DeepCommenter: A Deep Code Comment Generation Tool with Hybrid Lexical and Syntactical Information

Boao Li[*]
Zhejiang University
China

Meng Yan
Chongqing University
China

Xin Xia
Monash University
Australia

Xing Hu, Ge Li
Peking University
China

David Lo
Singapore Management University
Singapore

ABSTRACT

As the scale of software projects increases, the code comments are more and more important for program comprehension. Unfortunately, many code comments are missing, mismatched or outdated due to tight development schedule or other reasons. Automatic code comment generation is of great help for developers to comprehend source code and reduce their workload. Thus, we propose a code comment generation tool (DeepCommenter) to generate descriptive comments for Java methods. DeepCommenter formulates the comment generation task as a machine translation problem and exploits a deep neural network that combines the lexical and structural information of Java methods.

We implement DeepCommenter in the form of an Integrated Development Environment (i.e., Intellij IDEA) plug-in. Such plug-in is built upon a Client/Server architecture. The client formats the code selected by the user, sends request to the server and inserts the comment generated by the server above the selected code. The server listens for client's request, analyzes the requested code using the pre-trained model and sends back the generated comment to the client. The pre-trained model learns both the lexical and syntactical information from source code tokens and Abstract Syntax Trees (AST) respectively and combines these two types of information together to generate comments. To evaluate DeepCommenter, we conduct experiments on a large corpus built from a large number of open source Java projects on GitHub. The experimental results on different metrics show that DeepCommenter outperforms the state-of-the-art approaches by a substantial margin.

Demo URL: https://youtu.be/acdH5X-eBw4
Plug-in download: https://git.io/JegwQ

CCS CONCEPTS

• **Software and its engineering** → **Software development techniques**.

KEYWORDS

Comment Generation, Program Comprehension, Deep Learning

ACM Reference Format:
Boao Li, Meng Yan, Xin Xia, Xing Hu, Ge Li, and David Lo. 2020. DeepCommenter: A Deep Code Comment Generation Tool with Hybrid Lexical and Syntactical Information. In *Proceedings of the 28th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '20), November 8–13, 2020, Virtual Event, USA* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3368089.3417926

1 INTRODUCTION

As the scale of software projects increases, it's getting harder for developers to comprehend code. Code comments are helpful for program comprehension [18, 22]. Unfortunately, code comments may often be missing, mismatched or outdated due to tight development schedule or other reasons in many projects [11]. Automatic code comments generation can not only help developers in understanding source code, but also saving time needed to write comments.

The process of code comment generation is similar to the machine translation process. However, code comment generation is more challenging compared to machine translation since there are two main challenges: 1) *Source code is structured.* Source code written in programming languages is structured and unambiguous, and the main challenge and chance is how to apply unambiguous structure information to the existing Neural Machine Translation (NMT) techniques [1]. 2) *Vocabulary.* The vocabulary in natural language corpora is usually limited to 30,000 words, but in our Java code corpus, we get 794,711 unique tokens. If the common 30,000 tokens are used as the vocabulary, about 95% identifiers will be regarded as *unknown tokens*, i.e., ⟨*UNK*⟩, so it's not suitable for our task.

To address the challenges mentioned above, we proposed a deep code comment generation approach with hybrid lexical and syntactical information (i.e., Hybrid-DeepCom) in our previous work [11]. Hybrid-DeepCom customizes a sequence-based language model to analyze the Abstract Syntax Trees (AST) and source code at the same time. It learns the syntactic and lexical information from the AST and the source code respectively. The ASTs are converted into sequences before fed into Hybrid-DeepCom. Hybrid-DeepCom designs a new structure-based traversal (SBT) method to traverse ASTs. To address the vocabulary challenge, we analyze the composition of identifiers and find out that an identifier usually consists of multiple words, e.g. toString → {to, string}. These words are used to represent the functionality of the variables or methods.

[*] also with Pengcheng Laboratory, Shenzhen, China.

