TOWARD AUTOMATIC SUMMARIZATION OF ARBITRARY JAVA STATEMENTS FOR NOVICE PROGRAMMERS

Mohammed Sayed Hassan

Drew University, 2019

Abstract

Novice programmers sometimes need to understand code written by others. Unfortunately, most software projects lack comments suitable for novices. The lack of comments has been addressed through automated techniques for generating comments based on program statements (or lines of code). However, these techniques lack the context of how these statements function since they are aimed toward experienced programmers. In this thesis, I present a novel technique for automatically generating comments for Java statements suitable for novice programmers. My technique not only goes beyond existing approaches to method summarization to meet the needs of novices, it also leverages API documentation when available. In an experimental study of 30 computer science undergraduate students, explanations based on my technique were preferred over an existing approach.

TOWARD AUTOMATIC SUMMARIZATION OF ARBITRARY JAVA STATEMENTS FOR NOVICE PROGRAMMERS

by

Mohammed Sayed Hassan

An Honors Thesis Submitted in Partial Fulfillment of the Requirements for the Degree of Bachelor in Arts with Specialized Honors in Computer Science at Drew University

Spring 2019

ACKNOWLEDGMENTS

I would like to thank my sisters, Isra and Sara, for being great academic role models who were always there to support me. I would also like to thank my parents, Sayed and Sahar, for their motivational support to make my education possible. I would like to thank Dr. Sarah Abramowitz for her invaluable statistical guidance, and my thesis committee for their valued feedback.

TABLE OF CONTENTS

Chapter

Chapter 1

INTRODUCTION

All software used on computers is made of lines of instructions called code, which is written by programmers. Programmers can add new features to a program by writing new lines of code and can fix problems in programs by changing lines of code. Programmers must understand the lines of code to make the appropriate changes to the code to add new features or to fix a problem. As technology continues to evolve, more programmers are needed. To help novice programmers, I propose an automated method to explain, in full English text, any line of code.

1.1 A Motivating Example

Novice programmers may struggle to understand lines of code, and thus struggle to add new features and fix problems in programs. To better explain the importance of understanding code, I will present a common example scenario below:

As you were using your start menu on Windows, you notice the start menu looks grey and simple. Let's say that grey is not your favorite color and you prefer a more colorful, feature-filled start menu. Your start menu is made of some lines of code. To change the style of the start menu, you must change the lines of code, and thus you must understand the lines of code (see Figure 1.1) to change it correctly. After you correctly modified it, you get a much more aesthetically pleasing start menu (see Figure 1.2). This thesis proposes an automated method to explain lines of code to help make this entire process easier.

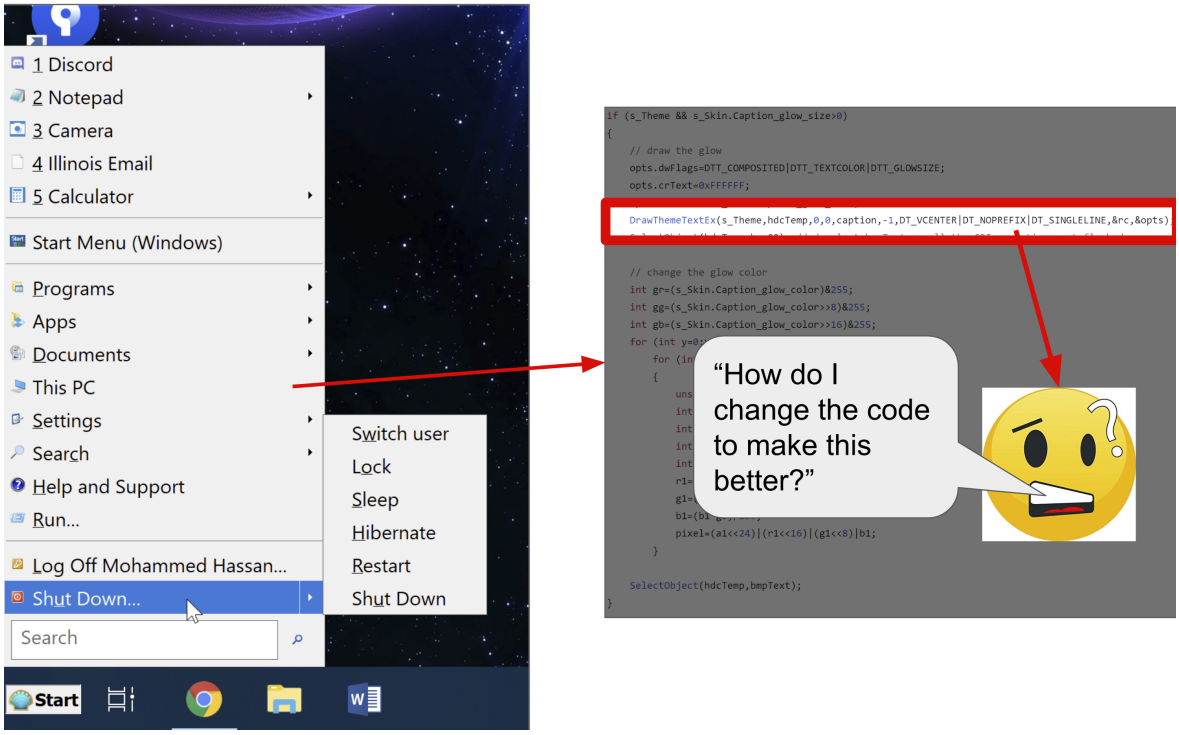


Figure 1.1: Example of why understanding lines of code is important

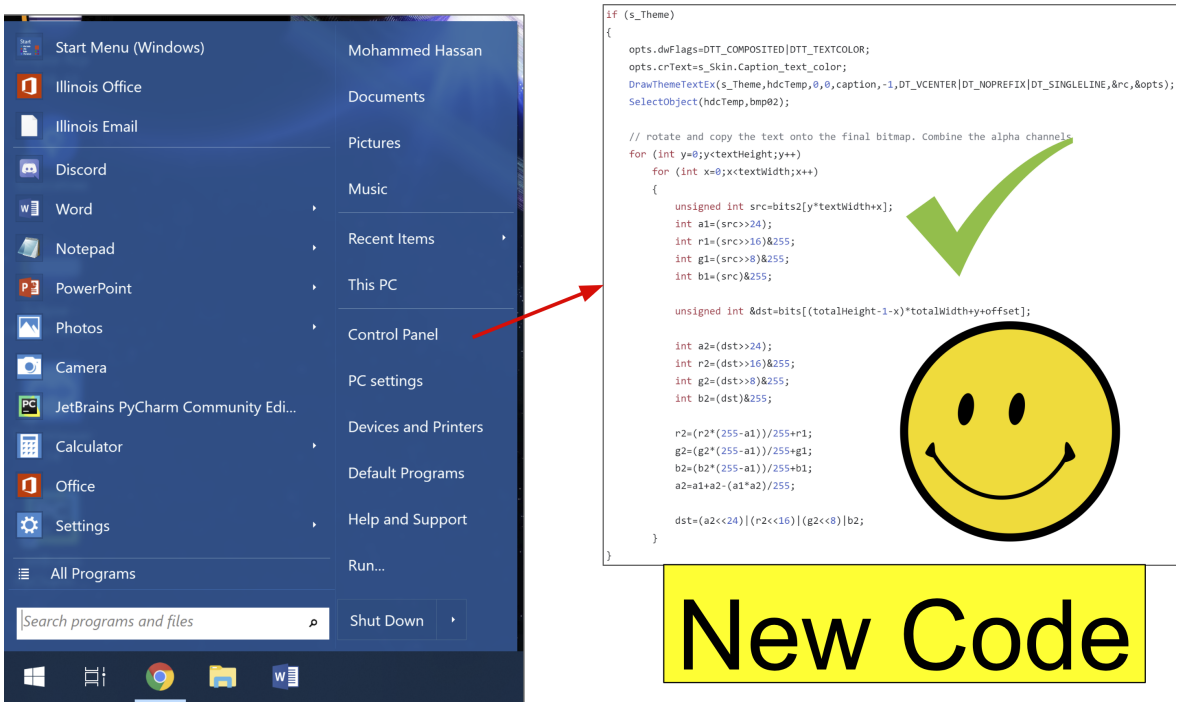


Figure 1.2: Changed lines of code for a better start menu, compared to 1.1

1.2 Understanding Code

While there are ways to learn programming such as courses and websites like Codeacademy, only limited help is for understanding specific lines of code from most projects. For instance, code often lacks documentation and comments explaining how the code works [2, 3] and the author may be unavailable to respond to questions regarding their code. Often code is published online in websites like GitHub for public reuse, but the code must be understood to be reused correctly. In classroom projects, teachers may be unfamiliar with reused code or may lack sufficient availability to help many students with code that lacks documentation.

There are infinite possibilities of what could be written in code, such as methods that contain more written lines of code and specific syntaxes like words, symbols, expressions, values, numbers, and variables which can store any value and even other variables. Due to these infinite possibilities of coding, searching online for a solution may not help novices, because example code found online is not always similar to the code in question. My proposed solution is an automated approach to generate English text explanations of any specific statement of code requested which aim to be helpful for novice programmers and can also be helpful for experts who may be unfamiliar with the code in question.

1.3 Summarizing Code

Lines of code may contain English words or phrases which can be used toward automatic summarization of code, such as method names, class names, variable names (see Figure 1.3). Prior work [4] has evaluated the potential of a Part-Of-Speech (POS) tagger for English words in code, called "SWUM", to be used for summarization of code by automatically processing the labeled POS of words to generate a grammatically correct summary using those words (see Figure 1.4). However, lines of code may not contain useful English word POS information, rendering SWUM to be not a viable option (see Figure 1.5). In addition, SWUM only outputs POS information, identifying

```
private ResultPoint getBlackPointOnSegment(float aX, float aY, float bX, float bY) {
  int dist = distanceL2(aX, aY, bX, bY);
  float xStep = (bX - aX) / dist;
  float yStep = (bY - aY) / dist;

  for (int i = 0; i < dist; i++) {
    int x = round(aX + i * xStep);
    int y = round(aY + i * yStep);
    if (image.get(x, y)) {
      return new ResultPoint(x, y);
    }
  }
  return null;
}
```

Figure 1.3: Example of English words found in code

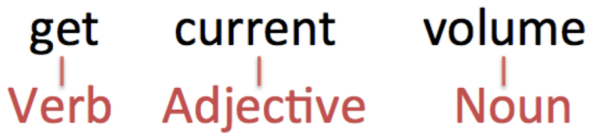


Figure 1.4: Example of POS tagging [4]

English words in code as "noun", "verb", etc, rather than producing a summary of code in English sentences.

