IEICE TRANS. INF. & SYST., VOL.E98–D, NO.7 JULY 2015

PAPER

Effect Analysis of Coding Convention Violations on Readability of Post-Delivered Code

Taek LEE[†a)], ***Student Member***, **Jung-Been LEE**[†b)], ***and*** **Hoh Peter IN**[†c)], ***Nonmembers***

SUMMARY Adherence to coding conventions during the code production stage of software development is essential. Benefits include enabling programmers to quickly understand the context of shared code, communicate with one another in a consistent manner, and easily maintain the source code at low costs. In reality, however, programmers tend to doubt or ignore the degree to which the quality of their code is affected by adherence to these guidelines. This paper addresses research questions such as "Do violations of coding conventions affect the readability of the produced code?", "What kinds of coding violations reduce code readability?", and "How much do variable factors such as developer experience, project size, team size, and project maturity influence coding violations?" To respond to these research questions, we explored 210 open-source Java projects with 117 coding conventions from the Sun standard checklist. We believe our findings and the analysis approach used in the paper will encourage programmers and QA managers to develop their own customized and effective coding style guidelines.

key words: *coding conventions, coding style standard, code readability, software quality, empirical study*

1. Introduction

In software development, producing code with high readability is essential. There are several reasons why this is an important concern.

Understanding the context of existing code is a necessity, not only to remind programmers of the context of what other programmers have written but also to understand their intention. Less readable code prevents programmers from understanding one another; highly readable code enables developers to easily communicate with each other.

Maintainability is another reason why code readability is important [1], [2]. To reduce maintenance costs, supporting code with high readability is a necessity. In a survey, a majority of developers (66%) agreed that the most serious problem affecting software development is understanding the code [3] and the most time-consuming activity during software maintenance is code reading [4]–[6]. In the maintenance process, it is typical that the programmer who initially developed the company's proprietary software is no longer available and thus a different developer must assume responsibility for these legacy systems. Reading and understanding the legacy code of existing software systems that others have developed is therefore an everyday maintenance issue. Therefore, less readable code can increase the cost of software maintenance.

Coding style conventions [7]–[9] aim at enhancing code readability. The primary purpose of coding style compliance is to develop more readable code and facilitate communication and collaboration between developers by maintaining a uniform coding style. For example, coding conventions usually cover file organization, indentation, comments, declarations, statements, white space, naming conventions, and programming practices.

The majority of programmers agree that coding conventions are necessary and that adherence to such conventions improves the final code quality. In reality, however, programmers rarely adhere to the conventions for various reasons. For example, Li and Prasad [10] found that although developers understood the importance of using code conventions, they did not follow them, especially when urgent code development was required. Even those programmers who wish to establish or adopt certain coding style guidelines in their company may have difficulty determining what to do first when subjected to limited time and resources.

Therefore, searching for an effective set of coding conventions to enhance code readability at low cost is a valuable endeavor. In this paper, to assess the degree to which intensive adherence to coding conventions improves code readability, we investigated 117 Sun Java coding conventions [11] for 210 open-source projects collected from sourceforge.net. For each of these 210 projects, we evaluated the conventions that were complied with or violated and then identified the rule violations degrading the code readability. We believe the findings and analysis approach utilized in this paper will assist programmers (or QA managers) to develop a customized, effective coding style guideline. In this paper, we are particularly interested in answering the following research questions (RQ):

- RQ1: "Do developer violations in coding conventions affect readability of the code that they produce?"
- RQ2: "What kinds of violation are particularly associated with declining code readability?"
- RQ3: "How much do factors such as level of developer experience, development team size, project size, and project maturity influence coding violations?"

Manuscript received October 3, 2014.
Manuscript revised February 4, 2015.
Manuscript publicized April 10, 2015.
[†]The authors are with Korea University, South Korea.
a) E-mail: comtaek@korea.ac.kr
b) E-mail: jungbini@korea.ac.kr
c) E-mail: hoh_in@korea.ac.kr (Corresponding author)
DOI: 10.1587/transinf.2014EDP7327

2. Related Work

Our study is primarily related to two aspects of the existing studies: one is measuring code readability; the other one is coding conventions. Many existing studies are founded on experience and feedback from educating students (or programmers) on coding conventions. Few have analyzed the correlation between programmer violations of coding conventions and the quality of the produced code.

2.1 Measuring Code Readability

Other than studies by Buse et al. [12] and Posnett et al. [13], papers attempting to objectively estimate code readability are limited.

A major contribution of Buse et al. [12] was that their estimation result for code readability was compared and verified with actual perception scores voted by humans. Buse et al. emphasized the importance of code readability [12]. To estimate readability of source code, they used such features as the line length, identifiers, identifier length, indentation, keywords, punctuation, operators, assignments, if-statements, loops, and blank lines. Furthermore, they employed machine-learning techniques to develop a prediction model using their feature set and verified the model validity and high-prediction performance with human reviewers.

Posnett et al. was based on the study of Buse et al. However, Posnett et al. significantly improved the performance of readability estimation by adopting three simple metrics, Halstead's volume metric, lines of code, and entropy of code tokens or characters. The readability estimation formula proposed in [13] was verified using 120 people and 100 samples of code as performed by Buse et al. The estimation accuracy was determined to be more than 80%, better than the performance result (approximately 75% on average) of the Buse formula [12].

There are other specific studies addressing indirect topics related to code readability. Aggarwal et al. [14] used the ratio of comments to lines of code for a code readability measurement. Flesch [15] and Börstler et al. [16] proposed a method to measure readability of natural language (English sentences) and compared it with Halstead & Cyclomatic complexity metrics.

Butler et al. [17] investigated eight open-source Java projects to analyze the impact of coding conventions (particularly, identifier naming) on code quality. They used eleven typographical and natural language-naming guidelines for Java to measure the quality of the identifier naming and FindBugs to measure the code quality. The authors found significant associations between a flawed identifier name and FindBugs warnings. Butler et al. [18] used three quality measurements: cyclomatic complexity, maintainability index, and readability metric to measure code quality. They found conclusively that poor-quality identifier names are strongly associated with more complex, less readable, and less maintainable source code.

2.2 Coding Conventions

Basic coding conventions of popular languages such as Java, C, and C++ are public and are widely available [7]–[9]. They are intended primarily to facilitate collaboration and maintenance of uniformity in code written by different developers.

Reed [19] introduced guidelines based on experience from teaching students Java programming. Reed claims that programming style is more than simple aesthetics and that compliance with coding conventions can assist early-stage programmers to avoid errors and better understand the underlying concepts.

