A Human Study of Comprehension and Code Summarization

Sean Stapleton
University of Michigan
seancs@umich.edu

Yashmeet Gambhir
University of Michigan
ygambhir@umich.edu

Alexander LeClair
University of Notre Dame
aleclair@nd.edu

Zachary Eberhart
University of Notre Dame
zeberhar@nd.edu

Westley Weimer
University of Michigan
weimerw@umich.edu

Kevin Leach
University of Michigan
kjleach@umich.edu

Yu Huang
University of Michigan
yhhy@umich.edu

ABSTRACT

Software developers spend a great deal of time reading and understanding code that is poorly-documented, written by other developers, or developed using differing styles. During the past decade, researchers have investigated techniques for automatically documenting code to improve comprehensibility. In particular, recent advances in deep learning have led to sophisticated summary generation techniques that convert functions or methods to simple English strings that succinctly describe that code's behavior. However, automatic summarization techniques are assessed using internal metrics such as BLEU scores, which measure natural language properties in translational models, or ROUGE scores, which measure overlap with human-written text. Unfortunately, these metrics do not necessarily capture how machine-generated code summaries actually affect human comprehension or developer productivity.

We conducted a human study involving both university students and professional developers ($n = 45$). Participants reviewed Java methods and summaries and answered established program comprehension questions. In addition, participants completed coding tasks given summaries as specifications. Critically, the experiment controlled the source of the summaries: for a given method, some participants were shown human-written text and some were shown machine-generated text.

We found that participants performed significantly better ($p = 0.029$) using human-written summaries versus machine-generated summaries. However, we found no evidence to support that participants perceive human- and machine-generated summaries to have different qualities. In addition, participants' performance showed no correlation with the BLEU and ROUGE scores often used to assess the quality of machine-generated summaries. These results suggest a need for revised metrics to assess and guide automatic summarization techniques.

ACM Reference Format:
Sean Stapleton, Yashmeet Gambhir, Alexander LeClair, Zachary Eberhart, Westley Weimer, Kevin Leach, and Yu Huang. 2020. A Human Study of Comprehension and Code Summarization. In *28th International Conference on Program Comprehension (ICPC '20), October 5–6, 2020, Seoul, Republic of Korea.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3387904.3389258

ICPC '20, October 5–6, 2020, Seoul, Republic of Korea

ACM ISBN 978-1-4503-7958-8/20/05…$15.00
https://doi.org/10.1145/3387904.3389258

1 INTRODUCTION

Source code comments play an invaluable role in facilitating program comprehension [6]. Short, descriptive summary comments preceding subroutines have been shown to significantly improve programmers' ability to answer questions about source code [66, 70]. Recent work has shown that developers consider comments to be the most important documentation artifacts for software maintenance tasks, other than the source code itself [14, 55]. Well-documented source code manifestly affects developer productivity both when investigating an existing software project for the first time and when maintaining existing large codebases [12].

Despite their well-known importance, comments are often incomplete or incorrect in practice [18], both in general, and for specific aspects (e.g., exceptions [9, Sec. 5] or commit messages [10, Sec. 3]). These missing or outdated comments can substantially impair the development process. In a large-scale study of professional programmers, Xia *et al.* [71] found that insufficiently-commented code was the most frequent cause of program comprehension difficulties. One solution is to enforce strict style guidelines that require up-to-date comments—however, these guidelines can be costly and difficult to enforce [32].

To address these problems, researchers have proposed numerous techniques to automatically generate summary comments for source code [47]. These techniques traditionally rely on elaborate heuristics and templates to generate comments that resemble natural language [44, 63]. However, designing these methods can entail substantial human effort and implicit assumptions about the ideal structure of comments (e.g., some Java documentation systems [24, 42, 57] rely on a verb-noun style imposed by the developer). In recent years, a new generation of code summarization techniques have emerged that take advantage of deep learning and large, publicly-available code repositories [2, 30, 34, 37, 45]. These neural-network-based approaches have demonstrated tremendous promise, as they are capable of producing summaries that are nearly indistinguishable from human-written comments.

Still, even state-of-the-art neural approaches to code summarization have room for improvement. Consider the examples shown in Figure 1. Each example contains a reference comment written by a programmer for a particular method paired with a summary generated using the neural model published by LeClair *et al.* [34]. The automatically-generated summaries, while potentially helpful, do not necessarily express what the function does or what its intended purpose is in the same way that the human-written summaries do.

Human Summary: begin UML doc
Machine Summary: copy the contents of this entry into another

```
public void copy(Entry otherEntry) {
 if (otherEntry == null)
   return;

 super.copy((NiCEObject) otherEntry);

 this.allowedValues = otherEntry.allowedValues;
 this.allowedValueType = otherEntry.allowedValueType;
 this.changeState = otherEntry.changeState;
 this.defaultValue = otherEntry.defaultValue;
 this.ready = otherEntry.ready;
 this.value = otherEntry.value;
 this.secretFlag = otherEntry.secretFlag;
 this.parent = otherEntry.parent;
 this.tag = otherEntry.tag;
}
```

(a) Example snippet whose summary has a low BLEU score.

Human Summary: sorts the specified range of the receiver into ascending numerical order
Machine Summary: sorts the receiver according to the order of the order by the

```
public void quickSortFromTo(int from, int to) {
 int mySize = size();
 checkRangeFromTo(from, to, mySize);

 short[] myElements = elements();
 java.util.Arrays.sort(myElements, from, to+1);
 elements(myElements);
 setSizeRaw(mySize);
}
```

(b) Example snippet whose summary has a moderately high BLEU score.

Figure 1: Example code snippets, each shown with a Human-written and Machine-generated summary using state-of-the-art neural summarization. The intended purpose of the code is not necessarily reflected in the machine-generated summaries or the associated BLEU scores.

The synthesis of text is often guided by common statistical evaluation metrics such as BLEU [51], ROUGE [38], and METEOR [4]. These metrics compare generated text to reference text without considering semantic meaning. In neural machine translation, these metrics have been shown to correlate reasonably well with human judgment [13], but such correlation has not been sufficiently established in the domain of automatic code summarization. For this reason, many proposed code summarization techniques are accompanied by a human study, in which programmers are asked to judge the quality of generated comments [23, 46, 62].

These types of "intrinsic" evaluation allow for comparison between different summarization techniques, but they fail to explicitly capture one key quality: the extent to which automatically generated comments actually improve program comprehension. It is well known that high-quality comments help to facilitate program comprehension, but misleading comments may actually produce the opposite effect. According to one subject in a study by Ibrahim *et al.* [29], "Wrong comments are worse than none at all."

We propose to evaluate the "extrinsic" quality of machine-generated summaries (i.e., their value to programmers during comprehension) by having programmers engage in code comprehension tasks aided by either machine-generated or human-generated summaries. By comparing the performance of programmers that are given machine-generated comments to those given human-generated comments, we can measure and determine, with high confidence, the degree to which automatic comment generation techniques can serve as a functional replacement for manual comment generation.

To our knowledge, there are no human studies of automatic comment generation techniques that include a thorough extrinsic evaluation of the impact on program comprehension. There are several possible reasons for the omission, including the high cost of hiring qualified programmers to participate, the difficulty of creating and evaluating tasks that adequately effectuate real program comprehension processes, and the fact that intrinsic evaluations are typically considered to be sufficient for publication.

In this paper, we aim to fill this gap by conducting an extrinsic study to evaluate the extent to which comments generated by a state-of-the-art deep learning technique facilitate program comprehension. We recruited 45 student and professional participants to complete a set of program comprehension tasks, aided by either human-written or machine-generated code summaries. These tasks involved answering questions about individual methods or adding new functionality to a program.

