

Automatic Documentation Generation via Source Code Summarization of Method Context

Paul W. McBurney and Collin McMillan
Department of Computer Science and Engineering
University of Notre Dame
Notre Dame, IN, USA
{pmcburne, cmc}@nd.edu

ABSTRACT

A documentation generator is a programming tool that creates documentation for software by analyzing the statements and comments in the software's source code. While many of these tools are manual, in that they require specially-formatted metadata written by programmers, new research has made inroads towards automatic generation of documentation. These approaches work by stitching together keywords from the source code into readable natural language sentences. These approaches have been shown to be effective, but carry a key limitation: the generated documents do not explain the source code's context. They can describe the behavior of a Java method, but not why the method exists or what role it plays in the software. In this paper, we propose a technique that includes this context by analyzing how the Java methods are invoked. In a user study, we found that programmers benefit from our generated documentation because it includes context information.

Categories and Subject Descriptors

D.2 [Software]: Software Engineering; D.2.9 [Software Engineering]: Management—*Productivity*

General Terms

Algorithms, Documentation

Keywords

Source code summarization

1. INTRODUCTION

Different studies of program comprehension show that programmers rely on good software documentation. [12, 24, 29, 52]. Unfortunately, manually-written documentation is notorious for being incomplete, either because it is very time-consuming to create [7, 21], or because it must constantly be updated [11, 19, 41]. One result has been the invention of the *documentation generator*. A documentation generator is a programming tool that creates documentation for software by analyzing the statements and com-

ments in the software's source code. The key advantage is that they relieve programmers of many tedious tasks while writing documentation. They offer a valuable opportunity to improve and standardize the quality of documentation.

Still, a majority of documentation generators are manual. They need considerable human intervention. Prominent examples include Doxygen [54] and JavaDoc [25]. These tools streamline the task of writing documentation by standardizing its format and presentation. But, they rely on programmers to write the documentation's content (in particular, a *summary* of each function or method) as specially-formatted metadata in the source code comments. The tools cannot generate documentation without this metadata. The burden of writing the documentation still lies with the programmers.

Recent research has made inroads towards automatic generation of natural language descriptions of software [2, 31, 34, 46–48, 55]. In particular, work by Sridhara *et al.* can form natural language summaries of Java methods [46]. The summaries can then be aggregated to create the software's documentation. The technique works by first selecting a method's most important statements, and then extracting keywords from the identifier names in those statements. Next, a natural language generator stitches the keywords into English sentences. Finally, these sentences are used to make a method summary. The process is automatic; so long as the source code contains meaningful identifiers, the summaries will describe the main behaviors of a given Java method.

What is missing from the method summaries is information about the *context* which surrounds the method being summarized. The context includes the dependencies of the method, and any other methods which rely on the output of the method [26]. A method's context is important for programmers to know because it helps answer questions about *why* a method exists and what role it plays in the software [8, 42, 43]. Because they summarize only select statements within a method, existing techniques will supply only limited context about a method.

In this paper, we hypothesize that existing documentation generators would be more effective if they included information from the context of the methods, in addition to the data from within the methods. We define “more effective” in terms of three criteria: programmers find the documentation's method summaries to be more helpful in understanding 1) what the methods do internally, 2) why the methods exist, and 3) how to use the methods. To test our hypothesis, we introduce a novel technique to automatically generate documentation that includes context. We then perform a case study with 12 Java programmers as participants.

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.

ICPC'14, June 2–3, 2014, Hyderabad, India
Copyright 2014 ACM 978-1-4503-2879-1/14/06...\$15.00
<http://dx.doi.org/10.1145/2597008.2597149>

During the study, the participants evaluated two configurations of our technique in comparison to documentation generated by a state-of-the-art solution [46]. In one configuration, the documentation consisted only of our generated summaries. In the second configuration, the documentation contained both our summaries and the state-of-the-art summaries. We found that our summaries were in general of higher quality and made a key contribution by providing more-thorough contextual information. The programmers consistently rated our summaries as more-helpful in understanding why given Java methods exist, and how to use them, than the state-of-the-art method.

Our tool works by collecting contextual data about Java methods from the source code, namely method calls, and then using the keywords from the context of a method to describe how that method is used. We use related work, the Software Word Usage Model by Hill *et al.* [17], to identify the parts of speech for the different keywords. We choose the contextual information to summarize using the algorithm PageRank, which we compute for the program’s call graph. We then build a novel Natural Language Generation system to interpret the keywords and infer meaning from the contextual information. Our system then generates a readable English description of the context for each method in a Java program. We will describe typical natural language generation systems and supporting technologies for our approach in Section 3, followed by our approach, our evaluation, and our evaluation results. Specifically, we contribute the following:

- A novel approach for generating natural language descriptions of source code. Our approach is different from previous approaches in that we summarize context as readable English text.
- A case study evaluating our approach and comparing it against documentation generated by a state-of-the-art approach. Our case study shows that our approach can improve existing documentation by adding important contextual information.
- A complete implementation of our approach for Java methods. For the purpose of reproducibility of our results, we have released our implementation to the public as an open-source project via our online appendix¹.

2. THE PROBLEM

The long-term problem we target in this paper is that much software documentation is incomplete [30], which costs programmers time and effort when trying to understand the software [12]. In Java programs, a typical form of this documentation is a list of inputs, outputs, and text summaries for every method in the software (e.g., JavaDocs). Only if these summaries are incomplete, do the programmers resort to reading the software’s source code [40]. What they must look for are clues in the source code’s *structure* about how the methods interact [18, 24, 51]. The term “structure” refers to both the control flow relationships and the data dependencies in source code. The structure is important because it defines the behavior of the program: methods invoke other methods, and the chain of these invocations defines how the program acts. In this paper, we aim to generate documentation that is more-complete than previous approaches, in

that our generated documentation contains structural information in each method’s summary.

We include this structural information from the *context* surrounding each method in the program. A method’s “context” is the environment in which the method is invoked [26]. It includes the statement which called the method, the statements which supplied the method’s inputs, and the statements which use the method’s output. *Context-sensitive program slicing* has emerged as one effective technique for extracting context [26]. Given a method, these techniques will return all statements in its context. However, some statements in the context are more-relevant to the method than other statements. This issue of relevance is important for this paper because we must limit the size of the text summaries, and therefore select only a small number of statements for use in generating the summaries.

Consider the manually-written examples of method summaries from NanoXML, a Java program for parsing XML, below. Item 1 is an example method we selected. It demonstrates how the default summary from documentation can be incomplete. In isolation, the method summary leaves a programmer to guess: What is the purpose of reading the character? For what is the character used? Why does the method even exist?

