On-Demand Developer Documentation

Martin P. Robillard*, Andrian Marcus†, Christoph Treude‡,
Gabriele Bavota§, Oscar Chaparro†, Neil Ernst¶, Marco Aurélio Gerosa||, Michael Godfrey**,
Michele Lanza§, Mario Linares-Vásquez††, Gail C. Murphy‡‡, Laura Moreno[x], David Shepherd[xi], and Edmund Wong**

*McGill University, Canada
†The University of Texas at Dallas, USA
‡University of Adelaide, Australia
§Università della Svizzera italiana, Switzerland
¶University of Victoria, Canada
||Northern Arizona University, USA
**University of Waterloo, Canada
††Universidad de los Andes, Colombia
‡‡University of British Columbia, Canada
[x]Colorado State University, USA
[xi]ABB, USA

***Abstract*—We advocate for a paradigm shift in supporting the information needs of developers, centered around the concept of automated *on-demand developer documentation*. Currently, developer information needs are fulfilled by asking experts or consulting documentation. Unfortunately, traditional documentation practices are inefficient because of, among others, the manual nature of its creation and the gap between the creators and consumers. We discuss the major challenges we face in realizing such a paradigm shift, highlight existing research that can be leveraged to this end, and promote opportunities for increased convergence in research on software documentation.**

I. The Vision

We advocate for a new vision for satisfying the information needs of developers, which we call On-Demand Developer Documentation (OD3). Development tasks typically involve a variety of artifacts, tools, processes, and other humans. Currently, when developers have questions, they may consult curated documentation, explore artifacts, browse Questions and Answers (Q&A) websites, or seek the advice of experts. Within this new perspective, an OD3 system would automatically generate high-quality documentation in response to a user query; the OD3 system would use a combination of knowledge extraction techniques on an underlying collection of structured and unstructured artifacts, including source code, issue tracking system metadata, and posts from Q&A forums. For example, a developer assigned to repair a fault related to copy-paste functionality might ask about the implementation of the system's Clipboard feature; in response, the OD3 system might generate a document that explains relevant design decisions for this feature (e.g., based on mining historical project data), and suggest alternatives (e.g., based on processing Q&A forum data). This paper is the outcome of a community effort; in the remainder, we motivate the need for OD3 and then discuss major research challenges that need to be addressed to realize a vision of OD3.

II. Motivation

Documentation pervades many, if not most, software engineering activities [1], [2]. A particular type of documentation, which we call *developer documentation*, is specifically intended to assist software developers in the creation or modification of a system. Common types of developer documentation include source code comments, tutorials and reference documentation for application programming interfaces (APIs), and design documentation. Developer documentation is considered to be one of the most useful pieces of information by developers during software maintentance [1].

Although the ideal of fully self-documented software has been with us since the dawn of the discipline [3], the reality of software development technology and practice falls short. Documentation is an essential resource for creating and maintaining software systems, but it suffers from two fundamental limitations. First, it is costly to create and maintain, and second, it is a non-executable artifact whose presence and correctness are not technically critical to the construction of software. The combination of high cost and low immediate return on investment is particularly nefarious, and reports on documentation being a low priority task are routine [1], [4]. Over the years, tools have been developed to reduce some of the accidental inefficiencies related to the production of documentation. However, documentation tools provide relatively little help with the creation of original content.

Curated documentation can provide coherent and authoritative answers to some classes of questions, but the scope of such documentation is necessarily limited. The field has benefited from many studies of information needs of developers and maintainers [5], [6], [7], from which questions arise that would be hard to document, especially in the absence of a clear promise of return on investment.

The advent of web-based collaboration platforms has created major new opportunities for supporting the creation of, and access to, developer documentation. However, the large quantity and heterogeneity of online resources makes searching, perusing and interpreting the information a daunting task. Crowd-sourced developer documentation may even compound the problems observed with documentation, as the sheer amount of resources available increases the documentation bloat and the authoritativeness of crowd-contributed resources can be difficult to assess.

