A Neural Model for Method Name Generation from Functional Description

Sa Gao[†], Chunyang Chen[*‡], Zhenchang Xing[§], Yukun Ma[†], Wen Song[†], Shang-Wei Lin[†]
[†]School of Computer Science and Engineering, Nanyang Technological University, Singapore
[‡]Faculty of Information Technology, Monash University, Australia
[§]Research School of Computer Sciecne, Australian National University, Australia
gaos0011@e.ntu.edu.sg, chunyang.chen@monash.edu, zhenchang.xing@anu.edu.au
mayu0010@e.ntu.edu.sg, songwen@ntu.edu.sg, shang-wei.lin@ntu.edu.sg

***Abstract*—The names of software artifacts, e.g., method names, are important for software understanding and maintenance, as good names can help developers easily understand others' code. However, the existing naming guidelines are difficult for developers, especially novices, to come up with meaningful, concise and compact names for the variables, methods, classes and files. With the popularity of open source, an enormous amount of project source code can be accessed, and the exhaustiveness and instability of manually naming methods could now be relieved by automatically learning a naming model from a large code repository. Nevertheless, building a comprehensive naming system is still challenging, due to the gap between natural language functional descriptions and method names. Specifically, there are three challenges: how to model the relationship between the functional descriptions and formal method names, how to handle the explosion of vocabulary when dealing with large repositories, and how to leverage the knowledge learned from large repositories to a specific project. To answer these questions, we propose a neural network to directly generate readable method names from natural language description. The proposed method is built upon the encoder-decoder framework with the attention and copying mechanisms. Our experiments show that our method can generate meaningful and accurate method names and achieve significant improvement over the state-of-the-art baseline models. We also address the cold-start problem using a training trick to utilize big data in Github for specific projects.**

***Index Terms*—Naming Convention, Encoder-Decoder Model, Transfer Learning.**

I. Introduction

Approximately 70% of the source code of a software system consists of identifiers [9]. Hence, the naming conventions, i.e. names chosen as identifiers, are of paramount importance for the readability and comprehensibility of computer programs. Several empirical studies showed that concise, clear and meaningful identifiers can help significantly reduce the efforts needed to read and understand source code [9], [4], and hence enhance the software quality [6]. This is especially important for software maintenance and future expansion. Method name is also a kind of identifier in software development. It should be carefully selected, because it densely represents a basic block of source code which constitutes the whole software. High-quality method names should be **meaningful** and **concise**. "Meaningful" means that the name of the method should accurately represent the design intent of the developers and be understood easily by reviewers. "Concise" means that the method name should use minimum characters.

It is not easy for developers, especially novices, to name the method in an appropriate way[1]. Meaningful and concise method names usually follow certain naming conventions [17], [5]. For novice developers with little expertise and experience, they need a massive amount of time to learn the rules themselves by reading books and others' code. Even for those experienced developers, figuring out a good method name without any suggestions is still time-consuming. It would be valuable if we could get high-quality method names directly from the developer's intention, which could be represented by natural language description of the method. Therefore, we propose a method name generation approach which aims at generating a meaningful and concise method name from the natural language description of the corresponding method.

This naming problem cannot be well solved by simple rule-based methods or unsupervised methods. The reason is that the natural language descriptions are always complete sentences with an average of 10 words per sentence, while in order to be concise, the method names are normally short phrases with an average of 3 subtokens inside based on the statistics in our collected datasets. A simple solution to this problem is to extract keywords from descriptions using tf-idf features. However, this method can only extract the existing words in source description and neglects the naming conventions that the developers follow. For example, *"called when a service request completes"* is the natural language description of the method *onComplete*. In this case, the subtoken *"on"* means that the method will be called when a certain event happens. Although this subtoken is not in the method description, an experienced developer will use it to name the method. Such name mapping conventions are quite common in practice, e.g., deleting the -s for the third person singular form verb and using abbreviations. Therefore, it is impossible for a rule-based method to enumerate all rules to solve the naming problem.

With the rapid development of deep learning techniques and sufficient code repository resources, we could learn these naming conventions automatically from the source code of

*Corresponding author.

[1]https://stackoverflow.com/questions/421965/anyone-else-find-naming-classes-and-methods-one-of-the-most-difficult-part-in-pr, https://www.quora.com/Why-is-naming-things-hard-in-computer-science-and-how-can-it-can-be-made-easier

978-1-7281-0591-8/19/$31.00 © 2019 IEEE

millions of projects. In this paper, we introduce a deep neural network model which is trained on big code data to learn the method naming conventions. We collect three datasets from Github code repository, Java official API documentation and Android API Reference documentation, respectively. Each dataset is prepared in the form of natural language method description sentence and method name pairs. The natural language sentences are tokenized as word sequences. Then, we split the method name into a subtoken sequence following the language convention. For example, we split the Java method name *"getByteArray"* to the sequence *"get byte array"* based on the Camel Case naming rule. Therefore, we formulate the problem as a sequence to sequence learning problem which targets at automatic method name subtoken sequence generation from complete natural language sentences. The basic sequence to sequence model utilizes RNN encoder to pass the final state of encoder to the decoder for generation, but the final state may not be effective in capturing the whole information of a long sentence. Therefore, we adopt the attention mechanism [3] in our model, which weights the different parts in input sequence for each subtoken in the output sequence. The attention mechanism has a soft alignment feature that associates the token in the decoder sequence with a token in the encoder sequence. This is widely used in deep learning approaches for various tasks, such as machine translation[3] and image caption generation [27]. It not only improves the generation performance for the long input sentence, but also helps explicitly show the naming convention mapping between natural language description and method name in our model.

In the encoder-decoder model, due to the sparsity nature of subtoken vocabularies, some subtokens are rare words with low word frequency. For example, one of the method descriptions in the collected datasets is *"gets the value of the extrinsic property"*, where *extrinsic* is a rare word that does not exist in the training set. We observe that there are some syntax patterns and location information that we can use to model these mapping relations, even if we do not understand the meaning of the rare word. In the previous example, the word between *"gets the value of"* and *"property"* has a high probability of being used in the method name. In this case, we could directly copy the key words in a specific context to the method name, and this can be done using the copying mechanism [11]. Therefore, we propose to utilize copying mechanism to learn the probability for each word in the method description to appear in a method name.

By incorporating the attention and copying mechanisms, we can help the developers generate method names directly and also let them explicitly understand the naming conventions. In addition, the transferability of the learned naming conventions is explored. The main contributions of this paper are:

- We formulate the task of generating high-quality method names from natural language as a sequence to sequence problem and propose using the encoder-decoder model to learn the naming conventions and automatically generate method names.
- We adopt the attention and copying mechanisms to improve the generation performance and explain the Java naming conventions.
- We explore the naming convention transferability across different projects and propose to use a pre-trained model to transfer the knowledge from massive open source data to a specific project.

