

FeaRS: Recommending Complete Android Method Implementations

Fengcai Wen, Valentina Ferrari, Emad Aghajani, Csaba Nagy, Michele Lanza, Gabriele Bavota
Software Institute – USI, Lugano, Switzerland

Abstract—Several techniques have been proposed in the literature to support code completion, showing excellent results in predicting the next few tokens a developer is likely to type given the current context. Only recently, approaches pushing the boundaries of code completion (e.g., by presenting entire code statements) have been proposed. In this line of research, we present FeaRS, a recommender system that, given the current code a developer is writing in the IDE, recommends the next complete method to be implemented. FeaRS has been deployed to learn “implementation patterns” (i.e., groups of methods usually implemented within the same task) by continuously mining open-source Android projects. Such knowledge is leveraged to provide method recommendations when the code written by the developer in the IDE matches an “implementation pattern”. Preliminary results of FeaRS’ accuracy show its potential as well as some open challenges to overcome.

Index Terms—Source code recommender, Code completion

I. INTRODUCTION

Recommender systems for software engineering have been defined by Robillard *et al.* as “applications that provide information items valuable for a software engineering task in a given context” [1]. In this context, source code recommender systems (i.e., techniques able to recommend useful pieces of code for an implementation task at hand) pursue one of the long-lasting dreams of software engineering research: The (semi-)automatic generation of source code.

These techniques aim at speeding up the implementation of new code, similarly to what is accomplished through in-IDE code completion [2]. For several years, most of the effort in this field targeted the improvement of the recommendations in terms of accuracy (i.e., the ability to correctly predict the code tokens the developer is going to type), with several works reporting impressive results [3]–[9]. However, little progress has been made regarding the *type of support these tools can provide to developers*. Indeed, techniques and tools able to recommend more complex code elements such as entire statements or even functions have been proposed only very recently [5], [8], [10].

In a recent work, we presented FeaRS [11], an approach that monitors the code of Android developers in the IDE and is able to recommend the complete code of the next method (i.e., signature and method body) they are likely to implement based on method(s) they have already implemented. FeaRS relies on a set of implementation patterns collected by mining open-source Android apps on GitHub.

In this paper, we present an improved version of FeaRS, which we release as an open-source project on GitHub [12].

As compared to the previous version [11], we integrated a new crawler in FeaRS to continuously mine Android apps and learn new implementation patterns (e.g., when developers implement method M_1 , they are likely to implement M_2 as well), thus increasing the size of the knowledge base FeaRS relies on, hopefully increasing its capabilities of recommending relevant methods.

FeaRS also features an Android Studio plugin monitoring the new methods written by the developer in the IDE to generate, based on it, the next method to implement. The communication between the plugin and the knowledge base is performed through a web service, which checks if the newly implemented method(s) received from the plugin match any implementation pattern in the knowledge base.

In this work, we summarize the approach and the empirical evaluation described in our technical paper [11], while we remind the reader of the paper for all details. On top of this, FeaRS’ architecture and a usage scenario are presented.

II. ARCHITECTURE

Fig. 1 depicts FeaRS’ architecture. It is composed of (i) an offline analysis implemented by the *FeaRS Crawler* and the *FeaRS Analyzer*, which aim at mining open-source Android apps and identify implementation patterns; and (ii) an online service implemented by the *FeaRS Web Service* and available through the *FeaRS Plugin*, where developers can check possible recommendations for the next method to be implemented. The recommendations are currently limited to the Java programming language. In the following, we detail the main components of FeaRS’ architecture.

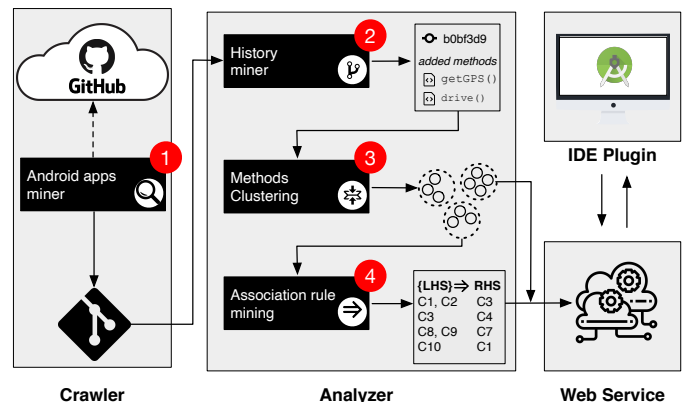


Figure 1. Overview of FeaRS’ architecture

A. Crawling and Analyzing Open-source Android Projects

The *FeaRS Crawler* retrieves and clones Android apps using the GitHub API (step 1 in Fig. 1). The targeted GitHub repositories meet the following criteria: (i) are written in Java; (ii) contain a `build.gradle` file with a dependency towards the Android SDK; (iii) have at least ten stars.