III. CHALLENGES

Realizing OD3 technology requires advances in three areas. First, advances in *information inference* are needed to derive, find, or link information useful to developers from different types of information elements. Second, developers must be able to form a *document request* based on an incomplete or even incorrect understanding of these needs. Third, an OD3 system must be able to perform *document generation* to produce high-quality documents that address the information needs of developers. Here we use the term *document* to denote a coherent collection of information available to a developer, and not necessarily a "document" meeting any existing documentation standard or intended for archival purposes.

A. Information Inference

Increasing automation in the documentation generation process carries the assumption that the information presented to developers will not already exist in a finished and well-formed state. Instead, it must be inferred or synthesized from other sources of information. We consider that, in an OD3 system, a document can be created from abstractions we will call *documentation elements* that can be represented as a model. In our Clipboard example where a developer is assigned to repair a fault related to the copy-paste feature, documentation elements could include the data structures used by the clipboard, usage protocols for APIs used to access temporary system memory, serialization encodings and the rationale for their selection, etc. We focus this section on the goal of obtaining elements for integration in a document.

1) Establishing Precise Links Between Artifacts: Generating documentation by combining elements requires that these elements can be effectively retrieved. Although classic [8] or specialized [9] information retrieval techniques can play a role in the identification of certain documents, the statistical nature of text-based information retrieval coupled with the technical nature of software engineering artifacts usually means that the results are not always fine-grained and accurate [10]. To create documents on demand in answer to specific questions from developers, advanced techniques are needed that can precisely and accurately link fined-grained elements (e.g., source code functions or paragraphs in documents such as blogs).

Research on *software traceability* [11] is concerned with all aspects of the process of linking high-level artifacts (typically requirements) with low-level ones (typically source code). In the traceability research agenda, it is generally assumed that the recovered traceability links will be inspected for validity by a human analyst. The requirements of an OD3 system make this manual step impossible, which greatly raises the expectations on the precision of the results. *Feature location* [12] can be viewed as a specialization of the general traceability problem to that of establishing links between a high-level feature (concern, or requirement) and the corresponding source code. The major limitation of feature location tools in the context of OD3 systems is that the code retrieved is typically provided without explanations. The goal of OD3 systems is to provide more semantic context than mere location; to explain why the code in question is related to a feature requires an additional level of information integration.

Finally, major advances have also been proposed in the area of *code element resolution*, which covers a number of techniques proposed to resolve a general (typically ambiguous) mention of a potential code element (e.g., a class or a method) to its definition [13], [14], [15], [16]. Resolution techniques have been shown to precisely link code elements mentioned in a variety of contexts, from tutorials to Stack Overflow posts. However, links have little explanatory power in themselves, and identifying elements mentioned in a document is only a first step in the further processing of this document. Recent developments have targeted the qualification of mentions in terms of their explanatory power for an element [17], [18], and used links to validate the accuracy of existing documentations [19], [20]. These two applications are examples of innovations that can help support the development of OD3.

2) Inferring Undocumented Properties: Many important pieces of information that developers need are neither explicitly documented nor easily extractable. These include, in particular, *program properties*, such as usage constraints for components, best practices and design patterns, typestates, and invariants. For example, knowledge of a critical initialization step for the Clipboard might directly answer the developer's question about the copy-paste behavior.

Many techniques have been proposed to automatically infer properties from software components [21]. These techniques, which typically rely on a combination of static and dynamic analysis and data mining, have reached a high level of maturity in terms of technical development. However, the amount of customization effort required to deploy such techniques in realistic contexts limits their practicality. One potential reason is the almost universal problem of false positives generated by data mining techniques applied to software systems. Specifically, inference techniques will infer any property supported by the data, whether it is sensible to a developer or not. Although this problem can be mitigated by various optimizations and filtering heuristics, the essential issue is that applying an inference algorithm, *in general*, provides insufficient context to automatically assess the value of the algorithm's output. The context of an OD3 system can play a major role in mitigating the problem of false positives in program property inference systems, by providing a detailed context for filtering results [22]. By inferring properties in the case of a specific request for documentation, OD3 research has the potential to improve the usability of property inference techniques.

