Available online at www.ijpe-online.com

International Journal of Performability Engineering

vol. 15, no. 10, October 2019, pp. 2718-2725
DOI: 10.23940/ijpe.19.10.p18.27182725

A Context Model for Code and API Recommendation Systems based on Programming Onsite Data

Zhiyi Zhang[a,b,c,*], Chuanqi Tao[a,b,c], Wenhua Yang[a,b], Yuqian Zhou[a,b,d], and Zhiqiu Huang[a,b]

[a]*Collage of Computer Science and Technology, Nanjing University of Aeronautics and Astronautics China, Nanjing, 211100, China*
[b]*Key Laboratory of Safety-Critical Software, Ministry of Industry and Information Technology, Nanjing, 211100, China*
[c]*State Key Laboratory for Novel Software Technology, Nanjing University, Nanjing, 210023, China*
[d]*State Key Laboratory of Cryptology, Beijing, 100878, China*

Abstract

Code and application programming interface (API) recommendation systems are important guarantees for efficient and accurate code reuse to improve the efficiency of software development. Context data plays a key role in code and API recommendation. A large amount of programming onsite data has been generated, but existing code and API recommendation systems rarely consider the context based on programming onsite data, which leads to low efficiency and poor accuracy of code and API recommendation. In this paper, we proposed a context model for code and API recommendation systems. Our context model is based on programming onsite data collected during programming. It includes four aspects: developer, project, time, and environment. Developer data is labeled data abstracted from information according to developers' programming habits and abilities, project data is information about the project, time data is information about temporal aspects of developers interacting with the project, and environment data is all environment elements used by developers during programming. Then, we collected programming onsite data in three ways: explicit collection, implicit collection, and reasoning. Lastly, we built the context model using a coarse-grained abstract model for recommendation. Our context model retains the key information in the code while eliminating redundant information that may affect the accuracy of the recommend task, and it can theoretically improve the efficiency and accuracy of recommendation.

Keywords: context model; code and API recommendation; programming onsite data

(Submitted on August 10, 2019; Revised on October 2, 2019; Accepted on October 25, 2019)

1. Introduction

In the field of software engineering, the efficiency of software development is a core concern in both industry and academia. Code reuse is an important way to improve the efficiency of software development. Both inexperienced and skilled programmers hope to find existing codes that meet their needs, which can help them reduce development time costs. Code and application programming interface (API) recommendation systems, which have been used in a wide variety of domains, are important guarantees for efficient and accurate code reuse. They can help programmers overcome information overload and find relevant codes and APIs to queries proposed by programmers [1]. To improve the accuracy of recommendation results, recommendation approaches commonly use context to supplement query conditions of programmers. From an operational perspective, context is defined as an aggregate of various categories that describe the setting in which a recommender is deployed [2], such as existing codes, current activity, and so on.

In recent years, as the scale and complexity of software systems have increased immensely, a large amount of programming onsite data has been generated, and it can be used to improve software productivity and quality. Programming onsite data is a collection of data generated from onsite programming during software development, including software tasks, testing report data, and source codes. It is of great significance to correctly understand, present, and analyze the process of software development, improve software development methods, and liberate us from heavy software development work. However, most current recommendation systems rarely consider the context based on programming

* Corresponding author.
E-mail address: zyzhang10@nuaa.edu.cn

onsite data, such as developer programming style and programming onsite behavior, which leads to low efficiency and poor accuracy. Thus, it is difficult for traditional code and API recommendation approaches to provide effective support for software development.

To alleviate this problem, in this paper, we proposed a context model for code and API recommendation systems. Our context model is based on programming onsite data collected during programming. It includes four aspects, which are developer, project, time, and environment. Then, we mined implicit inter-relationships between contexts and researched onsite dynamic correlation and knowledge representation technologies between contexts, such as developers, recommended tasks, and historical information. Based on these data, developers could construct a context model for recommendation, which could be used to improve the accuracy of code and API recommendation, provide customized recommendation services, and enrich semantic information recommendation oriented query conditions. For our context model, we firstly determined data categories that should be considered during code and API recommendation. Then, we collected programming onsite data in different ways. Lastly, we built the context model for recommendation. Our context model can theoretically improve the efficiency and accuracy of recommendations.