Once all repositories are cloned, the *FeaRS Analyzer* extracts the newly added methods from each commit of each project (step 2 in Fig. 1). The set of new methods added into the same file from the same commit is considered as a candidate implementation pattern. To identify recurring implementation patterns, the *FeaRS Analyzer* applies clustering (step 3 in Fig. 1) to group methods added in different commits, possibly from different systems, implementing equivalent or very similar functionalities.

For example, two commits c_k and c_j performed in two different repositories may implement different sets of new methods (e.g., $M_k = \{m_1, m_2\}$ and $M_j = \{m_3, m_4\}$) that represent the same implementation pattern (i.e., $m_1 = m_3$ and $m_2 = m_4$). FeaRS uses a customized version of the publicly available ASIA clone detector [13] to assess the similarity between different methods and then cluster them.

Different clustering results can be generated by tuning a threshold λ in the similarity algorithm (i.e., two methods are clustered together only if their similarity is higher than λ), and our evaluation showed that the best results are achieved with $\lambda = 0.90$ [11]. Finally, the *FeaRS Analyzer* takes a set of transactions as input, where each transaction is represented as a set of new methods added into the same file from the same commit using their cluster IDs. Then, it applies Association Rule Mining [14] to identify repetitive implementation patterns, relying on the *R arules* package (step 4 in Fig. 1).

The extracted association rules represent the FeaRS' knowledge base in the form of rules $\{M\} \implies m_i$ ($LHS \implies RHS$), where M represents a non-empty set of methods and m_i a method that FeaRS can recommend based on the fact that the developer implemented M .

In addition, this offline analysis automatically checks for projects' updates (i.e., new commits performed in already analyzed projects). It looks for new Android projects that have been created since the previous crawling of GitHub. The knowledge base (i.e., the set of learned implementation patterns) is then updated every three months in order to improve the recommendation capabilities of FeaRS continuously.

B. FeaRS Web Service and IDE Plugin



FeaRS is structured as follows:


- The *FeaRS IDE Plugin* acts as the front-end in direct contact with developers via an interactive GUI;
- The *FeaRS Web Service* plays a role as the back-end which processes the requests of the front-end and interacts with the database;
- The knowledge we learned from the offline analysis (i.e., clusters of methods, association rules) is considered as the database side (knowledge base).

When the developer implements a set of new methods $\{M\}$ in the IDE (note that $\{M\}$ could also be a singleton), this set is identified by the plugin, wrapped into JSON format, and sent to the web service via a POST request. For each method in the set, the web service checks if it can be matched to an existing cluster in the database and, if this is the case, the method set $\{M\}$ will be presented as $\{C\}$, meaning that each method is associated to an existing cluster when possible (the size of $\{C\}$ can be smaller than $\{M\}$ if some methods cannot be matched to any clusters). Finally, for any association rule $\{C_{lhs}\} \implies \{c_{rhs}\}$ in the database, the centroid method of the cluster $\{c_{rhs}\}$ will be sent to the plugin as a recommendation if the following formula is valid:

$$\exists \{C_i\} \subseteq \{C\}, \{C_i\} \subseteq \{C_{lhs}\} \wedge \{c_{rhs}\} \notin \{C\} \quad (1)$$

III. FEARS IN ACTION

Alice is working on the development of an Android app. After Alice installed the IDE plugin, she can start and stop FeaRS through the  and  icons in the IDE toolbar.

By clicking , FeaRS starts monitoring the code written by Alice and identifies when a new method is added. When this happens, the text of the new methods added by Alice is sent to the Web service.