3) Discovering Latent Abstractions and Rationales: Answering complex questions that require an explanation is unlikely to be achieved by the mechanical extraction of facts from software artifacts, such as software dependencies, coverage information, or method pre-conditions. Mature OD3 technology will need to support the production of documentation elements that represent more human-centered constructs, such as *abstractions* and *rationale*.

Although tools already exist that can create models from source code, they either proceed systematically without creating any new abstractions [23], require manual input to specify abstractions and mappings [24], or rely on statistical techniques [25] that produce abstractions that may be difficult to interpret. OD3 provides a new opportunity to create models that target specific developer questions by integrating information from multiple sources. OD3 may also spawn new research directions to capture and evolve descriptions of abstractions [26] as documentation elements that can be integrated into the generation process.

The notion of *explanation* implies that the corresponding documentation will include justification, or *rationale*, for design decisions. The research challenges for design rationale recovery and capture are similar to those of abstractions [27]. As a fuzzy human concept, their recovery is notoriously difficult, and may ultimately require the use of advanced natural language processing techniques. The presence of an OD3 system, however, may one day prove to be a sufficient incentive to motivate the more systematic capture of rationale as some sort of documentation element that, as for abstractions, can be integrated into the generation of documents.

B. Document Request

On-demand developer documentation implies that developers are able to express their demands in such a way that an OD3 system can produce high-quality documentation in response to their information needs. This challenge is exacerbated by the fact that developers might have an incomplete or even incorrect understanding of their needs. In addition, even in scenarios where two developers are working on identical tasks, their information needs are not necessarily identical: they depend on the wider context of the task as well as the on particular background knowledge of the developer. To support developers in expressing their information needs, an OD3 system should also be aware of what information needs it can address and, as much as possible, guide developers towards a successful inquiry.

1) Expressing Information Needs: Many researchers have studied information needs of software developers, for example identifying question types [28] or questions asked in particular scenarios (e.g., [5]). Studies of Q&A websites such as Stack Overflow have categorized different kinds of questions [29] and enumerated their topics [30], among others. Often, information needs of developers are difficult to address, such as questions about intent and rationale [31]. Understanding information needs may provide clues to techniques for enabling developers to express needs; expression may be particularly difficult if developers have an incomplete or even incorrect understanding of the information they need. An OD3 system should understand the technologies that a developer is using along with their documentation and interdependencies, as well as the developer's objective.

2) Capturing Task Context: To understand what a developer is trying to do, an OD3 system can build upon *task context*, defined as the program elements and relationships relevant to completing a particular task, which is formed from the interactions that a developer has with artifacts and the structure of those artifacts [32]. Building on this definition, an OD3 system should be able to understand a developer's task beyond program elements and their relationships alone, taking into account issue tracker information, relevant documentation, and related communication channels, for example. The major challenge lies in the interpretation of the large amounts of data available about a developer's activity, filtering out noise and focusing on relevant information only. The application of natural language processing to artifacts produced by software developers (e.g., method names [33], issue trackers [34], or documentation [35]) appears to be the most promising research direction for capturing and understanding task context.

3) Personalization: Information needs of developers do not depend only on the task that they are working on, but also on their individual background. To differentiate between personal backgrounds of developers, an OD3 system could rely on developer profiles that capture the characteristics of a software developer [36]. Similar to task context and building on a large body of work on automatically identifying developer expertise [37], such information could be captured from developers' interaction with the tools they use, focusing on the information they have seen, technologies they have used, and expertise they have provided, e.g., in communication channels. An OD3 system could learn implicitly from the queries that a developer makes, using queries as a proxy for a knowledge gap of a particular developer, or by explicitly asking developers what they know. An OD3 system would also need to ensure the privacy of developers, e.g., by not revealing potentially embarrassing knowledge gaps to other users of the system.

