On the Impact of Refactoring Operations on Code Naturalness

Bin Lin, Csaba Nagy, Gabriele Bavota, and Michele Lanza
Software Institute — Università della Svizzera italiana (USI), Switzerland

***Abstract*—Recent studies have demonstrated that software is natural, that is, its source code is highly repetitive and predictable like human languages. Also, previous studies suggested the existence of a relationship between code quality and its naturalness, presenting empirical evidence showing that buggy code is "less natural" than non-buggy code. We conjecture that this quality-naturalness relationship could be exploited to support refactoring activities (*e.g.,* to locate source code areas in need of refactoring). We perform a first step in this direction by analyzing whether refactoring can improve the naturalness of code.**

We use state-of-the-art tools to mine a large dataset of refactoring operations performed in open source systems. Then, we investigate the impact of different types of refactoring operations on the naturalness of the impacted code. We found that (i) code refactoring does not necessarily increase the naturalness of the refactored code; and (ii) the impact on the code naturalness strongly depends on the type of refactoring operations.

***Index Terms*—Naturalness, Refactoring, Open Source Software**

I. Introduction

Software is not unique. Researchers have discovered that for sequences of six tokens extracted from the source code, the probability of finding the same sequence in other software projects is higher than 50% [1]. Based on this finding, Hindle *et al.* [2] introduced the concept of source code "naturalness", to indicate that source code is highly repetitive and predictable, just like a text written in human language. They showed that this characteristic can be captured by statistical language models and can be leveraged for different software engineering tasks, such as code completion [3] and fault localization [4]. The latter application proposed by Ray *et al.* was possible thanks to the finding that buggy code is less natural (*i.e.,* less predictable) than correct code [4].

One interesting unanswered question is whether software refactoring (*i.e.,* the activity of improving code quality without modifying the system's external behavior) can be seen as a process implicitly aiming at improving code naturalness. Intuitively, we might think the source code is easier to maintain if it is more natural, as there are fewer "surprising" and "unfamiliar" code fragments for developers. Thus, it can be conjectured that developers focus their refactoring attentions on code exhibiting low naturalness. If such a conjecture is confirmed, information about the naturalness of code components could be leveraged to support refactoring operations (*e.g.,* by identifying code components in need of refactoring).

We perform a first step in that direction by investigating whether refactoring operations applied by software developers result in an improvement of the code naturalness.

We use RMiner [5], a state-of-the-art refactoring miner tool, to mine 1,448 real refactoring operations performed by software developers in 619 open source projects. These operations cover 10 different refactoring types (*e.g.,* move method, extract class). Once these operations are collected, we employ the statistical language model proposed by Tu *et al.* [3] to measure the naturalness of the code components before and after the refactoring. This allows us to verify whether different types of refactoring operations improve the code naturalness. Our results show that the impact on the code naturalness strongly depends on the specific type of refactoring operation. For example, "Extract Method" refactoring is more likely to increase the code naturalness, while "Pull Up Method" refactoring often leads to lower naturalness. These results suggest that leveraging code naturalness for identification of refactoring opportunities is far from trivial, and highlight the need for additional investigations in this direction.

II. Related Work

The naturalness of software has received considerable attention in the software engineering research community. After the seminal work by Hindle *et al.* [2], several studies have investigated the code naturalness from different perspectives. Tu *et al.* [3] found that the distribution of repetitive code is highly skewed in the source code. Lin *et al.* [6] disclosed that different parts of source code are not equally repetitive.

Researchers have also studied the relation between naturalness and software defects. Campbell *et al.* [7] found that syntax errors are less natural than other code, and this fact can be used to augment compilers' ability to locate missing and extra tokens. Ray *et al.* [4] evaluated the naturalness of buggy code and the corresponding fixes by analyzing over 8,000 fix commits from 10 Java projects. Their results showed that buggy code is less natural, and the naturalness increases once the bug is fixed. They also showed that focusing on unnatural code is cost-effective in finding bugs compared to other state-of-the-art static bug finders.