ESEC/FSE '20, November 8–13, 2020, Virtual Event, USA

ACM ISBN 978-1-4503-7043-1/20/11…$15.00
https://doi.org/10.1145/3368089.3417926

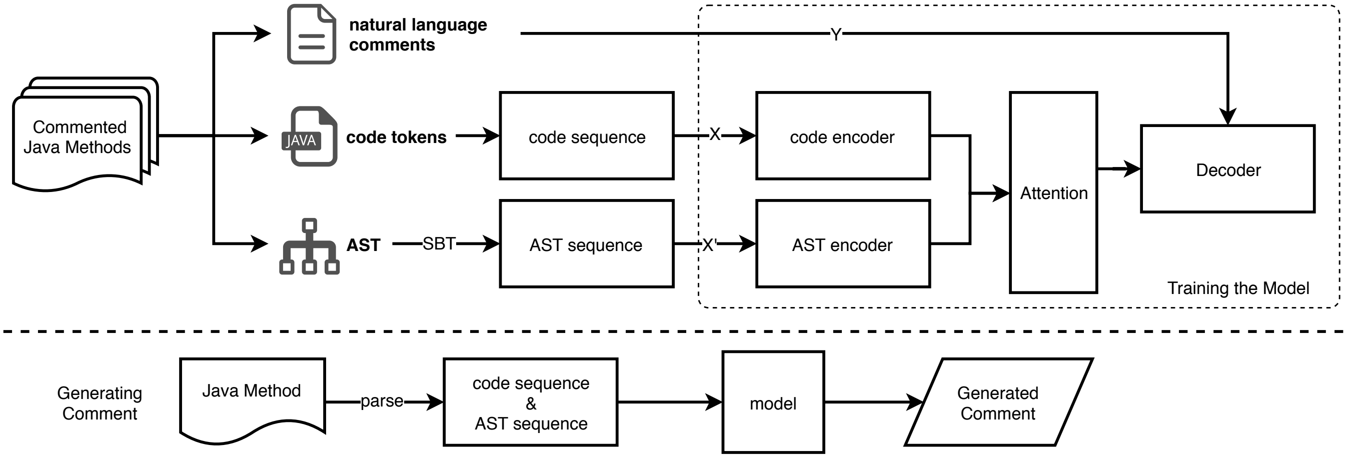


Figure 1: Framework of Our Approach

In this paper, we propose a tool DeepCommenter that is implemented based on our Hybrid-DeepCom [11]. DeepCommenter is in a form of an Integrated Development Environment (i.e., Intellij IDEA) plug-in that is built upon a Client/Server architecture. The most important reason for adopting such architecture is that developers just need to specify the code they want to generate comment for without any additional operations. As a result, our tool will generate the corresponding comment automatically. Moreover, since the model is deployed on the server, the tool does not require extra memory or CPU resources for users.

To evaluate DeepCommenter, we compare the performance of DeepCommenter with four baseline approaches on a corpus that consists of 9,714 Java projects from GitHub. Two different metrics are used in the evaluation process and the results show that DeepCommenter outperforms the state-of-the-art approaches by a substantial margin.

2 APPROACH

2.1 Overall Framework

Figure 1 provides the overall framework of our model. In general, our model consists of three stages: 1) *Data processing*. The Java methods obtained from GitHub are parsed into parallel corpus. The target comments are extracted from the corresponding Javadoc of the Java methods. In order to learn the structural information, the Java methods are converted to AST sequences before fed into the model. 2) *Model training*. We use Tensorflow, which is an open source deep learning framework, to build our models. 3) *Online testing*. Both Information Retrieval (IR) metrics (e.g., precision, recall, F-score and F-mean) and Machine Translation (MT) metrics (e.g., BLEU and METEOR) are used to evaluate our model.

2.2 Model Details

Our model is based on the Sequence-to-Sequence (Seq2Seq) architecture, which consists of two main parts, the encoder and the decoder. The encoder takes the input sequence, trains on it and passes the last state to the decoder part. The decoder receives the last state from the encoder part and uses it as the initial state to generate output sequence. It is effective to generate an output sequence from an input using Seq2Seq, and our model takes advantages of it. However, since we need to get the structural information from the AST, which is a different structure compared to code sequence, we need two input sequences to be fed into the model. Hence two encoders is required in the encoder part.

Encoders. Our model uses two encoders, code encoder and AST encoder, to encode code sequences and AST sequences respectively.

