Selection and Presentation Practices for Code Example Summarization

Annie T. T. Ying and Martin P. Robillard
School of Computer Science
McGill University, Montréal, Canada
{annie.ying,martin}@cs.mcgill.ca

ABSTRACT

Code examples are an important source for answering questions about software libraries and applications. Many usage contexts for code examples require them to be distilled to their essence: e.g., when serving as cues to longer documents, or for reminding developers of a previously known idiom. We conducted a study to discover how code can be summarized and why. As part of the study, we collected 156 pairs of code examples and their summaries from 16 participants, along with over 26 hours of think-aloud verbalizations detailing the decisions of the participants during their summarization activities. Based on a qualitative analysis of this data we elicited a list of practices followed by the participants to summarize code examples and propose empirically-supported hypotheses justifying the use of specific practices. One main finding was that none of the participants exclusively extracted code verbatim for the summaries, motivating abstractive summarization. The results provide a grounded basis for the development of code example summarization and presentation technology.

Categories and Subject Descriptors

D.2.7 [**Software Engineering**]: Distribution, Maintenance, and Enhancement

General Terms

Experimentation, Design, Human Factors

Keywords

Code Examples, Summarization

1. INTRODUCTION

Code examples are important in modern software development. Programmers search for code examples frequently and extensively: Nearly a third of the respondents in a survey of programmers searched for code examples every day, and programmers working on implementation tasks in the field conduct web search sessions almost exclusively for finding code examples [32]. Code examples are an expected component of formal API documentation [29]. On popular forums such as Stack Overflow, 65% of accepted answers contain code examples [35], while unanswered questions often lack code [1].

Although most will agree that code examples are useful and desirable in many software engineering contexts, the question of code example *effectiveness* is much more elusive. What makes a code example effective?

While the effectiveness or usability of a code example can generally be related to its intended usage, evidence is mounting that concise code examples are particularly desirable, especially for pedagogical purposes:

> "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"—a Team Lead at Microsoft [29].

Concise examples also tend to be in highly rated answers on the developer forum Stack Overflow [23]. In contrast, longer code examples can be difficult to understand [29] or even be misleading [5], and cause serious presentation problems for summarizing documents, e.g., in web search results [34].

Given the amount of publicly-available source code and the desirable properties of concise code examples, we see great potential for technology that can automatically shorten a source code fragment and adapt it to a concern of interest. No such technology exists, and current knowledge on natural-language summarization does not necessarily apply to source code because of fundamental differences in structure of the input data. For example, the assumption that topic and concluding sentences in paragraphs are likely summary sentence candidates does not directly apply to source code [15].

As part of our first steps toward automatic source-to-source summarization, we studied how humans summarize examples in order to understand how to automate the process. The study considered 156 summaries generated by 16 programmers on 52 code examples. Generating these summaries required determining *which* content to select and *how* to present this content. We analyzed common practices behind these decisions across the hand-generated representations, as well as the rationale behind the practices.

We found that none of the participants exclusively extracted code verbatim for the summaries. Participants employed many practices to modify the content, by trimming a line, truncating code, aggregating a large amount of code, and refactoring code. Not only were the participants concerned with the main goal of the task to shorten code, but also with whether the summary looked compilable, readable and understandable. In terms of the selection of content,

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org.

FSE'14, November 16–21, 2014, Hong Kong, China
Copyright 2014 ACM 978-1-4503-3056-5/14/11...$15.00
http://dx.doi.org/10.1145/2635868.2635877

we found that participants used language constructs and query terms, as other literature has suggested [2, 3, 14, 39]. In addition, we found that participants had the human in mind, for example, by including code deemed easy to miss by the reader and excluding code deemed obvious assuming the reader had previous knowledge of the API.

We present related work next, followed by the study set-up (Section 3) and a conceptual framework for interpreting the results (Section 4). We then detail common selection and presentation practices (Sections 5-6). Section 7 elaborates on the findings from the practice catalog.

2. RELATED WORK

Our study on summarizing source code examples is generally informed by foundational work on text summarization [13]. We also draw from three active research areas in software engineering and information retrieval: studies of code examples, automated source code explanation, and work on snippet generation.

Studies on Code Examples: Numerous studies have provided us with valuable knowledge on what is important in a code example. We will discuss the two most relevant ones.

Nasehi et al. investigated the characteristics of code examples in highly rated answers on Stack Overflow [23]. They found that these examples tend to be "concise": the examples are typically less than four lines and "shorter than similar code inside other answers to the same question", with "reduced complexity" and "unnecessary details" left out. Our summarization focus was motivated by these highly regarded concise examples.

Buse and Weimer studied code examples found in an authoritative source of code examples: the official Java JDK documentation [3]. Their two findings were that markers such as ellipses were employed to indicate an input variable's context-specific value, and that exception handling code was in many JDK examples. Buse and Weimer incorporated these two findings in their code examples synthesis tool. Not all of their findings are applicable to summarization: We observed the usage of ellipses in the summaries but we did not observe the inclusion of exception handling code in the summaries.

Source Code Explanation: Most of the efforts in code summarization are targeted to code-to-text summarization, e.g., producing a succinct set of textual keywords [10, 30], a textual summary given the source code of a method [33], or a textual summary given a class [22]. Sridhara et al. [33] used heuristics for selecting statements within a method and templates for generating textual summaries. For clarity we refer to automated code-to-text summarization as *source code explanation*. Our earlier effort in code example summarization has explored code-to-code summarization using a machine learning approach [38]. Regardless of the summary output format, all of these approaches involve selecting which part of the code is important for a summary or explanation. Rodeghero et al.'s recent study specifically looked into whether three types of Abstract Syntax Tree (AST) nodes were important for the selection task, by tracking eye movements of participants during a code-to-text summarization task [30].

Snippet Generation in Search Engines: Studying what is a good concise representation for source code also relates to work in the search engine domain on what information should be included in a textual snippet in the search result to more effectively help a user evaluate a search result [4, 37]. Our work investigates what is the best representation for a type of domain-specific query: code example queries. Programmers use these queries in many contexts: general search engines, code-specific search engines and question answering sites such as Stack Overflow.

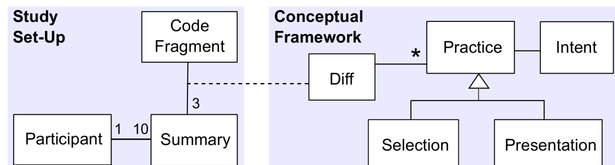

Figure 1: Study Set-Up and Conceptual Framework

3. STUDY SET-UP

The goal of the study was to learn code summarization practices and their justification from human participants to inform future development in source code summarization and presentation technology. We had two research questions:

1. **Selection:** Which parts of the code from an original code fragment should be selected for a summary, and why?
2. **Presentation:** How should the code be presented in a summary, and why?

To answer these questions, we recruited 16 participants and asked them to shorten ten code fragments each. We instructed the participants to verbalize their thought process using the think-aloud protocol [18]. For each code fragment studied, to be able to estimate differences in personal style, we asked three participants to shorten the code fragment, the result of which we call a *summary*. In total we collected 156 summaries on 52 code fragments and 26 hours of screen-recording with synchronized audio. We base the observations reported in this paper on this data.