The most relevant work is the study conducted by Arima *et al.* [8], which uses code naturalness as a metric to evaluate whether a refactoring operation is effective. With the assumption that appropriate refactoring should raise the code naturalness, the authors constructed a gold set of 28 refactoring operations extracted from `JUnit4`[1] by searching for the keywords "refactor" and "clean" in commit logs and manually filtering out

[1] https://github.com/junit-team/junit4

those commits containing more than one refactoring. As a result, the code naturalness increases after 19 out of the 28 refactorings, which indicates that naturalness might be a potential valid metric for evaluating the quality of refactoring. Our study, while having a similar objective (*i.e.,* studying the impact of refactoring operations on code naturalness) is performed on a much larger dataset composed of 1,448 refactorings extracted from 619 systems. We also investigate the impact of refactoring operations on the code naturalness by considering the type of implemented refactoring (*e.g.,* move method) as an independent variable to study (possibly having an effect on the "naturalness" dependent variable).

III. Study Design

Our *goal* is to investigate whether refactoring operations increase the naturalness of the refactored code. We assess how the code naturalness is impacted (i) overall, meaning when considering all types of refactoring operations together, and (ii) by specific types of refactoring.

A. Research Question

Our study aims at answering the following research question:

RQ: *How does refactoring impact the naturalness of source code?* This RQ assesses how the naturalness of source code changes after refactoring operations. We also investigate whether there is an observable difference for the change in naturalness for different kinds of refactorings. To the best of our knowledge, this is the first study running such an analysis on a large dataset while considering specific refactoring types.

The findings of this RQ will shed light on the possibility of using code naturalness to support the identification of code components in need of refactoring.

B. Study Context

The *study context* consists of 619 Java projects hosted on GitHub[2], mined on Nov. 6, 2018, using the following selection criteria:

- **Activity level.** To exclude inactive projects, the projects must have at least one commit in the three months preceding the data collection.
- **Popularity.** Projects need to have at least 100 forks[3] and 100 stars[4], to avoid the inclusion of likely "toy-projects". Forks and stars serve as two proxies for the popularity of software repositories on GitHub.

We found 2,663 projects satisfying these constraints. However, due to the computational cost of our experimental design that requires retraining the statistical language models assessing the naturalness several times (details follow), we selected from this set a random subset of 1,500 projects for our study. We believe that 1,500 projects still ensure a good generalizability of our results. After mining refactoring operations from these repositories, we found 619 projects containing at least one of the refactoring operations we study (discussed in Section III-C1). These 619 projects compose our study context.

C. Data Collection

To answer our research question and measure code naturalness, we first mine refactoring operations from the collected projects, and then assess the naturalness of the impacted code components before and after each refactoring commit.

1) Refactoring mining: We use RMiner [5] to mine the refactoring operations in the randomly selected 1,500 projects. RMiner extracts refactoring operations by inspecting two adjacent commits using an AST-based statement matching algorithm. RMiner is reported to have a precision of over 0.95 for most refactoring types, except "Change Package" (0.85) and "Move Field" (0.884). The recall achieved by RMiner is also fairly high: 0.80 for most refactoring types, except "Rename Class" (0.711), "Extract & Move Method" (0.412), and "Move Method" (0.764). Thus, adopting RMiner allows us to obtain different types of refactorings with considerable accuracy. While RMiner can detect various types of refactorings, in this study we only consider those do not requiring the creation of new source code files (*e.g.,* we exclude "Extract Class" refactoring), since this avoids the introduction of confounding factors in the computation of the code naturalness (*i.e.,* the naturalness of the same files before/after refactoring is compared). Table I reports the types of refactoring operations considered in our study.

TABLE I: Considered refactorings in our study

Level	Refactorings considered
Method	Extract Method, Inline Method, Pull Up Method, Push Down Method, Rename Method, Move Method, Extract and Move Method
Field	Pull Up Field, Push Down Field, Move Field

After obtaining all the commits with refactoring operations, we filtered out commits in which more than one refactoring type was applied, again to better isolate and study the effect of a single type of refactoring operation on the code naturalness. In the end, we obtained 1,448 refactoring operations from 619 projects, while no relevant refactorings were detected in the other 881 projects.

