CrowdSummarizer

Automated Generation of Code Summaries for Java Programs through Crowdsourcing

Sahar Badihi and Abbas Heydarnoori, Sharif University of Technology

// CrowdSummarizer exploits crowdsourcing, gamification, and natural-language processing to automatically generate high-level summaries of Java program methods. Researchers have implemented it as an Eclipse plug-in together with a web-based code summarization game that can be played by the crowd. //

MUCH OF SOFTWARE maintenance costs involves developers trying to comprehend a program's source code (which might include thousands of lines of code) to focus on the parts that require maintenance.[1] Code summaries, which briefly describe code's functionality and purpose, can help programmers find the relevant parts of code and perform maintenance more quickly and easily.[2,3] Code summaries primarily benefit newcomers, who have no prior knowledge of the code. However, senior developers can also benefit from code summaries because they might have worked on the code a while ago and forgotten some concepts.

Crowdsourcing outsources tasks to a crowd of people through an open call (for example, through the Internet) instead of traditional suppliers.[4] It now supports a range of software engineering activities such as requirements engineering, design, coding, testing, and evolution and maintenance.[5] However, to the best of our knowledge, using a crowd of developers to write code summaries is a novel idea in the context of program comprehension.

Toward that end, CrowdSummarizer applies crowdsourcing and gamification to motivate developers to write high-level summaries of Java program methods. Furthermore, it exploits natural-language-processing techniques to automatically generate summaries of newly submitted methods on the basis of summaries it has collected from the crowd. Studies of our implementation showed that CrowdSummarizer is practical for developer use and automatically generates accurate, comprehensible summaries.

Our Approach

Figure 1 shows an overview of our approach, which has two main components: the crowdsourcing component and the automated natural-language summary generator. Here, we describe these components. For further details, see our technical report.[6]

Collecting Summaries

Existing code summarization approaches (for some examples, see the sidebar) pose some restrictions. For instance, they might summarize only methods of a limited length or some special aspects of a program (such as the contexts or roles of methods).

To overcome these restrictions, CrowdSummarizer employs crowdsourcing. Specifically, individual developers decompose a Java program into its methods and submit them to the CrowdSummarizer website. Next, a crowd of developers writes summaries of those methods, and CrowdSummarizer presents the results to the original developers.

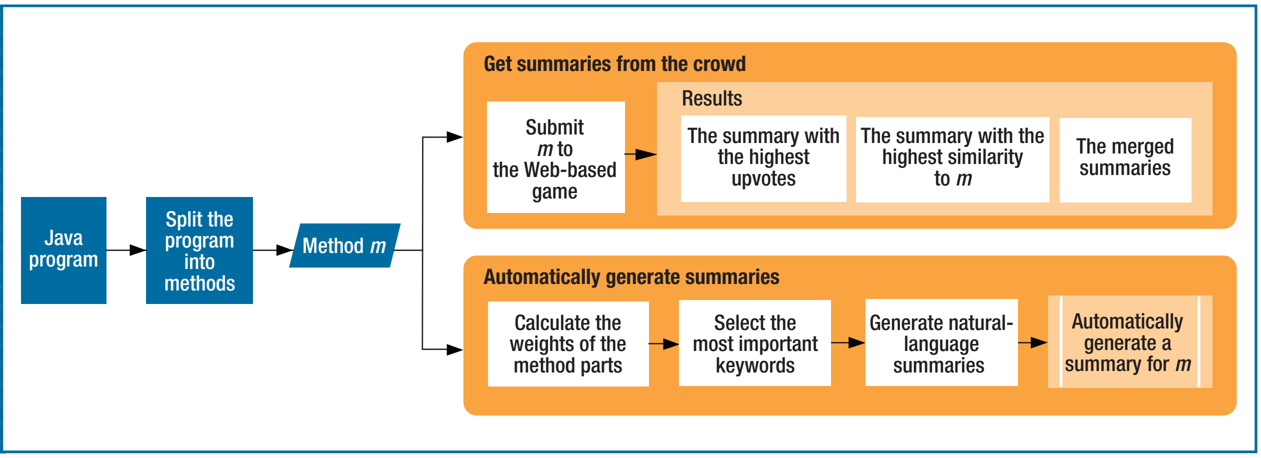

Java program
Split the program into methods
Method *m*
Get summaries from the crowd
Submit *m* to the Web-based game
Results
The summary with the highest upvotes
The summary with the highest similarity to *m*
The merged summaries
Automatically generate summaries
Calculate the weights of the method parts
Select the most important keywords
Generate natural-language summaries
Automatically generate a summary for *m*

FIGURE 1. An overview of CrowdSummarizer, which employs crowdsourcing, gamification, and natural-language processing to produce high-level summaries of Java program methods.

