An Empirical Study on Relationship Between Requirement Traceability Links and Bugs

Rizki Amelia[1], Hironori Washizaki[1*], 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.

* Tel.: +81-352863272; email: washizaki@waseda.jp

Manuscript submitted February XX, 2017; accepted XX XX, 2017.
doi: ???

Abstract: Early bug detection reduces the cost of software maintenance; but none of previous works have utilized requirement traceability links (RTLs) as a predictor 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. Based on that, we investigated the relationship using data from industrial software. We confirmed that classes related to more RTLs tend to have more bugs, and the classification gives better correlations although including RTL 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 maximum cyclomatic complexity.

Key words: 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 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?" None of previous works have utilized RTL as a predictor for bugs.

Before using RTLs to predict bugs, we must investigate whether RTLs and the number of bugs have a positive relationship. We hypothesize that as the number of RTLs of a class increase, the likelihood that the class has entangled concerns. 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 the 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. Based on the classification, 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 make a difference in the relationship between RTLs and bugs?

RQ3 Does including RTLs affect 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 the 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-IV.

- Type I contains explicit traceability links established during the software development process using knowledge of the developers. We assumed that an ideal explicit traceability links is delivered after all links between related sources and target artifacts are completely established. However, it is necessary to verify the links' consistency 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 the artifacts title.
- 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 explore thoroughly both the source and target artifacts to elucidate subtle traceability links not offered by the tools.

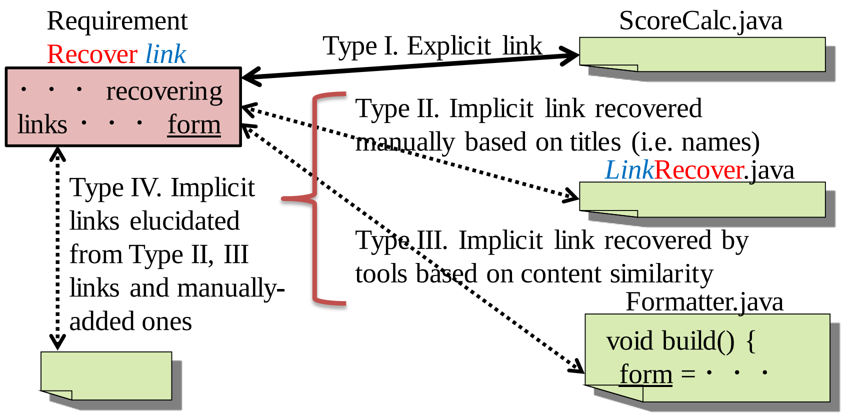

Requirement
Recover link
· · · recovering links · · · form
Type I. Explicit link
ScoreCalc.java
Type II. Implicit link recovered manually based on titles (i.e. names)
LinkRecover.java
Type III. Implicit link recovered by tools based on content similarity
Formatter.java
void build() {
form = · · ·
Type IV. Implicit links elucidated from Type II, III links and manually-added ones

Fig. 1. RTL Recovery Approach Classification

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, 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.

3.2. RTL Recovery Approach Classification Application

Type I RTLs occur based on ID matching where the requirement ID and the class file ID are matched via a 1-to-1 relationship. Type II RTL is impossible to recover for the software under study since 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 the ID or title, then whether the artifacts are related can be determined.

For Type IV RTLs, we treated the result from [12] since it targeted the same software. We did not validate candidate links from Type III RTLs due to time and cost limitation.

We grouped the class files based on the existence of RTLs by type as shown in Table I for further analysis. For example, a class having Type III RTLs and Type IV ones without Type I ones is grouped in g4. Due to the limitation of Type IV RTL recovery, there are classes (grouped in g1) without any RTL.

TABLE I. 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 the existing work [13], we analyzed similar metrics: CK metrics [14], OO metrics, complexity metrics, and volume metrics; these metrics are selected basically by following the work in [13]. Values of these metrics were measured from the project using Understand [15]. The complexity is based on McCabe's cyclomatic complexity. Table II lists the code metrics included in our analysis.

3.4. Correlation Analysis

The correlation analysis aimed to determine 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 the prediction model using multiple linear regression.

