Institute of Software Technology

University of Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Master Thesis Nr. MCS-0005

A study on the relationship between FindBugs warnings, metrics and expert judgments

Ling Xu

Course of Study:	Computer Science
Examiner:	Prof. Dr. Stefan Wagner
Supervisor:	Dipl. -Ing. Jan-Peter Ostberg

Commenced:	2015-11-04
Completed:	2016-05-04

CR-Classification:	D.2.7, D.2.8

Acknowledgment

Hereby, I would like to extend my sincere gratitude to everyone who has helped me in the past six months.

First and foremost, I would like to thank Prof. Dr. Stefan Wagner in Institute of Software Technology, provides me this meaningful research topic. I am very grateful to my supervisor in Universität Stuttgart, Jan-Peter Ostberg, for organizing bi-/weekly meetings with me. I got many instructions about applying the research methodologies and many relevant, valuable literatures from him.

Furthermore, I would like to thank Andreas Maier, my supervisor in NTTDATA for continuous guidance and giving useful suggestions in the writing part, and also all the colleagues involved in the interviews, who are willing to share their precious experience with me. Thank for the company for offering the access to some resources.

In the end, I would like to thank my family for great encouragement and continual support throughout the study life in Germany.

Abstract

The usage of static code analysis tools is one of the techniques to help to inspect the software quality. Many types of research have been made to evaluate such tools, but the investigate on what the analysis report further indicates and how these tools are applied in the industrial case is less concerned. This paper presents the study on three project cases in an IT Consultant company, where Findbugs, a bug pattern detection tool and SonarQube, a code quality monitoring tool are used in the development team. First, the correlation between 6 bug pattern categories and 5 source code metrics are investigated. The statistical analysis of the data extracted from tools has shown the most of the correlations are only specific to certain project. The result is partially different from expert judgments. Second, we decompose the maintainability into several characteristics, and a set of metrics measured by SonarQube are chosen to predict each of them on the basis of the practical experience. Third, the importance of metrics about test coverage and complexity is considered to be not constant among different types of projects.

Keywords: static source code analysis, software metrics, Findbugs, SonarQube, Maintainability

Abbreviations

Source Code Metric	
LOC	Lines Of Code
CLD	Comment Line Density
CPLX	Complexity
CVR	Coverage
EC	Efferent Coupling

Findbug bug category	
MT Correctness	Multithreaded Correctness
MC Vulnerability	Malicious code vulnerability
PERF	Performance

Others	
Spearman's rho	Spearman's rank correlation coefficient

Contents

Acknowledgment i
Abstract ii
Abbreviations iii

1. Introduction **1**
1.1. Motivation and Goal 1
1.2. Research questions 2
1.3. Related work 4
1.4. Outline 5

2. Background **7**
2.1. Code analysis tools used 7
2.1.1. Findbugs 7
2.1.2. SonarQube 7
2.2. Selected software quality - Maintainability 9
2.3. Selected software metrics 11

3. Study Design **15**
3.1. Projects and context 15
3.2. Research Question 1 16
3.3. Research question 2 19
3.4. Data collection procedure 19
3.4.1. Quantitative data collection 19
3.4.2. Qualitative data collection 21

4. Implementation **23**
4.1. Quantitative data collection 23
4.1.1. SonarQube Database structure 25
4.1.2. Local database structure 27
4.1.3. Building an automatic data collection system 31

4.1.4. Web UI of the system . . . 34
4.2. Qualitative Data Collection . . . 37
4.2.1. First-round Interview . . . 37
4.2.2. Second-round Interview . . . 40

5. Discussion - RQ1 **43**
5.1. Different versions approach . . . 43
5.1.1. Data analysis . . . 44
5.1.2. Restrictions . . . 48
5.2. Different projects approach . . . 49
5.2.1. Data analysis . . . 49
5.2.2. Restrictions . . . 54
5.3. Summary . . . 55

6. Discussion - RQ2 **57**
6.1. Coding of the interview notes . . . 57
6.2. RQ 2.1 . . . 61
6.2.1. Results . . . 61
6.2.2. Restrictions . . . 62
6.3. RQ 2.2 . . . 62
6.3.1. Results . . . 64
6.3.2. Restrictions . . . 67
6.4. Summary . . . 67

7. Conclusion and Outlook **69**

Appendices **73**

A. Table creation script in local database **75**

Bibliography **79**

1. Introduction

1.1. Motivation and Goal

The study of the software quality assessment has been conducted for many years. Software quality is an abstract and multidimensional concept[1]. It refers to the quality in different fields of software engineering, e.g. in software design level, product operational level, test processing level. Suppose that we discuss 'quality' with different roles. From the buyers'/consumers' perspective, 'quality' is an intangible term, they will think in an economical and time-saving way. The software is a good product if it could conform with their business expectations and there is no severe breakdown in usage. From the perspective of developing teams, they will think about this issue in a more technical way, e.g. whether the software meets with all of the specifications proposed by customers, how to make the software more user-friendly. During developing software, quality could be taken into consideration in every stage of whole software life cycle, i.e. design, requirements, implementation, testing, maintenance[2]. As for qualities of which aspects, the suitable evaluation time to be chosen and the way to evaluate the quality, these main issues in regard to quality assessment remain to be discussed without stop.

In this thesis, we intend to focus on one of software quality objects - product(source code) rather than cover broad topics in software engineering. Nowadays, several different methods are often used to judge the product quality[3].

- Adding testing management to see whether the software system or every single functionality work well without any defect.
- Code review in the initial phase. Sometimes, the code writers do not examine the code by themselves over and over due to restricted thoughts, but forward it to the fellow programmers to check it for mistakes. It largely depends on their practical experience.

- Checking with detailed software specification.
- Utilizing automatic static analysis tools. We run a static analysis on the source code, which means that the code will be examined without code execution. Afterwards, the analysis report is generated by such tools. Choosing this way, we can find out more bad codings than by manual inspections. However, the feature of 'completeness' results in some disadvantages as well. It is time-consuming in going through every warning, even for the possibility of false positives.

The goal of doing this study is to investigate the application of static code analysis tools, one of above methods mentioned to inspect the quality of the source code, in the enterprise environment. Many research works mainly used either the quantitative technique, which typically performs statistical analysis, or the qualitative technique, which gains information from human side [4]. In this study, we intend to apply both techniques. The combination can improve the empirical evaluation by balancing one limitation of one type with another's strength[5].

The term 'source code metric' is introduced in to present a degree of attributes of source codes. Metrics help to make software measurable and provide us a quantitative perspective to know about the internal facts of the product[6].

We first collect the data independently by different approaches. Through analyzing different types of data, we desire to know about causes of the bugs detected by Findbugs. Moreover, we also intend to study which measured metric may indicate the software external quality. Therefore, the conclusion helps to make quality assessment easier and give guidance to improve the quality in the future.

1.2. Research questions

Following questions are intended to solve in this thesis.

- **RQ1** *Whether the diagnosis result from Findbugs has a significant correlation with special metrics?*

  Findbugs locates the bug patterns in the code, which results in the potential defects in a software program or the system. Although the existence of bad codes is not the only factor influencing the software quality and the project still may be built and deployed successfully even with these error-prone codes, we cannot

neglect and should take efforts to fix them. Because the fewer bugs there are, the less risk the system has to fall down. Hence, for the further maintenance, we will not spend much time in locating the origin of failure.

With the aid of the code analysis tool, we could find out violations of programming rules sets at an earlier stage. In this thesis, we select **Findbugs** as the code analysis tool. These found violations are classified into bug pattern categories. We try to answer whether the rule violations in the sources code reported by such tool has a correlation with codes' internal attributes. Then we further infer what kind of the source code easily contains bug patterns.

- **RQ2** *How is the software metric related to software quality (maintainability)?*

  As DeMarco puts forward that 'you cannot control what you cannot measure' [7], the term 'software metrics' is introduced to quantify product internal characteristics. Project managers and developers are able to track the status in a measurable perspective and make some improvement according to it instead of difficultly-recorded and subjective feelings. Metrics could be deployed in a large scale of aspects concerned with software engineering, such as metrics for requirement definition, metrics for project management, metrics for source code, metrics for the test process, etc[6]. Here, due to the research point, we mainly focus on the product metrics.

  Maintenance is an important step in the development process. The software should be easier to be modified according to bugs or new requirement to improve performance. So, for the company, maintainability will be taken into consideration because of limited time, employees and most important, money in the future development. So, how to evaluate software maintainability is a worthy topic. In context, does software metrics, which expose the features of code in a quantitative way, provide a more persuasive basis for the evaluation? If we can find out such link between them, there is no doubt that it is efficient to help to judge the maintainability.

  To go into this problem comprehensively, we divide it into two sub-questions.

  - **RQ 2.1** *Which metrics are suitable to indicate software maintainability?*

    In this paper, three project cases in an IT Consultant company are our research objects. An effective approach intends to be raised to help developers to judge the project's maintainability. A Set of metrics are available to be

chosen as quality indicators.

– **RQ 2.2** *Whether the importance of some metrics, regardless of the quality aspect, is same among different projects?*

  In the enterprise, the application and users of several projects are quite distinct, which will result in the difference in their software architectures. For example, some online shopping software faces million-users business, the user-friendly interface and ability to deal with concurrent orders are required. Some accounting software manages complex transaction flow. Software architects will put forward various implementation emphases. Through doing this research, we try to analyze the generalization of metrics selection problem. When making the quality judgment in different projects, could we apply the same metric set?

1.3. Related work

Many related research works cover parts of above questions. They settle the problems from the smaller but deeper perspective.

S. Wanger et al.[8] make a research on Findbugs, PMD, the bug pattern tools for Java in an industrial case. This paper shows the results of cost-efficiency, effectiveness, fault-proneness in using these two tools. This paper helps to know about the performance of the static code analysis tools.

André [9] in his master thesis discusses an approach to studying the correlation between bug pattern categories in Findbugs and source code metrics in two Java open-source projects Axis and Tomcat. With the help of the PASW statistical tool, he finds that the bug pattern category 'Performance' has the strong correlation with the most source code metrics and the category has the weak correlation with some other metrics.

Lots of paper illustrates models for measuring software maintainability.

One work[10] of Don C. et al. compares five methodologies to quantify software maintainability from some metrics, such as hierarchical multidimensional assessment models, polynomial regression models, aggregate complexity measures, etc.

A. Kaur et al.[11] suppose a model which is a prominent enhancement on the basis of

the existing Oman and Hagemeister classical maintainability index model [12] (Equation 1.1) at the University of Idaho. A combination of predicted metrics is proposed in this paper. Except for the original metrics in O&H Model, Lines of Code, Cyclomatic Complexity, Percentage line of comments and Halstead Volume, there are about another ten new added metrics used for calculating the MI value.

$$\begin{aligned} Maintainability\,Index\,[12] = 171 - 3.42 * In(avg\,Halstead\,Volumn) \\ - 16 * In(avg\,Lines\,of\,Code) \\ - 0.23 * (avg\,Cyclomatic\,Complexity) \\ + 0.99 * (avg\,Comment\,Lines) \end{aligned} \tag{1.1}$$

The research from W. Li et al.[13] focuses on the validation of several software metrics in maintenance work, which are used in the object-oriented programs.

There are not so many researches, deploying subjective evluation techniques. Among a few studies, Dennis [14] uses a subjective method to relate seven complexity metrics to the maintenance activity and see if what the metric exposes is consistent with practical experience. The result presents that growth in complexity agrees with performance done in the projects.

1.4. Outline

This thesis is organized as followed.

Chapter 2 introduces the usage of tools, Findbugs and SonarQube as the static code analysis tools used in the thesis and gives the definition and importance of maintainability, especially the reason why we chose this quality aspect. At last, the software metrics are defined in a formal way, helping readers to have ideas how the source codes are quantified with some examples.

Chapter 3 compares the study projects in the company we decide to follow. Next, we give the study designs corresponding to each research question and a general explanation how both the quantitative data and qualitative data are collected for supporting conclusions.

Chapter 4 shows the whole implementation process. In this thesis, the implementation is mainly the activities to collect data. First, Data is gathered from the much measure-

ment information (e.g. the metrics and the detected bug pattern) of the source code, which is stored in a huge Sonar Database. We talk about how we establish a local automatic data gathering system in order to search for the most important data we require and give a visualization result in convenient of observation. Second, this chapter also describes how the qualitative data is collected through the interviews with the developers.

In Chapter 5 and Chapter 6, based on the quantitative and qualitative data we have got in the Chapter 4, we make the data analysis and answer each research question separately. Also, some restrictions we met in the implementation are thrown out to reflect the problems we have met.

Finally in the last chapter 7, the conclusion of the study is drawn, together with the possible future research direction for more exploitation in the application of static code analysis.

2. Background

2.1. Code analysis tools used

2.1.1. Findbugs

Findbugs [15] is a static analysis tool, developed by the University of MaryLand. It is an open source project, which helps to examine around 400 potential bugs in Java bytecode. Users either runs it in its Swing interface or integrate Findbugs as the plug-in into many platforms, e.g. Eclipse IDE, SonarQube, Maven, Jenkins.

As the definition of static analysis, Findbugs analyze the code without executing it. Strictly speaking, it is not a testing tool, since it does intend to consider about the business logic in the project. It aims to find out the violations of some programming rules that Findbugs thinks developers should not code in this way. Normally, these bugs detected out are subtle and easily neglected when doing programming. Chances are, the project can be built up successfully with them in a compilation period. However, remaining bugs still have hidden defects among the source code in the running environment. So, we require sophisticated analysis tools.

There are several bug pattern categories in Findbugs. Our study focuses on the relationship between a bug pattern category and metrics. The definition of each bug pattern category is shown in Table 2.1[15, 16]. In this paper, Findbugs as the plugin embedded in the SonarQube detects the potential bugs.

2.1.2. SonarQube

SonarQube is an open platform for continuously inspecting code quality. It covers seven axes in the quality: Architecture & design, Comments, Duplications, Unit tests,

Findbugs		
Bug pattern category	**Definition**	**Examples**
Bad practice	Bad coding that violates recommended rules.	**ES: Comparison of String objects using == or !=** When we want to compare whether two strings represent the same word or sentences, it is unsuitable to use == or !=, which in fact compare the object references. Instead, the *equals(Object)* method is better. **NP: Non-null field is not initialized** The non-null field is not initialized in the constructor or is not initialized before the usage.
Correctness	Coding mistakes, e.g. impossible casting, always the same return in control flow statements.	**NP: Null pointer dereference** A NullPointerException may happen due to the dereference when the code is executed. **Nm: Confusing method names** Some method namings differ by word capitalization.
Malicious code vulnerability (MC Vulnerability)	incorrect fields exposure	**May expose internal representation by returning reference to mutable object** Returning a reference to a mutable object value stored in one of the object's fields exposes the internal representation of the object.
Performance (PERF)	Wrong boxing operation, usage of non-static classes.	**Bx: Method invokes inefficient Number constructor** Initializing an Integer object, either use *new Integer(int)* or *Integer.valueOf(int)*. The latter is preferable because of faster processing speed and no object allocation.
Dodgy Code	Code that is confusing, and error-prone.	**BC: Unchecked/unconfirmed cast** Before doing the object casting, the casting feasibility is not checked or confirmed.
Multithreaded correctness (MT Correctness)	Thread synchronization issues.	**STCAL: Call to static DateFormat** DateFormats are inherently not safe for multithreaded use

Table 2.1.: Findbugs Category Definition[15, 16]

Complexity, Coding rules, Potential Bugs. It supports more than 20 programming languages including C#, C++, Java, Javascript, PHP, etc. Another important feature is that SonarQube integrates with standard Application Lifecycle Management components such as Git, SVN, JIRA.

All the static code analysis in this thesis is initially done by SonarQube. It performs the metric measurements in the scope of the project, subproject, package, class from above axes and records the metrics history. One of the axes - Potential bugs is detected by the embedded externals plugins, e.g. PMD, Checkstyle, Findbugs. That is to say, SonarQube applies all the coding patterns rule as these plugins define. We could also filter the analysis result according to one plugin.

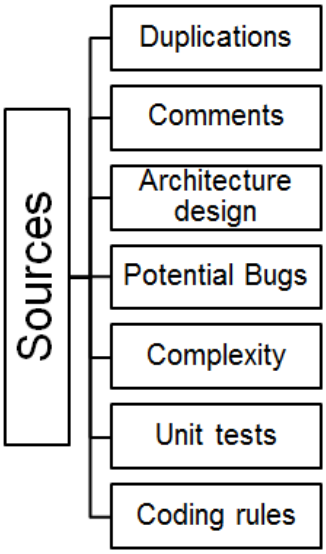


Figure 2.1.: Seven axes of SonarQube Code analysis [17]

The SonarQube used by current development team was initiated since 2012. In the dashboard, we find that there are 18 projects used to be deployed in the SonarQube, although only three of them are still under consistent analysis. Others are stopped because of the termination of the development work.

