Toward Automatic Summarization of Arbitrary Java Statements for Novice Programmers

Mohammed Hassan, Emily Hill
Drew University
Madison, NJ 07940
Email: {mhassan, emhill}@drew.edu

***Abstract*—Novice programmers sometimes need to understand code written by others. Unfortunately, most software projects lack comments suitable for novices. The lack of comments have been addressed through automated techniques of generating comments based on program statements. However, these techniques lacked the context of how these statements function since they were aimed toward experienced programmers. In this paper, we present a novel technique towards automatically generating comments for Java statements suitable for novice programmers. Our 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, we observed explanations based on our technique to be preferred over an existing approach.**

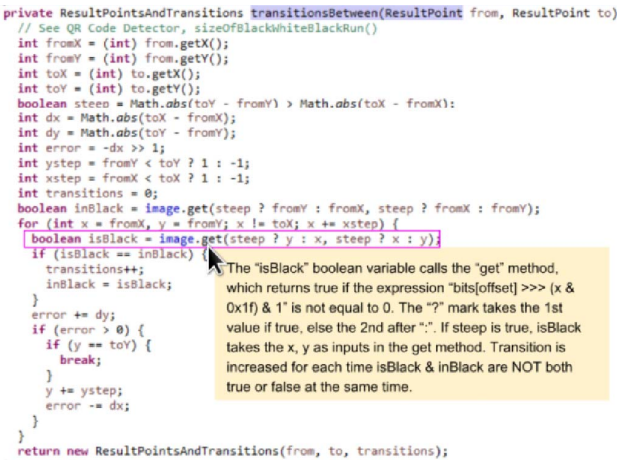

Figure 1: Example of the proposed approach

1. Introduction and Background

Novice programmers take roughly twice as long as experts to understand a program when debugging [4]. One reason is that novice programmers do not have the mental representations of programs that experts exhibit [2]. Although both novices and experts can make a hypothesis of code's purpose, novices can be unable to relate the source code to their own high level hypotheses. Expert programmers, unlike novices, concentrate on how parts of code interact, such as method calls and data dependencies [2].

In this paper, we present a novel technique towards automatically generating comments for Java statements on-demand [13], suitable for novices that seeks to address both (1) relating high-level explanations to the particular areas of the source code that implement them, and (2) communicating how parts of the code interact through control and data dependences. Our technique not only goes beyond existing approaches to method summarization to meet the needs of novices, it also leverages API documentation when available.

Prior approaches have utilized linguistic information in source code to generate summaries for classes [9], methods [5], [15], and parameters [17]. Some have also focused on the statement level, but using complimentary sources of information such as a corpus of code fragments [20], crowdsourced annotated statements gathered from stack overflow [7], ASTs [6], or corpora specially created by hand for this purpose [10], [11], and then applying machine learning techniques to generate summaries.

Our technique is inspired by identifier-based approaches that leverage the natural language found in source code identifiers to generate statement-level summaries without the need for a large training corpus [16], [19]. We go beyond these approaches by (1) targeting our summaries to novice programmers rather than experts; (2) recursively adding information to our summaries, potentially from many nested calls or def-use chains; and (3) pulling in relevant API documentation when available. Other researchers have looked into extracting or highlighting relevant portions of API documentation for comprehension [1], [3], [12], [18], but to our knowledge none have combined this technique to improve arbitrary statement comprehension.

The paper is organized as follows. In Section 2 we present our approach followed by an evaluation comparing our technique to Sridhara et al.'s [16] in Section 3. Finally, we conclude and discuss future work in Section 4.