The rest of this paper is organized as follows: Section II presents the background knowledge of RNN and RNN encoder-decoder models. Section III describes the attention and copying mechanisms that we use to model the naming conventions. Section IV describes the process of collecting and preprocessing the data from online resources as well as the implementation details. The proposed model is evaluated quantitatively in Section V and the related research questions are also discussed. We review the related work in Section VI, and the conclusion and future work are given in Section VII.

II. Background

A. RNN

Recurrent Neural Network (RNN) is a class of neural networks where the connections between units form directed cycles. Owing to its nature, it is especially useful for tasks with sequential inputs, such as speech recognition [10] and code completion [26]. Compared with traditional n-gram language model, an RNN-based language model [19] can predict the next word by the preceding words with variant distance, rather than with a fixed number. The architecture of a basic RNN model includes three layers. An input layer maps each word to a vector using word embedding or one-hot word representation. A recurrent hidden layer recurrently computes and updates a hidden state after reading each word. An output layer estimates the probability of the next word given the current hidden state. During training, the parameters are learned by backpropagation [25] with gradient descent to minimize the error rate. The most widely used RNN variants are LSTM [14] and GRU [8], which impove the performance by introducing gates to control the information flow.

B. RNN Encoder-Decoder

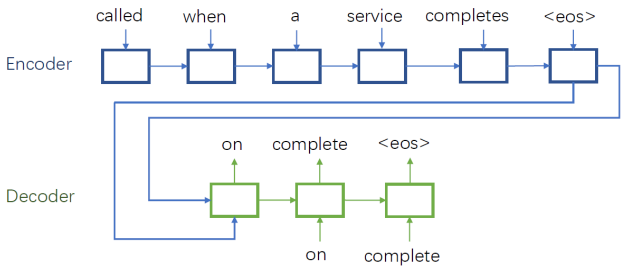

Fig. 1. The achitechture of RNN encoder-decoder model.

More complex RNN-based models have been developed for Natural Language Processing (NLP) tasks. For example, the RNN encoder-decoder model [8] is commonly adopted in machine translation tasks. This model includes two RNNs: one RNN to encode a variable-length sequence into a fixed-length vector representation, and the other RNN to decode

the given fixed-length vector representation into a variable-length sequence. From a probabilistic perspective, this model is a general method to learn the conditional distribution over a variable-length sequence conditioned on yet another variable-length sequence, i.e. $P(y_1, ..., y_{T'}|x_1, ..., x_T)$. The length of the input T and output T' may differ.

The architecture of this model can be seen in Figure 1. The encoder is an RNN that reads each word of an input sentence x sequentially. As it reads each word, the hidden state of the RNN encoder is updated. After reading to the end of the input (marked by an end-of-sequence symbol), the hidden state of the RNN is a vector c summarizing the whole input sentence. The decoder of the model is the other RNN which is trained to generate the output sentence by predicting the next word y_t given the hidden state $h_{(t)}$. However, unlike the basic RNN, y_t and h_t are not only conditioned on y_{t-1} but also on the summary vector c of the input sentence, i.e.

$$P(y_t|(w_1, ..., w_{t-1}), \mathbf{c}) = g(h_t, y_{t-1}, \mathbf{c}) \quad (1)$$

The two RNN components of the RNN encoder-decoder model are jointly trained to maximize the conditional log-likelihood

$$\max_{\theta} \frac{1}{N} \sum_{n=1}^{N} \log p_\theta(y_n|x_n) \quad (2)$$

where θ is the set of the model parameters and each (x_n, y_n) is a pair of input and output sequence from the training corpus.

III. Approach

Since both the description sentence and method name could be regarded as sequences, we formulate our task as a sequence to sequence learning task. Considering the characteristics of the method name generation, we investigate two improvement strategies for the basic RNN encoder-decoder. One is the attention mechanism to assist transformation from long document to a method name, and the other is the copying mechanism to copy the input words directly into the output method name.

A. Attention Mechanism

In the basic RNN encoder-decoder model, the decoder is supposed to generate a translation solely based on the last hidden state from the encoders. Although the model works well for short sentence translation, it seems unreasonable to assume that all information of a long input sentence can be encoded or compressed into a single vector. As there may be some information loss in the vector, the decoding process based on it will lead to bad results.

According to our observation, different parts of an input comment could be of different importance to the words in the target method name. For example, suppose that the method description is *"return an identifier for this item"* and the target method name is *getId*. The word *return* is more important than *an* to the target summarization word *get*. Therefore, to avoid the limitation of the basic RNN encoder-decoder, we adopt the attention mechanism [3] in our model.

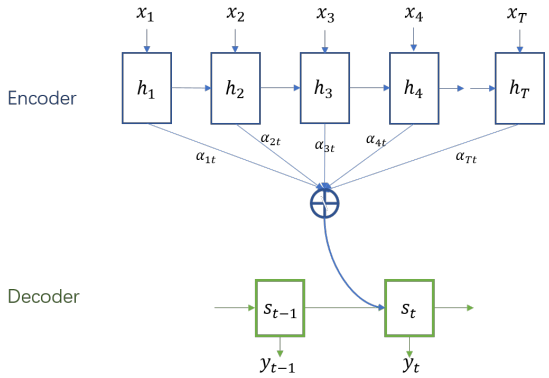

Fig. 2. The architecture of RNN encoder-decoder attention model.

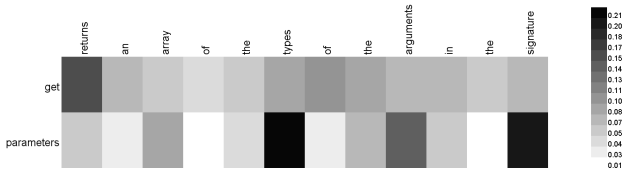

Fig. 3. The visualization of attention for the description "return the array of the types of the arguments in the signature". The X-axis represents the token in method description and the Y-axis represents the subtoken of method name.

With the attention mechanism, we no longer rely on merely the last hidden output to represent the entire input description. Instead, we allow the decoder to "attend" to different parts of the source sentence at each step of the output generation. Figure 2 illustrates the workflow of the attention mechanism. In this figure, y_t is the word generated by the decoder at the time step t, and $x_1, x_2, ..., x_i, ..., x_T$ are our source sentence words. The attention model defines the individual context vector c_t for each target word y_t as a weighted sum of all historical hidden states $h_1, ..., h_T$, rather than just the last state as in the original RNN encoder-decoder. That is,

$$c_t = \sum_{i=1}^{T} \alpha_{it} h_i \quad (3)$$

where α_{it} is the weight that defines how much of the input x_i should be considered for the output y_t. For example, if α_{34} is a large number, it means that the decoder pays much attention to the third state in the source sentence when generating the fourth word of the target method name.