RELATED WORK IN CODE SUMMARIZATION

Paige Rodeghero and her colleagues used an eye-tracking system to learn what keywords in the body of a method are more important for programmers who are summarizing that method.[1] However, the methods must have a small number of lines of code, and developer fatigue can affect the results.

Giriprasad Sridhara and colleagues' approach automatically generates natural-language summaries of a Java method's overall actions.[2] Unlike Rodeghero and her colleagues' eye-tracking approach, which employs a number of developers, this approach is fully automated.

Paul McBurney and Collin McMillan's approach helps programmers understand a method's context, such as how it's called or how its output is used.[3] So, their approach doesn't summarize the method itself.

References

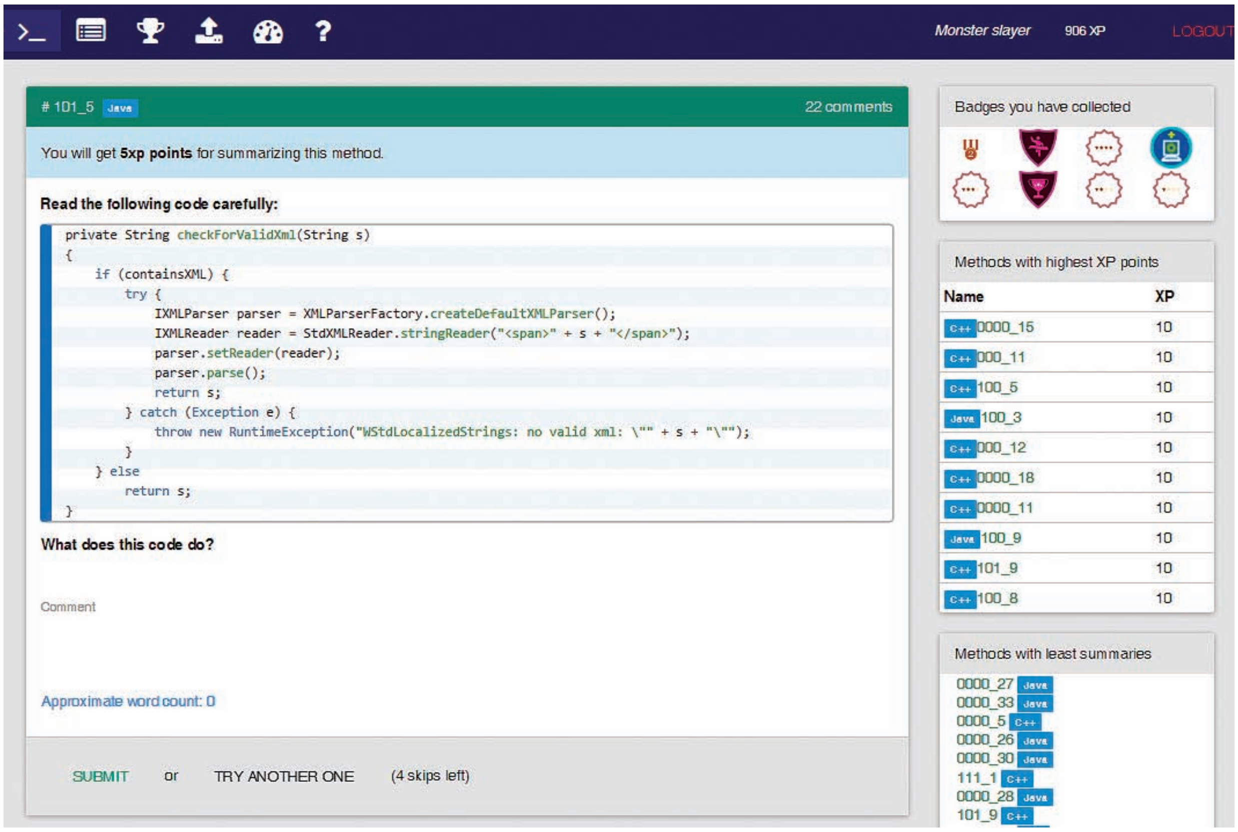
1. P. Rodeghero et al., "Improving Automated Source Code Summarization via an Eye-Tracking Study of Programmers," *Proc. 36th Int'l Conf. Software Eng.* (ICSE 14), 2014, pp. 390–401.
2. G. Sridhara et al., "Towards Automatically Generating Summary Comments for Java Methods," *Proc. 2010 IEEE/ACM Int'l Conf. Automated Software Eng.*, 2010, pp. 43–52.
3. P.W. McBurney and C. McMillan, "Automatic Source Code Summarization of Context for Java Methods," *IEEE Trans. Software Eng.*, vol. 42, no. 2, 2016, pp. 103–119.