2. Summarization of Arbitrary Statements

A novice programmer could be confused about what a specific statement does or how it works. Our approach aims

2576-3148/18/$31.00 ©2018 IEEE
DOI 10.1109/ICSME.2018.00063

```
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 2: Our comment: The "lastMeasuredDesiredWidth" integer is the current width of the column if its animated (at the moment). If its not animated, then its the minimum required width for the column. Its also combined with the left & right padding (extra spacing) values.

to automatically generate explanations of any statement that is appropriate for a novice programmer, and that the user can have as much information as needed to fully understand the statement in question.

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

An example of our approach can be seen in Figure 1. When a user mouses over an arbitrary Java code statement,

Algorithm 1 Process Method

function SUMMARIZEMETHOD(M)
  $output \leftarrow$ ""
  $list \leftarrow []$
  if M *is void* **then**
    $list \leftarrow$ set of variable assignments $\in M$, sorted by most frequent var, last occurrence first
  else
    $list \leftarrow$ set of return statements $\in M$
  end if
  for each $s \in list$ **do**
    $output \leftarrow$ output + LEXICALIZE(S, MD)
  end for
  return *output*
end function

```
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 3: Our comment: For every "element" in the "RootElements" list of the RoundEnvironment, "Stack" is a new "LinkedList" where the element is added. Then, the "collectElementsAnnotatedOrMetaAnnotatedWith" method adds new content to the stack given a stack & annotationType. The 1st item of stack (which is the element itself) is removed to prevent duplicates.

a description is displayed that combines information from other lines in the method following control and data flow dependencies and by incorporating descriptions of any called methods.

Our approach consists of rules to summarize loop statements, variable declarations and assignments, if and switch statements, complex arithmetic and comparison expressions, and method calls. Due to space constraints, we explain our approach to generate summaries for these statements within the context of our recursive method call summary approach.

If a statement contains a method call, our approach recursively summarizes the most pertinent statements within it or pulls in its API documentation. Our approach to summarizing methods varies for void vs. non-void methods. The process is shown in Algorithm 1. For non-void methods, we adapt the idea of Sridhara et al. [15] 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 last occurrence first. This is similar to the approach of Sridhara, et al [15] of data-facilitating s-units, however, unlike Sridhara et al., our idea is not limited to dependencies of "ending s-units", "void-return s-units", and "same action s-units". 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 (see Figure 2). We also consider statements that supply the return statement (see Figure 3).

2.1. Automatic Method Summarization

Our approach towards summarizing methods involves 4 main components: (1) Selecting the most significant lines of code within the method to summarize, (2) lexicalizing

Algorithm 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
```

those significant lines of code, (3) finding and lexicalizing their most significant dependencies, (4) repetition of these steps if the dependency is within another method (e.g. a used method call's declaration) or usage of API Documentation when available.

Our process of finding the most significant lines of code within a method is partially based s-units of Sridhara et al. [15]. S-units are a set of statements that consists of a method's return, method calls in void methods, def-use chains, and conditional statements [15]. Like Sridhara et al. our 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 by utilizing dependencies found from these s-units, as well as recursively processing complex expressions and usage of API Documentation. A complex expression is an expression that consists of multiple variables, multiple literals, multiple method calls, and/or multiple mathematical operators, etc.

This leads to Algorithm 2, where we recursively analyze the sub-expressions located within S. The most common sub-expressions our approach processes: Method calls, constructors, variables, complex expressions, Enumerators, literals, parameters, and lists. 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 on Figure 3.

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

Figure 4: Our comment: The "desiredWidth" integer variable is assigned the value of calling the max method, which returns the greater of two double values.

To prevent duplicate explanations of statements, we keep track of introduced variables and control flow statements in a list of statements, $seen$. Generally, a variable, parameter, or control flow statement that exists in $seen$ was already mentioned, or it's type and name from assignments with a common left hand side was 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. Because complex expressions may confuse novices with the many sub-cases to consider along with appropriate order of parenthesis that must be followed (e.g., nested sub-expressions in parenthesis with multiple method calls, variables, in between many operators), we generate detailed explanations for such statements.

In the case of parenthesis, we recursively process the subexpressions that exists within it, from leftmost subexpression in parenthesis to rightmost. Operators are intuitively lexicalized (greater than for '$>$', etc.). For more complex sub-expressions, we recursively analyze the multiple possible dependencies located within the left and right hand sub-expressions.

2.2. 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 4.

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 it's control and data dependencies, as well as relevant sub-expressions found within these dependencies. For non-built-in method calls, the same process of lexicalizing parameters is done after recursively analyzing it's declaration. Method parameters are also analyzed.

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 effect their numerical values. If we have multiple objects of the same types within the parameters

```
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 5: Our comment: Transition is increased for each time isBlack & inBlack are NOT both true or false at the same time.

arguments, we also mention "1st, 2nd, 3rd...etc" and use grammatically correct listing of the parameters (commas, and... etc). This helps introduce the arguments being used before recursive analysis of dependencies.