Smit et al. [1] extracted 71 coding conventions from seven experts and prioritized them into three levels of importance. Five open-source projects were analyzed in terms of adherence to the coding conventions. Further, Smit et al. investigated the evolution of adherence to coding conventions by analyzing the revision history in a version control system over a software lifecycle. However, the authors focused more on maintainability than on readability. They did not study the impact of coding convention violations on code readability.

Mohan and Gold [2] conducted an investigation into how the programming style (typographical style) changes over the lifetime of a program and suggested a method to measure the typographical style. Interestingly, Mohan and Gold found that programmers have a tendency to distinguish code blocks using blank lines, reporting that the blank line metric is a good indicator of the application of the typographical style rule.

Elish and Offutt [20] analyzed 100 classes of open-source Java projects using sixteen standard coding practices by involving open-source developers. They found that only 4% of the subject classes had no violations. Moreover, there were positive correlations between the number of violations found in a class and the number of lines of code, number of methods, and number of attributes.

3. Experiment Process

This section presents the process overview (Fig. 1) of our proposed analysis approach to explore the three research questions identified in Sect. 1. Each step in Fig. 1 will be explained one by one in the following subsections.

3.1 Data Collection (STEP 1)

For the experiment, we collected sample data of open-source Java projects from the website sourceforge.net. We collected not only source code files but also necessary metadata about developer experience, team size, project size, and project maturity. For developer experience, we collected data such as a period of developers' project involvement and the number of projects that they involved. For team size of a project, we collected a list of involved developers. For

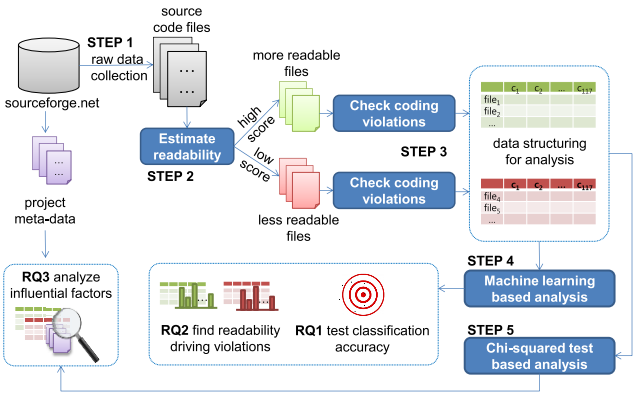

Fig. 1 Overview of the proposed experiment process.

project size, we investigated total lines of all the source code files consisting of a project. For project maturity, we used the definition of seven discrete stages originally provided by the website, such as Planning, Pre-alpha, Alpha, Beta, Production/stable, Mature, and Inactive.

3.2 Readability Estimation (STEP 2)

Each of the files collected in STEP1 was quantitatively measured in terms of code readability. For this measurement, we utilized the readability estimation formula proposed by Posnett et al. [13] that generates a readability score. The higher the score, the more readable the source code. All the source code files were rated using this readability estimation formula. In order of readability score, files ranked at top 25% and bottom 25% were considered sample groups of "more readable files" and "less readable files", respectively. To avoid using files with ambiguous boundary measures in code readability, we did not include files of the middle scoring range in sampling.

3.2.1 Readability Estimation Formula

Based on the study of Buse et al. [12], Posnett et al. designed a simple yet more accurate formula from the estimation performance viewpoint. This estimates the readability of a given code block as follows [13]:

$$\begin{aligned} \mathit{Readability} &= 8.87 - 0.033 \times \mathit{Halstead\ volume} \\ &\quad + 0.4 \times \mathit{total\ lines\ of\ code} - 1.5 \times \mathit{code\ entropy} \quad (1) \end{aligned}$$

Halstead's metrics are source code complexity metrics introduced in 1977 by Maurice Howard Halstead [21]. The *Halstead volume* metric is defined as $\mathit{Halstead\ volume} = N \times \log_2^n$, where N is called the *program length*, the sum of the total number of operators (N_1) and operands (N_2) in the code, $N = N_1 + N_2$, and n is called the *program vocabulary*, the sum of the number of unique operators (n_1) and unique operands (n_2), $n = n_1 + n_2$. The term *total lines of code* is the number of lines of the given code block. The term *code entropy* is an entropy measurement for the given code block. Entropy is considered the degree of disorder, or the amount of information in a signal or dataset. Entropy is calculated from the count of terms (tokens or characters) comprising the given code block. Assume that X is a document and x_i is a term in X. $\mathit{Count}(x_i)$ is the number of occurrences of x_i in the document X and $p(x_i) = \frac{\mathit{count}(x_i)}{\sum_{j=1}^{n} \mathit{count}(x_j)}$. Entropy $H(X)$ is defined as $H(X) = -\sum_{i=1}^{n} p(x_i) \log_2^{p(x_i)}$. The terms were considered as tokens or characters such that both character-based entropy and token-based entropy were compared in [13]. In our study, we replicated a formula version adopting the character-based instead of token-based entropy (i.e., x_i is one of the characters comprising the code in $H(X)$) because it was verified in [13] that readability estimation computed with character-based entropy was more accurate than token-based entropy in performance evaluation.

3.2.2 Readability Estimation at the File Level

The sample unit in our experiments is files (Fig. 1). Therefore, we needed to estimate the readability score of a file, so we utilized the readability estimation formula of Eq. (1) introduced in Sect. 3.2.1.

First, we divided the file into multiple code blocks each having 46 lines of code (LOC), which is the maximum block size used in [13]. If the file was smaller than 46 LOC, we did not divide it but treated the whole file as a single code block; however, a file whose size was smaller than four LOC was not included as a sample. We then computed the readability score of each of the divided code blocks using the estimation formula given by Eq. (1). Then, we averaged the readability scores of the code blocks to obtain the readability score for the given file.

We used the mean of the code-block readability scores as a representative measure of file-level readability. However, it might also be possible to consider the minimum of the code-block readability scores as the representative one. A reader can have an impression that a file is hard to review because just one particular code block of the file is regarded as seriously illegible (i.e., file-level readability depends on the lowest readability score for the code blocks). Another possible candidate is the mode of the code-block readability scores. It seems reasonable that file-level readability can be determined by the most frequently observed value among the readability scores of the code blocks comprising the file. These three methods of determining a representative statistic (mean, minimum, and mode) are discussed in Sect. 4.1.