By analyzing this data and answering the research questions, we learned how concrete code summarization practices lead to specific usability effects for a code fragment. This knowledge directly supports the design of tools to automatically extract and format code examples.

The rest of this section describes the details of the task, code fragment corpus, and participants involved in the study, also illustrated in the left part of Figure 1. The right part of Figure 1 illustrates the data analysis conceptual framework, which is described in Section 4. Sections 5 and 6 respectively present the result of the two research questions.

3.1 Summarization Task

We define of a code fragment summary by adapting the definition of a textual summary [25]. A code fragment summary is smaller in size than the original code fragment and conveys important information in the original fragment. The major goal of a code fragment summary is to present the main ideas in the original fragment in less space.

We asked the participants to provide free-form summaries. For each code fragment, a participant was instructed to write a summary of no more than three lines. To help participants envision the results, we asked them to make summaries as if they were to serve as content summaries in a result page for a search engine (for documents containing source code).

To provide a summary of a code fragment, the participants used a data collection tool we designed for this study

Query: Passing Events Back to the Dialog 's Host

Android API: Dialogs

Code example:

```
 1: public class MainActivity extends FragmentActivity
 2:                          implements NoticeDialogFragment.NoticeDialogListener{
 3:     ...
 4:
 5:     public void showNoticeDialog() {
 6:         // Create an instance of the dialog fragment and show it
 7:         DialogFragment dialog = new NoticeDialogFragment();
 8:         dialog.show(getSupportFragmentManager(), "NoticeDialogFragment");
 9:     }
10:
11:     // The dialog fragment receives a reference to this Activity through the
12:     // Fragment.onAttach() callback, which it uses to call the following methods
13:     // defined by the NoticeDialogFragment.NoticeDialogListener interface
14:     @Override
15:     public void onDialogPositiveClick(DialogFragment dialog) {
16:         // User touched the dialog''s positive button
17:         ...
18:     }
19:
20:     @Override
21:     public void onDialogNegativeClick(DialogFragment dialog) {
22:         // User touched the dialog''s negative button
23:         ...
24:     }
25: }
```

1. Shortened code example:	2. Line#s of the original
`...extends FragmentActivity implements NoticeDialogListener{`	1
`void showNoticeDialog() { //  create instance and show it}`	5-8
`void onDialogPositiveClick(DialogFragment dialog) {//same for negative click }`	14-25

Figure 2: Annotation tool, with a summary by P6

(Figure 2). The top section presents contextual information relating to the code fragment (Section 3.2). The middle section shows the original code fragment the participants were asked to summarize. Finally, the bottom section is a fixed-sized text box in which the participant was asked to enter the summary. In addition to the summary, for each line in the summary, we asked the participants to indicate which lines from the original code fragment the particular summarized line came from. This approach is used in experiments in the textual summarization community [24] for the purpose of evaluating how different summaries overlap. We used this information to determine the textual difference between the original and the summary described in Section 4.

We asked the participants to verbalize their thought process for the entire duration of their summarization activities. We recorded the verbalizations together with a video of the screen.

We chose to study the summarization practices in a lab setting with *access to the summary author*. The participants' verbalizations contained the rationale behind the decisions taken in generating the summaries. Such rationale is harder to reliably infer by the experimenters themselves in the absence of the author of a code example. We also designed the study to have *multiple* authors summarizing the same code example so that we could examine the variability among different code summary authors.

The summarization task was constrained by our decision to limit summaries to three lines. This forced the participants to make choices that were not necessarily the ones they would make if this experimental procedure had not been in place. Research in text summarization suggests that fixing the summary length in an experiment is crucial because summaries of different lengths directly affect the actual content of the summaries [12]. The artificial setup is a component of any lab study: although it decreases the ecological validity of the task, it has the major advantage that it supports the systematic analysis and comparison of the code summarization practices. We return to this issue in Section 4.

3.2 Code Fragments

Selecting code fragments to study summarization practices presents two challenges. First, the general idea of summarization is context-sensitive because of the requirement to assess the relative importance of the elements in the original code fragment. To distill a fragment to its essence, participants need a basic idea of what the fragment is about. Second, code summarization requires a non-trivial level of programming expertise: we cannot ask participants to summarize code they do not understand.

We addressed both challenges by selecting the code fragments from a well-defined corpus of programming documents: The Official Android API Guides.[1] This documentation contains a mix of natural-language text and code fragments meant to explain and demonstrate the usage of the Android API. Selecting code fragments from this documentation helps us address the first challenge above (context) by allowing us to draw from the structure of the text surrounding a code example to provide the context. It helps us address the second challenge by explicitly scoping the expertise required of participants. We discuss the implications of selecting code fragments from this corpus on the generalizability of the results in Section 4.

We extracted all code fragment candidates in the Android guide (1) that were enclosed in HTML `pre` tags; (2) that were non-XML, leaving only fragments with code (though not necessarily Java); (3) that had ten or more non-empty source code lines; and (4) whose closest enclosing heading started with a verb, e.g., "Passing events back to the dialog's host." Selecting candidates with verb-starting heading is due to the necessity in providing a context for the code example (Section 3.3). These four criteria produced 166 candidates. We randomly selected 52 fragments for the study.

3.3 Context Generation

We generated a context for each fragment using an automatic procedure based on the headings enclosing the code fragment. For each fragment, we constructed a context that consists of two parts. The immediately enclosing heading describes a specific purpose of what the code fragment is supposed to demonstrate. The rest of the enclosing headings point to the general area of the Android API the code fragment demonstrates.

Figure 2 demonstrates a code fragment taken from a page with two levels of subsections: First, the page is titled "Dialogs" and the second-level heading is titled "Passing Events Back to the Dialog's Host". In this example, the title of the page, "Dialogs", shows which part of the Android API the code fragment is taken from, as displayed in the annotation tool in Figure 2. The second-level heading provides a more specific purpose, which we marked as "Query" in the annotation tool. For the code examples with three levels headings, the third-level heading is displayed as "Query", and the first two levels are displayed under "Android API". This automatic procedure eliminates the threat of investigator bias in crafting the context, at the cost of a potential loss of precision in how well the context describes the fragment.

3.4 Participants

We assigned the 52 fragments to the 16 participants (P1 to P16) in a way that ensured that all fragments were summarized by exactly three participants. Twelve participants were assigned ten fragments and four were assigned nine fragments.

We required participants to have one year or more of Java programming experience, and have at least looked at

[1] http://developer.android.com/guide/components/index.html

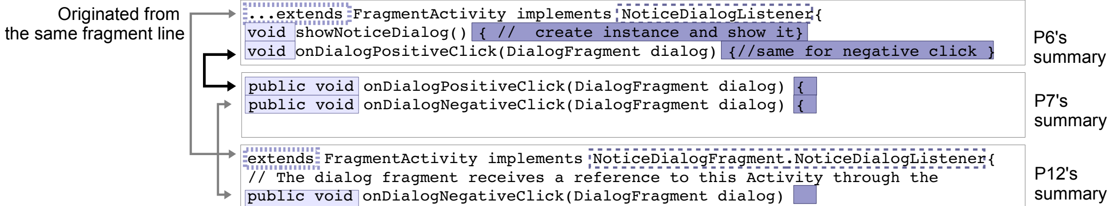

Figure 3: Summaries on the same fragment, with variations on the presentation highlighted

Table 1: Participants' Development Experience

Java \ Android	looked at Android API	developed an app	professional
1 year	P3,6,7,8	P4,5,10	
between 1 & 5 yrs	P9,14,15	P1	P11
between 5 &10 yrs		P2	P16
more than 10 yrs			P12,13

the Android API. Of the 16 participants in the study, five were recruited from local professional programmer meet-up groups, one through personal contacts, and the remaining ten from the McGill School of Computer Science (nine graduate students and one undergraduate). Table 1 presents the participants' Java and Android development experience. In total, seven had professional software development experience.

