers of an API (M5), explicit links to external documentation (M5), semantics of a function (M8), main concepts that someone should understand and know to use the API (M8), "what" the code is doing and "why" at a statement level (M6), and even a proof of correctness (M6). It is unsurprising that not all of this information was available for all of the APIs we saw during this study.

Choices to include or update documentation could also be based on preserving existing style, and writing comments that can stand the test of time. M6 preserved the style and placement of existing comments when making a one-off contribution to a common C++ utility method. M6 also noted the brittleness of concrete performance descriptions, describing one such inline comment:

> "... it had these numbers, like specific numbers, 50% speedup, 25x speedup, things like that, which are, like, naturally out of date, you know? They were for a machine in 2006 or something like that, like, a particular machine. They're surely not correct today, you know what I mean?"
>
> — M6

To resist "rot", M6 replaced this description with a proof that would hold even as computing infrastructure changed.

4.4.4 Factors Influencing Changes to API Documentation. M2 aptly described the somewhat solitary nature by which comments were often written and updated when they told us:

> "... It was mostly just me, a little bit of feedback from code reviewers, when I was, you know, initially checking this in on what needed to be documented and what didn't."
>
> — M2

While changes to documentation could be on the whole infrequent, maintainers mentioned several cases in which they might update documentation. The most obvious path was through routine development workflows, by getting feedback from code reviewers (M8, S14), or cleaning up comments while refactoring or updating existing code (M6, M8, S19). Beyond typical development process, some maintainers also told us they had considered (though did not always accept) suggestions raised through company email and chat (M1, M7), and questions raised on mailing lists (M4) as potential indications of usage that should be better explained.

5 DISCUSSION

In this study, we found that a minority of visits to C++ implementation code were for questions about API usage, some of which are conventionally covered in documentation and others that are not. Survey respondents reported it would be most convenient to find answers to many of these questions in header files, though interviewees indicated code could be accurate and quick enough to read in many cases. Maintainers had reason to be reticent to update documentation in response to some of these questions. What do these results, as a whole, imply for maintainers and researchers?

5.1 Implications

5.1.1 Contributing to a Conceptual Framework of Documentation. Dagenais and Robillard describe documentation process as comprising initial effort, incremental changes, and bursts, with decision points throughout [8]. For our sample of APIs, incremental changes and bursts were not frequent events. Among others factors, maintainers' attitudes toward updating documentation were impacted by factors including whether the API others were using was something they should be using, keeping explanations minimal for the intended audience, and the availability of documentation for comparable APIs. Our conversations with searchers also suggested additional factors that should be considered:

Should you document for the unexpected client? What happens when the internal utility you forgot about gets invoked by developers across the ocean as an API? This was the case for one of the maintainers we interviewed. At a company with globally visible source code, it's possible any file could become a template or API for another developer's code. Who should be watching that code and documentation when the team working on it moves on?

When isn't code enough to be self-documenting? Sometimes, developers had no problem reading code, and in fact preferred it for finding more accurate information. However, there are some cases where self-documentation isn't feasible, like code with overly complex method signatures and generated code. Other details, like recommended usage, just can't be conveyed by source code.

Which "implementation details" belong in docs? While documentation standards like Javadoc suggest that the behavior (i.e. input and output) of APIs functions should be documented, our study showed that developers had questions about implementation that didn't get answered in the headers, including some questions that would be most conveniently answered in headers. However, this type of information appears infrequently in reference documentation [18]. It's clear that it's not the best choice to break the abstraction of an API to discuss internals that are obvious from the code—which ones should be documented?

5.1.2 Implications for Maintainers. One actionable result of this study is confirming the existence of unanswered API questions in authentic programming settings beyond input-output specification (e.g., side effects). However, this study questions some assumptions about what belongs in documentation. The results suggest maintainers can answer questions with discoverable, understandable code instead of comments. If code is self-explanatory, it may be sufficient to answer searchers' questions, especially if the searchers are experienced in navigating code for answers. However, code is no substitute for high-level information or when it is very complicated to read. Furthermore, many surveyed developers felt *some* salient implementation details should be surfaced in headers. Updates to documentation should heed a diversity of user questions and code-vs-documentation expectations we observed.