3.3 Checking Coding Violations (STEP 3)

To determine if a file violated any coding conventions, we used Checkstyle, a popular static analysis tool [22], to automatically analyze well-known coding conventions. Checkstyle supports checking 117 standard coding conventions of Sun Java [11] in 12 categories. The categories are "annotations", "block checks", "class design", "coding", "import", "Javadoc comments", "metrics", "miscellaneous", "modifiers", "naming conventions", "size violations", and

"whitespace". Checkstyle generates four severity levels of alarm when a checked coding convention is violated; these are "ignore", "info", "warning", and "error". In our experiment, only the levels "warning" and "error" were considered.

Results of coding violation checks were structured in a tabular format for the subsequent analysis process. Each row in the table is a file sample and each column represents a violation check. A cell in the table is the number of violations per line of code in the file sample, as reported by Checkstyle. Because a large file generates a higher number of violation messages than a small file does, we normalized the number of observed coding violations with the number of lines of code of the given file.

3.4 Machine Learning-Based Analysis (STEP 4)

This step aims to explore RQ1 and RQ2. We interpreted and modeled the problem domain of RQ1 into a classification problem (i.e., supervised machine learning). In the classifier modeling, the features for model construction were checks themselves, and the classes to predict were binary quality labels of code readability (i.e., "more readable" or "less readable"). If the constructed classification model demonstrates high accuracy and outstanding performance, it implies that the features (i.e., coding violations) used in the model construction are useful factors for classifying files into the two sample groups, "more readable files" and "less readable files". That is, the distributions of coding violations observed in the two sample groups are sufficiently different, and can therefore be of significant use in predicting code readability. In the classification experiment, we used four of the most well-known algorithms: C4.5 (decision tree), k-NN, SVM, and Naïve Bayesian, which are popular classifiers among the top ten algorithms in the field of data mining [23]. To perform the experiments, we used the data-mining tool Weka [24] and its built-in default parameters.

For an evaluation measure of classification performance, we used area under the curve (AUC) [25], which is interchangeably called area under the receiver operating characteristic (ROC) curve [25]. The curve is defined by plotting the true positive rate against the false positive rate at various threshold settings. A binary classifier having an AUC measure close to 1 (the perfect classification) is considered good, while a classifier having an AUC measure close to 0.5 (the worst classification) is considered poor.

To avoid getting a biased result of a performance evaluation by chance, we used 10-fold cross validation [25] to get an answer for RQ1. In the validation process, the dataset from STEP 3 is divided into ten folds. Nine folds are used for model training, and one fold is used for model testing. In this method, each fold is rotated ten times for training and testing purposes.

Answering RQ2 is an exercise of searching for valid contributors of features for improving classification performance. If any coding violation is strongly associated with higher or lower code readability, the distribution of the coding violation feature will be observably different for the two sample groups of "more readable files" and "less readable files", meaning that the feature is a factor significantly influencing code readability.

To determine these contributing features (addressing RQ2), we used a feature selection algorithm called correlation-based feature selection (CFS). CFS is an algorithm that is used to identify a subset of features of reduced size without irrelevancy or redundancy between features in classification problems [26]. In our experiment process, CFS selects a subset of coding violations that are highly correlated with code readability while having a low intercorrelation between coding violations.

In addition, we performed sensitivity analysis to understand the correlation between the CFS-nominated coding violations and code readability. For the sensitivity analysis, the response of the code readability to incremental changes in the number of coding violations was simulated. While controlling the number of coding violations in the sample groups of files from the minimum to the maximum by 10% stepwise increments, we observed the number (portion) of "more readable" code files out of the total number of files at the controlled time. The results are presented in Sect. 4.2.

3.5 Chi-Squared Test-Based Analysis (STEP 5)

To address RQ3, we analyzed developer and project factors that can influence the extent of coding violations observed in the sample files. The chi-squared test method aims to explore the dependency between four paired variables: developer experience vs. coding violations, team size vs. coding violations, project size vs. coding violations, and project maturity vs. coding violations. The details will be addressed one by one in the subsections of Sect. 4.3.

Pearson's chi-squared test was specifically used in our experiments to assess whether paired observations on the two interesting variables expressed in a contingency table are independent of each other. In the chi-squared tests of Sect. 4.3, the null hypothesis (H_0) is "there is no correlation between the two variables (no dependency exists)" and the alternative hypothesis (H_a) is "there is a correlation between the two variables (a dependency exists)". We will confirm rejecting H_0 and accepting H_a with the Pearson's chi-squared test.

One variable in the contingency table was the number of coding violations. We assigned the extent of coding violations to the sample files using the binary quality labels "more violation" and "less violation". As a criterion for determining the binary quality labels, we used the median of the number of coding violations (per lines of code) for the sample files. If the violation frequency of a file was greater than the median, the file was labeled "more violation"; otherwise, the file was labeled "less violation".

The other variable in the contingency table represented one of the four factors developer experience, team size, project size, or project maturity, each of which had nomi-

nal values of five levels except project maturity, which had seven levels (Sect. 4.3.4). For example, in the case of level of experience (LOE) of a developer, the five levels were LOE1, LOE2, LOE3, LOE4, and LOE5 (Sect. 4.3.1). These five levels were determined in the following manner. First, LOEs of developers were measured by a formula, details of which are explained in Sect. 4.3.1. Second, the LOEs were sorted in ascending order. Finally, the sorted LOEs were divided into five 20-percentile intervals: $[0\%, 20\%)$, $[20\%, 40\%)$, $[40\%, 60\%)$, $[60\%, 80\%)$, and $[80\%, 100\%]$. The symbol "[x" means that the value x is included, and the symbol "x)" means that the value x is not included within the interval. Each of the boundary LOEs that divide the five intervals was determined in this way. Similarly, the variables of team size and project size were divided into five intervals represented by the 20-percentile discretization. With the discretization, each of the five intervals receives the same number of allocated samples, thereby avoiding the trouble of unbalanced sample sizes when computing and interpreting expected frequencies of a nominal value of the variable in a contingency table.

As discussed above, the degrees of freedom (df) about the variables used in the chi-squared tests of Sects. 4.3.1, 4.3.2, and 4.3.3 was four (i.e., one variable having binary values and the other variable having five nominal values). However, df in Sect. 4.3.4 was six (i.e., one variable having binary values and the other variable having seven nominal values). For the statistical tests, a 95% confidence interval was assumed in our study.

