An Empirical Study on Relationship between Requirement Traceability Links and Bugs

Rizki Amelia[1], Hironori Washizaki[1 3 4*], Yoshiaki Fukazawa[1], Keishi Oshima[2], Ryota Mibe[2], Ryosuke Tsuchiya[2]

[1] Dept. of Computer Science and Engineering, Waseda University, Tokyo, Japan.
[2] Hitachi, Ltd., Research & Development Group, Center for Technology Innovation - Systems Engineering, Japan.
[3] National Institute of Informatics, Tokyo, Japan.
[4] System Information CO.,LTD., Tokyo, Japan.

* Corresponding author. *Tel.: +81-352863272; email: washizaki@waseda.jp
Manuscript submitted March 3, 2017; accepted May 17, 2017.
doi: 10.17706/jsw.12.5.315-325

Abstract: Early bug detection reduces the cost of software maintenance, but previous works have not utilized requirement traceability links (RTLs) as predictors for bugs. To discuss how to use RTLs to predict the number of bugs, we propose an RTL recovery approach classification based on the ease of the recovery process. We investigate the relationship using data from industrial software. Classes related to more RTLs tend to have more bugs. The classification provides better correlations, and including RTLs in the bug prediction model does not affect the performance. Some class files with no and low RTLs also have bugs; we hypothesize that this occurs because the actual RTL is missing or not established, which is supported by the observation that bugs in these classes are highly correlated with the maximum cyclomatic complexity.

Keywords: Requirement traceability links, bug prediction, software metrics, software maintenance.

1. Introduction

Traceability indicates that the relationship between two objects can be traced [1]. Empirical evidence has shown that requirement traceability links (RTLs), which are specified associations between requirements and other artifacts, support maintenance [2], [3]. Many studies have revealed that software maintenance is the most expensive phase in the software lifecycle. Currently maintenance accounts for 60–90% of the total software costs and at least 50% of the total man hours for a software system [3], [4]. We argue that predicting bugs is one way to improve the efficiency of maintenance activities. This leads to the question, "Can RTLs be used to predict bugs as early as possible in order to minimize the maintenance costs?" Previous works have not utilized RTLs as predictors for bugs.

Before using RTLs to predict bugs, whether RTLs and the number of bugs have a positive relationship must be investigated. We hypothesize that as the number of RTLs of a class increases, the likelihood that the class has entangled concerns increases. Thus, classes with many traceability links should have more bugs. This is supported by [6] in which tangled source code related to other concerns causes defects.

Traceability is a key issue to ensure consistency among software artifacts of subsequent phases in the development cycle [7]. Despite the importance and advantages of traceability links, explicit traceability is rarely established unless there is a regulatory reason [8]. Herein we propose an RTL recovery approach classification based on the ease of the recovery process. The classification is divided into four types. Type I is an explicit RTL, whereas Types II–IV are implicit RTLs. In our approach, RTLs are modified to recover missing links using

software from a company.

We aim to answer the following research questions:

RQ1 Do classes that are related to more requirements as indicated by more RTLs tend to have more bugs?

RQ2 Does the type of implicit RTL recovery classification affect the relationship between RTLs and bugs?

RQ3 Does including RTLs influence the bug prediction model performance?

This paper makes the following contributions:

- An RTL recovery approach classification based on the ease of the recovery process is proposed.
- The results of an extensive investigation on the relationship between RTLs and bugs are discussed.
- A new bug prediction model with RTLs as a prediction factor is presented.
- The proposed RTL recovery classification successfully identifies class files that are most difficult to maintain (i.e., class files without explicit RTLs and ones with the highest number of bugs).

The rest of paper is organized as follows: Section 2 presents our RTL recovery approach classification. Section 3 details the design. Section 4 provides the analysis results, while Section 5 shows the experiment. Section 6 addresses the research questions. Section 7 presents related works. Finally, section 8 provides a conclusion and future direction.

2. RTL Recovery Approach Classification

[9] defined three possible scenarios to recover traceability links. In this study, we adopted a similar approach to recover implicit traceability links. In addition to the three implicit traceability links, we also include one explicit traceability link. This setup realizes the following:

1) There are two types of traceability links: explicit and implicit.