2) Naturalness measurement: Like the work by Tu *et al.* [3] and Ray *et al.* [4], we use *cross-entropy* to assess the naturalness of code components. The idea behind *cross-entropy* is that if a code snippet is more natural, it will be more likely to appear in the training corpus. The *cross-entropy* of a code snippet S composed by tokens $t_1 ... t_n$ of length N is calculated as

$$H_M(S) = -\frac{1}{N}\log_2 P_M(S) = -\frac{1}{N}\sum_{1}^{N}\log_2 P(t_i|h)$$

where $P_M(S)$ and $P(t_i|h)$ are the probabilities estimated by the language model M, t_i is the token to be predicted, and h is the preceding tokens followed by t_i. In our study, we adopted the cache language model proposed by Tu *et al.* [3]. This model combines a traditional n-gram language model and an added "cache" component to exploit the localness property of source code. Like other statistical language models, it learns

[2] https://github.com/

[3] https://help.github.com/articles/fork-a-repo/

[4] https://help.github.com/articles/about-stars/

from a corpus of source code, and then predicts the probability P of occurrence for each token in the new file. In practice, a low *cross-entropy* indicates high *naturalness*.

To understand how naturalness changes due to refactoring, we measure the naturalness for every commit that has a refactoring operation. For each refactoring operation, we construct a training corpus, which is composed of all the files in the commit before the refactoring, excluding the files being refactored. This corpus is used to compute the *cross-entropy* of the excluded files and their corresponding version after refactoring.

D. Data Analysis

We compare the *cross-entropy* change caused by each type of refactoring operation via violin plots. The comparison of *cross-entropy* of files before and after refactoring is also performed via statistical tests by using the Wilcoxon signed-rank test [9], with results intended as statistically significant at $p \leq 0.05$. We also estimate the magnitude of the differences by using the effect size r, which can be used for the Wilcoxon signed-rank test [10]. We follow well-established guidelines to interpret the effect size: negligible for $|r| < 0.10$, small for $0.10 \leq |r| < 0.3$, medium for $0.3 \leq |r| < 0.5$, and large for $|r| \geq 0.5$ [11].

IV. Preliminary Results

We first provide an overview of how the code naturalness changes after refactoring with statistical analysis. Then, we give some concrete examples of refactoring activities that had a positive/negative effect on the code naturalness. In the end, we compare our results with those achieved in the study by Arima *et al.* [8].

A. Statistical Analysis of Results

Table II reports the impact of the 1,448 detected refactoring operations on the cross-entropy of the involved code components. Despite the quite large set of refactoring operations considered in our study, it is worth noticing that the mined refactorings are not equally distributed regarding their refactoring type. Indeed, "Extract Method" and "Rename Method" account for 66.0% of the total refactorings. Among all refactoring types, "Push Down Method" and "Push Down Field" are the least performed, and account only for 1.0% of the overall dataset. In the following analyses, these two types of refactorings are excluded due to the low number of occurrences.

For all these refactorings, we calculated the cross-entropy change (*i.e.,* the difference between the cross-entropy after refactoring and cross-entropy before refactoring) of the file being refactored. When reading the table, we have to be aware of the fact that high cross-entropy stands for low naturalness. Therefore, when the cross-entropy change is above zero, the naturalness of the code actually drops. Similarly, the naturalness increases when the cross-entropy change is negative.

Table II shows that overall, although the decrease of cross-entropy (increase of naturalness) is more common than the increase of cross-entropy (decrease of naturalness), the difference is not substantial (*i.e.,* 50.8% *vs* 44.5%). When it comes to specific refactoring types, "Inline Method", "Pull Up Method", "Rename Method", and "Move Method" are more likely to reduce the code naturalness. Among these five refactoring types, "Pull Up Method" has the highest possibility (73.3%) to reduce the naturalness. All other refactoring types tend to increase the code naturalness, despite the fact that there is still a large percentage of cases in which the naturalness decreases. Thus, our preliminary analysis of the achieved results does not show any clear relationship between refactoring and code naturalness.

To better understand the impact of refactoring operations on the code naturalness, we applied statistical tests to the cross-entropy values before and after refactoring for all the files being refactored. In Table III, we can find that for half of the refactoring types, there is no statistically significant difference (p-value $\geq$ 0.05) between the cross-entropy before and after refactoring. Meanwhile, the magnitude of the difference is mostly limited (with negligible or small effect size). The only exception here is the "Pull Up Method" refactoring. The comparison of cross-entropy values result in a statistically significant difference (p-value < 0.05), with a medium effect size. The result is in line with our findings from Table II.