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 division of classes was based on the RTL median. Then the distribution of the number of bugs in each group was analyzed and 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 the 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 with a strong correlation 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 on Class Files Grouped by RTL Type

Fig. 2 shows that g4 followed by g6 are the class files with the highest number of bugs (by mean and median). Since g4 is the group of class files without Type I RTLs, we hypothesize that class files in this group will be difficult to maintain if their RTLs are not recovered.

Without considering the existence of Type III and Type IV, 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. This situation makes tracing the specification of the code difficult; software engineers should establish explicit RTLs, which reduce the corrective maintenance cost. Similarly, Type III and Type IV RTLs should help reduce the corrective maintenance cost.

TABLE II. CODE METRICS USED

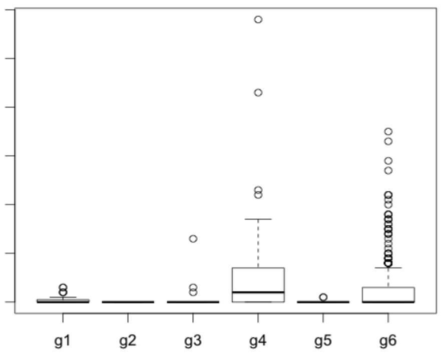
Catg.	Name	Description
CK	WMC	Count of Methods
	LCOM	Percent Lack of Cohesion
	DIT	Max Inheritance Tree
	CBO	Count of Coupled Classes
	NOC	Count of Derived Classes
	RFC	Count of All Methods
OO	NIM	Number of instance methods
	NIV	Number of instance variables
	IFANIN	Count of Base Classes
	Units	Number of non-nested modules, block data units, and subprograms
Comx	MaxCyclomati c	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 comment 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.

4.2. Correlation between RTL and Bugs

The boxplot in Fig. 3 and Table III show the difference of the number of bugs by group. Groups with more RTLs tend to have more bugs. The Type III RTL group shows the strongest difference. Meanwhile, the Type I class file groups do not differ significantly as there are only two Type I groups. This is because the company tried to match the requirement and class files in a 1-to-1 relationship by matching artifacts' IDs.

We conducted further analysis to determine which metrics contribute most to the number of bugs. Nine out of 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 group zero 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.



Wilcox p-value
between g4 with:
g1: 7e-4
g2: NA
g3: 5e-4
g5: 13e-4
g6: 11e-4

Fig. 2. Bugs Distribution in the Class Files Grouped by RTL Type

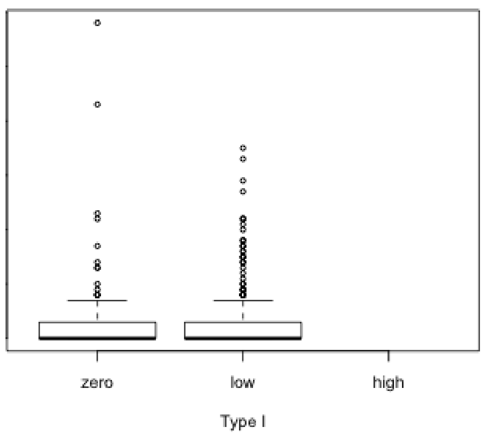


Fig. 3. Number of Bugs in the Class Files having Type I RTLs (Z: zero, L: low, H: high in terms of the amount of RTLs)

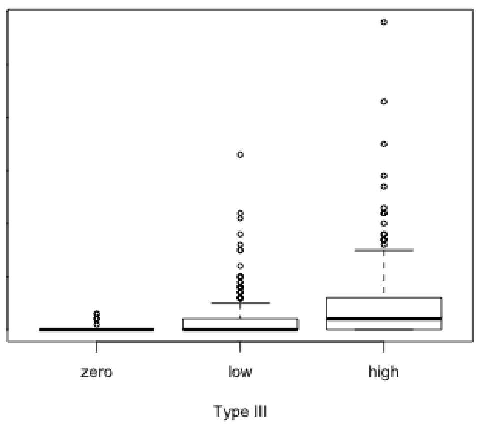


Fig. 4. Number of Bugs in the Class Files having Type III RTLs (Z: zero, L: low, H: high in terms of the amount of RTLs)