Example method with default summary from JavaDocs

- | | |
|------------------------|------------------|
| 1) StdXMLReader.read() | ◁ Method Name |
| “Reads a character.” | ◁ Method Summary |

Methods from context of example, with summaries from JavaDocs

- | | |
|---------------------------------|--|
| 2) XMLUnit.skipWhitespace() | “Skips whitespace from the reader.” |
| 3) XMLElement.addChild() | “Adds a child element.” |
| 4) StdXMLBuilder.startElement() | “This method is called when a new XML element is encountered.” |
| 5) StdXMLBuilder.addAttribute() | “This method is called when a new attribute of an XML element is encountered.” |

These questions can be answered by reading the context. The example method may be easier to understand when we know that Items 2 through 5 are in the example’s context. These methods are in the context because they all rely on the method `read` (e.g., they either call `read` directly, or call a method that calls `read`). We selected Items 2 through 5 above by hand to demonstrate this motivating example. However, in the remainder of this paper we will discuss how we automatically choose methods from the context and generate natural language descriptions, such as the one below in Item 6, for arbitrary Java methods. Our summaries provide programmers with key clues about how a method is used, and provide this information as English readable sentences:

Example method with summary including the method’s contextual information

- | | |
|------------------------|--|
| 6) StdXMLReader.read() | “This method reads a character. That character is used in methods that add child XML elements and attributes of XML elements. Called from method that skips whitespace.” |
|------------------------|--|

¹<http://www.nd.edu/~pmcburne/summaries/>

3. BACKGROUND

This section describes three supporting technologies for our work: the Software Word Usage Model (SWUM) [17], the design of Natural Language Generation (NLG) systems [39], and the algorithm PageRank [27]. These techniques were proposed and evaluated elsewhere. We emphasize them here because they are important concepts for our approach.

3.1 Software Word Usage Model

The Software Word Usage Model (SWUM) is a technique for representing program statements as sets of nouns, verbs, and prepositional phrases. SWUM works by making assumptions about different Java naming conventions, and using these assumptions to interpret different programs statements. Consider a method from NanoXML which has the signature `static String scanPublicId(StringBuffer, XML-Reader, char XMLEntityResolver)`. SWUM first splits the identifier names using the typical Java convention of camel case. Next, it reads verbs from the method as the starting word from the method identifier (e.g., “scan”). SWUM also extracts noun phrases, such as “public id”, and deduces a relationship of the nouns to the verbs. For example, “public id” is assumed to be the direct object of “scan” because it follows “scan” in the method identifier. Other nouns, such as “string” or “xml reader”, are read from the return types and arguments, and are interpreted under different assumptions. We direct readers to the relevant literature on SWUM for complete details [16, 17].

One strategy for using SWUM for text generation is to define templates of natural language sentences, and use the output from SWUM to fill these templates [46]. For example, a template for method call statements is “*action theme args and get return-type*”. The template may be further processed so that items such as *return-type* actually display as the variable name. Given a method call statement `systemID = XMLUtil.scanPublicID(publicID, reader, &, this.xmlEntityResolver)`; a summary for the statement is “scan public id and get system id”. To summarize an entire method from these summaries of statements, Sridhara *et al.* selected a subset of key statements by defining rules for which statements are typically the most-important (e.g., return or control-flow statements). A method summary was a combination of the summaries of these key statements.

3.2 Natural Language Generation Systems

The design of a Natural Language Generation (NLG) systems typically follows an architecture described by Reiter and Dale [39]. Figure 1 illustrates this architecture. Conceptually, the architecture is not complicated: a “communicative goal” is translated from a series of facts into readable natural language sentences, known as “surface text.” The NLG system has three main components, each of which is made up of several individual steps.

The first main component is the *Document Planner*. The input to this component is a list of facts that need to be communicated to a human reader. Through “content determination”, the document planner interprets the facts and creates “messages.” Messages are an intermediate representation between the communicative goal and readable text. For example, in a weather forecast generator such as FOG [14], facts about the temperature on given days result in a message offering an interpretation of those facts, e.g., that it is colder today than it was yesterday. After the messages are created,

“document structuring” takes place which sorts the messages into a sequence that makes sense to a human reader. This sequence of messages is known as the *document plan*.

The next main component, the *Microplanner*, decides which words will be used to describe each message. In “lexicalization”, the microplanner assigns specific words as parts of speech in a “phrase” about each message. Typically the subject, verb, and object for a given message are identified, along with any modifiers such as adjectives and adverbs. Next, two steps smooth the phrases so that they are more naturally read. “Reference generation” decides how nouns will be referred to in the phrases, such as whether to use a proper name or a pronoun. Finally, “aggregation” joins phrases based on how they are related, e.g., causally (joined by because) or via coordination (joined by and/or).

The final component of NLG is the *Surface Realizer*. The surface realizer generates natural language sentences from the phrases. Different grammar rules for the natural language dictate how the sentences should be formed. The surface realizer follows these rules to create sentences that contain the parts of speech and words given by the microplanner. These sentences are the surface text. They are human-readable descriptions of the information in the messages, interpreted from the facts given to the document planner, and in the order defined in the document plan.

3.3 PageRank

PageRank is an algorithm for approximating the importance of the nodes in a graph [27]. While a complete discussion of PageRank is beyond the scope of this paper, in general, PageRank calculates importance based on the number of edges which point to a given node as well as the importance of the nodes from which those edges originate. PageRank is well-known for its usefulness in ranking web pages for web search engines. However, PageRank has seen growing relevance in its applications in software engineering. In particular, a body of work has shown how PageRank can highlight important functions or methods in a software program [5, 20, 33, 38]. A common and effective strategy is to model a software program as a “call graph”: a graph in which the nodes are functions or methods, and the edges

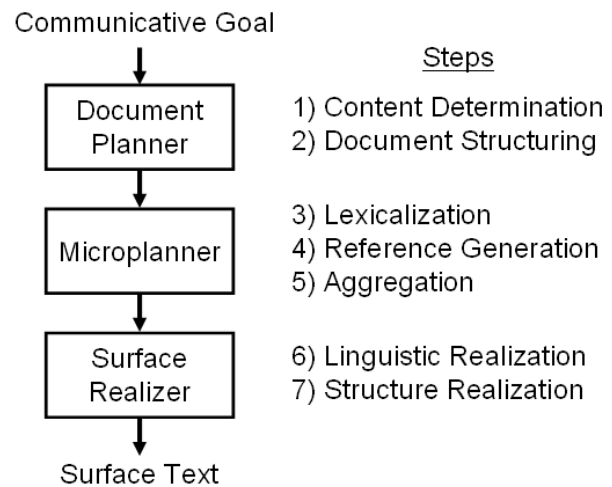


Figure 1: The typical design of a Natural Language Generation system as described by Reiter and Dale [39]. We built our NLG system around each of these seven steps.

are call relationships among the methods. Methods that are called many times or that are called by other important methods are ranked as more important than methods which are called rarely, and thus have few edges in the call graph. We follow this model of using PageRank for this paper.

4. APPROACH

This section describes the details of our approach, including each step of our natural language generation system. Generally speaking, **our approach creates a summary of a given method in three steps:** 1) use PageRank to discover the most-important methods in the given method’s context, 2) use data from SWUM to extract keywords about the actions performed by those most-important methods, and 3) use a custom NLG system to generate English sentences describing for what the given method is used.

The architecture of our approach is shown in Figure 2. In theory our system could summarize functions in many languages, but in this paper we limit the scope to Java methods. The data we collect about these Java methods is our “communicative goal” (see Section 3.2) and is the basis for the information we convey via NLG.