A big advantage of the attention mechanism is that it enables us to interpret and visualize what the model is doing. For example, by visualizing the attention weight matrix when a sentence being translated (Figure 3), we can understand how the model is translating. The heat map in Figure 3 shows the attention weights of each word in descriptions for the subtokens in method name. In this example, given the description *"return the array of the types of the arguments in the signature"*, the word *returns* leads to *get* in the method name, and our model predicts *parameters* by seeing words *types*, *arguments* and *signature*. This example shows the power of the attention mechanism in spotting the keywords in the description to help direct the predictions.

B. Copying Mechanism

Although the attention mechanism could softly align the source method descriptions and target method name by passing the weighted context vector, there are still some problems when dealing with the unknown words. Due to the training efficiency and the model accuracy, the deep neural network model always sets a limited vocabulary size to cut the infrequent word in practice. If the target word y_t is not in the vocabulary, the model could not predict that target word in any way. Different from the machine translation tasks where the encoder and decoder have different vocabulary for different language, in our model part of the method name could be obtained directly, i.e. copied from the method descriptions. Similar situations also exist in tasks such as text summarization [11] and key phrase generation [18] where the copying mechanism is utilized to solve the OOV problem.

In the previous section, the encoder passes only the context vector, which is embedded with semantic information, to decoder. With the copying mechanism, two parts of information is passed from the encoder to the decoder. The first part is the context vector embedded with the semantic information, and the other is the location information. Then the decoder could do the prediction based on both context vector and the vector embedded location information. When combining these two kinds of information, the objective function is defined as

$$P(y_t|(w_1,...,w_{t-1}),\mathbf{c}) = g(h_t, y_{t-1}, \mathbf{c}) + c(h_t, y_{t-1}, \mathbf{c}) \quad (4)$$

where the function g is the generate mode and the function c represents the copy mode. The prediction of the next word in decoder part is based on the probability of both the generate mode and copy mode. The generate mode is basically the same as the previous encoder-decoder model with attention mechanism. Although the context vector in attention mechanism is encoded with more alignment information in soft way, it is still a semantic vector. The copying mechanism directly uses the location information. Specifically, for the word appearing in encoder sentence, the decoder part will calculate an additional copy probability as follows:

$$\begin{aligned} c(h_t, y_{t-1}, \mathbf{c}) &= \frac{1}{Z} \sum_{j:x_j=y_t} e^{\psi_c(x_j)}, y_t \in \mathcal{X} \\ \psi_c(y_t = x_j) &= \sigma(h_j^\top W_c) S_t \end{aligned} \quad (5)$$

where the probability is normalized by summing up the scores for all the words in the generation and copy modes. $\psi_c(y_t = x_j)$ is the score function for copying the word x_j in encoder sentence to the word y_t, regardless of whether x_j is in vocabulary. The details of the state update are introduced in [11]. We can see from equation (4) that the objective function explicitly models the copy behavior. Actually, one of the main naming problems is how to choose the suitable words from the sentence descriptions, i.e. the word is already in the description, but we need to select which one should be used. For example, suppose that the function description of a method is *"return id of this item"*. We easily know that the name should be *"get _"*, but we are not sure whether the name is *"get id"*, *"get item"* or *"get item id"*. The copying mechanism explicitly models this problem and learns the word choosing convention from massive training instances.

IV. Data Collection and Implementation

A. Data Collection

To evaluate the proposed model, we first need to construct the datasets which include the method names and their corresponding natural language descriptions. We collect three datasets from one open source code repository and two Java official documentations, respectively. The statistic details of these datasets are shown in Table I.

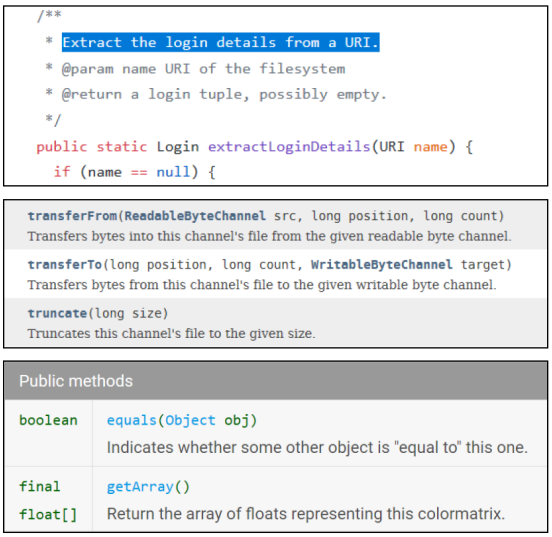

Fig. 4. Examples from three datasets. From top to bottom: source code Javadoc, Java Official Documentation, Android API Reference.

TABLE I
Dataset Details

	Github Dataset	Java Dataset	Android Dataset
pair count	946841	23739	18665
description length	9.64 ± 6.16	12.80 ± 6.84	12.37 ± 6.96
name length	2.75 ± 1.11	2.45 ± 1.01	2.87 ± 1.15
unique words	149706	5983	6797
vocab size	21816	2346	2177

1) Github Dataset: As an open source code repository, Github contains a massive number of open source projects. Although not all of the projects are high-quality projects developed by experienced developers with different naming conventions, they still provide abundant data to cover various kinds of conventions. Following the Javadoc comments writing guidelines[20], the first sentence of the Javadoc comments in source code for method "should be a summary sentence, containing a concise but complete description of the API item". Therefore, we treat this summary sentence as natural language description of the corresponding method name. We first randomly crawl Java projects from Github and extract all Java files from these projects. Then we use Eclipse Java Development Tools to parse all these source files to abstract syntax tree (AST). By retrieving all method declaration nodes within the source code, we extract all the methods that have Javadoc comments. After that, we derive the method descriptions by selecting the first sentence of the comments. Finally,

we randomly select 1,000,000 pairs of such descriptions with their corresponding method names.

2) Official API Documentation: Most of the APIs in Java projects have the one sentence summary in their official documentations as in Figure 4. Compared with the Java Doc comments of the source code in Github, the naming convention of these APIs are more professional and consistent. We select two most popular API documentations in Java: Java API Specification and Anrodid API Reference based on the evidence [28] that *oracle.com* and *android.com* are in top 10 most referenced websites in Stack Overflow.