The fundamental issue and key requirement in our approach is to get a large number of developers to write summaries. One of the great drivers for motivating and engaging people in a crowdsourced task is gamification.[7]

Gamification. Gamification employs game design concepts and mechanisms in nongame contexts to make those contexts more fun and engaging.[8] So, people enjoy themselves while performing the crowdsourced tasks. To motivate developers to write method summaries, we designed and implemented a web-based game in Python. (For a demonstration of this game, see sites.google.com/site/crowdsummarizertechreport.)

Any developer who registers for the CrowdSummarizer platform can be a player, earn points, and move up to the top-players list. The game has eight levels; as the levels increase, the methods become harder to summarize (see Figure 2a).

Evaluating submitted answers carefully to decide whether an answer is acceptable is important in any crowdsourcing process. So, in an evaluation step, CrowdSummarizer randomly displays a number of methods with their summaries and lets players *upvote* or *downvote* other players' summaries. To prevent personal biases, we hide the summary authors' names. Irrelevant summaries are removed automatically because of the downvotes they receive.



You will get **5xp points** for summarizing this method.

Read the following code carefully:

```
private String checkForValidXml(String s)
{
    if (containsXML) {
        try {
            IXMLParser parser = XMLParserFactory.createDefaultXMLParser();
            IXMLReader reader = StdXMLReader.stringReader("<span>" + s + "</span>");
            parser.setReader(reader);
            parser.parse();
            return s;
        } catch (Exception e) {
            throw new RuntimeException("WStdLocalizedStrings: no valid xml: \"" + s + "\"");
        }
    } else
        return s;
}
```

What does this code do?

Comment

Approximate word count: 0

SUBMIT or TRY ANOTHER ONE (4 skips left)

Badges you have collected

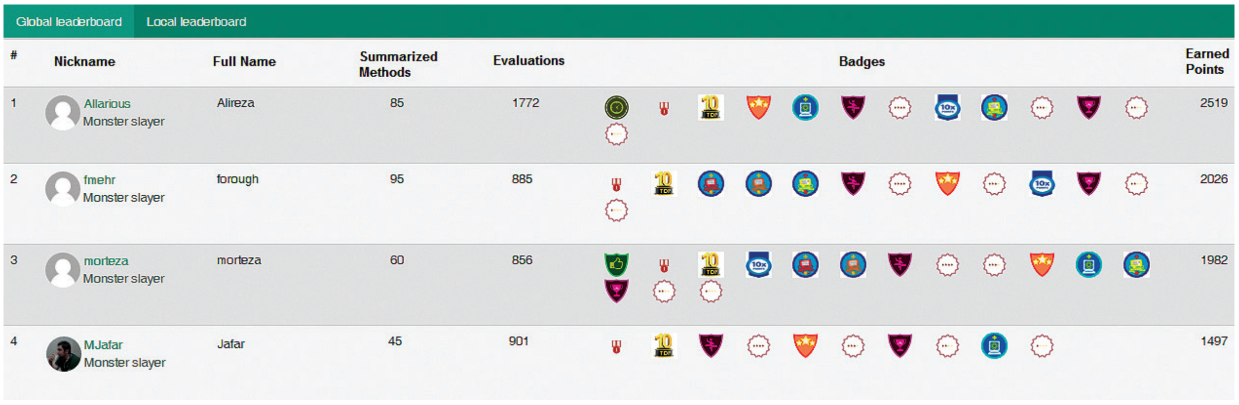
Methods with highest XP points

Name	XP
C++ 0000_15	10
C++ 000_11	10
C++ 100_5	10
Java 100_3	10
C++ 000_12	10
C++ 0000_18	10
C++ 0000_11	10
Java 100_9	10
C++ 101_9	10
C++ 100_8	10

Methods with least summaries

0000_27	Java
0000_33	Java
0000_5	C++
0000_26	Java
0000_30	Java
111_1	C++
0000_28	Java
101_9	C++

(a)



#	Nickname	Full Name	Summarized Methods	Evaluations	Badges	Earned Points
1	Allarious Monster slayer	Alireza	85	1772		2519
2	fmehr Monster slayer	forough	95	885		2026
3	morteza Monster slayer	morteza	60	856		1982
4	MJafar Monster slayer	Jafar	45	901		1497

(b)

FIGURE 2. CrowdSummarizer's web-based game. (a) The webpage that requests code summaries from the crowd. (b) The global leaderboard. The game has eight levels; as the levels increase, the program methods become harder to summarize.