5.1.3 Implications for Tool-Builders. This study presents a diversity of questions whose answers could be surfaced, beyond just function signatures, and even including implementation choices. Tools for editing and navigating implementation code may benefit from helping developers find answers to these uestions quickly. This study also shows the messiness of proposing updates to documentation. The ideal time to propose changes to documentation is during code authoring and review, possibly through a surrogate like a code reviewer. Documentation can get updated only infrequently after it is initially written, as future updates may raise questions of whether the information adds clutter or redundancy.

5.2 Results in Context

While this study was conducted at a software engineering company with billions of lines of code and dedicated code search tools, we expect our observations apply elsewhere whenever:

Implementation code is searchable. At Google, there are dedicated tools to support looking up implementation code. For APIs where source code is less accessible, developers probably won't ask the same questions of implementation code, or have the same preferences of where they find answers. Many professional and open source projects rely on both a mix of local APIs for which source code is readily available, and external APIs for which it isn't.

Developers search the source code. Perhaps because of mature search tools, Google developers frequently search code when asking questions about code [32]. Developers' willingness to reference code likely varies by company or project. However, we note that professional developers' willingness to read code over comments has been observed in several other research settings [31, 33].

Implementation code and documentation are separated. For other languages, code and documentation may not be in separate files (e.g., Javadoc). While one couldn't replicate our methodology for such languages, we expect there would be overlap in the questions developers ask. Specific instances of questions and frequency might change for other languages: for example, a Python developer may ask questions about side effects, though likely not the question about memory leaks one survey respondent reported.

Some APIs are unstable. This study considers APIs from widely-used utilities to team-specific libraries used by a handful of developers. Developers likely rely more on documentation for stable APIs where more effort has been put into documentation.

6 CONCLUSION

In this mixed-methods study, we collected a cross-section of developer questions about API usage, and API maintainers' attitudes about updating documentation in response to these questions. Reflecting on the resulting set of questions about APIs, and contextual factors that influenced maintainer attitudes on updating documentation, our observations provide a new set of questions maintainers and tool developers should consider in the pursuit of improving low-level documentation for APIs.

ACKNOWLEDGMENTS

Special thanks to members of Google's Engineering Productivity Research (EPR) team for guidance, particularly Matthew Jorde, Ciera Jaspan, and Edward K. Smith and to Edward Huang for interview transcription. We also thank documentation stakeholders at Google for providing ideas and feedback about this work, particularly Greg Miller and Ríona MacNamara.

