ISSN 1330-3651 (Print), ISSN 1848-6339 (Online)

https://doi.org/10.17559/TV-20181030091239
Original scientific paper

Metric and Tool Support for Instant Feedback of Source Code Readability

Sangchul CHOI, Suntae KIM, JeongAh KIM, Sooyong PARK

Abstract: In the software maintenance phase, comprehending the legacy source code is inevitable, which consumes most of the time of the phase. The better the code is readable, the easier it is for code readers to comprehend the system based on the source code. This paper proposes an enhanced source code readability metric to quantitatively measure the extent of code readability. In addition, we developed a tool support named *Instant R. Gauge* to update the code on the fly based on the readability feedback of the current code. The tool also provides the history of the readability change so that developers recognize the more readable code and gradually change their coding habit without any annoying advice. The suggested readability metric achieves 75.74% of explanatory power, and our experiment showed that readability of most of the methods authored in our tool is higher than that of the methods without our approach.

Keywords: multiple linear regression; readability; source code analysis

1 INTRODUCTION

Software maintenance accounts for a large portion of the entire cost of the software life cycle [1]. In the software maintenance phase, reading and analysing source code is one of the most time-consuming activities compared to implementation and testing [2, 7, 10]. However, the readability of the source code, defined as the extent of how easily the program code can be read by a source code reader, is not explicitly measured throughout the entire project life-cycle [2]. Otherwise, it is sometimes handled at the end of the project, however, the source code is not updated at all because the project has been already finished [8]. This is mainly attributed to the lack of a technique that helps one to measure the source code readability and give feedback on the fly in writing source code.

There has been much research on measuring source code readability (see [2, 3, 6, 11]). Previous research has been conducted to measure the readability of source code in the following steps: 1) sample code selection, 2) attribute extraction, 3) survey using questionnaires, and 4) statistical analysis. They tried to optimize the indicators to measure source code readability with the minimum number of indicators. However, previous studies primarily tried to discretely classify the code readability into two categories: readable or not-readable. Therefore, there is no way to quantitatively recognize the extent of source code readability. In addition, they are not integrated with the IDE, thus a developer should execute extra tools to obtain the code readability of current code.

In order to address the above issues, this paper proposes a software metric that instantly measures the readability of source code in writing source code, especially for the Java method as a base coding unit. We newly suggest indicators for measuring the readability metrics of the source code, and refine and verify them through questionnaires and multiple linear regression analysis. Based on these indicators, we establish an equation that can quantitatively measure the readability of a Java method on the fly. In addition, we have developed the tool support named *Instant R. Gauge* that instantly measures readability of the Java method. Also, it supports to visualize the current readability in the readability gauge and historical changes of the measurement through the line graph for a developer to recognize which part of updates in the method have an influence on the readability so that it contributes a developer to gradually change his coding habits.

For the evaluation of our approach, we sampled 60 Java methods from eight popular open source projects, and 45 subjects with 1 to 15 years of industrial work experience participated in the questionnaire. Thus, the suggested readability metric obtained 75.74% of explanatory power with seven indicators. In addition, we performed the experiment for 52 undergraduate students to recognize how much *Instant R. Gauge* helps a developer to improve the code readability. We observed that readability of most of the methods where our approach is used is higher than that of the methods without our approach.

The contributions of our research are summarized as follows:

- A suggestion of indicators that have an influence on source code readability: As the indicators are considered as key factors that affect the readability of the source code on the fly, a developer can keep them in his mind to improve the source code readability in making a program.
- A suggestion of an equation for quantifying the extent of source code readability: This equation allows you to determine the readability of the current code in real time, and can recognize how small changes in the code affect the readability of the code.
- Development of the tool support *Instant R. Gauge*: We developed the tool support that visualizes the readability of the current Java method in the readability gauge and historical changes of the readability through the line graph.

The remainder of this paper is structured as follows: Section 2 introduced related work on the software metric for measuring the readability of the source code and previous tool support. Section 3 describes our approach to establish a software metric for measuring the source code readability and Section 4 introduces software architecture of *Instant R. Gauge*. Section 5 presents a preliminary study to build the readability measurement model and an experiment to recognize how our tool support can help developers improve the code readability. Finally, we conclude this paper in Section 6.

2 RELATED WORK

There has been much research on software metric to measure code readability and its tool support. Buse and Weimer investigated source code readability through a questionnaire and analyzed the relationship between readability and source code elements such as LOC, the number of comments, identifiers and spaces [2]. The result of the analysis showed that the specific code element has a positive or negative effect to the code readability. However, it is hard to measure the degree of the code readability on the fly, because it only determines whether the code is readable or not. In addition, they only developed web-based questionnaire software for gathering readability data from human subjects without proposing tool support to determine the code readability for developers.