To further motivate developers to play the game, players must reach at least level four to be allowed to submit their own code for crowd-based summarization. Moreover, we used the following gamification elements to make the game more fun and motivating.

Almost every gamified system employs points. For each method, we calculate points corresponding to its complexity. To do this, we use the Metrics tool (metrics.sourceforge.net), which calculates a method's complexity with respect to metrics such as the method's number of lines of code, conditional statements, and branches. A player who writes a method summary gets the assigned points. In addition, the first three players who summarize a method earn double points. Evaluating other players' summaries and writing summaries that receive a high number of upvotes can also increase a player's points. Writing summaries that receive downvotes decreases a player's points. To prevent players from submitting worthless evaluations of summaries, we use a trap mechanism and fake account to identify cheaters.

Developers can get initial results from the platform and don't have to wait for the crowd to write method summaries.

Gamification research has indicated that ranking systems are an effective motivator.[8] So, our game employs both a global and a local leaderboard. The former shows the competition among all the players (see Figure 2b); the latter displays just the players at that player's programming experience level. The game also encourages players with statements such as, "Hurry up; writing two other summaries will shift you up to second place in the global leaderboard" and "Keep on!"

In addition, players receive tagged images as badges on the basis of their performance (see Figure 2b). For example, a player with the most upvotes will get the Good Summarizer badge.

Generally, levels indicate where players stand in a game. To increase our game's attractiveness, each level has a title. For example, the titles of levels one, four, and eight are Starting to See the Light, Middle of the Way, and Monster Slayer. A progress bar shows how much of the current level a player has completed.

To motivate players to continue, players receive a mystery box with a random gift (for example, points or a badge) at levels two, five, and seven.

Representing players with an avatar can increase their engagement.[8] So, if a player has created an avatar on the Gravatar website (en.gravatar.com), he or she can use it in our game.

Getting summaries. After CrowdSummarizer collects summaries from the crowd through our game, it can generate various kinds of output. It currently supports the following three kinds of output.

The first is the summary with the most upvotes. For a given method, CrowdSummarizer chooses the summary with the biggest positive difference between its upvotes and downvotes.

The second kind of output is the summary with the highest similarity to its method. CrowdSummarizer selects the summary that has the most keywords related to the given method (we discuss the equation for this later).

The third kind of output is merged summaries. CrowdSummarizer extracts the common concepts of the submitted summaries. To identify these concepts, it uses the *k*-means algorithm, and it groups the remaining words as additional detail. Users can also access each submitted summary.

Automatically Generating Summaries

CrowdSummarizer continuously learns a set of weights and sentence templates from the methods and summaries it has collected from the crowd. It uses these weights and templates to automatically generate summaries of new methods. So, developers can get initial results from the platform and don't necessarily have to wait for the crowd to write summaries of their methods. We implemented this part of the platform as an Eclipse plug-in. (For an online demonstration, see sites.google.com/site/crowdsummarizertechreport.)

Calculating weights. This step assigns weights to different parts of a method according to the frequency of their use in the crowdsourced summaries. CrowdSummarizer uses these weights to select the most important keywords, with which it automatically generates summaries.

In our evaluations of CrowdSummarizer (which we discuss in more detail later), we noticed that when developers wrote summaries, they distinguished mainly between nine parts of a method, in this order of importance (from most to least):

- the method's name and return type,
- parameters,
- ending units (for example, **return** and **printout** statements),
- method calls,
- branches (**if** and **switch** statements),
- loops (**for**, **while**, and **do-while** statements),
- assignments,
- variables, and
- error handlings (for example, **try–catch**, **exception**, and **throw**).

As in other approaches based on linguistic information, meaningless identifiers significantly affect our results' quality. So, our approach produces better results when developers use meaningful identifiers and follow naming conventions and style guidelines.