4. CONCEPTUAL FRAMEWORK

The study produced two different types of data: shortened source code and the verbalizations of participants. We analyzed this data using a combination of quantitative and qualitative [31] methods. The basis for the analysis was the systematic extraction of the textual differences between code fragments and the corresponding summaries ("Diff" in Figure 1). We then refined the difference into a structured list of *summarization practices*. For this purpose we followed *coding* (or classification) techniques [31, Section 2.3], guided by the categories of operations in textual summarization [13]. We distinguished practices concerning the type of content *selected* and the way the content was *presented* in a summary ("Selection" and "Presentation" in Figure 1). The categorization enabled a quantitative assessment of the frequency and generality of each practice. To make hypotheses justifying the use of different summarization practices, we relied on a quantitative analysis of the distribution of each summarization practice across code fragments and participants. This aspect of the analysis was directly enabled by the choice to have each code fragment summarized by multiple participants. Finally, we inspected the transcripts of the participants' verbalizations for evidence of the *intent* behind each practice.

We distinguished between selection and presentation because even summaries with content associated with the same part of the original fragment could have variations on how the summary content was presented. For example, P6's and P7's respective summaries (Figure 3) of the same fragment both included the signature of the method `onDialogPositiveClick` (marked by the arrow in bold), but P6 chose to leave out the keyword `public` and added a comment in the body (the dark shade in the third line), and P7 chose to provide the complete first line of the method declaration (the dark shade in the first line). P12, whose summary contained the signature of a different method, chose to provide the first line of the method declaration without the body (the third line). The decision of treating the removal of tokens (e.g., `public`) as presentation, rather than selection (i.e., deselecting `public`) is dictated by granularity. The granularity we chose for selection is at a higher level in the abstract syntax tree than individual Java tokens. A granularity at the token level would result in more decisions, complicating the conceptual framework of the analysis and the computational complexity of a summarization algorithm. This separation of content selection from presentation is typical in a natural language generation system, where the selection granularity is typically at the sentence level rather at the word level [28].

Links to the Evidence: This study reports primarily on evidence of a qualitative nature. A major challenge for reporting observations derived from qualitative data is linking to the evidence upon which an observation is based. In our case this amounts to providing, for each observation, the number of fragments where a practice is observed and the distribution of these fragments across participants. This amount of precision can quickly overwhelm the text to the point of unreadability. Instead, we use a new visual approach inspired by the idea of sparklines [36]. A sparkline is a small graphic embedded in the text, drawn without axes. In our context, a histogram presents the distribution of observations of a given practice for a participant (each bar) over the ten code fragments (the vertical axis). The 16 bars corresponding to participants are sorted in decreasing number of code fragments where the evidence was observed. The vertical axis represents the number of code fragments where the evidence was present. For example, the histogram for the practice *shortening identifiers* is , signifying that eight participants showed evidence of the practice in 8, 6, 5, 5, 4, 4, 3 and 3 code fragments respectively. We can compare different practices in terms of the amount of evidence that was observed across the participants and code fragments. For example, the practice *shortening identifiers* was observed in more participants and more summaries than the practice *shortening API names* (). This observation can be deduced by comparing the dark area of the two histograms.

The in-lined histograms are intended to provide a convenient and compact assessment of the amount of evidence for a practice. We provide more detailed links to the evidence in Table 2, which relates the histograms to the specific participants associated with the practices, as well as the number of code fragments observed and the number of occurrences in the summaries. Finally, each quote explaining the rationale for a practice is annotated with the corresponding participant identifier, whose characteristics can be found in Table 1.

Threats to Validity: The threats to validity for this study concern the reliability of the observations for the purpose of informing source code summarization technology. We consider the risk that a reader wishing to use this paper could be misled.

The corpus of code fragments is limited to 52 fragments in one technology. It is not representative of any defined population of code fragments besides the Android documen-

tation. However, the contributions we provide in this paper do not involve generalization from a sample to a population. We make no claim about how often the practices we noted are used *in general*, and we do not think such a projection would be particularly useful. Instead, the implications of our results concern the goodness of fit of a certain practice to achieve a particular selection or presentation goal, which is independent from frequency counts. We indicate frequency counts for each practice to be transparent about the strength of the evidence for the observations, without implying that they can be extrapolated.

One threat of using frequency counts as a measure of the strength of the evidence is that not all practices are equally likely to be observed in the 52 fragments. It is possible that our data misses some useful summarization practices, for example if they target special source code patterns that were not part of our code fragment corpus.

Our use of a grounded approach means that the data is collected directly from participants and, as such, is influenced by them. The corresponding threat is that a participant with an unusual background or behaving strangely could corrupt the data. Our experimental protocol required participants to justify most of their decisions, allowing us to discover such potential problems. We observed that all participants appeared to complete the task in earnest. To avoid injecting our own bias, we did not attempt to judge the quality of the summaries.

As mentioned in Section 3.1, the summarization practices we observed were employed in a context where participants were required to produce a short (three-line) summary. This decision was necessary to obtain comparable data. At the same time, it also means that different practices might be useful in contexts where the desired output summary is not constrained by size.

5. SELECTION PRACTICES

Selection practices concern how participants decided on which content to include in a summary, e.g., whether to include a specific method declaration that matched the query terms or whether to exclude exception handling code. We observed three types of selection practices, each using a distinct type of information: language constructs from the code example itself, query terms, and human considerations, such as the programming expertise of the reader.

Understanding these practices can help determine what type of content should automatically be selected (or filtered out) when presenting code examples in contexts where summarization is appropriate (e.g., in search results).

5.1 Practices Related to Language Constructs

Certain types of language constructs were consistently included (e.g., content of a class) or excluded (e.g., exception handling code) in a summary. Figure 4 shows how frequently a language construct appeared in a summary ("# of Selected"). To put these frequency numbers in context, we also provide how frequently the construct was eligible for selection ("# of Eligible for Selection"). This second set of frequency numbers represents either the occurrences of the construct in a code fragment shown to a participant (e.g., `try` exception handling blocks do not occur as often as method declarations), or the number of times the parent node was selected in a summary (e.g., a method signature can only be selected when a method declaration is selected). Finally, the pie charts show the ratio "# of Selected" divided by "# of Eligible for Selection".