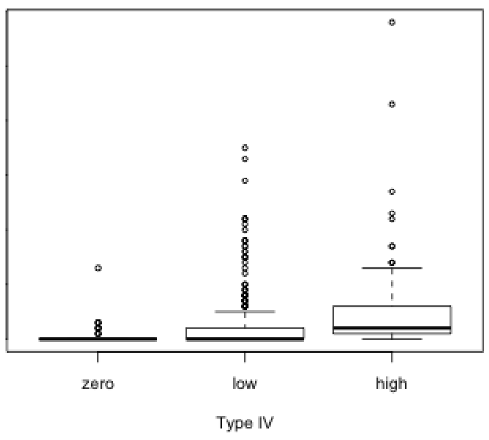


Fig. 5. Number of Bugs in the Class Files having Type IV RTLs (Z: zero, L: low, H: high in terms of the amount of RTLs)

TABLE III. DISTRIBUTION OF THE NUMBER OF BUGS IN 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	

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

TABLE IV. CORRELATION BETWEEN METRICS AND BUGS IN GROUP

4.3. Correlation between Code Metrics and Bugs

Of 28 code metrics in Table II, 12 have correlations with significant values < 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

The correlations between RTLs and bugs are weak for Type IV and moderate for Type III. There was almost no correlation for Type I. Among the metrics analyzed, RTL is the second weakest, indicating that code metrics play a larger role for predicting bugs in class files. Consequently, only Type III RTLs and code metrics with moderate and strong correlations are used as predictors in the following 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 model, while 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 the 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 used have 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 it showed the highest correlation with bugs among all types of RTL.

Using these four metrics and RTL, 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 predictive power of the model is performed with 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 to use 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 V summarizes the results of the explanatory power (R-squared) and predictive power (RMSE and Pearson's r) from the 50 random splits. The bug prediction model with or without RTLs does not perform strongly.

The R-squared shows that the model with RTL performs slightly better, whereas the predictive power performance of the bug prediction model without RTL is slightly better according to the mean of RMSE and Pearson's r. These results imply that the model with RTL is not more accurate than the model without RTL. Additionally, the low value of the standard deviation of the performance measures shows consistent results for both models.

TABLE V. 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

6. Discussion

6.1. Research Questions

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

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

RQ2 Does the type of implicit RTL recovery classification make a difference in the relationship between RTLs and bugs?

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

RQ3 Does including RTLs affect the bug prediction model performance?

In explanatory power, the model with RTL performed slightly better than the model without RTL. In predictive power, the model without RTL is slightly better than the model with RTL. However, the difference in the explanatory power performance is not significant. These results suggest that including RTL 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 better able to allocate their resources more effectively as it should be intuitive that class files with more RTLs will likely have more bugs than class files with fewer RTLs.

With the proposed RTL recovery classification approach, we grouped 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 that they are developing because this will lead to bug-prone class files.

6.3. Threats to Validity

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

Internal Validity: As we have speculated, the data of current project where we suspected that the established RTLs are incomplete or missing, it was challenging to determine a strong relationship between RTLs and bugs.

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

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

Many researches [13][17-22] 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 no predictor could perform well except for in the project it was originally designed. In [13], Marco D'Ambros et al. compared the performance of models with CK alone, OO alone, CK + OO, and LOC alone as predictors. They found that the model with CK + OO metrics exhibit the best predictor performance.

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 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 maximum cyclomatics. Our findings suggest that RTL is missing for these class files having a high maximum complexity since they must implement at least one requirement. Hence, implementing an explicit RTL recovery tool is recommended as it helps reduce the corrective maintenance phase for class files with many bugs. On the other hand, including RTL in a bug prediction model does not affect the model performance.

In the future, we plan to investigate which bugs on class files in g4 (Section 4) are actually caused by missing links to strengthen our suggestions about the importance of explicit RTL. Then, we plan to recover the real Type IV RTLs on the same software and repeat the analysis to see whether it makes difference result. We also plan to replicate the analysis for different datasets from software. Moreover we plan to employ other models for bug prediction such as machine-learning ones in addition to multiple linear regression.

References

[1] Kazuki Nishikawa, Hironori Washizaki, Yoshiaki Fukazawa, Keishi Ohshima, and Ryota Mibe, "Recovering Transitive Traceability among Software Artifacts," Proceedings of the 31st IEEE International Conference on Software Maintenance and Evolution (ICSME), pp.576-580, 2015.

[2] Patrick Mäder and Alexander Egyed, "Assessing the Effect of Requirements Traceability for Software Maintenance," Proceedings of the 28th IEEE International Conference on Software Maintenance (ICSM), pp.171-180, 2012.

