Toward Refactoring Evaluation with Code Naturalness

Ryo Arima
Osaka University
Suita, Osaka, Japan
r-arima@ist.osaka-u.ac.jp

Yoshiki Higo
Osaka University
Suita, Osaka, Japan
higo@ist.osaka-u.ac.jp

Shinji Kusumoto
Osaka University
Suita, Osaka, Japan
kusumoto@ist.osaka-u.ac.jp

ABSTRACT

Refactoring evaluation is a challenging research topic because right and wrong of refactoring depend on various aspects of development context such as developers' skills, development cost, deadline and so on. Many techniques have been proposed to evaluate refactoring objectively. However, those techniques do not consider individual contexts of software development. Currently, the authors are trying to evaluate refactoring automatically and objectively with considering development contexts. In this paper, we propose to evaluate refactoring with code naturalness. Our technique is based on a hypothesis: if a given refactoring raises the naturalness of existing code, the refactoring is beneficial. In this paper, we also report our pilot study on open source software.

CCS CONCEPTS

• **Software and its engineering** → *Software evolution*;

KEYWORDS

Refactoring, naturalness, n-gram language model

ACM Reference Format:
Ryo Arima, Yoshiki Higo, and Shinji Kusumoto. 2018. Toward Refactoring Evaluation with Code Naturalness. In *ICPC '18: ICPC '18: 26th IEEE/ACM International Confernece on Program Comprehension , May 27–28, 2018, Gothenburg, Sweden.* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3196321.3196362

1 INTRODUCTION

The source code of software systems decays due to repeated changes such as fixing bugs and adding new functions [1]. Refactoring is known as a promising technique to improve source code quality for future maintenance tasks. Fowler et al. listed typical symptoms in the source code that possibly indicate deeper problems (namely, *bad smells*) and sophisticated ways to get out from under the symptoms [2]. However, in practice, it is not easy to identify where/how to refactor the source code. In fact, programmers rely on their experiences and hunch to do refactorings.

Extract Method and *Inline Method* patterns should be a good example for the refactoring difficulty. The two patterns are oppositional operations on the source code. By applying *Extract Method*, methods become shorter and easier to understand while the code becomes fragmented. By applying *Inline Method*, the fragmented code gets together while the code becomes more complex. Using code metrics is a way to identify where to be refactored [8] but there is no general and strict standard for the code to be refactored. Different projects and different developers have different standards for good code. Thus, the authors consider that it is important to take into account such differences for suggesting refactoring opportunities.

In this paper, we propose to use "naturalness of code" to evaluate refactorings. If a given refactoring raises the naturalness of code, it is regarded as *appropriate*. Naturalness is a numerical value calculated with probabilistics language models. Naturalness means how natural a given word sequence is for the model. Recently, naturalness has been used in software engineering research [4, 7]. The advantage of the proposed technique is that it can evaluate a given refactoring as a numerical value by considering characteristics of its project and developers.

We have applied the proposed technique to a golden set of 28 good refactorings. As a result, the technique regarded 19 out of the 28 refactorings as good.

2 PROBABILISTICS LANGUAGE MODEL AND NATURALNESS

Probabilistics language models are models to calculate generation probability of a given word sequence. It can present naturalness of a given token sequence as a numerical value. The generation probability of word sequence $S = w_1 w_2 \cdots w_m$ can be represented by the following formula.

$$P(S) = P(w_1) \prod_{i=2}^{m} P(w_i | w_1, \cdots, w_{i-1}) \tag{1}$$

Herein, $P(w_i|w_1, \cdots, w_{i-1})$ is the probability that word w_i comes next $w_1, \cdots, w_{i-1}$. However, calculating $P(w_i|w_1, \cdots, w_{i-1})$ is unrealistic because the number of patterns of $w_1, \cdots, w_{i-1}$ becomes enormous. Thus, $n-gram$ language model is used, which consider only n-length subsequences in the given word sequence.

$$P(w_i|w_1, \cdots, t_{w-1}) \simeq P(w_i|w_{i-n+1}, \cdots, w_{i-1}) \tag{2}$$

$P(w_i|w_{i-n+1}, \cdots, w_{i-1})$ is calculated with the following formula.

$$P(w_i|w_{i-n+1}, \cdots, w_{i-1}) = \frac{C(w_{i-n+1}, \cdots, w_{i-1}, w_i)}{C(w_{i-n+1}, \cdots, w_{i-1})} \tag{3}$$

$C(w_{i-n+1}, \cdots, w_{i-1})$ is the number of occurrence of $w_{i-n+1}, \cdots, w_{i-1}$ in source code corpus, which is data used for constructing a model.