Construct	# of Selected	# of Eligible for Selection
Compilation Unit	0	3
Package Declaration	0	21
Import Declaration	0	21
Class Declaration	50	54
Class Signature	12	50
Class Body	50	50
Method Declaration	184	210
Method Signature	124	184
Method Body	126	184
Conditional Structure	65	165
Try Statement	9	12
Try Block	9	9
Catch Block	0	9
Finally Block	0	3
Comment	33	702

Figure 4: How often a construct was in a summary

The first two practices involve methods. All participants selected methods ( ), as P14 justified, "First, I want to know the functions I have to use.":

Practice - Including (or Excluding) the Method Signature: Depending of the code fragment a method signature can be included or excluded.

Including the method signature ( ) was considered as part of keeping the structure of the code. As one participant put it, "because there's a lot of them [code], it can be anything. It's the structure. The main part is the class `BillingReceiver` which extends `BroadcastReceiver`, the method that overrides inside. The rest can be ignored."$_{P9}$ Another participant chose to show the structure of the code rather than the control flow structure: "The `switch` is more about how the method functions. What are the possible functions and outcome. [...] I was just given `switch`, I have no idea of what it is."$_{P7}$ This structure can be important in code on the Android platform with a substantial amount of call-backs. There were fewer cases in which only the method signature was kept ( ), while more fragments had both the signature and the body selected for the summary ( ). One reason for keeping the method body was that the fragment has more computation-intensive code. For example, a fragment about the usage of the gyroscope had more than half of the lines on mathematical computations. For that fragment, all three participants included at least two statements from the method body.

Participants collapsed the method by displaying the content of the method *without* the signature ( ). One participant who eliminated the method signature said that the declaration was a common API call-back, saying, "This handler is pretty much for any activity."$_{P4}$ Another reason was that the participants expected the user of a summary could find the signature through the IDE.

Practice - Including Overriding Methods: Of the method declarations with an explicit `@Override` annotation (43 methods), most of the methods (36) were included in a summary by at least one participant. One participant even called it a "regular pattern"$_{P13}$ to include overriding methods. The seven methods not included by anyone were in code fragments with other choices of methods. However, the override annotation itself was rarely kept, only in six code fragments ( ).

Practice - Excluding Exception Handling Blocks: None of the exception handling code, enclosed in `catch` or `finally` blocks, appeared in a summary. There were several intents behind the practice of excluding exception handling code. First, exception handling code was not unique to an example ("Try-catch is part of almost all standard code."$_{P5}$) or too obvious to the reader ("Anyone working with sockets knows it will throw exceptions. I will remove the catches, and the try."$_{P2}$) Second, the code inside the `try` block was kept (while the `catch` clause was removed) to show one case of the code: "The first thing you should do in an example is [to] assume everything is OK."$_{P2}$ Third, participants expected that missing exception handling code would be suggested by an IDE: "[Missing] this exception you would have Eclipse complaining about it."$_{P11}$

Practice - Keeping Only One Case in a Parallel Structure: Some code fragments contained code with multiple cases. In the case of `if` or `switch` statements, more than one third of the instances only had one block selected for a summary. Keeping one case and dropping the others also happened with method calls ("I can just remove one of the buttons. Instead of having the cancel button, I can just have the OK button."$_{P4}$) or method declarations ("`onStop` is basically the reverse of `onResume`. It will be OK to just display everything in `onResume` [...]."$_{P9}$)

5.2 Practices Based on Query Terms

Not surprisingly, participants used terms from the query to determine whether a part of the code was relevant enough to include in a summary. Thirteen out of 16 participants explicitly mentioned the importance of the query in the decision of content selection. For example, "`startForeground` [a method declaration] actually starts the foreground service. Since the query ["Running as a foreground service"] doesn't have anything to do with the media player, even though it's part of the API being used, [...] I left it out."$_{P15}$

All 16 participants verbally justified the content selection when a code element matched terms in the query. Of these 70 distinct methods from the 52 code fragments ($70 \times 3 = 210$ instances of methods eligible for selection from Figure 4), twenty-two contained at least one query term (or stemmed). Of these 22, only one was not selected by any participant. Both from the verbalizations and from the summaries, we conclude that all participants used the query in content selection for summarization.

5.3 Practices Considering the Human Reader

Summaries are targeted to humans. Participants explicitly considered the expertise of the programmer.

Practice - Including Easy-to-Miss Code: Four participants mentioned including *easy-to-miss* parts of the code in the summary, e.g., the method declaration `onResume` "is something people tend to forget."$_{P11}$ Another participant made a similar comment and explicitly qualified the advice with the participant's own personal experience: "It reflects my own knowledge of this class [...]. If you set the layout in the wrong place, you can end up with a lot of problems. I want to be specific there."$_{P10}$ Another participant expressed frustration at not being able to include a call to the super class which was deemed easy to miss: "I'm still not happy to remove the `super`. If someone looks at the short one, copy and start. He could miss it."$_{P2}$

Practice - Accounting for Programming Expertise: Seven participants justified not including parts of the code that were too obvious to the reader. The code might be obvious because of (1) previous languages used ("This is C-style where you handle one byte at a time. It would be pretty obvious this is how you do it"$_{P11}$) (2) the assumption on the knowledge of Java ("The sockets [...] is not specific to Android. It's exactly the same in standard Java"$_{P2}$) , or previous knowledge of the Android API ("`onCreate` is a method where if he knows a little bit of Android development, he won't get a lot from this [`onCreate`] anyways."$_{P5}$) In general terms, one participant explicitly distinguished the different needs of an expert and a novice of an API: "Someone who's very experienced [...] may be looking for something very specific, [such as] methods [...]. Someone who is a complete novice would probably look at something very explanatory."$_{P3}$

Practice - Using the Query to Infer Expertise: Participants used the query to infer the level of expertise on the API of the query poser, and then excluded the part of the API deemed obvious. One participant commented on the decision of not including certain method declarations for a query about Near Field Communication (NFC), a topic the participant deemed advanced. "If someone is doing NFC, [...] someone already knows what `onPause` [or] `onResume` is, so I don't need to stress it. This is more advanced stuff than how the activity behaves."$_{P11}$ Interestingly, P11 was one of the participants who deemed `onResume` easy to miss in another fragment.

6. PRESENTATION PRACTICES

Presentation practices relate to decisions about *how* the selected content appeared in a summary. We observed participants made changes to the selected original content to make it fit into the space allowed for the summary through practices in three general categories: trimming a line when needed (Section 6.1), compressing a large amount of code (6.2), and truncating code (6.3). Beyond presentation decisions for the purpose of fitting the desired content into the space, we found that formatting decisions were personal and related to readability of the summary (6.4). Despite the task's focus on reducing code, we observed participants improved the code, e.g., by clarifying comments (6.5).

The presentation practices we collected provide practical insights into how source code fragments can be formatted in various situations including on search results pages, in forum posts, and in tutorial documents.

6.1 Trimming a Line When Needed

Ten participants ( ) performed transformations for the purpose of trimming a line, such as shortening variable names or removing a type qualifier. These operations happened when the content needed to fit into a line pre-allocated for that content.