Halstead suggested a method for calculating the effort of the source code to review [6]. Although they explicitly suggested the metric to measure complexity not readability, this research affected diverse following researches to measure the code readability. He defined several indicators through the number of the operator types (n_1), the total number of operators (N_1), the number of operand types (n_2) and the total number of operands (N_2). The equations to measure each indicator complexity are listed as follows:

$$Program\ Volume(V) = (N_1 + N_2) \times \log_2 (n_1 + n_2)$$

$$Difficulty(D) = \frac{n_1}{2} \times \frac{N_1}{n_2}$$

$$Effort(E) = D \times V$$

Posnett et al. proposed a way to minimize the number of indicators for readability measure [3]. They performed the correlation analysis for a selection of measures that best describe readability in Buse's model and Halstead's model with the smallest number of indicators. The analysis found that the *Program Volume* of the Halstead's model can representatively characterize code readability. Then, they additionally proposed *Entropy* to measure the amount of information existing in the source code. It is computed by equation $Entropy\ H(X) = \sum_{i=1}^{n} p(x_i) \log_2 p(x_i)$, where X is a document, x_i is a term, and $p(x_i)$ is calculated by the equation $p(x_i) \frac{p(x_i)}{\sum_{i=1}^{n} \text{count}(x_i)}$. Then, they showed that the proposed indicators are enough to measure the extent of the source code readability compared to the Buse's model. Although they refined the key indicators for code readability, these software metrics can be used only to determine whether the code is readable or not without showing the degree of the code readability. Thus, it is hard to apply to show the instant feedback of the code readability on the fly.

Yahya et al. proposed software metric for measuring source code readability and developed tool support named *CRT (Code Readability Tool)* that measures the code readability [11]. In the study, they used several basic information such as the number of identifiers, the number of comments and indentations. They developed the tool support CRT written in C# for a developer to import the Java files and generate the report of code readability of the file. Although they have proposed the new tool, the tool is not integrated with the IDE and has the disadvantage that developers have to do additional work to calculate the readability.

3 BUILDING SOFTWARE METRIC FOR MEASURING THE INSTANT SOURCE CODE READABILITY

This section describes steps to establish software metric to measure source code readability for giving an instant feedback of the current code. Fig. 1 presents detailed steps. It starts with conducting the questionnaire survey about the readability of the Java methods. Then, we define the candidate indicators and extract them from the source code of the survey. Based on the result of the survey, we carry out the regression analysis to establish the source code readability measure model for instant feedback. We describe detailed steps from the following subsections.

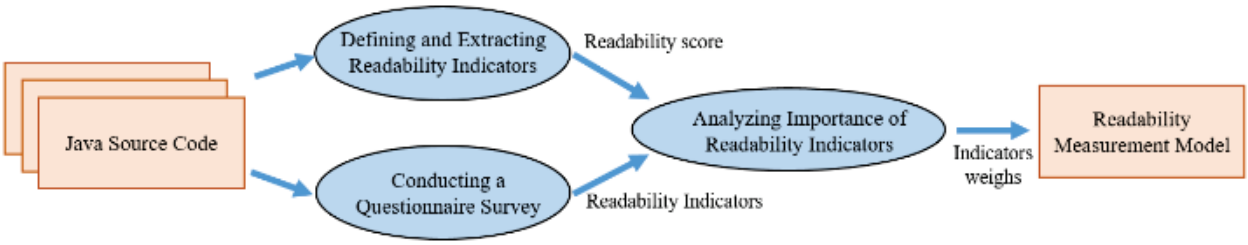

Figure 1 Steps for establishing source code readability measurement model

3.1 Conducting a Questionnaire Survey

As the first step for building the readability metric, we conducted a questionnaire survey to find out the readability score of the source code that the developers generally perceive. As the readability of the source code indicates the extent to which a reader can easily understand the source code [2], diverse factors can affect the readability. For example, the longer the line of the code is, the more difficult the reader generally reads the code. However, it is not always true in the case that similar code fragments repeatedly occur. Otherwise, even though the line of the code is very small, the code containing several combinations of bitwise and logical operators is very hard to analyze. Thus, it is important to investigate how much the code is readable for a human subject.