REFERENCES

[1] Appendix to this paper. https://goo.gl/wMg3vY.
[2] How to Write Doc Comments for the Javadoc Tool. http://www.oracle.com/technetwork/java/javase/documentation/index-137868.html. Accessed 12 February 2018.
[3] Stack Overflow. http://www.stackoverflow.com.
[4] Jürgen Börstler and Barbara Paech. 2016. The Role of Method Chains and Comments in Software Readability and Comprehension—An Experiment. *IEEE Transactions on Software Engineering* 42, 9 (Sept. 2016), 886–898.
[5] Joel Brandt, Philip J. Guo, Joel Lewenstein, Mira Dontcheva, and Scott R. Klemmer. 2009. Two Studies of Opportunistic Programming: Interleaving Web Foraging, Learning, and Writing Code. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. ACM, 1589–1598.
[6] Raymond P.L. Buse and Westley Weimer. 2012. Synthesizing API Usage Examples. In *Proceedings of the 34th International Conference on Software Engineering*. IEEE Press, 782–792.
[7] Mihaly Csikszentmihalyi and Reed Larson. 2014. Validity and Reliability of the Experience-Sampling Method. In *Flow and the Foundations of Positive Psychology*. Springer, 35–54.
[8] Barthélémy Dagenais and Martin P. Robillard. 2010. Creating and Evolving Developer Documentation: Understanding the Decisions of Open Source Contributors. In *Proceedings of the Eighteenth ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 127–136.
[9] Uri Dekel and James D. Herbsleb. 2009. Improving API Documentation Usability with Knowledge Pushing. In *Proceedings of the 31st International Conference on Software Engineering*. IEEE Press, 320–330.
[10] Ekwa Duala-Ekoko and Martin P. Robillard. 2012. Asking and Answering Questions about Unfamiliar APIs: An Exploratory Study. In *Proceedings of the 34th International Conference on Software Engineering*. IEEE Press, 266–276.
[11] Scott D. Fleming, Chris Scaffidi, David Piorkowski, Margaret Burnett, Rachel Bellamy, Joseph Lawrance, and Irwin Kwan. 2013. An Information Foraging Theory Perspective on Tools for Debugging, Refactoring, and Reuse Tasks. *ACM Transactions on Software Engineering and Methodology* 22, 2 (2013), 14:1–14:41.
[12] Björn Hartmann, Mark Dhillon, and Matthew K. Chan. 2011. HyperSource: Bridging the Gap Between Source and Code-Related Web Sites. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. ACM, 2207–2210.
[13] Donald E. Knuth. 1974. Structured Programming with go to Statements. *Comput. Surveys* 6, 4 (Dec. 1974), 261–301.
[14] Andrew J. Ko, Brad A. Myers, Michael J. Coblenz, and Htet Htet Aung. 2006. An Exploratory Study of How Developers Seek, Relate, and Collect Relevant Information during Software Maintenance Tasks. *IEEE Transactions on Software Engineering* 32, 12 (Dec. 2006).
[15] John Lakos. 1996. *Large-scale C++ Software Design*. Addison-Wesley.
[16] Thomas D. LaToza, Gina Venolia, and Robert DeLine. 2006. Maintaining Mental Models: A Study of Developer Work Habits. In *Proceedings of the 28th International Conference on Software Engineering*. ACM, 492–501.
[17] Timothy C. Lethbridge, Janice Singer, and Andrew Forward. 2003. How Software Engineers Use Documentation: The State of the Practice. *IEEE Software* 20, 6 (Nov.–Dec. 2003), 35–39.
[18] Walid Maalej and Martin P. Robillard. 2013. Patterns of Knowledge in API Reference Documentation. *IEEE Transactions on Software Engineering* 39, 9 (Sept. 2013), 1264–1282.
[19] Paul W. McBurney and Collin McMillan. 2014. Automatic Documentation Generation via Source Code Summarization of Method Context. In *Proceedings of the 22nd International Conference on Program Comprehension*. ACM, 279–290.
[20] João Eduardo Montandon, Hudson Borges, Daniel Felix, and Marco Tulio Valente. 2013. Documenting APIs with Examples: Lessons Learned with the APIMiner Platform. In *20th Working Conference on Reverse Engineering*. IEEE, 401–408.
[21] Laura Moreno, Gabriele Bavota, Massimiliano Di Penta, Rocco Oliveto, and Andrian Marcus. 2015. How Can I Use This Method?. In *Proceedings of the 37th International Conference on Software Engineering*. IEEE, 880–890.
[22] Yoann Padioleau, Lin Tan, and Yuanyuan Zhou. 2009. Listening to Programmers—Taxonomies and Characteristics of Comments in Operating System Code. In *Proceedings of the 31st International Conference on Software Engineering*. IEEE Press, 331–341.