Novice programmers commonly struggle to understand the connections between lines of code [12], but SWUM does not address the connection between different lines of code. Examples of connections between lines of code include "if statements" where a certain condition stated within the if statement must be true to execute another line of code, or a "method call" where a "method" holds some lines of code and "calling" it allows for execution of the lines of code inside that "method" (see Figure 1.6).

```
public boolean get(int x, int y) {
  int offset = y * rowSize + (x / 32);
  return ((bits[offset] >>> (x & 0x1f)) & 1) != 0;
}
```

Figure 1.5: Example of code that generally lack useful English words

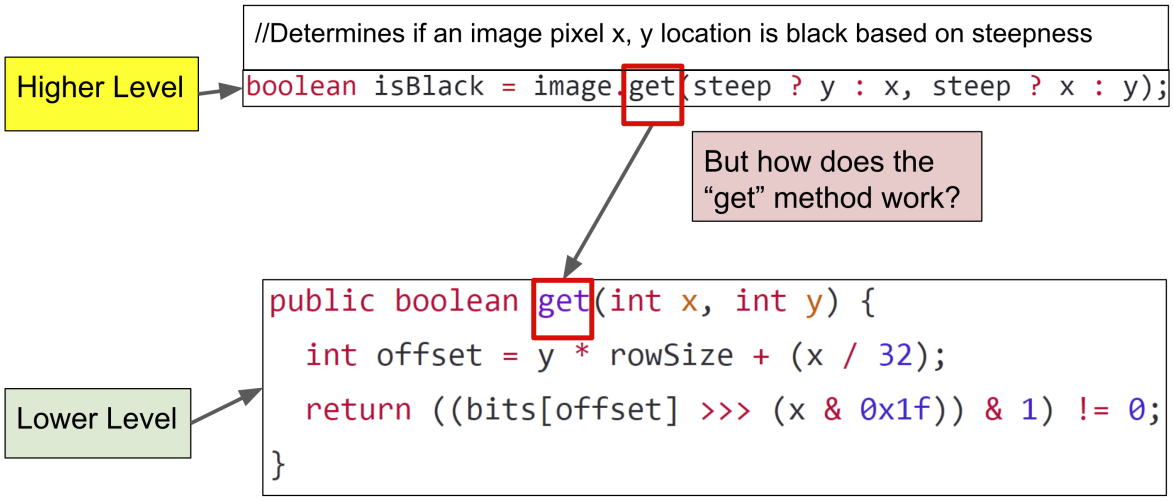


Figure 1.6: How a Novice programmer may struggle with connections between statements such as a method call

1.4 Novice and Expert Summary Differences

Prior summarization approaches were designed for expert programmers to help them understand how to use libraries, or publicly available methods, for their code. However, experts, unlike novices, excel at understanding connections between different lines of code [19]. Novices and experts have different needs for code summarization as novices may need more assistance to understand code. Gugerty, et al. studied how novice programmers debug programs differently from experts. They found that, because the novices' hypothesis of code, or their understanding of how the program works, is greatly inferior to experts, novice programmers tended to introduce more bugs to programs they were debugging. They also took substantially longer than experts to find the bug or did not find the bug at all. Experts took roughly half the time of novices to understand the program, and yet had significantly better hypothesis than the novices [18]. Thus, the need for novice-friendly documentation is crucial to help novice programmers understand code.

Novices' mental representations of code, based on their responses to technical questions about the code, lacked two crucial characteristics which are generally found

in experts' representations. Novices did not understand that code is: (1) Hierarchically structured and (2) explicitly mapped [12]. Hierarchically structured explanations mention the order of execution between lines of code within a program. For instance, a hierarchically structured explanation addresses that, after executing a line of code that is a method call, the lines of code within that method will be executed next in order from the first line of the method to the last, then the code outside that method and under the method call will be executed (see Figure 1.6). To correctly understand the code, the order of execution is crucial to recognize.

Explicitly mapped representations include a clear connection between both high-level (concise) and low-level (exhaustive) explanations of a program [12]. high-level lines of code depend on low-level lines of code. For example, a method call is a high-level line of code as it depends on its method declaration. A method declaration contains the lines of code of the method which are the low-level counterpart of the method call (see Figure 1.6). While both novices and experts can make a hypothesis of a method's purpose based on high-level details such as an intuitive method call name, novices generally can't relate the appropriate low-level lines of code to their own high-level hypothesis [12]. For instance, novices are not sure how the lines of code inside a method declaration lead to the output found within the line of code that called the method, which shows their lack of understanding of how the method works.

This thesis proposes a method that, unlike prior approaches, automatically explain all the significant connections between different lines of code mainly to assist novice programmers in understanding how programs work. The summarization technique will be based on the two main characteristics of experts' understanding of programs, hierarchically structured and explicitly mapped explanations of code, which novices lack in their understanding [12]. Prior approaches utilized a database of already existing explanations of code to automatically generate new explanations of related code [21, 29]; however, programs often lack such a database [2, 3]. The approach in this thesis will not require any prior made (previously existing) explanations as it will dynamically generate new explanations based only on the given source code.

However, if explanations already exist such as API Documentation, an online website explaining various common built-in methods for Java, this approach can optionally utilize such information to further supplement the auto-generated explanations. Prior approaches that are designed for experts are also limited to summarizing only specific types of statements at a higher level, such as a method declaration or class, while the approach this paper proposes aims to support any line of code possible and include low-level details when necessary.

This thesis makes the following contributions: (1) It evaluates current summarization approaches and how they fail to meet the needs of novice programmers. (2) It offers a novel approach toward summarization of any line of code that helps fulfill the needs for novice programmers. (3) It evaluates this approach through a study conducted within Computer Science classes at Drew University to find whether undergraduate students of Computer Science preferred my approach over a competing approach developed by Sridhara, et al [5].

Chapter 2

RELATED WORK

In this section, I will (1) analyze prior approaches that automatically generate English Explanations of source code. Then, I will (2) address the shortcomings of prior approaches.

2.1 Summarization through Previously-made Documentation

Prior approaches created machine learning algorithms that utilize a dataset of already-existent information to automatically generate explanations for lines of code. The information in the dataset could be comments within code, relevant search terms to code, explanations created by hired expert programmers, or other online documentation.

Ying, et al. developed a machine learning approach towards summarizing code fragments in order to better present a code example [21]. Their machine learning approach is trained through a corpus of code fragments. Through training, it can observe what kind of synantic features exist within code summaries. These synantic features include variable declarations, method invocations, assignments, and many more [21]. It also considers term queries and online searching terms that may be common within certain code fragments. These terms and queries can also come from documentation, such as FAQs [21].

Nazar, et al. developed a machine learning approach to summarize source code fragments [29] similar to Ying, et al's [21] through crowdsourcing, which is a process of hiring a group of experienced programmers online to annotate source code [29]. Nazar also uses different synantic features such as class declarations, constructor calls, return statements, part of method signatures, etc.

Oda, et al. developed an approach towards automatic generation of pseudo-code for statement level source code [31]. Pseudo-code allows programmers to more easily understand source code, especially if it is for a language they are not familiar with. Pseudo-code generally has some natural language explanations as well. For this approach to work, it requires a corpus of source code and pseudo-code translation statements. For example, a "%" in Python can be translated as "divisible by," which is more easily understood by those who are unfamiliar with Python. Oda, et al's approach require common tokenizer techniques depending on the goal natural language. For instance, English is separated by spaces which can use a simple tokenizer technique (dividing words by spaces), while Japenese text requires more sophisticated common tokenizer techniques as it has no spaces. Their approach also requires a tokenizer for the coding language being processed. For instance, Python has its own tokenizer. Lastly, their approach requires a parser of the coding language, which generates a tree describing the structure of the code (abstract syntax tree).

Chen, et al. developed an approach to summarize both high and low-level source code details and also developed an approach to find relevant source code based on input search terms resembling an English description of the code being looked for [37]. Their summarization technique utilizes already existent English descriptions that are relevant to the source code that needs to be summarized. They can utilize informal online resources like StackOverflow [37].

Haiduc, et al. developed an approach to summarize source code at many levels, such as classes, methods, package, etc [22]. This approach utilizes already existent comments and documentation within the source code to form a text corpus. Then, it utilizes common text retrieval techniques to determine the most relevant terms within the text corpus. It also uses method names, parameter names, parameter type names, etc to attach roles to every term on the text corpus [22].

2.1.1 Short Comings

These approaches require a pre-existing dataset of relevant English text information, such as summaries, comments, or documentation, of the appropriate code fragments. Approaches that require expert programmers to create summaries of code is limited by the availability of other expert programmers. In addition, previously-made explanations written by expert programmers may not be suitable for novices. Approaches that depend on previously-made comments and documentation are inapplicable because most projects lack comments and documentation [5].

The proposed approach in this thesis accommodates for the limitation of English text information by utilizing only the source code itself with no requirement of other English text information. Often code may lack relevant online solutions and any relevant English descriptions in general.

2.2 Highlighting Prior-Made Documentation

The Eclipse IDE, an advanced Java programming editor, displays upon a mouse hover the full API Documentation page for any specified statement that utilizes built-in methods. Built-in API methods are widely available methods within the Java programming language itself, thus, API documentation does not exist for most methods that are made by other programmers. Since API functions are designed for general use, API documentation will include all possible details that a programmer may need to know about a specific function. As a result, the full API Documentation page may include excessive verbose details that could be irrelevant to what a programmer is seeking [14]. Some prior approaches concentrated on highlighting the most useful details within the verbose API documentation.

Uri Dekel, et al. developed an Eclipse Plugin, "eMoose," which highlights directives within an API documentation page to help find what a programmer may be seeking within verbose text. These directives are Restrictions, Protocols, Locking, Parameters, Returns, Alternatives, Limitations, Side Effects, Performance, Threading, and Security. Restrictions are certain contexts in which it is not safe to invoke a

method; Protocols are restrictions of specific sequences of what is invoked before and after a method; Locking and Threading are synchronization requirements; Alternatives are other methods that are suggested within an API Documentation page; Performance is related to usage of memory and speed of algorithms; and Security are warnings with API Documentation of potential security vulnerabilities when using certain methods [14].

Pandita, et al. developed an approach that infers method specifications from API Documentation without requiring code contracts [15]. Code contracts specify required method inputs (pre-conditions), as well as expectations after execution (post-conditions). Most API Documentation does not have the correct formalized form of code contracts but is specified informally in natural language text descriptions [15]. Their approach is useful, for example, to quickly notify a programmer who encounters an exception from inputting parameters of incorrect type to a built-in method. Their evaluation involved determining the precision and recall of contract sentences from API Documentation. Contract sentences were identified grammatically through POS tagging [15].

Robillard, et al. similarly developed an approach to recommend fragments of API Documentation. This approach distinguishes usage directives, constraints, threats, alternative API elements, dependent API elements, best practices, and improvement options, which are all extracted from the API documentation text [16].

2.2.1 Short Comings

These approaches were not evaluated on novice student programmers. Unlike these approaches, my study involved college undergraduates ranging from freshman to seniors, from less than one year of experience in programming to 5 or more years. Thus, my approach is more suitable for novice student programmers.

In addition, these approaches can only be used on available API Documentation. Since most programmers don't write documentation on their code [5], these

approaches are limited to only built-in Java functions that have online API Documentation available, which majority of functions do not. The approach proposed in this thesis combines information both from available API Documentation and from my own approach towards explaining any statement without the need for written documentation. This thesis approach also uses a portion of API Documentation to help prevent verbose, unnecessary details. Thus, this thesis accommodates for the limitations of API-dependent approaches while still taking advantage of available API Documentation.

2.3 Highlighting Source Code

Fowkes, et al. developed a tool, called "TASSAL," which highlights what they consider to be the most important areas of source code to help developers concentrate on understanding the more informative areas of source code [36]. They hide unimportant code, commonly known built-in methods which have API documentation to bring more attention to the lines of code that are unique to the program in question [36]. However, this approach does not summarize source code. Novices may, therefore, struggle with the most important highlighted source code and other code in general. Their approach was also evaluated on expert programmers with at least 4 years of industry experience [36].

2.4 Summarization based on Source Code

Some prior approaches also utilize only information located within the source code itself to generate explanations. These could include English words & POS tagging within the source code like a method or class name, some dependencies between lines of code, inputs & outputs of method calls, frequency of method calls within a program, and so on. These approaches include the major advantage of supporting most code that does not include documentation, comments, or any other outside information.

Sridhara, et al [5] developed an approach towards automatic generation of Java comments in source code. Their approach utilizes SWUM for linguistic information

and a specific selection of statements from a method body. These specific selections are called s-units, which consist of a method's return, method calls in void methods, data dependencies in variable assignments, and conditional statements. Void methods lack return statements and would have other effects other than returning a value, thus defaulting to mentioning the names of method calls instead. Data dependency is when a statement utilizes a variable, which depends on the corresponding variable assignment(s) (see Figure A.4 for an example).

Sridhara, et al. also have another approach towards detecting and describing high-level actions with methods [7] that utilizes linguistic information within loops, conditions, and similar sequences. This is similar to the previous approach of automatic generation of Java comments [5], but with the aim of identifying the high-level actions within methods rather than generating source code comments.

Newman, et al. claimed to have developed an approach toward summarizing methods without any dependence on intuitive English words [39]. They fill in one manually written template with object names inside the return statement, parameter names and its object names, and method call names [39].

McBurney, et al. developed an approach towards automatic documentation summarization by using method context [6]. The method context includes the statement which called the method, statements that supply the method's input, and statements which use the method's output [6]. Their aim was to better explain when and how to use a Java method. This approach utilizes PageRank Call Graphs to obtain a method's context [6]. Call Graphs rank the most frequently method calls, as well as their location. Their approach uses locations of method calls as an example usage for documentation.