The selected Java Documentation version is Java SE 8u151. We download the documentation and extract all HTML files under the */api* folder. Then we obtain all the method names and their summaries from methods summary table. We ignore the inherited methods since their summary sentences are not meaningful. We also filter out the methods whose summary is empty. For the Android API Reference documentation, of which the version is API level 24, we extract all HTML files under */reference/android* folder and then follow the steps for Java Doc. In addition, we delete the methods whose summary starts with *"This method was deprecated"*. These two documentations are called Java dataset and Android dataset for short in the following text, respectively.

3) Data Preprocessing: We do a simple data preprocessing by deleting the invalid symbols and keeping only the characters that can be encoded by ASCII code. We also filter out the pairs containing too long comments or method names, since the encoder and decoder have a limitation on the maximum sentence length. We set the maximum comments length as 100 and the maximum method subtoken length as 10, and only 0.045% pairs are filtered out in this setting. We also observe that due to the domain nature of software engineering, there are lots of overloading and overriding methods having the same method names. Some of these methods also use the same natural language descriptions. These repetitive pairs may result in bias for evaluation results, so we delete redundant pairs with identical description and method names and keep only single record for these repetitive pairs. After the data preprocessing, we get 946,841 pairs from Github repository, 23,739 pairs from Javadoc and 18,665 pairs from Android Doc. The details of each dataset are shown in Table I.

B. Implementation Details

We build the encoder-decoder using Gated Recurrent Unit (GRU) [8], a variant of RNN. It has comparable performance with LSTM but is more efficient. Note that the models in our experiment use the same bidirectional GRU encoder. We experimented with varying number of input word embedding and hidden units. We choose to use 350 as the input word embedding size and 500 as the dimension of the hidden variable based on the trade-off between performance and efficiency. In the decoder, we use beam search to generate the subtoken sequence and the beam size is 10. The maximum length of decode sequence is also 10. We early stop our training after 10 epochs to prevent overfitting. Our implementation is based on Theano framework and runs on Nvidia Tesla M40 GPU.

V. Evaluation

The basic idea of our research is to learn the conventional mappings that enable generating the method name from natural language description automatically. Based on the learned model, we further explore the transferability of naming conventions across different projects, which in turn could potentially promote our method name generation strategy. In short, our study mainly focuses on three research questions.

RQ1: Can the encoder-decoder model with attention mechanism and copying mechanism learn the naming convention?

RQ2: Does the copying mechanism help improve the model robustness for the OOV words in our task?

RQ3: Can the naming convention knowledge transfer across different projects?

A. Evaluation Metrics

We use three evaluation metrics to evaluate the recommendation performance in subtoken level and character level, respectively.

1) Exact Match: The exact match is the most straightforward evaluation metric which measures how many of the generated subtoken sequences are identical to the ground truth.

2) BLEU: BLEU (bilingual evaluation understudy) [21] is an algorithm used by machine translation to measure the accuracy of the translated candidate sentences with reference sentences. In practice, it compares the decode sequence with the target sequence to measure how many words in target sequence appear in decode sequence. The score is calculated as follows:

$$BLEU = BP \cdot exp(\sum_{n=1}^{N} w_n log p_n) \tag{6}$$

where p_n is the modified n-gram precision calculated by dividing the clipped count of n grams in candidate sentence by the count of n grams in reference sentence.

To overcome the bias for short candidate sentence, brevity penalty is introduced to reduce the bias:

$$BP = \begin{cases} 1 & c > r \\ e^{(1-r/c)} & c \leq r \end{cases} \tag{7}$$

where r is the length of reference sequence and c is the length of candidate sequence. In the following experiment, we compare the recommend subtoken sequence with the real method name sequence to calculate the BLEU score. Since the average length of method name subtoken sequence is short, we set the maximum value of n to 2 for n-gram.

3) Edit Distance: Edit distance is a character level evaluation metric for measuring the difference between two strings. It computes the minimum number of operations required to correct the generated string to target string. The commonly used edit distance is Levenshtein distance that contains three types of correction operations (i.e. insertion, removal and substitution of single character). Less edit operations means higher

similarity between the generated string and the target string. In our experiment, we recover the method name by concatenating the generated subtokens as a single string following Camel Case naming rules. Then we compare the generated method name with the target method name in character level using edit distance. For example, the edit distance between the generated method name *getId* and the target name *getDocId* is 3 where the insertion of three characters is needed to turn the generated name into the target name.

B. Intra-project Evaluation and Results (RQ1)

The first experiment we conduct is for verifying if the encoder-decoder model could learn the naming conventions. We then validate the contribution of attention and copying via comparing the RNN-Att and RNN-Att-Copy models with baselines. Therefore, we firstly train and test different models on each dataset separately.

We split each dataset into training, validation and test sets. For Java and Android datasets, we randomly sample 3,000 data pairs as test set and another 3,000 pairs as validation set, and the rest is used as training set. For Github dataset which has a much larger size of data, we randomly take 10,000 samples as test set and another 10,000 samples as validation set, and the remaining data is training set. We have released our datasets online[2]. For each dataset, we train different models on training set and test models on test set to learn the naming conventions and answer RQ1. We select tf-idf model, basic RNN encoder-decoder model [24] and convolutional copy attention model [2] as baseline models. For tf-idf model, we use the same strategy from [29], where we treat all the method descriptions in training set as corpus to calculate the idf value and then transform each description sentence in test set into tf-idf features. To generate the method name, we extract the top N tokens with the highest tf-idf value. Different from [29] which concatenates the subtokens following their appearing order, we relax the matching condition and leave out the order. Thus we evaluate the tf-idf model only with exact match accuracy. Furthermore, since [2] can generate method names from token sequences obtained from code body, here we set it as a baseline to investigate if the model in [2] can be directly used to learn the naming convention mapping from natural language description.

The evaluation results are shown in table II. Our encoder-decoder model with the attention and copying mechanisms (RNN-Att-Copy) achieves the best results on all three datasets and the encoder-decoder model with the attention mechanism (RNN-Att) is better than the original one (RNN) in all metrics. The experiments results show that the attention and copying mechanisms could model effective naming conventions for generating method names from natural language descriptions. The RNN model shows a much better performance than the tf-idf based extraction methods and the convolutional copy attention model. Since the tf-idf model could only extract words from descriptions, it is hard to deal with the method names that contain subtokens not existing in the description sentences. We manually examine the results generated by convolutional copy attention model and observe that compared with RNN and tf-idf, the convolutional copy attention model tends to generate shorter method names with incomplete information. Besides, the generated method names are very likely to contain more subtokens copied directly from the descriptions. The reason behind this is that: the method body consists of formal tokens, e.g., the variable names, which are already well formed and can directly be used in method names while the tokens in descriptions are often natural language words. The RNN encoder could learn the syntactic dependency in the natural language sentence and our copy mechanism could keep a balance between the generation mode and the copy mode. In [2], it is concluded that the RNN fails to capture long-range topical attention features. In our case, we do not have this problem. The description is much shorter than method body based on table I .