The Web service tries to assign each received method to one of the clusters previously computed while crawling Android apps. Also, in this case, we leverage the ASIA clone detector. In particular, we compute the similarity between each received method and the centroid of all known methods' clusters. Once the most similar centroid is obtained for an added method, the latter is assigned to the corresponding cluster if its similarity is higher than a given threshold that we empirically set at 0.90 [11]. If the similarity is lower, the method is not assigned to any cluster and cannot be used by FeaRS to generate recommendations.

Once the newly implemented methods are matched with the clusters (when possible), all their permutations are used to generate candidate LHSs that can be compared with the implementation patterns (i.e., association rules) in FeaRS' knowledge base. For example, if three methods added by Alice are matched to clusters C_1 , C_2 , and C_3 , we generate 7 possible LHSs: $\{C_1\}$, $\{C_2\}$, $\{C_3\}$, $\{C_1, C_2\}$, $\{C_1, C_3\}$, $\{C_2, C_3\}$, and $\{C_1, C_2, C_3\}$.

Then, FeaRS checks if any of these LHSs is equal to the LHS of one of the association rules previously extracted. If this is the case, a recommendation is generated by exploiting the RHS of the corresponding association rule. For example, if $\{C_1, C_3\}$ is matched in a rule $\{C_1, C_3\} \implies C_7$, then a method representing the centroid of cluster C_7 is returned by the Web service to the plugin as a recommendation. It is important to note that for the same LHS, different RHSs may be recommended (i.e., it is possible to have in the knowledge base different association rules sharing the same LHS). In this

case, the Web service returns the centroid belonging to the RHS of the rule having the highest confidence.

Fig. 2 shows the GUI of the FeaRS Android Studio plugin.

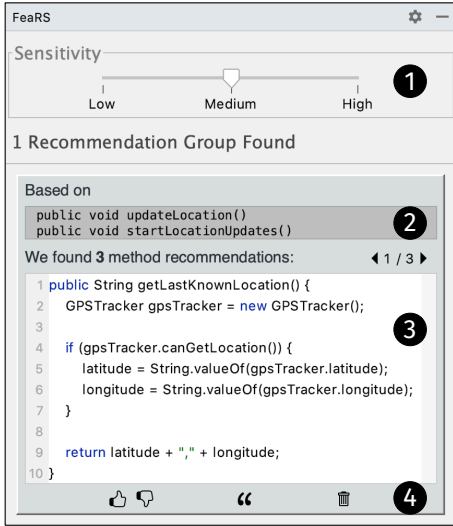


Figure 2. The FeaRS Android Studio plugin [11]

Starting from the top, the slider ① allows setting the sensitivity of the recommendations to one of three possible levels: *Low*, *Medium*, and *High sensitivity*. Moving the slider towards *Low*, FeaRS generates fewer recommendations that, however, are more likely to be relevant and correct. Instead, moving the slider towards *High*, the developer will receive a higher number of recommendations, including more false positives. The three levels are associated to specific configurations of the approach behind FeaRS that we empirically defined [11]. For example, in the *high sensitivity* configuration, only association rules having a confidence of at least 0.5 are considered when triggering recommendations. Such a value increases to 0.65 for the *medium sensitivity* and to 0.8 for the *low sensitivity* configuration. The tool becomes stricter (*i.e.*, lower recall and higher precision) when moving from high to low sensitivity.

The signatures ② shown in the GUI represent the methods implemented by Alice that, once matched to specific clusters, were recognized as part of the LHS of three association rules in the FeaRS’ knowledge base. The fact that three recommendations were generated can be seen from the small arrows in Fig. 2 (the ones showing 1 / 3), that allow Alice to scroll through the three recommendations. Indeed, FeaRS shows only one of the three possible recommendations (*i.e.*, the method `getLastKnownLocation`) that the implemented methods triggered ③.

Alice can then use the three buttons ④ at the bottom of the GUI to (i) like/dislike the suggestion; (ii) copy the recommended method to past it in the IDE; and (iii) discard the recommendation. If Alice pastes the snippet in the IDE, a comment documenting the GitHub repository from which the snippet has been taken is added to the code, so that Alice can check its reusability from a legal perspective.

IV. EVALUATION SUMMARY

FeaRS has been evaluated through a large-scale empirical study based on a dataset featuring the complete change history of 20,713 Android apps [11]. The dataset has been split into *training*, *validation*, and *test*.