To further understand how the impact of different types of refactoring on code naturalness differs, we also visualize the cross-entropy difference with violin plots in Fig. 1. In the violin plots, the thickness of the outer layer represents how likely the cross-entropy change will fall into this value. In the center of each violin plot, the white dot represents the median; the thick black bar represents the interquartile range, and thin black line represents the 95% confidence interval.

Looking at Fig. 1 we can see that "Extract Method", "Pull Up Field", "Rename Method", and "Extract And Move Method" refactorings are the least likely to impact the code naturalness, as most of the cross-entropy changes are close to zero. "Pull Up Method" can often bring large naturalness change to files, especially by reducing the code naturalness.

B. Examples of Cross-Entropy Change

To gain a more intuitive impression on how refactoring impacts the code naturalness, we extracted some examples from our dataset.

"Inline Method" refactoring was performed on the class "`View`" from the project "*Carbon*"[5]. In this refactoring operation, the calls to method "`setTint`" were replaced with the body of "`setTint`", consisting in a call to the method "`setTintList`". The replaced method "`setTint`" was also deleted in the class. After this refactoring, the cross-entropy of this class file increased from 2.418 to 2.430, thus resulting in a reduction of code naturalness. Intuitively, since the refactored method was used multiple times in the class, one might think that the increase of cross-entropy was caused by the fact that the replaced token "`setTint`" is much more common (*i.e.,* has a lower cross-entropy) in the source code than "`setTintList`". We inspected the cross-entropy of each token in the class

[5] https://goo.gl/NBRBah

TABLE II: Detected Refactorings and their Impact on the Code Naturalness

Refactoring type	**total**	**# cross-entropy increased**	**# cross-entropy unchanged**	**# cross-entropy decreased**
Extract Method	488	174 (35.7%)	0 (0.0%)	314 (64.3%)
Inline Method	57	37 (64.9%)	0 (0.0%)	20 (35.1%)
Pull Up Method	45	33 (73.3%)	0 (0.0%)	12 (26.7%)
Push Down Method	5	2 (40.0%)	0 (0.0%)	3 (60.0%)
Rename Method	468	220 (47.0%)	68 (14.5%)	180 (38.5%)
Move Method	126	76 (60.3%)	0 (0.0%)	50 (39.7%)
Extract and Move Method	162	60 (37.0%)	0 (0.0%)	102 (63.0%)
Pull Up Field	18	7 (38.9%)	0 (0.0%)	11 (61.1%)
Push Down Field	10	4 (40.0%)	0 (0.0%)	6 (60.0%)
Move Field	69	32 (46.4%)	0 (0.0%)	37 (53.6%)
Sum	1,448	645 (44.5%)	68 (4.7%)	735 (50.8%)