Like other natural-language-processing systems, CrowdSummarizer preprocesses the dataset to refine and prepare it for the next analysis tasks. This preprocessing involves

- tokenization,
- splitting (using Dawn Lawrie and her colleagues' approach[9]),
- expanding the abbreviations (using AMAP[10]),
- correcting misspelled words (performed by norvig.com/spell-correct.html),
- removing stopwords (using the Natural Language Toolkit; www.nltk.org/api/nltk.html), and
- lemmatization (using the WordNet Lemmatizer; www.nltk.org/api/nltk.stem.html).

Through these steps, we extract the keywords of each method *m* and each of its submitted summaries *s*, called *MethodKeywords*(*m*) and *SummaryKeywords*(*s*). We compute the common keywords between *m* and the corresponding *s* as

$$CommonWords(s,m) = SummaryKeywords(s) \cap \Big(MethodKeywords(m) \cup Syn\big(MethodKeywords(m)\big)\Big).$$

Syn(...) is the set of all the synonyms of the words in *MethodKeywords*(*m*). We use the WordNet lexical database (wordnet.princeton.edu) to find each word's synonyms. For example, suppose "calculate" is in the method's name and "bike" is in the parameters list. Some summaries might have used those words, whereas others might have used synonyms such as "compute" and "bicycle." Because we calculate the different method parts' importance, we take the synonyms into account to increase the results' precision.

To compute the weight of each word in *CommonWords*, we first calculate the weight of a word in that method and each of its summaries:

$$Weight\_in\_method(w,m) = \frac{The\ no.\ of\ occurrences\ of\ word\ w\ in\ MethodKeywords(m)}{The\ no.\ of\ words\ in\ MethodKeywords(m)},$$

$$Weight\_in\_summary(w,s) = \frac{The\ no.\ of\ occurrences\ of\ word\ w\ in\ SummaryKeywords(s)}{The\ no.\ of\ words\ in\ SummaryKeywords(s)}.$$

Then, for each word in *CommonWords*, we compute a normalized weight, which means the rate of using the method words in a summary:

$$NormalizedWeight(w) = \frac{Weight\_in\_summary(w,s)}{Weight\_in\_method(w,m)}.$$

Next, *weight*(*w*) is the average of the normalized weights for *w* among all the summaries submitted for a method.

Selecting keywords. This step selects the most important keywords for describing a method. In information retrieval, *term frequency–inverse document frequency* (tf–idf) is a numerical statistic that reflects how important a word is to a document in a corpus.[1] The tf–idf factor is used widely in natural-language text processing and text summarization.[3]

The problem with using tf–idf in code summarization is that it treats the source code as plain text; thus, the words in different parts of a method are treated equally. For example, the words in the method signature are treated the same as the words in a **while** loop. However, as we discussed before, developers distinguish between various parts of a method.

To address this issue, we also consider the weight factor we computed previously. More specifically, we compute the importance of word *w* in method *m* as

$$Importance(w,m) = tf\text{-}idf\,(w,m) \times weight\,(w).$$

```
public boolean remove(Listener listener)
{
 if (listeners != null {
    int index = listeners.lastIndexOf(listener);
    if (index != -1) {
       try{
       listeners.remove(index);
       return true;
       }
       catch (numberException e)
       {
        return false;
       }
       if (listeners.isEmpty()) {
          listeners = null;
       }
    }
 }
}
```

(a)

This method checks if it can remove a listener.

It checks if [listeners] is not equal to null to do actions.

This method calls the method which get last index of the Listener in listeners to get its value and assign to variable [index].

This method finally returns [true] if remove [index] from [listeners].

It handles the errors using try-catch mechanism and throwing an exception.

(b)

FIGURE 3. Employing the Software Word Usage Model (SWUM) to generate a summary. (a) An example method. (b) The automatically generated summary. SWUM captures a method's linguistic elements in terms of its action, theme, and any secondary arguments.

On the basis of our evaluations of the summaries' length, the number of keywords selected for a method's summary is related to that method's complexity. As we mentioned earlier, we use the Metrics tool to calculate a method's complexity. We then normalize this number to the range of 1 to 10 to identify the number of the most important keywords. This is because it's recommended that a summary use at most 10 keywords.[3]

Generating natural-language summaries. The last step is to automatically generate natural-language summaries with the help of the selected keywords. To generate descriptions of a method's name and its method calls, we employ the Software Word Usage Model (SWUM).[11] For other parts of the method, we employ the sentence templates we mentioned earlier. (These templates appear at sites.google.com/site/crowdsummarizertechreport.)

For instance, our evaluations showed that 90 percent of the time, developers used "till" or "until" in their summaries instead of the Java keyword `while`. Also, 86 percent of the methods that returned a Boolean value had "check" in their summaries.