For the survey, we carefully sampled Java methods from the open source Java projects in consideration of diverse source code characteristics such as the line of the code, diverse control statement (e.g., *if*, *for*) and the

number of diverse operators (e.g., *bitwise* or *relational operators*). Fig. 2 shows one of the sample Java methods of the survey. The *validate*() method has no parameters with about 40 lines of volume, some of the *if* and nested *if* statements, and several *for* loops.

```
private void validate() {
    if (classOnly ^ (callbacks == null)) {
        if (classOnly) {
            throw new IllegalStateException("createClass does not accept callbacks");
        } else {
            throw new IllegalStateException("Callbacks are required");
        }
    }
    if (classOnly && (callbackTypes == null)) {
        throw new IllegalStateException("Callback types are required");
    }
    if (callbacks != null && callbackTypes != null) {
        if (callbacks.length != callbackTypes.length) {
            throw new IllegalStateException("Lengths of callback and callback types array must be the same");
        }
        Type[] check = CallbackInfo.determineTypes(callbacks);
        for (int i = 0; i < check.length; i++) {
            if (!check[i].equals(callbackTypes[i])) {
                throw new IllegalStateException("Callback " + check[i] + " is not assignable to " + callbackTypes[i]);
            }
        }
    } else if (callbacks != null) {
        callbackTypes = CallbackInfo.determineTypes(callbacks);
    }
    if (filter == null) {
        if (callbackTypes.length > 1) {
            throw new IllegalStateException("Multiple callback types possible but no filter specified");
        }
        filter = ALL_ZERO;
    }
    if (interfaces != null) {
        for (int i = 0; i < interfaces.length; i++) {
            if (interfaces[i] == null) {
                throw new IllegalStateException("Interfaces cannot be null");
            }
        }
    }
}
```

Figure 2 One of the sample Java methods for the survey

A human subject puts a readability score with the five-point scaled value as shown in Fig. 3. The questionnaire additionally requests the subject to describe the reason for the score in more detail in a natural language. The detailed list of the open source projects and the explanation on the human subjects are presented in the preliminary study of the experiment.

How do you think about source code readability above? *

- ◯ Very readable
- ◯ Relatively readable
- ◯ Normally readable
- ◯ Relatively hard
- ◯ Very hard

Why do you think of the readability you have answered above? Please write the reason in detail. *

Your answer

Figure 3 Questionnaire response form to put a readability score

3.2 Defining and Extracting Readability Indicators

In order to quantitatively and instantly measure the source code readability, we defined 16 candidate indicators of the source code as shown in Tab.1. Among the indicators, 14 indicators are extracted based on our experience that can have an influence on source code readability. For example, LOC is related to the program size, while the number of branch/loops and the number of logical/bitwise operators are regarding the complexity of a method.

Table 1 Candidate indicators of a code readability of the Java method

Indicator	Description
Line Of Code (LOC)	Lines of a method
NumOfMethodInvocation (#MI)	The number of invoked methods
NumOfBranches (#Bm)	The number of branches in the source code (e.g., *if*, *switch*)
NumOfLoops (#Lop)	The number of loops in the source code (e.g., *for*, *while*)
NumOfAssignments (#Asn)	The number of assignment operators (i.e., =)
NumOfComments (#Cmt)	The number of comments
NumOfBlankLines (#BL)	The number of blank lines
NumOfStringLiteral (#SL)	The number of string literal
NumOfArithmaticOperators (#AO)	The number of arithmetic operators (e.g., +, -, *, /)
NumOfLogicalOperators (#LO)	The number of logical operators (e.g., &&, ||)
NumOfBitwiseOperators (#BO)	The number of bitwise operators (e.g., &, |)
AverageOfVariableNameLength (#AVNL)	The average of length of variable identifiers
AverageLineLength (#LN)	The average of length of lines
maxNestedControl (#NC)	The maximum depth of nested control statements
ProgramVolume (#PV)	The amounts of information that the source code has [6]
Entropy (#Epy)	The complexity of source code [3]

In addition to the indicators, we adopted the *Program Volume* [6] and *Entropy* [3]. The program volume indicates the amount of information in the method, while the entropy is a metric to measure the complexity. The equations for computing the program volume and entropy are presented in Section 2. Although these indicators can be applied to other programming languages, this paper exemplified the indicators using the Java programming language. It should be noted that all indicators in Tab. 1 are only for measuring the readability of a Java method for giving a developer instant feedback of the current code.

These indicators are used to quantify readability of a Java method *m*, which is $Readability_m$, using the prediction function Φ presented in Eq. (1). In the equation, LOC_m or $\#MI_m$ denote indicators presented in the table.

$$Readability_m = \Phi\left(LOC_m, \#MI_m, ..., \#PV_m, \#Epy_m\right) \quad (1)$$

3.3 Analysing Importance of Readability Indicators

In order to establish the prediction function Φ with the 16 candidate indicators, we apply the regression analysis techniques. Use of the regression analysis assumes that the readability of methods obtained from human subjects is valid (or appropriately labelled). Then, we establish the regression model, which is the prediction function Φ, and measure the readability of a specific method *m* based on the model.