Sridhara, et al further developed an approach towards parameter comment generation utilizing some of their own s-units, as well as some control and data dependencies of relevant variables [10].

Moreno, et al. developed an approach towards class summarization [11]. Their

approach utilizes method stereotypes classification definitions, class stereotypes classification definitions, and linguistic information within method signatures and class definitions to summarize classes. This provides summaries of linguistic information within method signatures and class definitions.

Wang, et al. developed an approach to summarize source code for object related statement sequences [32]. Object-related statement sequences are a set of statements that are assignments/declarations and/or method calls that are associated with each other. Statements are associated with each other if, for instance, it supplies an input of a method call statement [32]. Then, similarly to Sridhara et al [5], it utilizes Part-Of-Speech information, as well as actions and themes of every English word located within each object related statement. Themes could, for instance, be a noun or a method argument. Actions are verbs. The participants used to evaluate Wang, et al's explanations had experience ranging from 5-10 years; the majority of them considered themselves to be expert programmers [32].

2.4.1 Short Comings

These approaches depend on the words located only within the same method. If the words are not intuitive (not English words) or if dependencies exist outside the same method, these approaches may not work. For instance, these approaches would not explain a variable that stores a value returned from a method call, because that method call is based on its method declaration which is outside the method the variable is in (see Figure 1.6). Instead, these approaches default to mentioning the name of the method, which may not be intuitive English words or phrases. Novices may need help understanding how certain methods work as their approach was designed for experts.

Like Sridhara, et al's approach [7], the proposed approach in this thesis utilizes return statements, method calls, data dependencies in variables, and conditions to allow us to identify possible important lines of code within a method declaration. However, unlike Sridhara's this thesis automatically utilizes inter-dependencies, dependencies that exist outside the method in question, to generate explanations that are suitable

for novices. This thesis has the advantage of accommodating non-intuitive method and variable names by understanding the origin of these methods or variables through control and data dependency summarization that may exist in other methods. A control dependency is when a certain condition stated within a statement must be true to execute another statement(s). For example, a certain condition stated within an "if statement" must be true to execute certain other line(s) of code (see Figure A.7 for an example of an "if statement" in code). Prior approaches were restricted to methods or classes, while the approach in this thesis aims to go beyond to summarize any line of code. Although the thesis will analyze some of the s-units as Sridhara does [7], my approach goes beyond by analyzing the inter-dependencies within these s-units. Thus, with the addition of inter-dependency explanations within the thesis statement-level approach, the proposed approach can generate more detailed lower-level explanations. These detailed lower-level summaries can be useful for novices or for those who are unfamiliar with the lower-level source code of a project.

Prior approaches do not explain operators, such as addition and subtraction, or comparators like less than or greater than. This thesis has an approach to explain statements with multiple mathematical operators and comparators. The thesis approach proposes a method to automatically define operators and their usage in an expression.

Some prior approaches are limited to mentioning names of objects and methods which both Novices and Experts alike can do as its high-level details [12]. Some prior approaches also generalized a template that utilizes object names and method names for all cases without evaluation by human subjects to verify if their approach is helpful. Novices can make a reasonable hypothesis of high-level details such as method names [12], which may be the only aspect summarized by prior approaches. Novices struggle with low-level details, and dependency connections between high-low-level details [12] which my approach will address.

Prior approaches that are completely dependent on method call frequency are missing other potentially valuable information (such as returns, variables, etc). A

novice may not understand the origins of a variable or an expression within a return statement, for instance. Lastly, the subjects of the prior evaluations were conducted on expert programmers, while the evaluation of this thesis had undergraduate students less than 1-3 years of programming experience.

2.5 Observational Studies

Other prior work conducted observational studies to better understand what is needed for summarization of source code based on expert programmers' perception. They try to define what is a "good" code summary by analyzing documentation of code in comparison to its source code counterparts, analyzing experts' explanations of code in comparison to the authors' explanations and analyzing how current documentation is used to understand code.

Rodeghero, et al. made an eye-tracking study with experienced programmers to help identify what lines of code are the most important for explanations, then used the results of their study as a basis for summarizing code [8]. Unfortunately, this approach can produce explanations only for experienced programmers as it is based on the perceived needs for explanations of experts themselves; novices' explanation needs differ [12]. This approach has the same limitation of dependence on intuitive words within a single method declaration and still does not utilize inter-dependencies.

Rodeghero, et al also conducted a study of how often APIs are used within descriptions of source code [9]. They concluded that more API keywords exist in lower-level (exhaustive) manually written method summaries than higher-level (concise) ones. This finding aligns well with how my work uses APIs whenever available. However, user-defined source code often does not have API documentation, which creates the need for the proposed thesis approach for non-API code.

Nielebock, et al conducted a study on comments within programs and how they assist novice and expert programmers toward program comprehension [38]. They found that experts were much more likely to correctly complete coding tasks in comparison to novices even with the same API documentation, code, and tasks given [38]. This

suggests that API documentation alone is not enough to assist novice programmers. In general, they also found that programmers consider in-code comments to only be slightly useful for program comprehension [38]. This finding suggests that even if programs have comments within code, it is still not very useful in general and further summarization techniques are needed to assist novices.

McBurney, et al analyzed keyword similarities between the source code and explanations of the code from reader-written summaries and author-written summaries. Reader-written summaries are from programmers selected to explain the code. Author-written summaries are from documentation of a project. They found that the more similar author-written explanations are to source code, the more accurately it is perceived by outside readers of the code [20]. This finding aligns well with my approach that strives to summarize lower-level source code inter-dependencies. McBurney et al. also conducted a user-study comparing the approach of Sridhara, et al [5] to his own approach [6] which included method context unlike Sridhara, et al [20]. He found that programmers preferred his approach [6] to understanding how a method was used. My approach takes McBurney's approach [6] even a step further by also including inter-dependencies within statements of method contexts.

My approach addresses the limitations in the above-mentioned studies in the following ways: (1) The evaluation of my approach involved undergraduate students of Computer Science rather than experienced programmers. (2) My approach summarizes based on data-dependencies, control-dependencies, and inter-dependencies between any line(s) of code rather than dependence on English words within the source code. (3) My approach can optionally utilize API documentation, if available.

Chapter 3

SUMMARIZATION OF ARBITRARY STATEMENTS

The last chapter addressed the ways in which some prior approaches utilized already-existent explanations to either highlight important details or generate new explanations of similar code. However, these approaches are problematic as most code lacks such explanations. Thus, my approach aims to solve this problem by generating new explanations based on the given source code rather than utilizing any prior made explanations. However, if API Documentation exists, then this approach utilizes it to further supplement explanations. Some prior approaches have depended completely on intuitive English words, like method names, variable names, class names, etc. This approach aims to accommodate non-intuitive names (that are not words) by understanding the origin of methods or variables through control and data dependency summarization that may exist in other methods. Prior approaches do not explain operators such as +, -, / or divide, * or multiply. My approach proposes a method to automatically explain operators and their usage in an expression. Many prior approaches only summarize dependencies within the same method, but important dependencies may exist outside of the method. For instance, a variable may store the value returned from a method call, and that method call is based on its method declaration which is outside the method the variable is in (see Figure 1.6). This proposed approach will support dependencies outside the method, called "inter-dependencies." Prior source-code summarization approaches are limited to specific types of statements, such as method declarations or classes, while this approach aims to support any line of code possible.

This approach towards summarizing methods involves 4 main components: (1) Selecting the most significant lines of code within the method to summarize, (2) lexicalizing (or creating English explanations of) those significant lines of code, (3) finding

and lexicalizing their most significant dependencies, (4) using of API Documentation when available. This process of finding the most significant lines of code within a method is partially based on the s-units of Sridhara, et al [5]. An S-unit is a source code statement such as a return, a method call, a variable assignment, or a conditional statement [5]. Like Sridhara, my approach utilizes return statements, method calls, data dependencies in variables, and conditions to allow us to identify possible important lines of code within a method declaration. However, to generate method summaries that are more useful for novice programmers, we go beyond Sridhara by utilizing dependencies found from these s-units, as well as recursively processing complex expressions and usage of API Documentation.

We created a set of rules for complicated expressions, usage of API Documentation, and dependency lexicalization based on the authors' 10+ years collective experience tutoring novice programmers. This approach is aimed towards: (1) Automatically summarizing the main action of a method, (2) including information necessary for summaries to be understandable by novice programmers.

3.1 Non-Void and Void Methods

A novice programmer could be confused about what a specific statement does or how it works. My approach aims to automatically generate explanations of any statement that is appropriate for a novice programmer so that the user can have as much information as needed to fully understand the statement in question. An example of the practicality of this approach is shown in Figure A.2, where a novice is provided with a descriptive explanation of the mouse-hovered statement on-demand without the need for any external documentation or resources.

This statement could have a method call that is either void or non-void and, as a result, have other statements within its method body that could be helpful towards explaining how the method works. This method body exists outside the method the statement is in, which is also known as an inter-dependency. Thus, we must determine the most significant lines of code of the inter-dependent method. The process

is shown in Algorithm B.1. For non-void Methods, we adapt the idea of Sridhara, et al [5] that selects return statements for non-void methods, which are called "ending s-units." For void methods, we use the most frequent variable assignments, sorted by the last occurrence first. This approach is similar to the approach of Sridhara, et al [5] of data-facilitating s-units, however, unlike Sridhara, et al., this approach is not limited to dependencies of "ending s-units", "void-return s-units", and "same action s-units." Data-facilitating s-units are data dependencies (like assignments) that support a particular statement. Ending s-units are return statements [5]. Void-return s-units are method calls that are void, and same action s-units are method calls where its name has the same action POS as the method declaration it exists in [5]. As shown in Figure A.3 for void methods, we consider all variables within the method, starting with the most often assigned or appended, as well as its last occurrence to its first occurrence. As shown in Figure A.4 for non-void methods, we consider statements that supply the return statement.

This leads to Algorithm B.2, where we recursively analyze the sub-expressions located within S. The most common sub-expressions this approach processes are method calls, constructors, variables, complex expressions, enumerators, literals, parameters, and lists. Enumerators contain a list of constants all specified with a specific object type. Lists can contain a list of any objects. Enumerators are defined as their own class, while lists are defined as statements within methods. These sub-expressions tend to have other hard-to-notice dependencies that novice programmers may find useful to understand. An example of hard-to-notice dependencies is shown in Figure A.4.

To prevent duplicate explanations of statements, we keep track of introduced variables and control flow statements in a list of statements, *seen*. Control flow statements are statements that have a condition and are control dependent on other conditional statements. Generally, a variable, parameter, or control flow statement that exists in *seen* was already mentioned, or its type and name from assignments with a common left-hand side were already mentioned. Thus, a statement found in *seen*, later on, will not be explained again.

It is also possible to have an expression within statements that consists of multiple operators, comparators, variables, method calls, etc, which we consider "complex" expressions. We have determined that it is best to automatically explain complex expressions for novice programmers because complex expressions may confuse novices with the many sub-cases to consider and the appropriate order of parentheses that must be followed (E.g. Nested sub-expressions in parenthesis with multiple method calls, variables, in between many operators). The return statement and "offset" variable assignment in Figure A.5 is an example of complex expressions. We consider the following when analyzing complex expressions in Algorithm B.3: Comparators, Mathematical operations, short-hand if statements, BitWise Operators, "NOT" operators, other variables, and literals.

3.2 Lexcalizing Complex Expressions

Algorithm B.3 covers complex expressions that contain multiple mathematical operations, comparators, and bitwise operators. In the case of parentheses, we recursively process the subexpressions that exist within them, from leftmost subexpression in parentheses to rightmost.

3.2.1 Comparators and BitWise Operators

Comparators are generally processed first, as they contain subexpressions to recursively lexicalize in their left and right-hand sides. Table C.1 contains a set of templates for summarizing the most common comparators used, before lexicalizing the left and right subexpressions. The left-hand sub-expression is mentioned before the comparator explanation, and the right-hand sub-expression is mentioned after the comparator explanation.

For more complex sub-expressions, we recursively analyze the multiple possible dependencies located within the left and right-hand sub-expressions, as shown in Figure A.5 where offset and x are the dependencies being explained. Comparator summaries for more complex expressions are shown in Table C.2.