As the classification modeling approach was proper and sufficient for answering RQ1 and RQ2, we did not find it necessary to apply a chi-squared test method for these two research questions. The success of accurate classification served to substantiate the claim that a dependency exists between the variables of interest (i.e., features and class), obviating the need to apply a chi-squared test for RQ1 and RQ2.

4. Experiment Results

This section presents results from our experiment to answer the three research questions identified in Sect. 1. In the experiment, for the project data, we identified ten different domains: "audio and video", "business enterprise", "communication", "development", "games", "graphics", "home and education", "science engineering", "security utilities", and "system administration". For each domain, we selected the three most popular projects in seven different levels of project maturity. As a result, we analyzed 129,147 source code files (∗.java) from the 210 projects ($210 = 3 * 7 * 10$, i.e., the three most popular projects for each of seven maturity status levels in ten different domains). Table 1 briefs the statistics of the data experimented.

The following subsections introduce the experiment results in detail for each individual research question.

4.1 RQ1: "Do Developer-Coding Violations Affect the Readability of the Produced Code?"

To determine if RQ1 is true, as mentioned in STEP 4 of Sect. 3.4, we constructed classification models with several popular machine-learning algorithms. We wished to confirm that the results of the performance evaluation of the constructed models were consistently valid regardless of the algorithm used.

We evaluated the classification models in terms of the AUC measure. The classification experiment was performed for each of the three different class-labeling strategies (i.e., mean, minimum, and mode, as mentioned in Sect. 3.2.2).

As shown in Fig. 2, the AUCs of all the classification models were greater than 0.5, regardless of the algorithm and the class-labeling strategy used in the model construction. For example, the C4.5 algorithm yielded approximately 84% better performance than a random classifier (i.e., 0.92 compared with 0.5) in terms of AUC. In our experiment setup, the random classifier has AUC = 0.5 because the two sample groups, those of "more readable" and "less readable" files, were the same size.

Conclusively, we could infer that RQ1 is true based on the experiment results in Fig. 2. The fact that the performance of classification models is better than the random classifier indicates that coding violations are observed differently in each of the two sample groups of "more readable" and "less readable" files. This automatically supports our inference that coding violations during development affect the readability of the produced code.

Table 1 Summary of project data analyzed in the experiment.

Project size (LOC)	Number of projects	Average number of source code files	Project percentage
0-10K	71	28	34%
10-20K	18	91	9%
20-40K	37	140	18%
40-80K	21	236	10%
80-160K	28	542	13%
Over 160K	35	2297	17%

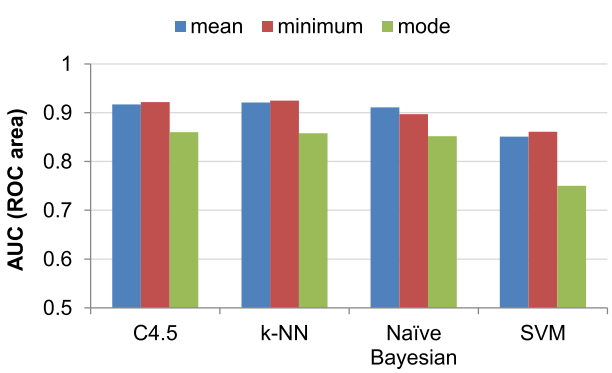

Fig. 2 Performance evaluation of classification models for predicting code readability.

4.2 RQ2: "What Kind of Coding Violations Are Particularly Associated with Declining Code Readability?"

As described in Sect. 3.4, we performed a CFS analysis using as input data the 117 checks for coding violations and the readability scores of the files; here, the mean-based estimation (Sect. 3.2.2) was used to obtain the file-level readability scores. Table 2 is the result generated by CFS with 10-fold cross validation. The whole sample data were divided into ten folds and then iterated such that CFS was applied to each of the 10 folds. The results of Table 2 include features of coding violations that were nominated more than 90% by CFS in the ten iterative selection rounds. Some of coding violations found in Table 2 concurred with findings by a study of Elish and Offutt [20].

In Table 2, TrailingCommentCheck is the check to ensure that comments are on a separate line. IndentationCheck is the validation that ensures that a proper number of spaces are used for indentation of new levels of code lines. IllegalTokenCheck checks for illegal tokens. Certain language features often lead to hard-to-maintain code or are not obvious to novice developers. RedundantModifierCheck is to identify redundant modifiers in interface and annotation definitions, the final modifier on methods of final classes, and inner interface declarations with static. HiddenFieldCheck verifies that a local variable or a parameter does not shadow a field that is defined in the same class. WriteTagCheck verifies that source code outputs a JavaDoc tag as information. LeftCurlyCheck is to verify the placement of left curly braces ("{") for beginning code blocks such as if-else, try, and catch tokens. UnnecessaryParenthesesCheck checks for the use of unnecessary parentheses. TypecastParenPadCheck is to check the policy on the padding of parentheses for typecasts. That is, whether a space is required after a left parenthesis and before a right parenthesis or such spaces are forbidden. JavadocStyleCheck is the check that validates Javadoc comments to ensure they are well formed. JavadocMethodCheck checks if a method or constructor has Javadoc comments, and likewise JavadocVariableCheck checks if variables have Javadoc comments. More detailed explanation of the coding violations listed in Table 2 is on the reference web page [11].

The information regarding the correlation between the coding violations listed in Table 2 and the affected code readability is an excellent source for understanding the violations that should be considered to be refactored or corrected to enhance the readability of code. Figure 3 is the sensitivity analysis results for presenting the correlation.

Table 2 Coding violations most likely to affect code readability.

Names of the coding violations (selected by CFS)	
TrailingCommentCheck	LeftCurlyCheck
IndentationCheck	UnnecessaryParenthesesCheck
IllegalTokenCheck	TypecastParenPadCheck
RedundantModifierCheck	JavadocStyleCheck
HiddenFieldCheck	JavadocMethodCheck
WriteTagCheck	JavadocVariableCheck

As indicated in Fig. 3, the coding violations listed in Table 2 showed different degrees and directions of sensitivity. In Fig. 3, lines above 0.5 on the Y-axis are coding violations that reduce code readability if they are frequently observed in source code files. In particular, TrailingCommentCheck was found to be the most sensitive one, indicating that a trailing comment attached on the same line as code is actually not a good practice in terms of code readability.

Lines below 0.5 on the Y-axis, on the other hand, are coding violations that increase code readability if they are frequently observed. The checks of JavadocMethod, JavadocVariable, and Indentation were the most sensitive violations.