2) Implicit traceability links are classified by the ease of the recovery process using the recovery scenarios in [9].

Therefore, our proposed RTL recovery approach (depicted in Fig. 1) is classified into the following four types:

- Type I contains explicit traceability links established during the software development process using knowledge of the developers. We assume that an ideal explicit traceability link is delivered after all links between related sources and target artifacts are completely established. However, the link's consistency must be verified if one or both of the linked artifacts are altered.
- Type II is the first implicit scenario in [5], which is manual tracing. All tracing activities and decisions are rendered by a human analyst. Assuming that both the source and target artifacts have representative titles for their contents, this process is considered easy because associating artifact titles recovers the links. It is less time consuming, and human knowledge can associate polysemy terms well when associating artifacts titles.
- Type III, which is the second implicit scenario, is automated tracing. In automated tracing, an analyst inputs the appropriate tracing tools and all necessary files. Then traceability links are automatically determined by examining content similarities between the source and target artifacts. This process is somewhat difficult and time consuming. Automated tracing provides candidates with the limitation that the retrieved links may be insufficient to directly use as explicit traceability links.
- Type IV, which is the third implicit scenario, is semi-automated tracing. These RTLs are difficult to recover. First, tools are used for automatic tracing. Then the candidate RTLs are studied by an analyst to determine the correctness and to thoroughly explore both the source and target artifacts to elucidate subtle traceability links not offered by the tools.

3. Study Design

3.1. Software under Study

We collected data from an enterprise software project developed by a Japanese company. The project consisted of 830 KLOC from 793 Java class code files with 962 requirements. We chose a project written in Java®

due to the domain expert's familiarity with Java®.

A traceability link is a specified association between a pair of artifacts where one represents the source artifact and other is comprised of the target artifacts. Links can be traversed in both directions [10]. Hence, an RTL is a specified association between the requirements and class files. In this project, class files have unique IDs, which represent an implemented requirement. Thus, the class file name and requirement name are matched using the same ID.

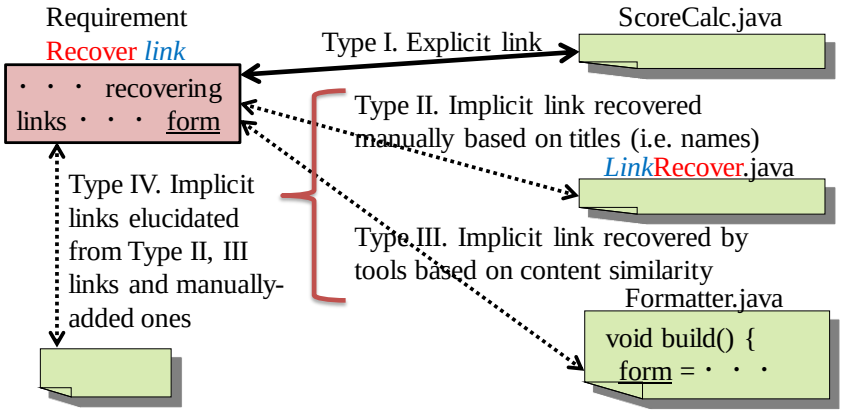

Fig. 1. RTL recovery approach classification.

3.2. RTL Recovery Approach Classification

Type I RTLs occur based on ID matching where the requirement ID and the class file ID are related via a one-to-one relationship. Type II RTLs are impossible to recover for the software in this study as the class files contain IDs only.

Type III RTLs have either a requirement ID or title in the class file contents. Because TraceLab [11], which is a common traceability link recovery tool, is limited to documents with English contents, we created our own simple tool for similarity analysis between the requirement ID and title with the class file's contents to find Type III RTLs. If class file contents contain either an ID or title, then whether the artifacts are related can be determined.

For Type IV RTLs, we treated the results from [12] since it targeted the same software. The results were obtained by applying various traceability recovery techniques [12], [23]-[25] with manual analysis and determination. We did not validate candidate links from Type III RTLs due to time and cost restrictions.