We select two regression analysis methods: multiple linear regression [5] and nonlinear regression techniques [9]. Multiple linear regression analysis is a method of estimating the relationship between one dependent variable and *n* independent variables using linear equations. The independent variable is understood as the cause of the influence on the results, and the dependent variable is the result influenced by independent variables. The nonlinear

regression analysis estimates the relationship between dependent and independent variables using mathematical transformation techniques such as squaring and logit transformation. The resulting model can be a shape of a polynomial equation.

With the two regression models, we apply a step-wise best parameter selection technique [4]. It allows the selection of optimal independent variables after eliminating the independent variables which have no significant influence on the formula through the combination of the independent variables used in the formula.

4 TOOL SUPPORT: *Instant R. Gauge*

This section describes steps of instant feedback based on our tool support *Instant R. Gauge* and describes its technical architecture. The steps are described in Fig. 4. When a developer presses the *Enter* key at the end of a statement in the Java method or requests to save the file containing the Java method, *Instant R. Gauge* instantly calculates the source code readability using the prediction function Φ. Then, it visualizes the result in the readability gauge like vehicle's fuel economy gauge. Based on the readability feedback, the developer can recognize the readability issue of the statement written right before and update the statement immediately without delaying the update later.

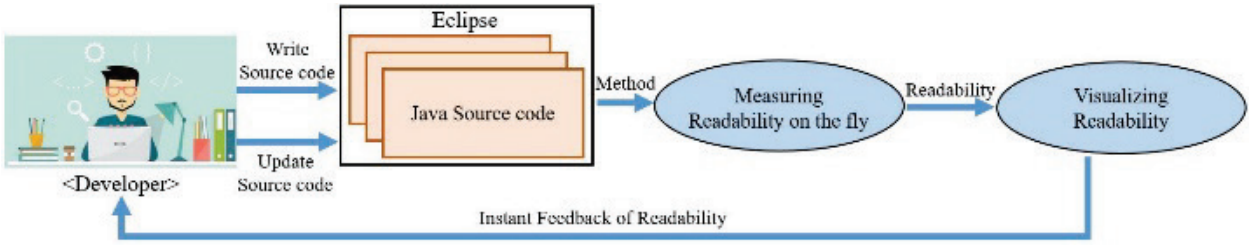

Figure 4 Steps for measuring and visualizing readability on the fly

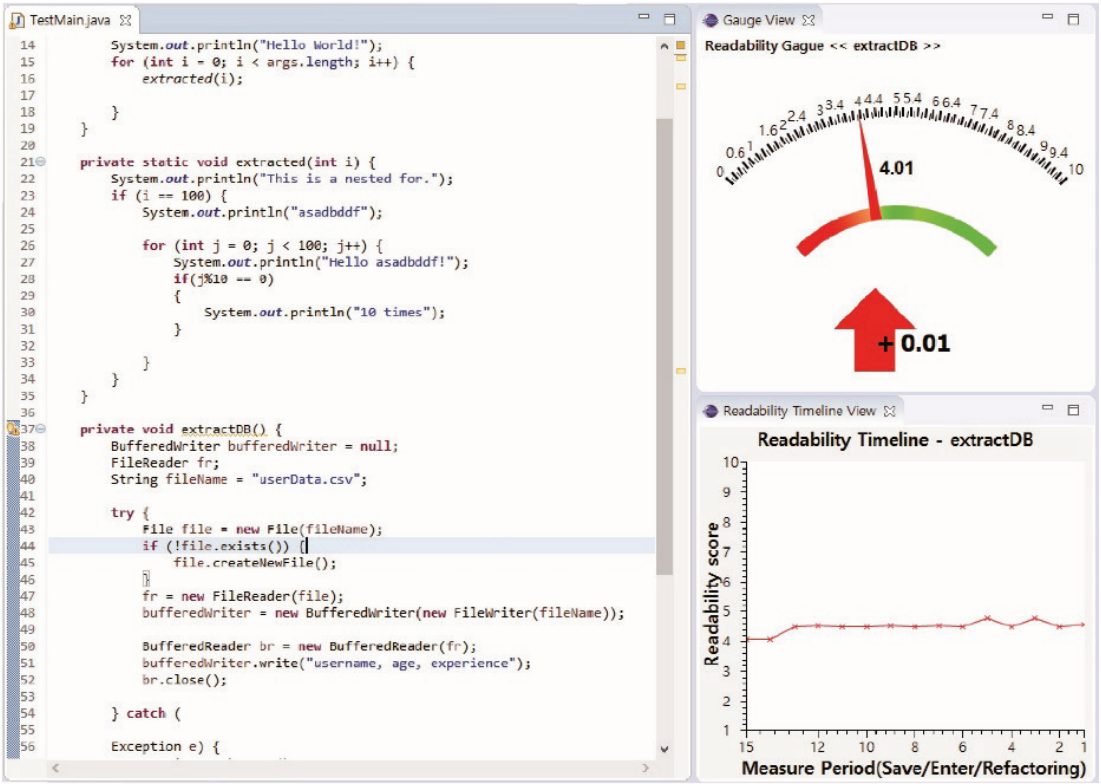

Figure 5 UIs with Eclipse editor

Fig. 5 presents a screen capture of *Instant R. Gauge*, composed of *Readability Gauge* view at the upper compartment and *Historical Readability Change* view at the lower. The readability gauge view shows the current readability and positive/negative change of the readability compared to the previous readability right after completing the current statement. The historical change view shows a historical readability update of the current Java method. It helps a developer to recognize which actions (e.g., refactoring or add more statements) have an influence on readability of the source code. Generally, the readability is gradually decreased when a developer adds some of the statements, however, it is instantly increased if a developer performs refactoring for the code. It is really important for a developer to change their code habit without any concrete and troublesome advice, just with instant feedback of the current readability. Notification of the readability issues (i.e., advice or violated rules) makes the developer annoyed so that the developer will turn off the feature.