Code Encoder. The code encoder is Gated Recurrent Unit (GRU) [4] and learns the lexical information of Java method. To encode a code sequence, the GRU unit transforms one token of the sequence into a hidden state at each time step, and the result will be fed into the attention layer.

AST Encoder. The other encoder is the AST Encoder. AST encoder learns the structural information from AST sequences. To transform AST into sequence, we proposed a Structure-based Traversal (SBT) method [11] to traverse AST. Similar to the code encoder, AST encoder is another GRU and each unit transforms one token of the sequence into a hidden state at each time step.

These two encoders learn different features from code sequences and AST sequences. After that, these features are transformed into a context vector. In addition, our model is armed with an attention mechanism to fuse the lexical information and structural information.

Attention. Attention mechanism is widely used in machine translation and related fields. Different from the typical machine translation problem, two encoders are adopted to learn different information in our model. Hence the hidden states of the two encoders should be projected into a shared space to fuse the lexical information with structural information, and the attention mechanism computes the attention weights over the projections.

Decoder. The decoder takes the weighted vector generated by the attention mechanism and its generated words as the input to generate the target word sequentially. Beam search [13] strategy and gradient descent optimization function are used to minimize the objective function cross-entropy.

3 IMPLEMENTATION

3.1 Data Collection

The data source of DeepCommenter is collected from GitHub Java repositories in recent few years, and in order to ensure the quality of the code and comments, only repositories with more than 10 stars are collected.

First, we select the Java methods with the corresponding Javadoc in each repository, and use the first sentence in the Javadoc as the target comment of the Java method, because the first sentence in the comment usually explains the meaning of the method. Second, we omit the getter, setter, constructor and the basic test methods marked with @Test annotation; because the meaning of these methods are too simple, they may hinder the model from learning deeper information of the code [10]. Third, we manually check the processed samples and omit the overridden methods to reduce the duplication. After the preprocessing of the dataset, we get 588,108 <method, comment> pairs. The whole dataset can be found in our GitHub repository[1].

3.2 Tool Implementation

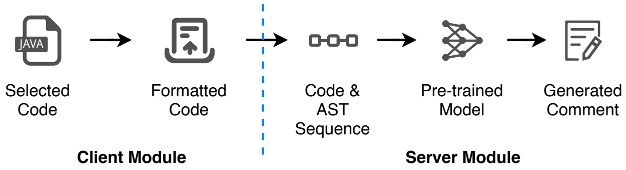


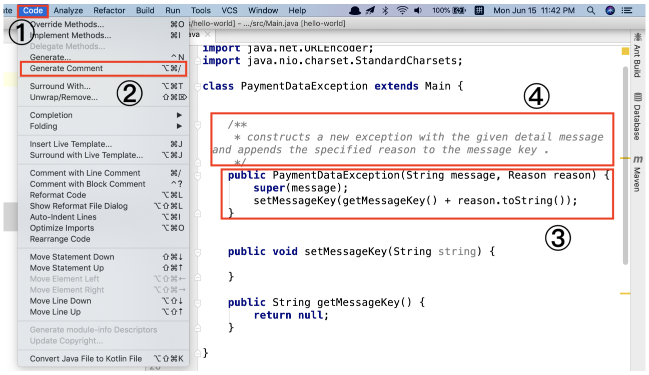
Figure 2: Workflow of Our Plug-in

DeepCommenter is implemented in the form of an Intellij IDEA[2] plug-in based on Client/Server architecture. The workflow of our plug-in is presented in Figure 2. When developers select the code that they want to generate comment for and press the *Generate Comment* button in the *Code* menu, DeepCommenter displays a progress bar and processes it in the background. In detail, the plug-in will wrap the selected code into a JSON[3] object, use Apache HttpClient[4] to send the formatted code to server and wait for response. We deploy a lightweight web application framework Flask[5] on the server to listen for client requests. After the server receives the request, the code is converted to two input sequences (the code sequence and AST sequence) to be fed into the model. The model is preloaded, it generates comments immediately and sends back to the client. After the client receives the response, it will insert the comment above the selected code and the whole process is finished.