SWUM captures a method's linguistic elements in terms of its action, theme, and any secondary arguments. For instance, the phrase "Compare string str2 to string str1" can summarize the statement `str1.compareTo(String str2);`. In this phrase, the action is "Compare," the theme is "string str2," and the secondary argument is "(to) string str1." The key point of SWUM is the assumptions it makes about Java naming conventions, such as camel case or starting most method names with a verb.

Here's an example of how we employ SWUM. For a method call, our template is "This method calls the method which [action, theme, and secondary arguments] to get its value and assign to the variable [the variable on the left side of the assignment]." Of course, we use the second part of this template when the method call is on the assignment's right side.

For instance, consider the statement `int index = listeners.lastIndexOf(listener);` (see Figure 3a). The generated summary of it would be "This method calls the method which get last index of the Listener in listeners to get its value and assign to variable [index]." In this example, SWUM tells us that the action is "(get) last index of," the theme is "Listener," and the secondary argument is "(in) listeners."

An Example

Here, we show how our approach is used in practice to generate the summary of the method in Figure 3a. Consider the keyword listener. Because listener occurs in the method's input parameter and one of the method calls' input parameters (lastIndexOf(…)), we compute the average of those two method parts' weights (named *wt*). Afterward, we calculate the tf–idf factor for listener (named *f*). Then, the importance of including this keyword in the generated summary is $wt \times f$.

Next, using the Metrics tool, we calculate a complexity of six for our example method. So, the generated summary should include the following six most important keywords:

- remove (the method's name),
- listener,
- listeners != null (the branches),
- lastIndexOf (the method call),
- try–catch (exception handling), and
- true (the ending unit and branches).

We use these keywords as input to our natural-language text generator and, with the help of the templates and SWUM, we generate the summary (see Figure 3b).

If the user isn't satisfied with the automatically generated summary, he or she can submit the method to our game and ask the crowd to write a summary. For this example method, in our evaluations, the participants found this summary easy to comprehend and were satisfied with its length. Furthermore, the measured values for the overall accuracy, precision, and recall were 0.87, 0.9, and 0.76, respectively.

TABLE 1 Participant feedback on CrowdSummarizer.

Question	Average no. of points (1–5)
How enjoyable is it to play? (1 = least, 5 = most)	3.82
How easy or difficult is it to play? (1 = most difficult, 5 = easiest)	3.24
How successful was the website in encouraging you to write summaries? (1 = poor, 5 = excellent)	4.02
How successful was the website in showing code summarization's importance? (1 = poor, 5 = excellent)	3.52
How successful were the website elements such as badges in enhancing your code summarization skill? (1 = poor, 5 = excellent)	3.39

User Studies

We evaluated CrowdSummarizer in the following two user studies.

CrowdSummarizer's Applicability

This study aimed to answer two questions:

- Is CrowdSummarizer applicable to developers?
- Does it motivate them to use it?

The setup. We issued a call for participation in the undergraduate and graduate software engineering courses and the advanced programming classes at the Sharif University of Technology. In response, 149 undergraduate and graduate students with an average of 5.29 years of programming experience played our web-based game. We collected approximately 4,000 summaries of 128 methods, with an average of 19 summaries for each method.

To increase the results' generalizability, we selected the methods randomly from 11 open-source Java applications of various sizes (containing 318 to 12,793 methods with an average of 2,158 methods) from different domains (including text editors, multimedia, and games). Moreover, the chosen methods had varying properties (for example, the number of lines of code, the number of parameters, and the type of return values).

After the participants played the game, they filled out a questionnaire to provide feedback about the game and our approach.

The results. Table 1 summarizes the participants' feedback. They felt that CrowdSummarizer is applicable to developers and that it motivated them to use it. However, some participants suggested ways to improve our approach. For instance, some of them felt that the part of the game in which they assessed other players' summaries was a little boring.