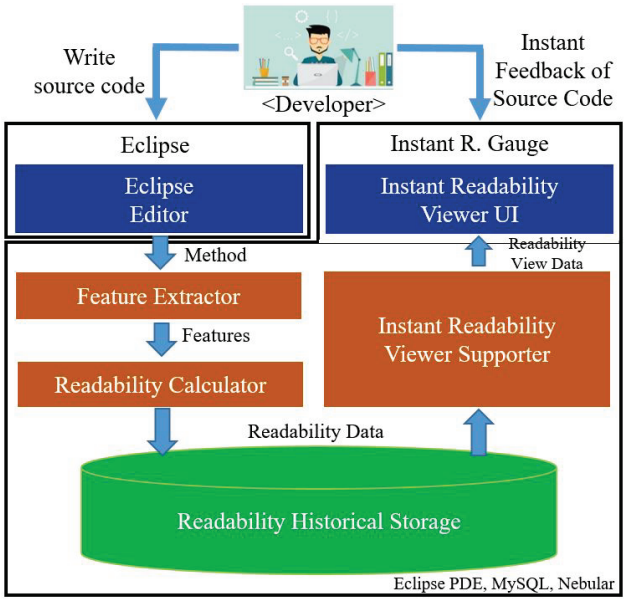

Figure 6 Architecture of instant feedback tool

We developed tool support *Instant R. Gauge* as a plug-in integrated with Eclipse IDE. Fig. 6 presents the software architecture of *Instant R. Gauge*. When a developer writes source code, *Feature Extractor* extracts the optimal indicators, and then the readability of the current method is measured by *Readability Calculator*. The result is stored in *Readability History Storage* and visualized in *Instant Readability Viewer* composing *Readability Gauge* and *Historical Readability Change* views as shown in Fig. 5.

5 EXPERIMENT

This section introduces an experiment and its result for our approach. It first describes the preliminary study to find the optimal software metric for the readability measure using the regression methods. Then, we carried out the experiment to figure out the feasibility of *Instant R. Gauge* built upon the readability metric on the fly. Finally, we discuss the threat to the validity of our experiment.

5.1 Preliminary Study

The aim of the preliminary study is to build the readability measurement function Φ based on the result of the questionnaire survey and the 16 candidate indicators suggested in the previous section. For the questionnaire, we identified 60 Java methods that contain broad data spectrum of the 16 indicators from the eight popular open source projects as summarized in Tab. 2. Then, we extracted 16 candidate indicators for each Java method.

Table 2 Open Source Projects for the Questionnaire

Project	Version	Explanation
Antlr4	4.1.2	A lexer and parser generator aimed at building and walking parse trees
Cglib	3.20	Byte Code Generation Library to generate and transform Java byte code
DBCP	4.2.2	Apache Commons Library for Database Connection Pool support
FileUpload	1.4	Apache Commons Library for uploading files in Web application
Groovy	2.5.0	A dynamic, scripting language for the JVM
HyperSQL	1.8.1.3	A relational database engine and a set of tools
Httpclient	3.0	Java Implementation of HTTP 1.1 protocol
JavaCC	6.1.0	Ascanner and parser generator written in Java

We collected 45 subjects ranging from 1 to 15 years of industrial work experience as summarized in Fig. 7(a). About 75% of the subjects have 1 to 6 years of work experience and the rest has over 7 years of work experience. Among the subjects, about 78% preferred the Java programming language as shown in Fig. 7(b). The questionnaire has been conducted using *Google Forms* with the answer form like Fig. 3 and Fig. 2 without time limit.