Our paper is structured as follows. Section 2 discusses the context data in detail. Section 3 proposes our data collection methods. Section 4 shows how the context model is built. Section 5 presents some related works. Finally we conclude in Section 6.

2. Context Data

In recommendation systems, a context model is a set of contextual factors that characterize a situation [3]. For code and API recommendation, since the set of all programming onsite data is excessively large [4], it is very expensive to collect all these data and use them for code and API recommendation. Moreover, some programming onsite data may be difficult to acquire [5], and some programming onsite data are not informative for query conditions of developers [6].

Since recommendation results are used for programming, project information, such as project function and project complexity, should be collected. Since different developers have different programming styles and characters, we should also collect information about developers. Moreover, software development has the characteristics of cross-time and cross-regional, so we also collect the programming time and environment information. In summary, we collect four categories of programming onsite data, which are developers, project, time, and environment.

2.1. Developer Data

In recommendation systems, the developer plays a very important role. The significant attributes and programming habits of developers may greatly influence the selection of recommendation results. For example, there are many ways to read file content in *Java*, such as by byte, character, line, or random reading. One developer may choose to read file content by character, but recommendation systems recommend codes of reading file content by line. Although recommended codes could also implement the same function, developers may not select these codes because they do not meet their programming habits. Another example is that there are many programming languages, and developers may only be familiar with some of them. If one developer is familiar with *C* but unfamiliar with *Java*, but the recommendation results are *Java* codes, it is clear that this developer will not accept these results. Moreover, in the programming process, developers' experiences also influence the selection of recommendation results. An expert may select different recommendation results than a novice would.

Developer data is labeled data abstracted from information according to developers' programming habits and abilities. These labeled data can clearly identify the significant attributes of the developers, which makes the recognition, analysis, and measurement of developers more effective. As shown in Table 1, our developer data is divided into two categories, the developer onsite data and the developer history data. Developer onsite data responds to the developers' behaviors and perception of current development projects, including data about developer programming behavior, current integrated development environment (IDE) familiarity, and current project familiarity. The developer programming behavior factor is what a developer is doing in this project. For example, which codes are written by this developer? In our work, we divide the programmers' behaviors into six categories: coding, debugging, using version control, testing performed outside the IDE, reviewing code, and others [7]. The current IDE familiarity factor is the level of IDE experience of a developer. There are many programming tools, and one developer may be familiar with Eclipse but unfamiliar with NetBeans. According to [8], we distinguish the current IDE familiarity factor as experts and novices. We use the average number of different IDE commands used by a developer as a threshold. The more different IDE commands he or she uses, the more skilled he or she is with this IDE. The current project familiarity factor is the level of a specific project experience of a developer. Similar to the current IDE familiarity factor, we also distinguish the current project familiarity factor as experts and novices. If one

developer has worked on five or more projects, which are similar to the project he or she is working on now, we consider this developer as an expert; otherwise, we consider him or her as a novice.

Table 1. Developer data

Developer classification	Data information
Onsite data	Developer programming behavior
	Current IDE familiarity
	Current project familiarity
History data	Developer experience
	Developer programming habit
	Programming social network
	Developer IDE experience

History data could provide information about developers' programming habits and experience. It includes data about developer experience, developer programming habits, developer social networks, and developer IDE experience. The developer experience factor describes the level of overall software development experience of the developer. We adopt the evaluation method from [9]. If a developer has five or more years of professional programming experience, we classify him or her as an expert, while between two and five years is competent, and less than two years is novice. The developer programming habit factor reflects the developer's behavioral tendencies when programming. For example, some developers may like to use "*goto*" statements while others do not. The developer social network factor focuses on the concern network and the cooperation network. The concern network refers to the social network formed by the relationship between developers in the open source community. The relationship in the network reflects the degree of concern of one developer to another developer. The cooperation network refers to the social connections between developers generated in the process of the project. The analysis of social relations can reflect the recognition degree of a developer and the participation degree of the developer in the development process, thus reflecting the programming ability of the developer. The developer IDE experience is similar to the current IDE familiarity. It can reflect the developer proficiency in various IDEs.

2.2. Project Data

Project data is information about the project, such as the function, construction, and artifacts. It is the most diverse in data collection. As shown in Table 2, it also contains onsite data and history data.