We found that human-written summaries help developers comprehend code significantly better ($p = 0.029$) than machine-generated summaries. Moreover, developer assessment of summary quality did not correlate with their comprehension, regardless of whether they were provided human-written or machine-generated summaries. Finally, we found that BLEU and ROUGE scores of machine-generated summaries did not strongly correlate with developer comprehension. Overall, our results indicate there is a need to reassess underlying assumptions about the use of internal statistics to assess the quality of automatic summarization techniques.

2 MOTIVATING EXAMPLES

There is a pressing need for extrinsic studies that show how automatic summarization techniques impact developer comprehension. In this section, we explore the two examples shown in Figure 1 as a basis for establishing potential issues with relying on metrics like BLEU and ROUGE for automatic summarization evaluation.

The first example in Subfigure 1a contains a human-written comment that refers to "UML." However, the code snippet itself does not mention UML anywhere—the human who wrote the comment knew about the context in which the method would be used or its intended purpose. Indeed, the human-written summary on its own

does not reflect what the method accomplishes in isolation. On the other hand, the machine-generated summary succinctly describes what behavior the method implements—a copy of an Entry object. While the machine-generated comment describes the method's behavior in isolation, its BLEU score is 0 because it does not overlap with the human comment. This low score does not necessarily match one's intuition for summary quality—a new developer may find the machine-generated summary more illuminating than the human-written summary in this example.

The second example in Subfigure 1b contains human-written and machine-generated summaries that both indicate a sorting operation will occur. However, the machine-generated comment is potentially misleading in that it does not communicate the type of data being sorted nor the final ordering. However, the BLEU score is moderately high (0.06, among top 50% of BLEU scores for our dataset of 2 million summaries). In this case, a new developer may not find the automatically-generated summary as useful. These examples illustrate potential limitations with the prevailing approach to using internal scoring like BLEU to assess summarization techniques. In this paper, we investigate the impact of such automatic summarization techniques on developer comprehension.

3 BACKGROUND AND RELATED WORK

We discuss two key research areas relevant to our problem: empirical studies of program comprehension, and automatic source code summarization. We introduce work in these areas, and discuss current evaluation methodologies for each.

3.1 Studies of Program Comprehension

Empirical studies of program comprehension can typically be characterized as either understanding and modeling the underlying cognitive processes, or evaluating the impact of various factors on comprehension. These two subdisciplines work hand in hand: theories of program comprehension can help guide researchers to develop tools to meet specific developer needs [65].

3.1.1 Comprehension Models. Researchers typically acknowledge broad categories of program comprehension models: top-down models, bottom-up models, and integrated models [59, 60, 67]. The process of top-down comprehension begins with a programmer making a general hypothesis about a program's purpose, and iteratively refining the hypothesis by developing subsidiary hypotheses, until a low-level understanding of the code is achieved. Bottom-up comprehension works the other way, with a programmer initially parsing low-level statements, and then iteratively grouping them together into higher-level abstractions until a hypothesis about the program's purpose can be developed. Integrated models combine both prior models, allowing for a programmer to use top-down processes when they are able to form an initial hypothesis and bottom-up processes when they cannot.

Bottom-up comprehension is a slower, more laborious task than top-down comprehension [60, 65]. To help facilitate top-down comprehension, programmers will often search for *beacons* — sets of features in source code that are familiar to the programmer, which are indicative particular structures or operations [7]. Source code comments can play a beacon-like role by explicitly describing the purpose, usage, or functionality of a code segment. As a result, the presence of comments has been shown to expedite and improve program comprehension [6, 66, 70].

3.1.2 Evaluation. With an understanding of the underlying mental processes, researchers have evaluated the impact of a plethora of different factors on program comprehension [56]. For instance, Ceccatto *et al.* [11] investigated the extent to which code obfuscation inhibits code comprehension, in the context of preventing reverse engineering attacks. They found that obfuscation makes code more difficult to comprehend, supporting its use as a security mechanism. Prechelt *et al.* [52] performed a study to determine whether in-line design pattern documentation helped to improve program comprehension. They found the comments containing pattern information helped programmers perform maintenance tasks faster and more accurately. Bauer *et al.* [5] investigated the role of indentation style on comprehension, while Hofmeister *et al.* [25] found that shorter identifiers take longer to comprehend. Researchers have proposed and evaluated tools to improve program comprehension, such as JRipples [8], DynaRia [3], Jsea [68], and many others.

In each of these studies, researchers had to determine an appropriate methodology to measure the impact of the factor or tool on the programmers' comprehension. Ceccatto *et al.* [11] asked participants to perform a series of code change tasks. They recorded the participants' times and success rates, as well as participants answers on a questionnaire rating perceived task difficulty, usefulness of various debugging features, etc. Lawrie *et al.* [33] performed a study in which they showed participants code snippets, and later asked them to recall and describe the snippets they had seen. Dunsmore *et al.* [15] compared and contrasted four broad measures of program comprehension: maintenance tasks, mental simulation, static questions, and subjective ratings. Each study must use an evaluation that is carefully designed to measure particular, relevant aspects of program comprehension.

While there is no single way to measure "program comprehension," researchers have identified key activities involved in comprehension and designed several questions and tasks to simulate those activities. Sillito *et al.* [61] performed empirical studies to determine the most common types of questions programmers ask during software evolution tasks. They observed 44 different question types, which they grouped into 4 categories: questions designed to find focal points, build on those points, understand subgraphs, and understand relationships between subgraphs. Pacione *et al.* [50] devised a set of 9 principle comprehension activities by reviewing tasks used in other comprehension evaluation literature. Several researchers have relied on the comprehension activities identified by Sillito and Pacione to develop appropriate comprehension tasks for their evaluations [1, 17, 20, 31, 69].

3.2 Automatic Source Code Summarization

Automatic Source Code Summarization is the task of generating a brief, natural language summary description of some unit of source code. Much of the research in this area is concentrated on summarizing individual subroutines [23, 30, 34, 39, 43], though researchers have also worked on class-level [41] and segment-level [19] summarization. Research in automatic source code summarization can be broadly categorized according to (1) the techniques used to generate summaries, and (2) the methods used to evaluate those techniques.

3.2.1 Summarization. Traditional approaches to automatic source code summarization consist of two key subtasks: selecting keywords, and assembling those keywords into a natural-language summary. A large body of work is devoted to the first subtask. Haiduc *et al.* [23] used Latent Semantic Indexing to generate a set of terms that were conceptually related to a Java method, demonstrating that existing summarization techniques may be applicable to the programming domain. Similarly, Rodeghero *et al.* [54] identified key terms in Java methods using an approach that combined term frequency-inverse document frequency (*tf-idf*) with empirical data about the locations in a method's structure that programmers were most likely to look for keywords. More recent approaches, such as those by Nazar *et al.* [49] and Allamanis *et al.* [2], have used machine learning techniques to identify keywords.

Other work has tackled both summarization subtasks by finding key terms within a piece of source code, and then placing those terms into a pre-defined template. Sridhara *et al.* [62] modeled each summary as a verb phrase, consisting of an *action*, a *theme*, and *secondary-arguments*. They used a custom language model and hand-crafted heuristics to select and arrange key terms, and used additional templates to account for nested methods and conditional statements. McBurney *et al.* [43] took a similar approach, but used more robust templates to ensure that the summary included information about the method's functionality, usage, and output.

Recent advancements in deep learning have given rise to a new generation of data-driven summarization techniques that use end-to-end neural networks to generate summaries, without using separate processes to select terms and to arrange them into sentences. These techniques are largely inspired by those used in the field of neural machine translation. They typically involve using an *encoder-decoder* model to generate vector representations of source code, from which summary descriptions are inferred. These techniques rely on big data in the form of massive source code repositories in order to train the neural models. Initial results from a neural approach used by Iyer *et al.* [30] proved remarkably effective at generating short summaries for Java methods. Loyola *et al.* [40] demonstrated the efficacy of a similar approach on commit message generation. Liang *et al.* [37] and Hu *et al.* [27] achieved even better results by incorporating structural information about the code in the form of an Abstract Syntax Tree (AST). Subsequent approaches have demonstrated additional improvements [39, 45, 58].