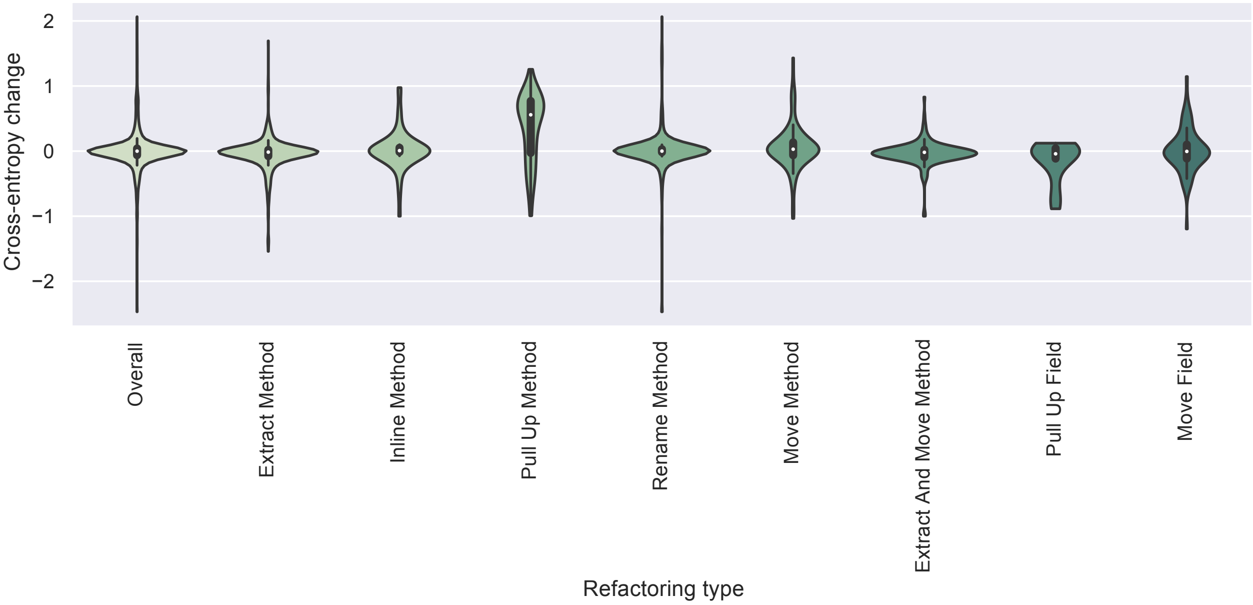

Fig. 1: Cross-entropy change after refactoring

TABLE III: Statistical tests of file cross-entropy before and after refactoring

Refactoring type	**P-Value**	**Effect Size**
Extract Method	< 0.001	0.180 (small)
Inline Method	0.202	0.119 (small)
Pull Up Method	< 0.001	0.414 (medium)
Rename Method	0.177	0.044 (negligible)
Move Method	0.029	0.138 (small)
Extract and Move Method	< 0.001	0.213 (small)
Pull Up Field	0.122	0.258 (small)
Move Field	0.727	0.030 (negligible)
Overall	0.453	0.003 (negligible)

before and after the refactoring to verify this assumption. However, we found out that the cross-entropy of the tokens "`setTint`" and "`setTintList`" are actually similar (whose value varies in different token positions due to the difference of preceding tokens). As a matter of fact, the removed tokens with significantly lower cross-entropy were those composing the method declaration, such as "`public`" and "`void`". Indeed, since the idea behind naturalness is based on the repetitiveness of tokens, these reserved keywords often have a much lower cross-entropy. One interesting direction to explore in future is how the cross-entropy of identifiers, which are the tokens carrying semantic information, changes during refactoring.

"Extract Method" refactoring was performed on the class "`CacheHandler`" from the project "*AutoLoadCache*"[6]. In this refactoring operation, multiple lines of code in the method "`proceedDeleteCacheTransactional`" were moved to a newly created method "`clearCache`", and these lines were replaced with a call to "`clearCache`". After refactoring, the cross-entropy of this class file was reduced to 3.776 from 3.813, namely the code naturalness increased. "Extract Method" is the opposite operation of "Inline Method", therefore, it is unsurprising that the naturalness change caused by "Extract Method" displays an opposite trend. Similarly, the major

[6] https://goo.gl/r4FE26

difference between the versions (before and after refactoring) is the extra tokens needed for declaring the new method.

C. Comparison with the Study by Arima et al.

We compare the results we achieved with the results from the study by Arima *et al.* [8]. Some interesting facts are spotted.

In our study, only 50.8% of the total refactorings increase the code naturalness, which is much lower than what has been observed in [8] (67.9%). The reason behind this different finding could be explained by the different datasets employed in the two studies. First, the dataset used in [8] is composed of only 28 refactorings (as compared to the 1,448 considered in our study), thus possibly indicating peculiarities of the specific refactoring operations considered. Second, the 28 refactorings used in [8] have all been mined from a single, well-known project, namely JUnit 4, while in our study we extracted the studied refactorings from a variegated set of 619 projects. It is possible that the "quality" of the refactorings applied in JUnit 4 is higher, thus resulting in a naturalness increase that we did not observe in our dataset. Clearly, this is only an assumption, which needs to be carefully verified. However, it also indicates a direction to work with: We might need to better understand the association between code quality and naturalness, which is not fully disclosed in the research community.

In the work of Arima *et al.* [8], 7 out of 9 (77.8%) "Extract Method" refactorings increase the code naturalness, which is in line with our result: 64.3% of the "Extract Method" refactorings result in increased naturalness. Although no significant difference between the cross-entropy before and after refactoring was found during our statistical analysis, there are indications that "Extract Method" refactoring might help in improving the naturalness of code. Similarly, 2 out of 3 "Inline Method" refactorings in their study lead to a naturalness decrease, meanwhile, the same trend applies to 64.9% of our cases. However, since they inspected a smaller number of "Inline Method" refactorings, much more refactorings need to be examined to make a solid comparison.