We grouped the class files based on the existence of the type RTLs as shown in Table I for further analysis. For example, a class with Type III and Type IV RTLs without Type I ones is grouped in g4. Due to the limitation of Type IV RTL recovery, some classes do not have any RTLs (grouped in g1).

Table 1. Class Groups Based on the Existence of RTL Type

Group	Type			Class	Group	Type			Class
	I	III	IV			I	III	IV	
g1	0	0	0	24	g4	0	1	1	55
g2	0	0	1	2	g5	1	1	0	13
g3	0	1	0	21	g6	1	1	1	678

3.3. Code Metrics for Predictors

To build a bug prediction model, we also analyzed other code metrics as candidates of predictors. Based on existing work [13], we analyzed similar metrics: CK metrics [14], OO metrics, complexity metrics, and volume metrics; these metrics were selected by following the work in [13]. The values of these metrics were measured

from the project using Understand [15]. Complexity was based on McCabe's cyclomatic complexity. Table II lists the code metrics included in our analysis.

3.4. Correlation Analysis

Correlation analysis aims to determine the correlations between RTLs and bugs as well as to determine correlations between code metrics and bugs. We employed correlation coefficient analysis using Pearson's correlation coefficient (r). Although Spearman's rank correlation coefficient is robust towards a nonlinear association, we selected r because this research focuses on linear correlations between two objects to build a prediction model using multiple linear regressions.

To investigate the correlation between RTL and bugs, the class files were sorted into three groups based on the amount of RTLs: zero, low, and high. The classes were divided based on the RTL median. Then the distribution of the number of bugs in each group was analyzed. The population significance was determined using a Wilcoxon rank sum test between the zero group and the target group.

To investigate the correlations between code metrics and bugs, we computed r for each metric and extracted the p-value to find the significance of the correlation. Only metrics with p-values < 0.05 were compared. Metrics strongly correlated with bugs were employed as predictors in the bug prediction model. To determine the relationship strength based on the obtained r, we used an existing categorization [7].

4. Analysis Results

4.1. Number of Bugs in Class Files Grouped by RTL Type

Figure 2 shows that g4 followed by g6 are the class files with the highest number of bugs (by mean and median). We hypothesize that class files in this group will be difficult to maintain. Without considering the existence of Type III and Type IV RTLs, g4 will be very costly with respect to bug fixing activities relative to other groups without RTLs because g4 has many bugs but lacks Type I RTLs, creating difficulties when tracing code specifications. To reduce the maintenance costs, software engineers should establish explicit RTLs. Similarly, Type III and Type IV RTLs should help reduce the maintenance cost.

4.2. Correlation between RTL and Bugs

The boxplots in Fig. 3, Fig. 4, Fig. 5 and Table III show the difference in the number of bugs by group. Groups with more RTLs tend to have more bugs. The Type III RTL group shows the strongest difference. In contrast, the Type I class file groups do not differ significantly. There are only two Type I groups because the company tried to match the requirement and class files in a one-to-one relationship using artifacts' IDs.

We conducted further analysis to determine which metrics contribute most to the number of bugs. Nine of the 28 metrics in Table II show uniform low values for the class files in the zero group without bugs (Table IV). The Pearson's r between these metrics and bugs for classes in the zero group with bugs indicates that only MaxCyclomatic has a strong correlation to the number of bugs. Thus, MaxCyclomatic is used as a metric to predict bugs in class files with no and low RTLs.

Table 2. Code Metrics Used

Catg.	Name	Description
CK	WMC	Count of Methods
	LCOM	Percent Lack of Cohesion
	DIT	Max Inheritance Tree
	CBO	Count of Coupled Classes
	NOC	Count of Derived Classes
	RFC	Count of All Methods
OO	NIM	Number of instance methods
	NIV	Number of instance variables
	IFANIN	Count of Base Classes