In practice, the user may not select the complete method but a segment of the method, and incomplete methods may lead to the problem that the model cannot obtain complete AST information and code semantics. To avoid this problem, the plug-in will check the code segment selected by the user. If the selected code segment is not a complete code segment, it will automatically locate the beginning and end of the method, and send the complete method to server for processing. In addition, the information carried by the RNN may disappear as the length of the code segment increases, which will result in poorly generated comments. This is a problem currently difficult to overcome in code comment generation task. Therefore, if the plug-in detects that the user selected more than k lines of code, it will send a warning to the user. We empirically set k to 50.

Moreover, to meet the efficiency requirement as a real time code comment generation plug-in, we optimize the code base of the previous work in the following way. Unlike the test process, in which the model is loaded into memory every time it is executed, the model is deployed on server and we only need to load it into memory once. It saves time in loading the model into memory, and reduces the memory pressure of the client.

3.3 Installation & User Interface



```
import java.net.URLEncoder;
import java.nio.charset.StandardCharsets;

class PaymentDataException extends Main {

    /**
     * constructs a new exception with the given detail message
and appends the specified reason to the message key .
     */
    public PaymentDataException(String message, Reason reason) {
        super(message);
        setMessageKey(getMessageKey() + reason.toString());
    }

    public void setMessageKey(String string) {

    }

    public String getMessageKey() {
        return null;
    }

}
```

Figure 3: User Interface of the Plug-in

The plug-in adopts the Client/Server architecture, which means that it is quite easy for users to install the plug-in without worrying about any model deployment. Users only need to download the plug-in from the website, open settings of Intellij IDEA, select the *Plugins* tab, choose *Install Plugin from Disk*, and then restart Intellij IDEA to activate the plug-in.

Figure 3 shows the user interface of our plug-in. The plug-in follows the workflow in Figure 2 to generate comments. To use the plug-in, users need to select the code they want to generate comment for (③), and press the *Generate Comment* button (②) in the *Code* menu (①). DeepCommenter displays a progress bar and processes the task in the background. The plug-in will insert the generated comment (④) above the code automatically. To make the inserted comment look natural, the indentation of inserted comment is made consistent with the code. Moreover, we provide keyboard shortcuts "ctrl + alt + /" (Windows) and "command + option + /" (MacOS) to make it easier for developers to use the plug-in.

In addition, the core process of generating comments takes place on the server, which means that it is easy for developers to develop plug-in for other Integrated Development Environment (IDE), e.g., Eclipse[6]. Plug-in developer only needs to wrap the code selected

[1] https://github.com/xing-hu/EMSE-DeepCom
[2] https://www.jetbrains.com/idea/
[3] https://www.json.org/
[4] http://hc.apache.org/httpcomponents-client-ga/
[5] https://palletsprojects.com/p/flask/
[6] https://www.eclipse.org

by the user into a JSON object and send it to the server, receive the response from the server and parse the response data in UTF-8 format, and insert the generated comment into the editor. This design makes it convenient for plug-in developers to extend the plug-in. For example, when the new model needs to use the fully qualified name of the method name or the API call sequence of the method, plug-in developer only needs to add the required fields in the JSON object and parse the corresponding fields on the server side.

4 EVALUATION

We compare our tool with four baseline approaches:

(1) CODE-NN: CODE-NN [12] is one of the state-of-the-art methods for code comment generation and code summarization. It generates code comment for code snippets from end to end. It integrates code token embeddings and uses RNN with attention mechanism to generate code comment.
(2) Seq2Seq model: The Seq2Seq model takes the source code sequence as input, which means that Seq2Seq model only takes the lexical information into consideration. The comment generated by Seq2Seq model reflects the performance of NMT techniques for code comment generation task.
(3) Attention based Seq2Seq model: The Seq2Seq model armed with attention mechanism is able to disregard the noise and focus on the relevant information.
(4) DeepCom with classical traversal method: DeepCom [10] is the base version of Hybrid-DeepCom proposed in our previous work. Compared to Hybrid-DeepCom, DeepCom only uses structural information from the traversed AST sequences and generates comment word by word.