Table 3 Result of Multiple Linear and Nonlinear Regression Analysis

	Multiple L. Reg.	Step-Multiple L. Reg.
Number of Indicators	16	7
Multiple *R*-Squared	0.8421	0.7988
Adjusted *R*-Squared	0.7411	0.7574
Mean of Squared Err.	0.5933973	0.489533

	Nonlinear Reg.	Step-Nonlinear Reg.
Number of Indicators	16	14
Multiple *R*-Squared	0.9639	0.9496
Adjusted *R*-Squared	0.7268	0.7571
Mean of Squared Err.	2.590327	0.8867525

After collecting the survey, we obtained the average readability for 60 Java methods. Then, we divided the methods by a ratio of 7:3, where 7 is for establishing the readability measurement model and 3 is used for testing the model. We carried out multiple linear regression analysis and nonlinear regression analysis to build the readability prediction function Φ based on the 16 candidate indicators and the average readability from the questionnaire response. Tab. 3 summarizes the result of the experiment, composed of two compartments: multiple linear regression with 16 indicators and optimal indicators (see *Step-*) obtained from step-wise best parameter selection technique and nonlinear regression with the same combination. With the training data set, we build the regression model and obtained *Multiple* and *Adjusted R-Squared* values, and we evaluated the model with the test data so that we obtained the *Mean of Squared Error* (*MSE*) that indicates the average error between the real-test data and the prediction from the regression model.

For the multiple linear regression analysis with 16 indicators, we initially obtained 0.7411 of Adjusted *R*-Squared and 0.5933973 of *MSE*. As the Adjusted *R*-Squared is understood as the explanatory power [5], we can recognize that 74.11% of the data can be explained by the model. Detailed results of the multiple linear regression is presented in Tab.7. When we apply the *Step* function to figure out the optimal indicators, we obtained 7 indicators LOC, NumOfComments (*#Cmt*), NumOfBlankLines (*#BL*), NumOfBitOperators (*#BO*), maxNestedControl (*#NC*), ProgramVolume (*#PV*) and Entropy (*#Epy*), and it produced a better performance (see the Step-Multiple L.

Reg. column). Detailed result of the step-multiple regression analysis is summarized in Tab. 8.

The lower compartment of the table presents the result of the nonlinear regression analysis. The nonlinear regression analysis showed relatively worse performance than that of the multiple regression analysis. For example,

As a result of the preliminary study, we selected the model resulting from the Step-Multiple Linear Regression analysis. It is because the result of the Step-Multiple Linear Regression only uses 7 indicators with similar Adjusted *R*-Squared value and the smallest *MSE*. From the model (see Tab. 8), we extracted the *Estimate* value indicating the *gradient* of the regression model or the importance of each indicator. Then we finally built the readability prediction function Φ as shown in Eq. (2).

$$\begin{aligned} Readability = &-0.014 \times LOC + 0.029 \times \#Cmt + \\ &+0.032 \times \#BL - 0.873 \times \#BO - 0.205 \times \#NC - \\ &-0.001 \times \#PV - 0.739 \times Epy + 8.072 \end{aligned} \quad (2)$$

5.2 Experiment for *Instant R. Gauge*

We also carried out the experiment of *Instant R. Gauge* that can compute the code readability based on the metric obtained from the preliminary study. For the experiment, we applied our tool support into the term projects of the 2nd grade's *Source Code Analysis* class and 3rd grade's *Advanced Web Programming* class in the university. Tab. 4 summarized the experimental environment. The *Feedback* teams used our tool support to carry out the term projects with instant feedback on their code readability, while the *Non-Feedback* teams did not use our tool support but we monitored the change of the readability in the background. Their term projects are to improve the code from the same initial code to implement their improvement idea, though the 2nd grade's class project is a Java Swing based game and the 3rd grade's is the JSP/Servlet based product management system. Thus, the number of initial Java methods in each class are 86 and 68 respectively and the number of the methods of each project is multiplied by the number of the teams of each group and obtained the total methods as shown in the last column in Tab. 4.

The experiment has been carried out for about three weeks, all readability changes for each method have been stored in the central repository to trace them. After collecting the data, we classified the code changes into three categories: *Non-Modification*, *Modification* and *Deletion* (see Tab. 5). *Non-Modification* is not a modified code from the initial code, accounting for 67.12%. Also, 8.40% of the methods have been deleted throughout the project. We did not count newly created code because the readability of the created code tends to be consistently decreased and it is impossible to compare the gap of two groups.

MSE of the nonlinear regression is a lot higher than that of the linear analysis and 14 indicators from the Step-Nonlinear Regression is a higher number of indicators than that of the linear regression analysis. Due to the space limit, detailed results of the two experiments of the nonlinear regression analysis are not presented here.