The sensitivity result of checks for JavadocMethod and JavadocVariable indicates that the more highly readable files had fewer Javadoc comments, in terms of both methods and variables.

The number of blank spaces required for Indentation in the checking tool Checkstyle was four for a new indentation level, case label in switch statements, and throw statements on a new line. However, more than four spaces could be inserted for improved code readability. Files that were more readable actually did not have limitation of four spaces in the indentation size.

To achieve a higher level of categorical understanding than the coding violations identified in Fig. 3, we performed the sensitivity analysis in terms of the 12 categories listed in Sect. 3.3. Figure 4 is the result for the 12 categories. The categories included all of the 117 coding violations.

Code readability was relatively insensitive to increases

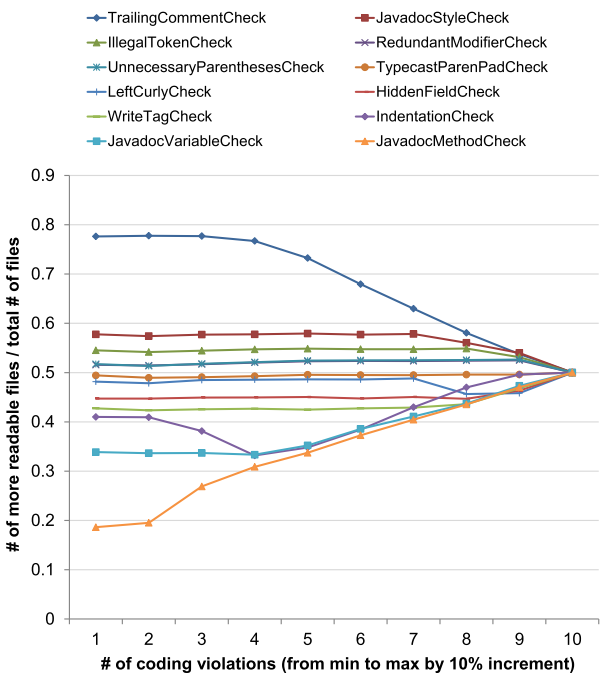

Fig. 3 Sensitivity analysis between coding violations and code readability (major violations reported in Table 2).

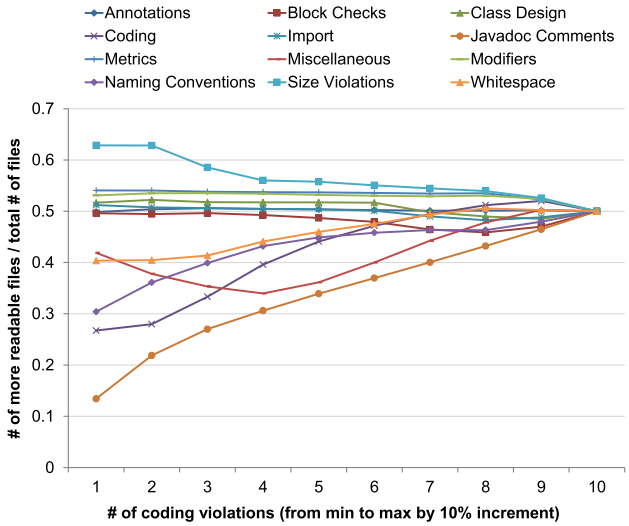

Fig. 4 Sensitivity analysis between coding violations and code readability (in terms of 12 categories).

and decreases in the coding violations, especially in the categories of Annotations, Block Checks, Class Design, Imports, Metrics, and Modifiers.

Conversely, categories of Size Violations, Naming Conventions, Coding, and Javadoc Comments were quite sensitive ones. For example, code readability decreased (i.e., 21% drop from 0.63 to 0.5 in the Y-axis) as the number of coding violations increased in the category of Size Violations. In addition, the most sensitive category was Javadoc Comments (i.e., 3.85 times gap from 0.13 to 0.5 in the Y-axis). Comments related to Javadoc are negative at least in terms of improving code readability even though comments are usually considered helpful in understanding code context. As shown in Fig. 4, the files that had many violations in the category Javadoc Comments were relatively more readable than the files that had fewer in the category. The category Javadoc Comments includes, for example, coding violations of JavadocStyle, JavadocVariable, JavadocMethod, and WriteTag mentioned in Table 2.

4.3 RQ3: "How Much Do the Factors of Developer Experience, Team Size, Project Size, and Maturity Influence Coding Violations?"

Through the analysis of RQ1 and RQ2, we could confirm that violations of coding conventions affect the readability of post-delivered code. Perhaps the next resulting question could be "What kind of developer and project factors are related to committing violations?" By analyzing project metadata as mentioned in Sect. 3.1, we respond to this question (RQ3).

4.3.1 Developer Experience vs. Coding Violations

In this section, we are interested in two variables and their dependency. One variable is level of experience (LOE) of the developers; the other variable is extent of coding violations that the developers commit.

Table 3 Brief statistics on each LOE of developers in terms of years and number of projects that they experienced (figures are averages).

LOE stage	Years of project involvement	The number of projects involved
LOE1	3	1.1
LOE2	9.7	1.3
LOE3	11.4	2.5
LOE4	12.1	4.5
LOE5	12.6	13.6

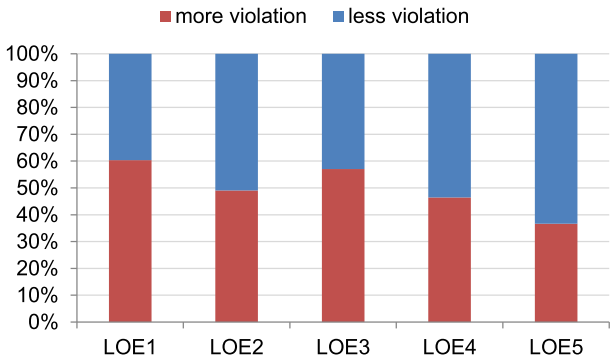

Fig. 5 Correlation between LOE of developers and their coding violations.

To measure the LOE of a developer, we considered the number of projects and the total years related to the projects during which the developer had an involvement with sourceforge.net. Thus, in this study, we defined LOE as follows:

$$LOE = \textit{the number of involved projects} * \textit{total years of the project involvements} \quad (2)$$

We surveyed the LOE range with five discrete stages: LOE1 through LOE5 as explained in Sect. 3.5. Table 3 presents brief statistics on the LOE stages in our study; the figures of years and number of project involvements are averages from the analyzed samples for each LOE stage.