4) Awareness of Available Information: By being aware of the information available, an OD3 system can guide developers towards queries that are likely to address their information needs. A particular challenge is the vocabulary gap that exists between documentation producers and consumers [38]. To address this challenge, an OD3 system should be aware of domain-specific synonyms [39]. However, synonyms alone are not sufficient to communicate to the user of an OD3 system which queries are likely to succeed or to advise on the best way to phrase a particular query. To support developers in expressing their information needs, structured queries, semantic search [40], indexing of the available documentation [41], and content-sensitive auto-complete interfaces could be employed [35]. However, such approaches would not understand a developer's information need. Accommodating information needs will require a coordinated effort, joining research on information needs with research on automated interpretation of documentation [19], [35].

C. Document Generation

Following a request by a developer, an OD3 system will combine and transform relevant documentation elements into a document. In many cases, the amount of information relevant to a developer request can be staggering. Assembling all this information into a document in its entirety would be impractical and is likely to overwhelm the user. In consequence, the related challenges are selecting the relevant information, determining the abstraction level and amount of information to include in the document, and generating the content and presentation format.

1) Selection: One challenge is to determine which documentation elements from a heterogeneous collection of relevant information should be included in the generation of a requested document. What makes this problem particularly difficult is that the selection criteria depend on the task context and the specific information need of the developer. For example, an experienced developer who needs to use the Clipboard feature in her code may need information about only the most commonly used APIs and examples of their use. Conversely, a less experienced developer, who plans to implement a similar feature in a new application, may need detailed design and implementation information, including dependencies to other parts of the system, and possibly information about external libraries used in the Clipboard implementation. Existing research in generating documentation [42], [43] has focused on context-independent approaches and typically generates generic documentation, based on built-in heuristics. For example, when documenting API uses, some approaches generate abstract usage examples [44], whereas other approaches produce actual usage examples, which are exemplars from clusters of related uses [42]. Selecting the documentation content based on the developer task and context remains an open research issue.

2) Summarization and Synthesis: A second challenge is deciding how many of the selected documentation elements should be included in the documentation. Research on software summarization [45] promises to address this challenge. As before, the developer context presents a problem. Existing research that generates summaries of software artifacts, such as code elements [46], [47], [48], bug descriptions [49], or code changes [50], [43], [51], is also context independent. For example, work on Java class summarization [47] includes public methods in the summaries at the expense of private methods and fields. Reducing textual artifacts, such as discussions or bug descriptions, to their most essential phrases or words relies on decades of research in natural language summarization [52]. Summarizing code or mixed artifacts is substantially more challenging; as yet, little research has been done on determining which code statements best summarize a given code element [48]. Many existing automated software summarization approaches generate extractive summaries [53], [54], which include information directly extracted from the artifacts that are being summarized. A more difficult problem is to generate abstractive summaries, which include information not present in the artifact [55]. For example, if the Clipboard feature is implemented in four classes, the responsibility of each class can be summarized independently from the others. However, the collection of those summaries may not provide a satisfactory answer to the question of how the Clipboard is implemented. To provide a more comprehensive answer, information about the relationships between the four classes and their uses (likely contained in other places) should be also included. It is likely that the answer to most complex developer questions will include both code and natural language elements. Combining them to achieve readable and coherent documents is an active area of research [46], [47], [51].

3) Presentation: Once all the relevant information is selected, prioritized, and summarized, the challenge in generating the documentation relates to the presentation of the information to ensure high-quality documentation. Existing research focused on producing hierarchical documents that reveal information at lower abstraction level on-demand, via interaction with the user [43]. Simple questions can have very complex answers. For example, asking what changed in a software system since the last release can generate a document with hundreds of elements. Presentation is one important way to tackle the documentation complexity.

IV. Conclusion

We exposed our vision of on-demand developer documentation (OD3) as a promising avenue for the fulfillment of developer information needs, and as a means to bring convergence in the current space of research on this area. The research challenges we discussed are by no means the only way to present or discuss current research in this field. However, with this discussion, we aimed to show how many relevant research areas are maturing and becoming increasingly complementary. The time is ripe for a concerted effort.