Table 4 Experimental Environment

Grade	Feed. or No-Feed.	# of Students	# of Teams	# of initial Methods	Total Methods
2nd Grade	Feedback	17	8	86	688 (8×86)
	Non-Feedback	14	7	86	602 (7×86)
3rd Grade	Feedback	12	3	68	204 (3×68)
	Non-Feedback	9	2	68	136 (2×68)
Sum		52	20	154	1630

Table 5 Summary of Modifications of the projects

Grade	Feed. or No-Feed.	No-Mod.	Modification	Deletion
2nd Grade	Feedback	65.70% (524)	25.44% (175)	8.87% (61)
	Non-Feedback	67.77% (408)	26.91% (162)	5.32% (32)
3rd Grade	Feedback	61.27% (125)	19.12% (39)	19.61%(40)
	Non-Feedback	80.15% (109)	16.91% (23)	2.94% (4)
Sum		67.12% (1094)	24.48% (399)	8.40% (137)

Table 6 Result of the Experiment for *Instant R. Gauge*

	Increased				Decreased			
	+10%~	+5 ~ +10%	0 ~ 5%	Sum of increased	0 ~ −5%	−5 ~ −10%	−10% ~	Sum of increased
Feedback	20 (64.52%)	7 (87.50%)	39 (60.00%)	66 (63.43%)	55 (53.40%)	28 (52.83%)	65 (46.76%)	148 (50.17%)
Non-Feedback	11 (35.48%)	1 (12.50%)	26 (40.00%)	38 (36.54%)	48 (46.60%)	25 (47.17%)	74 (53.24%)	147 (49.83%)
Sum	31	8	65	104 (26.07%)	103	53	139	295 (73.93%)

We only examined the 399 modified Java methods in order to figure out how our tool helps developers to improve the code readability in writing their code. Thus, we analysed the gap between readability of the initial methods and that of the final methods and summarized the result in Tab.6. The readability of 26.7% of modifications was increased, while that of 73.93% of modifications was decreased. The gap of two proportions is understood as typical because the number of codes increases and the readability deteriorates as the project progresses. Among the methods with the increased readability, all methods of the *Feedback* group obtained better performance compared

to those of the *Non-Feedback*. Particularly, the *Feedback* group had a higher number of methods that increased the readability more than 5%. It implies that our tool *Instant R. Gauge* contributed the developers to increase the readability of their code. For the decreased readability, the numbers of the two groups were similar for the method ranging from 0% to -10%, however the *Feedback* group had the smaller number of methods decreased more than 10%, compared to that of the *Non-Feedback* group. It implies that the tool impacted less readability of the methods.

Table 7 Result of optimized multiple linear regression

Parameter	Estimate	Pr
(intercept)	7.3270443	$9.09e^{-06}$
LOC	−0.0122666	0.37608
#OfMethodInvocation	−0.0321940	0.04239
#OfBranch	0.0059510	0.91144
#OfLoops	0.1228757	0.32510
#OfAssignment	0.0127585	0.69880
#OfComments	0.0525633	0.16557
#OfBlankLines	0.0340361	0.33630
#OfStringLiteral	0.0787146	0.05614
#OfArithmaticOperators	−0.0128280	0.76433
#OfLogicalOperators	0.0609629	0.69963
#OfBitOperators	−1.6666902	0.07399
AverageOfVariableNameLength	0.0086161	0.96850
AverageLineLength	0.0007599	0.92428
maxNestedControl	−0.2019072	0.001348
ProgramVolume	−0.0012515	0.10700
Entropy	−0.5814477	0.00493
Multiple *R*-squared	0.8421	
Adjusted *R*-squared	0.7411	
p-value	$1.98e^{-06}$	

Table 8 Result of optimized multiple linear regression

Parameter	Estimate	Pr
(intercept)	8.0728926	$3.00e^{-13}$
LOC	−0.0146526	0.07948
#OfComments	0.0288382	0.33559
#OfBlankLines	0.0320093	0.20410
#OfBitOperators	−0.8730367	0.12903
maxNestedControl	−0.2050921	0.00190
ProgramVolume	−0.0007208	0.00223
Entropy	−0.7391025	$4.95e^{-05}$
Multiple *R*-squared	0.7988	
Adjusted *R*-squared	0.7574	
p-value	$3.923e^{-10}$	

5.3 Threat to Validity

Construct Validity The regression analysis is generally used to describe the trend and predict missing or the future value corresponding to the independent value. The concept of the readability is a vague concept to define. Thus, the concept may not be properly captured by the regression analysis. However, the regression analysis statistically describes the trend of the questionnaire results with the higher performance which is less *MSE* and adjusted *R*-Squared. There might be better analysis methods that have higher performance.