	Units	Number of non-nested modules, block data units, and subprograms
Comx	MaxCyclomatic	Maximum cyclomatic complexity of all nested functions or methods.
	AvgCyclomatic	Average cyclomatic complexity for all nested functions or methods
	Modified	Modified cyclomatic complexity
	Strict	Strict cyclomatic complexity
	Essential	Essential complexity
Vol	AvgLines	Average number of lines for all nested functions or methods
	AvgCodes	Average number of lines containing source code for all nested functions or methods
	AvgComment	Average number of lines containing comments for all nested functions or methods
	AvgBlank	Average number of blanks for all nested functions or methods
	Lines	Total lines in a file
	Comments	Total lines with a comment
	Blanks	Total lines without a comment or code
	Code	Total lines with code
	ExeLines	Number of lines containing an executable code
	DecLines	Total lines with declarative code
	ExeStmt	Number of executable statements
	DecStmt	Number of declarative statements
	RatioComment	Ratio of comment lines to code lines

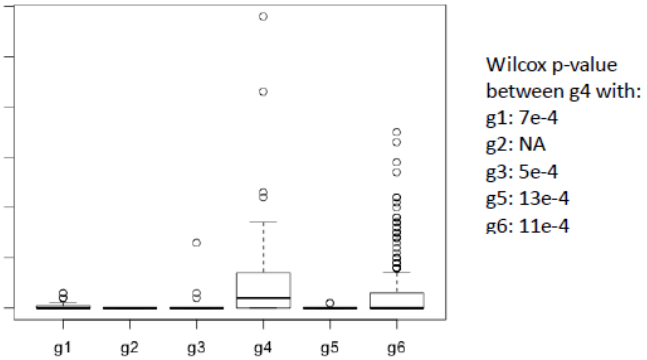

Fig. 2. Bugs distribution in the class files grouped by RTL type.

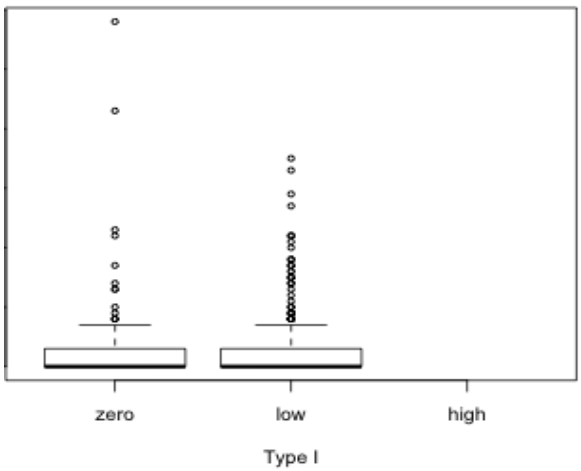

Fig. 3. Number of bugs in the class files with type I RTLs (zero, low, high in terms of the number of RTLs).

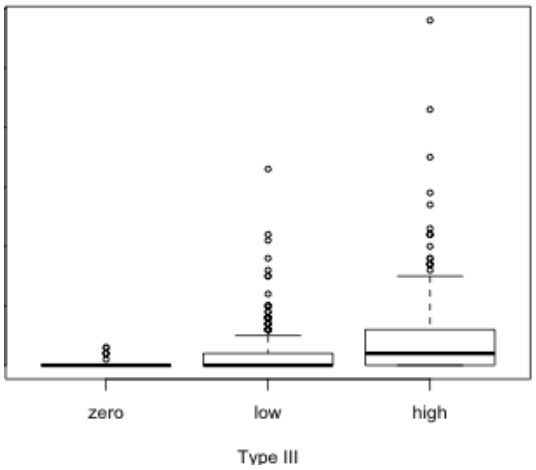

Fig. 4. Number of bugs in the class files with type III RTLs (zero, low, high in terms of the number of RTLs).

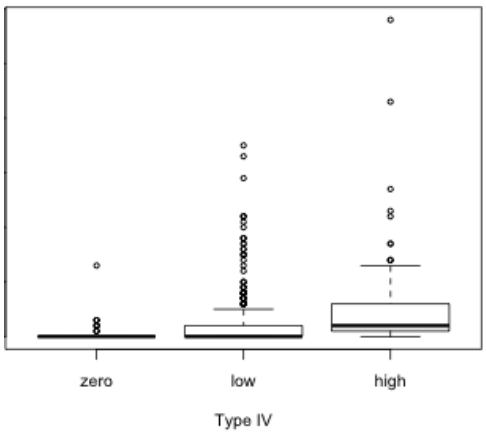

Fig. 5. Number of bugs in the class files with type IV RTLs (zero, low, high in terms of the number of RTLs).