For our proposed model, its performances on the three different datasets vary from each other. The Github dataset has the lowest exact match rate compared with other two datasets and our model has the best performance on the Java dataset. This is consistent with the nature of these datasets that we described in Section IV, i.e. Java official documentation is well maintained and consistently formatted while the Github dataset is collected from different projects in various areas.

We also manually examine our generated results and check the success and failure cases. We firstly check if our model can only deal with the easy cases which could be handled by simple rule-based method. In table III, we can see that our model could learn the difficult naming conventions that are hard to be solved by simple rule-based method. Some of the words need to be transformed to abbreviation forms when used in method names. For example, *id* is short for *identifier* and *chars* is short for *characters*. Some of the words should be replaced with a more appropriate form in code, such as that *specify* is replaced by *set* based on the context. It could be impractical to have such a look up table to handle all these kinds of transformation due to the diversity of natural language. Similarly, the tf-idf method exhibits much worse performance compared with the baseline tf-idf model in [29]. In [29], names are generated from test bodies that consist of formal words while our method generates names from pure natural language. Therefore, by manually checking the generated results, we confirm that our method could solve the complicated cases that simple tf-idf method and rule-based method could not handle.

Then we also analyze the failed cases in test set, and find that although some of the generated method names are different from the ground truth, they are still reasonable. We observe that the generated method names sometimes convey the same meaning with the ground truth (e.g., *has_error* vs. *contains_error*), even if there might be different word choices. It is also frequently seen that the generated method name is an abbreviation or acronym of the ground truth, e.g., *setDocumentId* and *setDocId*, where *Doc* is short for

[2]https://drive.google.com/file/d/1fNSmPluXbhQ9cfcAHkTtNrHlD8Jkh2XD

TABLE II
INTRA-PROJECT EVALUATION RESULTS

Model	Github Dataset			Java Dataset			Android Dataset		
	Exact Match	BLEU	Edit Distancce	Exact Match	BLEU	Edit Distancce	Exact Match	BLEU	Edit Distancce
Tf-idf	6.41	N.A.	N.A.	2.27	N.A.	N.A.	2.03	N.A.	N.A.
Conv-Copy	16.31	34.35	8.37	25.43	40.67	6.52	12.47	30.14	9.16
RNN	23.27	56.50	8.03	41.67	65.99	5.49	20.47	54.31	8.18
RNN-Att	25.93	57.88	7.69	45.00	68.61	4.98	28.50	59.48	7.24
RNN-Att-Copy	29.40	60.61	7.68	49.53	71.68	4.58	32.07	63.12	7.05

TABLE III
EXAMPLES OF SUCCESSFUL RECOMMENDATION

Description	Recommendation
returns a unique identifier for this item	getId
you can say which characters you can accept	getAcceptedChars
called when smooth scroller is stopped	onStop
gets the number of columns	getColumnCount
returns a string representation of the object	toString
gets the serif font family name	getSerifFontFamily
sets the brand color for the browse fragment	setBrandColor
returns the formatting pattern to display the time in 12-hour mode	getFormat12Hour
returns am / pm strings	getAmPmStrings
specify a label as the tab indicator	setIndicator

Document. In addition, since the quality of Github projects varies, some of the method descriptions are not clear enough to generate correct method names. These examples indicate that exact match does not suffice to evaluate the true efficacy of the method name generation techniques.

To evaluate our approach in a more accurate way, we conduct a survey to analyze the failed cases. We randomly select 100 failed test cases from the Github dataset, which has the lowest exact match accuracy rate among the three. We prepare the survey materials as follows: given the description of a method, we randomly switch the position of the target method name and the suggested method name to avoid bias. We ask three experienced Java Developers, who volunteered to participate in our survey, to score the two types of method names from 1 to 10. 1 means that the method name has nothing to do with the method description and 10 means that the method name matches perfectly with the description. We compute the average of the scores of the three subjects and use it as the final score for each target name and suggested name pair. We split the results into four groups, which are shown in Tables IV to VII.

In sum, we find 14 samples, of which the scores for both the target and suggested names are below 5. In Table IV, we show 5 examples with very low scores for both the target and suggested names. These poor results are due to the low-quality descriptions. For example, some of the descriptions are automatically generated by a program, such as the first example in Table IV. Some of the descriptions are incomplete, e.g., the third example (*default constructor*) in Table IV.

Table V shows the examples where both the target and suggested names are relatively meaningful with similar scores. There are 21 out of 100 samples, of which at least one of the two scores is no less than 5 and the difference between the two scores is within 2. In this group of samples, both the target and the suggested names are meaningful enough to represent the descriptions. In some of these cases, the difference between the target and suggested names is very small, e.g., the use of word *till* or *until* in the first example of Table V. Moreover, the second example in Table V shows an impressive result: despite that the description contains non-English words *fecha de nacimiento* (which means birthdate), our model can still generate a meaningful method name *onDateOpen*. The fifth example shows another impressive observation that our model sometimes can correct the typos in the target method names.

Table VI shows the examples where the suggested names are more meaningful than the target names. Up to 41% samples fall in this group. Some of the target names are of poor quality due to the skill level diversity of Github developers, e.g., the second example (*toolDone*) in Table VI. In such samples, our suggestions can repair the bad target names. Besides, some of the suggested names have a higher score because they contain more details, such as the third example in Table VI where the suggested name *getClockwiseNeighborHex* is more accurate than the target name *getClockwiseHex*. Actually, if this method is a member of class *Page*, the target name *getTitle* should be better. However, in our settings, we only know the description of the method. Therefore, we cannot tell whether the class information has already been included in the description. Nevertheless, in practice, this issue can be easily solved by detecting whether the class name is contained in the suggested method name.

Table VII shows the examples where the target names are better than the suggested ones. Based on our statistics, in 24% of the samples, the target name gets a higher score with the score difference no less than 2. In most of these samples, the suggested name is related to the description but some detailed information is missing. For instance, in the first and third examples in Table VII, the target names, *savePageInfo* and *showErrorMessage*, are more accurate than the suggested ones, *savePage* and *showError*. In some cases, the descriptions are composites of rare words. For example, the description of the third example in Table VII is written in Dutch, so all the words in this description are out-of-vocabulary, which makes our model invalid in these cases.

In summary, we could answer the RQ1 and conclude that our encoder-decoder model with attention and copying mechanisms could learn the naming conventions and suggest readable method names.