Table 2. Project data

Onsite project classification	Data classification	Data information
Onsite data	Current status of the project	Current using command
		Current running module
		Call type by developer
		Relevant labels and windows
		Variable description
		Method description
		Method call
		Project structure
	Task information	Task type
		Query of keywords and codes
		Task characteristics
		Programming error recommendation
	Other information	Context priority
History data	Project information	Project description
		Item category
		Project model
		Code dependencies
		Project complexity
		Item length
		Solved code defects
		History recommendation information

Onsite data includes the current status of the project, task information, and other information. The current status of the project is the collection of current state information for the entire software project, from which we could obtain explicit or implicit associations between the context. It includes eight factors, which include the current using command and current running module. These factors have long been recognized as relevant context in software engineering [10-11]. Task information is the collection of programmers' codes or API query tasks, including task type, query of keywords and codes, task characteristics, and programming error recommendation. Through these data, we could find code and API recommendation tasks that the programmer may need and then recommend codes and APIs by combing developer data.

Moreover, in the programming onsite context, we propose the concept of context priority information. In general, we consider that the more contextual data there is, the more accurate the recommended codes and APIs are, but the longer time the recommendation takes. Due to the real-time requirements of onsite programming, developers may artificially choose some contextual data. In order to improve the recommendation efficiency and meet the query of the developer, the developer may select some context data with higher priority and remove the low priority context data that has less influence on the recommendation results.

In addition, although code and API recommendation is used for onsite programming, in order to mine the hidden conditions of recommendation task and improve the accuracy of recommendation, we still need to collect some programming history data. We collect the project description, item category, project model, code dependencies, project complexity, item length, solved code defects, and history recommendation information in our context model. When considering these data, the recommended accuracy may be further improved. For example, during onsite programming, if a developer has searched for code recommendations in a previous historical version and conducted a similar code and API search in the current project version, codes and APIs that are similar to the previous results could be recommended based on historical recommendations. Since these codes and APIs have been accepted by the developer in a previous search, they also have high probability to be accepted in this query.

2.3. Time and Environment Data

Time data is the information about temporal aspects of developers interacting with the project. Three factors, time information, project version number, and last modified time of the project, are collected as the *Time* context data in Table 3. From these data, we can find when the project is being performed or modified, which could improve the efficiency and accuracy of recommendations. For example, since modifications at different times have different bugginess, codes and APIs committed between 7 AM and noon could be recommended because they are less buggy [12].

Table 3. Time and environment data

Programming data classification	Data information
Programming time	Time information (year, month and day)
	Project version number
	Last modified time of the project
Programming environment	Programming location
	Project platform IDE
	Interface elements used by developers
	Interface elements that developers care about

As shown in Table 3, *environment* data is all environment elements used by developers during programming. The programming location represents the location where developers are programming. The project platform IDE shows the IDE tool that developers are using. Interface elements used by developers and interface elements that developers care about reflect which elements a developer may interact with. These data can provide additional related information for recommendation. For example, when programming in *Eclipse*, the project explorer user interface displays the hierarchical structure of the project, from which we could capture the information about the project structure.

3. Data Collection

We used three methods, explicit collection, implicit collection, and reasoning, to collect data in our context. For the onsite data and some historical data of developers, such as the developers' project experience and IDE experience, we collect these data explicitly. Developers submit basic personal information and participation information in the software development. For example, we could use the questionnaire to query developer data. For developers' programming habits and programming social networks, we use implicit collection methods to collect these data. We determine the developer's programming style by analyzing his or her previous code documentation and bug reports. Meanwhile, we crawl the developer's recorded data in the programming forum through the crawler and use social relationship analysis techniques to reason the developer's programming social network.

In the collection of project onsite information, some data information, such as current running modules and current using commands, are captured implicitly by methods including screen monitoring and mouse operation monitoring. For project structure and call method information, we analyze the project requirements, design documents, and code structure to infer relationships between parameters and methods. At the same time, we obtain the task data according to the developer's input, and the task data contains passive recommendation tasks with query inputs and active recommendation tasks without query inputs. Then, we extract the key features of the query inputs through text-based processing and analysis. Moreover, if

the project has historical data, we obtain this information by explicitly collecting the documentation of the software, including development documents, log reports, and so on.