Table 3. Distribution of the Number of Bugs by Group

Type	Group	Total Class	Mean	s.d.	Wilcox p-value	Pearson's r
I	zero	102	3.324	8.138		-0.083
	low	691	2.111	4.173	0.795	
	high	0	NA	NA	NA	
III	zero	26	0.5	0.99		0.409
	low	560	1.411	3.011	0.226	
	high	207	4.807	7.605	6.01E-18	
IV	zero	58	0.569	1.855		0.384
	low	629	1.943	4.092	0.001	
	high	106	5.123	8.179	1.5E-15	

Table 4. Correlation between Metrics and Bugs by Group

Catg.	Metrics	No Bugs: uniformity		With Bugs: Pearson's r	
		zero	low	zero	Low
CK	DIT	Yes	Yes	NA	0.079
	NOC	Yes	Yes	0.097	NA
OO	IFANIN	Yes	Yes	0.097	0.066
Comx	Modifier	Yes	No	0.54	0.48
	Strict	Yes	No	0.44	0.5
	AvgCyclo	Yes	No	0.54	0.48
	MaxCyclo	Yes	No	0.97	0.73
Vol	AvgLines	Yes	No	0.42	0.52
	AvgComment	Yes	No	0.27	0.54

4.3. Correlation between Code Metrics and Bugs

Of the 28 code metrics in Table II, 12 have correlations with significant values (i.e., < 0.05): MaxCyclomatic (0.714), ExeStmt (0.712), ExeLines (0.703), LOC (0.533), Strict (0.497), AvgComment (0.49), AvgCode (0.475), AvgLines (0.473), Modified (0.46), CBO (0.446), and Essential (0.394).

4.4. RTL Recovery Approach Classification Application

Type IV and Type III show weak and moderate correlations between RTLs and bugs, respectively. There is almost no correlation for Type I. Among the metrics analyzed, RTL is the second weakest, indicating that code metrics play a larger role in predicting bugs in class files. Consequently, only Type III RTLs and code metrics with moderate and strong correlations were used as predictors in our experiment.

5. Bug Prediction Based on Relationship Analysis

5.1. Experimental Setup

We used a standard evaluation technique called data splitting [16] to evaluate the predictive performance. We randomly chose two-thirds of all class files as training data to build the prediction models. The remaining one-third was used as test data. We performed 50 random splits to ensure the stability and repeatability of our results.

To build a multiple regression model, we analyzed the multi-collinearity among the independent variables. Because the common indicator of multi-collinearity is the variance inflation factor (VIF), we removed metrics with VIF ≥ 4 iteratively. Hence, none of the metrics displayed statistical evidence of multi-collinearity. The metrics with VIF < 4 after eight iterations are LOC, AvgComment, MaxCyclomatic, CBO, and the number of Type III RTLs since they showed the highest correlations with bugs among all types of RTLs.

Using these four metrics and RTLs, we built our bug prediction models. Two types of models were constructed: (M1) with RTLs and (M2) without RTLs. The models' performances were assessed via an explanatory power evaluation and a predictive power evaluation. To measure the quality of the model built from the training data, we computed R-square ranging from 0 to 1, where a higher value indicates a higher explanative power. The evaluation of the predictive power of the model was performed with respect to accuracy and sensitivity. For the accuracy, we computed the root mean squared error (RMSE) to determine the difference between the predicted number of bugs and the actual number of bugs. We chose RMSE instead of MSE because RMSE has the same unit as the dependent variable, making the results easier to interpret. A smaller RMSE value indicates fewer errors and a smaller difference between the predicted and actual bugs. For the sensitivity, we computed the Pearson's r to assess the correlation between the predicted bug and the actual bugs; the closer the absolute value is to 1, the stronger the correlation.