2.2. Selected software quality - Maintainability

IEEE Std 1219-1998 defines software maintenance as "*Modification of a software product after delivery to correct faults, to improve performance or other attributes, or to adapt the*

product to a modified environment." [18].

To specify maintenance activities in more details, they are classified into four kinds [19]. Figure 2.2 demonstrates Hans[20]'s research result of the proportion of each kind to the total amounts.

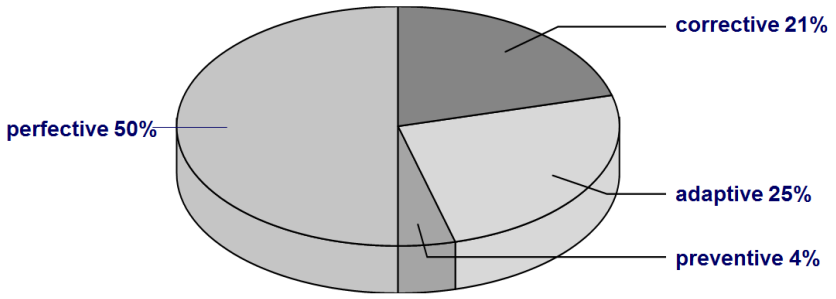


Figure 2.2.: Distribution of maintenance activities

1. **Corrective maintenance** Modifying the software product after delivery to correct found bugs or solve breakdown problems. Normally, users will report these issues to developers. Through diagnosing some logs, the cause is located and fixed correspondingly.

2. **Adaptive maintenance** Modifying the software product after delivery to make it adaptable and runnable in a new environment. For example, the old program developed ten years ago was designed to run on the server equipped with less memory and capacity. Obviously, these low resource configurations are not able to follow up the increasing business demand currently. Consequently, it may be required to immigrate to another operation systems. Alternatively, the source code repository changes from Git or SVN. Any interaction with other OSs, Database Management Systems, network protocols, etc., is viewed as adaptive maintenance.

3. **Perfective maintenance** As the word indicates, the software is to be modified after delivery to improve the performance. There is no such kind of product satisfying everyone. It has the potential to increase the processing speed, to beautify the user interface, or to consider more user cases in order to offer more services, etc.

4. **Preventive maintenance** It belongs to high-level maintenance. The software is

modified to redesign the internal structure or interface in consideration of laying the solid basis for the further development or maintenance work. This requires the developers to have enough foresight to predict the diversity in the future.

With years' accumulation of the practices on the software development, software maintainability becomes a more and more significant concern in the industry. Robert L. Glass [21] points out that the activities of maintenance take a larger share of the software budget than the activities in the development phase. Maintenance mainly occupies about 40 to 80 percentage (60 percentage on average) of the total software cost. The costs are affected by the service life of the product, the architectural framework of the product, the update of the equipment and new technologies, the professional capability of employees.

The software never stops to conform with changes and to be a better, reliable one. If a software product proves to be useful in practice and contented by product buyers, they are willing to pay money to extend new functionalities. IT service providers have the ability to make continuous development it in a cheaper and time-saving way when the software product is maintainable. This advantage is absolutely attractive, resulting in a win-win relationship.

So, to judge the maintainability is one of the major factors in evaluating the software quality. Moreover, in the thesis, we principally choose software maintainability as the concerned aspect instead of going deep into software quality to boundless topics.

2.3. Selected software metrics

The concept of 'Metrics' allows us to quantify an object to a degree according to some defined rules. Then, we can easily trace the status of current product or process by reviewing values[6].

Software metrics are deployed in many areas of software engineering and can mainly be classified into three categories: product metrics, process metrics, and project metrics. As mentioned in RQ2, we only discuss product metrics here. This category of metrics shows the internal characteristics of the product from its size, complexity, design features.

Therefore, developers can evaluate software by these objective data - metrics, with some rule, not just describing it with some 'ambiguous' adjectives. For example, we

say that the complexity of this business case is higher that that one. This sentence is rather confusing, raising several questions. First, how complexity is defined here. It refers to many possible outcomes or refers to the program's logic is confusing. Second, how to compare 'complexity'. Can man's instinct tell us this is complicated or simple?

These measurements of the software product at any stage of software development lifecycle are worth studying ranging from requirement statement to ongoing working process. These metrics are then used to estimate/predict product costs, schedules and to evaluate productivity and product quality[22].

Following product metrics are selected as several characteristics we are going to research on the product. The measurement of these metrics is referred to the definition given by SonarQube[17].

Lines of code (LOC) This metric is a product attribute from the aspect of the physical size. It shows us the number of total non-blank lines in a class file (i.e. those lines contain at least one letter which is neither a whitespace nor a tabulation). The reason we skip the blank line is that it does not contribute to the same effort like the one presenting complex algorithms or functional issues. Sometimes, it estimates the programming productivity or maintainability. For example, the company could use total lines of code to compare the scales of different projects or to see the trends in a project during the developing process. Within a class file, we can also make a deeper analysis to find out whether the size will influence the occurrence of bugs.

Comment line density (CLD) Although typing comments for variables, methods has no influence on software running, it greatly helps developers to maintain the software. Especially, when the company takes over the project which used to be developed by others a long time ago, it is important for them to get quickly familiar with its business cases by reading the documentation. Reviewing the code line by line is also a possible approach, but it costs much time. Comment Line Density represents the percentage of comment lines in the whole class file. It is chosen as metric instead of a direct count of comment lines because it is hard to make the judgment of the completeness of documentation just by the number. A small-scale class with a few comment lines might also contain the readable description. The calculation of comment line density in a class file shows

as below.

$$Density\,of\,comment\,lines = \frac{Comment\,lines}{(Lines\,of\,code + Comment\,lines) * 100} \tag{2.1}$$

McCabe's cyclomatic complexity (CPLX) Cyclomatic complexity tells us how many independent paths are there in a program. The function is split into different flows in Java by adding decision-making statements, such as *if* , *if-else* and *switch*. Each method has a minimum complexity, i.e. 1. In Accessors (Getters) and Mutators (Setters), CPLX equals to 0. A program with high complexity values needs to be handled carefully. Developers should know very well about the logic behind, the activities should be performed under which conditions.

Calculating of McCabe's cyclomatic complexity for a single class [23]:

$$cc = E - N + 2 \tag{2.2}$$

Where:

$E = the\,number\,of\,edges\,of\,the\,graph$
$N = the\,number\,of\,nodes\,of\,the\,graph$

For example:

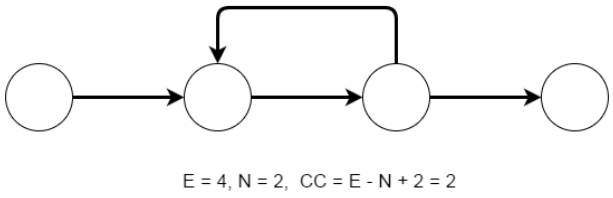


Figure 2.3.: McCabe's cyclomatic complexity calculation example

Coverage (CVR) In SonarQube tool, Coverage is defined as a mix of line coverage and Condition coverage[17]. This provides a more accurate answer about how many percentages of the source code has been covered by the unit tests[6]. Especially, in some service beans with complicated business logic, it is necessary to implement testing codes. Then, we run our tests when we build the project or we utilize the plugin in IDE, e.g. JUnit. We are checking whether we have got the expected results. Higher coverage value gives us more confidence of success in a real running environment because more cases are taken into consideration to be

exercised by testing.

$$Coverage = (CT + CF + LC)/(2 * B + EL) \tag{2.3}$$

Where[17]

CT = conditions that have been evaluated to 'true' at least once
CF = conditions that have been evaluated to 'false' at least once
LC = covered lines = lines_to_cover - uncovered_lines
B = total number of conditions
EL = total number of executable lines (lines_to_cover)

Efferent coupling (EC) A coupling metric belongs to software package metrics, which target at the usage in the object oriented language. One of the characteristics of such programming language different than others is that every data item is treated as an object that may inherit other objects or implements interfaces, and also invoke the variables, functions in other class. So one class may depend on others, or be depended on by others.

Efferent coupling measures the number of other data types which a class imports. This includes inheritance, interface implementation, parameter types, variable types, and exceptions[24]. From the following code snippet, the class Car has both a HAS-A and IS relationship. The car is composed of many parts, such as an engine and wheels. Engines and wheels are also specific data types, which there might couple with many other attributes. What's more, the car's color might be red, yellow, blue. These alternative colors are listed in the enumeration data type Color.

```
public Class Car {
        private Engine engine;
        private List<Wheel> wheels;
        private Color color;
}
```

3. Study Design

In the section 1.2, we have listed two research questions to be solved in this thesis. Now, we set up study designs for them.

3.1. Projects and context

Three projects which have been being mainly developed in NTT DATA, a IT Service provider, recent years are selected as the study projects. Each project team consists of around eight team members, including a software architect, a requirement engineer, several developers and testers. They all are implemented, built and deployed in Java Enterprise Environment(J2EE)[25]. Many J2EE standard service interfaces, such as Java Database Connectivity(JDBC), Enterprise Java Bean(EJB), Java Persistent API(JPA), JavaServer Faces(JSF), are used as main technologies. The development team applies the Agile development management and release a Sprint version every two or three weeks.

Project A Project A is a repository, the central instance administrating, storing and providing flashware, delta updates or map data to vehicles of some automotive brand worldwide. The logic behind each use case is rather complicated. Some of the use cases result in more than ten possible responses. It refers to many verifications of the request schema, resources, and hash code validation. Until now, the project attains the level with 30,200 lines of code and 410 classes and will be continued working on in the future.

Project B Project B is a relatively larger and maturer project, which has been conducted for several years. It centralizes the flashware calculation for electronic control units (ECU) integrated into vehicles. The client tier provides Eclipse Rich Client Platforms. The latest version contains around 120,000 lines of code.

Project C This project provides business cases regarding flashing and coding processes

Table 3.1.: Brief projects' introduction

Project	LOC	#packages	#Classes
Project A	30461	51	419
Project B	41612	98	530
Project C	46181	123	576

of the After-Sales for several vehicle categories. Clients can connect to this system and submit different requests via HTTPS protocol. So, different from previous two projects, the presentation tier is of better design. The development of this project was terminated in May 2015. The last version contains approximate 46,000 code lines and 576 classes.

3.2. Research Question 1

RQ1 *Whether the diagnosis result from Findbugs has a significant correlation with special metrics?*

For this question, at first, we investigate the correlation between two objects via the statistical technique. We depict the cross factorial design which was used in Andre[9]'s work. Here, software metrics and bug pattern category are set as two independent factors. A level is a subdivision of a factor. We choose six bug pattern categories defined by Findbugs and five interested metrics. So, in this case, metrics have five levels and bug patterns have six levels. We cross each metric with each category.

Their correlation relationship will be expressed as a quantitative value - *Spearman's rank correlation coefficient*[26]. In statistics, Spearman's rho is abbreviated as the Greek character ρ(rho). It is a nonparametric measure of statistical dependence between two variables. It presents how the relationship is between two variables by using a monotonic function. The input is at least two sets of a pair of values. The output is ρ with a P-Value (which measures the observed sample results about a statistical model). If ρ exceeds the predefined threshold and P-Value indicates highly statistical significance, we think there exists a strong correlation between two variables. Many online calculators or power statistical computing softwares, such as IBM SPSS, R, etc. can be chosen to get the correlation coefficient value.

The correlation coefficient between ρ is calculated on the basis of the selected classes. In one class, its internal attributes defined as metrics are measured by SonarQube as

well as the number of detected bug patterns. Table 3.2 presents the types of data in the class file, which could be collected from SonarQube.

Any two columns of data are chosen for paired data. If the number of sample classes is denoted by n, the bivariate distribution (X,Y) is denoted by sample values (x_1,y_1),(x_2,y_2), ... ,(x_n,y_n)[26]. Here, we set the factor *Metric* as X, the factor *Category* as Y.

As a result, a 5*6 matrix will be generated. Each number in the matrix stands for the correlation of one corresponding category and metric. Table 3.3 reflects the correlation matrix between each metric and each category happening in Project * at a time point.

Table 3.2.: Information contained classes in one project case

	Metrics			Number of bugs		
	Metric 1	Metric 2	...	Pattern Category 1	Pattern Category 2	...
Class A						
Class B						
...						

Table 3.3.: Correlation coefficients matrix - Project * - Version *

		Findbugs bug pattern categories					
		Correcteness	Bad practice	MT correctness	Doddy	PERF	MC vulnerability
metrics	LOC						
	CLD						
	CPLX						
	CVR						
	EC						

We intend to analyze the relationship from two dimensions as Figure 3.1 presents.

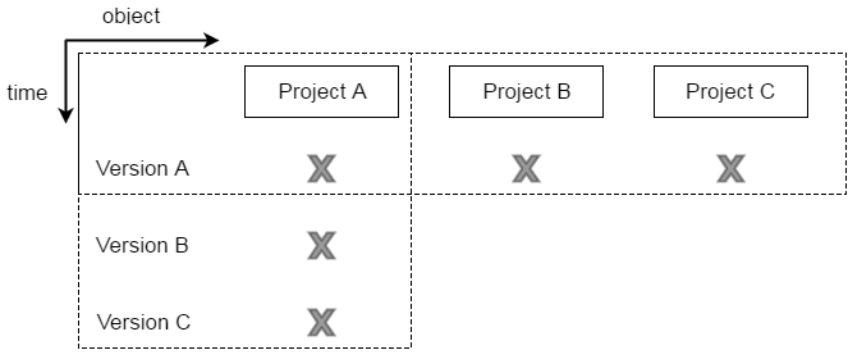


Figure 3.1.: Two approaches design

Different versions approach Project A is selected as the research object that matches the selection criteria because of continuous development till now. In the thesis-writing period, the measurement of project A takes place three times. In the end, we can get three matrixes. The sample classes in each matrix are the same, i.e. the classes in Project A. During different versions, source codes are changed due to the addition of new functionality and code modification. Following this approach, we are able to see the tendency of the coefficient values in order to find out whether this correlation relationship will be strengthened or weaker or keep stable along with the development of the software.

Different projects approach Projects are measured and bug detected in different periods due to manual configuration. Project A, B are measured after each new version delivery. Project C is measured every day. We decide to select the last revision in 2015 for three projects. For each project, a matrix can be generated based on classes in that project, containing some bugs, which are worth being picked as sampling data. The 'perfect' null-bug classes are not able to reveal the relationship between bugs and metrics.

As a result, three matrixes for three projects respectively are generated. We can observe that:

1. How is the correlation relationship between metrics and bug pattern categories in each project

2. Through comparison among different projects, we can find out whether such correlation is only specific to a certain project or could be concluded in general cases.

Except for analyzing the correlations obtained from a quantitative measure of Spearman's rank correlation correlations, expert judgment is introduced in assisting verifying the conclusion. The usage of the subjective evaluation is motivated by the importance of the years' experience in software developing. Experts have the sharp judgment to know what the result of analysis tools indicates and what kind of bad codes are easy to contain bug pattern. We compare their evaluation with the actual correlation data we get from quantitative research. Two different research results might not be identical. We need to find out which of conclusions they can reach an agree, which they can not and why.

3.3. Research question 2

RQ2 *How is the software metric related to software quality (maintainability)?*

The relationship between software metrics and maintainability could be probed into with the quantitative research or qualitative research. A Quantitative research places emphasis on objective measurements. Normally, we can get the result in the form of the statistical or numerical values. Differently, a qualitative research generates the exploratory output (usually in words).

Coding approach Carolyn B. Seaman raised a technology calling 'coding' in empirical software engineering in her paper [27]. Coding is a combination of both quantitative and qualitative methods. The main idea of this technology is to extract out data of quantitative variables from qualitative information.

We are going to code the interview conversation notes. By means of transforming experts' subjective perspectives into straightforward and accurate data, we could get quantitative results, which are easier for the further data processing. During the interviews, several topics concerned with quality assessment are expanded. Developers are expected to answer them with their rich experience. But, We can obtain quantitative values directly as well. For example, we could invite the interviewees to do grading on the degree of correlation between selected software metrics and maintainability. These metrics are *Lines of code*, *Comment lines density*, *Complexity*, *Test success density*,*Coverage*. They will give the mark between 1 and 5. 1 means the weakest correlation, on the other hand, 5 means the strongest correlation. If the majority agree on the same type, we can draw the conclusion that which metrics are suitable to indicate maintainability.