Practice - Shortening Identifiers: We expected participants to shorten variable and parameter names, because these changes do not change the semantics of the program. Eight participants ( ) did so in 29 (56%) code fragments. We observed a number of ways to shorten a name: (1) using acronyms, e.g., from `sharedPreferences` to `sp`, (2) shortening words in an identifier, e.g., from `defaultValue` to `defaultVal`, (3) using discourse aggregation for reducing the complexity [27] by dropping words (e.g., from `defaultValue` to `default`) or para-phrasing (e.g., from `tagFromIntent` to `intenttag`), and (4) using a combination of these operations, e.g., `mInputStream` to `in`. These observations concur with Eshkevari et al.'s taxonomy on identifier renaming in a code base [6].

Table 2: Evidence of the presentation practices

Presentation practices	#Participants (out of 16)		#Fragments (out of 52)	# Instances
Trimming a Line When Needed	10	P2,4,6,8,10,11,13,14,15,16	33	95
Shortening Identifiers	8	P2,4,6,8,10,11,15,16	29	72
Shortening API Names	4	P6,10,11,15	5	7
Eliding Type Information	10	P2,4,6,8,10,11,13,14,15,16	9	16
Compressing a Large Amount of Code	13	P2,3,4,5,6,7,8,9,10,11,14,15,16	28	46
Shortening Multiple Statements	10	P3,4,6,7,8,9,10,11,15,16	15	51
Shortening Method Declarations	7	P4,6,8,10,11,14,16	10	11
Shortening Control Structures	8	P2,3,5,6,8,10,11,16	12	14
Truncating Code	12	P1,2,4,5,6,8,9,10,12,13,15,16	28	63
Eliminating a Parameter	9	P1,4,5,6,8,9,10,15,16	16	28
Truncating a Signature	9	P2,4,5,8,10,12,13,16	18	35
Formatting for Readability	16	all	52	140
Indenting	8	P1,2,4,7,8,9,13,14	20	27
Treating Lines as Separate	15	all except P10	52	135
Improving Code	9	P1,2,4,6,8,10,11,12,16	9	35
Fowler's Refactorings	4	P1,2,8,11	4	5
Generalization	4	P2,4,8,16	5	8
Clarification	6	P6,8,10,11,12,16	14	22

#Summaries (out of 156)

Practice - Shortening API Names: We expected that the names of API calls and overridden methods would remain the same in a summary. As P6 asserted, "I am assuming overridden methods cannot have their names changed." Surprisingly, we observed changes to these API elements, by four participants ( ). P6's justification on the shortening of the name of an API method was that the name is "abnormally long"P6: "`unregisterOnSharedPreferenceChangeListener`, what kind of name is that?"P6 P6 renamed the method to "`unregister...`". Note that the context was important in this case, as the API call is expected to be made inside a class that inherited from `SharedPreferenceChangeListener`, which defined the unregister method.

Practice - Eliding Type Information: Java requires a variable to have a declared type in an unambiguous namespace and explicit down casts. In the context of summarization, ten participants ( ) relaxed this requirement and elided type information in the following situations: (1) Four participants eliminated a type qualifier; e.g., "I'm removing the name-space. [...] Someone can [put a] `import static`."P2 In addition, the type information is a piece of information expected to be found easily: "[For] the flag, if they are in the definition of the type, they can see which flags are in the type."P12 (2) Five participants removed the variable type in assignments in six assignments that were selected over five summaries. (3) In the only selected line that contained a type cast in the whole corpus, all three participants selected that line and removed the type cast in consensus. (4) Two participants shortened a type reference or a primitive type: e.g., from `Object` to `obj`.

6.2 Compressing a Large Amount of Code

Twelve participants ( ) employed more complex abstraction and aggregation practices that greatly reduced the code from its original size. These changes involved compressing a block of code that contained one or more method declarations, control statements, or multiple statements and replacing the code with ellipses or a comment. Four participants (P5,7,9,15) only employed ellipses when compressing a large block of code, four (P4,11,14,16) only employed comments, and four (P3,6,8,10) employed both.

It is inevitable that when a block of code deemed important exceeded the space available for summaries, the participant needed to somehow compress the code. We observed that participants either compressed the code using ellipses ("'...' [indicates] additional important things"P8) or comment. Ellipses and comments also could "abstract a particular block."P8 Certainly, comments conveyed more information than ellipses. However, choosing comments or ellipses was affected by the trade-off between information and space: All the comments in the summaries were longer than three characters.

Practice - Shortening Multiple Statements: Ten participants ( ) shortened multiple statements including the whole method body. The use of comments versus ellipses was split almost evenly: Six of the participants (P3,4,6,8,10,11) used comments 22 times and seven (P3,6,7,8,9,10,15) used ellipses 29 times. P15 who only used ellipses to summarize multiple statements said, " Most of the time I put '...' when there are lines in between. If you don't put that in, it's less clear there's other stuff in there."P15

Practice - Shortening Method Declarations: Seven participants ( ) aggregated whole method declarations by replacing the whole declaration with comments or with ellipses. Unlike in abstracting multiple statements, most participants (six out of the seven) used comments rather than ellipses (one out of seven) to abstract method declarations. The ten comments demonstrated a number of different ways to abstract content: (1) listing the method declarations (eight comments), e.g., lines 1 and 2 in Figure 5; (2) aggregating lexically [27] through the use of the quantifier "all" (one comment), as in, e.g., "all inherited methods" in line 3 in Figure 5; and (3) aggregating semantically [27] (one comment), e.g., the comment `//same for negative click` which referred to the code for handling the positive click. Lexical aggregation is a way to summarize a list of elements with a few words rather than explicitly listing the methods [27].

Practice - Shortening Control Structures: Eight participants ( ) shortened control structures. Four participants replaced a block in a conditional statement or a

```
1 /* implement SensorEventListener @override ↩
      onAccuracyChanged(), onSensorchanged() */
2 @override onCreate(), onAccuracyChanged(), onResume()↩
      , onPause() onSensorChanged(){...}}
3 // remember to override all inherited methods ↩
      appropriately
```

Figure 5: All three summaries on the same example contained a comment listing overriding methods

```
1 while(cur.moveToNext()) {...}}
2 if (checked)... else...}
3 if (resultCode == Activity.RESULT_OK && requestCode ↩
      == PICK_CONTACT_REQUEST) { //code for activity }
```

Figure 6: Sample of summarized control structures

`switch` statement, or in a `while` or `for` loop, with a comment or ellipses. Figure 6 illustrates three such examples. Beside using ellipses and comments, five participants (P2,6,8,11,16) compressed the whole structure through program semantics preserving transformations. Participants either turned an `if` statement into a more compact conditional expression (with operators `?` and `:`), or turned a `switch` statement into an `if`: "`switch` is going away because [...] they become too big. I'm just going to put an `if`."$_{P2}$

In brief, ellipses and comments were approaches to shorten a large of piece of code. This result concurred with one of Nasehi et al.'s findings [23] on concise code examples that contain "place-holders, such as comments or ellipses, which usually transforms the code to a solution skeleton." We found that to summarize method declarations, almost all participants employed comments instead of ellipses. The majority of the comments were simply listing the name of the method declarations or using lexical aggregation.

6.3 Truncating Code

Code truncation transformations involve shortening a line while violating syntax. Twelve participants ( ) performed such truncation. These code truncation transformations affected code compilability, which some participants considered important. One factor that affected the presentation of code was a participant's view on the importance of making the code compilable. This view varied between the participants. Those who were more indifferent to the importance of compilability tended to perform code truncating operations violating syntax, using ellipses, or cutting off parts of a statement or having unmatched brackets or parentheses.