In table C.2, $lhs(e)$ is the left hand side sub-expression and $rhs(e)$ is the right hand side sub-expression (of the comparator). There are special cases where both the left and right-hand expressions of a comparator are variables or boolean literals, which allow more simplistic and intuitive explanations. For instance, as shown in Figure A.7, for 2 boolean values to be equal, they both must be "true" or they both must be "false", and that leads to the entire expression becoming true (otherwise false), which demonstrates a quick explanation for the special case of 2 booleans, as well as the special case of operator ++. Mentioning the special cases as expressions or numbered values based on the previous tables is less intuitive. The special cases for comparator explanations are shown in Table C.3.

Table C.4 contains a set of templates for summarizing the most common BitWise Operators, before lexicalizing the left and right subexpressions.

3.2.2 Simple Expression Exceptions

Exceptions to these generalizations are Java short-hand operators '++', '–', Java not, and short-hand-if statements. These kinds of subexpressions can use a more simple and direct explanation for novices, as shown in Table C.5. These kinds of subexpressions also deviate in appearance from most general mathematical operations and regular control flow statements, which is why a simple and direct explanation is preferred for novices. Figure A.8 shows an example of a short-hand if explanation.

3.3 Lexicalizing Method Calls and Constructors

If an expression is built-in, we automatically use the API Documentation's "return" section, as it generally aligns well at the end of a variable assignment explanation or return statement explanation as shown in Figure A.9.

If the expression in question is a constructor call (e.g... returning a new object), we mention the actual parameters used, as well as their object type. Later, we recursively lexicalize every parameter found to consider its control and data dependencies,

as well as the relevant sub-expressions found within these dependencies. For non-built-in method calls, the same process of lexicalizing parameters is done after recursively analyzing its declaration. Method parameters are analyzed within Algorithm B.4.

When we have a constructor, we just add the phrase "a new" after the variable name but before the name of the constructor. When we have a method call, we just mention that we call the method then the parameters are analyzed in the same recursive analysis in Algorithm B.5. An example of a method call, short-hand-if, and control dependencies are shown in Figure A.10.

When given numerical value variable parameters, we mention them with a phrase such as "calculated" as often their dependencies involve multiple mathematical operations or assignments that affect their numerical values. If we have multiple objects of the same type within the parameters arguments, we also mention "1st, 2nd, and 3rd... etc" and use grammatically correct listing of the parameters (commas, and... etc), which helps introduce the arguments being used before recursive analysis of dependencies in algorithm B.5 to better understand what is being summarized. See Figure A.11 for an example of a constructor call with multiple parameters.

3.4 Data and Control Dependencies

A variable may contain many data and control dependencies. The approach of Sridhara, et al [5] primarily relied on one line summary of s-units, which means that his approach doesn't take any dependencies of the s-unit into account. Unlike the approach Sridhara proposes, this approach considers dependencies (as shown in Figure A.10) since a novice programmer may not easily understand the origins of a variable without explanation of dependencies.

Algorithm B.5 searches for all assignment and appending statements that a variable V depends on, introduces the variable and its type, then recursively lexicalizes the right-hand side of the assignments V depends on. If V depends on an appending statement, we lexicalize the expression being appended. Examples of these have been shown throughout the Figures.

For every statement S that V depends on, we consider all control dependency statements of S. Algorithm B.5 covers a common control dependency combination such as an if statement inside of a for loop, as well as for loops and if statements alone. As shown in Figure A.12, we also recursively lexicalize the conditions of the control dependency statement to ensure understanding of the conditions in which a dependency is covered.

Chapter 4

EVALUATION

The goal of my evaluation is to answer the following research questions:

- How helpful are the explanations generated by my approach?
- Do students with more coding experience tend to prefer less detailed explanations? (and vice-versa)
- Do students have a specific preference for API documentation or purely my own lexicalization technique in my approach?

Since expert programmers tend to understand how a program works more accurately and quickly than a novice [18], I evaluate the effect of the amount of experience on the preference for detail in order to understand the needs of a novice programmer. I have a secondary goal of identifying if students with varying levels of experience may have a preference for different explanations. I have another secondary goal of knowing if subjects have a preference of my approach with API documentation appended or my approach without API documentation (but with my own lexicalization technique). Since my approach can use both API Documentation and my own lexicalization technique, it could be useful to understand if API documentation is more helpful or if my own lexicalization technique is better without any API information.

4.1 Setup

4.1.1 Participants

For my subjects, I had 30 undergraduate students ranging from first-year students to seniors, from less than one year of experience of coding to five or more years. At minimum, students were expected to have taken a semester, an introductory course

in Python Programming or JavaScript and are taking or have taken a semester, introductory course to Java. No other experience was required for this study. I asked for the years of coding experience they had in general, coding experience in Java (because the snippets are in Java), as well as other classes they have taken or are taking.

4.1.2 Questions (for Subjects)

I had a set of 4 different questions coming from 3 different domain open-source projects. Two of the questions contained information from API documentation, and the other two did not contain any API information. Two of the questions had the same code snippet and bug, but one has API documentation and the other does not. Every subject was given a randomized set of any 2 different questions from the 4 questions. Every question consisted of a snippet of Java code; all snippets have one bug existing in one statement of the snippet.

4.1.3 Measures to Determine if my Approach is Preferred

Every question consisted of four explanations of the code; participants were asked to select the explanations that the subjects found useful for understanding the code and understanding how to solve the bug. Subjects could indicate that none of the explanations were useful or could have chosen any one or more of the explanations. Subjects were given a multiple choice question that asked to specify the statement with the bug.

One of the four explanations was generated by the approach of Sridhara, et al [5] for comparison. Three of the four explanations were generated by my own approach. The reason my approach generated more possible explanation choices than Sridhara [5] is due to my approach utilizing lower-level source code inter-dependencies within one selected statement. My first explanation is a higher-level explanation of a statement, which is of the same level as Sridhara. My next two choices explained the lower-level inter-dependent statements that supply the first explanation.

All explanations were for the closest statement to the bug. To determine the closest statement to the bug, I found the nearest statement that the bug depends on. I don't explain the statement that consists of the bug itself, since a programmer may not know the exact location of a bug in a real scenario. However, to prevent exhaustion of subjects, I chose a snippet of code nearby the bug, but also representative of an entire function (method) of a program.

Two of the questions used the same snippet of code and the same bug, however, my three explanations were different for each. For one of those two questions, I rely on API documentation. For the latter, I rely on my own approach, regardless of API Documentation availability. These choices help us analyze how useful API Documentation is compared to my manual explanations. There was no need to alter the explanation generated by Sridhara [5] since it does not use API Documentation, regardless of availability.

4.2 Results

According to the results of the Chi-Squared goodness of fit test, students have a statistically significant difference in preference between choosing my explanations and the explanations of Sridhara because $\tilde{\chi}^2 = 40.88$ where N = 75, df = 2, p <.0005. Additional post hoc goodness of fit tests were performed to understand the nature of these differences. Students preferred my hints to the Sridhara explanations. They preferred any explanation to no explanations, as shown in Table 4.1 and Figure 4.1. Comparing my explanations to none, $\tilde{\chi}^2 = 34.57$ where N = 56, df = 1, p <.0005. Comparing Sridhara to none, $\tilde{\chi}^2 = 6.76$ where N = 25, df = 1, p = .009, and experimental to Sridhara $\tilde{\chi}^2 = 13.93$ where N = 69, df = 1, p <.0005. On the other hand, there are three of my choices and only one Sridhara choice. Students do not prefer the experimental choice in a ratio that is greater than 75%:25%, $\tilde{\chi}^2 = 0.24$ where N = 69, df = 1, p = .63.

Next, I evaluated the number of each of my 3 explanations chosen, as well as the frequency of at least one of my explanations being chosen for each individual

Table 4.1: Which explanations do people prefer more?

Total	Sridhara	H	None
60	19	50	6

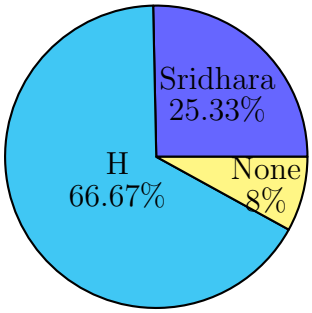


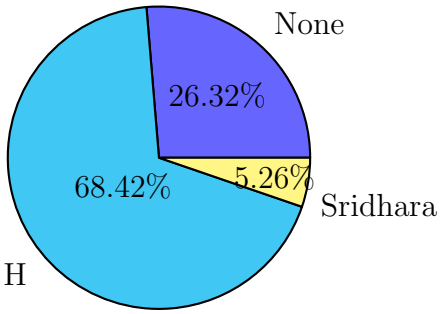
Figure 4.1: Total percentages of those who preferred explanations of Sridhara, et al., those who preferred at least one of our explanations per question "H", and those who preferred no explanations.

question. This could allow us to weigh the strengths and weaknesses of my algorithm and Sridhara, et al., as well as possibly observe effects from the specific code snippets. These results are displayed in table 4.2, Figures 4.2a, 4.2b, 4.3a, and 4.3b.

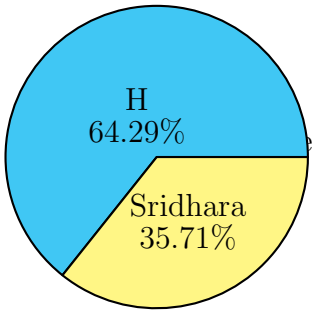
The next question was whether the subjects tend to find the bugs correctly if they chose my explanation or Sridhara. According to the results of the Chi-Squared test of independence, there was not a statistically significant difference in the percentage correct between Sridhara, et al. explanations (15.79%) and my explanations (28.00%), $\tilde{\chi}^2 = 1.11$ where N = 69, df = 1, p = .29. This means that correctly identifying the bug

Table 4.2: Which explanations do people prefer for each question? H1: Our 1st explanation, H2: Our 2nd explanation, and so on, H: At least one of our explanations chosen in one question.

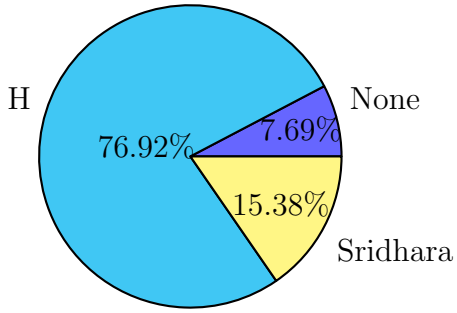
	Sridhara	H1	H2	H3	None	H
Bug A	1	12	3	2	5	13
Bug B	5	2	1	8	0	9
Bug C	2	6	4	4	1	10
Bug D	11	9	9	7	0	18



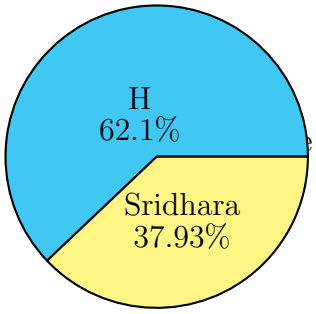
(a) Percentages for bug A (non-API supplemented), where "H" is at least one of our explanations chosen for questions.



(b) Percentages for bug B (non-API supplemented), where "H" is at least one of our explanations chosen for questions. Note that "none" is 0% (thus not showing above)



(a) Percentages for bug C (API supplemented explanations), where "H" is at least one of our explanations chosen for questions.



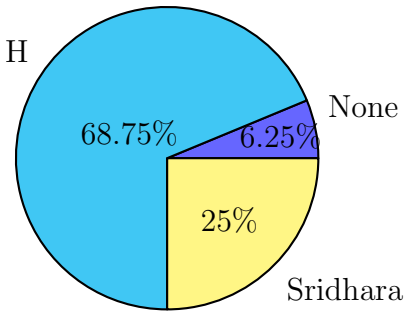
(b) Percentages for bug D (API supplemented), where "H" is at least one of our explanations chosen for questions.

had no significant impact on preference for explanations. Thus, whether they correctly identified the bug or not, the participants were more likely to prefer my explanations.