3.4. Data collection procedure

3.4.1. Quantitative data collection

All measured data we intend to extract and refine are stored in SonaeQube Database.

The potential bugs are detected by Findbugs-plugin, which is embedded in the SonarQube. all found bugs are stored into a table 'RULES_FAILURES' in the database. Joined with Table 'RULES', the bugs detected by the Findbugs plugin are filtered out.

Table 3.4.: table 'RULES_FAILURES' and 'RULES'

RULES_FAILURES	
id	int
snapshot_id	int
rule_id	int
failure_level	int
message	varchar(4000)
line	int
cost	decimal(3,2)
created_at	time
checksum	varchar(1000)

RULES	
id	int
plugin_rule_key	varchar(200)
plugin_name	varchar(250)
prority	int
plugin_config_key	varchar(500)
name	varchar(200)

Table 3.5.: table 'PROJECT_MEASURES'

PROJECT_MEASURES	
id	int
value	decimal(3,2)
project_id	int
metric_id	int
snapshot_id	int
rule_category_id	int

The metrics are measured by SonarQube system. All the information concerned with metrics whatever in the level of project, component, package, class are stored in table 'PROJECTS_MEASURES' (The table schema shows in Table 3.5). Research samples are classes in three mentioned projects, we can filter classes by selecting its 'project_id'.

Finally, joining table 'RULES_FAILURES' and 'PROJECT_MEASUARES' with the key column 'snapshot_id', following information in each class could be got from SonarQube as shown in Table 3.2.

Problem

After five-year usage, there are millions of entries in the table 'PROJECT_MEASUARES'. Moreover, in the database design, there are many references among tables. After typing SQL query for joining 'PROJECT_MEASURES' and 'RULES_failure' and other relative tables, we can not get the output as we expect because of 'java.lang.OutOfMemoryError:Java heap space' error.

Solution

In order to solve this problem, we decide to establish an automatic data collection system locally.

We need to make clear how tables and their schemas in SONAR database are structured at first, then, extract data we actually make use of and store them into a new local database. Therefore, it will be quicker to join the multiple tables on a smaller scale.

Besides, we can also further process these extracted data and make mining out more useful information possible.

3.4.2. Qualitative data collection

Interviewing is chosen as the technique for gathering qualitative data. Three developers, who are working in NTT DATA and are also involved in these projects, are our interviewees. Each developer may have their own ideas of using static code analysis tools in evaluating software quality. Learning their practical experience helps to inquire about how software maintenance works in practice.

The interview will be taken separately and last for forty-five minutes. First of all, an invitation is sent to them to inform of the interview date, place, also attached with the interview goal and topics to be discussed. Then, around ten questions are designed to be asked during the interview. What's more, considering that we should guide developers to think in a relatively academic way and give an answer in accordance with our expectation.

Before the interview, we prepare some visual reports, i.e. line charts, tables, about data describing project's attributes. Developers could be fast familiar with project's situation in the interview instead of reviewing the code. We hope that their judgments could be given based on the powerful evidence. Conversation records are taken in order to write interviews report and fetch essential viewpoints afterwards.

4. Implementation

In the chapter 3, we give the study designs to each research question and list out which data we need to gather in order to help to derive out the final conclusion. This chapter mainly describes how we gather these quantitative or qualitative data.

4.1. Quantitative data collection

In the development team of NTT DATA, the mature continuous integration system is deployed for the benefit of the automatic implementation in building, testing and managing projects in a specific period.

In this integration system shown in Figure 4.1, the developers synchronize with others' work by maintaining their codes in the version control system - SVN. The codes are committed to SVN after the completion of the new task or the modification of the previous code. Jenkin, the continuous integration server, has a trigger on the repository, ca every five minutes. Then, it checkouts the project code from SVN and makes a normal continuous build on it. Developers could also manually start it step in GUI. Jenkins triggers SonarQube periodically due to the project configuration. For example, Jenkins asks SonarQube to inspect code quality in Project C nightly. SonarQube in version v.2.11 was first established in the company in 2012.

Until now, around 20 projects have been deployed in this platform. Except for the finished projects, three active projects are left currently. Due to different configurations, they are measured in fixed period or after each delivery. The each project is measured in a detailed way. SonarQube drills down each project into many levels, in the scope of projects, components, packages, classes.

Figure 4.2 shows general metrics of Project A in the version of 1.2.0.3 measured on 09 Mar 2016. In the set of widgets. We can quantify the project by these values in categories of size, comment, complexity, coverage, test, violations, etc.

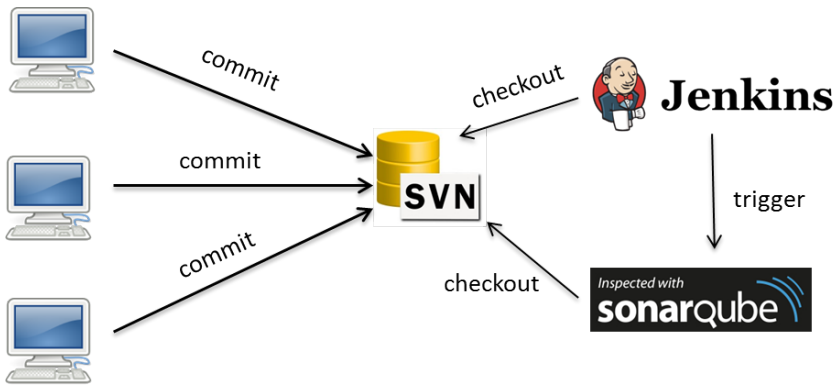


Figure 4.1.: Continous integration deployment in company

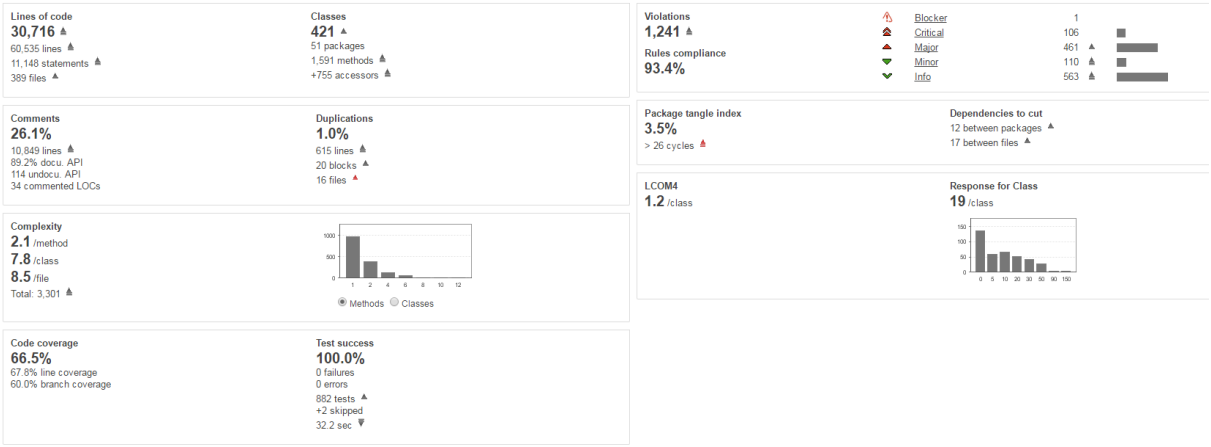


Figure 4.2.: SonarQube UI of project metrics

Then we step into the portal further, we can observe the metrics in the scope of class as Figure 4.3 displays. Some different from the metrics in the scope of project, the metrics which are specially defined for the class are taken at the same time. For example, in the 'Source tab', we can also see the metrics called *Public API*, *Number of Children*, *Depth in Tree*, *Response for Class*. Another important thing is that in 'Violation' subpage, each bug detected by Findbugs Plugin, PMD Plugin, Checkstyle plugin, etc. is highlighted with red color. We can also filter out the interested bug category in the drop-down menu.

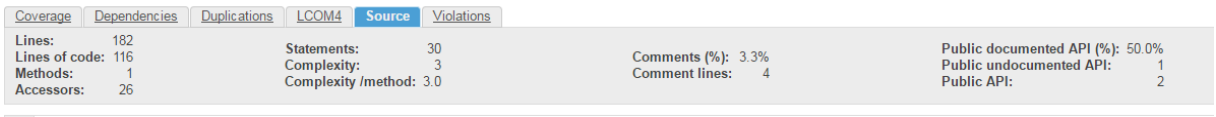


Figure 4.3.: SonarQube UI of class metrics

Each information displaying in the user interface is data stored in the SonarQube Database. There are 110 available metrics and thousand of rules provided by SonarQube. Each metric and bug detection rule will be applied to all the classes in each project at every measurement point. As a result, the data storage grows linearly. For the convenience of analyzing data we principally focus on, it is necessary to extract out part of data from massive records in the database.

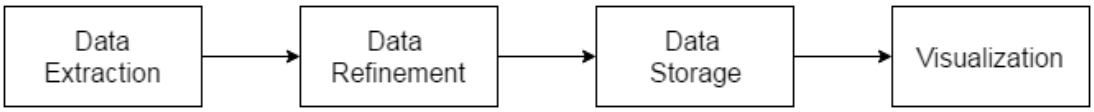


Figure 4.4.: Data processing step

4.1.1. SonarQube Database structure

After having access to SonarQube database in the company, we make a remote connection by a SQL Client. Each table including its primary keys, foreign keys and other attributes is reviewed, especially together with the references with other tables. Figure 4.5 presents the part of data modeling structure of the SonarQube database. Some tables concerning with the widget configuration in the web user interface are omitted here.

Table 'PROJECTS' Every measurement object is viewed as a 'project' in this table. It may be the entire project, then recursively deep inside, a component, a package and last the small unit - class. The type of the measurement object is marked in the column 'qualifier'. The column 'root_id' shows the project id of the top level, i.e. project level.

Table 'METRICS' This table lists the metrics SonarQube will measure on each item in Table 'projects'. We can find metric's name, the full description, and unit of the measurement.

Table 'SNAPSHOTS' The word 'snapshot' means one view of status repository as it was at a particular time for the purpose of revision control in software development [28]. As mentioned in the former chapter, the projects are measured in some period. Although the SonarQube web interface shows the newest situation, it also keeps some historical data in the database for the convenience of tracking the changes. Thus, we are able to tell differences among revisions of the same measurement item by the unique snapshots id.

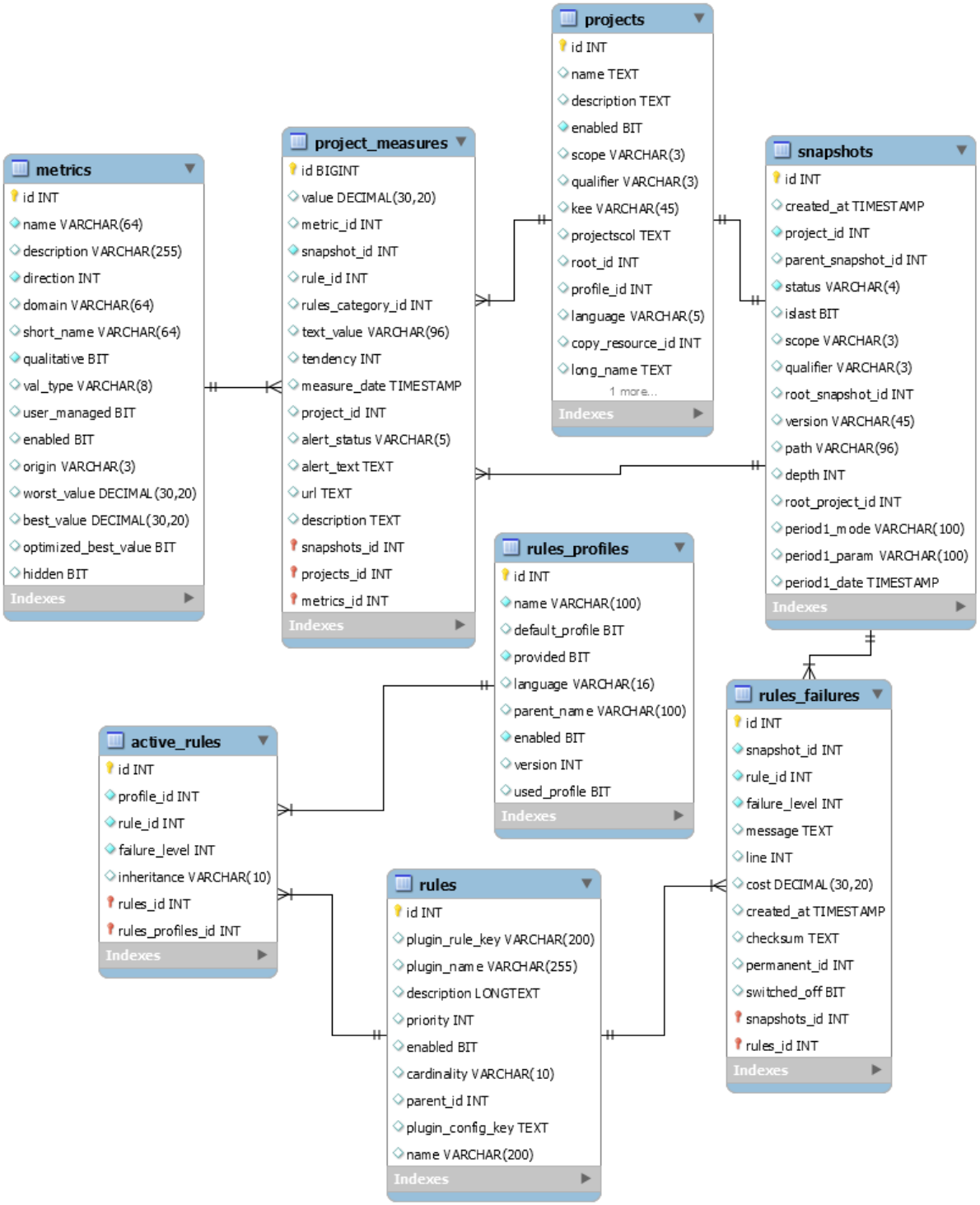


Figure 4.5.: SonarQube Database structure

Table 'PROJECT_MEASURES' This table plays an central role in the database. All actual metric values are stored there. Each row tells us the value of one metric for one version of measurement item. For example, a class in a project has been taken snapshots for six times in one month. For each time, there are 50 metrics applied for this class. As a result, total 300 rows are inserted for this class monthly. The record for one time measurement is very complete, so the capacity of this table is high. Currently, there are five millions of entries in this table.

Table 'RULES' Each embedded plugin in the SonarQube defines plentiful violation rules for possible bugs, bad coding styles and flaws. Now, the plugin 'Findbugs', 'checkstyle', 'pmd' are configured in SonarQube. Total 742 rules are provided by them.

Table 'RULES_PROFILES' Users have an option to select sets of rules in Table 'rules', which they think are more meaningful to examine the code quality. The name of the rule group is put here.

Table 'ACTIVE_RULE' Related to the above table description, table 'active_rule' gives out which rules among all are activated in one rule profile.

Table 'RULES_FAILURES' All the rules violation information is found here. The figure 4.3 is a visualization of boring data entry in the database. It helps to answer the question, which line of which version of class violates which rule. Furthermore, the violation is graded, from 0 to 4. This table is used to help to gather bug information, however, unfortunately, after each measurement of a project, the original data will be overwritten. Then, we can only check the bugs in the current version.

4.1.2. Local database structure

Figure 4.6 displays local database schema. The designing goal is to simply the original SonarQube database, in order to extract out most needed information and also reduce the number of joins among several tables.

Table 'PROJECTS' This table contains every measurement entry in the *level of project*, whose qualifier is 'TRK' in 2015. As we know, projects are measured according to different period. So, we distinguish different versions from unique 'refer_snapshot_id'.

Figure 4.6.: Local Database structure

Datasets from SonarQube database:

```
SELECT less.id as REFER_SNAPSHOT_ID, less.created_at as
    CREATED_AT, less.project_id as REFER_PROJECT_ID,
    projects.name as PROJECT_NAME
FROM (SELECT * FROM SNAPSHOTS WHERE DATE(created_at)
    >='2015-01-01' and qualifier='TRK') as less, PROJECTS
WHERE less.project_id = projects.id;
```

Table 'CLASS_ENTRY' This lists every measurement entry in the *level of class*, whose qualifier is 'CLA'. Classes have different versions due to code changes among different version. So, the column 'refer_snapshot_id' is set uniquely as same as the project. What's more, we can know about which version of project the class belongs to, through its attribute - 'refer_root_project_snapshot_id'.