3.2.2 Evaluation. The methods used to evaluate summarization techniques can be classified as *automatic statistical* evaluations, *intrinsic human* evaluations, or *extrinsic human* evaluations.

Automatic statistical evaluation generally involves comparing a generated summary to a reference or baseline summary [48]. Approaches that use existing source code repositories to train their summarization models often set aside a portion of the dataset as a "test set" for evaluation. During testing, the automatically-generated comments for each subroutine or class are compared to the original, human-written comments for the same subroutine or class in the test set. The summaries are often scored using Machine Translation metrics like BLEU [51] and METEOR [4]. When baseline summaries are not readily available, researchers may first perform a human study in which programmers are asked to generate summaries to serve as baselines [22, 54].

As noted in Section 1, the application of these automatic metrics to code summaries can be problematic for a variety of reasons. First, as Reiter [53] points out, BLEU was designed as a diagnostic metric to provide quick, approximate evaluation — its actual correlation with human judgment can vary greatly. Second, BLEU was designed specifically to evaluate machine translation systems [53]. Although automatic summarization is often framed in terms of machine translation, they are fundamentally different problems as summarization inherently reduces information. Third, BLEU (and similar metrics) are heavily dependent on the context and quality of the reference strings, which are often inconsistent across different evaluations [35]. Fourth, machine translation metrics frequently depend on syntactic properties of strings rather than nuanced semantics. For example, BLEU is a composite score based upon n-gram overlaps between two strings that does not account for style or quality of information. Finally, these metrics can be hard to interpret. As we note in the example in Subfigure 1a, the automatically-generated summary has a BLEU score of 0 even though it is a readable, useful summary. These limitations imply that machine translation metrics may not always be appropriate for evaluating summarization.

Intrinsic human evaluations are ones in which human participants judge the output of a summarization technique. They generally involve human participants rating various aspects of the content or naturalness of the summaries [72]. These evaluations overcome some of the issues of automatic statistical evaluations, but have other weaknesses. They are considered to be more robust, as humans are more capable of interpreting the syntax and semantics of summaries than metrics like BLEU. However, they introduce other threats to validity — human ratings are subjective, and different raters may have different backgrounds or levels of experience that influence their scores. When analyzing human ratings, it is important to report some metric of inter-rater reliability [26].

Extrinsic human evaluations of automatic code summarization techniques, in which summaries are indirectly analyzed in the context of some programming task, are rare. McBurney *et al.* [44] and Sridhara *et al.* [64] performed extrinsic evaluations by showing human participants code snippets accompanied by automatically-generated summaries, and asking them to rate how helpful they believed the summaries to be to their understanding. Note that both evaluations asked programmers to rate how useful they *believed* the summaries to be — neither measured the objective impact of the summaries on the programmers' ability to comprehend code. In the following sections, we describe the methodology we use to measure that impact.

4 SUMMARY COMMENT GENERATION

In this section, we discuss the models, methodology, and data we used to generate machine written comments. For our model we use the encoder-decoder+AST model proposed by LeClair *et al.* [34] which they made available in an online repository. For our data set we used the Java method-comment parallel corpus provided by LeClair *et al.* [35]. We chose to use this particular model architecture for three reasons: (1) it achieved state of the art results against multiple baselines, (2) the source code, training scripts, and sample models are available online and the models were easily reproduced, and (3) the model is based on a standard architecture that has seen wide use in the literature for translation and summarization tasks.

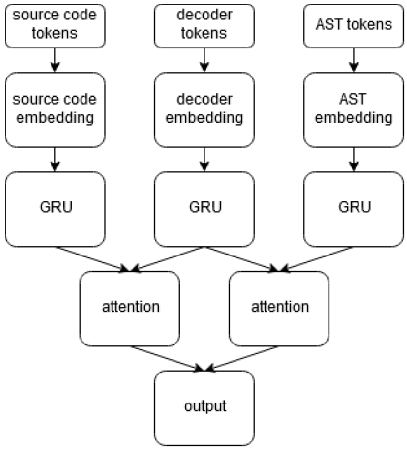

Figure 2: LeClair *et al.* [34] Model Architecture.

Note that we claim no novelty in the model or algorithm used to generate summary comments. While this section briefly describes that model architecture for completeness, the focus of this paper is on the extrinsic evaluation of those comments in the context of realistic code comprehension tasks (Section 5).

4.1 Model Overview

The model proposed by LeClair *et al.* is based on an encoder-decoder architecture with an added encoder for the AST [34]. Their work uses the Structure Based Traversal (SBT) method proposed by Hu *et al.* [27] to create a flattened AST sequence. This gives the model three inputs: (1) the source code token sequence, (2) the flattened AST token sequence, and (3) the predicted sequence *so far*. A high-level diagram of the model architecture is shown in Figure 2.

Then, the model has two encoders: (1) source code token encoder and (2) flattened AST encoder. These encoders use an embedding layer and recurrent layer to generate a vector representation of the input sequence. The recurrent layers process the input sequences over time, with each token of the input sequence representing a time-step. For example, given the sequence [‘the’, ‘cat’, ‘sat’], the recurrent layer would process the sequence as:

```
['the',0,0] -> ['the','cat',0] -> ['the','cat','sat']
```

creating a vector representation of the input at each time-step, and then using that vector as the initial state for the next time step. The decoder, similarly to the encoders, utilizes an embedding layer and recurrent layer to generate a vector representation for the current predicted summary.

Next, there are two attention mechanisms to combine the output of the recurrent layers from the encoders and decoder: (1) attention between the source code and summary, and (2) attention between the AST and summary. The attention mechanisms learn which tokens are most important to the prediction of the next token. The attention mechanisms are combined to create an overall representation of the context of the input sequences. Next, a dense layer is applied and flattened to create a single vector representation which is then used to predict the next token in the sequence.

Additional model-specific details are found in LeClair et al. [34]; in summary, we note that this is a representative, high-performance approach using a now-standard model architecture.

Human Summary: frees all resources consumed by this components and destroys it
Machine Summary: p release all resources allocated to the application

```
public void destroy() {
    if (properties != null) {
        properties.clear();
        properties = null;
    }
    if (supportedLocales != null) {
        supportedLocales.clear();
        supportedLocales = null;
    }
}
```

Figure 3: Example code and corresponding summaries.

4.2 Model Example

To further illustrate the model’s operation, we present an example showing how the source code and the AST contribute to producing a summary of code shown in Figure 3. In this example, the human-written and machine-generated summaries do not match perfectly (yielding a BLEU-1 score of 0.15).

Since the token ‘resources’ does not appear in the source code text, the model cannot rely on the source code sequence to provide the next token. Instead, it uses the structure of the code to help it determine which token should be predicted. In this example, the model has focused on the first portion of the AST sequence; in the training set, the token ‘resources’ is used frequently as a function name or in parameter lists. This allows the generated summary to (correctly) contain ‘resources’ even though the local source text does not. In brief, summaries generated by this technique focus on structural and syntactic properties in individual methods and throughout a training set, and evaluated with respect to metrics such as BLEU and ROUGE. Our experiments explore the degree to which these metrics reflect developer comprehension of code.

4.3 Dataset Overview

In our evaluation, we use a published dataset made available by LeClair *et al.* [35]. Their work on effective methods for creating a dataset for source code summarization outlined common mistakes that are made in data handling for this task. Their dataset contains 2.1 million Java methods and associated JavaDoc comments in two forms: (1) *filtered*: where they have already excluded methods for which no comment exists or those which appear in automatically-generated files but keep the the comment and source code unchanged from the original Java file, and (2) *tokenized*: where text preprocessing steps have already been applied to the code and comment. For this paper, we use the provided tokenized data set unchanged to train, validate, and test the model, achieving similar results to those reported in LeClair *et al.* [34].