For the programming time, we obtain these data through explicit ways of communicating with developers or document queries. For the programming environment, we collect relevant information through implicit methods such as screen monitoring and mouse operation monitoring.

Moreover, we build an acquisition server to collect all kinds of collected data. When the network is abnormal or congested, the redundant data to be collected will be saved to the card terminal and become historical data. Since some data may miss during the collecting, we used the K-nearest neighbor filling algorithm to fill these missing data, which could solve the random missing problem in incomplete data. There are two steps in our methods. In first step, we will cluster initial data, and then use K-nearest neighbor filling algorithm to fill the incomplete data and form a complete data set. In the last step, when we obtain new data, we cluster all data and update the clustering results, then based on the basis of the new clustering results, the K-nearest neighbor filling algorithm is applied to fill the missing values. According to the continuous alternate clustering and filling work, we make full use of complete data and incomplete data information, to get all of context data used for code&API recommendation.

4. Model Building

Context data can be captured from many sources, such as codes and developers. For different kinds of contexts, different context models need to be defined for representation and abstraction, and different technologies can be used in this process for different contexts. For example, when building a code model, developers could refer to the syntax tree analysis technique in program analysis. Since codes have good logical structures, they could be modeled using a model similar to the abstract syntax tree, but the abstraction granularity is too fine, which makes the matching too cumbersome when recommending similar codes. In order to avoid this problem, we use a coarse-grained abstract model when building the code model, which retains the key information in the code while eliminating redundant information that may affect the accuracy of the recommend task. For each code, the coarse-grained abstract model could extract the statement declaration, class creation, function call, conditional statement declaration, and other information in the source code. We will do a static analysis of the source code based on these key information. During the static analysis, we can complete source code data flow analysis, variable dependency analysis, code slicing analysis, and so on. In our method, the function in the source code is taken as a unit when doing code slicing analysis, and we could obtain code slicing information of different variables. Each slice information contains usage pattern information of a specific variable or a group of variables, which is the basis for the recommend.

For code slice information, we use code abstract to represent them. In the abstract representation, in order to a uniform code abstract representation, concrete variable names are removed from the code, all statement variables are abstracted with class names, and classes of code are further abstracted according to code inheritance and interface relationship. In order to complete the pattern extraction of code, we need to quantify the code on the basis of code abstraction. Based on the result of code abstraction, we assign different number values to each class name and conditional statement, to represent data discretely. After the discrete numerical representation of the statement is completed, each slice statement can be converted into a vector, and each dimension of the vector represents the number value of a function call, class creation and conditional statement. In the vector representation of slice statements, the vector is used for pattern mining, so the order between different dimensions of each vector has a certain significance, which represents the order of invocation of different statements.

When building a developer programming style model, different developers have different development habits and knowledge backgrounds during software development, and we can capture these data through feature extraction and learning techniques. For example, a developer's habits could be learned from his or her participation projects and codes written by him or her in the open source community, then these data could be used to provide customized code and API recommendation. When building relationships between entities such as code and files, we could use program analysis and relational extraction techniques. The calling relationship between codes and the association between files are also important context in the onsite programming, and we could capture these data through relationship analysis and extraction techniques and then abstract and express them for recommendation.

Specifically speaking, developer data information can be presented in a variety of ways, such as plain text labels or specific numerical values. In our method, we plan to use a hybrid representation method, a combination of semantic text features and quantitative numerical features, to present developer data information. For example, when representing a developer's familiarity with the project, we scan specify this feature as < commit, 100> for the developer. The "*commit*" represents the number of codes submitted by the developer, while "*100*" is used to identify a total of 100 code submissions by the developer. By using this combination of text and numerical, we can visually and quantitatively express the

characteristic of a developer that familiarity with the project.

In previous work, developer data can be organized by flattening or structured method. Our method plans to adopt a structured multi-level organizational form, that is, we set multidimensional attributes at the first level, and each dimension includes more finer-grained information. For example, we can specify the dimensions of "development activity" and "community activity" in the first level. Furthermore, "development activity" can include data information such as "number of commits submitted", "number of lines of code submitted", and so on. And "community activity" can also include data information such as "number of requests submitted", "number of comments published", and so on. This structured multi-tier organization is convenient for building the model of the developer's data in multi-dimension and different granularity.