V. Threats to Validity

Threats to construct validity concern the relation between theory and observation. In this work, we use RMiner to detect refactorings. While the precision achieved by this tool is very high [5], we are aware that our results can be affected by the presence of false positives. Also, RMiner can identify a specific set of refactoring operations, while the definition of refactoring is broader.

Threats to internal validity concern external factors we did not consider that could affect the variables and the relations being investigated. In our study, when calculating the entropy for source code, we did not experiment with all possible configurations of the used language model. An adapted 3-gram model with an additional cache is used. We do not expect to observe a significant difference in the overall result trend with different configurations.

Threats to external validity concern the generalizability of our findings. While we investigated a large number of refactoring operations, we are aware that only Java and open source software projects are considered in our study.

VI. Conclusion and Future Work

We investigated how refactoring impacts the naturalness of source code by inspecting 1,448 refactoring operations from 619 Java projects. We studied the impact of refactoring types on the naturalness of the modified code components. Our results show that refactorings do not necessarily make source code more natural, and that naturalness changes in different ways for different types of refactorings.

Our study serves as the first step towards using naturalness information to support refactoring activities. In the future, we will conductmore thorough empirical studies to understand the correlation between refactoring quality and code naturalness. That is, we would like to examine whether naturalness can be a good indicator for effective refactorings with high quality code. We will also investigate the possibility of use the naturalness of source code combined with other metrics, such as Chidamber and Kemerer metrics [12], to support the identification of code components in need of refactoring.

Acknowledgments

We gratefully acknowledge the financial support of the Swiss National Science Foundation for the projects PROBE (SNF Project No. 172799) and JITRA (SNF Project No. 172479).

References

[1] M. Gabel and Z. Su, "A study of the uniqueness of source code," in *Proceedings of FSE 2010 (18th ACM SIGSOFT International Symposium on Foundations of software engineering).* ACM, 2010, pp. 147–156.

[2] A. Hindle, E. T. Barr, Z. Su, M. Gabel, and P. Devanbu, "On the naturalness of software," in *Proceedings of ICSE 2012 (34th International Conference on Software Engineering).* IEEE, 2012, pp. 837–847.

[3] Z. Tu, Z. Su, and P. Devanbu, "On the localness of software," in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering.* ACM, 2014, pp. 269–280.

[4] B. Ray, V. Hellendoorn, S. Godhane, Z. Tu, A. Bacchelli, and P. Devanbu, "On the naturalness of buggy code," in *Proceedings of ICSE 2016 (38th International Conference on Software Engineering).* ACM, 2016, pp. 428–439.

[5] N. Tsantalis, M. Mansouri, L. M. Eshkevari, D. Mazinanian, and D. Dig, "Accurate and efficient refactoring detection in commit history," in *Proceedings of ICSE 2018 (40th International Conference on Software Engineering).* New York, NY, USA: ACM, 2018, pp. 483–494.

[6] B. Lin, L. Ponzanelli, A. Mocci, G. Bavota, and M. Lanza, "On the uniqueness of code redundancies," in *Proceedings of ICPC 2017 (25th International Conference on Program Comprehension).* IEEE, 2017, pp. 121–131.

[7] J. C. Campbell, A. Hindle, and J. N. Amaral, "Syntax errors just aren't natural: improving error reporting with language models," in *Proceedings of the 11th Working Conference on Mining Software Repositories.* ACM, 2014, pp. 252–261.

[8] R. Arima, Y. Higo, and S. Kusumoto, "Toward refactoring evaluation with code naturalness," in *Proceedings of ICPC 2018 (26th International Conference on Program Comprehension).* ACM, 2018, pp. 316–319.

[9] F. Wilcoxon, "Individual comparisons by ranking methods," *Biometrics bulletin*, vol. 1, no. 6, pp. 80–83, 1945.

[10] A. Field, *Discovering statistics using IBM SPSS statistics.* sage, 2013.

[11] J. Cohen, "A power primer." *Psychological bulletin*, vol. 112, no. 1, p. 155, 1992.

[12] M. Hitz and B. Montazeri, "Chidamber and Kemerer's metrics suite: A measurement theory perspective," *IEEE Transactions on software Engineering*, no. 4, pp. 267–271, 1996.