5 EVALUATION METHODOLOGY

In this section, we describe our methodology for measuring the impact of code summaries on developer productivity. We designed an

IRB-approved human study involving 45 undergraduate and graduate computer science students and industrial software developers. Participants were asked to complete two tasks in an anonymous online survey. First, they were shown methods written in Java alongside corresponding summaries and asked to answer comprehension questions. Second, participants were given partially-completed Java classes including summaries for all the methods in these classes and asked to complete a method in the class with respect to a held-out test suite. By measuring time taken and answer accuracy, we can develop a model of developer comprehension as a function of type and quality of the code summary used.

5.1 Participant Selection

In this study, we recruited combination of undergraduate computer science students, graduate computer science students, and professional industrial developers with less than five years experience. Participants were eligible if they had completed a data structures and algorithms course at the undergraduate level (i.e., second-year CS students and above). We drew students from three undergraduate courses and a graduate student mailing list at the University of Michigan, as well as industrial developers from a local startup company who had fewer than five years of professional experience. This participant cohort enables the examination of newer developers who we suspect are more sensitive to changes in documentation than more seasoned developers.

Participants were given a URL to visit to complete the study, which was administered as an online survey (described in further detail in Section 5.5). By controlling whether documentation was human-written or machine-generated, we can measure performance differences among sub-populations of participants who received one type of summary compared to another. Participant data was anonymized, but they could optionally leave contact information to obtain a $20 USD cash reward for their participation. All told, participation in the survey took between 45 and 75 minutes to complete in one sitting.

5.2 Code Comprehension Task

Each participant in this study was presented with 10 randomly-chosen Java snippets from a set of 50, filtered from the dataset created by LeClair *et al.* [35]. To filter the 2.1 million snippets in this dataset, we first limited the population of snippets to only those that had 6 or more immediate children in the body of the code's Abstract Syntax Tree (AST).

We found that this heuristic significantly increased the quality of the code over more rudimentary approaches such as filtering by line length. We chose our final set of 50 methods by uniformly sampling from this sub-population. These methods contained between 1 and 5 control flow statements, between 0 and 17 method invocations, and between 0 and 3 parameters.

For each participant and method, we randomly presented a human-written or machine-generated summary for that method. We visited the AST of each method to obtain a list of invoked methods for which we presented summaries as well. The participant was shown one entire method at a time, its summary, and a list of methods invoked (statically) with their corresponding summaries. Participants were shown a single question at a time.

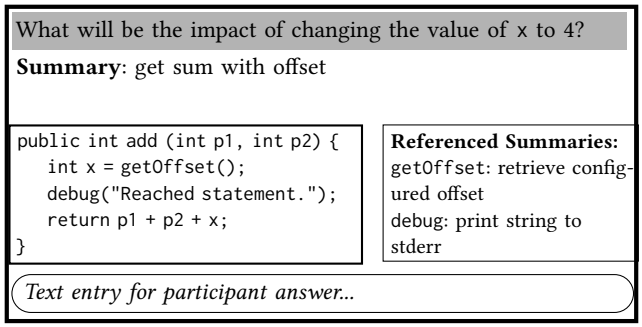

Figure 4: Example comprehension stimulus shown in our study. We adapted questions Q1–Q8 to target each specific method shown to participants. We also provided summaries for child methods invoked by the method shown to provide the participant with additional context.

For each Java method shown, participants were asked to complete three randomly-assigned, open-ended questions intended to assess their ability to comprehend the code. These questions cover three broad categories of program comprehension described in Table 1. We employed modifications of eight questions presented in Sillito *et al.* [61]. Specifically, we considered the following questions:

Q1 What data can we access from this object?
Q2 What are the arguments to this function?
Q3 What data is being modified in this code?
Q4 What is the correct way to use or access this data structure?
Q5 Under what circumstances is this function called?
Q6 How are these types or objects related?
Q7 How can we tell that this function has executed correctly?
Q8 What will be (or has been) the direct impact of this change?

These questions were chosen because they relate to the comprehension of single methods (i.e., we did not consider questions that relate to entire classes or software ecosystems). For each method, we further adapted the questions above to relate to the specifics of the method and randomly chose 3 of these adapted questions to present to the user. An example of such a question is shown in Figure 4.

Finally, participants subjectively rated the quality of the code summaries provided for each method on a scale of 1 (lowest quality) to 5 (highest quality).

5.3 Code Writing Task

Participants also completed a high-level code writing task: implementing a new class method based upon summaries of other methods in a class.

For each task, participants were shown a semi-complete Java data structure class taken from a entry-level course: a Binary Tree, a Singly-Linked List, or an Adjacency Matrix. We selected these data structures because they did not require domain-specific expertise.

In this code implementation task, a data structure was randomly chosen and assigned either human-written or machine-generated summaries (for all methods in that class). We removed one method from the class at random and tasked participants with implementing the held-out method given a method description as well as the other methods and summaries present in the class.

Table 1: Taxonomy of program comprehension questions in our study.

Comprehension Activity	Description	Questions Used
Static and Dynamic Comprehension	Must reason about syntactic properties (e.g., variable names) and runtime properties (e.g., variable values)	Q1, Q2, Q3
Context-Dependent Interpretation	Must reason about situations in which a method could be used by understanding context or assessing high level behavior	Q4, Q5, Q6
Reverse-engineering	Must reason about changes made to a snippet or test cases to elicit particular program behavior	Q7, Q8

We implemented a web-based IDE akin to LeetCode [36] and HackerRank [21] that allowed users to submit code and automatically evaluated it against a held-out test suite. We developed held-out test suites of 6–8 test cases covering all methods in each class, accounting for exceptions and edge cases. Participants were allowed to submit answers up to 15 times, and we recorded the subset of test cases passed with each submission.

5.4 Answer Annotation and Grading

At a high level, we collected two types of data: (1) Comprehension, in which participants read code and summaries and answered several comprehension questions, and (2) Implementation, in which participants were given an incomplete Java class and asked to write a missing method based on a held-out test suite. We describe our methodology for quantitatively analyzing this raw participant data.

Comprehension. Because participants are given a free space to write an open answer to each comprehension question, we developed a rubric for each snippet-question pair to provide a robust quantitative assessment of participant data. We ranked answers on a 1 to 5 scale, where 1 indicated a low quality response (e.g., complete misunderstanding of the method shown), and 5 indicated a high quality response (e.g., the participant correctly identified high-level function behavior). We refer to this scale as rater-assessed Correctness. Four raters graded all responses with respect to these snippet-question rubrics. Additionally, all responses were rated "Yes" or "No" for Completeness (i.e., is the answer complete, or did the participant skip the question?) and Relevance (i.e., is the answer in scope, or did the participant provide garbage input?). We show examples of answers marked as Relevant, Complete, and with different levels of rater-assessed Correctness in Figure 5.

We achieved moderate inter-rater agreement [16] for the 5-point Correctness rubrics ($\kappa = 0.655$), and high agreement for the Completeness ($\kappa = 0.947$) and Relevance ($\kappa = 0.993$) rubrics. The results and interpretation of participant answers are discussed in Section 6.

Implementation. Participants wrote and submitted code online that automatically evaluated their submission against a held-out test suite. We used the fraction of passed test cases as a proxy for correctness in conjunction with the number of submissions required to attain their highest score.

Data Filtering. For the comprehension questions, we only considered data marked as Complete and Relevant by all graders. Additionally, because we used a web survey, we excluded any answers that took longer than 30 minutes to complete—participants who left the web survey, closed the browser, or otherwise stopped participating. After filtering our data as described above, we retained a total 1,100 comprehension question data points. Among these, 564 were answered with machine-generated comments, and 536 were answered with human-written comments.