We conducted a Pearson's chi-squared test for the two variables: LOE of developers and the extent of coding violations reported from the files that they developed. In the chi-squared test, observed frequencies of samples in the contingency table were $((7491, 5818, 8138, 6027, 4740), (4928, 6046, 6123, 6957, 8184))$. x^2-value was 547.67 and p-value was less than 2.2e-16. Conclusively, we could statistically confirm that the LOE of the developers affects the developer-coding violations.

Figure 5 illustrates the results of the correlation between LOE and coding violations. From Fig. 5, we see a trend that as LOE increased, fewer violations in the sample files were observed. The coding violation gap between LOE1 and LOE5 was about 1.64 times (60% vs. 37%). That is, developers with approximately 13 years and 14 projects (LOE5, Table 3) tended to commit 38% (= (60 − 37)/60) fewer violations than those with approximately 3 years and one project (LOE1, Table 3).

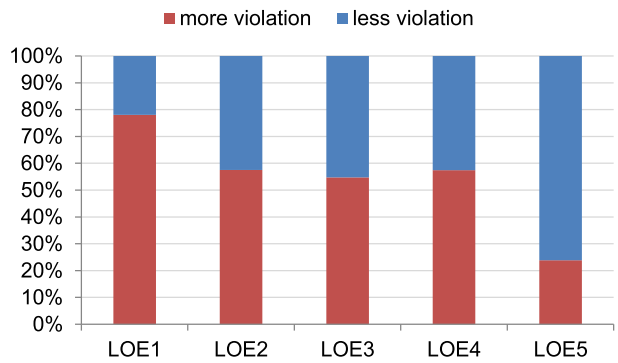

Fig. 6 Correlation between LOE of developers and their coding violations (after omission of the data that were anomalous).

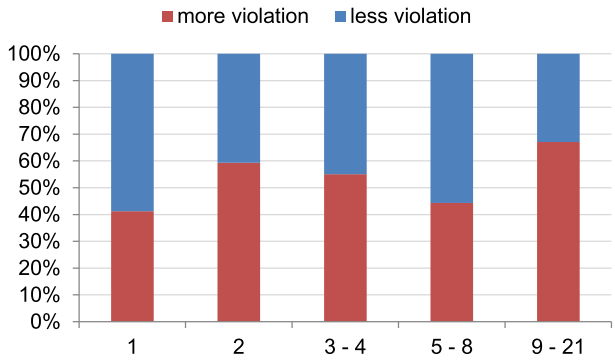

Fig. 7 Correlation between team size and coding violations.

It is interesting to note that developers of LOE3 committed slightly more violations than did developers having a somewhat greater or lesser LOE (i.e., LOE2 or LOE4). We can speculate that developers of LOE3 may become overconfident and more careless about violations of coding conventions compared with developers of LOE2 or LOE4. However, we also find that developers who had higher counts of project involvement experiences than did those of LOE3 showed comparatively fewer coding violations. Possibly, then, developers who are beyond a certain LOE recognize the importance of coding conventions and make a further effort to comply with them.

We observe in passing that while exploring the data for this section, we found that some of the project meta-data collected seemed anomalous. For example, there were developers who had had more than ten years of project activity period yet were involved in just only one project during that long period; it is usual that developers who have many years of project experience tend to involve in several projects. Therefore, we reran the chi-squared test and the analysis done in Fig. 5 after filtering out these anomalous sample data. We found that the chi-squared test run against these filtered data was still positive; that is, we could still statistically confirm the fact that the LOE of developers affects the number of their coding violations. However, the trend was somewhat different from that shown in Fig. 5; Figure 6 shows the new result after filtering out the anomalous data. In Fig. 6 the correlation trend appeared stronger than in Fig. 5. For information, the linear trend (regression) line of Fig. 5 was $Y = -0.05 \cdot X + 0.65$ ($R^2 = 0.72$). In contrast, the linear trend line of Fig. 6 was $Y = -0.11 \cdot X + 0.87$ ($R^2 = 0.78$). From this exploratory comparison analysis, we found that the number of involved projects plays a substantial role in LOE and consequently affects coding violations.

4.3.2 Team Size vs. Coding Violations

Adherence to a guideline of coding conventions is challenging as more developers join a project. A developer working on a single-developer project may comply more easily with a guideline than a developer who joins a multiple-developer project.

We conducted a Pearson's chi-squared test for the two variables: team size (i.e., the number of developers involved in a project) and the extent of coding violations reported from the files that developers made. We surveyed team size with five discrete intervals: 1, 2, 3–4, 5–8, and 9–21 by using the 20-percentile discretization as explained in Sect. 3.5. The largest team size was 21 in our survey. In the chi-squared test, observed frequencies of samples in the contingency table were $((8790, 4097, 8663, 5997, 4667), (12521, 2809, 7084, 7532, 2292))$. x^2-value was 10422 and p-value was less than 2.2e-16. Conclusively, we could statistically confirm that team size affects the developer-coding violations.

Figure 7 illustrates the results of the correlation between team size and coding violations. We found a trend that coding violations increased with team size. For example, a team (i.e., more than two developers) often committed more violations than a single developer did. Possibly, multiple developers experience greater communication challenges, making it more difficult for them to produce a significant amount of code that complies with rigorous coding guidelines as the number of team members increases.

Interestingly, teams with five to eight members tended to commit relatively fewer coding violations than did teams of nearby sizes. The extent of coding violations in teams having five to eight members was closer to the extent of the coding violations of a single developer. However, once a team size became too large (e.g., having greater than nine members), coding violations increased again. Possible reasons might be that a particular protocol of coding conventions did not work well or that communication between team members was inefficient.

4.3.3 Project Size vs. Coding Violations

Adherence to coding conventions may become more challenging as the project size increases. In larger projects, communication with many other developers and producing code free from coding violations on standard conventions are difficult.

We conducted a Pearson's chi-squared test for the two variables: project size (i.e., total lines of source code files) and the extent of coding violations reported from the

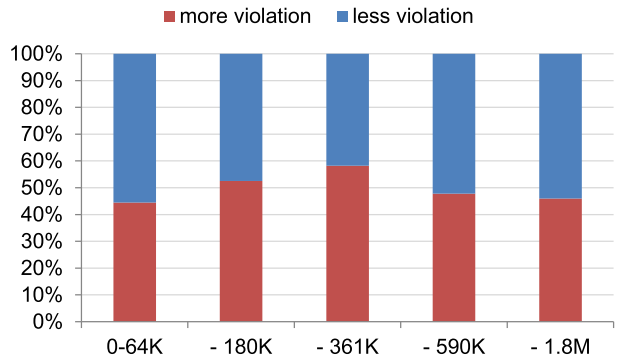

Fig. 8 Correlation between project size and coding violations.