The Automatically Generated Summaries' Quality

This study aimed to answer three questions:

- What's the quality of CrowdSummarizer's generated

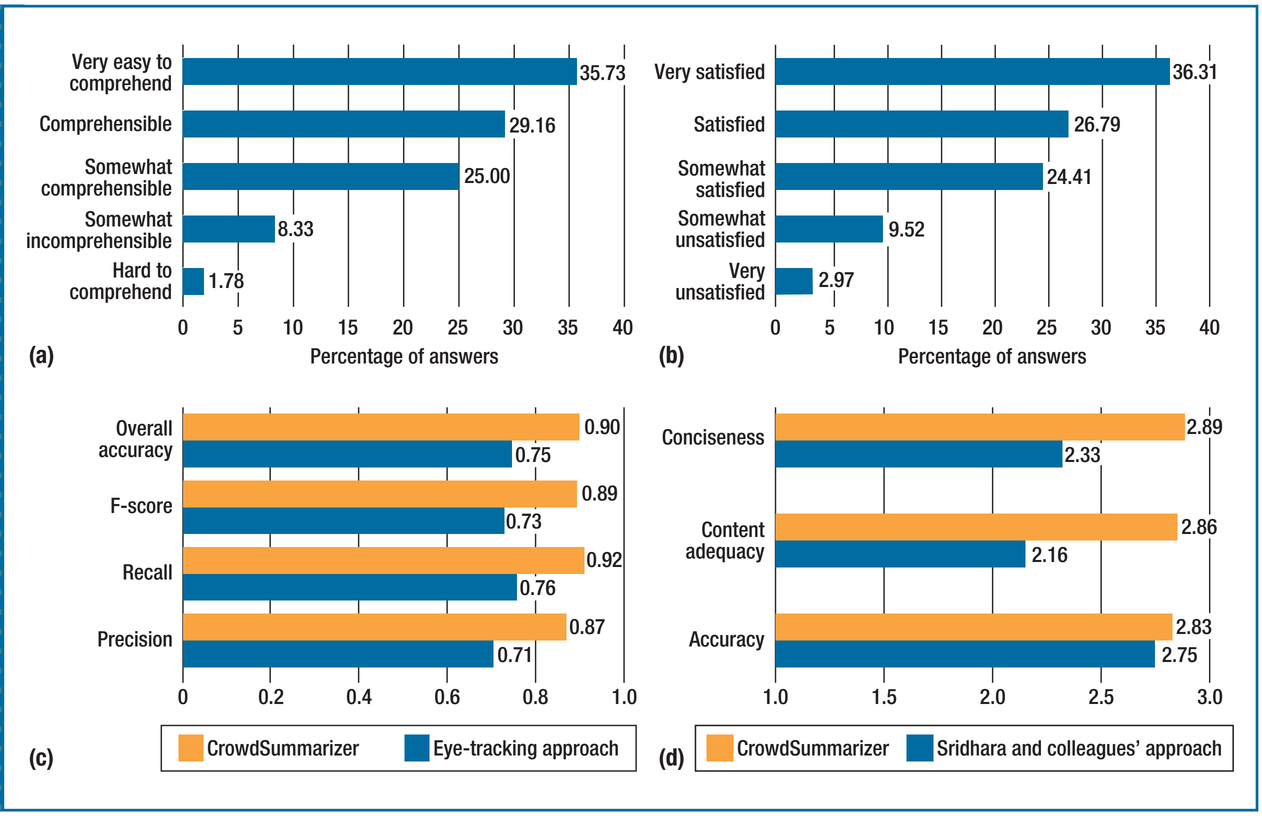


FIGURE 4. Some results of the CrowdSummarizer evaluations. (a) Experts' opinions on the automatically generated summaries' comprehensibility. (b) Experts' satisfaction with the generated summaries. (c) Comparing CrowdSummarizer with an eye-tracking approach. (d) Comparing CrowdSummarizer with Giriprasad Sridhara and colleagues' approach. (For more on these other approaches, see the sidebar.)

summaries in terms of comprehensibility and experts' satisfaction?

- Does CrowdSummarizer extract important keywords better than existing code summarization approaches?
- Does CrowdSummarizer generate higher-quality summaries than existing code summarization approaches?

Comprehensibility refers to how understandable the generated summaries were for developers. Experts' satisfaction measures to what extent the developers were satisfied with the generated summaries' length.

The setup. We recruited 14 expert developers who hadn't participated in the first study. All of them were graduate students in the Sharif University of Technology's Department of Computer Engineering. They had an average of eight years of programming experience, five-and-a-half years of Java experience, and at least one year of industry experience.

We used our approach to automatically generate summaries of 78 methods randomly selected from the programs used in the first study. Each subject wrote keywords and summaries for 12 methods selected randomly from the 78 methods. Then, they viewed the summaries our approach generated for each of the 12 methods and gave us feedback on those summaries' quality.