Raw context data could be transformed into a concrete model representation using the above techniques. Based on these programming onsite behavior data and developer information data, by mining implicit context information and researching the association and knowledge representation technology between onsite context elements such as developers, recommendation tasks, and historical information, we can construct a context model for code and API recommendation. This model could be utilized to improve the recommendation accuracy of codes and APIs, provide customized recommendations, and enrich the semantic information for the recommended query condition.

5. Related Works

Although research on code and API recommendation mainly focuses on recommendation methods, there are still some works on context models that are used for code and API recommendation. Holmes et al. described an approach for locating relevant codes in an example repository [13]. This approach takes structure as context data and is based on heuristically matching the structure of the code under development to the example code. To increase the possibility that recommendations will be accepted, Fogarty et al. used task engagement as an indicator of human interruptibility [14]. They observed how developers interact with the IDE when programming and discovered that if the developers performed certain editing work, they were less interruptible. Kersten et al. presented a new task context model that could reduce information overload [15]. This model focuses on the developers' work by filtering and ranking the information presented by the development environment, and it was created by monitoring the developers' activity and extracting the structural relationships of program artifacts. DebugAdvisor allowed programmers to search codes using a fat query [16]. This query is a context of a bug and includes all the information a programmer has about the bug, such as natural language text, debugger output, and so on.

Moreover, Sumner et al. proposed a technique that could encode the current calling context at any point during execution and leverage the stack depth to remove unnecessary encoding [17]. It encodes an acyclic call path into a number and divides a recursive path into subsequences to encode them independently. Murphy-Hill et al. found that while recommendation is feasible to automatically recommend commands to developers based on their usage history, using patterns of past discovery is a useful way during code and API recommendation [8]. Danylenko et al. proposed a recommender system that could suggest the best-fit component variants for certain actual contexts [18]. These contexts were used by a composition technique to improve application runtime performance. Janjic et al. adapted the ranking of recommendation results by considering additional metrics, such as lines of code and cyclomatic complexity [19].

Zolaktaf proposed a novel algorithm called CoDis that could be deployed in a user developer community to recommend commands [20]. CoDis generates personalized recommendations for a developer by analyzing his or her command usage history, command discovery and co-occurrence within the entire community, and the elapsed time between the developer's last activity and the time of recommendation. Gasparic et al. statistically evaluated the correlations between IDE command usage and different situations and discovered that the contextual factors included in the model statistically correlated with the usage of IDE commands [3]. Moreover, they took into account the contexts in which a developer worked, and different commands were usually executed. They presented a novel IDE command recommendation algorithm to provide relevant recommendations [21]. Chattopadhyay et al. observed that a programming context crosscut activities and artifacts [22].

Heinemann et al. used the code edited within an IDE as development context during recommendation [23]. Wu et al. divided the context into five groups, which were query, statement, method body, class body and project artifact, and used them for code recommendation in Android development [24-25]. Nguyen et al. found fine-grained code changes by statistical learning and then determined the likelihood of a developer inserting an API method call on the recommendation point [26]. McBurney et al. summarized the context surrounding of a method, rather than details of the internals of the method, during code recommendation. They compared their generated summaries to summaries written manually by experts and a state-of-the-art automatic summarization tool. The experiment results showed that their approach did not reach the quality of human-written summaries, but it improved the state-of-the-art summarization tool [27]. Ponzanelli et al. presented

a holistic recommender system called Libra to help analyze developers' semantic relationships by constructing a holistic meta-information model of developers [28]. Bing Developer Assistant (BDA) could automatically detect the APIs under editing in the current programming context via the IntelliSense features of IDEs, and it could recommend sample codes mined from public software repositories and web pages [29].

Recently, most of the existing context models used for code and API recommendation only focus on the context from code, and they rarely consider the context from developers, time, and the environment. In our model, we divided the context into four aspects. For each aspect, we collected the context in three ways. Moreover, we built our model using a coarse-grained abstract model. All of these can help developers improve the efficiency and accuracy of recommendation.