ICPC '18, May 27–28, 2018, Gothenburg, Sweden

ACM ISBN 978-1-4503-5714-2/18/05…$15.00
https://doi.org/10.1145/3196321.3196362

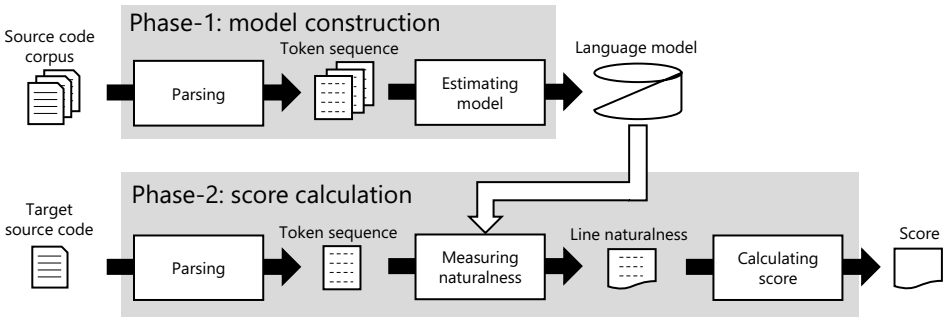


Figure 1: An overview of our technique

A raw value of $P(w_i|w_{i-n+1}, \cdots, w_{i-1})$ is not easy to treat because it often become very small. In this research, we use naturalness $N(w_i)$ of word w_i. $N(w_i)$ is a logarithmic value as follows.

$$N(w_i) = \log P(w_i|w_{i-n+1}, \cdots, w_{i-1}) \tag{4}$$

Consequently, $N(S)$, which is the naturalness of word sequence $S = w_1 w_2 \cdots w_m$ can be represented with the following formula.

$$N(S) = \log P(w_1) + \sum_{i=2}^{m} \log P(w_i|w_{i-n+1}, \cdots, w_{i-1}) \tag{5}$$

Probabilistics language models and naturalness are widely used in the field of machine transition and speech recognition. Hindle et al. found that language models also work well for program source code [4]. Allamanis et al. proposed a technique to suggest appropriate identifier names with language models [4]. Ray et al. investigated the relationship between buggy code and naturalness [7]. They found that buggy code tended to have a low naturalness and fixing bugs tend to increase the naturalness. In this research, we propose to use naturalness for evaluating refactorings.

3 NATURALNESS-BASED REFACTORING EVALUATION

Herein, we propose a new technique to evaluate refactorings. Our key idea is that *source code becomes easier to understand by conducting good refactorings. If source code consists of stereotypical program statements, developers will not find it difficult to understand it. However, if unusual and unfamiliar program statements exist, they will find it difficult to understand.*

In this research, we use naturalness based on n-gram language model to represent the stereotypical degree of program statements. Figure 1 shows an overview of the proposed technique. The proposed technique takes source code corpus and target source code. Its output is a numerical score of the given target source code. This score gets lower if the source code becomes more natural. Thus, if the score of the post-code of a given refactoring is lower than the score of the pre-code, the refactoring is regarded as appropriate.

The proposed technique consists of two phases.

Phase-1 (Model construction): methods in the given source code corpus are extracted. Then, a token sequence is generated from each of the extracted methods. Every n-gram in the token sequences is used to construct a language model. In this research, we use the model proposed in literature [9].

Phase-2 (Score calculation): token sequences are generated in the same way as Phase-1. Then, naturalness is measured for each line in the code. At last, a score is calculated by using lines' naturalness.

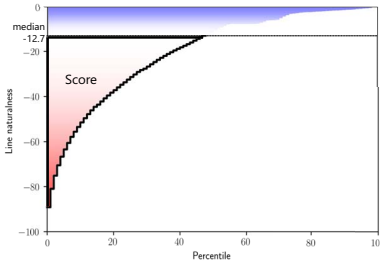


Figure 2: An intuitive model of score calculation

In Phase-2, the following formula is used to calculate the score.

$$score = \frac{1}{M} \sum_{i=0}^{M} max(0, t - N(i)) \tag{6}$$

Herein, M is the number of lines in the given source code, t is the threshold, $N(i)$ means the naturalness of the i-th line in the given source code. We use the median value of naturalness of source code corpus as the threshold. The lower score means that the given source code is more natural.