Content Validity The features suggested in this paper are initially proposed based on our experience. Then, they are added later based on the comments of the questionnaire survey from human subjects. Also, we adopted the features such as program volume and entropy from the previous research. For the regression analysis, we did not use only one regression method but analysed the data using two methods, linear and nonlinear regressions. Based on the results, we used the results of multiple linear regression analysis, which results in better performance.

Internal Validity For the experiment of Instant R. Gauge, half of the improvement idea is different from each other, and the remainder is common. Thus, all modifications of the code are not intended to implement the same idea. However, the number of methods for the statistics are 399, which is a statistically big number to compare the two groups.

External Validity Our approach may show different results depending on our subjects or open-source projects for writing the questionnaire. Such programs may have significantly different readabilities of the methods for the questionnaire. In addition, different human subjects may put different readability score to each method. Thus, the result of our study can be different if different subjects participated in the study.

6 CONCLUSION

In this paper, we proposed a software metric that quantitatively measures the source code readability in writing source code. For the metric, we suggested 16 candidate indicators and obtained seven optimal indicators with step-wise best parameter selection technique based on the multiple regression analysis. The software measure model has 75.74% of the explanatory power. In addition, we developed the tool support named Instant R. Gauge integrated with Eclipse IDE and showed our tools contribution to increasing the readability of the source in the experiment. As the future work, we have a plan to expose *Instant R. Gauge* as an open source project in order to help developers to improve their source code readability.

Acknowledgements

This research was supported by the MSIT (Ministry of Science and ICT), Korea, under the ITRC (Information Technology Research Center) support Program (IITP-2019-2017-0-01628) supervised by the IITP (Institute for Information & communications Technology Promotion).

7 REFERENCES

[1] Boehm, B. & Basili, V. R. (2001). Software defect reduction top 10 list. *IEEE Computer*, 34(1), 135-137. https://doi.org/10.1109/2.962984

[2] Buse, R. P. L. and Weimer, W. R. (2010). Learning a metric for code readability. *IEEE Transactions on Software Engineering*, 36(4), 546-558. https://doi.org/10.1109/TSE.2009.70

[3] Hindle D. Posnett, A. & Devanbu, P. (2011). A simpler model of software readability. *The 8th working conference on Mining software repositories (MSR)*, 11, 73-82. https://doi.org/10.1145/1985441.1985454

[4] Draper, N. & Smith, H. (1981). *Applied Regression Analysis, 2nd Edition*. Hoboken, John Wiley & Sons, Inc.

[5] Freedman, D. A. (2009). *Statistical models: Theory and practice*. Cambridge, In Cambridge University Press.

[6] Halstead, M. (1977). *Elements of software science*. New York, Elsevier New York.

[7] Poshyvanyk, D., Oliveto, R., Scalabrino, S., & Linares-Vsquez, M. (2016). Improving code readability models with

textual features. *In Proc. Int'l Conf. Program Comprehension (ICPC)*, 1-10. https://doi.org/10.1109/ICPC.2016.7503707

[8] Sedano, T. (2016).Code readability testing, an empirical study. *In IEEE 29th International Conference*, 111-117. https://doi.org/10.1109/CSEET.2016.36

[9] Sun, H. S. & Choi, S. H. (2009). A nonlinear regression analysis method for frame erasure concealment in voip network. *The institute of internet, broadcasting and communication*, 9(5), 129-132. http://www.earticle.net/article.aspx?sn=113182

[10] Vijay-Shanker, K., Wang, X., & Pollock, L. (2011). Automatic segmentation of method code into meaningful blocks to improve readability. *In 18th Working Conference on Reverse Engineering*, 35-44. https://doi.org/10.1109/WCRE.2011.15

[11] Alsmadi, I., Tashtoush, Y., Odat, Z., & Yatim, M. (2013). Impact of programming features on code readability. *International Journal of Software Engineering and its Applications*, 7(6), 441-458. https://doi.org/10.14257/ijseia.2013.7.6.38

Contact information:

Sangchul CHOI, Master
Dept. of Software Engineering, CAIIT,
Chonbuk National University,
Jeonju 54896, Jeonbuk, Republic of Korea
E-mail: 114477aa@gmail.com

Suntae KIM, PhD, Associate Professor
(Corresponding Author)
Dept. of Software Engineering, CAIIT,
Chonbuk National University,
Jeonju 54896, Jeonbuk, Republic of Korea
E-mail: stkim@jbnu.ac.kr

JeongAh KIM, PhD, Professor
(Corresponding Author)
Dept. of Computer Education,
Catholic Kwandong University,
Chuncheon-si, Gangwon-do, 24341 Republic of Korea
E-mail: clara@cku.ac.kr

Sooyong PARK, PhD, Professor
Dept. of Computer Science & Engineering,
Sogang University,
Seoul, 04107 Republic of Korea
E-mail: sypark@sogang.ac.kr