Datasets from SonarQube database:

```
SELECT less.id AS REFER_SNAPSHOT_ID, less.root_snapshot_id
    AS REFER_PROJECT_SNAPSHOT_ID, less.project_id AS
    REFER_CLASS_ID, less.root_project_id as
    REFER_ROOT_PROJECT_ID,projects.name AS CLASS_NAME
FROM (SELECT * FROM SNAPSHOTS WHERE DATE(created_at)
    >'2015-01-01' and qualifier='CLA') AS less, PROJECTS
    WHERE less.project_id = projects.id;
```

Table 'BUGS_PATTERN_ENTRY' It stores all the programming rules or bug pattern rules defined by external plugins in the current active rule profile. It is worth mentioning that setting severity for each rule is helpful to filter the bugs by its level, providing another perspective to classify the bugs for the later research.

Datasets from SonarQube database:

```
SELECT r.id AS REFER_BUG_PATTERN_ID, r.plugin_rule_key AS
    RULE_KEY,r.name AS PATTERN_NAME, ar.failure_level AS
    SEVERITY, r.plugin_name AS PLUGIN_NAME
FROM RULES r, ACTIVE RULES ar
WHERE r.id = ar.rule_id and ar.profile_id=11;
```

Table 'BUGS_ENTRY' All the potential bugs detected out by external plugin 'Findbugs' are stored here. Each bugs belongs to one class of the certain version. That

is mean, we do not only just identify the bug by the class name, otherwise,it will occur duplicated rows if the bug still remains in the later version. So, we also add the column 'refer_snapshot_id' as well to indicate the class of certain version. By using the key 'refer_bug_entry_id', we can track the detailed violation information in table 'BUGS_PATTERN_ENTRY'.

What's more, the original table 'RULES_FAILURES' in SonarQube database will refresh itself after each code analysis instead of keeping the historical data. We need to gather bugs several times and add new entries in 'BUGS_ENTRY'.

Datasets from SonarQube database:

```
SELECT rf.id AS REFER_BUG_ENTRY_ID, rf.snapshot_id AS
    REFER_SNAPSHOT_ID, rf.rule_id AS RULE_ID,ru.name as
    PATTERN_NAME, rf.failure_level AS FAILURE_LEVEL
FROM RULES_FAILURES rf, RULES ru
WHERE rf.rule_id = ru.id and ru.plugin_name='findbugs'
```

Table 'STATISTICAL_METRICS' We select several interested metric whatever is for the level of class or the level of project.

Due to large redundancy data in table 'PROJECT_MEASURES' in the SonarQube database, it takes much time to go through the whole table and filter out part of data.

This table joins table 'BUG_ENTRY' with the key 'refer_snapshot_id'. Finally, we could both metrics and bugs information in one class as Table 3.2 designs.

Datasets from SonarQube database:

```
SELECT snapshot_id as REFER_SNAPSHOT_ID,metric_id as
    REFER_METRIC_ID, value as VALUE, id as REFER_MEASURE_ID
FROM PROJECTS_METRICS
WHERE metric_id in (3,15,20,34,36,60,61,21,80)
```

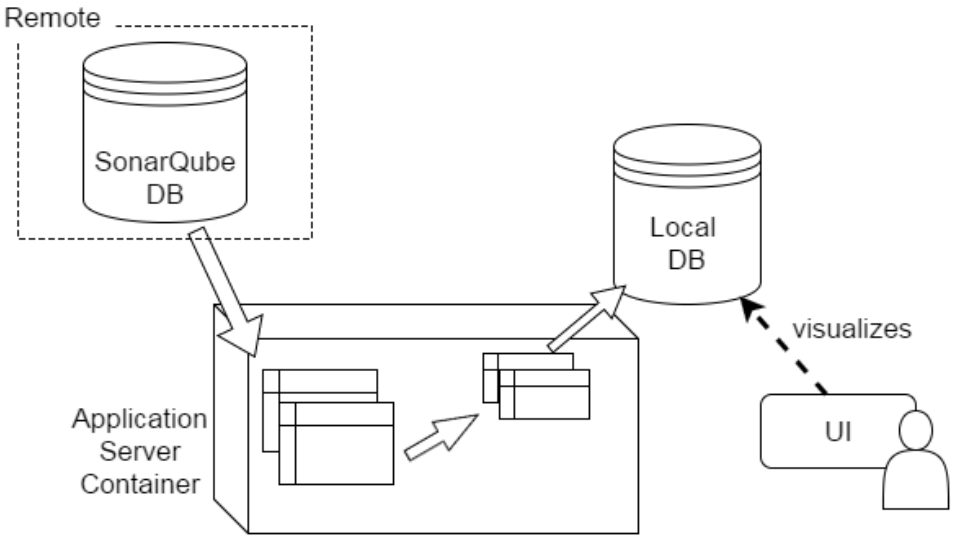


Figure 4.7.: local data gathering system

4.1.3. Building an automatic data collection system

Goal

As we have summarized in the last section 4.1.1, there are a vast amount of data in the SonarQube database, it will take much time to filter out required data from tables, even will cause heap space out of memory problem when we need to handle 'SQL join' among several tables.

The primary goal to set up data gathering system is that we could extract part of data from the original data source and store them in the form of new data structures in another database through the operation in a user interface. This user interface can also show some meaningful statistic results after doing some data processing.

We set up a web application system for data gathering and visualization. The used architectural framework is built up as Figure 4.7 shows.

System Specification

Following tools and technologies used:

JDK 1.7.0 Java Development Kit provides the platform for Java Standard Edition, Java Enterprise Edition, implementing cross-platform compatibility - "Write once, compile anywhere". The version 1.7.9 is downloaded from Oracle official site

[29]. Then we set JAVA_HOME environment variable in local advanced system settings.

MySQL Community Server 5.7 MySQL is an open-source relational database management system. Structured Query Language (SQL) is used as the language for handle the record in the database. Self-defined tables will be created in the local database. Therefore, data selection is quicker and easy to handle.

Glassfish 3 server Glassfish as Java EE Application server framework offers a server environment to run web application. Normally, user could access the application with an absolute URL in the localhost.

Further more, we need to configure both remote Sonar JDBC data source and local JDBC data source in Glassfish in order to implement the database connectivity in the web application.

Java Server Faces JSF is a standardized component-based User Interface technology in Java EE. We utilize its components to generate the menu, buttons, tables and graphic charts with Ajax technology in the front end.

Java Persistence API JPA is a Java specification for doing the mapping between Java objects and a relational database [30]. It gives us a way to manage the data in the java program, including inserting, updating, selecting, deleting data in the database. The manager called 'Entity manager' is responsible for persisting the connective context with one database.

Eclipse IDE It is an integrated development toolkit. In the workspace, we can write the code with help of many auxiliary plugins. Then we export this web application into a WAR file, that later will be deployed in the Glassfish Server. Importantly, we need to import external jar packages, e.g. support for JSF 2.0 and JPA in the jar library.

These steps will be done concretely in building up the data gathering system:

1. Support tools installation and configuration

    a) MySQL 5.7 is downloaded from the official website. In MySQL 5.7 Command Line Client, we initialize one database called 'METRIC_DB' and create a user who is granted privileges required for the database operations. The JDBC URL of connecting to a MySQL server in the localhost is

jdbc:mysql//localhost:3306/METRIC_DB

b) Glassfish is downloaded from the official website. Start the glassfish, and login the glassfish admin console, located at *localhost:4848*. Then we set up two JDBC connections in Glassfish. First is for remote SonarQube database. Second is for local MySQL database. In the left side of console, we expand the tree -> Resources -> JDBC -> JDBC connection pools. The we created connections for each database. Following properties is needed for configuration: pool name, resource type, driver class name, database URL, user and password.

2. Programming

a) Design the schema and references of tables in the local database. The 'create table *TABLE_NAME*' scripts are attached in the appendix A.

b) According to each table's structure in both local and remote database, we create corresponding JPA entity objects, which persist states with a whole/-part of tables in the relational database.

c) Create the entity managers, who control the persistence context and entities.

d) Create the transformation classes, responsible for converting one data structure into another.

e) Build the front-end for the web application. We could observe following information, essentially speaking, we visualize the status of entity objects.

For each research project,

- The tendency of seven metrics in the project level in 2015 are separately shown as a line chart. More specifically, we pick the first measurement entry for each month, as a result, twelve data samples for one project are chosen. then we can see how project metrics have varied in the whole year.

- As we have mentioned in the former chapter, bugs detected by Findbugs plugin are divided into several categories. In each category subpage, we find out those classes, whose contains at least one bug belonging to that category. First, a bar chart is generated, displaying the distribution of the classes, i.e. how many classes have only one bug in that bug pattern,

how many classes have two bugs, have three, four... Second, a table lists class metrics for each error-prone class. Last, an array represents the Spearman's rank correlation coefficients between every metric and current bug pattern category.

f) Select appropriate gathering time points. Due to the configuration of the remote SonarQube database, although after each measurement, the project metrics will always be kept, the record of class metrics and bugs will be refreshed. In our local database, we intend to keep all the data at different gathering points. For project A, the project will be entirely measured after every delivery. So, the data gathering is performed right after that, normally monthly.

3. Data analysis

When gathering data once for a certain project, a matrix like Figure 3.3 is formed. As we have designed in Chapter 3, we could compare matrices of the same project but in different versions, or compare matrices among different projects. Through analyzing why the element in the matrixes displays like this, we expect to answer the research questions.

4.1.4. Web UI of the system

After we complete the programming part, the project is exported as WAR file. Then in the Glassfish Administration console, we deploy the war file on the web server. The link to the web application is *http://pc45722:8080/MetricsDataAnalysisProject/.*

In the left navigation bar, all three projects are listed, and the hyperlink of pages for metrics in system level and bugs information can be chosen in the toggleable menu.

- **Metrics in project level**

Clicking 'ProjectMetrics' in the drop-down list, Figure 4.8 is the result of the project metrics in project A. Each graph presents the tendency of one project metric in 2015. We find out that this project is under continuous development in 2015. The scale of the project becomes larger and larger because of the increasing value of *LINES OF CODE*. The total *COMPLEXITY* rises with the size of the project as well. The average complexity among classes, i.e. *CLASS_COMPLEXITY* varies between 6 and 8 unsteadily. The *RULES_COMPLIANCE* is stable around 94%

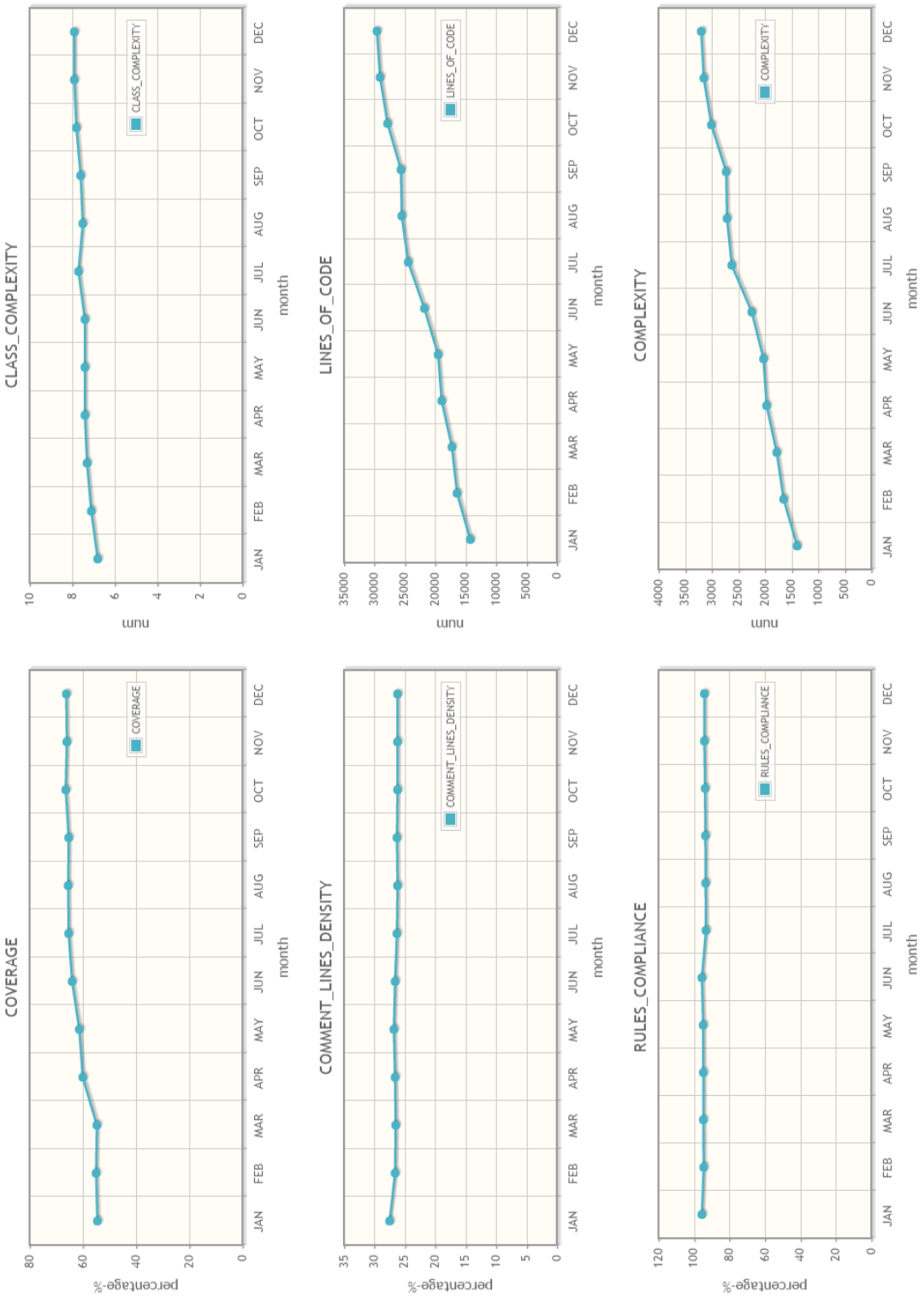


Figure 4.8.: Web UI - Project A - Project metrics in 2015

although more codes are added. Towards *COVERAGE*, as a whole, this metric is growing despite of small ups and downs in the second half of 2015.

The project metrics of other two projects are also available for the observation. Here is not described in detail.

- **Bugs and metrics information in class files**

  In the right-top panel of bug information page, nine command buttons are listed to click for the subpage about the statistical data under particular bug pattern category of Findbugs.

  Figure 4.9 is an example of 'Correctness' bugs found in Project A at version A. As we design, the samples are classes in that version of project. From this page, we get following information.

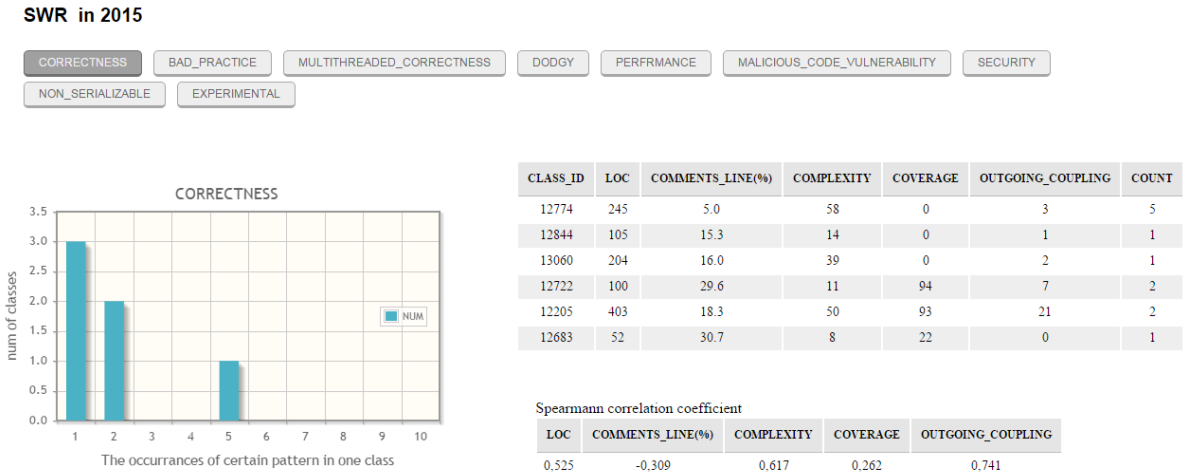


Figure 4.9.: Web UI example - Project A - Version A - Bug Category 'CORRECTNESS'

1. The bar chart displays that three classes contain only one 'Correctness' bug, two classes contain two 'Correctness' bugs, one class contains five 'Correctness' bugs.