We explain the meanings of the score calculation with Figure 2. In the figure, all lines in the given source code are sorted in the ascending order fo their naturalness: the leftmost line is most unnatural; the rightmost line is most natural. The dashed horizontal line is the threshold. Thus, the formula means the score is the square measure of the red-colored part in the graph. Consequently, if the square measure is small, the given source code is more natural.

3.1 Example

We show a measurement example by using Figure 3a. The source code in the figure includes a function to sum two hex numbers. The two numbers are given to method *add* as `String`. Herein, we assume that the median of the naturalness of source code corpus is −5.7. The score of this source code becomes 145.6.

Figure 3a includes duplicated instructions that convert a string to a hex number. Thus, we extracted the duplicated instructions as a new method, *convertHex* (Figure 3b). The score of this source code is 66.6. The score dropped by the refactoring.

The extracted method includes a complex instruction. Thus, we dissolved this instruction into multiple simple instructions (Figure 3c). The score becomes 27.3, which means the second refactoring also dropped the score.

With the three types of source code, we can see that the two refactorings make the source code easier to understand. The score dropping represents the source code becomes easier to understand.

4 EXPERIMENT

We applied the proposed technique to actual good refactorings on open source software. Then, we investigated whether the proposed technique regarded each of the refactorings as good or not.

4.1 Procedure

We used an online dataset published by Higo et al [3]. This dataset includes the source code of the Java projects in Apache Software Foundation[1]. There are two reasons why we used this dataset.

- This dataset is large enough to construct an n-gram language model. More concretely, it consists of 84 projects, 66,724 Java source files, and 11.5 MLOC.
- Testing code and auto-generated code in the Java projects were manually eliminated.

We selected JUnit4[2] as our target. JUnit4 is managed with GitHub. The developers review committed code each other, which pledges that survived refactoring commits include good refactorings. Firstly, we identified commits related to refactoring with keyword searches on commit logs. We used "*refactor*" and "*clean* " as keywords. Then, we checked each of committed source code carefully. If the commit included only one refactoring, we added the refactoring to our golden set.

For each of the refactorings in the golden set, we measured naturalness of each line of its pre-code and post-code. As a threshold, we used median of line naturalness of the dataset by using leave-one-out method. Then, we measured the score of the pre-code and post-code of each refactoring and then we compare the two values. If the score of the post-code is lower than the pre-code, a given refactoring increases naturalness of the code. In other words, the proposed technique regardes a given good refactoring as good.

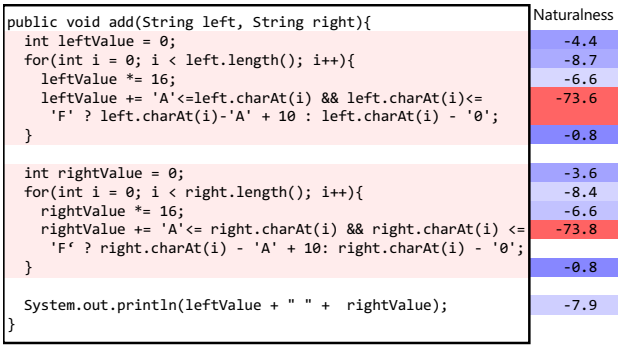
4.2 Results

We confirmed that the proposed technique worked well for 19 out of the 28 target refactorings. The accuracy resulted in 67.8%. However, currently, the proposed technique is naive. The proposed technique performs a lexical analysis for given source files. However, we used obtained token sequences as they were. A couple of optimization can be considered to make the proposed technique better. For example, normalizing user-defined names with special tokens will be effective because we construct language models from different projects. Different project should have different name conventions. Another enhancement is to eliminate trivial tokens (e.g., `private`, and `final`) included in token sequences. Source code contains many such tokens and eliminating those tokens is effective for code completion [5].