The data set we used to evaluate our tool is introduced in section 3.1. Both Information Retrieval (IR) metrics and Machine Translation (MT) metrics are used to evaluate the model. The IR metrics used include precision, recall, F-score and F-mean [5]. The MT metrics used include BLEU [17] and METEOR [5].

The experimental results on IR metrics show that our tool outperforms other approaches by a substantial margin. It illustrates the importance of structural information in code comment generation task. Our model combines the lexical and structural information together, which helps generate more accurate and user-friendly comments.

The MT metrics (BLEU and METEOR) are smoother compared to IR metrics. The BLEU metrics include the Sentence-level BLEU (S-BLEU) and the Corpus-level BLEU (C-BLEU). The experimental results show that our tool achieves the best performance in all three metrics. Our model is more capable of learning the structural information compared to DeepCom using pre-order traversal. More details of our evaluation results can be found in our journal paper [11].

In addition, excessively high latency is beyond the tolerance of a real-time code comment generation tool, so we conducted efficiency tests on the plug-in. we evaluate the plug-in on Ubuntu 16.04 LTS with 6-core Intel 3.60GHz CPU, four NVIDIA GTX 1080 GPU and 64GB RAM. The result shows that generating a comment takes an average of 140 milliseconds.

5 RELATED WORK

Previous work shows that comments can be generated given a source code snippet [15, 16, 18]. Traditional approaches generate comment manually or simply use the IR-based approach [7, 8]. Approaches requiring much manual input may not be practical to be applied in real-world scenario. IR-based approach has two main limitations: one is that when the methods and identifiers are poorly named, they cannot extract the exact keywords used to identify similar code snippets, and the other is that it only depends on the similarity of the code snippets. Although the precision of IR-based approaches is high, they cannot achieve high recall since the models tend to generate short comment.

Recently, more and more researches use probabilistic models for large-scale source code data sets. Hindle et al. proved that probabilistic models can be used to model source code [9], and a series of subsequent studies have developed probabilistic models for different software engineering tasks [6, 14, 20, 21]. For code comment generation task, current probabilistic model-based methods often use source code to generate comments directly, such as the RNN model with attention mechanism (i.e., the CODE-NN approach [11]). The experimental results also prove the effectiveness of the probabilistic model for the code comment generation task.

Besides, code comment generation can also be seen as a variant of NMT problem [2, 3, 10, 19]. The biggest difference between typical NMT techniques and code comment generation is: NMT translates a sentence of one natural language (e.g., English) to another natural language (e.g. Chinese), and code comment generation translates source code which is written in a programming language to a piece of text in a natural language. Our tool is inspired by the aforementioned studies, but differs in that we implement a tool in the form of an Integrated Development Environment (i.e., Intellij IDEA) plug-in.

6 CONCLUSION

We propose a code comment generation tool DeepCommenter that treats the code comment generation task as a machine translation problem. DeepCommenter is implemented in the form of an Integrated Development Environment (i.e., Intellij IDEA) plug-in based on Client/Server architecture. Its approach is a variant of Seq2Seq model with attention mechanism, which learns lexical information and structural information from code sequence and AST sequence respectively to generate comments for Java methods. In the future, we will improve the way the plug-in interacts with users, allowing users to interact with the plug-in by accepting or rejecting the generated comments. We will enhance DeepCommenter to support more programming languages (e.g., Python, C++) and add domain-specific customizations. We will explore more features of source code to make the tool more robust and effective.

ACKNOWLEDGMENTS

This research was partially supported by the National Key R&D Program of China (No. 2018YFB1003904), NSFC Program (No. 61972339), the Australian Research Council's Discovery Early Career Researcher Award (DECRA) (DE200100021), and Alibaba-Zhejiang University Joint Institute of Frontier Technologies.

REFERENCES

[1] Miltiadis Allamanis, Earl T Barr, Premkumar Devanbu, and Charles Sutton. 2018. A survey of machine learning for big code and naturalness. *ACM Computing Surveys (CSUR)* 51, 4 (2018), 1–37.

[2] Miltiadis Allamanis, Hao Peng, and Charles Sutton. 2016. A convolutional attention network for extreme summarization of source code. In *International conference on machine learning*. 2091–2100.

[3] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. 2014. Neural machine translation by jointly learning to align and translate. *arXiv preprint arXiv:1409.0473* (2014).