6. Conclusions

In this paper, we proposed a context model for code and API recommendation systems. Our context model is based on programming onsite data collected during programming. It includes four aspects: *developer*, *project*, *time*, and *environment*. Developer data is labeled data abstracted from information according developers' programming habits and abilities, project data is information about the project, time data is information about temporal aspects of developers interacting with the project, and environment data is all environment elements used by developers during programming. Then, we collected programming onsite data in three ways, which were explicit collection, implicit collection, and reasoning. We explicitly collected some data, such as onsite data and historical data of developers. Some data project onsite information were captured implicitly. The relationship between some data was obtained by reasoning. Lastly, we built the context model using a coarse-grained abstract model for recommendation. Our context model retains the key information in the code while eliminating redundant information that may affect the accuracy of the recommend task, and it can theoretically improve the efficiency and accuracy of recommendation.

Acknowledgements

This research is supported in part by the National Key Research and Development Program of China (No. 2018YFB1003902), Key Laboratory of Safety-Critical Software at Nanjing University of Aeronautics and Astronautics, Ministry of Industry and Information Technology (No. XCA17007-04), Open Fund of the State Key Laboratory for Novel Software Technology (No. KFKT2019B11), Fundamental Research Funds for the Central Universities (No. 3082018NS2018056), National Natural Science Foundation of China (No. 61402229, 61602267, and 61901218), and Open Fund of the State Key Laboratory for Novel Software Technology (No. KFKT2018B19).

References

1. B. Antunes, J. Cordeiro, and P. Gomes, “An Approach to Context-based Recommendation in Software Development,” in *Proceedings of the 6th ACM Conference on Recommender Systems*, pp. 171-178, Dublin, Ireland, September 2012
2. K. Verbert, N. Manouselis, X. Ochoa, M. Wolpers, H. Drachsler, I. Bosnic, et al., “Context-Aware Recommender Systems for Learning: A Survey and Future Challenges,” *IEEE Transactions on Learning Technologies*, Vol. 5, No. 4, pp. 318-335, April 2012
3. M. Gasparic, G. C. Murphy, and F. Ricci, “A Context Model for IDE-based Recommendation Systems,” *Journal of Systems and Software*, Vol. 128, pp. 200-219, June 2017
4. A. K. Dey, “Context-Aware Computing,” *Ubiquitous Computing Fundamentals*, *Chapman and Hall/CRC*, pp. 335-366, 2018
5. W. Maalej, T. Fritz, and R. Robbes, “Collecting and Processing Interaction Data for Recommendation Systems,” *Recommendation Systems in Software Engineering*, Springer, pp. 173-197, 2014
6. A. Odic, M. Tkalcic, J. F. Tasic, and A. Kosir, “Relevant Context in a Movie Recommender System: Users Opinion vs. Statistical Detection,” *ACM RecSys*, Vol. 12, September 2012
7. A. N. Meyer, T. Fritz, G. C. Murphy, and T. Zimmermann, “Software Developers' Perceptions of Productivity,” in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pp. 19-29, Hong Kong, China, November 2014
8. E. Murphy-Hill, R. Jiresal, and G. C. Murphy, “Improving Software Developers' Fluency by Recommending Development Environment Commands,” in *Proceedings of the 20th ACM SIGSOFT International Symposium on the Foundations of Software Engineering*, pp. 42, Chapel Hill, USA, November 2012
9. J. Sillito, G. C. Murphy, and K. De Volder, “Asking and Answering Questions During a Programming Change Task,” *IEEE Transactions on Software Engineering*, Vol. 34, No. 4, pp. 434-451, April 2008
10. R. DeLine, M. Czerwinski, and G. Robertson, “Easing Program Comprehension by Sharing Navigation Data,” in *Proceedings of the 2005 IEEE Symposium on Visual Languages and Human-Centric Computing*, pp. 241-248, Dallas, USA, September 2005
11. M. P. Robillard, “Automatic Generation of Suggestions for Program Investigation,” *ACM SIGSOFT Software Engineering Notes*, Vol. 30, No. 5, pp. 11-20, ACM, September 2005
12. J. Eyolfson, L. Tan, and P. Lam, “Do Time of Day and Developer Experience Affect Commit Bugginess?” in *Proceedings of*