The first 80% of the apps’ history has been used as a training set to extract the association rules used by FeaRS to build the knowledge base. The subsequent 10% (“validation set”) has been used to tune the parameters of FeaRS, and allowed to define the three configurations (*i.e.*, low, medium, high sensitivity) we previously described. Finally, the last 10% has been used as a “test set” to assess the performance of FeaRS.

To better understand how the correctness of the generated recommendations has been computed, consider the following running example. Three association rules have been learned in the training set: $\{C_1, C_2\} \implies C_3$, $\{C_4, C_5\} \implies C_6$, and $\{C_2, C_6\} \implies C_7$. The test set features a commit c_i in which a developer implemented three new methods that can be matched to clusters C_1 , C_2 , and C_3 . In this case, we assume that if the developer was using FeaRS while working on c_i , she would have received a correct recommendation. Indeed, our tool could have matched the association rule $\{C_1, C_2\} \implies C_3$, thus correctly recommending C_3 to the developer, saving her time. Clearly, we are assuming that the implementation order followed by the developer in c_i left C_3 as the last method to implement.

Similarly, if we find a commit c_j in which the developer implemented three methods matched to clusters C_1 , C_2 , and C_4 , then we can assume that FeaRS would have generated a wrong recommendation if used by the developer, since also in this the C_3 recommendation would have been triggered.

The details of this study and of how we compute recall and precision based on this data are reported in our technical paper [11]. Also, the same analysis has been performed on the validation set to define the best configurations.

Table I
FEARS PERFORMANCE

Sensitivity	High	Medium	Low
#commits	69,480	69,480	69,480
recall	0.07	0.05	0.04
precision	0.50	0.62	0.72

Table I reports the results achieved by the three FeaRS configurations on the test set. As expected, the precision is quite low in the *high sensitivity* configuration, where FeaRS achieves 0.50. However, the precision increases up to 0.72 in the most conservative scenario (*low sensitivity*). The recall values move instead in an inverse direction, decreasing from 0.07 (*high sensitivity*) to 0.04 (*low sensitivity*).

The precision values indicate that once FeaRS generates a recommendation, it is likely to be adopted by the developer. However, the recall shows a strong limitation of our tool, that we hope to overcome by expanding the knowledge base thanks to the continuous mining of Android apps.

Also, while the recall is low, it still corresponds to thousands of methods correctly recommended.

In addition to the above-summarized quantitative analysis, we performed a qualitative analysis by manually inspecting some examples where FeaRS managed or failed to generate a good recommendation. There are two main observations we made from the manual analysis. First, FeaRS often recommends quite short methods that, while potentially useful, are likely to represent trivial recommendations with little time-saving potential for developers. Second, not all “false positive recommendations” in our study are actually wrong recommendations. Indeed, we found cases in which the recommended method, while slightly different from the one implemented by the developers in the test-set commits, accomplished a very similar goal, thus being potentially useful [11].

V. RELATED WORK

FeaRS is mainly related to code completion techniques and code search engines.

A. Code Completion Techniques

Several techniques have been proposed to improve code completion. Some works focused on proposing smarter ranking of the recommendations: Instead of sorting them alphabetically, context-sensitive approaches have been defined to build better ranking mechanisms [3], [5]. IDEs have also recognized the importance of context-sensitive recommendations: Eclipse has plugins to extend its core code completion, among these, *aiX Code Completer* [15] and *Codota* [16] use AI techniques and can even recommend a full line of code.

Hindle *et al.* [4] pioneered the work on statistical language models applied to software. They used n -gram models to create a language-agnostic algorithm that predicts the next token in a given statement. This work opened the research to several other attempts to improve language models [6], [8], [9] for the specific task of code completion.

Robbes and Lanza used information extracted from the change history to improve code completion of method calls and class names. Their tool is able to propose a correct match in the top-3 results in 75% of cases [7].

Karampatsis *et al.* [17] suggested that neural networks are the best language-agnostic algorithm for code completion, showing its superiority compared to the state-of-the-art language model [18]. Since then, several works have leveraged deep learning-based architectures to create code recommender systems (see, *e.g.*, [19]–[21]). The recently proposed GitHub Copilot tool [10] builds on top of this literature.