2.3. Data and Control Dependencies

A variable may have many data and control dependencies. Our lexicalization approach searches for all assignment and appending statements that a variable V depends on, introduces the variable and it's type, then recursively lexicalizes the right hand side of the assignments V depends on. If V depends on an appending statement, we lexcalize the expression being appended. For every statement S that V depends on, we consider all control dependency statements of S. As shown in Figure 5, we recursively lexicalize the conditions of the control dependency statement.

3. Evaluation

We propose to answer the following research questions:

- Do novices prefer our explanations (H) over the competing state of the art (Sridhara) [15]?
- Do novices prefer any explanation over none at all?
- Do novices prefer explanations based on API documentation or recursively summarizing method calls?

3.1. Setup

We recruited 30 undergraduate CS students taking a second programming course in Java and more senior students taking a course in algorithms (with Java and data structures as a prerequisite) as our novice programmers. We gave each participant an online survey, asking them to understand and attempt to fix a bug in a small snippet of source code. Along with each code snippet were 5 checkboxes: an explanation by Sridhara et al.'s approach, 3 different explanations from our approach (one our high-level equivalent like those generated by Sridhara et al., and two more detailed), and 'none of the above'. Thus, participants were free to select any combination of explanations as helpful for solving the bug. Each participant was asked to analyze two of the four bugs. To create our 4 bugs, we randomly selected 3 methods from 3 popular open source Java programs on github (Sugoi Survey, Spring Boot, rxTools) and manually inserted a single bug into each method. One bug was used twice with different explanations generated by our approach: API-based and recursive.

TABLE 1: Which explanations do people prefer more?

Total	Sridhara	H	None
60	19	50	6

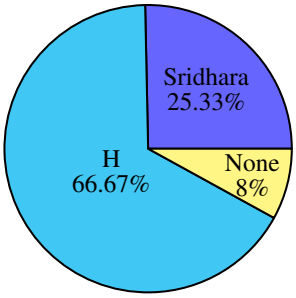

Figure 6: Total percentages of those who preferred explanations of Sridhara et al., those who preferred at least one of our explanations per question "H", and those who preferred no explanations.

3.2. Results and Discussion

Students preferred our explanations more than they did the Sridhara explanations and they preferred any explanation to no explanations, as shown in Table 1 and Figure 6. According to the results of the Chi-Squared goodness of fit test, students have a statistically significant difference in preference between choosing our 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. Comparing our 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 our choices and only one Sridhara choice. Students do not prefer our choices in a ratio that is greater than 75%:25%, $\tilde{\chi}^2$ = 0.24 where N = 69, df = 1, p = .63. We did not find a statistically significant difference in the degree to which novices (73.33%) and experts (70.83%) preferred our hints, $\tilde{\chi}^2$ = 0.05 where N = 69, df = 1, p = .83.

Next, we investigated whether participants preferred API explanations over recursive explanations from the source code identifiers. We tested this hypothesis in two ways: (1) by comparing responses to the same bug, one survey using API-based explanations and one solely depending on source code, and (2) by comparing responses from the two bug questions that used API-based explanations versus the two questions that did not. In both cases we did not find a statistically significant difference: (1) $\tilde{\chi}^2$ = 1.29 where N = 19, df = 1, p = .26 and $\tilde{\chi}^2$ = 2.58 where N = 50, df = 1, p = .11; (2) $\tilde{\chi}^2$ = 2.58 where df = 1, N = 50, p = .11.

TABLE 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

Next, we evaluate the number of times each explanation is chosen by bug. This allows us to observe effects from particular code snippets, show in Table 2. From this table we observe that participants preferred explanations that summarized the statement and at most one level of recursion deep (Bug A H1, Bug B H3). It is interesting to note that the two bugs with API-based explanations (C and D) were at most one level of recursion deep.

Finally, we observe the consistency of subject preferences. For instance, we can answer the following question: Does a subject who chose Sridhara, et al explanation for the first question tend to choose Sridhara again for the 2nd question? 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 our explanations selected two of our 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.