5.2. Experimental Results

Table 5. Results of Model Performance in 50 Splits

		Min	Max	Mean	s.d.
M1. With RTL	R-squared	0.573	0.723	0.648	0.038
	RMSE	2.290	3.870	3.197	0.440
	Pearson's r	0.650	0.868	0.775	0.046
M2. Without RTL	R-squared	0.562	0.719	0.644	0.037
	RMSE	2.300	3.880	3.159	0.437
	Pearson's r	0.657	0.873	0.779	0.044

Table V summarizes the explanatory power (R-squared) and predictive power (RMSE and Pearson's r) from the 50 random splits. Neither bug prediction model (with or without RTLs) performs strongly. The R-squared shows that the model with RTLs performs slightly better, but the predictive power performance of the bug prediction model without RTLs is slightly better according to the mean of RMSE and Pearson's r. These results

imply that the model with RTLs is not more accurate than the model without RTLs. Additionally, the low value of the standard deviation of the performance measures indicates both models provide consistent results.

6. Discussion

6.1. Research Questions

RQ1 Do classes related to more requirements as indicated by more RTLs tend to have more bugs?

Classes related to more RTLs tend to have more bugs as moderately supported by the correlation analysis result of Pearson's r of 0.409 (significant below the 0.05 level). We assume that class files in the zero or low groups have numerous missing RTLs. It is likely that the correlations will improve as the RTLs in these classes are recovered.

RQ2 Does the type of implicit RTL recovery classification affect the relationship between RTLs and bugs?

The recovery classification gives insights into correlations between the recovered RTLs and bugs. For the current project, the best relationship is shown by Type III RTLs.

RQ3 Does including RTLs influence the bug prediction model performance?

The explanatory power of the model with RTLs is slightly better than the model without RTLs, but the difference is insignificant. However, the model without RTLs has a slightly better predictive power than the model with RTLs. These results suggest that including RTLs in the bug prediction model does not affect the performance, at least for the current project.

6.2. Usage of Findings

Establishing RTLs explicitly helps trace the code from the class files to the requirements, improving the efficiency of fixing bugs. Moreover, engineers should be able to allocate their resources more effectively as it should be intuitive that class files with more RTLs have more bugs than class files with fewer RTLs. The proposed RTL recovery classification approach groups the class files based on the existence of RTLs by type to confirm which groups are in endangered states and whether they are maintained easily. Our findings indicate that software engineers should be aware of the maximum cyclomatic complexity of class files in a development because this will lead to bug-prone class files.

6.3. Threats to Validity

External Validity: Because the analysis results and current prediction model cannot be generalized beyond the specific software used in the experiment, validation using other software projects is necessary.

Internal Validity: Determining a strong relationship between RTLs and bugs is challenging. We suspect that the established RTLs are incomplete or missing for the current project.

Statistical Validity: All the results from the analysis and experimental study, including the performance of the bug prediction model, are significant below the 0.05 level.

7. Related Works

A previous study demonstrated that crosscutting concerns do cause defects by examining three small-sized to medium-sized Java® open-source projects [6]. On the other hand, our work focuses on analyzing tangling concerns indirectly. [6] suggested a method to realize software reliability by modularizing crosscutting concerns, whereas our work suggests that software developers establish RTLs, which are used to predict bugs, to estimate the maintenance costs. If RTLs are not established during development, we suggest using our proposed approach to recover implicit RTLs.

Many works [13], [17]-[22] have examined bug prediction models using code metrics. One standard set of metrics is the Chidamber and Kemerer (CK) metrics suite, which is used in [17], [18], [21], [22]. The bug prediction models built in [13], [19], [20] used other code metrics as predictors, while [14] found that a predictor only performed well in the project it was originally designed. Marco D'Ambros et al. compared the

performance of models with CK alone, OO alone, CK + OO, and LOC alone as predictors, and found that the model with CK + OO metrics exhibit the best predictor performance [13].

8. Conclusion and Future Work

There is a moderate correlation between RTLs and bugs. Some class files with no and low RTLs also have bugs. We hypothesize that this is because the actual RTL is missing or not established, which is consistent with the observation that bugs in these classes are highly correlated with maximum cyclomatics. Our findings suggest that the RTL is missing for class files with a high maximum complexity since they must implement at least one requirement. Hence, implementing an explicit RTL recovery tool is recommended as it can reduce the corrective maintenance phase for class files with many bugs. On the other hand, including RTLs in a bug prediction model does not affect the model performance.