Four participants mentioned the importance of compilable summaries. The most common reason was that participants wanted to copy and paste the code directly: "It's very important for the code to compile correctly. [...] I'm a very lazy person. I would Google [...] the snippet, [...] copy it, and pretend it's mine."$_{P6}$ Another reason was for understandability: "Having something compilable allows me to actually see the effects, and that will help me to understand the code better."$_{P8}$ Compilable code was also important because otherwise the summary could look "sloppy."$_{P2}$

It was not always possible to make the code compile. When the code was not compilable, participants wanted to minimize the non-compilable parts: "I want to copy the least amount of code or through the least number of places. Copy code with the least number of changes [that] would [make the code] work."$_{P6}$ One participant wanted to clearly mark the non-compilable parts of the code: "It's important to either compile on its own, or if it does not compile, it is readily identifiable what needs to be done to

```
/*convert ns to s */ omega = sqrt(X*X + Y*Y + Z*Z); if (omegaMagnitude > EPSILON)
 { X /= omega; ... } theta = omega * deltaT / 2.0f; deltaR[0] = sin(theta) * X...
deltaR[3] = cos(theta); SensorManager.getRotationMatrix(new float[9], deltaR);
```

Figure 7: A summary without formatting, by P10

make it compile."$_{P8}$ P8 invented a language construct, a pair of angle brackets to indicate variables not declared in the summary: "Here I am going to add uncompilable code [replacing the variable `name` with `<NAME>`]."$_{P8}$

On the other hand, one participant did not see the importance of having compilable or runnable code: "For such a short and abstract example, [...] we are not talking about runnable code."$_{P2}$ Less so, P4 said, "If it's not compilable, Eclipse or whatever editor you use will give some hints. This expects a pointer, or this expects an object of this class, inherit this class, the kind of auto-fix suggestions that Eclipse give."$_{P4}$

Practice - Eliminating a Parameter: It was sometimes desirable to eliminate a parameter which is deemed to be a detail: "When we search for something, we don't want too much stuff that is irrelevant, [for example,] the parameters."$_{P9}$ Nine participants ( ) shortened the parameter list in a method call or a method declaration. When eliminating a parameter, participants chose to replace a parameter with ellipses (eight participants over thirteen code fragments) as well as simply eliminating a parameter (three participants over four code fragments). P1 justified the use of ellipses: "Here [where the parameters were eliminated] you have to put '...' because there are multiple parameters."

Practice - Truncating a Signature: Because many of the method and class declarations are part of the callback mechanism of the Android framework, we expected that when such a method or class signature was selected for a summary, the signature would be kept intact. For both method declarations and class declarations, leaving the signature intact was indeed the most common way for a signature appeared in a summary. However, a significant number of participants ( ) had summaries with the method or class signature modified. These changes involved Java keywords (such as `public` or `static`), identifier names, or the whole signature replaced by a comment. One participant justified the removal of keywords, saying, "`public class` something, `extend` something [...]. This is rudimentary. [...] All this stuff is meaningless."$_{P6}$ This justification corroborates with conclusions from work on statistical modeling of source code [11], pointing out that source code contains redundancy.

6.4 Formatting Code for Readability

Participants explicitly expressed the importance of two different readability dimensions that related to formatting: indenting and treating lines as separate. Figure 7 illustrates a summary with little formatting, i.e., without any indentation, and with no separate lines. In Table 2, for formatting practices we report the number of summaries exhibiting the practice instead of the number of instances because formatting practices apply to the whole summary, not necessarily to a specific line.

Practice - Indenting Code: Indentation in code can increase readability: "It's easier to see the layers, the level of importance. You look at [the code] from top to bottom."$_{P9}$ Two participants mentioned that indentation is a standard coding convention and is required by languages such as Python. Eight participants ( ) intentionally indented at least one summary. We did not count cases when the indentation in the summary was not intentionally put in, e.g.,

```
1 if (item.isChecked()) item.setChecked(false); else ↩
      item.setChecked(true);
2 item.setChecked(!item.isChecked());
```

Figure 8: P4 refactored code from line 1 to line 2

indentation that were simply copied and pasted from the code that contained indentations.

Practice - Keeping Lines as Separate: Keeping the summary as separate lines can be seen as desirable, whereas wrapping lines and putting two lines into one can be seen as undesirable. P11 declared, "it's ugly," when a comment expected to fit a line wrapped around to the following line. P2 considered eliminating a line break between the class signature and the method signature as undesirable: "The class definition and the method on the same line. That will be really crazy." All participants ( ) treated at least one summary with all separate lines, i.e., not wrapping lines and not putting two lines into one in a summary.

Participants' views on readability was divided. Half of the participants (P2,4,8,9,11,12,13,15) explicitly expressed importance in readability. P2 despised original code with poor readability: "I look at this and I'm scared. Oh my god, what's happening here? There's not a break line." The other half of the participants included P3 who did not think readability is important because summaries are short: "Since it's just three lines of code, [...] I don't think he [the reader] would mind the formatting."P3 P10 thought packing more information is more important than readability: "If it [the summary] is more readable and as a consequence there is less information, you still would not know whether you want to click [the link to the whole example." Despite the eight participants who did not think readability was important, the two formatting practices were used by all participants ( ).

6.5 Improving Code

We observed three types of transformations that improve the code: refactoring, generalization, and clarification. Nine participants ( ) took the effort to improve the code: "Can I, interesting, well I guess I can. I should. From my experience, I will just do this [refactoring]."P2 Because the main objective was to shorten the code, we found any improvements surprising, especially when some improvements, such as adding in clarifications as comments, lengthened the code.

Practice - Fowler's Refactorings: Four participants ( ) applied two different types of refactoring to control flow structures [8]. P2 and P11 applied refactorings in the spirit of the "Consolidate Conditional Fragments" refactoring on the same code fragment. P2 eliminated unnecessary control flow branches, turning line 1 in Figure 8 into line 2. P1 and P8, on two code fragments, applied the "Consolidate Duplicate Conditional Fragments" refactoring, moving part of code that is in all branches of a conditional expression to the outside of the expression.

Practice - Generalization: Four participants generalized a value or variable specific to the examples to something more likely applicable to other contexts ( ). P2 and P4 generalized a constant specific to the examples to a variable, e.g., from the constant `Intent.CATEGORY_ALTERNATIVE` to the variable `myCategory`. P8 and P16 replaced a variable specific to the example with a place-holder. The place-holder employed by P8 was an invented notation, an angle bracket (e.g., replacing the variable `R.id.menu_search` with `<NAME>`) or with a comment for P16, (e.g., replacing the string constant `"landscape"` with `/*orientation*/`). P8 and P16 essentially treated the summary as a closure and noted the free variables with the place-holder and comment. The termination of this type of generalization is called conceptual aggregation in the natural language generation domain [27].