4.1 Data Collection

The comment generator requires three external tools to produce the necessary input data: SWUM, the call graph generator, and PageRank. SWUM parses the grammatical structure from the function and argument names in a method declaration. This allows us to describe the method based on the contents of its static features. Specifically, SWUM outputs the keywords describing the methods, with each keyword tagged with a part-of-speech (Figure 2, area 3). Next, we produce a call graph of the project for which we are generating comments. Our call graph² allows us to see where a method is called so that we can determine the method’s context (Figure 2, area 2). Finally, we obtain a PageRank value for every method by executing the PageRank algorithm with the procedure outlined in Section 3.3.

In addition to gleaning this information from the project to produce our comments, we also use the source code of the project itself. For every method call in the call graph, the Data Organizer searches through the code to find the statement that makes that call. The purpose of collecting these statements is to provide a concrete usage example to the programmer. The Data Organizer combines these example statements with the call graph and SWUM keywords to create the Project Metadata (Figure 2, area 4).

4.2 Natural Language Generation

This section covers our NLG system. Our system processes the Project Metadata as input (Figure 2, area 5), following each of the NLG steps shown in Figure 1.

Content Determination. We create four different types of “messages” (see Section 3.2) that represent information about a method’s context. While all message types may be downloaded from our online appendix, due to space limitations, we discuss only four representative messages here. First, a *Quick Summary Message* represents a brief, high-level action summarizing a whole method. For example, “skips whitespace in character streams.” We create these messages from the noun/verb labeling of identifier names

²Generated using java-callgraph, available via <https://github.com/gousiosg/java-callgraph>, verified 9/12/2013

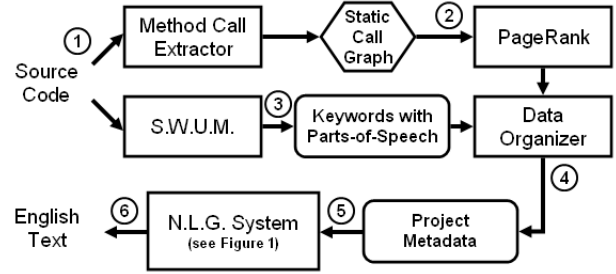


Figure 2: Overview of our approach.

extracted by SWUM from the method’s signature. Our system makes a simplifying assumption that all methods perform some action on some input. If the keyword associated with the input is labeled as a noun by SWUM, and the keyword associated with the method name is a verb, we assume that there is a verb/direct-object relationship between the method name and the input name. This relationship is recorded as a Quick Summary Message.

Another type of message is the *Importance Message*. The idea behind an importance message is to give programmers clues about how much time to spend reading a method. The importance message is created by interpreting both the PageRank value of the method and the PageRank values of all other methods. The importance message represents how high this value is above or below average. At the same time, an importance message will trigger our NLG system to include more information in the method’s description if the method is ranked highly (see Aggregation below).

A third message type is the *Output Usage Message*. This message conveys information about the method’s output, such as “the character returned by this method is used to skip whitespace in character streams.” Our system uses data from quick summary messages, importance messages, and the call graph to create output usage messages. Given a method, our system creates an output usage message by first finding the methods in the call graph which depend on the given method. Then, it picks the two of those methods with the highest PageRank. It uses the quick summary message from those two methods to describe how the output is used.

The last message type we will examine in detail is the *Use Message*. This message serves to illustrate how a programmer can use the method by highlighting a specific example in the code. For example, one message we generated was “the method can be used in an assignment statement ; for example: `Date releaseDate=getReleaseDate();`.” Our system uses the call graph to find a line of code that calls the method for which we are generating the message. It then classifies, based on static features with the line of code, whether the calling statement is a conditional, iteration, assignment, or procedural statement. If a source code example cannot be found, the Use Message is omitted.

Document Structuring. After generating the initial messages in the content determination phase, we organize all the messages into a single document plan. We use a templated document plan where messages occur in a pre-defined order: Quick Summary Messages, Return Messages, Output Used Messages, Called Messages, Importance Messages, and then Use Messages. Note that this order may change during the Aggregation phase below.

Lexicalization. Each type of message needs a different type of phrase to describe it. This section will describe how we decide on the words to be used in each of those phrases, for the four message types described under Content Determination. Note that the phrases we generate are not complete sentences; they will be grouped with other phrases during Aggregation and formed into sentences during realization.

The *Quick Summary Message* records a verb/direct-object relationship between two words extracted by SWUM. The conversion to a sentence is simple in this case: the verb becomes the verb in the sentence, and likewise for the direct-object. The subject is assumed to be “the method”, but is left out for brevity. To give the reader further information about the method’s purpose, we add the input parameter type as an indirect object using the preposition “in”.

We create a phrase for an *Output Usage Message* by setting the object as the return type of the method, and the verb as “is”. The subject is the phrase generated from the Quick Summary Message. We set the voice of the phrase to be passive. We decided to use passive voice to emphasize how the return data is used, rather than the contents of the Quick Summary Message. An example of the phrase we output is under the Content Determination section.

The *Use Message* is created with the subject “this method”, the verb phrase “can be used”, and appending the prepositional phrase “as a *statement type*,”. Statement type is pulled from the data structures populated in our content determination step. Additionally, we append a second dependent clause “for example: *code*”.

Reference Generation and Aggregation. During Aggregation, we create more-complex and readable phrases from the phrases generated during Lexicalization. Our system works by looking for patterns of message types, and then grouping the phrases of those messages into a sentence. For example, if two Output Usage Messages are together, and both refer to the same method, then the phrases of those two messages are conjoined with an “and” and the subject and verb for the second phrase is hidden. In another case, if a Quick Summary Message follows a Quick Summary Message for a different method, then it implies that the messages are related, and we connect them using the preposition “for”. The result is a phrase such as “skips whitespace in character streams for a method that processes xml”. Notice that Reference Generation occurs alongside Aggregation. Rather than hiding the subject in the phrase “processes xml”, we make it explicit as “method” and non-specific using the article “a” rather than “the.” Due to space limitations, we direct readers to our online appendix for a complete listing of the Aggregation techniques we follow.

Surface Realization. We use an external library, *simplenlg* [13], to realize complete sentences from the phrases formed during Aggregation. In the above steps, we set all words and parts-of-speech and provided the structure of the sentences. The external library follows English grammar rules to conjugate verbs, and ensure that the word order, plurals, and articles are correct. The output from this step is the English summary of the method (Figure 2, area 6).

5. EXAMPLE

In this section, we explore an example of how we form a summary for a specific method. We will elaborate on how we use SWUM, call graph, PageRank, and source code to form our messages.

Consider `getResult()` from `StdXMLBuilder.java` in NanoXML. The method’s signature, `public Object getResult()`, is parsed by SWUM which will tell us the verb is “get” and the object is “result.” Additionally, it will note the return type as “object.” This will be used to generate the *Quick Summary Message* “This method gets the result and returns an Object.” Then, using the call graph, we determine that the top two methods (as scored by PageRank) that call `getResult()` are `scanData()` and `parse()`. Initially, in the document planning phase, we generate two separate messages, one using the SWUM information for each function. However, these are combined in the aggregation step with the conjunction “and”, and eventually produces the *Output Usage Message* “That Object is used by methods that scans the data and that parses the std XML parser.”