the 8th Working Conference on Mining Software Repositories, pp. 153-162, Honolulu, USA, May 2011
13. R. Holmes and G. C. Murphy, "Using Structural Context to Recommend Source Code Examples," in *Proceeding of the 27th International Conference on Software Engineering*, pp. 117-125, St. Louis, USA, May 2005
14. J. Fogarty, A. J. Ko, H. H. Aung, E. Golden, K. P. Tang, and S. E. Hudson, "Examining Task Engagement in Sensor-based Statistical Models of Human Interruptibility," in *Proceedings of the 2005 SIGCHI Conference on Human Factors in Computing Systems*, pp. 331-340, Portland, USA, April 2005
15. M. Kersten and G. C. Murphy, "Using Task Context to Improve Programmer Productivity," in *Proceedings of the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pp. 1-11, Portland, USA, November 2006
16. B. Ashok, J. Joy, H. Liang, S. K. Rajamani, G. Srinivasa, and V. Vangala, "Debugadvisor: A Recommender System for Debugging," in *Proceedings of the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on Foundations of Software Engineering*, pp. 373-382, Amsterdam, Netherlands, August 2009
17. W. N. Sumner, Y. Zheng, D. Weeratunge, and X. Zhang, "Precise Calling Context Encoding," *IEEE Transactions on Software Engineering*, Vol. 38, No. 5, pp. 1160-1177, July 2011
18. A. Danylenko and W. Lowe, "Context-Aware Recommender Systems for Non-Functional Requirements," in *Proceedings of the 3rd International Workshop on Recommendation Systems for Software Engineering*, pp. 80-84, Zurich, Switzerland, June 2012
19. W. Janjic, O. Hummel, and C. Atkinson, "Reuse-Oriented Code Recommendation Systems," *Recommendation Systems in Software Engineering*, pp. 359-386, Springer, 2014
20. S. Zolaktaf and G. C. Murphy, "What to Learn Next: Recommending Commands in a Feature-Rich Environment," in *Proceedings of IEEE 14th International Conference on Machine Learning and Applications*, pp. 1038-1044, Miami, USA, December 2015
21. M. Gasparic, T. Gurbanov, and F. Ricci, "Context-Aware Integrated Development Environment Command Recommender Systems," in *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering*, pp. 688–693, Urbana-Champaign, USA, October 2017
22. S. Chattopadhyay, N. Nelson, T. Nam, M. Calvert, and A. Sarma, "Context in Programming: An Investigation of How Programmers Create Context," in *Proceedings of 11th IEEE/ACM International Workshop on Cooperative and Human Aspects of Software Engineering*, pp. 33-36, Gothenburg, Sweden, May 2018
23. L. Heinemann, V. Bauer, M. Herrmannsdoerfer, and B. Humme, "Identifier-based Context-Dependent API Method Recommendation," in *Proceedings of 16th European Conference on Software Maintenance and Reengineering*, pp. 31-40, Szeged, Hungary, March 2012
24. J. Wu, L. Shen, W. Guo, and W. Zhao, "Code Recommendation for Android Development: How does it Work and What can be Improved?" *Science China Information Sciences*, Vol. 60, No. 9, pp. 1-14, July 2017
25. J. Wu, L. Shen, W. Guo, and W. Zhao, "How is Code Recommendation Applied in Android Development: A Qualitative Review," in *Proceedings of 2016 International Conference on Software Analysis, Testing and Evolution*, pp. 30-35, Kunming, China, November 2016
26. A. T. Nguyen, M. Hilton, M. Codoban, H. A. Nguyen, L. Mast, E. Rademacher, et al., "API Code Recommendation using Statistical Learning from Fine-Grained Changes, " in *Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pp. 511-522, Seattle, USA, November 2016
27. P. W. McBurney and C. McMillan, "Automatic Source Code Summarization of Context for Java Methods," *IEEE Transactions on Software Engineering*, Vol. 42, No. 2, pp. 103-119, August 2015
28. L. Ponzanelli, S. Scalabrino, G. Bavota, A. Mocci, R. Oliveto, D. P. Massimiliano, et al., "Supporting Software Developers with a Holistic Recommender System," in *Proceedings of IEEE/ACM 39th International Conference on Software Engineering*, pp. 94-105, Buenos Aires, Argentina, May 2017
29. H. Zhang, A. Jain, G. Khandelwal, C. Kaushik, S. Ge, and W. Hu, "Bing Developer Assistant: Improving Developer Productivity by Recommending Sample Code," in *Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pp. 956-961, Seattle, USA, November 2016