C. Impact of OOV Words (RQ2)

In the previous closed-vocabulary setting, we set the decoder vocabulary cut-off frequency to 10, so only the words with

TABLE IV
EXAMPLES OF LOW SCORES FOR BOTH THE TARGET AND THE SUGGESTED METHOD NAMES

Description	Ground Truth	Recommendation	GT score	Rec score
this method was generated by my batis generator	setCitycode	getAgentsIntroduction	1	1
the class must have a public or protected , no - argument constructor	route	module	1	1
default constructor	abstractYamaQuestion	outputPreparation	1	1
private singleton constructor	application	synthesizerAudioProcessor	1	1
default 0 - arguments constructor	context	substanceConfigurator	1.33	2.33

TABLE V
EXAMPLES OF HIGH SCORES FOR BOTH THE TARGET AND THE SUGGESTED METHOD NAMES

Description	Ground Truth	Recommendation	GT score	Rec score
waits until done is set to true	waitTillDone	waitUntilDone	9	9.33
called when the user clicks the fecha de nacimiento edit text	selectDate	onDateOpen	5.33	6
returns true if the point d is inside the circle defined by the points a, b, c	inCircle	isPointInCircle	7.67	9.33
returns the maximum size of this list	capacity	getCapacity	8	9.33
creates a new instance of gui factory	guifActory	guiFactory	6.67	8.33

TABLE VI
EXAMPLES IN WHICH THE SUGGESTED METHOD NAMES GET HIGHER SCORES

Description	Ground Truth	Recommendation	GT score	Rec score
return the hex that is the clockwise neighbor of the target hex when looking from the starting hex	getClockwiseHex	getClockwiseNeighborHex	5	9.33
sets the default tool of the editor	toolDone	setDefaultTool	1.33	9
returns title for page	getTitle	getPageTitle	6.67	9
a repeat region containing repeat units 2 to 4 bp that is repeated multiple times in tandem	microsatellite	repeatRegion	1.67	8

TABLE VII
EXAMPLES IN WHICH THE TARGET METHOD NAMES GET HIGHER SCORES

Description	Ground Truth	Recommendation	GT score	Rec score
save page information to the database	savePageInfo	savePage	9	6
generates a public - key / private - key pair , create new key objects	generateKeyPair	generateKeys	9	5.67
tekent het ruimteschip	paintSpaceship	toString	8	1
retrieves the crumbs to display as a list	getCrumbsToDisplay	getCrumbs	8.67	6
shows an error dialoge with the title " error "	showErrorMessage	showError	9	7
construct a ship square representing part of a battle ship either a middle or end piece	shipSquare	battleSquare	8	5

TABLE VIII
JAVA TEST RESULTS BASED ON DIFFERENT VOCABULARY SIZE

		RNN-Att			RNN-Att-Copy		
Word Freq	Voc Size	Exact Match	BLEU	Edit Distance	Exact Match	BLEU	Edit Distance
≥ 0	5983	48.00	70.76	4.696	48.57	70.13	4.729
≥ 5	3224	46.07	69.23	4.843	47.53	70.78	4.831
≥ 10	2346	45.00	68.61	4.986	49.53	71.68	4.584
≥ 15	1614	40.83	64.98	5.466	46.67	70.26	4.769
≥ 30	1234	38.03	63.44	5.765	44.37	68.76	5.055

their frequency no less than 10 will be kept as vocabulary. The choice of vocabulary size is basically a trade-off between efficiency and coverage. As a result, there might exist a lot of Out-of-vocabulary (OOV) words. Moreover, even if we could train the training set using all its vocabulary, there will still exist new words in the test set or real-world data, which are not in the training set. Therefore, the copying mechanism is proposed in this paper to address the problem of OOV by allowing copying words from the encoder input.

To investigate the impact of OOV words on different models, we control the vocabulary size of the dataset so that we can evaluate the model performance with different OOV word sizes. We conduct the experiment on Java dataset, in which we are allowed to adjust the vocabulary size in varying scales. In this experiment, we test two models, the RNN-Att and RNN-Att-Copy models. The tests are executed on customized Java Datasets with different cut-off word frequencies and vocabulary sizes. As shown in Table VIII, while the cut-off word frequency increases from 0 to 30, the vocabulary size decreases from 5983 (i.e. the full vocabulary) to 1234.

We can see from Table VIII that RNN-Att-Copy outperforms RNN-Att in most cases. The two models achieve almost the same performance when the cut-off frequency is 0 (i.e. using the full vocabulary). This indicates that the performance improvement mainly results from addressing OOV words or infrequent words with the copying mechanism. As the cut-off frequency increases from 0 to 10, the performance of the RNN-Att-Copy model only changes a little. The best exact-match accuracy is achieved when the frequency is 10. Meanwhile, the performance of RNN-Att model decreases significantly with the decreasing of the vocabulary size. When the cut-off frequency reaches 30, the exact-match accuracy decreases by

TABLE IX
EXAMPLE OF OOV CASE

Source/Description	determine if the object is showing	retorna true si esta reproduciendo
Target/Method name	is showing	is running
Input	determine if the object is UNK	retorna true si esta UNK
Output(RNN-Att)	is UNK	is known
Output(RNN-Att-Copy)	is showing	esta reproduciendo

about 10% for RNN-Att model while the same metric only decreases by about 4% for RNN-Att-Copy model.

Two examples are shown in Table IX to illustrate the advantage of the RNN-Att-Copy model when encountering OOV words. In the first example, *showing* is considered as an OOV word and denoted by *UNK*. The RNN-Att model fails in this case while the RNN-Att-Copy model could directly copy *showing* in the description without knowing its exact meaning. In the other example, the RNN-Att-Copy model could directly copy the last two words based on the vocabulary words *retorna true si esta*, which means *return true if it is*.

Based on this experiment, we can draw a conclusion that the copying mechanism makes a model more robust to different vocabulary sizes. This result is also consistent with our previous conclusion, which is that the copy mechanism could utilize the location information directly and hence could deal with the OOV situation.

D. Cross Projects Evaluation and Results (RQ3)

Since the training set is typically from a single domain or single language, it is likely that the automatically learned naming convention is language or domain dependent. Here, the question is whether these naming conventions can be transferred from one project to another. More specifically, it means that given the description, whether the model trained on one dataset could be used for generating method names for another dataset. To explore this question and investigate whether the knowledge learned by our method is transferable, we conduct an experiment where the models are trained on one dataset but then tested on another.