[3] Giuliano Antoniol, Gerardo Canfora, Gerardo Casazza, Andrea De Lucia, and Ettore Merlo, "Recovering traceability links between code and documentation," IEEE Transactions on Software Engineering, Vol.28, No.10, pp.970-983, 2002.

[4] Bennet P. Lientz, and E. Burton Swanson, "Software maintenance management," Reading, Ma.: Addison-Wesley, 1980.

[5] Jose M. Conejero, Eduardo Figueiredo, Alessandro Garcia, Juan Hernandez, and Elena Jurado, "On the relationship of concern metrics and requirements maintainability", Information and Software Technology, Vol.54, No.2, pp.212-238, 2012.

[6] Marc Eaddy, Thomas Zimmermann, Kaitlin D. Sherwood, and Vibhav Garg, "Do Crosscutting Concerns Cause Defects?," IEEE Transactions on Software Engineering, Vol.34, No.4, pp. 497-515, 2008.

[7] Christine Dancey, and John Reidy, "Statistics Without Maths for Psychology," Pearson Prentice Hall, 2011.

[8] Andrea De Lucia, Andrian Marcus, Rocco Oliveto, and Denys Poshyvany, "Information Retrieval Methods for Automated Traceability Recovery," in Software and Systems Traceability, Springer, pp.71-98, 2012.

[9] Alex Dekhtyar, and Jane Huffman Hayes., "Studying the Role of Humans in the Traceability Loop", in Software and Systems Traceability, Springer, pp.241-261, 2012.

[10] Jane Cleland-Huang, Orlena Gotel, and Andrea Zisman, "Software and systems traceability," London: Springer, 2012.

[11] Coest.org (last visit: April 27th 2016), http://www.coest.org/index.php/resources/dat-sets

[12] Ryosuke Tsuchiya, Hironori Washizaki, Yoshiaki Fukazawa, Keishi Oshima, and Ryota Mibe, "Interactive Recovery of Requirements Traceability Links Using User Feedback and Configuration Management Logs, " Proceedings of the 27th International Conference on Advanced Information Systems Engineering (CAiSE), pp.247-262, 2015.

[13] Marco D'Ambros, Michele Lanza, and Romain Robbes, "Evaluating Defect Prediction Approaches: A Benchmark and An Extensive Comparison," Empirical Software Engineering, Vol.17, No.4, pp 531-577, 2011.

[14] Shyam R. Chidamber and Chris F. Kemerer, "A Metrics Suite for Object Oriented Design," IEEE Transactions on Software Engineering, Vol.20, No.6, pp.476-493, 1994.

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

[16] Junjie Wang, Juan Li, Qing Wang, and Da Yang, "Can Requirements Dependency Network be Used as Early Indicator of Software Integration Bugs?," Proceedings of the 21st IEEE International Requirements Engineering Conference (RE), pp.185-194, 2013.

[17] Khaled El Emam, Walcelio Melo, and Javam C. Machado, "The prediction of Faulty Classes Using Object-Oriented Design Metrics," Journal of Systems and Software, Vol.56, No.1, pp.63-75, 2001.

[18] Tibor Gyimothy, Rudolf Ferenc, and Istvan Siket, "Empirical validation of object-oriented metrics on open source software for fault prediction," IEEE Transactions on Software Engineering, Vol.31, No.10, pp.897-910, 2005.

[19] Nachiappan Nagappan, Thomas Ball, and Andreas Zeller, "Mining Metrics to Predict Component

Failures," Proceedings of the 28th International Conference on Software Engineering (ICSE), pp.452-461, 2006.

[20] Thomas Zimmerman, Rahul Premraj, and Andreas Zeller, "Predicting Defects for Eclipse," International Workshop on Predictor Models in Software Engineering (PROMISE), p.9, 2007.

[21] Niclas Ohlsson, and Hans Alberg, "Predicting fault-prone software modules in telephone switches," IEEE Transactions on Software Engineering, Vol.22, No.12, pp.886-894, 1996.

[22] Victor R. Basili, Lionel C. Briand, and Walcelio L. Melo, "A validation of object-oriented design metrics as quality indicators," IEEE Transactions on Software Engineering, Vol.22, No.10, pp.751-761, 1996.