5.5 Survey Instrument

To deliver our code comprehension questionnaire and code implementation test to participants, we created a lightweight web application allowing them to complete the survey remotely at their convenience. This permitted anonymous completion of the survey. We required desktop versions of Chrome or Firefox to complete the survey (i.e., no mobile devices).

For each of the snippets shown in the comprehension section of the survey, participants could view the snippet in a syntax-highlighted box, along with any relevant code summaries, which appear alongside the code in another column.

We developed an in-browser IDE for participants to complete the code writing task while easily referring to the rest of the data structure class and relevant summaries. Methods in each class contained either entirely human- or machine-generated comments, placed to help participants understand and use other methods in the class. Test cases for each class were run in a containerized Flask application to prevent malicious inputs and to set equitable resource consumption limits among participant submissions. We make our survey instrument available for reuse in further human studies.[1]

6 EXPERIMENTAL RESULTS

We seek to understand how developer productivity is influenced by the presence of and perception of human-written and machine-generated summaries of code. We evaluate several research questions based on the data we collected:

RQ1 Do developers better comprehend code snippets when given human-written versus machine-generated summaries?

RQ2 Do developers interpret human-written as being higher quality than machine-generated summaries?

RQ3 Do internal measurements of machine-generated summaries such as BLEU and ROUGE scores correlate with code comprehension?

We present quantitative results and interpretations for each of these research questions. We also provide a qualitative discussion of the data collected from the code writing task.

[1] https://dijkstra.eecs.umich.edu/code-summary/

Human Summary: calculates the crc and size of the resource and updates the jar entry

Machine Summary: calculate the checksum for a jar entry

```
private void calculateCrcAndSize(JarEntry jarEntry,
    IFile resource, byte[] readBuffer)
 throws IOException, CoreException {
  InputStream contentStream =
      resource.getContents(false);
  int size = 0;
  CRC32 checksumCalculator= new CRC32();
  int count;
  try {
    while ((count= contentStream.read(readBuffer, 0,
        readBuffer.length)) != -1) {
      checksumCalculator.update(readBuffer, 0, count);
      size += count;
    }
  } finally {
    if (contentStream != null)
      contentStream.close();
  }
  jarEntry.setSize(size);
  jarEntry.setCrc(checksumCalculator.getValue());
}
```

(a) Example snippet.

Q: What would be the effect of changing "readBuffer.length" to "readBuffer.length/2"?

A: The function would still eventually read the entire contentStream out, but it would take twice as many iterations. And I believe that the checksumCalculator. update() function would then be using the same values for the latter half of the readBuffer every iteration (potentially uninitialized values too), which could introduce undefined behavior.

(b) Example response with high rater-assessed Correctness.

Q: What is the relationship between "size" and "checksumCalculator.getValue()"?

A: equal

(c) Example response with low rater-assessed Correctness.

sdahfjkafea

(d) Example response that is considered not Relevant.

No clue.

(e) Example response that is considered not Complete.

Figure 5: Example code snippet, summaries, comprehension questions, and corresponding answers collected in our study. We highlight examples that were graded as highly Correct (Subfigure 5b), not Correct (Subfigure 5c), not Relevant (Subfigure 5d), and not Complete (Subfigure 5e). The answer marked as Correct contains a detailed assessment of the code and effects the question introduces. The not-Correct answer is Relevant, but missed a key step understanding checksums. The other answers are removed from consideration because the participant did not provide a coherent answer or any understanding of the snippet.

6.1 RQ1: Code Comprehension

We seek to understand the impact had on developer code comprehension when given machine-generated summaries versus human-written summaries. Participants were shown ten snippets of code, each randomly paired with machine or human summaries. For each snippet, they were asked three randomly-selected code comprehension questions (see Q1–Q8 in Section 5.2), and were asked to rate from 1 to 5 their perception of the summary's helpfulness in answering the questions. We then graded all responses in terms of Correctness, Completeness, and Relevance (see Section 5.4).

We divide our raw data into two groups: responses given human-written comments, and responses given machine-generated comments. We considered rater-assessed Correctness for each response and the time taken for participants to complete each question. We note a significant difference in rater-assessed Correctness between these two populations. We show the distribution of Rater-Assessed Correctness scores in Figure 6. Participants given human-written summaries scored an average of 3.52 out of 5, while those given machine-generated summaries scored an average of 3.40 out of 5. This difference is significant using a two-tailed Mann-Whitney U test ($p = 0.029$).

We did not find a significant difference in times to complete comprehension questions in general ($p = 0.30$). The distribution of average time taken by participants per snippet is shown in Figure 7—while we filtered out participants who took longer than 30 minutes per question, some of the snippets elicited substantial variability in time taken to complete (e.g., snippets 8, 23, and 24).

We find that human-written summaries help participants answer comprehension questions more correctly compared to machine-generated summaries. This result is reasonably intuitive—recall from Section 1 that automatic summarization techniques can produce unhelpful comments. We suspect that these results are explained by participants struggling to answer some comprehension questions when given low-quality machine-generated summaries.

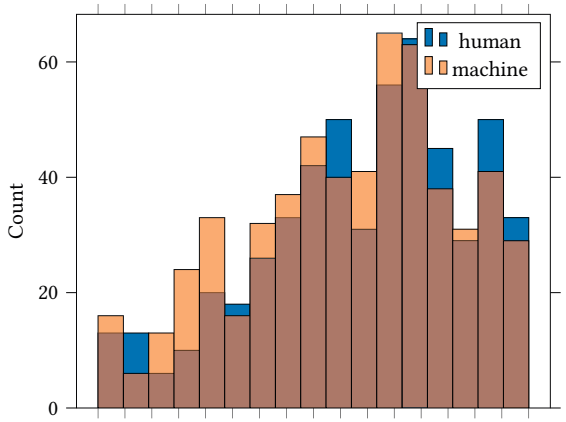

Figure 6: Distribution of rater-assessed Correctness. Each quarter-point increment has two semi-transparent bars that indicate the number of answers, given either machine-generated or human-written comments, that scored in that range. The distributions differ significantly using a two-tailed Mann Whitney U test ($p = 0.029$).

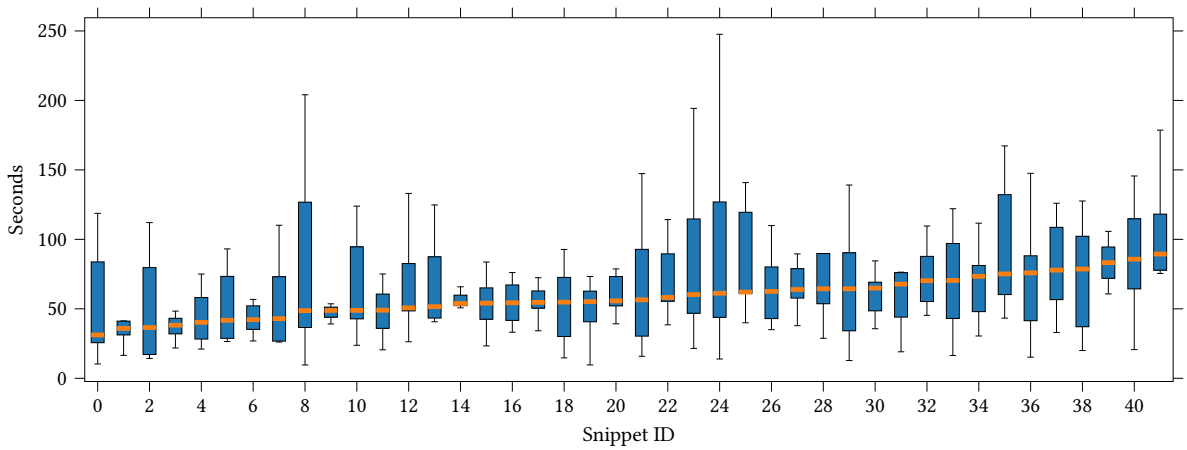

Figure 7: Distribution of times taken by participants for each snippet shown in our study. In general, questions took between 30 to 120 seconds for participants to answer. The data are sorted by ascending average time to complete; in the survey, participants were shown snippets in random order to mitigate fatigue and learning effects.