Table X shows the cross projects evaluation results. By comparing Table X and Table II, it is found that the inter-dataset testing performances reduce to half of the intra-dataset testing performances for the same test dataset. It is also observed from Table X that in the aspect of the training set, the performance of the models trained on small datasets, such as Java and Android datasets, declines significantly when tested on other datasets. The exact match accuracy decreases to around 10% while that in the intra-evaluation ranges from 30% to 50%. Meanwhile, the best performance of the model trained on Github dataset decreases from 32.07% to 21.63% for Android dataset, and from 49.53% to 29.33% for Java dataset. Therefore, we could conclude that all models have significant performance loss when tested on another dataset. Comparatively, the models trained on the Github dataset (i.e. the largest dataset among the three) get relatively smaller performance loss when tested on the other two datasets. This indicates that the knowledge learned from small datasets is too restricted to be applied to other projects. For example, there is hardly any information about location service in Java dataset while this kind of information is prevalent in Android dataset. Owing to the nature of the Github dataset, it has a much broader coverage of domains when compared with a single project. However, due to the variety in the writing styles, cross projects still suffer from significant performance loss.

TABLE X
CROSS PROJECTS EVALUATION RESULTS

Test on	Model		Train on		
			Github	Java	Android
Github	RNN-Att	Exact Match		8.20	8.10
		BLEU		37.52	38.53
		Edit Distance		10.93	10.76
	RNN-Att-Copy	Exact Match		10.43	10.27
		BLEU		41.28	40.37
		Edit Distance		10.40	10.52
Java	RNN-Att	Exact Match	29.33		12.37
		BLEU	60.78		42.80
		Edit Distance	6.822		9.653
	RNN-Att-Copy	Exact Match	28.57		15.67
		BLEU	60.71		47.00
		Edit Distance	7.174		9.314
Android	RNN-Att	Exact Match	20.17	9.57	
		BLEU	54.78	38.49	
		Edit Distance	8.880	11.25	
	RNN-att-Copy	Exact Match	21.63	12.23	
		BLEU	57.19	43.08	
		Edit Distance	8.822	10.42	

TABLE XI
ANDROID TEST RESULTS FOR DIFFERENT TRAINING STRATEGY

Training Data	Exact Match	Exact Match@5	BLEU	Edit Distance
Android	32.07	44.07	63.12	7.050
3K-Android	15.90	23.30	50.74	9.471
Github	21.63	40.40	57.19	8.822
3K-Android+Github	28.20	55.73	60.10	7.316

We then compare the performance of the RNN-att and RNN-att-Copy models with the same experiment settings. Based on the intra-project evaluation result in Table VIII, we conclude that the RNN-att-Copy model outperforms the RNN-att model in most cases, especially when the OOV words are prevalent. When comparing these two models in cross projects settings, we find that the previous conclusion is still valid in most cases, except for the pair of the Github and Java datasets. In most cases in our experiment, the two models are trained on a small dataset with a small vocabulary and tested on a dataset with relatively larger vocabulary or vocabulary containing many rare words. Thus the performance of the RNN-att-Copy model is better than that of the RNN-att model. However, in the case of using the Github dataset for training and Java dataset for testing, the exact match accuracy of the RNN-att model is 0.76%, which is higher than that of the RNN-att-Copy model. These results are also consistent with the nature of each dataset and the previous conclusion. Basically, the Github dataset itself is a mixture of several independent projects, thus it could cover most areas in software engineering domain.

E. Naming Convention Transfer Learning Strategy (RQ3)

The results of cross projects evaluation shows that it is hard to directly use the model trained on one dataset to another

project. Nevertheless, the results also show that when we transfer the knowledge from the Github dataset to a specific project dataset, the outcome is still acceptable, despite of the performance loss. In practice, we could not get the dataset split as in the intra-project evaluation experiment where most of the data is used as the training data. A good recommendation system should be able to deal with insufficient data. Therefore, we propose a pre-training strategy to leverage the model pre-trained on Github dataset as a start point. Different from the strategy that we use in Section V-D, after we get the pre-trained model, we do not directly test it on specific dataset such as Android dataset. Instead, we use a small part of data in target domain to continue training the pre-trained model and then go to test. To evaluate whether our training strategy could help improve the knowledge transferability, we train the RNN-att-Copy model with four different training datasets. The first model is trained on the Android training set used in Intra-project evaluation and the second one is trained on 3000 instances randomly sampled from the Android training set. The third model is trained on the Github training set. The fourth model is trained with our proposed strategy. We leverage the third model as a pre-trained model and continue training it using 3000 sampled instances.

For the evaluation, we introduce another metric, which is exact match accuracy in top 5 recommendation. The max size of beam search is 10, so we could get top 10 recommended sequences from decoder. In previous experiments, we only output the sequence with the highest probability as the generated result. In practice, we could provide more suggestions to users. The evaluation results in Table XI show that the performance of our proposed method (the fourth model) is close to the intra-project evaluation for the exact match accuracy. The fourth model even shows a better performance than the first model in terms of the exact match@5 accuracy. The second model trained with data of limited target domain performs the worst. Based on the results, we can conclude that our proposed transfer learning strategy successfully combines the knowledge learned from large code repository and a small part of data in a specific project.

VI. Related Work

We introduce the related works in two aspects: the code naming conventions and the encoder-decoder model in software engineering domain.

Much research has been done on the code naming conventions. The empiarical study [6] shows that the quality of the method name is associated with that of the source code. Several works propose to repair the identifier names based on the code context, e.g., [22]. [7] focuses on analyzing the syntactic pattern of the identifier names. The most relevant works are [1] and [2]. [1] proposes using bi-linear model to learn the semantic representation of the method names and the identifiers in the method body. [2] proposes a convolution attention neural network to summarize the method body as a method name. Our work is different from them in two aspects. Firstly, we generate the method names directly from the functional descriptions of the methods, so our main focus is on the user intention, rather than the method body. Our experiment shows that we could not adopt their models in our case by simply changing the input from code to the method description. Secondly, we not only study the naming conventions from single project but also conduct a detailed research on how to transfer the naming convention knowledge across different projects.

As we discussed in section II, the RNN encoder-decoder model [24] was firstly introduced in natural language processing domain for neural machine translation (NMT). Owing to its highly end to end fashion, it has been then widely used in different kinds of NLP tasks such as short conversation [23], sentence summarization and keyphrase generation [18] and so on. In software engineering domain, sequence to sequence model has been utilized in different kinds of applications to bridge the programming language and natural language. [12] uses the sequence to sequence model to learn API sequence from the natural language descriptions. This work extracts API sequences from the method body as the representation of this method. [16] proposes to adopt an encoder-decoder model to generate commits descriptions from code diffs. [15] introduces an attention model to summarize code by training an encoder-decoder model on Stack Overflow posts title and code snippets pairs. The three works above all use the RNN model with attention mechanism. As far as we know, we are the first one to utilize the RNN encoder-decoder model with copying mechanism in software engineering domain.