The last message we generate is the *Use Message*. We search through the most important calling method, which in this case is `scanData()`. We take a line of code that calls `getResult()`, and determine based on its content whether it is a conditional, iteration, assignment, or procedural statement. Using this information, we generate the *Use Message* “The method can be used in an iteration statement ; for example: `while ((!this.reader.atEOF()) && (this.builder.getResult() == null)) {` ”. Each of these messages are then appended together to make the final summary.

6. EVALUATION

Our evaluation compares our approach to the state-of-the-art approach described by Sridhara *et al.* [46]. The objective of our evaluation is three-fold: 1) to assess the degree to which our summaries meet the quality of summaries generated by a state-of-the-art solution, 2) to assess whether the summaries provide useful contextual information about the Java methods, and 3) to determine whether the generated summaries can be used to improve, rather than replace, existing documentation.

Assessing Overall Quality. One goal of our evaluation is to quantify any difference in quality between our approach presented in this paper and the existing state-of-the-art approach, and to determine in what areas the quality of the summaries can be most improved. To assess quality, we ask the three following Research Questions (RQs):

- RQ₁* To what degree do the summaries from our approach and the state-of-the-art approach differ in overall accuracy?
- RQ₂* To what degree do the summaries from our approach and the state-of-the-art approach differ in terms of missing important information?
- RQ₃* To what degree do the summaries from our approach and the state-of-the-art approach differ in terms of including unnecessary information?

These Research Questions are derived from two earlier evaluations of source code summarization [34,46], where the “quality” of the generated comments was assessed in terms of accuracy, content adequacy, and conciseness. Content adequacy referred to whether there was missing information, while conciseness referred to limiting unnecessary information in the summary. This strategy for evaluating generated comments is supported by a recent study of source code comments [49] in which quality was modeled as a combination of factors correlating to accuracy, adequacy, and conciseness.

Assessing Contextual Information. Contextual information about a method is meant to help programmers understand the behavior of that method. But, rather than describe that behavior directly from the internals of the method itself, context explains how that method interacts with other methods in a program. By reading the context, programmers then can understand what the method does, why it exists, and how to use it (see Section 2). Therefore, we study these three Research Questions:

RQ₄ Do the summaries help programmers understand what the methods do internally?

RQ₅ Do the summaries help programmers understand why the methods exist?

RQ₆ Do the summaries help programmers understand how to use the methods?

The rationale behind *RQ₄* is that a summary should provide programmers with enough details to understand the most-important internals of the method—for example, the type of algorithm the method implements—without forcing them to read the method’s source code. Our summaries aim to include this information solely from the context. If our summaries help programmers understand the methods’ key internals, it means that this information came from the context. For *RQ₅*, a summary should help programmers understand why the method is important to the program as a whole. For example, the programmers should be able to know, from reading the summary, what the consequences might be of altering or removing the method. Likewise, for *RQ₆*, the summary should explain the key details about how a programmer may use the method in his or her own code.

Orthogonality. While the ultimate goal of this research is to generate documentation purely from data in the source code, we also aim to improve existing documentation by adding contextual information. In particular, we ask:

RQ₇ Do the summaries generated by our solution contain orthogonal information to the information already in the summaries from the state-of-the-art solution?

The idea behind this RQ is that to improve existing summaries, the generated summaries should contribute new information, not merely repeat what is already in the summaries. We generate summaries by analyzing the context of methods, so it is plausible that we add information from this context, which does not exist in the summaries from the state-of-the-art solution.

6.1 Cross-Validation Study Methodology

To answer our Research Questions, we performed a cross-validation study in which human experts (e.g., Java programmers) read the source code of different Java methods, as well as summaries of those methods, for three different rounds. For each method and summary, the experts answered eight questions that covered various details about the summary. Table 2 lists these questions. The first six correspond to each of the Research Questions above, and were multiple choice. The final two were open-ended questions; we study the responses to these two questions in a qualitative evaluation in Section 8.

In the cross-validation study design, we rotated the summaries and Java methods that the human evaluators read.

Table 1: The cross-validation design of our user study. Different participants read different summaries for different programs.

Round	Group	Summary	Program 1	Program 2
1	A	Our	NanoXML	Jajuk
	B	S.O.T.A.	Siena	JEdit
	C	Combined	JTopas	JHotdraw
2	A	Combined	Siena	Jajuk
	B	Our	JTopas	JEdit
	C	S.O.T.A.	NanoXML	JHotdraw
3	A	S.O.T.A.	JTopas	Jajuk
	B	Combined	NanoXML	JEdit
	C	Our	Siena	JHotdraw

Table 2: The questions we ask during the user study. The first six are answerable as “Strongly Agree”, “Agree”, “Disagree”, and “Strongly Disagree.” The last two are open-ended.

<i>Q₁</i>	Independent of other factors, I feel that the summary is accurate.
<i>Q₂</i>	The summary is missing important information, and that can hinder the understanding of the method.
<i>Q₃</i>	The summary contains a lot of unnecessary information.
<i>Q₄</i>	The summary contains information that helps me understand what the method does (e.g., the internals of the method).
<i>Q₅</i>	The summary contains information that helps me understand why the method exists in the project (e.g., the consequences of altering or removing the method).
<i>Q₆</i>	The summary contains information that helps me understand how to use the method.
<i>Q₇</i>	In a sentence or two, please summarize the method in your own words.
<i>Q₈</i>	Do you have any general comments about the given summary?

The purpose of this rotation was to ensure that all evaluators would read summaries from each different approach for several different Java programs, and to mitigate any bias from the order in which the approaches and methods were presented [32]. Table 1 shows our study design in detail. Upon starting the study, each participant was randomly assigned to one of three groups. Each of those groups was then assigned to see one of three types of summary: summaries from our approach, summaries from the state-of-the-art approach, or both summaries at the same time.

6.2 Subject Java Programs

The summaries in the study corresponded to Java methods from six different subject Java programs, listed in Table 3. We selected these programs for a range of size (5 to 117 KLOC, 318 to 7161 methods) and domain (including text editing, multimedia, and XML parsing, among others). During the study, participants were assigned to see methods

from four of these applications. During each of three different rounds, we rotated one of the programs that the groups saw, but retained the fourth program. The reason is so that the group would evaluate different types of summaries for different programs, but also evaluate different types of summaries from a single application. From each application, we pre-selected (randomly) a pool of 20 methods from each application. At the start of each round, we randomly selected four methods from the pool for the rotated application, and four from the fixed application. Over three rounds, participants read a total of 24 methods. Because the methods were selected randomly from a pool, the participants did not all see the same set of 24 methods. The programmers could always read and navigate the source code for these applications, though we removed all comments from this code to avoid introducing a bias from these comments.

6.3 Participants

We had 12 participants in our study. Nine were graduate students and from the Computer Science and Engineering Department at the University of Notre Dame. The remaining three were professional programmers from two different organizations, not listed due to our privacy policy.

6.4 Metrics and Statistical Tests

Each of the multiple choice questions could be answered as “Strongly Agree”, “Agree”, “Disagree”, or “Strongly Disagree.” We assigned a values to these answers as 4 for “Strongly Agree”, 3 for “Agree”, 2 for “Disagree”, and 1 for “Strongly Disagree.” For questions 1, 4, 5, and 6, higher values indicate stronger performance. For questions 2 and 3, lower values are preferred. We aggregated the responses for each question by approach. For example, all responses to question 1 for the summaries from our approach, and all responses to question 1 for the summaries from the state-of-the-art approach.