6.2 RQ2: Developer Perception of Summaries

Next, we consider how helpful developers perceive the given summary to be when completing a comprehension task. We split our data into two groups: participants given human-written summaries, and those given machine-generated summaries. After seeing each snippet, participants were asked to rate the quality of summaries on a scale from 1 to 5. Using a two-tailed Mann-Whitney U test, we fail to reject the null hypothesis that the two groups of samples follow the same distribution ($p = 0.108$). While we cannot conclude that the distributions are the same, our data suggests the participants did not see a clear difference in quality between human-written and machine-generated comments.

Moreover, participants' quality ratings did not correlate with their correctness on comprehension questions. We compared each participant's perception of summaries to the rater-assessed Correctness scores for each participant's answers. Using Spearman's ρ, we found $\rho = -0.022$ with $p = 0.518$, indicating no significant correlation between developer-perceived summary quality and rater-assessed Correctness.

We find that participants exhibited no significantly predictable patterns in assessing the quality of summaries for code they are examining. Our data suggests that developers' subjective ratings are not reliable predictors of how much a summary helps them understand code. This aligns with previous findings that developer intuitions can align poorly with reality when assessing which information is most relevant for software maintenance tasks (e.g,. [20, Sec. 4.3]).

6.3 RQ3: Summaries, Metrics and Comprehension

We next investigate the extent to which BLEU and ROUGE scores reflect how well machine-generated summaries improve a developer's ability to comprehend code. Prevailing summarization techniques are evaluated using these metrics, based on the assumption that a higher BLEU or ROUGE score indicates the quality of the resulting summary is higher (and thus more useful to a developer). We assess the degree to which that assumption holds in practice for developer comprehension tasks.

For this research question, we consider all participant responses that were given machine-generated comments. Next, we average the rater-assessed Correctness for all participant responses for each code snippet in our dataset. We note there is a very weak ($\rho = .151$) but significant ($p = 0.0004$) correlation between a summary's ROUGE score and rater-assessed Correctness. A scatter plot of this data is shown in Figure 8. We also applied the same methodology using BLEU scores. The results are similar: a very weak but significant correlation between BLEU scores and rater-assessed Correctness ($\rho = .140, p = 0.0008$).

BLEU and ROUGE scores were very weakly correlated with rater-assessed Correctness (Spearman's $\rho = 0.151, p = 0.0004$ for ROUGE vs. Correctness; $\rho = 0.140, p = 0.0008$ for BLEU vs. Correctness). Developer comprehension, as measured by extrinsic correctness at performing comprehension-based tasks, is not strongly related to the internally-measured quality of the machine-generated summary they receive. We suggest considering alternative methodologies to the use of BLEU and ROUGE to evaluate summarization models.

6.4 Qualitative Discussion of Writing Tasks

In this subsection, we discuss qualitative data to augment our quantitative evaluation, with a specific focus on the writing section of our study. Recall from Section 5.3 that participants completed code writing tasks. For writing tasks, they were given a Binary Tree, Linked List, or Graph class with one method missing. They were given either human-written or machine-generated summaries for all methods in the class. Based on a held-out test suite, they were tasked with writing the missing method. Participant submissions were executed against a held-out test suite as a proxy for correctness. While we do not report significant correlations or results with our writing tasks, we note two points for qualitative discussion.

First, keywords present in summaries appeared to influence variable names used by participants in their solutions. For example, our Graph class was implemented with an adjacency matrix

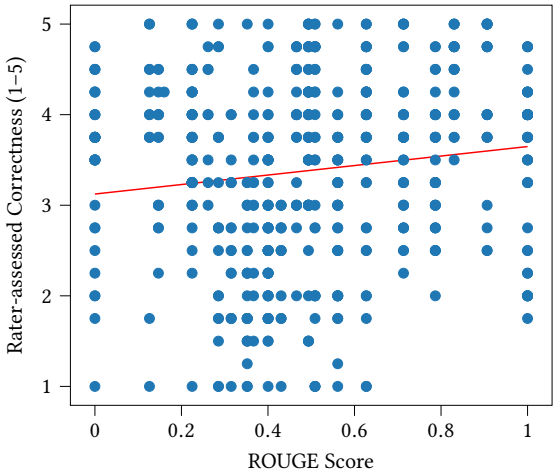

Figure 8: Scatter plot showing the distribution of ROUGE scores for each machine-generated code summary and the rater-assessed Correctness for participants who answered comprehension questions using that summary. We note a very weak correlation using Spearman's $\rho = 0.151$ ($p = 0.0004$). The best fit line is shown in red. High or low ROUGE scores do not appear to influence developer comprehension.

called `adjMatrix`. Participants were tasked with implementing a `getUnvisitedChildNode` method. The machine-generated summaries for this method did not mention the adjacency matrix. We found that only 20% of participants who were given machine-generated summaries used the matrix, whereas 75% of participants who were given human-written summaries used the matrix.

Second, we note that method length appeared to modulate the usefulness of the summary. For example, one task involved a Binary Tree `delete` method that contained over 100 lines of code. None of the participants who were given this method completed the task (i.e., all submissions passed 0 test cases), regardless of whether they were given machine-generated or human-written summaries.

6.5 Results Summary

First, we find that human-written summaries help developers comprehend code significantly better than do machine-generated summaries. Second, developer perception of summary quality, whether human-written or machine-generated, did not significantly correlate with developer comprehension—developers cannot assess which summaries are most helpful. Finally, we found that BLEU and ROUGE scores were significantly uncorrelated (i.e., $\rho = 0.151$ with $p = 0.0004$ for ROUGE and $\rho = 0.140$ with $p = 0.0008$ for BLEU) with developer comprehension—developers do not benefit from summaries with higher-valued BLEU or ROUGE scores. This indicates a need for new metrics for measuring automatic summarization techniques.

7 THREATS TO VALIDITY

We acknowledge threats to validity in our study, including our participant selection process, our code snippets, our code summarization model, and the design and sensitivity of our methodology.

Our participant selection pool varied in programming experience, from second-year undergraduate computer science students to professional developers with under five years of experience. The overwhelming majority were undergraduate students (i.e., 35 participants), thus subgroup analysis was difficult. Moreover, our institution primarily focuses on a C/C++ curriculum—several participants were wary of participating in a Java-based survey.

We selected 50 random snippets from a larger dataset containing over 2 million methods. While we constrained our search, it is possible that our 50 selected snippets are not representative of the entire dataset. Moreover, our results are based on Java snippets, which may not generalize to other languages. This is endemic to code summarization—we mitigate this threat by working with a dataset published specifically for this task.

We considered one code summarization model in our experiments [34]. While others are available (e.g., [28]), we selected this model because it represents the state-of-the-art and for experimental expediency (i.e., because the dataset was readily available). While possible that other techniques could influence comprehension differently (e.g., models that use templates or that enforce particular comment styles), we choose to focus on deep learning techniques.

Finally, we selected particular program comprehension activities (answering questions and writing code). These activities may not elicit the clearest patterns relating to the impact of summaries. We referred to a large body of literature, building tasks from established best practices (i.e., Sillito *et al.* [61] and Pacione *et al.* [50]).

8 CONCLUSION

Automatic code summarization is becoming increasingly important for addressing the shortage of well-documented code. Well-documented code is known to influence comprehension for both new and experienced developers. However, while state-of-the-art automatic code summarization techniques exist, they are frequently internally evaluated with respect to metrics like BLEU and ROUGE, designed for machine translation tasks. These metrics do not necessarily reflect what developers rely on when examining code. To our knowledge, no prior work has explored the impact of automatic summarization on extrinsic developer comprehension.