[4] Kyunghyun Cho, Bart Van Merriënboer, Caglar Gulcehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. 2014. Learning phrase representations using RNN encoder-decoder for statistical machine translation. *arXiv preprint arXiv:1406.1078* (2014).

[5] Michael Denkowski and Alon Lavie. 2014. Meteor universal: Language specific translation evaluation for any target language. In *Proceedings of the ninth workshop on statistical machine translation*. 376–380.

[6] Xiaodong Gu, Hongyu Zhang, Dongmei Zhang, and Sunghun Kim. 2016. Deep API learning. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. 631–642.

[7] Sonia Haiduc, Jairo Aponte, and Andrian Marcus. 2010. Supporting program comprehension with source code summarization. In *2010 acm/ieee 32nd international conference on software engineering*, Vol. 2. IEEE, 223–226.

[8] Sonia Haiduc, Jairo Aponte, Laura Moreno, and Andrian Marcus. 2010. On the use of automated text summarization techniques for summarizing source code. In *2010 17th Working Conference on Reverse Engineering*. IEEE, 35–44.

[9] Abram Hindle, Earl T Barr, Zhendong Su, Mark Gabel, and Premkumar Devanbu. 2012. On the naturalness of software. In *2012 34th International Conference on Software Engineering (ICSE)*. IEEE, 837–847.

[10] Xing Hu, Ge Li, Xin Xia, David Lo, and Zhi Jin. 2018. Deep code comment generation. In *Proceedings of the 26th Conference on Program Comprehension*. ACM, 200–210.

[11] Xing Hu, Ge Li, Xin Xia, David Lo, and Zhi Jin. 2019. Deep code comment generation with hybrid lexical and syntactical information. *Empirical Software Engineering* (2019), 1–39.

[12] Srinivasan Iyer, Ioannis Konstas, Alvin Cheung, and Luke Zettlemoyer. 2016. Summarizing source code using a neural attention model. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. 2073–2083.

[13] Philipp Koehn. 2004. Pharaoh: a beam search decoder for phrase-based statistical machine translation models. In *Conference of the Association for Machine Translation in the Americas*. Springer, 115–124.

[14] Pablo Loyola, Edison Marrese-Taylor, and Yutaka Matsuo. 2017. A neural architecture for generating natural language descriptions from source code changes. *arXiv preprint arXiv:1704.04856* (2017).

[15] Paul W McBurney and Collin McMillan. 2014. Automatic documentation generation via source code summarization of method context. In *Proceedings of the 22nd International Conference on Program Comprehension*. ACM, 279–290.

[16] Laura Moreno, Jairo Aponte, Giriprasad Sridhara, Andrian Marcus, Lori Pollock, and K Vijay-Shanker. 2013. Automatic generation of natural language summaries for java classes. In *2013 21st International Conference on Program Comprehension (ICPC)*. IEEE, 23–32.

[17] Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. 2002. BLEU: a method for automatic evaluation of machine translation. In *Proceedings of the 40th annual meeting on association for computational linguistics*. Association for Computational Linguistics, 311–318.

[18] Giriprasad Sridhara, Emily Hill, Divya Muppaneni, Lori Pollock, and K Vijay-Shanker. 2010. Towards automatically generating summary comments for java methods. In *Proceedings of the IEEE/ACM international conference on Automated software engineering*. ACM, 43–52.

[19] Ilya Sutskever, Oriol Vinyals, and Quoc V Le. 2014. Sequence to sequence learning with neural networks. In *Advances in neural information processing systems*. 3104–3112.

[20] Song Wang, Taiyue Liu, and Lin Tan. 2016. Automatically learning semantic features for defect prediction. In *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*. IEEE, 297–308.

[21] Martin White, Michele Tufano, Christopher Vendome, and Denys Poshyvanyk. 2016. Deep learning code fragments for code clone detection. In *2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 87–98.

[22] Xin Xia, Lingfeng Bao, David Lo, Zhenchang Xing, Ahmed E Hassan, and Shanping Li. 2017. Measuring program comprehension: A large-scale field study with professionals. *IEEE Transactions on Software Engineering* 44, 10 (2017), 951–976.