Acknowledgments. This paper is an outcome of the First International Workshop on Dynamic Software Documentation, held at the Bellairs Research Institute in February 2017.

References

[1] T. C. Lethbridge, J. Singer, and A. Forward, "How software engineers use documentation: the state of the practice," *IEEE Software*, vol. 20, no. 6, pp. 35–39, 2003.

[2] S. C. B. de Souza, N. Anquetil, and K. M. de Oliveira, "A study of the documentation essential to software maintenance," in *Proc. Int'l. Conf. Design of Communication*, 2005, pp. 68–75.

[3] D. E. Knuth, "Literate programming," *The Computer Journal*, vol. 27, no. 2, pp. 97–111, 1984.

[4] G. Uddin and M. P. Robillard, "How API documentation fails," *IEEE Software*, vol. 32, no. 4, pp. 68–75, 2015.

[5] J. Sillito, G. C. Murphy, and K. De Volder, "Questions programmers ask during software evolution tasks," in *Proc. Int'l. Symposium Foundations of Software Eng.*, 2006, pp. 23–34.

[6] B. de Alwis and G. C. Murphy, "Answering conceptual queries with ferret," in *Proc. Int'l. Conf. Software Eng.*, 2008, pp. 21–30.

[7] A. J. Ko, B. A. Myers, M. J. Coblenz, and H. H. Aung, "An exploratory study of how developers seek, relate, and collect relevant information during software maintenance tasks," *IEEE Trans. Software Eng.*, vol. 32, no. 12, pp. 971–987, dec 2006.

[8] C. D. Manning, P. Raghavan, H. Schütze *et al.*, *Introduction to information retrieval*. Cambridge Univ. Press, 2008, no. 1.

[9] A. Marcus and J. I. Maletic, "Recovering documentation-to-source-code traceability links using latent semantic indexing," in *Proc. Int'l. Conf. Software Eng.*, 2003, pp. 125–135.

[10] G. Antoniol, G. Canfora, G. Casazza, A. De Lucia, and E. Merlo, "Recovering traceability links between code and documentation," *IEEE Trans. Software Eng.*, vol. 28, no. 10, pp. 970–983, 2002.

[11] J. Cleland-Huang, O. C. Z. Gotel, J. Huffman Hayes, P. Mäder, and A. Zisman, "Software traceability: Trends and future directions," in *Proc. Future of Software Eng.*, 2014, pp. 55–69.

[12] B. Dit, M. Revelle, M. Gethers, and D. Poshyvanyk, "Feature location in source code: a taxonomy and survey," *J. Software: Evolution and Process*, vol. 25, no. 1, pp. 53–95, 2013.

[13] A. Bacchelli, M. Lanza, and R. Robbes, "Linking e-mails and source code artifacts," in *Proc. Int'l. Conf. Software Eng.*, 2010, pp. 375–384.

[14] B. Dagenais and M. P. Robillard, "Recovering traceability links between an API and its learning resources," in *Proc. Int'l. Conf. Software Eng.*, 2012, pp. 47–57.

[15] P. C. Rigby and M. P. Robillard, "Discovering essential code elements in informal documentation," in *Proc. Int'l. Conf. Software Eng.*, 2013, pp. 832–841.

[16] S. Subramanian, L. Inozemtseva, and R. Holmes, "Live API documentation," in *Proc. Int'l. Conf. Software Eng.*, 2014, pp. 643–652.

[17] G. Petrosyan, M. P. Robillard, and R. De Mori, "Discovering information explaining API types using text classification," in *Proc. Int'l. Conf. Software Eng.*, 2015, pp. 869–879.

[18] H. Jiang, J. Zhang, Z. Ren, and T. Zhang, "An unsupervised approach for discovering relevant tutorial fragments for APIs," in *Proc. Int'l. Conf. Software Eng.*, 2017, pp. 38–48.