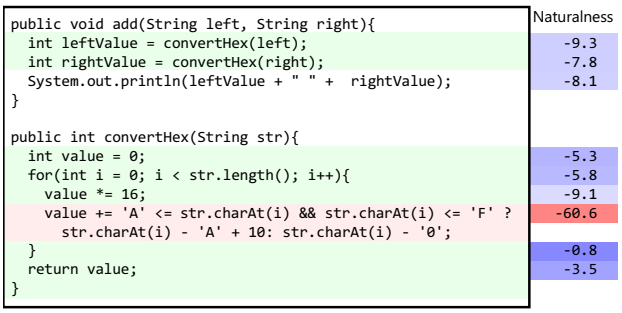
There were three refactorings on that the score did not change. The refactorings were changing modifiers of a class field and moving method to another class. The proposed consider only source code inside methods, which is why the proposed technique was not able to treat the three refactorings.

Table 1 shows the detail results for each target refactoring. ✓in the rightmost column means whether the proposed technique evaluated the refactoring appropriately or not.

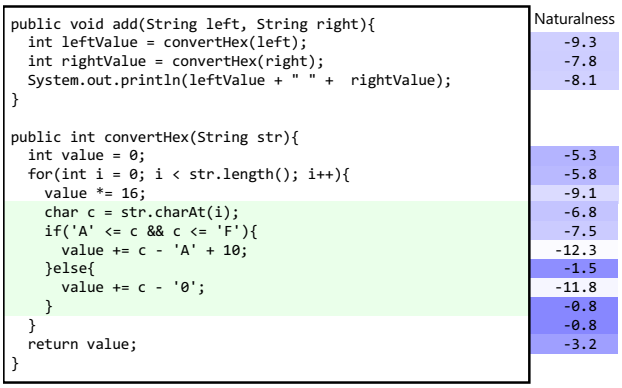
We show a refactoring that the proposed technique evaluated appropriately in Figure 4. Two methods were extracted in this refactoring. We can see that the naturalness of the pre-code is low: −84.4, −62.3, −90.8, and − 60.3. On the other hand, the post-code has higher naturalness: −50.5, −44.2, −32.1, −36.7, and − 56.7. These values mean that the two refactorings eliminated unnatural program statements.



(a) Original source code



(b) After extracting a new method



(c) After dissolving a complex statement

Figure 3: Example

[1]http://www.apache.org/
[2]https://github.com/junit-team/junit4

```
+ private List getAnnotatedFieldsByParameter() {                    Naturalness
+  return getTestClass().getAnnotatedFields(Parameter.class);       -50.5
+ }

+ private boolean fieldsAreAnnotated() {
+  return !getAnnotatedFieldsByParameter().isEmpty();               -44.2
+ }
```

(a) Extracted methods

```
 @Override                                                          Naturalness
 protected void validateConstructor(List<Throwable> errors) {
    validateOnlyOneConstructor(errors);                             -19.2
-    List<FrameworkField> annotatedFieldsByParameter =             -84.4
       getTestClass().getAnnotatedFields(Parameter.class);
-   if (annotatedFieldsByParameter.size() > 0) {                    -62.3
+   if (fieldsAreAnnotated()) {                                     -32.1
        validateZeroArgConstructor(errors);                         -2.9
    }
  }

  @Override
  protected void validateFields(List<Throwable> errors) {
    super.validateFields(errors);                                   -37.1
-    List<FrameworkField> annotatedFieldsByParameter =             -90.8
       getTestClass().getAnnotatedFields(Parameter.class);
-   if (annotatedFieldsByParameter.size() > 0) {                    -60.3
+   if (fieldsAreAnnotated()) {                                     -36.7
+     List<FrameworkField> annotatedFieldsByParameter =             -56.7
      getAnnotatedFieldsByParameter();
```

(b) Replacing code fragments with method invocations

Figure 4: A refactoring on that our technique worked well

4.3 Threats to Validity

The accuracy of the proposed technique depends on the quality of source code corpus. In this experiment, we used popular Java projects in Apache Software Foundation. Testing code and auto-generated code are not included. Thus, we consider that the quality of source code corpus is high enough.

We evaluated the proposed technique on only 28 refactorings. Finding pure refactorings are not easy because many refactorings are floss ones [6]. Anyway, we are going to find more pure refactorings for more rigorous evaluations.