In the future, we plan to investigate which bugs in class files in g4 (Section 4) are actually due to missing links to strengthen our suggestions about the importance of explicit RTLs. We also plan to recover actual Type IV RTLs on the same software and repeat the analysis to evaluate the impact on the results. We will replicate the analysis for different datasets from the software as well as employ other models for bug prediction such as machine-learning ones.

References

[1] Kazuki, N., Hironori, W., Yoshiaki, F., Keishi, O., & Ryota, M. (2015). Recovering transitive traceability among software artifacts. *Proceedings of the 31st IEEE International Conference on Software Maintenance and Evolution (ICSME)* (pp. 576-580).

[2] Patrick, M., & Alexander, E. (2012). Assessing the effect of requirements traceability for software maintenance. *Proceedings of the 28th IEEE International Conference on Software Maintenance (ICSM)* (pp. 171-180).

[3] Giuliano, A., Gerardo, C., Gerardo, C., Andrea, D. L., & Ettore, M. (2002). Recovering traceability links between code and documentation. *IEEE Transactions on Software Engineering*, *28(10)*, 970-983.

[4] Bennet, P. L., & Burton, E. S. (1980). *Software Maintenance Management*, Reading, Ma.: Addison-Wesley.

[5] Jose, M. C., Eduardo, F., Alessandro, G., Juan, H., & Elena, J. (2012). On the relationship of concern metrics and requirements maintainability. *Information and Software Technology*, *54(2)*, 212-238.

[6] Marc, E., Thomas, Z., Kaitlin, D. S., & Vibhav, G. (2008). Do crosscutting concerns cause defects? *IEEE Transactions on Software Engineering*, *34(4),* 497-515.

[7] Christine, D., & John, R. (2011). *Statistics Without Maths for Psychology*, Pearson Prentice Hall.

[8] Andrea, D. L., Andrian, M., Rocco, O., & Denys, P. (2012). Information retrieval methods for automated traceability recovery. *Software and Systems Traceability*, Springer.

[9] Alex, D., & Jane, H. H. (2012). Studying the role of humans in the traceability loop. *Software and Systems Traceability*. Springer.

[10] Jane, C. H., Orlena, G., & Andrea, Z. (2012). Software and systems traceability.

[11] Coest.org. Retrieved August 27, 2016, from http://www.coest.org/index.php/resources/dat-sets

[12] Ryosuke, T., Hironori, W., Yoshiaki, F., Keishi, O., & Ryota, M. (2015). Interactive recovery of requirements traceability links using user feedback and configuration management logs. *Proceedings of the 27th International Conference on Advanced Information Systems Engineering*.

[13] Marco, D., Michele, L., & Romain, R. (2011). Evaluating defect prediction approaches: A benchmark and an extensive comparison. *Empirical Software Engineering*, *17(4)*, 531-577.

[14] Shyam, R. C., & Chris, F. K. (1994). A metrics suite for object oriented design. *IEEE Transactions on Software Engineering*, *20(6)*, 476-493.

[15] SciTools.com. (April 27th 2016), http://SciTools.com

[16] Wang, J. J., Juan, L., Qing, W., & Da, Y. (2013). Can requirements dependency network be used as early

indicator of software integration Bugs? *Proceedings of the 21st IEEE International Requirements Engineering Conference (RE)*.

[17] Khaled, E. E., Walcelio, M., & Javam, C. M. (2001). The prediction of faulty classes using object-oriented design metrics. *Journal of Systems and Software*, *56(1)*, 63-75.

[18] Tibor, G., Rudolf, F., & Istvan, S. (2005). Empirical validation of object-oriented metrics on open source software for fault prediction. *IEEE Transactions on Software Engineering*, *31(10)*, 897-910.

[19] Nachiappan, N., Thomas, B., & Andreas, Z. (2006). Mining metrics to predict component failures. *Proceedings of the 28th International Conference on Software Engineering (ICSE)*.

[20] Thomas, Z., Rahul, P., & Andreas, Z. (2007). Predicting defects for eclipse. *International Workshop on Predictor Models in Software Engineering*.