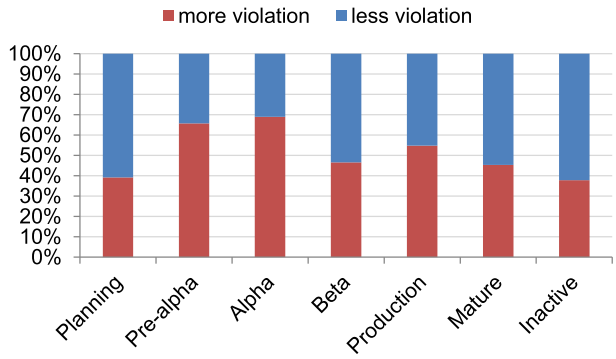

Fig. 9 Correlation between project maturity and coding violations.

files that developers made. We surveyed project size with five discrete intervals: 0–64K, –180K, –361K, –590K, and –1.8M by using the 20-percentile discretization as explained in Sect. 3.5. The largest project size was 1.8M in our survey. In the chi-squared test, observed frequencies of samples in the contingency table were ((5864, 7066, 8000, 5622, 5662), (7325, 6377, 5738, 6150, 6648)). x^2-value was 791 and p-value was less than 2.2e-16. Conclusively, we could statistically confirm that project size affects the developer-coding violations.

Figure 8 illustrates the correlation between project size and coding violations. The number of coding violations was lower in relatively small-sized projects, and the trend was for coding violations to increase as the project size increased, up to a certain level (e.g., 361K), thus indicating that larger-sized projects can involve more coding violations than relatively smaller projects do. However, this trend was not maintained; coding violations decreased again as the size of projects grew even larger (e.g., more than 590K). One plausible explanation for this phenomenon is that on small-sized projects it may be relatively easy to maintain software quality, and very large-scale projects could be strictly managed in terms of compliance with internal QA guidelines (e.g., coding conventions).

4.3.4 Project Maturity vs. Coding Violations

Projects generally mature over time. Therefore, projects in the early development stage usually do not have a well-arranged protocol of coding conventions and hence there is more opportunity for developers to generate coding violations. Moreover, developers prefer to implement ideas quickly in the early stage of a project and then refine them later.

We conducted a Pearson's chi-squared test for the two variables: degree of project maturity and the extent of coding violations reported from the files that developers made. In sourceforge.net, project maturity is represented with seven discrete stages: planning, pre-alpha, alpha, beta, production/stable, mature, and inactive. We adopted this representation as it is for the chi-squared test. In the chi-squared test, observed frequencies of samples in the contingency table were ((664, 1659, 2933, 2328, 9892, 13683, 1055), (1031, 865, 1318, 2664, 8156, 16468, 1736)). x^2-value was 15657.7 and p-value was less than 2.2e-16. Conclusively, we could statistically confirm that project maturity affects the developer-coding violations.

As illustrated in Fig. 9, there was a trend that coding violations steadily increased to the Alpha stage. However, it was observed that the coding violations stabilized at a lower level from the Beta stage onward. Although the extent of coding violations increased slightly in the Production stage, there was a trend that the coding violations were decreasing as the project advanced to the Mature and Inactive stages.

5. Discussion

In this section, we discuss assumptions and debatable issues that we recognized while performing the study.

What is the justification for using the readability estimation formula for a code clock (Sect. 3.2.1) at the file level? Suppose that there are two files, A and B for example. File A consists of code blocks most of which have low readability. In contrast, file B consists of code blocks having relatively high readability. In this example, we would assume that file B could be more easily reviewed by readers than could file A. Thus, we assume that the readability of a file could be represented by the readabilities of the code blocks comprising the file. Of course, for example, lexical or structural perspectives can be regarded as additional factors that determine the readability of a file. However, these are somewhat ambiguous and not easily measurable; therefore, they were out of scope for our study.

The readability estimation step must be consistent and fair for all sample files compared. In particular, the procedure for dividing a file into code blocks must be defined statically. This is why we chose to use a fixed size of code block (i.e., 46 LOC, Sect. 3.2.2). If instead we had divided a file into code blocks using blank lines or curly braces as division points, for example, the sizes of code blocks might have been dynamic, dependent on the coding styles of individual developers. For example, some developers may habitually write source code without using blank lines to separate paragraphs. Under such a system, readability scores of the sample files could have been inconsistently estimated; our intent was to avoid that pitfall.

6. Research Limitations

Note that the findings in this study are subject to our experiment conditions. We cannot claim that all the findings are universally acceptable in other project cases. The limitations or threats to validity in this study are as follows:

We only analyzed open-source projects. It is possible that closed-source projects would present different analysis results. Therefore, it is our future work to apply the proposed analysis approach to other projects, particularly closed-source projects.

Coding style and code readability can have diverse definitions depending on the individual. In this study, coding style was defined with an observable combination of rule violations or compliance in terms of given coding conventions. Code readability was defined by the visual complexity of code. However, programmers, for example, may wish to include an aspect of understandability into the definition of code readability. It is possible that programmers will have a negative feeling when asked to read code that is difficult to understand and thus assign a lower readability score. Therefore, to address this threat, we may be required to perform additional case studies in the future with extended definitions.

7. Conclusion

It has generally been believed that adherence to coding conventions could ensure a good quality of code readability. However, there were no many studies exploring whether this proposition was true or how strongly it could be accepted with supporting data. Therefore, we conducted a study to explore the related three research questions. In the study, we determined that:

- The adherence or violation of coding conventions affected the quality of readability of post-delivered code (RQ1, Sect. 4.1).
- There exist particular coding violations that have a relatively stronger influence on the quality of code readability (RQ2, Sect. 4.2). For example, violations related to trailing comments, Javadoc, and indentation were particularly sensitive ones, so they need to be respected and controlled to enhance code readability, if necessary.
- Developer experience, team size, project size, and project maturity were factors that are able to influence the extent of coding violations (RQ3, Sect. 4.3). Findings in Sect. 4.3 will be helpful, for example, in recruiting a developer, organizing a team, or managing project risks.

We believe that our findings and research approach can assist programmers or QA managers to understand where they should focus in developing their own customized coding styles.