Eight participants explicitly mentioned that some parts of the code were important for the example to work, but too specific to the example. One participant shortened a path because "someone [the query] assumes the data is an image, but it doesn't need to be an image."P2

Practice - Clarification: Six participants ( ) added to the summary clarifications that were not present in the original code. Five clarified names of the variables; e.g., P16 justified replacing the variable `cur` with a more descriptive name, `queryResult`, especially important for a variable that is the input or the result of a piece of code: "Intermediate variables don't matter, but what needs to be fed in and what needs to come out, those two variables [`cur` and `cr`], [matter.]"P16 We observed 16 comments in nine different code examples. Fifteen of those comments reiterated what the code summary presented, while one comment (`//Start and stop download when activity is in foreground`) clarified that the callbacks `onResume` and `onStop` were run in the foreground, an insight not explicit in the original code. Some of these clarification transformations are found in automatic algorithms to expand and improve identifiers [16].

7. DISCUSSION

Most of the work on code example generation and summarization has focused on the content selection aspect. Language constructs have been used for code fragment summarization [38] and code example synthesis [14]. Systems for extracting [2], synthesizing [3, 14], or summarizing [38] code examples have made heavy use of query terms. These practices of using language constructs and query terms were used by the participants, a result concurring with existing work. Section 7.1 discusses some novel types of information we observed beyond the existing use of code itself and query terms. Sections 7.2 to 7.4 focus on implications related to the presentation practices. Finally, Section 7.5 discusses some implementation ideas.

7.1 Selection Beyond Code and Query Terms

Accounting for expertise information to determine which content should be included can be a promising type of information to complement existing code example search engines that are based heavily on the code itself and the query as the input to the analyses. Existing measures to quantify expertise include the use of commit logs and interaction history [9, 21]. These measures all share the assumption that the more a developer changes the code or calls a method, the more expertise of the corresponding code or API method the developer has. We observed that participants either assumed the reader had a certain expertise or inferred expertise from the query. In information retrieval, the foundational research on inferring intention is whether a query is informational or navigational [17].

7.2 Most Summaries are Abstractive

Current textual summarizers generate two types of summaries: *Extractive* summaries have the content obtained solely from copying and pasting whole sentences from the

original document, whereas *abstractive* summaries can contain text modified from the original document [19].

If all participants were to provide extractive summaries, we would only observe selection practices and formatting practices (modifications involving white spaces) in the summaries. However, all 16 participants ( ) employed modifications beyond changing white spaces, namely, modification involving trimming a line ( ), compressing a large amount of code ( ), and truncating code ( ). As we saw in Section 6, the participants made changes to the selected content to make it fit into the space allowed for the summary.

Modifications associated with abstractive summaries were present in 90% (47 out of 52) of the the code fragments; thus, these 90% of the code fragments had at least one abstractive summary provided by a participant.

7.3 Abstractive Summary Generation

The code-shortening transformations found in the field of code transformation and example generation and extraction typically generate syntactically correct code and preserve program semantics. For example, shortening identifiers using acronyms is used in Buse and Weimer's code example synthesizer [3]. Knowing which words to shorten in an identifier or using discourse aggregation require a deeper understanding of the linguistic aspects of the identifier, such as part-of-speech information. We observed the shortening of API names only in exceptional cases, when the context was clear and when the name was long.

Several presentation practices however did not necessarily result in source code. In the *shortening method declarations* practice, we observed summaries with both code and natural language. Overall, seven participants ( ) injected additional natural language (in the form of comments or place-holders described in Section 6) into the code summaries. This motivates a novel type of transformations that mix code and text. The only work we know of in this area is the natural summaries generated by Rastkar et al. [26]. Their summaries describe a commit as part of a software concern. The patterns found in their summaries include listing the method declarations changed in a commit and using lexical aggregation to to describe a commit (e.g., "All of the methods involved in implementing 'Undo' are named `undo`"). We have observed both patterns (listing and lexical aggregation), as shown in Figure 5.

The presence of improvement transformations was surprising. However, the proportion of these transformations were small. The five instances of refactoring of control flow structure were out of 65 on conditional structures selected, and the ten instances on generalization and clarification on values and variables were a small fraction of the total number of statements containing a variable. The 16 comments were only inserted by two participants, and most of the comments were redundant because they reiterated the code. Also, generalization and clarifications are challenging to generate even in a natural language generation system. Extraneous to the main goal of a summarizer, improvement transformations should be of lower priority for a summarizer to address.

7.4 Silhouette of a Summary is Important

Formatting practices determine how much space a summarizer has for a summary. We observed that all participants employed some formatting in their summaries. The formatting included respecting indentation and keeping lines as separate lines. These results uniquely apply to the problem of summarizing and presenting source code, as opposed to text. Text summarizers typically use the space as a contiguous stream of characters with no indentation.

The amount of space in the summary could affect readability. "I don't like packing more stuff. I always want readability. That helps me, in one glance, to assess whether the particular code is helpful or not. [...] If the code is packed, it's pretty hard. It would go for another example which has more clarity."$_{P4}$ We did not derive any specific practice from this behavior due to the difficulty of objectively defining what "packing more stuff" means. We nevertheless observed that there did not appear to be a strong correlation between the length of a code fragment and its summary (Pearson $R = 0.0758$, $p = 0.347$), indicating that many other factors could influence the density of a summary.

7.5 Implementation Directions

To transform a source code example to a summary, one can opt for edit scripts to encode the transformations. Edit scripts are sequences of edit operations including insertions, deletions, and updates. A number of existing approaches can infer these scripts effectively on two versions of the code [7, 20]. Selection and presentation summarization practices can be conceptually mapped to the two main components of an edit script: where in the code (i.e., the AST) an edit operation should be applied (selection) and which edit operations should be applied (presentation). One challenge in adapting existing approaches for generating summaries is to relax the assumption that a significant portion of the two versions of the code is the same, because the difference between a fragment and a summary can be substantial.

8. CONCLUSION

This study elicited selection and presentation practices we observed from 156 concise code representations obtained from 16 participants. The goal of the study was to inform the design of concise representations of source code and automatic summarization algorithms. The selection practices we observed reinforce the existing usage of code and query terms in content selection in the summarization domain. The selection practices revealed the importance of the human reader, as we observed that participants targeted summaries to the expertise level inferred from the query. Moreover, participants did not simply copy and paste parts of the the original fragment to the summary verbatim; all 16 participants employed practices to modify the content, mostly with the intent to make it more concise but also make it more compilable, readable, and understandable. The practices directly inform the design and the generation of concise source code representations.

9. ACKNOWLEDGMENTS

Thanks to the participants; Wesley Weimer and the anonymous reviewers for their comments on the paper; and Pablo Duboue, Francisco Ferreira, Miryung Kim, Kathryn McKinley, and Christoph Treude for the discussion. This work was supported by NSERC and a McGill Tomlinson Scholarship.

10. REFERENCES

[1] M. Asaduzzaman, A. S. Mashiyat, C. K. Roy, and K. A. Schneider. Answering questions about unanswered questions of stack overflow. In *Proceedings of the Working Conference on Mining Software Repositories, Challenge Track*, pages 97–100, 2013.

[2] S. Bajracharya, J. Ossher, and C. Lopes. Leveraging usage similarity for effective retrieval of examples in code repositories. In *Proceedings of the International Symposium on the Foundations of Software Engineering*, pages 157–166, 2010.