[21] Niclas, O., & Hans, A. (1996). Predicting fault-prone software modules in telephone switches. *IEEE Transactions on Software Engineering*, *22(12)*, 886-894.

[22] Victor, R. B., Lionel, C. B., & Walcelio, L. M. (1996). A validation of object-oriented design metrics as quality indicators. *IEEE Transactions on Software Engineering*, *22(10)*, 751-761.

[23] Kentaro, K., Ryosuke, T., Hironori, W., & Yoshiaki, F. (2012). Supporting commonality and variability analysis of requirements and structural models. *Proceedings of the 4th International Workshop on Model-driven Approaches in Software Product Line Engineering*.

[24] Ryosuke, T., Hironori, W., Yoshiaki, F., Tadahisa, K., Masumi, K., Kentaro, Y. (2013). Recovering traceability links between requirements and source code in the same series of software products. *Proceedings of 17th International Software Product Line Conference*.

[25] Ryosuke, T., Hironori, W., Yoshiaki, F., Tadahisa, K., Masumi, K., Kentaro, Y. (2015). Recovering traceability links between requirements and source code using the configuration management log. *IEICE Transactions on Information and Systems*.

Rizki Amelia is currently working as radio network performance engineer at MOTiV Research in Tokyo, Japan. She received her master's degree in computer science and engineering from Waseda University in 2015. Her bachelor degree in information system was obtained from University of Indonesia in 2013 with advantage of attending one year exchange program in ICT Area at Daejeon University, Korea. Her research interests are mainly focus on software quality in software development and engineering.

Hironori Washizaki is the director and a professor at Global Software Engineering Laboratory, Waseda University, Japan. He also works at the National Institute of Informatics as a visiting professor and at System Information CO.,LTD. as an outside director. He was a visiting professor at Ecole Polytechnique de Montreal in 2015. He obtained his doctor's degree in information and computer science from Waseda University in 2003. His research interests include systems and software requirements, design, modeling, reuse, quality assurance, processes, management, and education.

He has been involved in organizing a number of international conferences, including serving as the program co-chair of SPAQu 2007-2009, ICST 2017, CSEE&T 2017 and APSEC 2018, the chair of AsianPLoP 2011-2016, the co-organizer of MAPLE/SCALE 2013 and PPAP 2016, the local chair of SPLC 2013 and COMPSAC 2018, the workshop co-chair of ASE 2006, the Publicity Chair of APSEC 2007, ASE 2012, CSEE&T 2015 and BICT 2015, and the steering committee of DEPEND 2016-2017 and FASSI 2016-2017. He has served as the chair of IEEE Computer Society Japan Chapter, the Chair of SEMAT Japan Chapter, the Convenor of ISO/IEC JTC1/SC7/WG20, the Director of ACM-ICPC 2014 Asia Regional Tokyo Contest, and the Director of IPSJ SamurAI Coding 2014-17. He has served as a member of the Editorial Board for many journals, including Int. J. Soft. Eng. & Know. Eng.,

IEICE Trans. Info. & Sys., Heliyon, and the Journal of Information Processing. He is appointed as IEEE Computer Society Member-at-Large for the Professional and Educational Activities Board, and Editor in Chief of Int. J. of Agile and Extreme Software Development.

Yoshiaki Fukazawa received the B.E., M.E. and D.E. degrees in electrical engineering from Waseda University, Tokyo, Japan in 1976, 1978, and 1986, respectively. He is currently a professor of the Department of Information and Computer Science, Waseda University as well as the director, Institute of Open Source Software, Waseda University. His research interests include software engineering, especially reuse of object-oriented software and agent-based software.

Keishi Oshima is a senior researcher of systems innovation center at Hitachi. He received his masters degree in information science and technology from Waseda University in 2002. His research interests are centered on legacy system analysis and repository mining.

Ryota Mibe is a senior researcher of systems innovation center at Hitachi. He received his masters degree in information science and technology from the Tokyo Institute of Technology in 1992. His research interests are centered on legacy system analysis and repository mining.

Ryosuke Tsuchiya is a researcher of systems innovation center at Hitachi. He received his masters degree in computer science and engineering from Waseda University in 2015. His research interests are centered on legacy system analysis and traceability recovery.