2. The upper table lists the metric information for total six classes. Among them, CLD (Comment Line Density) is value in the unit of percentage. For each class with at least one Correctness bug, we could inspect its class metrics and the number of Correctness bug. If we intend to review the code in a class, we fetch it 'class_id', then search its class name in the local database

system by 'SELECT * FROM class_entry WHERE refer_class_id = *class_id*' SQL statement.

3. The lower table gives the result of correlation coefficients between pattern category 'Correctness' and each metric in this version of project. It is shown that the Correctness correlates positively in various degrees with LOC, CPLX, CVR, EC and correlates negatively with CLD.

   Doing similar operation, other bug pattern correlation situations are displayed on the screen for research.

4.2. Qualitative Data Collection

We conduct 'Interview' as a qualitative data collection method. This is an efficient approach to gain the firsthand practical information about any issue during the software development process. The result of qualitative methods is richer and contains much more information than of quantitative methods[27].

We has made two rounds of interview in total. The first one takes place around in the end of January. It aims to collect developers' impressions or viewpoints on some research objects. The second one takes place at the beginning of April. Before that, we should have gained the conclusions about our research questions after finishing the analysis based on automatic data gathering system talked about in the Section 4.1 and results in the first-round interviews. The objective is to refine our current conclusions.

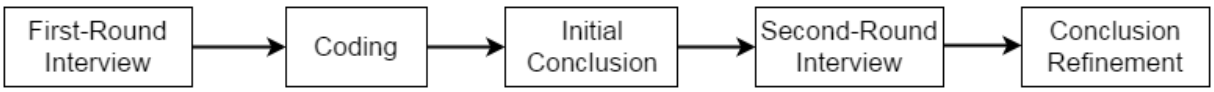


Figure 4.10.: Qualitative Data Collection Process

4.2.1. First-round Interview

The main steps in the whole process:

1. Preparation

a) Setting the interview goal and topics

First, to make clear what we want to know about from developers, the outline is written down. Here, two topics are the focus of the interview. (1) the usage of static code analysis (2) the practice of maintainability.

Due to the limitation of the duration - 45 minutes, around 10 questions (Figure 4.11 lists) covering above topics are designed. We hope that all the responses are open-ended, not simply given by yes/no.

In addition, in order to remind of the whole flow and avoid ambiguity when asking the question, we make a PowerPoint slide, including the questions and vivid graphs to help understand. And all the materials (e.g. correlation results calculated in Sec 4.1, the UI for data gathering system), used for demonstration are examined through once before the interview. Otherwise, you may waste time on searching them and feel overwhelmed.

b) Selecting the interviewees

Three developers in NTT DATA, who all have more than five-year developing experience and engage in projects for a long time, are selected as the candidates. The interview is made separately for them.

c) Making an appointment of interview date

The invitation is sent to each of them, attached with the proposed date and time, also with the aims of interviews.

2. Face-to-Face interview

We ask for the permission to record the conversation in the interview.

a) Opening

At the beginning of the interview, a brief introduction about the research work being conducted currently is presented to the developer. Then, it is more helpful for them to participate fully in the certain topics quickly.

b) Questions and answers

We follow the designed flow, ask the developers questions. At the same time, simple snippets are written down on the papers if some viewpoints arouse the interest, which is worth deeper discussion and thoughts. Some

- What's your definition of maintainability?
    - Combined with various definitions online, from which aspects, the team will implement 'maintainability'
- Briefly introduce the software development cycle of current projects in the company, when to implement 'maintainability', once or iterative?
- Which analysis tools would you prefer to use during development, PMD plugin, Findbugs plugin, Sonar? Have you ever used Findbugs?
    - Will the detected bugs influence your judgment about software quality?
    - What kind of bugs you will fix normally? Why? Take an example
- Brief introduce the researched projects, their main functionalities and properties
- How will/did the team judge quality (maintainability) during the development
    - Which factor will influence your judgment? Metrics?
- Is there any delivered version (in 2015) of some project, on which we mainly did the improvement of maintainability?
    - Goal : analyze how the metrics changed between two versions
- Grading the correlation between maintainability and certain metric (1-5, 1 is the lowest, 5 is the highest)

Metrics	Grade
Line of Code	
Comment_lines_density	
Complexity_Per_Class	
Test_success_density	
Coverage	

- Estimate the current situation of the project, from which aspect, we can do some improvement

Figure 4.11.: Designed interview questions

unforeseen questions will flexibly be put forwards according to the response from the interviewees.

What's more, for RQ1, we expect to let developers give the evaluation of the correlation by themselves as much as possible. It is hard for them to cover every bug patterns, metrics because of limited experience. In the local data collection system, we have already got three matrixes for either approach. We look into them as well. In some case, the developer will have doubts about the computed value, e.g. why one correlation drops between two versions. We check the specific information in the UI of our system and review the code. The developer could express his conjecture of the change.

From the perspective of the interviewees, they might tend to give answers out of the scope we expected. Then, we should cut off them politely when the talking topics have wandered too far.

c) Summary

At the end of the interview, we give the feedback to the developer about self-understanding of the central theme.

3. Post-mortem processing

After each interview, we need to deal with lots of information in time. We retrospect the record and make the transcript. The transcript aids in sorting the key points. Every 45-minute conversation content will be converted to a couple of sheets of A4 papers. It is insignificant to neglect the written skills here since it makes no sense to make many efforts to consider how to organize plenty of oral expressions in a precise way.

Then, we go to the most important step '**coding**'[27] to extract variables in quantitative values from the interview notes. Coding produces more precise quantitative date. In principal, it is restricted to simple, objective, straightforward information.

4.2.2. Second-round Interview

We also invite the developers, who are involved at the first time. The whole procedure is same as what we have done in the first-round interview, but the emphasis of

questions raised are quite different.

We come straight to the point in the second-round interview. After first-round interviews, we collect the viewpoints from all the interviewees and utilize the method of 'Coding' to make a summary in the form of graph. The graph is used to show the developer not only how his/her opinions are interpreted by the coder, but also what the other developers have responded to the same question. 'Coding' results are in essence subjective, because it is still done manually. Imprecise expressions and comprehension may cause incorrect codes. As a result, developers may approve or reject part of results.

For two sub-questions of RQ2, we show them the proposal answers and seek for agreement.

Last, we refine our conclusion after the second-round interview.

5. Discussion - RQ1

RQ1 *Whether the diagnosis result from Findbugs has a significant correlation with special metrics?*

As mentioned in Chapter 3, there are two designed approaches to research on the correlation. For each approach, we are going to draw the conclusion based on:

Statistical results Three correlation matrixes should be gained in either approach. We can observe the correlation values directly from the local data collection system. The high Spearman's rho theoretically proves the strong correlation and vice versa. Though reviewing some source codes, we intend to analyze in a pure technique way why the computed correlation is like that.

Experts' evaluation In the interview, the evaluation of the correlation between certain bug pattern and metrics has been made by experts. A part of it may agree with the computed correlation result or may not agree with. We will analyze the reason behind the distinction.

5.1. Different versions approach

Table 5.2, Table 5.3, Table 5.4 are correlation matrixes generated from three versions of Project A. The collection period is approximately every two months. Depending on the report of the correlation computing tools, we mark out the existence of the strong relationship.

Table 5.5, Table 5.6 calculate the differences between two adjacent versions, toghther with the number of bug patterns changed in the certain pattern category.

After gainning the access to the JIRA in the company - an issue tracking platform in the agile development, we check the change log to view what issues have been done

between two adjacent versions. There are 6 types selection, Bugs, Epic, Story, Incident, Task, Sub-task, which will be labeled to any issue.

From Version A to Version B, 3 issues type of bugs, 1 issue type of story, 2 issues type of tasks for delivery are updated. From Version B to Version C, 2 issues types of bugs, 1 issues type of tasks for delivery are updated. Also combined with the commit log in remote SVN repository, we can find that the development of work in this project in the past several months is not so heavy. Most of them belongs to fix the bugs detected in the former version.

Table 5.1.: General information in three versions of Project A

Version	**Build date**	**LOC**
Version A	11th Dec 2015	29911
Version B	11th Feb 2015	30249
Version C	4th Apr 2015	30654

5.1.1. Data analysis

Table 5.2.: Correlation coefficients ρ - Project A - Version A

		Findbugs bug pattern category					
		Correcteness	Bad practice	MT correctness	Dodgy	PERF	MC vulnerability
metrics	LOC	0,525	NaN	1	0,694	0,131	0,244
	CLD	- 0,309	NaN	1	0,283	0,219	0,224
	CPLX	0,617	NaN	1	0,667	0,219	- 0,487
	CVR	0,262	NaN	1	0,134	0,044	0,264
	EC	0,741	NaN	1	- 0,031	- 0,176	- 0,052

Table 5.3.: Correlation coefficients ρ - Project A - Version B

		Findbugs bug pattern category					
		Correcteness	Bad practice	MT correctness	Dodgy	PERF	MC vulnerability
metrics	LOC	0.442	0.137	1	0,666	0.001	0,244
	CLD	- 0.626	- 0.411	1	0.130	- 0.101	0,224
	CPLX	0.286	- 0.138	1	0,620	- 0.051	- 0,487
	CVR	-0.211	- 0.358	1	0,057	- 0.154	0,264
	EC	0.038	- 0.420	1	- 0.126	- 0.356	- 0,052

Table 5.4.: Correlation coefficients ρ - Project A - Version C

		Findbugs bug pattern category					
		Correcteness	Bad practice	MT correctness	Dodgy	PERF	MC vulnerability
metrics	LOC	0.362	0.204	1	0,666	- 0.045	0,244
	CLD	- 0.615	- 0.408	1	0.130	- 0.091	0,224
	CPLX	0.287	- 0.204	1	0,620	0.000	- 0,487
	CVR	- 0.170	- 0.338	1	0,026	- 0.161	0,264
	EC	0.018	- 0.424	1	- 0.127	- 0.364	- 0,052

Table 5.5.: Correlation coefficient differences - Project A - between Version A and Version B

		Increase in correlation coefficients (%)				
	bugs	LOC	CLD	CPLX	CVR	EC
Correctness	+7	- 15.8	102.6	-53.6	-180.5	94.9
Bad practice	+3					
MT Correctness	0	0	0	0	0	0
Dodgy	+3	-4.0	-54.1	-7.0	-57.5	306
PERF	+3	-100	-146.1	-123.3	-450	102
MC Vulnerability	0	0	0	0	0	0

Table 5.6.: Correlation coefficient differences - Project A - between Version B and Version C

		Increase in correlation coefficients (%)				
	bugs	LOC	CLD	CPLX	CVR	EC
Correctness	+2	- 18.1	-2	-0.1	-19.4	-52.6
Bad practice	-2	48.9	-0.1	47.8	-5.6	-1.0
MT Correctness 0	0	0	0	0	0	0
Dodgy	0	0	0	0	-54.4	0.1
PERF	+1	460	-9.9	100	-4.5	-0.1
MC Vulnerability 0	0	0	0	0	0	0

Statistical value analysis

Following points are observed from above tables. We give them the reasons after doing some code review.

1. Some correlations are not consider to be strong, although their ρ values are relatively high.

   Many references of Spearmann's correlation defines that if the absolute value of ρ is larger than 0.5, then we draw the conclusion that this value indicates strong relationship. But in the Table 5.2, Table 5.3, Table 5.4, some correlation coefficients exceed the critical points, for example, the relationship between *CPLX* and *Correctness* in Version A, the relationship between *EC* and *Correctness* in Version A, however, they are not highlighted.

   The reason is that their accompanied P-Value, larger than chosen significance level (0,05), fails to reject the null hypothesis. To explain in an understandable way, in some case, the p-value reaches 0.25 despite the high ρ, which means that the sample data can not provide enough convincing evidence to prove the strong correlation. There is still very low possibility to draw a conclusion for such strong relation relationship, but just by chance.

2. The appearance of NaN between *Bad practice* and any metrics in Version A

   In computing, NaN stands for Not A Number. NaN is used in the floating-point standard. For example, it is caused by the calculation of dividing any number by 0. In version A, there is seven classes, containing *Bad practice* bug pattern. Moreover, all of them just have one. Based on the principle of Spearman's rho, it is not necessary to rank these bug number in order, because they all tie for first place. Consequently, the denominator of the equation appears 0 in this situation.

3. ρ values between *Multithreaded Correctness* and any metrics in all versions are equal to 1.

   Although ρ values are equal to 1, the perfect case for a strongest positive relationship, we fail to correlate *Multithreaded Correctness* and any metrics. At the time we got these numbers from UI, the output of such 'perfectness' is under suspicion. Consequently, we check the class information table in the UI of data collection system. It turns out that the sample data are too few, only two cases. One class have one *Multithreaded Correctness* bug, another has two *Multithreaded Correctness*.

Whatever the metrics value is, ρ will always be equal to 1 or -1 because there are only two ranking possibilities totally. Moreover, also, no more *Multithreaded Correctness* bug patterns were created in the latter versions, so we gain the illusion of stable strong correlation.

We will not accept this column data as the evidence to derive any result.

4. Significant change in correlation between *CPLX* and *Correctness*, between *EC* and *Correctness* from Version A to Version B

   To figure out the reason, we reviewed code. In Version B, two classes with seven *Correctness* bugs are added. Now, there are 8 classes with *Correctness* bugs. After observation, the business logic is very simple both classes. The bugs number rank 2nd and 3rd respectively among 8 classes, however their rank in complexity 7th and 8th respectively among 8 classes, same as ranking in efferent coupling metric. Thus it strongly contradicts the result given in version A, which shows the strong *positive* correlation between *Correctness* and *CPLX*, between *Correctness* and *EC*.

   Tracing the incentive of Correctness bugs, same bugs '(Possible) Null pointer dereference' are shown. The developer deployed the static class System.out-/System.err in many places in convenient of showing a testing message in local console. This violates the Findbugs defined the rule.

5. Strong correlation between LOC/CPLX and Doddy in all versions. The relationship is stable with the development of the projects. 14 classes are taken as samples, so we can say the number of samples is enough to be used in calculating the statistical results.

   After reviewing those files, the truth is same as the correlation values indicate that the larger, the more complex class is, the more *Doddy Code* it contains.

6. More stable relationship in Version B and Version C

   Comparing Table 5.5 and Table 5.6, we find that the difference between Version B and Version C is not obvious than between Version A and Version B. In reality, concluded from Table 5.1, the size of a project increases linearly. Nevertheless, the number of bugs in each category seems to change irregularly.

7. The correlation between *MC vulnerability* and any metrics keeps the same in these three versions.

This phenomenon shows that the not a new *MC vulnerability* bug pattern are produced and also not an existing *MC vulnerability* bug pattern is fixed. Actually, in the Findbugs definition, this category occupies small portion in all. We infer that the chance of generating such bug pattern is little.

Experts' evaluation

In the interview, we first showed the developers above three tables. We looked into how the correlation changes during the development process. They gave their evaluation of correlation results in the different versions approach.

1. The correlation relationship should be stable.

   In these three version, there is no particular task of improving the code quality by referring to a warning given by Findbugs. So, not so many bug patterns have been fixed. As a result, few codes have been refactored. Thus, they make an estimation that the correlation values should vary slightly.

   As a matter of fact, the reason of causing the instability is that new classes with bad codes are added based on the last version. The relationship between the number of bug patterns and metrics in the new-added class are not in accordance with the previous correlation result. For example, developers are surprised with the huge change in the correlation value between *EC* and *Correctness*.

We think that with the development of the software, it is difficult for experts to predict how many bug patterns, an added class with certain attribute (i.e., metric) will produce.

5.1.2. Restrictions

The result of this approach is hindered by the limitation of resources and uncontrollable factors.

First, as the background described above, there are not so many development issues in these three versions because of few new requirements of this project temporarily. So, the data analysis result can not reflect the normal developing situations well.

Second, in the first proposal of study design, we should track the developing process

for three months. After every data collection, we report to the developers and discuss how to deal with the Findbugs warning in the next version, then observe changes in the metrics and changes in correlation value. Unfortunately, this design is given up, in consideration of the team's willingness to spend more time and budget in optional and research-based issues, which may also have an impact on routines since it relates to the code submission to remote SVN, the build in Jenkins and SonarQube. The criteria to implement this design is not mature.