[19] B. Dagenais and M. P. Robillard, "Using traceability links to recommend adaptive changes for documentation evolution," *IEEE Trans. Software Eng.*, vol. 40, no. 11, pp. 1126–1146, 2014.

[20] H. Zhong and Z. Su, "Detecting API documentation errors," in *Proc. ACM SIGPLAN Int'l. Conf.Object-Oriented Programming Systems Languages and Applications*, 2013, pp. 803–816.

[21] M. P. Robillard, E. Bodden, D. Kawrykow, M. Mezini, and T. Ratchford, "Automated API property inference techniques," *IEEE Trans. Software Eng.*, vol. 39, no. 5, pp. 613–637, 2013.

[22] M. Bruch, M. Monperrus, and M. Mezini, "Learning from examples to improve code completion systems," in *Proc. 7th Joint Meeting European Software Eng. Conf. ACM SIGSOFT Symposium Foundations of Software Eng.*, 2009, pp. 213–222.

[23] D. Jackson and A. Waingold, "Lightweight extraction of object models from bytecode," *IEEE Trans. Software Eng.*, vol. 27, no. 2, pp. 156–169, 2001.

[24] G. C. Murphy, D. Notkin, and K. J. Sullivan, "Software reflexion models: Bridging the gap between design and implementation," *IEEE Trans. Software Eng.*, vol. 27, no. 4, pp. 364–380, 2001.

[25] E. Linstead, P. Rigor, S. Bajracharya, C. Lopes, and P. Baldi, "Mining concepts from code with probabilistic topic models," in *Proc. Int'l. Conf. Automated Software Eng.*, 2007, pp. 461–464.

[26] M. P. Robillard and G. C. Murphy, "Representing concerns in source code," *ACM Trans. Software Eng. and Methodology*, vol. 16, no. 1, pp. 1–38, 2007.

[27] W. C. Regli, X. Hu, M. Atwood, and W. Sun, "A survey of design rationale systems: approaches, representation, capture and retrieval," *Engineering with computers*, vol. 16, no. 3, pp. 209–235, 2000.

[28] S. Letovsky, "Cognitive processes in program comprehension," in *Papers Presented at the 1st Workshop on Empirical Studies of Programmers on Empirical Studies of Programmers*, 1986, pp. 58–79.

[29] E. C. Campos and M. de Almeida Maia, "Automatic categorization of questions from Q&A sites," in *Proc. Symposium on Applied Computing*, 2014, pp. 641–643.

[30] A. Barua, S. W. Thomas, and A. E. Hassan, "What are developers talking about? an analysis of topics and trends in stack overflow," *Empirical Software Eng.*, vol. 19, no. 3, pp. 619–654, 2014.

[31] T. D. LaToza and B. A. Myers, "Hard-to-answer questions about code," in *Workshop on Evaluation and Usability of Programming Languages and Tools*, 2010, pp. 8:1–8:6.

[32] M. Kersten and G. C. Murphy, "Using task context to improve programmer productivity," in *Proc. Int'l. Symposium Foundations of Software Eng.*, 2006, pp. 1–11.

[33] E. Hill, L. Pollock, and K. Vijay-Shanker, "Improving source code search with natural language phrasal representations of method signatures," in *Proc. Int'l. Conf. Automated Software Eng.*, 2011, pp. 524–527.

[34] D. Shepherd, Z. P. Fry, E. Hill, L. Pollock, and K. Vijay-Shanker, "Using natural language program analysis to locate and understand action-oriented concerns," in *Proc. Int'l. Conf. Aspect-oriented Software Development*, 2007, pp. 212–224.

[35] C. Treude, M. P. Robillard, and B. Dagenais, "Extracting development tasks to navigate software documentation," *IEEE Trans. Software Eng.*, vol. 41, no. 6, pp. 565–581, 2015.

[36] A. T. Ying and M. P. Robillard, "Developer profiles for recommendation systems," in *Recommendation Systems in Software Eng.*, 2014, pp. 199–222.