[3] R. Buse and W. Weimer. Synthesizing API usage examples. In *Proceedings of the International Conference on Software Engineering*, pages 782–792, 2012.

[4] E. Cutrell and Z. Guan. What are you looking for? An eye-tracking study of information usage in web search. In *Proceedings of the Conference on Human Factors in Computing Systems*, pages 407–416, 2007.

[5] E. Duala-Ekoko and M. Robillard. Asking and answering questions about unfamiliar APIs: An exploratory study. In *Proceedings of the International Conference on Software Engineering*, pages 266–276, 2012.

[6] L. M. Eshkevari, V. Arnaoudova, M. Di Penta, R. Oliveto, Y.-G. Guéhéneuc, and G. Antoniol. An exploratory study of identifier renamings. In *Proceedings of the Working Conference on Mining Software Repositories*, pages 33–42, 2011.

[7] B. Fluri, M. Wursch, M. Pinzger, and H. C. Gall. Change distilling: Tree differencing for fine-grained source code change extraction. *Transactions on Software Engineering*, 33(11):725–743, 2007.

[8] M. Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Professional, 1999.

[9] T. Fritz, J. Ou, G. Murphy, and E. Murphy-Hill. A degree-of-knowledge model to capture source code familiarity. In *Proceedings of the International Conference on Software Engineering*, pages 385–394, 2010.

[10] S. Haiduc, J. Aponte, L. Moreno, and A. Marcus. On the use of automated text summarization techniques for summarizing source code. In *Proceedings of the Working Conference on Reverse Engineering*, pages 35–44, 2010.

[11] A. Hindle, E. T. Barr, Z. Su, M. Gabel, and P. Devanbu. On the naturalness of software. In *Proceedings of the International Conference on Software Engineering*, pages 837–847, 2012.

[12] H. Jing, R. Barzilay, K. McKeown, and M. Elhadad. Summarization evaluation methods: Experiments and analysis. In *AAAI Symposium on Intelligent Summarization*, pages 51–59, 1998.

[13] H. Jing and K. R. McKeown. The decomposition of human-written summary sentences. In *Proceedings of the Annual International Conference on Research and Development in Information Retrieval*, pages 129–136, 1999.

[14] J. Kim, S. Lee, S.-W. Hwang, and S. Kim. Enriching documents with examples: A corpus mining approach. *Transactions on Information Systems*, 31(1):1–27, 2013.

[15] J. Kupiec, J. Pedersen, and F. Chen. A trainable document summarizer. In *Proceedings of the Annual International Conference on Research and Development in Information Retrieval*, pages 68–73, 1995.

[16] D. Lawrie and D. Binkley. Expanding identifiers to normalize source code vocabulary. In *Proceedings of the International Conference on Software Maintenance*, pages 113–122, 2011.

[17] U. Lee, Z. Liu, and J. Cho. Automatic identification of user goals in web search. In *Proceedings of the International Conference on World Wide Web*, pages 391–400, 2005.

[18] C. Lewis and J. Rieman. *Task-Centered User Interface Design: A Practical Introduction*, chapter 5: Testing The Design With Users. Self-published, 1993. http://grouplab.cpsc.ucalgary.ca/saul/hci_topics/tcsd-book/contents.html.

[19] I. Mani. *Automatic summarization*. John Benjamins Publishing, 2001.

[20] N. Meng, M. Kim, and K. S. McKinley. Lase: locating and applying systematic edits by learning from examples. In *Proceedings of the International Conference on Software Engineering*, pages 502–511, 2013.

[21] A. Mockus and J. Herbsleb. Expertise browser: a quantitative approach to identifying expertise. In *Proceedings of the International Conference on Software Engineering*, pages 503–512, 2002.

[22] L. Moreno, J. Aponte, G. Sridhara, A. Marcus, L. Pollock, and K. Vijay-Shanker. Automatic generation of natural language summaries for Java classes. In *Proceedings of the International Conference on Program Comprehension*, pages 23–32, 2013.

[23] S. M. Nasehi, J. Sillito, F. Maurer, and C. Burns. What makes a good code example? A study of programming Q&A in StackOverflow. In *Proceedings of the International Conference on Software Maintenance*, pages 25–34, 2012.

[24] A. Nenkova and R. Passonneau. Evaluating content selection in summarization: The pyramid method. In *Proceedings of the Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pages 145–152, 2004.

[25] D. Radev, E. Hovy, and K. McKeown. Introduction to the special issue on summarization. *Computational Linguistics*, 28(4):399–408, 2002.

[26] S. Rastkar, G. C. Murphy, and A. W. Bradley. Generating natural language summaries for crosscutting source code concerns. In *Proceedings of the International Conference on Software Maintenance*, pages 103–112, 2011.

[27] M. Reape and C. Mellish. Just what is aggregation anyway. In *Proceedings of the European Workshop on Natural Language Generation*, pages 20–29, 1999.

[28] E. Reiter and R. Dale. *Building natural language generation systems*. MIT Press, 2000.

[29] M. Robillard and R. DeLine. A field study of API learning obstacles. *Empirical Software Engineering*, 16(6):703–732, 2011.

[30] P. Rodeghero, C. McMillan, P. W. McBurney, N. Bosch, and S. D'Mello. Improving automated source

code summarization via an eye-tracking study of programmers. In *Proceedings of the International Conference on Software Engineering*, pages 390–401, 2014.

[31] C. B. Seaman. Qualitative methods in empirical studies of software engineering. *Transactions on Software Engineering*, 25(4):557–572, 1999.

[32] S. Sim, R. Gallardo-Valencia, K. Philip, M. Umarji, M. Agarwala, C. Lopes, and S. Ratanotayanon. Software reuse through methodical component reuse and amethodical snippet remixing. In *Proceedings of the Conference on Computer-Supported Cooperative Work*, pages 1361–1370, 2012.

[33] G. Sridhara, E. Hill, D. Muppaneni, L. Pollock, and K. Vijay-Shanker. Towards automatically generating summary comments for Java methods. In *Proceedings of the International Conference on Automated Software Engineering*, pages 43–52, 2010.

[34] J. Stylos and B. Myers. Mica: A web-search tool for finding API components and examples. In *Proceedings of the Symposium on Visual Languages and Human-Centric Computing*, pages 195–202, 2006.

[35] S. Subramanian and R. Holmes. Making sense of online code snippets. In *Proceedings of the Working Conference on Mining Software Repositories, Challenge Track*, pages 85–88, 2013.

[36] E. R. Tufte. *Beautiful evidence*. Graphics Press, Cheshire, CT, 2006.

[37] R. White, J. Jose, and I. Ruthven. A task-oriented study on the influencing effects of query-biased summarisation in web searching. *Information Processing & Management*, 39(5):707–733, 2003.

[38] A. T. T. Ying and M. P. Robillard. Code fragment summarization. In *Proceedings of the Joint Meeting of the European Software Engineering Conference and the International Symposium on the Foundations of Software Engineering, New Ideas Track*, pages 655–658, 2013.

[39] H. Zhong, T. Xie, L. Zhang, J. Pei, and H. Mei. MAPO: Mining and recommending API usage patterns. In *Proceedings of the European Conference on Object-Oriented Programming*, pages 318–343, 2009.