5.2. Different projects approach

5.2.1. Data analysis

Table 5.7, Table 5.8, Table 5.9 are correlation matrixes generated from three projects on the same collection date. The highlighted cells presents the strong correlation after calculating Spearman's rho.

Table 5.7.: Correlation coefficients ρ - Project A

		Findbugs bug pattern category					
		Correcteness	Bad practice	MT correctness	Doddy	PERF	MC vulnerability
metrics	LOC	0,525	NaN	1	0,694	0,131	0,244
	CLD	- 0,309	NaN	1	0,283	0,219	0,224
	CPLX	0,617	NaN	1	0,667	0,219	- 0,487
	CVR	0,262	NaN	1	0,134	0,044	0,264
	EC	0,741	NaN	1	- 0,031	- 0,176	- 0,052

Table 5.8.: Correlation coefficients ρ - Project B

		Findbugs bug pattern category					
		Correcteness	Bad practice	MT correctness	Dodgy	PERF	MC vulnerability
metrics	LOC	- 0,289	0,007	/	0,246	0,756	0,052
	CLD	- 0,000	0,081	/	- 0,382	- 0,630	0,300
	CPLX	- 0,289	- 0,136	/	0,296	0,756	- 0,078
	CVR	- 0,645	- 0,032	/	0,081	- 0,433	0,156
	EC	- 0,889	0,247	/	0,088	0,805	-0,074

Table 5.9.: Correlation coefficients ρ - Project C

		Findbugs bug pattern category					
		Correcteness	Bad practice	MT correctness	Dodgy	PERF	MC vulnerability
metrics	LOC	0,406	0,466	0,616	0,130	0,206	0,365
	CLD	0,072	0,176	- 0,154	-0,216	- 0,084	- 0,023
	CPLX	0,444	0,431	0,763	0,013	0,249	0,093
	CVR	0,240	- 0,394	0,000	- 0,052	0,155	0,119
	EC	0,192	0,201	- 0,079	0,186	0,084	0,336

Statistical value analysis

1. *Malicious Code Vulnerability* does not correlate with every metric in all projects.

   In this category, only two kinds of bugs have been detected, 'May suppose internal representation by returning a reference to mutable objects' and 'May suppose internal representation by incorporating a reference to mutable objects'. These kinds of bugs happen, when we give the object reference to a new object instance, if the original object is modified, the new instance is modified as well with it unintentionally[31]. Following code snippet 5.1 shows this problem.

```
Children child = new Children();
Calendar  date = Calendar.getInstance();

child.setBrithday(date);

// also change the child birthday
date.add(Calendar.DATE, 1);
```

Figure 5.1.: Code example of returning reference to mutable objects

   After searching for classes, containing such bugs, we find out almost of them belong to DTO files. DTO stands for Data Transfer Object programming pattern. The usage of DTO classes is to encapsulate the business data in order to easy exchange among different tiers[32] . The structure of such class is clear, i.e. variables and setter and getter' methods to modify and check the value of variables. So, it is likely that bugs in pattern of malicious vulnerability are generated in DTO because of 'setting' some variables when the variable field is an object rather than primitive data type such as int, char, boolean.

We conclude that this bug pattern does not have strong correlation with any metric for the following reasons. (1) Such bug occurrence largely depends on how many variables in object type are used, then no matter with *LOC* of the source file. (2) The comments of setting and getting methods are simple, automatically generated by Eclipse. The number of comment lines for methods is related with the number of the variables. So, *CLD* is relatively even. (3) CPLX in DTO usually equals to 0. Because we only write the accessors (Getters) and the mutators (Setters) instead of functions. (4) In case of EC metrics, it is hard to verify, because if the variable type is in object type, it may either be from JRE library, e.g. 'java.util.Date' or other classes in other packages. It is an unstable factor.

2. Stronger correlation between *Performance* and every metric in Project B than in other projects

   We find that, in those class files that this phenomenon is largely related with project property. Compared to other two projects, this project needs many manipulations of the complicated data structure. There is a special component in Project B, which is used to visualize the datasets stored in the database. First, due to the application of Java Persistence API technique, the entries in the database are mapped into the Java objects and then the further transformation will be done . Normally, we import a JS library to generate attractive graphs. So, such library has the requirement on the data type. Thus, *Performance* bugs are likely to happen when handling with different data types, e.g. initialization of the static Integer, Double objects or application of the Hashmap data structure.

3. All of the bug pattern categories have no correlation with *CLD* in all projects.

   This observation result infers that adding comments in the file whatever more or less will not prevent the appearance of bad codes, even though it helps to make the intention of classes, methods, fields clear.

4. Stronger correlation between *Dodgy* and *LOC* in Project A than in other projects

   After reviewing the code, 'uncheck/unconfirmed casting' bugs occupy large part in Dodgy. In all three projects, they contain a common component – JAXB service, which can make it easier to access XML files from Java programs. Some classes are to map response/request in form of XML into Java object via public API. Normally, these APIs accept parameter objects or return objects in generic type. When we need to cast concrete Java object into generic type or cast generic type

into a concrete Java object, you must confirm if an object is an instance of an object type. In Project A, due to many requirements of such mappings, developers will put similar methods in one class file and program them in the similar coding snippets. As result, it is very likely to cause the same bug frequently in one class.

If a class has many bugs (abnormal value) in the same pattern, for example, the class is a data access object that accesses the operation interface to database, then the functions in the class are coded in the same way, it is usual that the developer makes the same mistake everywhere.

5. Stronger correlation between *Bad practice* and *LOC*,*CPLX*,*CVR* in Project C than in other projects

   The column *Bad practice* in the matrix of Project A has been analyzed in the first approach. The reason of showing 'NaN' is that all classes have only one *Bad practice* bug, thus we are not able to rank these classes. In project B, 20 of 24 classes have one or two *Bad practice* bugs, although their metrics vary a lot. So, the correlation is relatively weaker.

Experts' evaluation

Like RQ1.1, in the interview process, we also showed the developers above three tables, they gave their estimation or judgment of correlation results in the different projects approach.

1. Any pattern category should correlate with *LOC*.

   The explanation they gave is very simple, "The larger the class file is, the more bad codes the programmers will write down possibly.". However, the result of statistic data does not totally support this viewpoint. Table 5.7, Table 5.8, Table 5.9 presents that the strong correlation occurring between LOC with any metric cannot be generalized, but the values show the high possibility.

   Correctness bug pattern is the biggest category among the all[16]. The rule regulations are made with most confidence of the improper codes. Unlike the bug patterns in other categories, the appearance of Correctness bugs does not restrict to a certain type of the class file. They are common bad/improper usage of basic Java packages, e.g. all the types (String, Integer, Array, etc,.) in java.long.object. However, the probability of the occurrence of some bug patterns depends on the

properties of the project. If there exists more data management in the database, the issue of transient and serialization in Bad practice may happen. Alternatively, if the project contains UI components with Date selector, the static issue belonging to MT Correctness may happen in DTO class and controller in MVC architecture.

Another point, programmers tend to copy the snippet codes in the same class file or in different files if they want to implement same logic. Then, bad code will be copied as well. So, from our data gathering system, we can check out that not a few classes have more than 4 bugs in the same category, even the size of them is not so large. The origin is that the same bug pattern is detected out repeatedly.

It is a misunderstanding that the bugs in the small files must be few. So, when developers review the code, they should not skip some classes because of their size as a matter of course, but make a judgment on the file type at first.

2. *Dodgy Code* should correlate with *CPLX*

   Referring to the definition of the bug pattern in table 2.1, *Dodgy Code* are concerned with the error-prone codes. However, for the majority, this description is still very confusing. According to developers, only two or three bug patterns in this category are relatively remarkable in practice, because they have experience that such bug patterns are very likely to happen during the programming.

   We find that developers are more familiar with casting issues and control flow issues in category *Dodgy Code*. Then, from their perspective, *Dodgy Code* may occur in those methods in scenarios of upcasting/downcasting, which is required if we implement inheritance, e.g. for specifying the thrown exception type or for converting a superclass of file processing into a particular subclass (i.e.AddFile, ModifyFile, PurgeFile) under some criteria. Normally, such class files are used to implement business logic, resulting in the high complexity. This statement is in accordance with the third point we have put forwards in the above section.

3. There may be weaker correlation between *CVR* and some bug patterns in the project, which includes front-end component.

   The front-end component must include some classes files that are used for transmitting data between the presentation layer and business logic layer. During development, it is not necessary and not easy to write the test cases for such classes. Then, that the coverage percentage of them equals to 0%, which they think may

disturb the correlation result.

The three matrixes almost meet with the manual estimation, especially in Project C.

4. *Correctness* and *Dodgy Code* may correlate with *EC*

   The coupling number tells the interdependence[33] between different modules. Developers give a conjecture that more type of classes in other package are imported, which may predict the fault-prone due to incorrect controlling access, reference/dereference defined by above two categories.

   However, the correlation values cannot entirely support their judgments. The correlation result only shows unclear *Correctness*'s relationship with *EC*. After further code reviews of the sample classes, the reason why the actual value does not achieve empirical expectation is that 90% classes have one or two *Dodgy Code* bugs, although their *EC* will vary from 0 to 23 in uneven distribution. The fact data fails to provide evidence that if the class more depends on externalities, it will produce more bad codes.

5. Developers have relatively fewer knowledge on *Performance* and *MC Vulnerability*

   These two categories make up a small portion of the total bug patterns, containing 27, 15 rules separately in 400. In general, their bugs are more dedicated in limited fields that could be counted on the fingers. Even so, we have observed the total number of bugs detected under these categories are not less than other large categories.

   Although they are easily neglected because of the minority, the importance is equivalent.

5.2.2. Restrictions

In this approach, we do not exclude any abnormal value. For example, in Project B, the correlation between *Correctness* and any metrics are all negative, which also has aroused developers' interest. After reviewing the code, we find that among five sample classes, one 27-line classes with 3 *Correctness* bugs messes up the Spearman Ranking correlation. This case may also exist somewhere else but is hindered by seemingly innocuous data samples. Although the appearance of such class is a small probability

event, it may largely influence the value of some correlation, further influences our judgment. It is better to exclude the distinctive data sample in the calculation. However, then, We also meet with another difficulty about to how to select appropriate samples.

5.3. Summary

For RQ1, we have applied two approaches to analyze the correlation between Findbugs warning categories with common software metrics.

First, we tracked three versions of a project in the past six months to see whether the correlation will change during the development. We find that in this selected industrial case, in the situation of not heavy development tasks, 80% relationships still keep stable. This result is agreed with by both the data in correlation matrixes and expert judgment.

Second, through the comparison of correlation values among three projects, the result shows that most of the correlation cannot be generalized due to various system architectures. Accompanied with the empirical investigation from developers in the project team, their given judgment partly is not consistent with the statistical measurement (i.e., calculating Spearman's rho). The bias is caused by (1) incomplete knowledge of bug pattern definition (2) inadequate consideration of the influence by the system composition.

However, it is still reasonable to accept both subjective and objective results, because they reveal the truth but one from the aspect of the statistical calculation, another from the aspect of empiricism in practice.

6. Discussion - RQ2

6.1. Coding of the interview notes

Figure 6.1 visualizes the results of the coding. In the interview, several aspects concerned with maintainability are concentrated on. The second level codes, `Judgment`, `Implementation period`, `Activities`, `Correlations with metrics` composes the top level `Maintainability`.

`Judgment` aggregates the codes about how the expert judges whether the project is in a maintainable status. Thus, we use codes to make a summary of properties that a maintainable project should have. The sentence like 'In a long term, we are interested in the software architecture is developed in a proper way' is coded as `Good architecture`. A viewpoint 'I would like to make software configurable that I can change the software at runtime, or restart the server, but without any change in the code' said by one developer, is coded as `Configurability`. The java doc issue was repeatedly mentioned among all the developers. They expressed the same meaning but in different words, 'when we take over an old project, if there is no java doc or quotation, I have no idea what this class or class is used for in a short time' or 'Comments helps to quickly familiar with the new code'. We code this issue as `Readability`. `Few bugs` stands for the opinion that if there are many failures in the runtime caused by bugs, the software must not be maintainable. The code `Shared components` related with the section, 'We are persisting the maintenance for many projects currently, that we put the source code for a component in a code base. Then we do not have to copy it everywhere'. The relevant viewpoints are also mentioned frequently. For example, another developer said, it is helpful to build a table template, and applied in many web pages.

`Activities` aggregates the codes about what activities the expert will do to improve the maintainability. The code `Document supplements` is used to summarize the sentence 'I will add more comments or documents for the further development.'. The notes have the statements such like 'I will add more testing to make it maintainable',

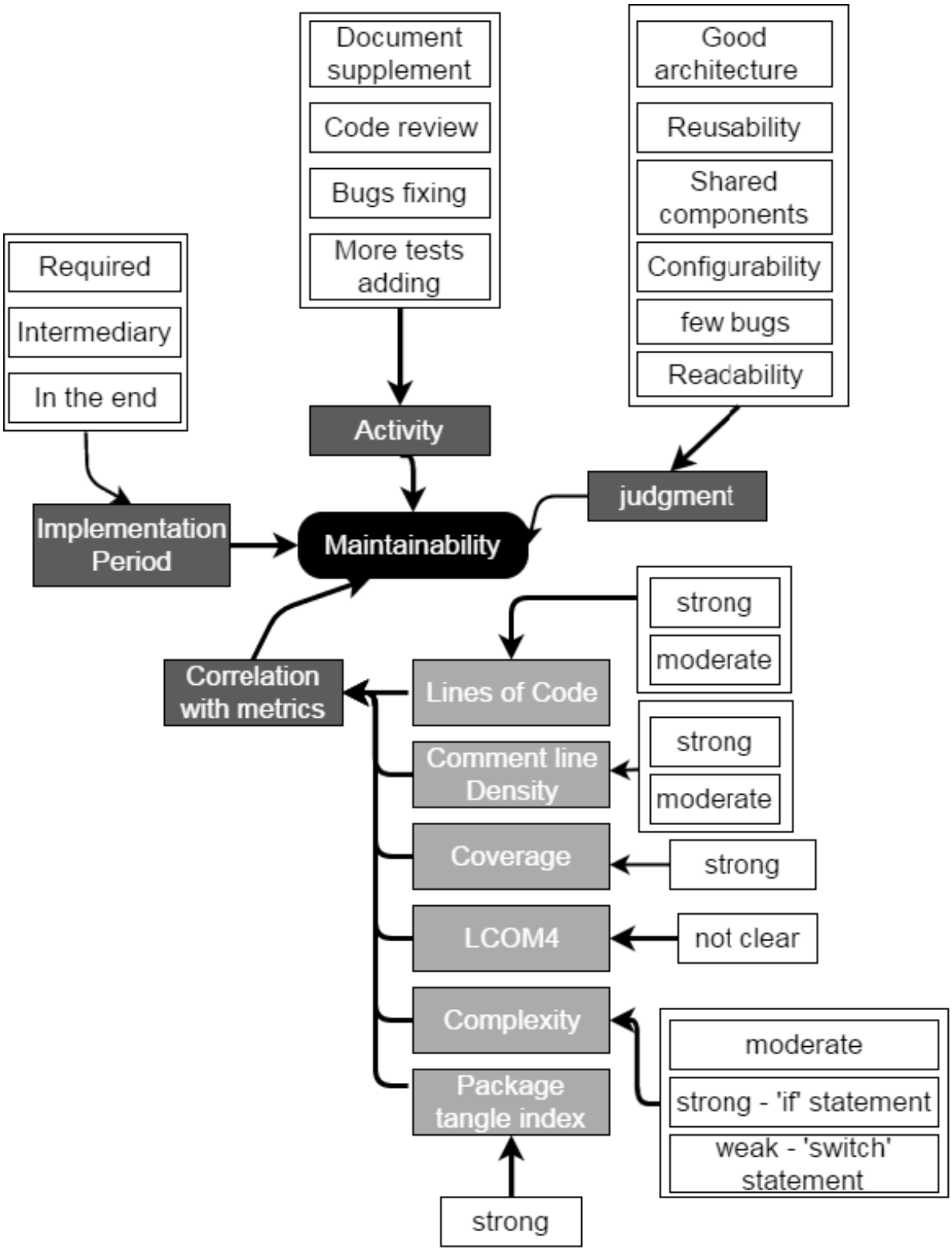


Figure 6.1.: Coding result of maintainability

then we code it as `More testing`. Moreover, next, `Bug fixing` highlights the sentence 'For me, bug fixing is an importance issue to implement maintainability.'. However, this code is relatively controversial. There is another point saying that even if you fix all the bugs, the quality may still be very poor. These two opinions do not conflict with each other because bug fixing can help to improve to some extent from the aspect of defects caused by programming styles, but cannot ensure it will help in other logical factors. One phenomenon shows that developers mention that through examining the code whatever by him/herself or the fellows in the same team, they rely on the manual judgment of status of the project and make corresponding plans to improve the quality in some way. For such idea, we code it as `Code reviewing`.