To determine the statistical significance of the differences in these groups, we used the two-tailed Mann-Whitney U test [44]. The Mann-Whitney test is non-parametric, and it does not assume that the data are normally distributed. However, the results of these tests may not be accurate if the number of participants in the study is too small. Therefore, to confirm statistical significance, we use the procedure outlined by Morse [35] to determine the minimum population size for a tolerated p value of 0.05. We calculated these minimum sizes using observed, not expected, values of U .

6.5 Threats to Validity

As with any study, our evaluation carries threats to validity. We identified two main sources of these threats. First, our evaluation was conducted by human experts, who may be influenced by factors such as stress, fatigue, or variations

Table 3: The six Java programs used in our evaluation. KLOC reported with all comments removed. All projects are open-source.

	Methods	KLOC	Java Files
NanoXML	318	5.0	28
Siena	695	44	211
JTopas	613	9.3	64
Jajuk	5921	70	544
JEdit	7161	117	555
JHotdraw	5263	31	466

in programming experience. We attempted to mitigate these threats through our cross-validation study design, which altered the order in which the participants viewed the Java methods and summaries. We also recruited our participants from a diverse body of professionals and students, and confirmed our results with accepted statistical testing procedures. Still, we cannot guarantee that a different group of participants would not produce a different result.

Another source for a threat to validity is the set of Java programs we selected. We chose a variety of applications of different sizes and from different domains. In total, we generated summaries for over 19,000 Java methods from six projects, and randomly selected 20 of these methods from each project to be included in the study (four to twelve of which were ultimately shown to each participant). Even with this large pool of methods, it is still possible that our results would change with a different projects. To help mitigate this threat, we have released our tool implementation and all evaluation data in an online appendix, so that other researchers may reproduce our work in independent studies.

7. EMPIRICAL RESULTS

This section reports the results of our evaluation. First, we present our statistical process and evidence. Then, we explain our interpretation of this evidence and answer our research questions.

7.1 Statistical Analysis

The main independent variable was the type of summary rated by the participants: summaries generated by our solution, summaries from the state-of-the-art solution, or both presented together. The dependent variables were the ratings for each question: 4 for “Strongly Agree” to 1 for “Strongly Disagree”.

For each question, we compare the mean of the participants’ ratings for our summaries to the summaries from the state-of-the-art approach. We also compare the ratings given when both summaries were shown, versus only the state-of-the-art summaries. We compared these values using the Mann-Whitney test (see Section 6.4). Specifically, we posed 12 hypotheses of the form:

H_n The difference in the reported ratings of the responses for Q_m is not statistically-significant.

where n ranges from 1 to 12, and m ranges from 1 to 6, depending on which question is being tested. For example, in H_{11} , we compare the answers to Q_5 for the state-of-the-art summaries to the answers to Q_5 for the combined summaries.

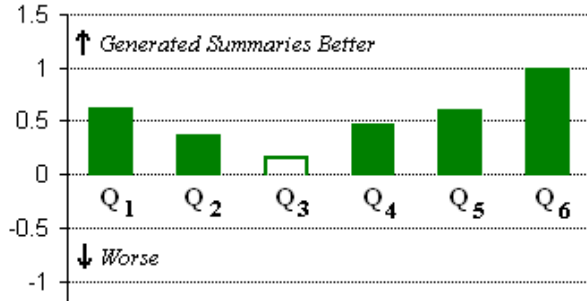
Table 4 shows the rejected hypotheses (e.g., the means with a statistically-significant difference). We made a decision to reject a hypothesis only when three criteria were met. First, $|Z|$ must be greater than Z_{crit} . Second, p must be less than the tolerated error 0.05. Finally, the calculated minimum number of participants (N_{min}) must be less than or equal to the number of participants in our study (13). Note that due to space limitations we do not include values for the four hypotheses for which we do not have evidence to reject. For reproducibility purposes, these results are available at our online appendix.

7.2 Interpretation

Figure 3 showcases the key evidence we study in this evaluation. We use this evidence to answer our Research Questions along the three areas highlighted in Section 6.

Table 4: Statistical summary of the results for the participants’ ratings for each question. “Samp.” is the number of responses for that question for a given summary type, for all rounds. Mann-Whitney test values are U , U_{expt} , and U_{vari} . Decision criteria are Z , Z_{crit} , and p . N_{min} is the calculated minimum number of participants needed for statistical significance.

H	Q	Summary	n	\tilde{x}	μ	Vari.	U	U_{expt}	U_{vari}	Z	Z_{crit}	p	N_{min}	Decision
H_1	Q_1	Our S.O.T.A.	65 59	3 3	3.015 2.390	0.863 0.863	1223	1917	34483	3.74	1.96	<1e-3	2	Reject
H_2	Q_2	Our S.O.T.A.	65 58	3 3	2.492 2.862	0.973 1.139	2272	1885	36133	2.03	1.96	0.042	2	Reject
H_3	Q_3	Our S.O.T.A.	65 59	2 2	1.815 1.983	0.497 0.982	2011	1917	34204	0.503	1.96	0.615	15	Not Reject
H_4	Q_4	Our S.O.T.A.	65 59	3 3	2.877 2.407	0.641 0.832	1410	1917	34475	2.736	1.96	0.006	4	Reject
H_5	Q_5	Our S.O.T.A.	65 58	3 3	2.585 1.983	0.809 0.930	1251	1885	35789	3.351	1.96	0.001	2	Reject
H_6	Q_6	Our S.O.T.A.	65 58	3 3	2.769 1.776	0.649 0.773	799	1885	35447	5.771	1.96	<1e-3	3	Reject
H_7	Q_1	Combined S.O.T.A.	59 59	3 3	2.847 2.390	0.580 0.863	1266	1741	29026	2.788	1.96	0.005	3	Reject
H_8	Q_2	Combined S.O.T.A.	59 58	2 3	2.322 2.862	0.843 1.139	2219	1711	31205	2.876	1.96	0.004	3	Reject
H_9	Q_3	Combined S.O.T.A.	59 59	2 2	2.542 1.983	1.149 1.149	1227	1741	31705	2.884	1.96	0.004	5	Reject
H_{10}	Q_4	Combined S.O.T.A.	58 59	3 3	2.879 2.407	0.564 0.832	1241	1711	27964	2.811	1.96	0.005	5	Reject
H_{11}	Q_5	Combined S.O.T.A.	59 58	3 2	2.508 1.983	0.634 0.930	1176	1711	30423	3.064	1.96	0.002	4	Reject
H_{12}	Q_6	Combined S.O.T.A.	59 58	3 2	2.746 1.776	0.503 0.773	705	1711	29972	5.814	1.96	<1e-3	2	Reject



(a) Our vs. S.O.T.A. Summaries



(b) Combined vs. S.O.T.A. Summaries

Figure 3: Performance comparison of the summaries. The chart shows the difference in the means of the responses to each question. For example in (a), the mean of Q_5 for our approach is 0.602 higher than for the state-of-the-art summaries. The sign is reversed for Q_2 and Q_3 because lower scores, not higher scores, are better values for those questions. Solid bars indicate differences which are statistically-significant. In general, our summaries were more accurate and provided more-thorough contextual information.