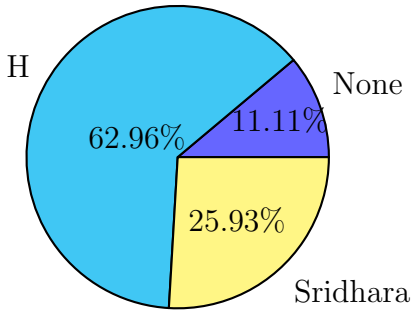
To determine if experience can have an effect on explanation preferences, all of my subjects were divided into 2 cases: participants who are taking algorithms and have taken Data Structures (expert case), and participants who are taking introduction to Java and have taken introduction to python (novice case). Do note that participants in the expert case have also taken introduction to Java and Python as well. The Novice and Expert explanation preference comparison is shown in table 4.3. The "None" category is omitted from this analysis because there are so few cases. There is no statistical significant difference in the degree to which novices (73.33%) and experts (70.83%) preferred my hints, $\tilde{\chi}^2 = 0.05$ where N = 69, df = 1, p = .83.

Table 4.3: Does experience have an effect on preference?

	Total	Sridhara	H	None
Novice	38	12	33	3
Expert	22	7	17	3



(a) Novice preference percentages



(b) Expert preference percentages

Next, I observe the difference of preference of explanations between the 2 questions that have the same code snippet and bug, but one uses API Documentation appended to part of my approach, and the other uses purely my own lexicalization technique. Results are shown in table 4.4. There was no statistically significant difference in the number selecting the API supplemented explanations and non-API pure lexicalized explanations, $\tilde{\chi}^2 = 1.29$ where N = 19, df = 1, p = .26 and $\tilde{\chi}^2 = 2.58$ where N = 50, df = 1, p = .11. However, more subjects chose Sridhara explanations for the Non-API version of my explanations, and there was only one less explanation of mine chosen in the non-API version. There was also one subject who chose that none of the explanations were useful for the API-version, while no subjects chose "None" in the non-API version.

I also observe the difference of preference between all API explanations and all

Table 4.4: API supplement or our pure lexicalization Preference

	Total	Sridhara	H	None
API Bug C	11	2	10	1
Non-API Bug B	11	5	9	0

Table 4.5: API supplement or our pure lexicalization Preference (all questions)

	Total	Sridhara	H	None
API Bugs C & D	31	13	28	1
Non-API Bugs A & B	29	6	22	5

Table 4.6: How consistent are participants? "1 & 2" means "chosen one explanation and at least one other of this approach." "Either Y" means they chose 1 other of this approach, "Both Y" means they chose 2 other of this approach, "No Y" means none was chosen.

	Either Y	Both Y	No Y
Giri=Y 1 & 2	15	4	15
H = Y 1 & 2	29	21	1
NA = Y 1 & 2	0	0	24

non-API explanations, regardless of it being the same code snippet and bug to further look for a possible specific preference between the two. Results are shown in table 4.5. We can come to a similar conclusion that there is no significant difference between the number of my explanations selected and Sridhara's explanations depending on whether API explanations were appended to my approach or not since $\tilde{\chi}^2 = 0.88$ where df = 1, N = 69, p = .35. However, this result contradicts table 4.4 in that Sridhara was chosen more in the amount in API bugs (41%) than non-API (20.7%), whereas Table 4.4 is nearly the opposite.

Finally, I observe the consistency of subject preferences. For instance, I can answer the following question: Does a subject who chose the Sridhara, et al explanation for the first question tend to choose Sridhara again for the 2nd question? The results are shown in table 4.6. Of the people who selected at least one Sridhara explanation, only 21% selected two Sridhara explanations and 42% of the people who selected at least one of my explanations selected two of my explanations. According to the chi-square test of independence, these percentages are not statistically significantly different since $\tilde{\chi}^2 = 2.62$ where N = 69, df = 1, p = .11.

4.3 Discussion

It is evident that students were much more likely to prefer my explanations over Sridhara's explanations, regardless of whether the students found the correct bug. Question C, which consists of my technique supplemented with API, appears to be my best result out of the other questions (highest percentage of my explanation chosen). It also meets the 25%:75% ratio to prevent the threat of validity of having 3 explanations to choose from while Sridhara only has 1. All 3 explanations in Question C were almost evenly chosen, with 6 choosing the 1st explanation, and 4 choosing the 2nd and 3rd explanations. The 1st explanation does not consist of API documentation, but my pure lexicalization technique with some lower-level detail. The 1st explanation performed slightly better than the 2nd and 3rd API explanations, but the difference is not significant.

Question B is the same code snippet and bug as question C, however, question B does not consist of API supplemented explanations. C performed somewhat better than B overall, mostly because more subjects preferred to choose Sridhara explanations in B and my explanation was chosen somewhat less in B, as shown in Figures 4.3a and 4.2b. Since there are no other varying factors between B and C, this result could mean that my approach performs better when supplemented with API Documentation. More subjects may prefer to choose Sridhara in B because my non-API explanations were less clear in comparison than my API-explanations. As a result, the less clear my explanations are, the more likely that participants will choose Sridhara. This shows that it is possible that it's better to utilize API documentation in addition to my approach whenever possible. However, the differences between these results were not statistically significant. This result shows potential for further analysis as future work, where there could be a bigger population dedicated to differentiating between non-API explanations and API supplemented explanations of my approach.

in Question A, 26.32% of subjects thought none of the explanations were useful, while only 0% thought none was useful for questions B and D, and only 7.69% for question C. The worst performance of Sridhara also took place in question A, with

only 5.26% choosing his approach, as opposed to 15.38% in question C and slightly above 35% in questions B and D. This result might indicate that the explanations in question A were the least helpful compared to the other questions. However, my approach still performs significantly better than Sridhara in Question A, especially if the "None" category is excluded where I would surpass the 25%:75% ratio (92% of my explanations versus 7.14% of Sridhara).

Question A consists of non-API supplemented explanations of my approach. It also utilizes inter-dependencies at least one method outside the snippet. A, in particular, consisted of multiple bitwise operators in single expressions, as shown in Figure A.5, for example. It is possible that this question had the most "none" useful explanation choices because bitwise operators were never taught in this school. Explanations 2 and 3 of A covered the bitwise operator in detail, while explanation 1 mentioned more high-level detail without explaining bitwise operators. 12 participants chose explanation 1, whereas only 3 chose explanation 2, and 2 chose explanation 3. This shows that my algorithm better explains high-level detail than less commonly used operators such as bitwise.

Sridhara, et al. had their best performance in D while I had my worst performance, although my approach still had a much higher percentage in D than Sridhara. D consists of explanations that also utilizes API Documentation. Considering that C (which also consists of API supplemented explanations) had my best performance, D slightly contradicts that API supplemented explanations are more preferred. Comparing Figures 4.3a and 4.3b, however, this contradiction may not be significant, as there is only a 14.82% difference between the percentage of chosen explanations of mine from C compared to D. The difference for Sridhara is slightly more significant with a difference of 22.55% but is still an overall insignificant difference from C.

Total percentages of non-API bug explanation preferences and API bug explanation preferences are shown in Figures 4.5 and 4.6. It would appear that I have the same percentage between the 2 Figures, however, there is a difference between the percentage of "None" and Sridhara chosen. Overall, there is no significant difference in preference

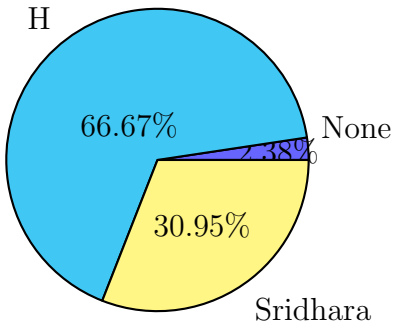


Figure 4.5: API supplement C & D Preference

between my API supplemented approach and my non-API supplemented approach. However, my non-API supplemented approach tends to have more "None" chosen, and my API supplemented approach tends to have more explanations of Sridhara chosen. This is likely an effect of how the explanations of Sridhara differs between the 2 questions. Question A consists of many complex expressions, and since complex expressions are unsupported by Sridhara, it is less likely that subjects will choose Sridhara in A. However, Sridhara likely supports simpler expressions in D. Thus, Sridhara performs better in pairs C & D (where D is supported) than A & B (where A is unsupported). Although A is unsupported, I allowed Sridhara's approach to explain the supported aspects of A (all other aspects such as variable assignment, variable name, and method name). I have chosen a randomized set of domains for picking code snippets to most emulate the chances of project encounters in the real world, and by chance, one of the four questions was not fully supported by Sridhara.

The level of experience among my subject population had little to no effect on the preference of explanation, as shown in Figures 4.4a and 4.4b. This result may be due to the fact that all subjects were undergraduate students. There is potential future work for comparing between undergraduate students and professionals in the field to better evaluate if there could be a significant difference, as it is evident in the study of Gugerty, et al.[18] that novice programmers have significantly slower and inferior comprehension of debugging programs compared to expert programmers. The gap in

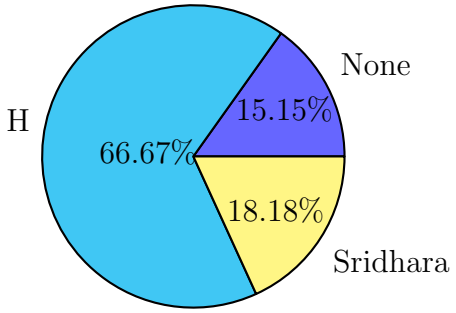


Figure 4.6: non-API supplement A & B Preference

experience between senior undergraduates and freshman undergraduates, for instance, may not be significant enough in comprehension of real-world programs.

Subjects were somewhat more consistent in picking 2 of my explanations in addition to picking my first explanation, as compared to Sridhara. This shows that it may be possible that the way my approach supplements lower-level explanations to higher-level explanations is more useful, however, it is not statistically significant. This shows potential for future studies to compare this with more participants, especially of a more varied experience level ranging from freshman undergraduates to professionals in the field.

4.4 Threats to Validity

Since my approach utilizes lower-level source code inter-dependencies within one selected statement, I had three possible choices of my approach while Sridhara only had one possible choice. This difference of the number of explanations is purely a difference in approaches. To accommodate for this difference, I considered the number of explanations of Sridhara chosen compared to only at least one of my explanations being chosen per question. For instance, even if all three of my explanations were chosen, that increases my count by only one, not three. The cases were also weighed equally, by considering whether or not I meet the 25%:75% ratio, to accommodate for Sridhara only having one possible choice. Despite this, the number of my explanations chosen were significantly greater.

Although I can add more choices of Sridhara, these choices will be irrelevant to the bug as it will be a higher-level approach of a line of code further away from the bug (it may not supply the bug in any way, or explain a line of code that's different from the bug). Even if I added 3 Sridhara explanations regardless (to test whether it is irrelevant to the bug or not), there will be 7 choices in total per question. I chose not to do this in order to prevent subject exhaustion. I already had a task of them having to read 2 snippets of code, find 2 bugs, and read a pair of explanation choices, in addition to entering demographic information. I always put Sridhara's choice first above all other choices in every question to help accommodate that Sridhara has only one choice.

There was also not a statistically significant difference between bugs correctly found and preference of choices. However, since there was a statistically significant difference between the preference of students between the approaches, this means that my approach was preferred in both cases of the correct and incorrect bug found. There was also no statistically significant difference between Novice and Expert preferences. However, the experience ranged from only freshman undergraduates to senior undergraduates. This is not as significant of a difference to, for example, a freshman undergraduate to an experienced professional in the field. In fact, this shows potential for future work where I compare my approach preference between experts in the field to student undergraduates. There were also only minor to no differences between preference for my approach supplemented with API Documentation or not. This result shows that my approach could work equally well in both cases. There is also more potential for future work for comparing only API documentation to my approach to better understand the differences.

Chapter 5

CONCLUSIONS & FUTURE WORK

This thesis presented an approach for summarizing any line of code for novice programmers. This approach was evaluated by undergraduate students of Computer Science at Drew University. Overall, my approach was found to be statistically significantly preferred over a competing approach [5]. The presented approach addresses common struggles of novice programmers to understand the following characteristics of programs: the connection between different lines of code; the dependence of lines of code on other lines; and the order of lines of code executed. These characteristics, called hierarchically structured & explicitly mapped [12], were found to be missing from novice programmers' understanding of code in comparison to experts [12]. Therefore, this thesis presented a method to auto-generate explanations of these characteristics to help novices understand any line of code written by other programmers.