`Implementation Period` aggregates the codes about how often the developer will consider maintaining the software. Some developer said that if there were not so enough budget, we would not spend much time on this issue. Thus, we interpret in such way that when the `Requirement` of maintaining the software is distributed to the development team by the cooperation partner, then the realization of maintainability as a separate task will be put on a schedule. Another sentence expressed by another developer shows that it depends on the software development process models. In the waterfall pattern, maintenance is the final step in the whole process. However, now, we implement in an agile way. It may take place in the middle. For the above points, we add codes `In the end` and `Intermediary`.

`Correlation with metrics` aggregates the codes about how the developers make the judgments on the correlation between the maintainability with given metrics. Then, its child level consists of codes, `Correlation with LOC`, `Correlation with CLD`, `Correlation with CPLX`, `Correlation with LCOM4`, `Correlation with CVR`, `Correlation with Package tangle index`, which aggregates the correlation result for each metric.

In the interview, the developers are asked to grade the correlation between the metrics and maintainability to avoid some confusing expressions such as some comparative word 'more','most' etc. We assume that the lowest grade is 1, the highest is 5. Then, we do the directly mapping between the grade number and level of correlation. Grade of 5 corresponds to the `Strong` correlation. Grade of 3,4 corresponds to the `Moderate` correlation. Grade of 1,2 is for the `Weak` correlation. Also, we would like to let developers provide the reason why they make such grading.

`Correlation with LOC` contains code `Strong` and `Moderate`. The reason for grading

LOC to the high grade is that many lines imply more functions internally. However, another viewpoint says that it depends on the scope of the projects or what the class does and I will give the medium value.

`Correlation with CLD` also contains code `Strong` and `Moderate`. Higher density means more comments are written for the classes/methods/variables. In the future, it makes other developers easier to understand what they are used for. The neutral attitude raises 'Yes if there are no comment lines, it is difficult to maintain the code. However, sometimes you do not need so many texts. Reading text is time-consuming'.

`Correlation with CPLX` covers all the level of correlation, `Strong`, `Moderate` and `Weak`. The unbalanced situation is caused by developers' different depth of consideration. To speaking in a general way, the strong correlation is due to prevent the bugs/defects in the future. If we explore this issue further to the level of class, the situation will be technically divided into two parts, the usage of 'if-else' and 'switch'. Sentences are like, 'for switch statement, I will give 0, for if-else statement, I will give 5'.

`Correlation with LCOM4` only contains the code `Not clear`. Surprisingly, all the developers have no idea about this metrics, although, in many academic researches, the term 'Lack of Cohesion of Method' are often referred in the object-oriented programming. It is worth further knowing why the popularity of this metric is low in the enterprise and whether it is necessary to bring it in.

`Correlation with CVR` contains the single code as well `Strong`. The developers have lots to say anything related with testing. They left a deep impression on repeating the importance of tests many times in the interview. High test coverage means the greater extent to which that the source code in the projects has been tested under the simulation case.

`Correlation with Package tangle index` contains the code 'Strong'. The metric is not included in the set of given metrics we first prepared in the interview. It is put forwards by the software architect in particular. He uses this metric to control the project in the component-level. He thinks that the appearance of cyclic dependence is bad for the further maintenance work because if in the future, one piece of code needs to be refactored, this action will implicate other codes in another components. We must take care of the dependency relationship.

6.2. RQ 2.1

RQ 2.1 *Which metrics are suitable to indicate software maintainability?*

6.2.1. Results

In the first-round interview, the question is designed to ask interviewees the correlation between the general term - maintainability and metrics.

From the coding result, we have known that the application of maintainability in practice covers a few fields. We think it is necessary to map maintainability into different concrete pointcuts rather than a abstract concept. In order to specify the main objective, we break down it into several characteristics. In addition to the explanation of grading in the first-round, we try to associate the aspect where the developers made the evaluation with a particular characteristic. Each characteristic is composed of lower level criterias. This is how we propose a correlation assumption on metrics.

The top level is divided based on the code `Judgment`, which aggregates interviewees' opinion of from which aspects they make a judgment on the maintainability of the software.

Then in the second-round interview, we make a further discussion about the accuracy of the proposal and refine it. Figure 6.2 shows the tiered structure of the maintainability evaluation model after two-round interviews. Table 6.1 specifies the reason why the selected metric is related with one of the characteristics.

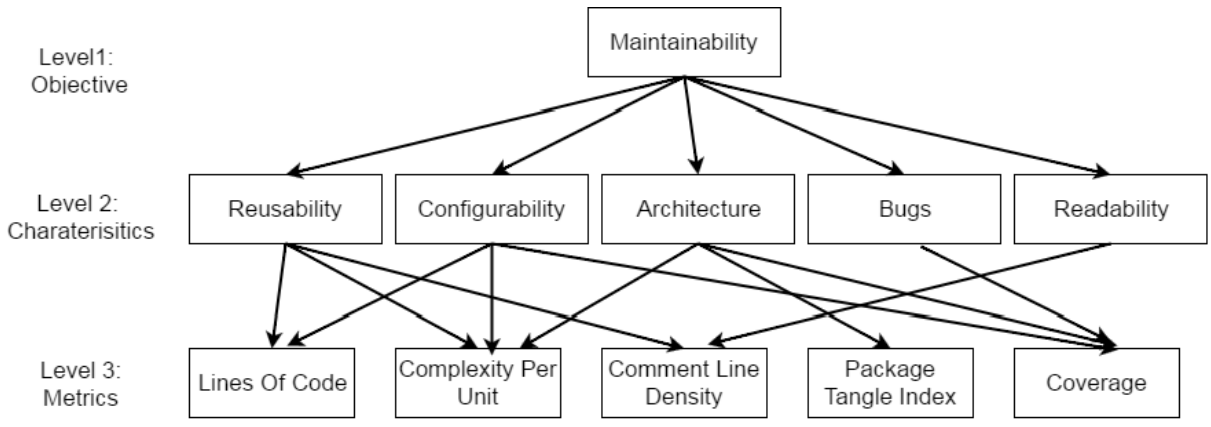


Figure 6.2.: Designed hierarchy in maintainability

6.2.2. Restrictions

Generally speaking, quantitative data analysis is not complete and more subjective. The whole 'coding' process is done by the thesis writer alone. Due to the time constraints, the coding result has not been reviewed by others. The reviewers do not necessarily have to be professionals since coding depends on the capability of the text comprehension to a large extent. In principle, it is better to assign the interview transcripts to fellows or friends. The fixation of the coding part should be performed at least two rounds to ensure objectivity.

Second point, the classification of the main subject - Maintainability is dispersive and used expressions in Level 2 (e.g. Architecture, Bugs) seem not be formalized in a standard way. Currently, we have simply summarized five characteristics from all interviewee's response about how they judge maintainability. If the more are involved, it is inappropriate to list all of them in this level. For more precise classification, another level called sub-objective could be added upon Level 2. The terms applied to the new level could be defined concerning the quality model of ISO 9216 [34].

Last point, as many papers [11, 12, 35, 10] have proposed, the multi-metric polynomial is used for measuring the numeric data of maintainability. For each involved metric, its weight of influence is considered. Here, for this question, we discuss about from which aspects the chosen metrics are related with maintainability but lack in the analysis of its impact degree.

6.3. RQ 2.2

SonarQube [17] is the main quality management platform used in the company. Developers trace the status of the project by looking at the information shown in GUI.

In the interview, we have acquired the following impression when asking RQ 2.2. First, the metrics are necessary to check regularly during the development process; Second, that regard to properties of various projects, although developers have their evaluation standard, there is no fixed value most metrics must reach except that *Test Success Density* should always be 100%. Moreover, for each project, the emphasis of the observation is partly different because of various components in the architecture.

Table 6.1.: Suitable metrics for maintainability evaluation

Characteristic	**Description**	**Related Metrics**	**Reason**
Reusability	A component created could be shared by many projects. Then we do not copy the source code anywhere. For example, in the company, a component is used for producing the unified request frame, making cross communication possible.	LOC	Decides the scope of the component, The larger it is, the more difficult it is to reuse.
		CPLX	Decides the logic inside the component. The higher values indicates more control flows, the reusability should be applied discreetly.
		CLD	Helps to understand what the class/method/fields are used for.
Configurability	Easy to be deployed in another platform without too much changes in the code	LOC	More codes may contain more internal/exteranl dependencies
		CPLX	High CPLX indicates more consideration of necessity of each flow in the new environment
		CVR	High CVR means that the product is trusty and predicts the possbility of high success rate will also happen somewhere else.
Architecture	A clear and hierarchical folder structure; Well designed Interface; In OO programs, implementation in inheritance and polymorphism	CPLX	High CPLX may result in more errors in complicated logics and this metrics decides corresponding testing amount.
		CVR	Low CVR causes more chances of bugs in the codes. Then the system is easy to break down.
		Package Tangle Index	The cyclical dependencies among packages should be avioded. Otherwise, the layers in the architecture becomes fuzzy
Bugs	Less system failures caused by bad codes	CVR	High CVR means more methods are tested. Bugs could be prevented earlier in the test cases.
Readability	Fast familiar with the project by reading the comments/docs rather than reviewing the code	CLD	High CLD means more texts are written with the purpose of making code easier to understand.

6.3.1. Results

RQ 2.2 *Whether the importance of some metrics, regardless of the quality aspect, is same among different projects?"*

Following texts are concerned with the concrete analysis of RQ 2.2. We select the metrics in the scope of project that developers usually keep an eye on.

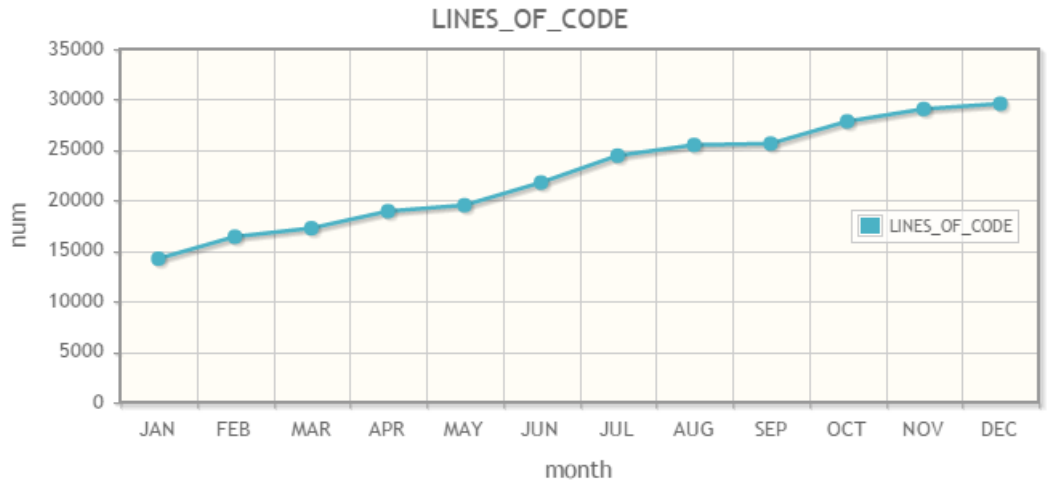
Lines of code Yes. This metric determines the scale of the project. Many views define whether the project is the small/medium/large one by measuring the project size.

What's more, when a project is under the regular development process, e.g. bi-weekly sprint period, *LOC* of which should grow with stability. Figure 6.3a is an example of this case. The size of Project A is nearly doubled in the last year. However, for those projects, if there is not any new requirement but some maintenance work, such as bug fixing, increasing the process speed, *LOC* of them is relatively stable. We can infer from Figure 6.3b, lots of tasks in Project B concentrates in the first half of 2015, but seems to stop new implementations since August.

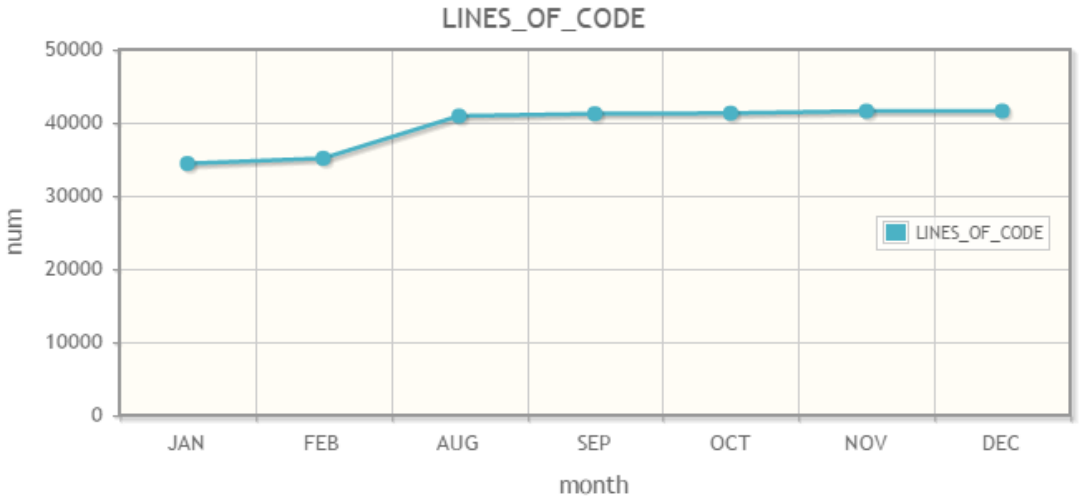
LOC is a good indicator of the scope of project and development trend not matter for what kind of project.

Comment line density Yes. *CLD* reflects readability of the project as we have analyzed in RQ2.1. The code complemented with some amount of comments whatever in the Java classes or in the configuration files, or in HTML/CSS files are all helpful to make the project understandable.