Overall Quality. The summaries from our approach are superior in overall quality to the summaries from the state-of-the-art approach. Figure 3(a) shows the difference in the means of the responses for survey questions. Questions Q_1 through Q_3 refer to aspects of the summaries related to overall quality, in particular to our Research Questions RQ_1 to RQ_3 . In short, participants rated our summaries as more accurate and as missing less required information by a statistically-significant margin. While these results are encouraging progress, they nevertheless still point to a need to improve. In Section 8, we explore what information that the participants felt was unnecessary in our summaries.

Contextual Information. The summaries from our approach included more contextual information than the state-of-the-art summaries. The differences in responses for questions Q_4 , Q_5 , and Q_6 are higher for our summaries by a statistically-significant margin. These results mean that, in comparison to the state-of-the-art summaries, our summaries helped the programmers understand *why* the methods exist and *how to use* those methods. Therefore, we answer RQ_4 , RQ_5 , and RQ_6 with a positive result. The answers to these research questions point to an important niche filled by our approach: the addition of contextual information to software documentation.

Orthogonality. We found substantial evidence showing that our summaries improve the state-of-the-art summaries. When participants read both types of summary for a given method, the responses for Q_4 , Q_5 , and Q_6 improved by a significant margin, pointing to an increase in useful contextual information in the documentation. Overall quality did not decrease by a significant margin compared to when only our solutions were given, except in terms of unnecessary information added. Consider Figure 3(b): Accuracy and missing information scores showed similar improvement. While the combined summaries did show a marked increase in unnecessary information, we still find evidence to positively answer RQ_7 : the information added to state-of-the-art summaries by our approach is orthogonal. This answer suggests that our approach can be used, after future work to reduce unnecessary information, to improve existing documentation.

8. QUALITATIVE RESULTS

Participants in the evaluation study had the opportunity to write an opinion about each summary (see Q_8 in Table 2). In this section, we explore these opinions for feedback on our approach and directions for future work.

One of the results in our study was the significantly worse performance of Q_3 in the combined comments, suggesting an increase in the amount of unnecessary information. Several user comments from our survey note concerns of repetitious information, as well as difficulties in processing the longer comments that result from the combination.

- “The description is too verbose and contains too many details.”
- “The summary contains too much information and confuses the purpose of the method...”
- “The summary seems accurate but too verbose.”
- “Too much information, I cannot understand the comment.”

Another result is the increase in the scores for Q_5 and Q_6 , which deal with how a programmer can use the method within the system. This increase appears to be due to the *Use Message*. Several users noted a lack of any form of usage message in the state-of-the-art approach. A selection of these comments follows.

- “Nice and concise, but lacking information on uses...”
- “The summary is clear. An example is expected.”
- “The summary...does not tell me where the method is called or how it is used.”

Additionally, in a method summary from our approach that did not generate a *Use Message*, a participant noted “I feel that an example should be provided.” However, one participant in our study had a largely negative opinion of the *Use Message*. This participant repeatedly referred to the “last sentence” (the *Use Message*) as “unnecessary”, even stating “Assume every one of these boxes comments about removing the last line of the provided comment.”

Participants often felt the state-of-the-art approach lacked critical information about the function. Comments indicating a lack of information appeared consistently from many participants. The following comments (each from a different participant) support this criticism:

- “A bit sparse and missing a lot of information.”
- “Comment details the inner workings but provides no big picture summary.”
- “Only provides a detail for one of the possible branches.”
- “It seems the summary is generated only based on the last line of the method.”

These comments occur more frequently with the state-of-the-art compared to our approach. A possible reason for this is our approach focuses much more on method interactions (e.g., method calls), and avoids the internal details of the function. By contrast, the state-of-the-art approach focuses on a method’s internal execution, selecting a small subset of statements to use in the summary. Participants felt this selection often leaves out error checking and alternate branches, focusing too narrowly on particular internal operations while ignoring others.

Several of our generated summaries and the state-of-the-art generated summaries had grammar issues that distracted users. Additionally, the state-of-the-art approach often selected lines of source code, but did not generate English summaries for those lines. Several users commented on these issues, noting that it made the summaries either impossible or difficult to understand. Our aim is to correct these issues going forward with refinement of our NLG tool.

Another common theme of participant comments in both our approach and the state-of-the-art centered on function input parameters. Many participants felt an explanation of input parameters was lacking in both approaches, as well as the combination approach. A selection of these comments follows. These comments were selected from our approach, the state-of-the-art, and the combined approach respectively:

- “The input parameters publicID and systemID are not defined – what are they exactly?”
- “The summary could mention the input required is the path for the URL”
- “... It would be better if the summary described the types of the inputs...”

9. RELATED WORK

The related work closest to our approach is detailed in a recent thesis by Sridhara [45]. In Section 3, we summarized certain elements of this work that inspired our approach. Two aspects we did not discuss are as follows. First, one approach creates summaries of the “high level actions” in a method [47]. A high level action is defined as a behavior at a level of abstraction higher than the method. The approach works by identifying which statements in a method implement that behavior, and summarizing only those statements. A second approach summarizes the role of the parameters to a method. This approach creates a description of key statements related to the parameter inside the method. Our approach is different from both of these approaches in that we create summaries from the context of the method – that is, where the method is invoked. We help programmers understand the role the method plays in the software.

There are a number of other approaches that create natural language summaries of different software artifacts and behaviors. Moreno *et al.* describe a summarization technique for Java classes that match one of 13 “stereotypes.” [34]

The technique selects statements from the class based on this stereotype, and then uses the approach by Sridhara [46] to summarize those statements. Work by Buse *et al.* focuses on Java exceptions [3]. Their technique is capable of identifying the conditions under which an exception will be thrown, and producing brief descriptions of those conditions. Recent work by Zhang *et al.* performs a similar function by explaining failed tests [56]. That approach modifies a failed test by swapping different expressions into the test to find the failure conditions. Summary comments of those conditions are added to the test. Another area of focus has been software changes. One approach is to improve change log messages [4]. Alternatively, work by Kim *et al.* infers change rules, as opposed to individual changes, that explain the software's evolution [23]. The technique can summarize the high-level differences between two versions of a program. Another approach, developed by Panichella *et al.*, uses external communications between developers, such as bug reports and e-mails, and structures them to produce source code documentation [37].

The key difference between our approach and these existing approaches is that we summarize the context of the source code, such as how the code is called or the output is used. Structural information has been summarized before, in particular by Murphy [36], in order to help programmers understand and evolve software. Murphy's approach, the software reflexion model, notes the connections between low-level software artifacts in order to point out connections between higher-level artifacts. There are techniques which give programmers some contextual information by listing the important keywords from code. For example, Haiduc *et al.* use a Vector Space Model to rank keywords from the source code, and present those keywords to programmers [15]. The approach is based on the idea that programmers read source code cursorily by reading these keywords, and use that information to deduce the context behind the code. Follow-up studies have supported the conclusions that keyword-list summarization is useful to programmers [1], and that VSM is an effective strategy for extracting these keywords [6, 9].