5 CONCLUSION

In this paper, we proposed a new technique to evaluate refactorings. The proposed technique measures naturalness of before/after code of refactoring and then it compares the two values. The features of the proposed technique are (1) evaluating refactoring as quantitatively not qualitatively and (2) considering project-specific and developer-specific standards on the source code. We applied the proposed technique to a golden set of 28 good refactorings. As a result, the proposed technique evaluated 19 out of the 28 refactorings appropriately.

We are still in the early stage of this research. Currently, our model is naive. By using the optimization in the section4.2, we are going to generate more sophisticated token sequences for constructing language models shortly. We are also going to compare the method using naturalness and the method using source code metrics. From the results, we are going to consider advantages of our proposed technique and a combination of naturalness and metrics.

ACKNOWLEDGMENTS

This work was supported by MEXT/JSPS KAKENHI 25220003 and 17H01725.

REFERENCES

[1] Stephen G. Eick, Todd L. Graves, Alan F. Karr, J. S. Marron, and Audris Mockus. 2001. Does Code Decay? Assessing the Evidence from Change Management Data. *IEEE Transactions on Software Engineering* 27, 1 (2001), 1–12.
[2] Martin Fowler, Kent Beck, John Brant, William Opdyke, and Don Roberts. 1999. *Refactoring: Improving the Design of Existing Code.* Addison-Wesley Professional.
[3] Yoshiki Higo and Shinji Kusumoto. 2014. How Should We Measure Functional Sameness from Program Source Code? An Exploratory Study on Java Methods. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. 294–305.
[4] Abram Hindle, Earl T. Barr, Zhendong Su, Mark Gabel, and Premkumar Devanbu. 2012. On the Naturalness of Software. In *Proceedings of the 34th International Conference on Software Engineering*. 837–847.
[5] Bin Lin, Luca Ponzanelli, Andrea Mocci, Gabriele Bavota, and Michele Lanza. 2017. On the uniqueness of code redundancies. In *Proceedings of the 25th International Conference on Program Comprehension*. 121–131.
[6] Emerson Murphy-Hill, Chris Parnin, and Andrew P. Black. 2009. How We Refactor, and How We Know It. In *Proceedings of the 31st International Conference on Software Engineering*. 287–297.
[7] Baishakhi Ray, Vincent Hellendoorn, Saheel Godhane, Zhaopeng Tu, Alberto Bacchelli, and Premkumar Devanbu. 2016. On the "Naturalness" of Buggy Code. In *Proceedings of the 38th International Conference on Software Engineering*. 428–439.
[8] Frank Simon, Frank Steinbrückner, and Claus Lewerentz. 2001. Metrics Based Refactoring. In *Proceedings of the Fifth European Conference on Software Maintenance and Reengineering*. 30–38.
[9] Zhaopeng Tu, Zhendong Su, and Premkumar Devanbu. 2014. On the Localness of Software. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. 269–280.

Table 1: The detailed results. Delta means the score difference between before/after refactoring."✓" means that the proposed technique evaluated the refactoring appropriately

Commit ID	Refactoring Pattern	Delta	Aptness
#23793cd	Extract class	0.46	
#a7c4d03	Extract class	-1.76	✓
#7a2b046	Extract class	-0.62	✓
#fd1ef3c	Extract method	-0.21	✓
#dbe7711	Extract method	-0.49	✓
#2240984	Extract method	0.07	
#862f41c	Extract method	0.91	
#5976b1d	Extract method	-0.15	✓
#0215c66	Extract method	-2.78	✓
#24cbcbc	Extract method	-2.19	✓
#467dd07	Extract method	-0.26	✓
#e48f6d4	Extract method	-1.09	✓
#fe5d86e	Inline method	-1.01	✓
#6838ac0	Inline method	0.49	
#0030e51	Inline method	1.02	
#ce9bc58	Move method	0.00	
#f1f4fe2	Move method	-0.72	✓
#4c1758d	Algorithm change	-0.38	✓
#66bfb24	Algorithm change	-1.14	✓
#df016dc	Algorithm change	-0.24	✓
#17a2f11	Remove unused code	-1.15	✓
#9a0aec8	Remove unused inheritance	0.00	
#a30e87b	add final to variable	0.00	
#db8d580	Remove code clone	-0.49	✓
#e77e1c4	Reuse existing method	-0.23	✓
#759061a	Introduce Strategy pattern	-0.04	✓
#7f2569f	Pull up method	-0.46	
#d064212	Rename method and class	-0.50	✓