In summary, this thesis makes the following contributions: (1) It evaluates current summarization approaches, and how it differs from the needs of novice programmers. (2) It proposes a novel approach toward summarization of any line of code that will help fulfill the needs of novice programmers. (3) It evaluates this approach through a study conducted within Computer Science classes at Drew University, to find whether undergraduate students of Computer Science preferred my approach over a competing approach by Sridhara, et al [5].

There are prior approaches that use a database of already made explanations [21, 29], but my approach does not require any information beyond the source code itself. This approach can, however, optionally augment existing API documentation to further supplement auto-generated explanations. Prior approaches utilize dependencies only within the same method in question [5, 10, 11, 20, 32], while my approach can

utilize dependencies outside the method, called inter-dependencies, to further fulfill the needs of novices as lines of code may depend on other lines of code outside of the method. This thesis approach also summarizes any line of code possible, while prior approaches are limited to only higher level lines of code such as methods and classes [5, 11, 22]. Prior approaches with evaluations of programmer subjects aim their study toward expert programmers [5, 8, 21, 31, 32], while the evaluation of this thesis was aimed toward undergraduate computer science students who are considered novice programmers. Finally, these prior approaches either utilize source code information alone or API documentation information alone [16, 15, 14], while the approach in this thesis can utilize both simultaneously.

5.1 Future Work

Including eye tracking studies within my evaluation might help identify more accurately the process of how novice programmers understand code. In addition, including expert programmers in the study and eye tracking for comparison might also be fruitful. There has been prior work that utilized eye tracking to better find details that expert programmers look for in code [8], however, this approach can lead to information useful to only expert programmers, so eye tracking for novice programmers should also be taken into account.

While the evaluation in this thesis included novice programmers as undergraduate students, it did not include novice industry programmers, or novice programmers entering the industry recently. Having industry novice programmers in this study could be useful to compare whether academic environments could have an effect in comparison to industry environments on novices, as well as comparing those who recently graduated to those who are still undergraduates to determine if specific coursework taken recently can have a significant effect.

Although undergraduates preferred the explanations generated by the thesis approach, the explanations of this approach are significantly lengthier in comparison to

the competing approach [5]. It might be useful to determine if having shorter explanations is possible and if the length may have a positive or negative effect on novice programmers, and how or why experts may or may not prefer lengthier explanations. Finding methods to better present lengthy explanations shows potential for an interactive automated tutoring application which could provide information requested through user input (E.g. Clicking a "More info on..." link) to help prevent the display of unnecessary information. Having experts and novices identify what information may be extraneous in these lengthy explanations could also be important information to better understand what is required for program comprehension.

For future work, we can augment my explanations with informal online resources such as StackOverflow (a popular question-answer forum for programmers), code reviews, online posts, video tutorials that may describe functions, methods of source code. Currently, this thesis approach only augments explanations from API documentation but a variety of other useful online resources exist. Many existing approaches are dedicated to utilizing information from informal documentation [17, 23, 24, 25, 26, 27, 28, 30]. For example, Treude, et al. developed an approach towards augmenting API documentation with stack overflow information by utilizing the Stack Overflow API and the search query of the type name found from a statement. Their approach is a machine learning algorithm that considers POS tags, code within stack overflow answers, position of sentence, position of API type name in the sentence, reputation, whether the API type name is in the title, answer score, and whether it was the accepted solution. Then, they tested the similarity of the answer to the API Documentation description to determine the relevance of the stack overflow answer [17]. Lastly, for future work, we should ask questions related to comprehension of code rather than identifying bugs, because there was no statistical significance on whether correctly identifying the bug affected the preference for explanations.

Currently, the approach proposed in this thesis is being developed into a form of an Eclipse IDE plugin. The Eclipse IDE is an advanced Java programming editor, and the plugin would give explanations of any line of code upon a mouse hover (see

Figure A.2).

Appendix A

FIGURES

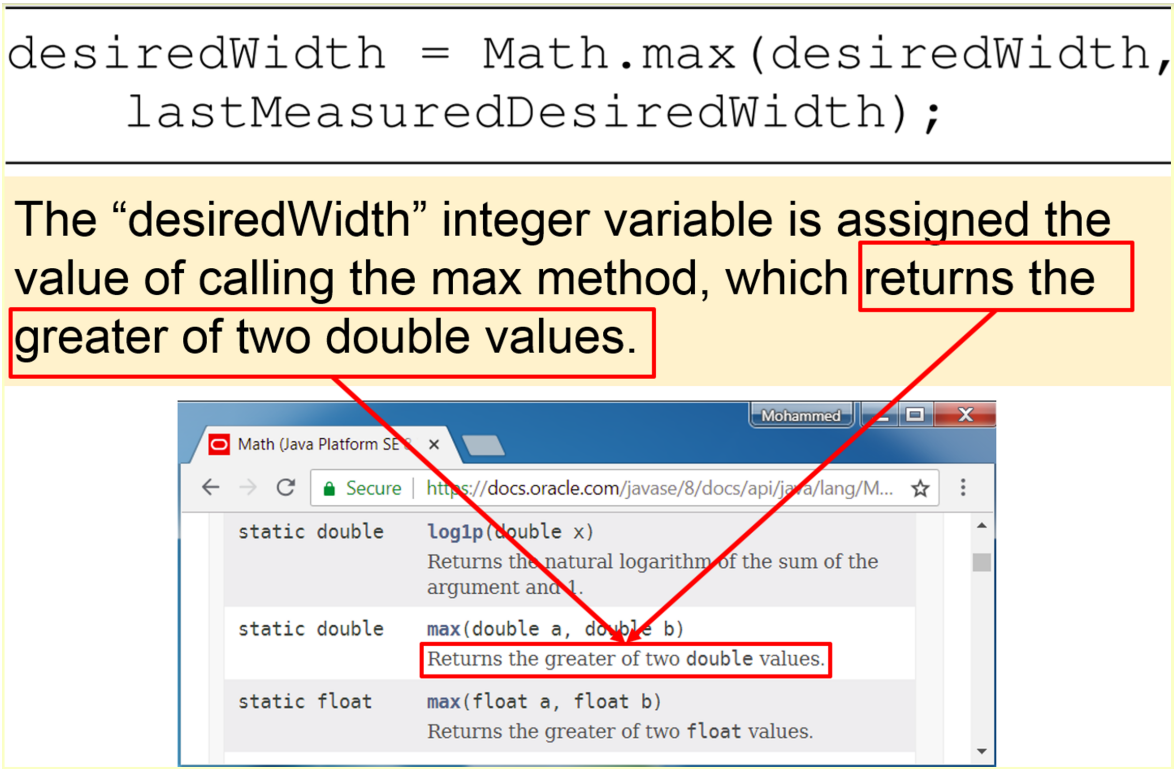


Figure A.1: Example of this thesis approach optionally utilizing available API

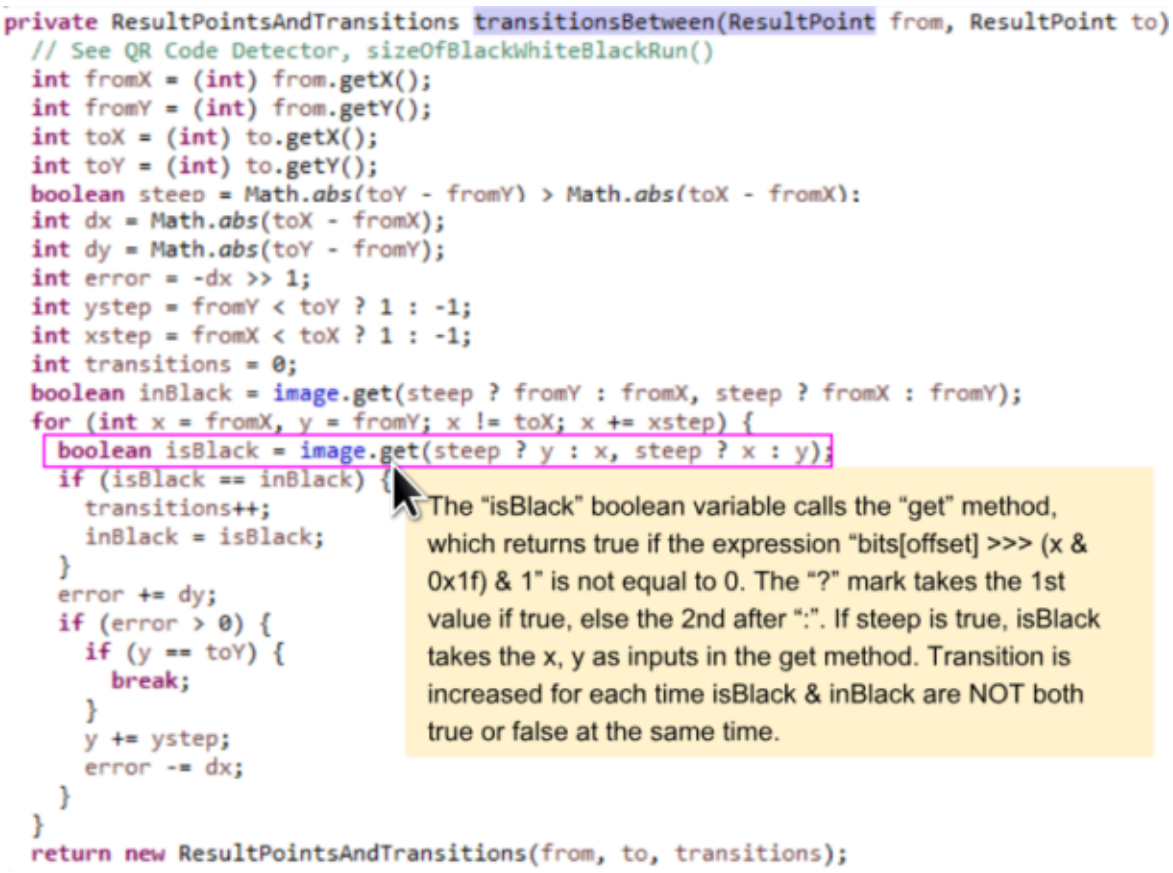


Figure A.2: Example of the proposed approach

```
protected void onMeasure(int widthMeasureSpec, int heightMeasureSpec) {
       final int widthMode = MeasureSpec.getMode(widthMeasureSpec);
       int desiredWidth = MeasureSpec.getSize(widthMeasureSpec);
       lastMeasuredDesiredWidth = computeDesiredWidth();
       switch (widthMode) {
           case MeasureSpec.EXACTLY:
               break;
           case MeasureSpec.AT_MOST:
               desiredWidth = Math.min(desiredWidth,
                   lastMeasuredDesiredWidth);
               break;
           case MeasureSpec.UNSPECIFIED:
               desiredWidth = lastMeasuredDesiredWidth;
               break;
       }
```

Figure A.3: Ordering of variable assignments lexicalized: desiredWidth (3 assignments, 1 input parameter), lastMeasuredDesiredWidth (1 assignment, 1 input parameter, used for desiredWidth assignment), widthMode (1 assignment)

```
private Map<Element, List<Element>> getElementsAnnotatedOrMetaAnnotatedWith(
        RoundEnvironment roundEnv, TypeElement annotation) {
       Map<Element, List<Element>> result = new LinkedHashMap<>();
     for (Element element : roundEnv.getRootElements()) {        //***
        LinkedList<Element> stack = new LinkedList<>();
        stack.push(element);                                      //***
        if (!stack.isEmpty()) {                                   //***
           result.put(element, Collections.unmodifiableList(stack)); //***
        }
     }
     return result;
   }
```

Figure A.4: An example of hard-to-notice dependencies: To explain a non-void method, we start with explaining the return statement. However, for a novice, the name "result" alone may not be enough. For example, what if there was a bug? Then a novice would have to understand how "result" was created. Thus, we consider the lines labeled with *** which supplies "result." For instance, a novice could miss the detail of how "stack" is created and used with the for loop, and stack is appended as a part of result. Control dependencies like ifs and fors are also considered. We lexicalize statements labeled with ***, and lexicalization is shown in our other algorithms.

```
public boolean get(int x, int y) {
    int offset = y * rowSize + (x / 32);
    return ((bits[offset] >>> (x & 0x1f)) & 1) != 0;
  }
```