While these approaches are undoubtedly valuable to speed up code writing, most of them are limited to recommendations related to the next few tokens the developer is likely to type. With FeaRS, we forge another step ahead to predict the next full method a developer is likely to implement by exploiting implementation patterns learned from open-source repositories.

B. Code Search Engines

FeaRS is also related to code search engines, namely techniques and tools that allow retrieving code samples and reusable open-source code from the Web [22]–[27].

For example, Thummalapenta *et al.* developed a code search engine exploiting static analysis to return relevant code samples for search queries [24], [28]. The authors also extended their approach [25] to assist users by detecting hotspots that can serve as starting points for reusing APIs. McMillan *et al.* [26], [27] combined three sources of information (*i.e.*, the textual descriptions of applications, the API calls used inside each application, and the dataflow among those API calls) to locate relevant software in a large code base.

Compared to code search engines, FeaRS also queries a database of code elements (in our case, code implementation patterns) to identify those possibly relevant for a coding task at hand. However, developers do not need to formulate any query, with FeaRS automatically inferring what they need by monitoring their development activities.

VI. CONCLUSIONS

We presented FeaRS, a tool able to recommend the next full method a developer is likely to implement given the new method(s) implemented in the IDE. FeaRS exploits implementation patterns learned from thousands of open-source Android apps, with the number of patterns growing over time thanks to a continuously running mining process.

The whole idea behind FeaRS is to exploit what has been defined by Hindle *et al.* [4] as the naturalness of software: *What a developer is doing has a high chance of having been done by someone else, somewhere else before.*

Our future work focus on a number of directions aimed at improving FeaRS. First, there are non-trivial licensing issues related to code reuse from open-source projects. Such issues are shared with most of the learning-based code recommender systems. As of now, our strategy has been to “delegate” the final decision to the developer. However, better support can be provided by at least informing the developer through an automated assessment of possible licensing issues. Second, while the developer can copy/paste the recommended methods, FeaRS does not help in any way in integrating a copied method in the code under development, for example, by automatically adapting the identifiers in the recommended method to those already defined in the code written in the IDE, if possible. Finally, we plan to run a controlled experiment with developers to assess the actual usefulness of our recommender system during implementation tasks.

FeaRS is available on GitHub as an open-source project [12].

ACKNOWLEDGMENT

This project has received funding from the European Research Council (ERC) under the European Union’s Horizon 2020 research and innovation programme (grant agreement No. 851720).