3.3. Threats to Validity

One potential threat to validity is that there was only one possible choice of Sridhara while we have three possible choices of our approach. We attempted to mitigate this threat by counting a selection of any of our three explanations as just one choice in the analysis. We helped focus the participant's attention on Sridhara's explanation by placing it first of all the explanations available in each survey.

4. Conclusions and Future Work

In this paper we presented an approach to automatic statement summarization. We compared our approach to the competing state of the art technique by Sridhara et al. [15] and asked 30 undergraduate CS students to select what automatically generated comments were useful during a debugging task. The students preferred our approach over that of Sridhara et al., with statistical significance.

These results show promise, motivating further work. In the short term, we plan to expand our evaluation to observe novice programmers using our tool in Eclipse rather than as a survey, and we would like to expand our study to include early career novice programmers as well as more expert programmers. We plan to investigate to what extent the length of the statement summaries influenced the results, and adapt the approach to allow users to *pull* the information as they are reading, rather than presenting a lengthy paragraph as seen in Figure 1. In the longer term we plan to further improve our technique by leveraging observations of eye tracking of what keywords are most important [14] or including additional information about calling context [8].

Acknowledgments

The authors would like to thank Dr. Sarah Abramowitz for her invaluable statistical guidance, and Wyatt Olney and Andrew Castelluccio for their contributions to early stages of this research project. This work is supported in part by National Science Foundation Grant No. 1527093.

References

[1] U. Dekel and J. Herbsleb. *Improving api documentation usability with knowledge pushing*. ICSE 2009.

[2] V. Fix, S. Wiedenbeck, J. Scholtz. *Mental representations of programs by novices and experts*. CHI 1993.

[3] L. Guerrouj, D. Bourque, P. C. Rigby, *Leveraging informal documentation to summarize classes and methods in context*, ICSE 2015.

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

[5] S. Haiduc, J. Aponte, A. Marcus, *Supporting program comprehension with source code summarization*, ICSE 2010.

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

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

[8] P.W. McBurney and C. McMillan. *Automatic documentation generation via source code summarization of method context*. ICPC 2014.

[9] L. Moreno, J. Aponte, G. Sridhara, A. Marcus, L. Pollock, V. Shanker, *Automatic Generation of Natural Language Summaries for Java Classes*, ICPC 2013.

[10] N. Nazar, H. Jiang, G. Gao, T. Zhang, X. Li, Z. Ren, *Source code fragment summarization with small-scale crowdsourcing based features*, Frontiers of Computer Science: Selected Publications from Chinese Universities, v.10 n.3, 2016.

[11] Y. Oda, H. Fudaba, G. Neubig, H. Hata, S. Sakti, T. Toda, S. Nakamura, *Learning to Generate Pseudo-Code from Source Code Using Statistical Machine Translation*, ASE 2015.

[12] M. P. Robillard , Y. B. Chhetri, *Recommending reference API documentation*, ESE, v.20 n.6, 2015.

[13] 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.

[14] P. Rodeghero C. McMillan P. W. McBurney N. Bosch S. D'Mello *Improving Automated Source Code Summarization via an Eye-Tracking Study of Programmers*. ICSE 2014.

[15] G. Sridhara E. Hill D. Muppaneni L. Pollock and K. Vijay-Shanker. *Towards Automatically Generating Summary Comments for Java Methods*. ASE 2010.

[16] G. Sridhara, L. Pollock, K. Vijay-Shanker. *Automatically detecting and describing high level actions within methods*. ICSE 2011.

[17] G. Sridhara, L. Pollock, K. Vijay-Shanker. *Generating Parameter Comments and Integrating with Method Summaries*, ICPC 2011.

[18] C. Treude, M. Sicard, M. Klocke and M. Robillard. *TaskNav: Task-based navigation of software documentation*, ICSE 2015.

[19] X. Wang, L. Pollock, K. Vijay-Shanker. "Automatically generating natural language descriptions for object-related statement sequences", SANER 2017.

[20] A. T. T. Ying , M. P. Robillard. *Code fragment summarization*, FSE 2013.