Figure A.5: Our explanation generates the following for a statement level summary of the return statement: "Bits[offset]" represents a value located in the "offset" index in the Bits list. ">>>" will shift the value of "bits[offset]" to the right by the value of (x & 0x1f). Offset is the value of "y * rowSize + (x >> 5)" where >> pushes the binary form of x to the right by 5. y is 1st input, x is 2nd input. (x & 0x1f) represents the binary number where each 0 and 1 of x is compared to the same index of the 0 and 1 of 0x1f, where 1 & 1 is true, 1 & 0 is false, 0 & 0 is false for each spot.

```
x < 5
```

Figure A.6: Our explanation generates: x is less than 5. "x" is rhs, "5" is lhs.

```
isBlack = (a boolean value)
inBlack = (a boolean value)
if (isBlack != inBlack) {
       transitions++;
     }
```

Figure A.7: In the expression "isBlack != inBlack", "isBlack" is lhs and "inBlack" is rhs. Our explanation generates: Transition is increased for each time isBlack & inBlack are NOT both true or false at the same time.

```
public static int max(a, b) {
    return (a>=b)?a:b
}
```

Figure A.8: Our explanation generates: The max method returns "a" if "a >= b" is true. The "?" mark takes the 1st value if true, else the 2nd after ":". If "a >= b" is false, it takes "b".

```
desiredWidth = Math.max(desiredWidth, lastMeasuredDesiredWidth);
```

Figure A.9: Our approach generates: The "desiredWidth" integer variable is assigned the value of calling the max method, which returns the greater of two double values.

```
*if (steep) {
      int temp = fromX;
      fromX = fromY;
      fromY = temp;
      temp = toX;
      toX = toY;
      toY = temp;
    }

    int dx = Math.abs(toX - fromX);
    int dy = Math.abs(toY - fromY);
    int error = -dx >> 1;
    int ystep = fromY < toY ? 1 : -1;
    int xstep = fromX < toX ? 1 : -1;
    int transitions = 0;
    boolean inBlack = image.get(steep ? fromY : fromX, steep ? fromX :
        fromY);
*    for (int x = fromX, y = fromY; x != toX; x += xstep) {
***      boolean isBlack = image.get(steep ? y : x, steep ? x : y);
      if (isBlack != inBlack) {
        transitions++;
        inBlack = isBlack;
      }
```

Figure A.10: Our approach generates: "isBlack is assigned the value of calling the "get" method, which returns true if the expression "bits[offset] >>> (x & 0x1f) & 1" is not equal to 0. The "?" mark takes the 1st value if true, else the 2nd after ":". If steep is true, isBlack takes the x, y as inputs in the get method." Remainder of explanation is generated and shown in figure A.5. ***is line being summarized, *is a control block dependency.

```
return new ResultPointsAndTransitions(from, to, transitions);
```

Figure A.11: Our explanation generates: "Returns a new ResultPointsAndTransitions given the 1st resultpoint "from", the 2nd resultpoint "to", & a calculated "transitions" integer variable." Then, "transitions" is explained by our approach automatically, as shown in Figure A.7. The same process is done on "from" & "to".

```
for (int x = fromX, y = fromY; x != toX; x += xstep) {
  boolean isBlack = image.get(steep ? y : x, steep ? x : y);
  if (isBlack != inBlack) {
    transitions++;
    inBlack = isBlack;
  }
```

Figure A.12: Our approach generates: Transition is increased for each time isBlack & inBlack are NOT both true or false at the same time.

Appendix B

ALGORITHMS

Algorithm B.1 Process Method

```
function summarizeMethod(M)
  output ←″
  list ← [] n
  if M is void then
    list ← set of variable assignments ∈ M,
                sorted by most frequent var, last occurrence first
  else
    list ← set of return statements ∈ M
  end if
  for each s ∈ list do
    output ← output + lexicalize(S, MD)
  end for
  return output
end function
```

Algorithm B.2 Lexicalize Expression

```
function LEXICALIZE(S, MD)
  S ←Statement Input
  MD ←Method Declaration Input
  explain ←Output String
  switch s do
    case s is method or constructor call
      explain += SUMMARIZEMETHODCALL(S)
    case s is a variable
      explain += SUMMARIZEVARIABLEDEPENDENCE(S, MD)
    case s is a Complex expression
      explain +=SUMMARIZEEXPRESSION(S,MD)
    case s is an Enum
      explain += SUMMARIZEENUM(S, MD)
    case s is a literal parameter
      explain += "the " + type(s) + s
    case s is literal
      explain += s
    case s is a list
      explain += (
      s + "represents a value located in the "
      + getSquareBracketNum(s) + " index in the "
      + getListName(s) + "list".
      )
      seen.add(getSquareBracketNum(s))
      ComplexParameterFollow += (
      LEXICALIZE(getSquareBracketNum(S), MD)
      )
  return explain
end function
```

Algorithm B.3 Summarizing Complex Expressions

```
function summarizeExpression(E, MD)
  E ←Expression
  MD ←Method Declaration
  switch E do
    case e is return statement
      return "returns"+lexicalize(stripReturn(E))
    case e has comparator
      explain += summarizeComparator(E)
    case e has ++
      explain += is increased
    case e has −−
      explain += is decreased
    case e has −−
      explain += is decreased
    case e increased or decreased by more than 1 or non-literal only
      subExpressions ← []
      for each exp ∈ E do
                                          ▷ exp ∈ subexpressions between operators
        subExpressions.append(lexicalize(exp))
      end for
      explain += (
       "the following integer values combined: "
      + each subExpression
       )
    case e has short hand if
      explain += ShortHandTable(short-If)
end function
```

Algorithm B.4 Process Method Calls and Constructors

```
function summarizeMethodCall(M)
  Output ←"
  follow ←[]
  if constructor(M) then
    output += a new + name(M)
  end if
  if methodcall(M) then
    output += calling the+name(m)+method, which
  end if
  if hasParameters(M) and NOT Built-in then
    if hasVariableParameters(M) then
      output += given
    end if
    num ← 0
    for each parameter P in M do
      seen.add(parameter)
      if isLiteral(P) then
        output += the + type(P) + name(P)
      end if
      if isVariable(P) then
        follow +=P
        if type(P) exists multiple times then
          output += ordinal(++num) //1st,etc
        end if
        if type(P) is int, float, or double then
          output += calculated
          + name(parameter) + type(parameter)
          + variable
        else
          output += name(parameter)
          + type(parameter)
        end if
        follow += parameter
        if parameter not last and not 2nd last then
          output += " ,"
        end if
        if parameter is 2nd last then
          output += ", and"
        else
          output += "."
        end if
      end if
    end for each
  end if
  if method M is built-in then
    output += return section of API Doc
  else
    output += Algorithm B.1 analysis of M
  end if
```

Algorithm B.5 Summarizing Variable Data and Control Dependencies

function SUMMARIZEVARIABLEDEPENDENCE(V, MD)
$V \leftarrow$Variable
$MD \leftarrow$Method Declaration
$explain \leftarrow$"
$follow \leftarrow$"
while v is data dependent on assignment or append statement $x \in md$ (reverse order of appearance) **do**
$explain$ += variableName(v)
if v is append statement **then**
$follow$ += v represents the following: +
lexicalize(getValue(v), md)
else
$follow$ += is assigned the value of: +
lexicalize(rhs(x), md)
end if
for each control statement c that x depends on **do**
if $c \notin seen$ **then**
$seen.add(c)$
if c is an if Statement **then**
if c is within on loop condition **then**
$explain$ += for each time
end if
$explain$ +=lexicalize(ifStatementCond(c), md)
end if
if c is a loop statement **then**
if c is a short-loop statement **then**
$explain$ += every lhs(varName(c)) in the +
rhs(varName(c)) rhs(varType(c))
if $(lhs(varName(c))) \notin seen$ **then**
$explain$ +=lhs(varName(c)) +
represents: lexicalize(lhs(varName(c)), md)
end if//Repeat above for rhs as well
end if
end if
end if
end for each
$explain$ += follow
$v \leftarrow x$
end while
If not dependent on any assignment statement: return varName(v) + most recently used method it is assigned in
return ($explain$)
end function

Appendix C

TABLES

Table C.1: Comparator Summaries (Literals, single variables)

Symbol	Explanation
Always append	$lhs(e)$ is
<	less than
<=	less than or equal to
==	is equal to
>	greater than
! =	is not equal to
>=	greater than or equal to
Always append at end	$rhs(e)$

Table C.2: Comparator Summaries (Complex Expressions)

Symbol	Explanation
Always append	The expression $lhs(e)$ is
<	less than
<=	less than or equal to
==	is equal to
>	greater than
! =	is not equal to
>=	greater than or equal to
Always append at end	the expression $rhs(e)$

Table C.3: Comparator Summaries (2 variables or booleans)

Symbol	Explanation
Always append in beginning	$lhs(e)$ and $rhs(e)$ are
==	both
! =	not both
Always append at end (booleans)	true or false at the same time
Always append at end (variables)	equal to each other

Table C.4: BitWise Summaries

Symbol	Explanation
$<<<$	$<<<$ will shift the value of + $lhs(e)$ + to the left by the value of + $rhs(e)$
$>>>$	$>>>$ will shift the value of + $lhs(e)$ + to the right by the value of + $rhs(e)$

Table C.5: ShortHand Summaries

Symbol	Explanation
!	is not (expression method call noun/adj or boolean literal)
$++$	(variable name) is increased
$--$	(variable name) is decreased
$Short - If$	getTrueConditions(e) if getCondition(parameter) is true. The ? mark takes the 1st value if true, else the 2nd after :. If getCondition(parameter) is false, it takes getFalseConditions(e)

Appendix D

GLOSSARY

- API-Documentation: Documentation for Built-in methods and classes.
- Binary: Numbers expressed as bits in 1's or 0's with base 2.
- Bitwise signed left-shift operator (<<): Shifts the 0's and 1's of a binary number to the left, which as a result, decreases numbers by powers of 2.
- Bitwise signed right shift operator (>>): Shifts the 0's and 1's of a binary number to the right, which as a result, increases numbers by powers of 2.
- Bitwise unsigned left-shift operator (<<<): Similar to signed but is a strictly positive number.
- Bitwise unsigned right shift operator (>>>): Similar to signed but is a strictly positive number.
- Built-in: Anything that is included within the coding language. This could be methods or classes that can be utilized without the need to be programmed by users, as it already exists as part of the language itself. All built-in methods and classes have API-Documentation.
- Character: A letter, which can include symbols and numbers. Note that mathematical operations can not be done on character numbers. For instance, adding characters is string concatenation, not arithmetic.
- Class: Contains a collection of methods and statements for any specified purpose. Every object is made from a "class." For instance, a float object has a float class, which contains built-in methods such as a round method, which returns an integer that is rounded to a specified decimal place. User-made classes can contain user-made methods specifically designed for the class.
- Constructor: Specifications for how an object of a class can be instantiated in code. For instance, a "Person" object can have a height, age, and weight, all of which are specified in the constructor can be inputted later in a statement. See Figure A.11 for an example of returning an expression that instantiates an object through its constructor specifications.

- Control-Dependency: Lines of code can determine if other "control-dependent" lines of code should be executed, as well as the number of times of execution. If statements, For loops, and While loops are examples of Control Dependencies.
- Complex Expressions: Expressions containing two or more literals, objects, and so on (see Figure 1.5 for an example).
- Data-Dependency: Statement(s) that utilizes variable(s), which depend on corresponding variable assignment(s) (see Figure A.4 for an example). If a variable is assigned to other variables in prior lines of code, then that variable is "data-dependent" on other lines of code.
- Double: Decimal Numbers.
- Enumerator: List of constants all specified with an object type. Enumerators are defined as their own class while lists are defined as statements within methods.
- Equals Comparator ("=="): Checks whether 2 numbers are equal or not. It has two equal signs adjacent to each other because a single equal sign is interpreted to be a variable assignment.
- Expression: A collection of objects as part of one line of code. Can contain one or more mathematical operations, such as add (+), multiply (*), divide (/), subtract (-), and so on. Method calls that return an object can also be part of an expression.
- Float: Decimal Numbers.
- For-loops: A control dependency that allows for the execution of the same set of lines of code for a specified amount times. See Figure A.12 for an example of an if statement and for loop statement.
- If-statements: A control dependency where a certain condition stated within the if-statement must be true to execute another line of code. See Figure A.12 for an example of an if statement and for loop statement.
- Integer or int: Whole Numbers.
- Inter-Dependency: Any dependency of lines of code that is located outside of the method that line of code is in. For instance, A method call can be located inside of a method, but it depends on its own method declaration which is outside of the method that the method call is located in (see Figure 1.6).
- Literals: Any value or data in code that is utilized directly without a variable (without the need to find data from computer memory). For example, 2 is an integer literal and "Hello" is a string literal. The integer variable "x" of value 2 is NOT literal, because it must reference some location in computer memory to obtain the value of 2.