VII. Conclusion and Future Work

The method name is vital for the comprehension of programs. In this paper, we define the method naming convention problem as learning the conventional mappings from natural language description to concise and meaningful method names. Based on the sequence to sequence nature of this issue, we propose an RNN encoder-decoder model with the attention and copying mechanisms to learn the naming conventions from big data. Based on the experiments, the attention and copying mechanisms not only improve the generation accuracy but also help us understand the naming convention in an explicit way. We also examine the robustness of the proposed model on various vocabulary sizes and further prove that our approach could well cope with the OOV problem. We investigate the naming convention knowledge transferability across different projects and find it potential to use the knowledge learned from large repository to specific projects. Thus we propose a transfer learning strategy to leverage the pre-trained Github model as a start point and continue training the model using a small part of data in the target project. The experiment results indicate that the proposed method could suggest meaningful method names using limited labeled data from the target project.

Fine tuning [13] technology is a commonly used approach which uses pre-trained model and continues the training on specific layers instead of the whole model. In the future, we are interested in utilizing fine tuning technology to further explore the naming convention transfer among different projects.

REFERENCES

[1] M. Allamanis, E. T. Barr, C. Bird, and C. Sutton. Suggesting accurate method and class names. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, pages 38–49. ACM, 2015.

[2] M. Allamanis, H. Peng, and C. Sutton. A convolutional attention network for extreme summarization of source code. In *International Conference on Machine Learning*, pages 2091–2100, 2016.

[3] D. Bahdanau, K. Cho, and Y. Bengio. Neural machine translation by jointly learning to align and translate. *CoRR*, abs/1409.0473, 2014.

[4] D. Binkley, M. Davis, D. Lawrie, J. I. Maletic, C. Morrell, and B. Sharif. The impact of identifier style on effort and comprehension. *Empirical Software Engineering*, 18(2):219–276, 2013.

[5] D. Boswell and T. Foucher. *The art of readable code*. " O'Reilly Media, Inc.", 2011.

[6] S. Butler, M. Wermelinger, Y. Yu, and H. Sharp. Exploring the influence of identifier names on code quality: An empirical study. In *Proceedings of the 2010 14th European Conference on Software Maintenance and Reengineering*, CSMR '10, pages 156–165, Washington, DC, USA, 2010. IEEE Computer Society.

[7] C. Caprile and P. Tonella. Nomen est omen: analyzing the language of function identifiers. In *Sixth Working Conference on Reverse Engineering (Cat. No.PR00303)*, pages 112–122, Oct 1999.

[8] K. Cho, B. van Merriënboer, Ç. Gülçehre, D. Bahdanau, F. Bougares, H. Schwenk, and Y. Bengio. Learning phrase representations using rnn encoder–decoder for statistical machine translation. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 1724–1734, Doha, Qatar, Oct. 2014. Association for Computational Linguistics.

[9] F. Deissenboeck and M. Pizka. Concise and consistent naming. *Software Quality Journal*, 14(3):261–282, 2006.

[10] A. Graves, A.-r. Mohamed, and G. Hinton. Speech recognition with deep recurrent neural networks. In *2013 IEEE international conference on acoustics, speech and signal processing*, pages 6645–6649. IEEE, 2013.

[11] J. Gu, Z. Lu, H. Li, and V. O. K. Li. Incorporating copying mechanism in sequence-to-sequence learning. *CoRR*, abs/1603.06393, 2016.

[12] X. Gu, H. Zhang, D. Zhang, and S. Kim. Deep api learning. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 631–642. ACM, 2016.

[13] G. Hinton and R. Salakhutdinov. Reducing the dimensionality of data with neural networks. *Science (New York, N.Y.)*, 313:504–7, 08 2006.

[14] S. Hochreiter and J. Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.

[15] S. Iyer, I. Konstas, A. Cheung, and L. Zettlemoyer. 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)*, volume 1, pages 2073–2083, 2016.

[16] S. Jiang, A. Armaly, and C. McMillan. Automatically generating commit messages from diffs using neural machine translation. *arXiv preprint arXiv:1708.09492*, 2017.

[17] R. C. Martin. *Clean code: a handbook of agile software craftsmanship*. Pearson Education, 2009.

[18] R. Meng, S. Zhao, S. Han, D. He, P. Brusilovsky, and Y. Chi. Deep keyphrase generation. In *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 582–592, Vancouver, Canada, July 2017. Association for Computational Linguistics.

[19] T. Mikolov, M. Karafiát, L. Burget, J. Cernockỳ, and S. Khudanpur. Recurrent neural network based language model. In *Interspeech*, volume 2, page 3, 2010.

[20] Oracle. How to write doc comments for the javadoc tool. https://www.oracle.com/technetwork/articles/java/index-137868.html, 2018. [Online; accessed 20-December-2018].

[21] K. Papineni, S. Roukos, T. Ward, and W.-J. Zhu. Bleu: a method for automatic evaluation of machine translation. In *Proceedings of the 40th annual meeting on association for computational linguistics*, pages 311–318. Association for Computational Linguistics, 2002.

[22] V. Raychev, M. Vechev, and A. Krause. Predicting program properties from "big code". In *Proceedings of the 42Nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '15, pages 111–124, New York, NY, USA, 2015. ACM.

[23] L. Shang, Z. Lu, and H. Li. Neural responding machine for short-text conversation. In *Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*, pages 1577–1586, Beijing, China, July 2015. Association for Computational Linguistics.

[24] I. Sutskever, O. Vinyals, and Q. V. Le. Sequence to sequence learning with neural networks. In *Advances in neural information processing systems*, pages 3104–3112, 2014.

[25] P. J. Werbos. Backpropagation through time: what it does and how to do it. *Proceedings of the IEEE*, 78(10):1550–1560, 1990.

[26] M. White, C. Vendome, M. Linares-Vásquez, and D. Poshyvanyk. Toward deep learning software repositories. In *2015 IEEE/ACM 12th Working Conference on Mining Software Repositories*, pages 334–345. IEEE, 2015.

[27] K. Xu, J. Ba, R. Kiros, K. Cho, A. Courville, R. Salakhudinov, R. Zemel, and Y. Bengio. Show, attend and tell: Neural image caption generation with visual attention. In *International conference on machine learning*, pages 2048–2057, 2015.

[28] D. Ye, Z. Xing, and N. Kapre. The structure and dynamics of knowledge network in domain-specific q&a sites: a case study of stack overflow. *Empirical Software Engineering*, 22(1):375–406, 2017.

[29] B. Zhang, E. Hill, and J. Clause. Towards automatically generating descriptive names for unit tests. In *Automated Software Engineering (ASE), 2016 31st IEEE/ACM International Conference on*, pages 625–636. IEEE, 2016.