In terms of future applications, the classification model designed in Sect. 4.1 can be used to predict the returned benefit aspect of code readability enhancements after the correction of a coding violation. Assume that coding violations must be corrected. An analyst should be able to determine what violation should be tackled first within a limited time or the given human resources. That is, the analyst should understand the cost and benefit aspects of the options to make cost-effective decisions. The benefit aspect can be predicted using the classification model that outputs a probability of whether the quality of the violation-corrected code is closer to "more readable" or "less readable". Using the classification model, the simulation of cost-benefit analysis can be performed for all possible correction scenarios.

Acknowledgments

This research was supported by the Next-Generation Information Computing Development and Basic Science Research Program through the National Research Foundation of Korea (NRF) funded by the Ministry of Science, ICT & Future Planning (2012M3C4A7033345), and the Ministry of Education, Science and Technology (2012R1A1A2009021).

References

[1] M. Smit, B. Gergel, H.J. Hoover, and E. Stroulia, "Maintainability and source code conventions: An analysis of open source projects," Technical report TR11-06, Department of Computing Science, University of Alberta libraries, 2011.
[2] A. Mohan and N. Gold, "Programming style changes in evolving source code," Proc. 12th IEEE International Workshop on Program Comprehension, pp.236–240, June 2004.
[3] T.D. LaToza, G. Venolia, and R. DeLine, "Maintaining mental models: A study of developer work habits," 28th International Conference on Software Engineering, pp.492–501, 2006.
[4] L.E. Deimel, "The uses of program reading," ACM SIGCSE Bulletin, vol.17, no.2, pp.5–14, 1985.
[5] D.R. Raymond, "Reading source code," Proc. 1991 Conference of the Centre for Advanced Studies on Collaborative Research, pp.3–16, 1991.
[6] S. Rugaber, "The use of domain knowledge in program understanding," Annals of Software Engineering, vol.9, pp.143–192, 2000.
[7] S. Ambler, "Java coding standards," Softw. Dev., vol.5, no.8, pp.67–71, 1997.
[8] L.W. Cannon, R.A. Elliott, L.W. Kirchho, J.H. Miller, J.M. Milner, R.W. Mitze, E.P. Schan, N.O. Whittington, H. Spencer, D. Keppel, and M. Brader, Recommended C Style and Coding Standards, Specialized Systems Consultants, Seattle, Washington, 1990.
[9] H. Sutter and A. Alexandrescu, C++ Coding Standards: 101 Rules, Guidelines, and Best Practices, Addison-Wesley Professional, 2004.
[10] X. Li and C. Prasad, "Effectively teaching coding standards in programming," 6th Conference on Information Technology Education, pp.239–244, 2005.
[11] Coding conventions supported by Checkstyle, http://checkstyle.sourceforge.net/availablechecks.html
[12] R.P.L. Buse and W. Weimer, "Learning a metric for code readability," IEEE Trans. Softw. Eng., vol.36, no.4, pp.546–558, 2010.
[13] D. Posnett, A. Hindle, and P. Devanbu, "A simpler model of software readability," Proc. 8th Working Conference on Mining Software Repositories, MSR '11, pp.73–82, 2011.
[14] K.K. Aggarwal, Y. Singh, and J.K. Chhabra, "An integrated measure

of software maintainability," Reliability and Maintainability Symposium, pp.235–241, 2002.
[15] R. Flesch, "A new readability yardstick," Journal of Applied Psychology, vol.32, no.3, pp.221–233, 1948.
[16] J. Börstler, M. Caspersen, and M. Nordström, "Beauty and the beast: Toward a measurement framework for example program quality," Umeå University, 2008.
[17] S. Butler, M. Wermelinger, Y.J. Yu, and H. Sharp, "Relating identifier naming flaws and code quality: An empirical study," 16th Working Conference on Reverse Engineering, pp.31–35, 2009.
[18] S. Butler, M. Wermelinger, Y.J. Yu, and H. Sharp, "Exploring the influence of identifier names on code quality: An empirical study," 14th European Conference on Software Maintenance and Reengineering, pp.156–165, 2010.
[19] D. Reed, "Sometimes style really does matter," Journal of Computing Sciences in Colleges, vol.25, no.5, pp.180–187, 2010.
[20] M.O. Elish and J. Offutt, "The adherence of open source JAVA programmers to standard coding practices," 6th International Conference on Software Engineering and Applications, 2002.
[21] M. Halstead, Elements of software science, Elsevier, New York, 1977.
[22] P. Tomas, M.J. Escalona, and M. Mejias, "Open source tools for measuring the internal quality of Java software products: A survey," Computer Standards & Interfaces, vol.36, no.1, pp.244–255, 2013.
[23] X. Wu, V. Kumar, J.R. Quinlan, J. Ghosh, Q. Yang, H. Motoda, G.J. McLachlan, A. Ng, B. Liu, P.S. Yu, Z.-H. Zhou, M. Steinbach, D.J. Hand, and D. Steinberg, "Top 10 algorithms in data mining," Journal Knowledge and Information Systems, vol.14, no.1, pp.1–37, 2007.
[24] Weka (ver 3.6.1), http://www.cs.waikato.ac.nz/ml/weka/
[25] E. Alpaydin, Introduction to Machine Learning, 2nd ed., The MIT Press, 2010.
[26] M.A. Hall and G. Holmes, "Benchmarking attribute selection techniques for discrete class data mining," IEEE Trans. Knowl. Data Eng., vol.15, no.6, pp.1437–1447, 2003.

Taek Lee is currently a Ph.D. candidate in Computer Science and Engineering at Korea University in Seoul, Korea. He received his M.Sc. in Computer Science and Engineering at Korea University in 2006. His research interests include man-machine interaction, user behavior modeling in software systems, software defect prediction, information security, and information risk analysis.

Jung-Been Lee is a Ph.D Course in the Department of Computer Science and Engineering at Korea University in Seoul, Korea. His major areas of study are self-adaptive software, software architecture evaluation and potential defect analysis. He received the M.S. degrees in Computer Science and Engineering from Korea University in 2011.

Hoh Peter In received his Ph.D. degree in Computer Science from the University of Southern California (USC). He was an Assistant Professor at Texas A&M University. At present, he is a professor in Department of Computer Science and Engineering at Korea University in Seoul, Korea. He is an editor of the EMSE and TIIS journals. His primary research interests are software engineering, social media platform and services, and software security management. He earned the most influential paper award for 10 years in ICRE 2006. He has published over 100 research papers.