REFERENCES

- [1] M. P. Robillard, W. Maalej, R. J. Walker, and T. Zimmermann, *Recommendation Systems in Software Engineering*. Springer Publishing Company, Incorporated, 2014.
- [2] G. C. Murphy, M. Kersten, and L. Findlater, “How are Java software developers using the Eclipse IDE?” *IEEE Software*, vol. 23, no. 4, pp. 76–83, 2006.
- [3] M. Bruch, M. Monperrus, and M. Mezini, “Learning from examples to improve code completion systems,” in *Proceedings of the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, ser. ESEC/FSE 2009, 2009, pp. 213–222.
- [4] A. Hindle, E. T. Barr, Z. Su, M. Gabel, and P. Devanbu, “On the naturalness of software,” in *Proceedings of the 34th International Conference on Software Engineering*, ser. ICSE 2012. IEEE Press, 2012, pp. 837–847.
- [5] A. T. Nguyen, T. T. Nguyen, H. A. Nguyen, A. Tamrawi, H. V. Nguyen, J. Al-Kofahi, and T. N. Nguyen, “Graph-based pattern-oriented, context-sensitive source code completion,” in *Proceedings of the 34th International Conference on Software Engineering (ICSE 2012)*, 2012, pp. 69–79.
- [6] Z. Tu, Z. Su, and P. Devanbu, “On the localness of software,” in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. FSE 2014, 2014, pp. 269–280.
- [7] R. Robbes and M. Lanza, “Improving code completion with program history,” *Automated Software Engineering*, vol. 17, no. 2, pp. 181–212, 2010.
- [8] V. Raychev, M. Vechev, and E. Yahav, “Code completion with statistical language models,” in *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI 2014, 2014, pp. 419–428.
- [9] A. T. Nguyen, H. A. Nguyen, and T. N. Nguyen, “A large-scale study on repetitiveness, containment, and composability of routines in open-source projects,” in *Proceedings of the IEEE/ACM 13th Working Conference on Mining Software Repositories (MSR 2016)*, 2016, pp. 362–373.
- [10] “GitHub Copilot <https://copilot.github.com>.”
- [11] F. Wen, E. Aghajani, C. Nagy, M. Lanza, and G. Bavota, “Siri, write the next method,” in *Proceedings of the IEEE/ACM 43rd International Conference on Software Engineering (ICSE 2021)*, 2021, pp. 138–149.
- [12] “FeaRS GitHub project. <https://github.com/USI-INF-Software/FeaRS>.”
- [13] E. Aghajani, G. Bavota, M. Linares-Vásquez, and M. Lanza, “Automated documentation of Android apps,” *IEEE Transactions on Software Engineering*, 2019.
- [14] R. Agrawal, T. Imieliński, and A. Swami, “Mining association rules between sets of items in large databases,” *SIGMOD Rec.*, vol. 22, no. 2, pp. 207–216, Jun. 1993.
- [15] “aiX Code Completer. <https://tinyurl.com/ydb2ux8x>.”
- [16] “Codota. <https://www.codota.com>.”
- [17] R. Karampatsis and C. A. Sutton, “Maybe deep neural networks are the best choice for modeling source code,” *CoRR*, vol. abs/1903.05734, 2019. [Online]. Available: <http://arxiv.org/abs/1903.05734>
- [18] V. J. Hellendoorn and P. Devanbu, “Are deep neural networks the best choice for modeling source code?” in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE 2017, 2017, p. 763?773.
- [19] S. Kim, J. Zhao, Y. Tian, and S. Chandra, “Code prediction by feeding trees to transformers,” in *Proceedings of the IEEE/ACM 43rd International Conference on Software Engineering (ICSE 2021)*, 2021, pp. 150–162.
- [20] G. A. Aye, S. Kim, and H. Li, “Learning autocompletion from real-world datasets,” in *Proceedings of the IEEE/ACM 43rd International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP 2021)*, 2021, pp. 131–139.
- [21] F. Liu, G. Li, Y. Zhao, and Z. Jin, “Multi-task learning based pre-trained language model for code completion,” in *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE 2020. Association for Computing Machinery, 2020.
- [22] S. Bajracharya, T. Ngo, E. Linstead, Y. Dou, P. Rigor, P. Baldi, and C. Lopes, “Sourcerer: A search engine for open source code supporting structure-based search,” in *Companion to the 21st ACM SIGPLAN Symposium on Object-Oriented Programming Systems, Languages, and Applications*, ser. OOPSLA ’06. ACM, 2006, p. 681–682.
- [23] S. P. Reiss, “Semantics-based code search,” in *Proceedings of the 31st International Conference on Software Engineering*, ser. ICSE ’09. IEEE Computer Society, 2009, p. 243–253.
- [24] S. Thummalapenta and T. Xie, “Parseweb: A programmer assistant for reusing open source code on the web,” in *Proceedings of the 22nd IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE ’07. Association for Computing Machinery, 2007, p. 204–213.
- [25] —, “SpotWeb: Detecting framework hotspots and coldspots via mining open source code on the web,” in *Proceedings of the 23rd IEEE/ACM International Conference on Automated Software Engineering*, 2008, pp. 327–336.
- [26] M. Grechanik, C. Fu, Q. Xie, C. McMillan, D. Poshyvanik, and C. Cumby, “A search engine for finding highly relevant applications,” in *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1*, ser. ICSE ’10. ACM, 2010, p. 475–484.
- [27] C. McMillan, M. Grechanik, D. Poshyvanik, C. Fu, and Q. Xie, “Exemplar: A source code search engine for finding highly relevant applications,” *IEEE Transactions on Software Engineering*, vol. 38, no. 5, pp. 1069–1087, 2012.
- [28] S. Thummalapenta, “Exploiting code search engines to improve programmer productivity,” in *Companion to the 22nd ACM SIGPLAN Conference on Object-Oriented Programming Systems and Applications Companion*, ser. OOPSLA ’07. ACM, 2007, p. 921–922.