[23] Chris Parnin, Christoph Treude, and Margaret-Anne Storey. 2013. Blogging Developer Knowledge: Motivations, Challenges, and Future Directions. In *Proceedings of the 2013 IEEE 21st International Conference on Program Comprehension*. IEEE Press, 211–214.
[24] Gayane Petrosyan, Martin P. Robillard, and Renato De Mori. 2015. Discovering Information Explaining API Types Using Text Classification. In *Proceedings of the 37th International Conference on Software Engineering*. IEEE Press, 869–879.
[25] David Piorkowski, Austin Z. Henley, Tahmid Nabi, Scott D. Fleming, Christopher Scaffidi, and Margaret Burnett. 2016. Foraging and Navigations, Fundamentally: Developers' Predictions of Value and Cost. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 97–108.
[26] David J. Piorkowski, Scott D. Fleming, Irwin Kwan, Margaret M. Burnett, Chris Scaffidi, Rachel K.E. Bellamy, and Joshua Jordahl. 2013. The Whats and Hows of Programmers' Foraging Diets. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. ACM, 3063–3072.
[27] Luca Ponzanelli, Alberto Bacchelli, and Michele Lanza. 2013. Seahawk: Stack Overflow in the IDE. In *Proceedings of the 35th International Conference on Software Engineering*. IEEE Press, 1295–1298.
[28] Rachel Potvin and Josh Levenberg. 2016. Why Google Stores Billions of Lines of Code in a Single Repository. *Commun. ACM* 59, 7 (July 2016), 78–87.
[29] Martin P. Robillard. 2009. What Makes APIs Hard to Learn? Answers from Developers. *IEEE Software* 26, 6 (Nov.–Dec. 2009), 27–34.
[30] Martin P. Robillard and Robert DeLine. 2011. A field study of API learning obstacles. *Empirical Software Engineering* 16, 6 (Dec. 2011), 703–732.
[31] Tobias Roehm, Rebecca Tiarks, Rainer Koschke, and Walid Maalej. 2012. How Do Professional Developers Comprehend Software?. In *Proceedings of the 34th International Conference on Software Engineering*. IEEE Press, 255–265.
[32] Caitlin Sadowski, Kathryn T. Stolee, and Sebastian Elbaum. 2015. How Developers Search for Code: A Case Study. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*. ACM, 191–201.
[33] Felice Salviulo and Giuseppe Scanniello. 2014. Dealing with Identifiers and Comments in Source Code Comprehension and Maintenance: Results from an Ethnographically-informed Study with Students and Professionals. In *Proceedings of the 18th International Conference on Evaluation and Assessment in Software Engineering*. ACM, 48:1–48:10.
[34] Jonathan Sillito, Gail C. Murphy, and Kris De Volder. 2006. Questions Programmers Ask During Software Evolution Tasks. In *Proceedings of the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, 23–34.
[35] Giriprasad Sridhara, Emily Hill, Divya Muppaneni, Lori Pollock, and K. Vijay-Shanker. 2010. Towards Automatically Generating Summary Comments for Java Methods. In *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering*. ACM, 43–52.
[36] Giriprasad Sridhara, Lori Pollock, and K. Vijay-Shanker. 2011. Generating Parameter Comments and Integrating with Method Summaries. In *Proceedings of the 2011 IEEE 19th International Conference on Program Comprehension*. IEEE, 71–80.
[37] Christoph Treude and Martin P. Robillard. 2016. Augmenting API Documentation with Insights from Stack Overflow. In *Proceedings of the 38th International Conference on Software Engineering*. ACM, 392–403.
[38] Gias Uddin and Martin P. Robillard. 2015. How API Documentation Fails. *IEEE Software* 32, 4 (July–Aug. 2015), 68–75.
[39] Hao Zhong and Zhendong Su. 2013. Detecting API Documentation Errors. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object-Oriented Programming Systems Languages & Applications*. ACM, 803–816.
[40] Hao Zhong, Tao Xie, Lu Zhang, Jian Pei, and Hong Mei. 2009. MAPO: Mining and Recommending API Usage Patterns. In *Proceedings of the European Conference on Object-Oriented Programming*. Springer, 318–343.
[41] Yu Zhou, Ruihang Gu, Taolue Chen, Zhiqiu Huang, Sebastiano Panichella, and Harald Gall. 2017. Analyzing APIs Documentation and Code to Detect Directive Defects. In *Proceedings of the 39th International Conference on Software Engineering*. IEEE Press, 27–37.