Tools such as Jadeite [51], Apatite [10], and Mica [50] are related to our approach in that they add API usage information to documentation of those APIs. These tools visualize the usage information as part of the interface for exploring or locating the documentation. We take a different strategy by summarizing the information as natural language text. What is similar is that this work demonstrates a need for documentation to include the usage data, as confirmed by studies of programmers during software maintenance [22, 28, 53].

10. CONCLUSION

We have presented a novel approach for automatically generating summaries of Java methods. Our approach differs from previous approaches in that we summarize the *context* surrounding a method, rather than details from the internals of the method. We use PageRank to locate the most-important methods in that context, and SWUM to gather relevant keywords describing the behavior of those methods. Then, we designed a custom NLG system to create natural language text about this context. The output is a set of English sentences describing why the method exists in the program, and how to use the method. In a cross-validation study, we compared the summaries from our approach to

summaries written by a state-of-the-art solution. We found that our summaries were superior in quality and that our generated summaries fill a key niche by providing contextual information. That context is missing from the state-of-the-art summaries. Moreover, we found that by combining our summaries with the state-of-the-art summaries, we can improve existing software documentation. Finally, the source code for our tool's implementation and evaluation data are publically available for future researchers.

11. ACKNOWLEDGMENTS

The authors would like to thank Dr. Emily Hill for providing key assistance with the SWUM tool. We also thank and acknowledge the Software Analysis and Compilation Lab at the University of Delaware for important help with the state-of-the-art summarization tool. Finally, we thank the 12 participants who spent time and effort completing our evaluation.

12. REFERENCES

- [1] J. Aponte and A. Marcus. Improving traceability link recovery methods through software artifact summarization. In *Proceedings of the 6th International Workshop on Traceability in Emerging Forms of Software Engineering*, TEFSE '11, pages 46–49, New York, NY, USA, 2011. ACM.
- [2] H. Burden and R. Heldal. Natural language generation from class diagrams. In *Proceedings of the 8th International Workshop on Model-Driven Engineering, Verification and Validation*, MoDeVVA, pages 8:1–8:8, New York, NY, USA, 2011. ACM.
- [3] R. P. Buse and W. R. Weimer. Automatic documentation inference for exceptions. In *Proceedings of the 2008 international symposium on Software testing and analysis*, ISSTA '08, pages 273–282, New York, NY, USA, 2008. ACM.
- [4] R. P. Buse and W. R. Weimer. Automatically documenting program changes. In *Proceedings of the IEEE/ACM international conference on Automated software engineering*, ASE '10, pages 33–42, New York, NY, USA, 2010. ACM.
- [5] W.-K. Chan, H. Cheng, and D. Lo. Searching connected api subgraph via text phrases. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, FSE '12, pages 10:1–10:11, New York, NY, USA, 2012. ACM.
- [6] A. De Lucia, M. Di Penta, R. Oliveto, A. Panichella, and S. Panichella. Using ir methods for labeling source code artifacts: Is it worthwhile? In *Program Comprehension (ICPC), 2012 IEEE 20th International Conference on*, pages 193–202, June 2012.
- [7] S. C. B. de Souza, N. Anquetil, and K. M. de Oliveira. A study of the documentation essential to software maintenance. In *Proceedings of the 23rd annual international conference on Design of communication: documenting & designing for pervasive information*, SIGDOC '05, pages 68–75, New York, NY, USA, 2005. ACM.
- [8] E. Duala-Ekoko and M. P. Robillard. Asking and answering questions about unfamiliar apis: an

- exploratory study. In *Proceedings of the 2012 International Conference on Software Engineering*, ICSE 2012, pages 266–276, Piscataway, NJ, USA, 2012. IEEE Press.
- [9] B. Eddy, J. Robinson, N. Kraft, and J. Carver. Evaluating source code summarization techniques: Replication and expansion. In *Proceedings of the 21st International Conference on Program Comprehension*, ICPC '13, 2013.
- [10] D. S. Eisenberg, J. Stylos, and B. A. Myers. Apatite: a new interface for exploring apis. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '10, pages 1331–1334, New York, NY, USA, 2010. ACM.
- [11] B. Fluri, M. Wursch, and H. C. Gall. Do code and comments co-evolve? on the relation between source code and comment changes. In *Proceedings of the 14th Working Conference on Reverse Engineering*, WCRE '07, pages 70–79, Washington, DC, USA, 2007. IEEE Computer Society.
- [12] A. Forward and T. C. Lethbridge. The relevance of software documentation, tools and technologies: a survey. In *Proceedings of the 2002 ACM symposium on Document engineering*, DocEng '02, pages 26–33, New York, NY, USA, 2002. ACM.
- [13] A. Gatt and E. Reiter. Simplenlg: a realisation engine for practical applications. In *Proceedings of the 12th European Workshop on Natural Language Generation*, ENLG '09, pages 90–93, Stroudsburg, PA, USA, 2009. Association for Computational Linguistics.
- [14] E. Goldberg, N. Driedger, and R. Kittredge. Using natural-language processing to produce weather forecasts. *IEEE Expert*, 9(2):45–53, April 1994.
- [15] 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 2010 17th Working Conference on Reverse Engineering*, WCRE '10, pages 35–44, Washington, DC, USA, 2010. IEEE Computer Society.
- [16] E. Hill. *Integrating Natural Language and Program Structure Information to Improve Software Search and Exploration*. PhD thesis, Newark, DE, USA, 2010. AAI3423409.
- [17] E. Hill, L. Pollock, and K. Vijay-Shanker. Automatically capturing source code context of nl-queries for software maintenance and reuse. In *Proceedings of the 31st International Conference on Software Engineering*, ICSE '09, pages 232–242, Washington, DC, USA, 2009. IEEE Computer Society.
- [18] R. Holmes and G. C. Murphy. Using structural context to recommend source code examples. In *Proceedings of the 27th international conference on Software engineering*, ICSE '05, pages 117–125, New York, NY, USA, 2005. ACM.
- [19] W. M. Ibrahim, N. Bettenburg, B. Adams, and A. E. Hassan. Controversy corner: On the relationship between comment update practices and software bugs. *J. Syst. Softw.*, 85(10):2293–2304, Oct. 2012.
- [20] K. Inoue, R. Yokomori, H. Fujiwara, T. Yamamoto, M. Matsushita, and S. Kusumoto. Component rank: relative significance rank for software component search. In *Proceedings of the 25th International Conference on Software Engineering*, ICSE '03, pages 14–24, Washington, DC, USA, 2003. IEEE Computer Society.
- [21] M. Kajko-Mattsson. A survey of documentation practice within corrective maintenance. *Empirical Softw. Engg.*, 10(1):31–55, Jan. 2005.
- [22] T. Karrer, J.-P. Krämer, J. Diehl, B. Hartmann, and J. Borchers. Stacksplorer: call graph navigation helps increasing code maintenance efficiency. In *Proceedings of the 24th annual ACM symposium on User interface software and technology*, UIST '11, pages 217–224, New York, NY, USA, 2011. ACM.
- [23] M. Kim, D. Notkin, D. Grossman, and G. Wilson. Identifying and summarizing systematic code changes via rule inference. *IEEE Transactions on Software Engineering*, 39(1):45–62, Jan. 2013.
- [24] A. J. Ko, B. A. Myers, and H. H. Aung. Six learning barriers in end-user programming systems. In *Proceedings of the 2004 IEEE Symposium on Visual Languages - Human Centric Computing*, VLHCC '04, pages 199–206, Washington, DC, USA, 2004. IEEE Computer Society.
- [25] D. Kramer. Api documentation from source code comments: a case study of javadoc. In *Proceedings of the 17th annual international conference on Computer documentation*, SIGDOC '99, pages 147–153, New York, NY, USA, 1999. ACM.
- [26] J. Krinke. Effects of context on program slicing. *J. Syst. Softw.*, 79(9):1249–1260, Sept. 2006.
- [27] A. N. Langville and C. D. Meyer. *Google's PageRank and Beyond: The Science of Search Engine Rankings*. Princeton University Press, Princeton, NJ, USA, 2006.
- [28] T. D. LaToza and B. A. Myers. Developers ask reachability questions. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1*, ICSE '10, pages 185–194, New York, NY, USA, 2010. ACM.
- [29] D. Lawrie, C. Morrell, H. Feild, and D. Binkley. What's in a name? a study of identifiers. In *14th International Conference on Program Comprehension*, pages 3–12. IEEE Computer Society, 2006.
- [30] T. C. Lethbridge, J. Singer, and A. Forward. How software engineers use documentation: The state of the practice. *IEEE Softw.*, 20(6):35–39, Nov. 2003.
- [31] S. Mani, R. Catherine, V. S. Sinha, and A. Dubey. Ausum: approach for unsupervised bug report summarization. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, FSE '12, pages 11:1–11:11, New York, NY, USA, 2012. ACM.
- [32] C. D. Manning, P. Raghavan, and H. Schtze. *Introduction to Information Retrieval*. Cambridge University Press, New York, NY, USA, 2008.
- [33] C. McMillan, M. Grechanik, D. Poshyanyk, Q. Xie, and C. Fu. Portfolio: finding relevant functions and their usage. In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, pages 111–120, New York, NY, USA, 2011. ACM.
- [34] L. Moreno, J. Aponte, S. Giriprasad, A. Marcus, L. Pollock, and K. Vijay-Shanker. Automatic generation of natural language summaries for java classes. In *Proceedings of the 21st International*