In this paper, we present a human study of 45 undergraduate students, graduate students, and professional developers. These participants examined Java methods documented with human-written or machine-generated summaries, and answered comprehension questions about each. We found that human-written summaries yielded significantly improved developer comprehension compared to machine-generated summaries ($p = 0.029$). However, we did not find a significant correlation between developer perception of summary quality and their comprehension. Finally, we found that BLEU and ROUGE scores were both significantly uncorrelated with comprehension ($\rho = 0.140, p = 0.0008$ and $\rho = 0.151, p = 0.0004$, respectively). These results suggest a need to develop more appropriate metrics for evaluating the quality or effectiveness of automatic code summarization techniques.

9 ACKNOWLEDGMENT

The authors gratefully acknowledge the partial support of US National Science Foundation (NSF) (CCF-1452959, CCF-1717607, CCF-1908633, CCF-1763674).

REFERENCES

[1] Abbes, M., Khomh, F., Gueheneuc, Y.-G., and Antoniol, G. An empirical study of the impact of two antipatterns, blob and spaghetti code, on program comprehension. In *2011 15th European Conference on Software Maintenance and Reengineering* (2011), IEEE, pp. 181–190.

[2] Allamanis, M., Peng, H., and Sutton, C. A convolutional attention network for extreme summarization of source code. In *International Conference on Machine Learning* (2016), pp. 2091–2100.

[3] Amalfitano, D., Fasolino, A. R., Polcaro, A., and Tramontana, P. The dynaria tool for the comprehension of ajax web applications by dynamic analysis. *Innovations in Systems and Software Engineering 10*, 1 (2014), 41–57.

[4] Banerjee, S., and Lavie, A. METEOR: An automatic metric for MT evaluation with improved correlation with human judgments. In *Proceedings of the ACL Workshop on Intrinsic and Extrinsic Evaluation Measures for Machine Translation and/or Summarization* (Ann Arbor, Michigan, June 2005), Association for Computational Linguistics, pp. 65–72.

[5] Bauer, J., Siegmund, J., Peitek, N., Hofmeister, J. C., and Apel, S. Indentation: simply a matter of style or support for program comprehension? In *International Conference on Program Comprehension* (2019), pp. 154–164.

[6] Brooks, R. Using a behavioral theory of program comprehension in software engineering. In *Proceedings of the 3rd international conference on Software engineering* (1978), IEEE Press, pp. 196–201.

[7] Brooks, R. Towards a theory of the comprehension of computer programs. *International journal of man-machine studies 18*, 6 (1983), 543–554.

[8] Buckner, J., Buchta, J., Petrenko, M., and Rajlich, V. Jripples: A tool for program comprehension during incremental change. In *13th International Workshop on Program Comprehension (IWPC'05)* (2005), IEEE, pp. 149–152.

[9] Buse, R. P. L., and Weimer, W. Automatic documentation inference for exceptions. In *International Symposium on Software Testing and Analysis* (2008), pp. 273–282.

[10] Buse, R. P. L., and Weimer, W. Automatically documenting program changes. In *Automated Software Engineering* (2010), pp. 33–42.

[11] Ceccato, M., Di Penta, M., Nagra, J., Falcarin, P., Ricca, F., Torchiano, M., and Tonella, P. The effectiveness of source code obfuscation: an experimental assessment. pp. 178–187.

[12] Cioch, F. A., Palazzolo, M., and Lohrer, S. A documentation suite for maintenance programmers. In *Proceedings of the 1996 International Conference on Software Maintenance* (1996), pp. 286–295.

[13] Coughlin, D. Correlating automated and human assessments of machine translation quality. In *Proceedings of MT summit IX* (2003), pp. 63–70.

[14] de Souza, S. C. B., Anquetil, N., and de Oliveira, K. M. 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* (2005), ACM, pp. 68–75.

[15] Dunsmore, A., and Roper, M. A comparative evaluation of program comprehension measures. *The Journal of Systems and Software 52*, 3 (2000), 121–129.

[16] Fleiss, J. L. Measuring nominal scale agreement among many raters. *Psychological bulletin 76*, 5 (1971), 378.

[17] Floyd, B., Santander, T., and Weimer, W. Decoding the representation of code in the brain: an fmri study of code review and expertise. In *International Conference on Software Engineering* (2017), pp. 175–186.

[18] Fluri, B., Wursch, M., and Gall, H. C. Do code and comments co-evolve? on the relation between source code and comment changes. In *14th Working Conference on Reverse Engineering (WCRE 2007)* (2007), IEEE, pp. 70–79.

[19] Fowkes, J., Chanthirasegaran, P., Ranca, R., Allamanis, M., Lapata, M., and Sutton, C. Autofolding for source code summarization. *IEEE Transactions on Software Engineering 43*, 12 (2017), 1095–1109.

[20] Fry, Z. P., Landau, B., and Weimer, W. A human study of patch maintainability. In *Proceedings of the 2012 International Symposium on Software Testing and Analysis* (2012), pp. 177–187.

[21] HackerRank. HackerRank. https://www.hackerrank.com/.

[22] Haiduc, S., Aponte, J., and Marcus, A. Supporting program comprehension with source code summarization. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 2* (2010), ACM, pp. 223–226.

[23] Haiduc, S., Aponte, J., Moreno, L., and Marcus, A. On the use of automated text summarization techniques for summarizing source code. In *2010 17th Working Conference on Reverse Engineering* (2010), IEEE, pp. 35–44.

[24] Hill, E., Pollock, L., and Vijay-Shanker, K. Automatically capturing source code context of nl-queries for software maintenance and reuse. In *2009 IEEE 31st International Conference on Software Engineering* (May 2009), pp. 232–242.

[25] Hofmeister, J. C., Siegmund, J., and Holt, D. V. Shorter identifier names take longer to comprehend. *Empirical Software Engineering 24*, 1 (2019), 417–443.

[26] Hsieh, H.-F., and Shannon, S. E. Three approaches to qualitative content analysis. *Qualitative health research 15*, 9 (2005), 1277–1288.

[27] Hu, X., Li, G., Xia, X., Lo, D., and Jin, Z. Deep code comment generation. In *Proceedings of the 26th Conference on Program Comprehension* (2018), ACM, pp. 200–210.

[28] Hu, X., Li, G., Xia, X., Lo, D., and Jin, Z. Deep code comment generation with hybrid lexical and syntactical information. *Empirical Software Engineering* (2019), 1–39.

[29] Ibrahim, W. M., Bettenburg, N., Adams, B., and Hassan, A. E. On the relationship between comment update practices and software bugs. *Journal of Systems and Software 85*, 10 (2012), 2293–2304.

[30] Iyer, S., Konstas, I., Cheung, A., and Zettlemoyer, L. Summarizing source code using a neural attention model. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)* (2016), pp. 2073–2083.

[31] Jbara, A., and Feitelson, D. G. On the effect of code regularity on comprehension. In *Proceedings of the 22nd international conference on program comprehension* (2014), ACM, pp. 189–200.

[32] Kajko-Mattsson, M. A survey of documentation practice within corrective maintenance. *Empirical Software Engineering 10*, 1 (2005), 31–55.

[33] Lawrie, D., Morrell, C., Feild, H., and Binkley, D. Effective identifier names for comprehension and memory. *Innovations in Systems and Software Engineering 3*, 4 (2007), 303–318.

[34] LeClair, A., Jiang, S., and McMillan, C. A neural model for generating natural language summaries of program subroutines. In *2019 International Conference on Software Engineering (ICSE)* (May 2019).

[35] LeClair, A., and McMillan, C. Recommendataions for datasets for source code summarization. In *2019 Annual Conference of the North Americal Chapter of the Association for Computational Linguistics (NAACL)* (June 2019).