- Method Call: Refers to lines of code that is wrapped inside a method declaration of the same name, the same number of parameters, and the same type parameters. See Figure 1.6 for an example of a variable being assigned to a method call and a method call referring to its declaration.

- Method Declaration: Contains lines of code that can later be called without having to repeat those lines of code again.

- Method (non-void): Method declarations that have return statements, which returns the value of an object or type specified in the method declaration. For instance, a method can return an integer, string, and so on (see Figure A.4).

- Method (void): Method declarations that lack return statements and do not return any value. It is not practical to assign variables to void methods or to have void methods as part of any expression, as it does not return any value. The purpose of a void method is to have a side effect of some kind. For instance, a void method might have the purpose of printing out information to the screen in a specific format, can call other methods, and/or reassign variables (see Figure A.3).

- Objects: A data type that is either made by a programmer or already built into the programming language. For example, String is an object that is built-in to the Java programming language and as a result, has API documentation because it is built-in.

- Parameter: Inputs within a method, which are then utilized for lines of code that depend on the value of these inputs within the method declaration. These inputs or parameters can be any expression as long as it obeys the correct type of object specified in the parameter. For instance, a parameter can be declared as an integer, and any expression that ultimately calculates into an integer can be inputted as that parameter. If it is a float or double, it must be rounded to an integer in some way or else an error will occur.

- Print Statements: Output any specified information to the computer screen.

- Recursion: Call a method inside of the same method itself repeatedly, until a certain condition specified is met. This is the only time a method call's declaration is inside the same declaration (non-inter-dependent). For instance, the method can call itself inside an if-statement until the if-statement is false.

- String: A list of characters together, which can be Words, phrases, or sentences (that can be English sentences) which can also include numbers, shown in code between quotes. Note that mathematical operations cannot be done on string numbers. For instance, adding strings is string concatenation, not arithmetic.

- String Concatenation: Combine strings by putting them adjacent to each other. For example: “Hello” + “There” is equivalent to “HelloThere”.
- Variable assignment: Assign any literal, object, or any kind of expression to a certain part in the computer memory which can be later referred to for any future statements. A variable has a name, and its name can be referred to in any expression to be utilized from memory. With variable assignments, values can be organized into a name, and the value does not have to be repeated, recalculated, or restated in code to be utilized repeatedly. Variables can be assigned to an expression which could contain a collection of literals of any objects, mathematical operators, comparators, variables, values returned by method calls, and many more.
- While-loop: A control dependency that executes the same set of lines of code until a certain condition specified is no longer met.

BIBLIOGRAPHY

[1] L. Gugerty, G. Olson, Debugging by skilled and novice programmers, CHI 1986.

[2] S. C. B. de Souza, N. Anquetil, and K. M. de Oliveira. A Study of the Documentation Essential to Software Maintenance. Intl Conf on Design of Communication (SIGDOC), 2005

[3] M. Kajko-Mattsson. A Survey of Documentation Practice within Corrective Maintenance. Empirical Softw. Eng., 10(1):3155, 2005.

[4] Wyatt Olney, Emily Hill, Chris Thurber, Bezalem Lemma. Part of Speech Tagging Java Method Names. Proceedings of the 32nd IEEE International Conference on Software Maintenance, Early Research Achievements (ERA) Track, 2016.

[5] G. Sridhara E. Hill D. Muppaneni L. Pollock and K. Vijay-Shanker. *Towards Automatically Generating Summary Comments for Java Methods*. In 25th IEEE/ACM International Conference on Automated Software Engineering 2010 pp. 43-52.

[6] P.W. McBurney and C. McMillan. *Automatic documentation generation via source code summarization of method context*. Proceedings of the International Conference on Program Comprehension (ICPC) pp. 279-290 2014.

[7] Giriprasad Sridhara, Lori Pollock, K. Vijay-Shanker, *Automatically detecting and describing high level actions within methods*. Proceedings of the 33rd International Conference on Software Engineering, May 21-28, 2011, Waikiki, Honolulu, HI, USA

[8] P. Rodeghero C. McMillan P. W. McBurney N. Bosch S. D'Mello *Improving Automated Source Code Summarization via an Eye-Tracking Study of Programmers*. Proceedings of the 36th International Conference on Software Engineering ser. ICSE 2014 pp. 390-401 2014.

[9] Paige Rodeghero, Collin McMillan, and Abigail Shirey. 2017. *API usage in descriptions of source code functionality.* In Proceedings of the 1st International Workshop on API Usage and Evolution (WAPI '17). IEEE Press, Piscataway, NJ, USA, 3-6.

[10] Giriprasad Sridhara, Lori Pollock, K. Vijay-Shanker, *Generating Parameter Comments and Integrating with Method Summaries*, Program Comprehension (ICPC) 2011 IEEE 19th International Conference on, pp. 71-80, 2011, ISSN 1092-8138.

[11] L. Moreno, J. Aponte, G. Sridhara, A. Marcus, L. Pollock, V. Shanker, *Automatic Generation of Natural Language Summaries for Java Classes*, Proceedings of 21st International Conference on Program Comprehension ICPC'13t, pp. 23-32, 2013.

[12] Fix, V., Wiedenbeck, S., and Scholtz, J. 1993. *Mental representations of programs by novices and experts*. In Proceedings of the INTERACT '93 and CHI '93 Conference on Human Factors in Computing Systems (CHI'93). ACM, New York, NY, USA, 74-79.

[13] Steven S. Muchnick. 1998. *Advanced Compiler Design and Implementation*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

[14] U. Dekel and J. Herbsleb. *Improving api documentation usability with knowledge pushing*. In Software Engineering, 2009. ICSE 2009. IEEE 31st International Conference on, pages 320 330, may 2009.

[15] R. Pandita, X. Xiao, H. Zhong, T. Xie, S. Oney, and A. Paradkar, *Inferring method specifications from natural language API descriptions*, in ASE, 2012.

[16] Martin P. Robillard , Yam B. Chhetri, *Recommending reference API documentation*, Empirical Software Engineering, v.20 n.6, p.1558-1586, December 2015

[17] Christoph Treude , Martin P. Robillard, *Augmenting API documentation with insights from stack overflow*, Proceedings of the 38th International Conference on Software Engineering, May 14-22, 2016, Austin, Texas

[18] L. Gugerty , G. Olson, *Debugging by skilled and novice programmers*, Proceedings of the SIGCHI Conference on Human Factors in Computing Systems, p.171-174, April 13-17, 1986, Boston, Massachusetts, USA

[19] Barthlmy Dagenais , Martin P. Robillard, *Creating and evolving developer documentation: understanding the decisions of open source contributors*, Proceedings of the eighteenth ACM SIGSOFT international symposium on Foundations of software engineering, November 07-11, 2010, Santa Fe, New Mexico, USA

[20] Paul W. McBurney, *Automatic documentation generation via source code summarization*, Proceedings of the 37th International Conference on Software Engineering, May 16-24, 2015, Florence, Italy

[21] Annie T. T. Ying , Martin P. Robillard, *Code fragment summarization*, Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering, August 18-26, 2013, Saint Petersburg, Russia

[22] Sonia Haiduc, Jairo Aponte, Andrian Marcus, *Supporting program comprehension with source code summarization*, Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering, May 01-08, 2010, Cape Town, South Africa

[23] Latifa Guerrouj, David Bourque, Peter C. Rigby, *Leveraging informal documentation to summarize classes and methods in context*, Proceedings of the 37th International Conference on Software Engineering, May 16-24, 2015, Florence, Italy

[24] Peter C. Rigby, Martin P. Robillard, *Discovering essential code elements in informal documentation*, Proceedings of the 2013 International Conference on Software Engineering, May 18-26, 2013, San Francisco, CA, USA

[25] Preetha Chatterjee, Benjamin Gause, Hunter Hedinger, Lori Pollock, *Extracting code segments and their descriptions from research articles*, Proceedings of the 14th International Conference on Mining Software Repositories, May 20-28, 2017, Buenos Aires, Argentina

[26] S. Panichella, J. Aponte, M. D. Penta, A. Marcus, and G. Canfora, *Mining source code descriptions from developer communications*, in Program Comprehension (ICPC), 2012 IEEE 20th International Conference on, June 2012, pp. 63–72.

[27] Siddharth Subramanian, Laura Inozemtseva, Reid Holmes, *Live API documentation*, Proceedings of the 36th International Conference on Software Engineering, May 31-June 07, 2014, Hyderabad, India

[28] Luca Ponzanelli, Gabriele Bavota, Andrea Mocci, Massimiliano Di Penta, Rocco Oliveto, Mir Hasan, Barbara Russo, Sonia Haiduc, Michele Lanza, Too long; didn't watch!: extracting relevant fragments from software development video tutorials, Proceedings of the 38th International Conference on Software Engineering, May 14-22, 2016, Austin, Texas

[29] Najam Nazar, He Jiang, Guojun Gao, Tao Zhang, Xiaochen Li, Zhilei Ren, *Source code fragment summarization with small-scale crowdsourcing based features*, Frontiers of Computer Science: Selected Publications from Chinese Universities, v.10 n.3, p.504-517, June 2016

[30] Jairo Aponte, Andrian Marcus, *Improving traceability link recovery methods through software artifact summarization*, Proceedings of the 6th International Workshop on Traceability in Emerging Forms of Software Engineering, May 23-23, 2011, Waikiki, Honolulu, HI, USA

[31] Yusuke Oda, Hiroyuki Fudaba, Graham Neubig, Hideaki Hata, Sakriani Sakti, Tomoki Toda, Satoshi Nakamura, *Learning to Generate Pseudo-Code from Source Code Using Statistical Machine Translation*, Automated Software Engineering (ASE) 2015 30th IEEE/ACM International Conference on, pp. 574-584, 2015.

[32] Xiaoran Wang, Lori Pollock, K. Vijay-Shanker, "Automatically generating natural language descriptions for object-related statement sequences", Software Analysis Evolution and Reengineering (SANER) 2017 IEEE 24th International Conference on, pp. 205-216, 2017.

[33] M. P. Robillard, A. Marcus, C. Treude, G. Bavota, O. Chaparro, N. Ernst, M. Aurelio Gerosa, M. Godfrey, M. Lanza, M. Linares-Vasquez, G. C. Murphy, L. Moreno, D. Shepherd, E. Wong, *On-Demand Developer Documentation*, ICSME 2017.

[34] S. Iyer, I. Konstas, A. Cheung, L. Zettlemoyer. *Summarizing Source Code using a Neural Attention Model*, ACL 2016.

[35] X. Hu, G. Li, X. Xia, D. Lo, Z. Jin. *Deep Code Comment Generation*, ICPC 2018.

[36] Fowkes, J., Chanthirasegaran, P., Ranca, R., Allamanis, M., Lapata, M., Sutton, C. (2016, May). *TASSAL: Autofolding for source code summarization*, In 2016 IEEE/ACM 38th International Conference on Software Engineering Companion (ICSE-C) (pp. 649-652). IEEE.

[37] Chen, Q., Zhou, M. (2018, September). *A neural framework for retrieval and summarization of source code*, In Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering (pp. 826-831). ACM.

[38] Nielebock, S., Krolikowski, D., Krger, J., Leich, T., Ortmeier, F. (2018). *Commenting source code: is it worth it for small programming tasks?*, Empirical Software Engineering, 1-40.

[39] Newman, C., Dragan, N., Collard, M. L., Maletic, J., Decker, M., Guarnera, D., Abid, N. (2018, September). *Automatically Generating Natural Language Documentation for Methods*, In 2018 IEEE Third International Workshop on Dynamic Software Documentation (DySDoc3) (pp. 1-2). IEEE.