[37] A. Mockus and J. D. Herbsleb, "Expertise browser: A quantitative approach to identifying expertise," in *Proc. Int'l. Conf. Software Eng.*, 2002, pp. 503–512.

[38] P. Mika, E. Meij, and H. Zaragoza, "Investigating the semantic gap through query log analysis," *Int'l. Semantic Web Conf.*, pp. 441–455, 2009.

[39] C. Chen, Z. Xing, and X. Wang, "Unsupervised software-specific morphological forms inference from informal discussions," in *Proc. Int'l. Conf. Software Eng.*, 2017, pp. 450–461.

[40] S. Gottipati, D. Lo, and J. Jiang, "Finding relevant answers in software forums," in *Proc. Int'l. Conf. Automated Software Eng.*, 2011, pp. 323–332.

[41] A. Tang, P. Liang, and H. v. Vliet, "Software architecture documentation: The road ahead," in *Proc. Working IEEE/IFIP Conf. Software Architecture*, 2011, pp. 252–255.

[42] L. Moreno, G. Bavota, M. Di Penta, R. Oliveto, and A. Marcus, "How can I use this method?" in *Proc. Int'l. Conf. Software Eng.*, 2015, pp. 880–890.

[43] L. Moreno, G. Bavota, M. Di Penta, R. Oliveto, A. Marcus, and G. Canfora, "ARENA: an approach for the automated generation of release notes," *IEEE Trans. Software Eng.*, vol. 43, no. 2, pp. 106–127, 2017.

[44] H. Zhong, T. Xie, L. Zhang, J. Pei, and H. Mei, "MAPO: mining and recommending API usage patterns," in *Proc. European Conf. Object-Oriented Programming*, 2009, pp. 318–343.

[45] L. Moreno, "Software documentation through automatic summarization of source code artifacts," Ph.D. dissertation, The University of Texas at Dallas, 2016.

[46] G. Sridhara, E. Hill, D. Muppaneni, L. Pollock, and K. Vijay-Shanker, "Towards automatically generating summary comments for Java methods," in *Proc. Int'l. Conf. Automated Software Eng.*, 2010, pp. 43–52.

[47] L. Moreno, J. Aponte, G. Sridhara, A. Marcus, L. Pollock, and K. Vijay-Shanker, "Automatic generation of natural language summaries for Java classes," in *Int'l. Conf. Program Comprehension*, 2013, pp. 23–32.

[48] A. T. Ying and M. P. Robillard, "Selection and presentation practices for code example summarization," in *Proc. Int'l. Symposium Foundations of Software Eng.*, 2014, pp. 460–471.

[49] S. Rastkar, G. C. Murphy, and G. Murray, "Automatic summarization of bug reports," *IEEE Trans. Software Eng.*, vol. 40, no. 4, pp. 366–380, 2014.

[50] R. P. L. Buse and W. Weimer, "Automatically documenting program changes," in *Proc. Int'l. Conf. Automated Software Eng.*, 2010, pp. 33–42.

[51] L. F. Cortes-Coy, M. Linares-Vásquez, J. Aponte, and D. Poshyvanyk, "On automatically generating commit messages via summarization of source code changes," in *Int'l. Working Conf. on Source Code Analysis and Manipulation*, 2014, pp. 275–284.

[52] K. Spärck Jones, "Automatic summarising: The state of the art," *Inf. Process. Manage.*, vol. 43, no. 6, pp. 1449–1481, 2007.

[53] P. Rodeghero, S. Jiang, A. Armaly, and C. McMillan, "Detecting user story information in developer-client conversations to generate extractive summaries," in *Proc. Int'l. Conf. Software Eng.*, 2017, pp. 49–59.

[54] S. Haiduc, J. Aponte, L. Moreno, and A. Marcus, "On the use of automated text summarization techniques for summarizing source code," in *Working Conf. Reverse Eng.*, 2010, pp. 35–44.

[55] P. W. McBurney and C. McMillan, "Automatic source code summarization of context for Java methods," *IEEE Trans. Software Eng.*, vol. 42, no. 2, pp. 103–119, 2016.