[36] LeetCode, LLC. LeetCode - The World's Leading Online Programming Learning Platform. https://leetcode.com/.

[37] Liang, Y., and Zhu, K. Q. Automatic generation of text descriptive comments for code blocks. In *Thirty-Second AAAI Conference on Artificial Intelligence* (2018).

[38] Lin, C.-Y. ROUGE: A package for automatic evaluation of summaries. In *Text Summarization Branches Out* (Barcelona, Spain, July 2004), Association for Computational Linguistics, pp. 74–81.

[39] Liu, B., Wang, T., Zhang, X., Fan, Q., Yin, G., and Deng, J. A neural-network based code summarization approach by using source code and its call dependencies. In *Proceedings of the 11th Asia-Pacific Symposium on Internetware* (2019), pp. 1–10.

[40] Loyola, P., Marrese-Taylor, E., and Matsuo, Y. A neural architecture for generating natural language descriptions from source code changes. *arXiv preprint arXiv:1704.04856* (2017).

[41] Malhotra, M., and Chhabra, J. K. Class level code summarization based on dependencies and micro patterns. In *2018 Second International Conference on Inventive Communication and Computational Technologies (ICICCT)* (2018), IEEE, pp. 1011–1016.

[42] McBurney, P. W., and McMillan, C. Automatic documentation generation via source code summarization of method context. In *Proceedings of the 22nd International Conference on Program Comprehension* (New York, NY, USA, 2014), ICPC 2014, Association for Computing Machinery, p. 279–290.

[43] McBurney, P. W., and McMillan, C. Automatic documentation generation via source code summarization of method context. In *Proceedings of the 22nd International Conference on Program Comprehension* (2014), pp. 279–290.

[44] McBurney, P. W., and McMillan, C. Automatic source code summarization of context for java methods. *IEEE Transactions on Software Engineering 42*, 2 (2015), 103–119.

[45] Moore, J., Gelman, B., and Slater, D. A convolutional neural network for language-agnostic source code summarization. In *ENASE* (2019).

[46] Moreno, L., Aponte, J., Sridhara, G., Marcus, A., Pollock, L., and Vijay-Shanker, K. Automatic generation of natural language summaries for java classes. In *2013 21st International Conference on Program Comprehension (ICPC)* (May 2013), pp. 23–32.

[47] Nazar, N., Hu, Y., and Jiang, H. Summarizing software artifacts: A literature review. *Journal of Computer Science and Technology 31*, 5 (Sep 2016), 883–909.

[48] Nazar, N., Hu, Y., and Jiang, H. Summarizing software artifacts: A literature review. *Journal of Computer Science and Technology 31*, 5 (2016), 883–909.

[49] Nazar, N., Jiang, H., Gao, G., Zhang, T., Li, X., and Ren, Z. Source code fragment summarization with small-scale crowdsourcing based features. *Frontiers of Computer Science 10*, 3 (2016), 504–517.

[50] Pacione, M. J., Roper, M., and Wood, M. A novel software visualisation model to support software comprehension. In *11th working conference on reverse engineering* (2004), IEEE, pp. 70–79.

[51] Papineni, K., Roukos, S., Ward, T., and Zhu, W.-J. Bleu: a method for automatic evaluation of machine translation. In *Proceedings of the 40th Annual Meeting of the Association for Computational Linguistics* (Philadelphia, Pennsylvania, USA, July 2002), Association for Computational Linguistics, pp. 311–318.

[52] Prechelt, L., Unger-Lamprecht, B., Philippsen, M., and Tichy, W. F. Two controlled experiments assessing the usefulness of design pattern documentation in program maintenance. *IEEE Transactions on Software Engineering 28*, 6 (2002), 595–606.

[53] Reiter, E. A structured review of the validity of BLEU. *Computational Linguistics 44*, 3 (2018), 393–401.

[54] Rodeghero, P., McMillan, C., McBurney, P. W., Bosch, N., and D'Mello, S. Improving automated source code summarization via an eye-tracking study of programmers. In *Proceedings of the 36th international conference on Software engineering* (2014), pp. 390–401.

[55] Roehm, T., Tiarks, R., Koschke, R., and Maalej, W. How do professional developers comprehend software? In *Proceedings of the 34th International Conference on Software Engineering* (2012), IEEE Press, pp. 255–265.

[56] Schröter, I., Krüger, J., Siegmund, J., and Leich, T. Comprehending studies on program comprehension. In *International Conference on Program Comprehension* (2017), pp. 308–311.

[57] Shepherd, D., Fry, Z. P., Hill, E., Pollock, L., and Vijay-Shanker, K. Using natural language program analysis to locate and understand action-oriented concerns. In *Proceedings of the 6th International Conference on Aspect-Oriented Software Development* (New York, NY, USA, 2007), AOSD '07, Association for Computing Machinery, p. 212–224.

[58] Shido, Y., Kobayashi, Y., Yamamoto, A., Miyamoto, A., and Matsumura, T. Automatic source code summarization with extended tree-lstm. *ArXiv abs/1906.08094* (2019).

[59] Siegmund, J. Program comprehension: Past, present, and future. In *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)* (2016), vol. 5, IEEE, pp. 13–20.

[60] Siegmund, J., Peitek, N., Parnin, C., Apel, S., Hofmeister, J., Kästner, C., Begel, A., Bethmann, A., and Brechmann, A. Measuring neural efficiency of program comprehension. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering* (2017), pp. 140–150.

[61] Sillito, J., Murphy, G. C., and De Volder, K. Questions programmers ask during software evolution tasks. In *Proceedings of the 14th ACM SIGSOFT international symposium on Foundations of software engineering* (2006), pp. 23–34.

[62] Sridhara, G., Hill, E., Muppaneni, D., Pollock, L., and Vijay-Shanker, K. Towards automatically generating summary comments for java methods. In *Proceedings of the IEEE/ACM international conference on Automated software engineering* (2010), ACM, pp. 43–52.

[63] Sridhara, G., Pollock, L., and Vijay-Shanker, K. Automatically detecting and describing high level actions within methods. In *Proceedings of the 33rd International Conference on Software Engineering* (2011), ACM, pp. 101–110.

[64] Sridhara, G., Pollock, L., and Vijay-Shanker, K. Generating parameter comments and integrating with method summaries. In *2011 IEEE 19th International Conference on Program Comprehension* (2011), IEEE, pp. 71–80.

[65] Storey, M.-A. Theories, tools and research methods in program comprehension: past, present and future. *Software Quality Journal 14*, 3 (2006), 187–208.

[66] Tenny, T. Program readability: Procedures versus comments. *IEEE Transactions on Software Engineering 14*, 9 (1988), 1271–1279.

[67] Von Mayrhauser, A., and Vans, A. M. Program comprehension during software maintenance and evolution. *Computer 28*, 8 (1995), 44–55.

[68] Wang, T., and Liu, Y. Jsea: A program comprehension tool adopting lda-based topic modeling. *International Journal of Advanced Computer Science and Applications 2*, 3 (2017).

[69] Wettel, R., Lanza, M., and Robbes, R. Software systems as cities: A controlled experiment. In *Proceedings of the 33rd International Conference on Software Engineering* (2011), pp. 551–560.

[70] Woodfield, S. N., Dunsmore, H. E., and Shen, V. Y. The effect of modularization and comments on program comprehension. In *Proceedings of the 5th international conference on Software engineering* (1981), IEEE Press, pp. 215–223.

[71] Xia, X., Bao, L., Lo, D., Xing, Z., Hassan, A. E., and Li, S. Measuring program comprehension: A large-scale field study with professionals. *IEEE Transactions on Software Engineering 44*, 10 (Oct 2018), 951–976.

[72] Zhao, L., Zhang, L., and Yan, S. A survey on research of code comment auto generation. In *Journal of Physics: Conference Series* (2019), vol. 1345, IOP Publishing, p. 032010.