Next, we compared our approach to Paige Rodeghero and her colleagues' eye-tracking approach, which also extracts the most important keywords and performs better than other approaches in this area. (For more on this approach, see the sidebar.) Specifically, we compared the keywords that our approach and the eye-tracking approach extracted with the keywords our experts chose. This way, we compared the two approaches in terms of precision, recall, F-score, and overall accuracy.

We computed the overall accuracy as

$$\textit{Overall Accuracy} = \frac{TP + TN}{TP + TN + FP + FN},$$

where

- TP (true positives) was the number of keywords the approach correctly extracted,
- TN (true negatives) was the number of keywords the approach correctly didn't extract,
- FP (false positives) was the number of keywords the approach incorrectly extracted, and
- FN (false negatives) was the number of keywords the approach incorrectly didn't extract.

The correctly extracted keywords were those that a domain expert would use in a summary.

Finally, the participants rated the generated summaries in terms of

- conciseness (whether a summary contained unnecessary information),
- content adequacy (whether a summary missed important information), and
- accuracy (whether a summary correctly showed the actions done by the method).

To do this, they used a range of 1 to 3 (low, medium, or high). Afterward, we compared the results with those that Giriprasad Sridhara and colleagues reported for their approach. (For more on that approach, see the sidebar.)

The results. Most of the experts were satisfied with the generated summaries' comprehensibility (see Figure 4a) and quality (see Figure 4b). Additionally, CrowdSummarizer extracted the most important keywords better than the eye-tracking approach (see Figure 4c). Finally, although Sridhara and colleagues' approach requires less human involvement, CrowdSummarizer generated higher-quality summaries (see Figure 4d). This suggests that taking humans' intelligence into account can improve code summarization. However, further evaluation is needed.

ABOUT THE AUTHORS

SAHAR BADIHI is a master's student in software engineering at the Sharif University of Technology. Her research interests include software engineering, program comprehension, and code summarization. Badihi received a bachelor of software engineering from the Isfahan University of Technology. Contact her at sbadihi@ce.sharif.edu.

ABBAS HEYDARNOORI is an assistant professor in the Department of Computer Engineering at the Sharif University of Technology. His research interests include software evolution, mining software repositories, and recommendation systems in software engineering. Heydarnoori received a PhD from the University of Waterloo's School of Computer Science. Contact him at heydarnoori@sharif.edu.

We can easily extend CrowdSummarizer to include other programming languages such as C++. Also, we plan to evaluate how useful it is for performing a particular software maintenance task, such as feature location or debugging.

References

1. G. Salton, *Automatic Text Processing: The Transformation, Analysis, and Retrieval of Information by Computer*, Addison-Wesley, 1989.
2. P.W. McBurney and C. McMillan, "Automatic Source Code Summarization of Context for Java Methods," *IEEE Trans. Software Eng.*, vol. 42, no. 2, 2016, pp. 103–119.
3. S. Haiduc et al., "On the Use of Automated Text Summarization Techniques for Summarizing Source Code," *Proc. 17th Working Conf. Reverse Eng.*, 2010, pp. 35–44.
4. J. Howe, "The Rise of Crowdsourcing," *Wired*, June 2006; www.wired.com/2006/06/crowds.
5. K. Mao et al., *A Survey of the Use of Crowdsourcing in Software Engineering*, tech. report RN/15/01, Univ. College London, 2015.
6. S. Badihi and A. Heydarnoori, *Generating Method Summaries Using the Power of the Crowd*, tech. report CoRRabs/1612.03618, Cornell Univ., 2016.
7. B. Morschheuser, J. Hamari, and J. Koivisto, "Gamification in Crowdsourcing: A Review," *Proc. 49th Hawaii Int'l

Conf. System Sciences (HICSS 16), 2016, pp. 4375–4384.
8. G. Zichermann and C. Cunningham, *Gamification by Design: Implementing Game Mechanics in Web and Mobile Apps*, O'Reilly, 2011.
9. D. Lawrie, D. Binkley, and C. Morrell, "Normalizing Source Code Vocabulary," *Proc. 17th Working Conf. Reverse Eng.*, 2010, pp. 3–12.
10. E. Hill et al., "AMAP: Automatically Mining Abbreviation Expansions in Programs to Enhance Software Maintenance Tools," *Proc. 5th Int'l Working Conf. Mining Software Repositories*, 2008, pp. 79–88.
11. G. Sridhara et al., "Towards Automatically Generating Summary Comments for Java Methods," *Proc. 2010 IEEE/ACM Int'l Conf. Automated Software Eng.*, 2010, pp. 43–52.