- Conference on Program Comprehension, ICPC '13*, 2013.
- [35] D. T. Morse. Minsize2: A computer program for determining effect size and minimum sample size for statistical significance for univariate, multivariate, and nonparametric tests. *Educational and Psychological Measurement*, 59(3):518–531, June 1999.
 - [36] G. C. Murphy. *Lightweight structural summarization as an aid to software evolution*. PhD thesis, University of Washington, July 1996.
 - [37] S. Panichella, J. Aponte, M. Di Penta, A. Marcus, and G. Canfora. Mining source code descriptions from developer communications. In *Program Comprehension (ICPC), 2012 IEEE 20th International Conference on*, pages 63–72, June 2012.
 - [38] D. Puppin and F. Silvestri. The social network of java classes. In *Proceedings of the 2006 ACM symposium on Applied computing, SAC '06*, pages 1409–1413, New York, NY, USA, 2006. ACM.
 - [39] E. Reiter and R. Dale. *Building natural language generation systems*. Cambridge University Press, New York, NY, USA, 2000.
 - [40] T. Roehm, R. Tiarks, R. Koschke, and W. Maalej. How do professional developers comprehend software? In *Proceedings of the 2012 International Conference on Software Engineering, ICSE 2012*, pages 255–265, Piscataway, NJ, USA, 2012. IEEE Press.
 - [41] L. Shi, H. Zhong, T. Xie, and M. Li. An empirical study on evolution of api documentation. In *Proceedings of the 14th international conference on Fundamental approaches to software engineering: part of the joint European conferences on theory and practice of software, FASE'11/ETAPS'11*, pages 416–431, Berlin, Heidelberg, 2011. Springer-Verlag.
 - [42] J. Sillito, G. C. Murphy, and K. De Volder. Asking and answering questions during a programming change task. *IEEE Trans. Softw. Eng.*, 34(4):434–451, July 2008.
 - [43] S. E. Sim, C. L. A. Clarke, and R. C. Holt. Archetypal source code searches: A survey of software developers and maintainers. In *Proceedings of the 6th International Workshop on Program Comprehension, IWPC '98*, pages 180–, Washington, DC, USA, 1998. IEEE Computer Society.
 - [44] M. D. Smucker, J. Allan, and B. Carterette. A comparison of statistical significance tests for information retrieval evaluation. In *CIKM*, pages 623–632, 2007.
 - [45] G. Sridhara. *Automatic Generation of Descriptive Summary Comments for Methods in Object-oriented Programs*. PhD thesis, University of Delaware, Jan. 2012.
 - [46] G. Sridhara, E. Hill, D. Muppaneni, L. Pollock, and K. Vijay-Shanker. Towards automatically generating summary comments for java methods. In *Proceedings of the IEEE/ACM international conference on Automated software engineering, ASE '10*, pages 43–52, New York, NY, USA, 2010. ACM.
 - [47] G. Sridhara, L. Pollock, and K. Vijay-Shanker. Automatically detecting and describing high level actions within methods. In *Proceedings of the 33rd International Conference on Software Engineering, ICSE '11*, pages 101–110, New York, NY, USA, 2011. ACM.
 - [48] G. Sridhara, L. Pollock, and K. Vijay-Shanker. Generating parameter comments and integrating with method summaries. In *Proceedings of the 2011 IEEE 19th International Conference on Program Comprehension, ICPC '11*, pages 71–80, Washington, DC, USA, 2011. IEEE Computer Society.
 - [49] D. Steidl, B. Hummel, and E. Juergens. Quality analysis of source code comments. In *Proceedings of the 21st International Conference on Program Comprehension, ICPC '13*, 2013.
 - [50] J. Stylos and B. A. Myers. Mica: A web-search tool for finding api components and examples. In *Proceedings of the Visual Languages and Human-Centric Computing, VLHCC '06*, pages 195–202, Washington, DC, USA, 2006. IEEE Computer Society.
 - [51] J. Stylos, B. A. Myers, and Z. Yang. Jadeite: improving api documentation using usage information. In *CHI '09 Extended Abstracts on Human Factors in Computing Systems, CHI EA '09*, pages 4429–4434, New York, NY, USA, 2009. ACM.
 - [52] A. A. Takang, P. A. Grubb, and R. D. Macredie. The Effects of Comments and Identifier Names on Program Comprehensibility: An Experimental Study. *Journal of Programming Languages*, 4(3):143–167, 1996.
 - [53] Y. Tao, Y. Dang, T. Xie, D. Zhang, and S. Kim. How do software engineers understand code changes?: an exploratory study in industry. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering, FSE '12*, pages 51:1–51:11, New York, NY, USA, 2012. ACM.
 - [54] D. van Heesch. Doxygen website, 2013.
 - [55] A. T. T. Ying and M. P. Robillard. Code fragment summarization. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2013*, pages 655–658, New York, NY, USA, 2013. ACM.
 - [56] S. Zhang, C. Zhang, and M. D. Ernst. Automated documentation inference to explain failed tests. In *Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering, ASE '11*, pages 63–72, Washington, DC, USA, 2011. IEEE Computer Society.