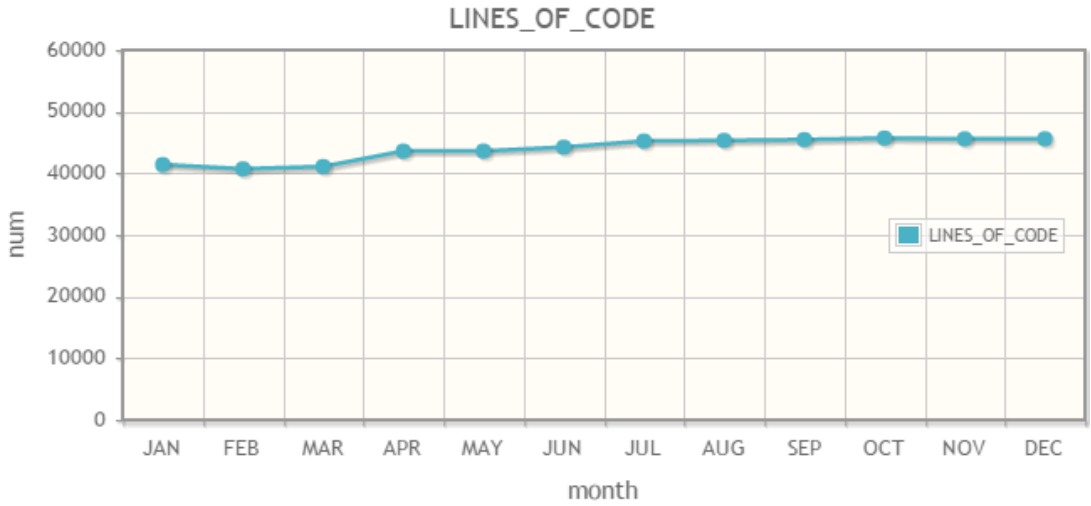
Coverage No. When we referred to this metric, the developers responded with 'Well, it depends...' at the beginning. As we see in the Figure 6.4, this metric differentiates a lot from about 15% to 62% among three projects. When we raised the doubt about the low value in Project C, the developers have thought that this value is still reasonable and acceptable here, *CVG* is meaningless if the project has many UI implementation. Many web components will cause more data encapsulation by using accessors and mutattors also some input validation in the controller class file in the MVC architectural pattern. So, it is hard to estimate how it will affect the test coverage in the whole project. However, for the pure backend application, *CVG* is an unneglectable indicator. Higher value is expected. So, for



(a) LOC - Project A

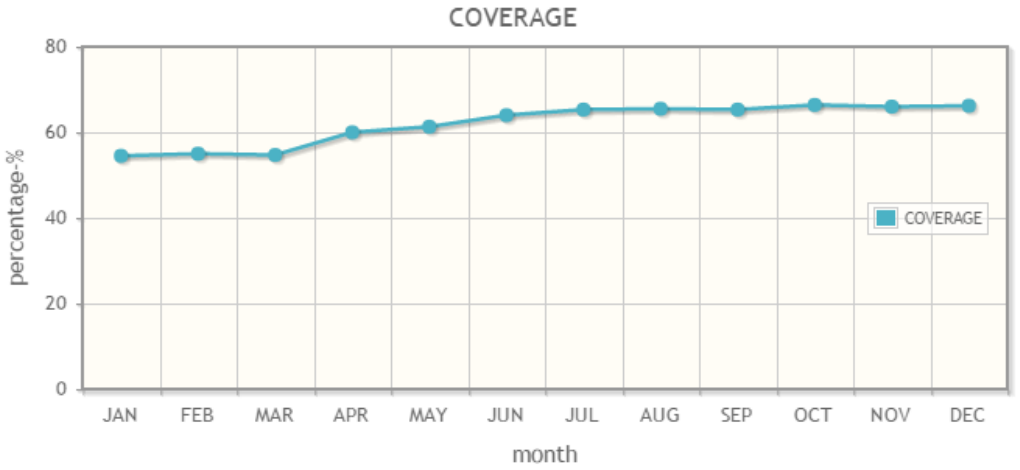


(b) LOC - Project B

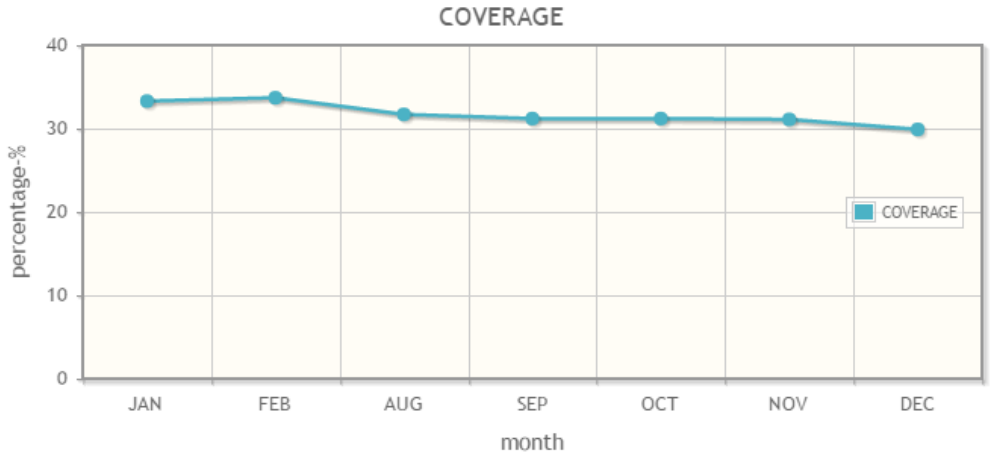


(c) LOC - Project C

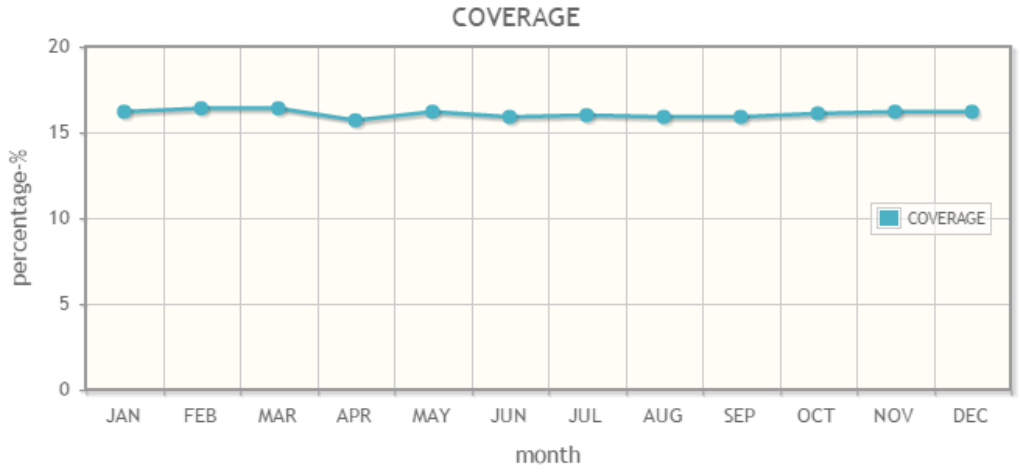
Figure 6.3.: Projects' LOC change in 2015



(a) CVR - Project A



(b) CVR - Project B



(c) CVR - Project C

Figure 6.4.: Projects' CVR change in 2015

this metric, developers' concern level differs among types of projects.

Complexity Per File No. As we did the coding shown in Figure 6.1, the evaluation of this metric is controversial. The count of the independent path is not totally acknowledged in the case of 'switch' statement. If a programmer has usual practice to apply this control flow, then the general average of the complexity of the whole project will be bound to be higher. On the other hand, some classes, e.g. the type of DTO, UI controller, or service invoking, containing lots of 'simple 'methods can not also reflect the inner complexity level, although *CPLX* values are still high.

Rules compliance Yes. This percentage shows how the codes in the project comply with recommended programming rules. The value is expected to be as high as possible in each project. If this metric is not so high, developers will consider to take actions to improve it. In SonarQube, the BLOCKER bug pattern with the highest warning priority should not occur in any situation. One developer mentioned that the goal of *Rules compliance* is more than 80% approximately. All of three projects exceed this 'lower bound', especially two of them almost approach 96%.

6.3.2. Restrictions

The same restriction mentioned in RQ2.1, the 'coding' result presents the coder's summary on interview transcripts. Second, we have chosen three studied projects that do not cover a plenty of architecture examples. So, the analysis of the metrics' importance is based on interviewees' perspective on the projects they are working with.

6.4. Summary

In this chapter, we mainly talk about how to utilize metrics to evaluate the maintainability of the project. In the interview with the developers, they have shared their viewpoints from different aspects of maintainability, e.g. the definition, the maintenance activities, judgments. According to the coded word, we split maintainability into several characteristics and then correlate each of them with suitable metrics. For RQ2.2, we select often-observed metrics in the project-level measured by SonarQube.

We find that the importance of *CVR* and *CPLX Per File* are not same for developers among different types of projects.

7. Conclusion and Outlook

The objective behind this study is to explore topics of the evaluation of the source code quality. In the company, except for the direct review the code, developers also make use of the static code analysis tool to get comprehensive knowledge about the objective facts of the project and therefore make the judgment of the quality on the basis of the analysis report. Here, we particularly use the term 'metric' to present a standard measurement of these internal source code facts. This thesis helps to make a deeper investigation on what kind of information the static code analysis tool accompanied could reveal.

Both the quantitative and qualitative methods are applied to solve two research questions. First, we set up an automatic collection system to gather, process metrics and bugs information, which are stored in the remote SonarQube database. This system shows us a visualization analysis of bug patterns distribution, metric tendency, etc in the chosen projects and provide the statistical calculation result. Second, the interviews with developers have been held to collect their experience of applying the code analysis tools in the working environment. The 'coding' method is used to interpret the interview transcript.

Chapter 1 first introduced a popular research field currently of studying three main questions 'what','how','where' concerned with software quality. And we pointed out that we restricted the research orientation into source code quality. Among approaches in evaluating code quality, we were motivated to make a research on the relationship among usage of the static code analysis tool, metrics, expert judgments. Two research questions were put forwards in order to be solved in the thesis.

Chapter 2 gave the detailed description of Findbugs, SonarQube, the analysis tools used in the later chapters and gave the proper definition of maintainability and several metrics in regard to how we measure the code with a standard rule.

Chapter 3 discussed about the study designs separately. For RQ1, we would like to compare the correlation matrixes from two dimensions to see whether the correlation

between the occurance of bugs and code metrics will vary in the long development period and will vary among different projects. And we also expected to see how experts will think about what the reports given by Findbugs, SonarQube implicate. For RQ2, we would use 'coding' method to convert the interview notes into the simplified codes.

Chapter 4 presented how we built up an automatic data collection system to gather quantitative information, such as bugs, metric data and how we conducted the interview to gather qualitative information, which is the developers' experience in judging the quality.

Chapter 5 gave the discussion about RQ1. Through the calculation of exact correlation values between code metrics and the occurrence of bugs classified in Findbugs categories, we find that, first, the most correlation keeps stable in the Project A in the past three months of development duration as manual judgments from the developers involved in the team. Second, the most correlation is hard to be generalized among projects because of various system architectures. However, in the interviews, developers gives some contradictory opinions. The main reason is that although the practical experience gives them more sharp judgments of how the correlation should be like, it is impossible for them to take all of the cases in Findbugs into consideration. They normally focus on the most frequent cases instead of all. We can not say, between the subjective result and the objective result, which one is more acceptable. The objective result lacks flexibility and practicality, and another one is too subjective, which can not be supported by powerful evidence.

Chapter 6 gave the discussion about RQ2. After doing 'coding' of interviews' notes, we have gained a comprehensive knowledge about maintainability from its definition, activities, judgment, correlation with metrics. We resolve maintainability into five characteristics for the sack for the diverse application of the maintenance work. We relate each of them with sets of applied metrics which could be often observed by developers in SonarQube. We have also concluded that the importance of metric *CVR* and *CPLX per file* varies among projects.

Several issues are raised for the further research work.

In this thesis, we classify the bugs pattern by its property into a defined category. In the interview, we got the usage feedback from developers that they normally fix the bugs according to the level of severity. In SonarQube portal, there is a widget about the summary of potential bugs, showing the total number of each level. We can further

investigate the correlation between bugs in the category of severity and code metrics. If we are able to conclude what kinds of class file are easy to incur possible threatening bugs, we can prevent its appearance during development and review phase.

Furthermore, some metrics we have applied are considered to be out of date in the evaluation of maintainability [36]. In the future, we can explore new metrics and verify its feasibility.

Appendices

A. Table creation script in local database

```
-- Table BUGS_ENTRY
CREATE TABLE BUGS_ENTRY (
    ID int NOT NULL AUTO_INCREMENT,
    REFER_BUG_ENTRY_ID int NOT NULL,
    REFER_SNAPSHOT_ID int NOT NULL,
    RULE_ID int NOT NULL,
    PATTERN_NAME varchar(200) NOT NULL,
    PATTERN_CATEGORY varchar(50) NOT NULL,
    FAILURE_LEVEL int NOT NULL,
    CONSTRAINT BUGS_ENTRY_pk PRIMARY KEY (ID)
);
```

```
-- Table BUGS_PATTERN_ENTRY
CREATE TABLE BUGS_PATTERN_ENTRY (
    ID int NOT NULL AUTO_INCREMENT,
    PATTERN_NAME varchar(200) NULL,
    RULE_KEY varchar(200) NULL,
    SEVERITY int NULL,
    PLUGIN_NAME varchar(20) NULL,
    REFER_BUG_PATTERN_ID int NULL,
    CONSTRAINT BUGS_PATTERN_ENTRY_pk PRIMARY KEY (ID)
);
```

```
-- Table CLASS_ENTRY
CREATE TABLE CLASS_ENTRY (
    ID int NOT NULL AUTO_INCREMENT,
    REFER_CLASS_ID int NOT NULL,
    CLASS_NAME varchar(200) NULL,
```

```
    REFER_PROJECT_SNAPSHOT_ID int  NOT NULL,
    REFER_ROOT_PROJECT_ID int  NOT NULL,
    REFER_SNAPSHOT_ID int  NOT NULL,
    CONSTRAINT CLASS_ENTRY_pk PRIMARY KEY (ID)
);
```

```
-- Table PROJECTS
CREATE TABLE PROJECTS (
    ID int  NOT NULL AUTO_INCREMENT,
    PROJECT_NAME varchar(100)  NOT NULL,
    CREATED_AT date  NULL,
    REFER_PROJECT_ID int  NOT NULL,
    REFER_SNAPSHOT_ID int  NOT NULL,
    CONSTRAINT PROJECTS_pk PRIMARY KEY (ID)
);
```

```
CREATE TABLE STATISTICAL_METRICS (
    ID int  NOT NULL AUTO_INCREMENT,
    VALUE decimal(30,20)  NULL,
    REFER_MEASURE_ID bigint  NOT NULL,
    REFER_SNAPSHOT_ID bigint  NOT NULL,
    REFER_METRIC_ID bigint  NOT NULL,
    CONSTRAINT STATISTICAL_METRICS_pk PRIMARY KEY (ID)
);
```

```
create view view_swr_bugs AS
(SELECT swr.REFER_CLASS_ID,
                bugs.REFER_SNAPSHOT_ID,
                bugs.PATTERN_CATEGORY,
                count(*) AS NUM
                from (
                select REFER_SNAPSHOT_ID,REFER_CLASS_ID from
                   class_entry where REFER_ROOT_PROJECT_ID = '
                   11860'
                ) as swr, bugs_entry as bugs
                where swr.REFER_SNAPSHOT_ID = bugs.
                   REFER_SNAPSHOT_ID GROUP BY bugs.
                   REFER_SNAPSHOT_ID, bugs.PATTERN_CATEGORY);
```

```
create view view_vponline_bugs AS
(SELECT swr.REFER_CLASS_ID,
                bugs.REFER_SNAPSHOT_ID,
                bugs.PATTERN_CATEGORY,
                count(*) AS NUM
                from (
                select REFER_SNAPSHOT_ID,REFER_CLASS_ID from
                   class_entry where REFER_ROOT_PROJECT_ID = '
                   8829'
                ) as vpo, bugs_entry as bugs
                where vpo.REFER_SNAPSHOT_ID = bugs.
                   REFER_SNAPSHOT_ID GROUP BY bugs.
                   REFER_SNAPSHOT_ID, bugs.PATTERN_CATEGORY);
```

```
create view view_aaservice_bugs AS
(SELECT swr.REFER_CLASS_ID,
                bugs.REFER_SNAPSHOT_ID,
                bugs.PATTERN_CATEGORY,
                count(*) AS NUM
                from (
                select REFER_SNAPSHOT_ID,REFER_CLASS_ID from
                   class_entry where REFER_ROOT_PROJECT_ID = '
                   10821'
                ) as ass, bugs_entry as bugs
                where ass.REFER_SNAPSHOT_ID = bugs.
                   REFER_SNAPSHOT_ID GROUP BY bugs.
                   REFER_SNAPSHOT_ID, bugs.PATTERN_CATEGORY);
```

Bibliography

[1] STEPHEN H, Kan: *Metrics and Models in software quality engineering*. Pearson Education, Inc., 2003

[2] PATEL, J. ; LEE, R. ; KIM, H. K.: Architectural View in Software Development Life-Cycle Practices. In: *6th IEEE/ACIS International Conference on Computer and Information Science (ICIS 2007)*, 2007, S. 194–199

[3] AYEWAH, N. ; HOVEMEYER, D. ; MORGENTHALER, J. D. ; PENIX, J. ; PUGH, W.: Using Static Analysis to Find Bugs. In: *IEEE Software* 25 (2008), Sept, Nr. 5, S. 22–29. `http://dx.doi.org/10.1109/MS.2008.130`. – DOI 10.1109/MS.2008.130. – ISSN 0740–7459

[4] *Qualitative research methods*. `http://www.ccs.neu.edu/course/is4800sp12/resources/qualmethods.pdf`

[5] CARAVALHO, Soniya ; WHITE, Howard: *Combining the quantitative and qualitative approaches to poverty measurement and analysis*. The World bank

[6] FENTON, Norman E. ; PFLEEGER, Shari L.: *Software Metrics: A Rigorous and Practical Approach*. 2nd. Boston, MA, USA : PWS Publishing Co., 1998. – ISBN 0534954251

[7] DEMARCO, Tom: *Controlling Software Projects: Management, Measurement & Estimation*. New Jersey : Yourdon Press, 1982

[8] WAGNER, S. ; DEISSENBOECK, F. ; M.AICHNER ; J.WIMMER ; M.SCHWALB: An Evaluation of Two Bug Pattern Tools for Java. In: *Software Testing, Verification, and Validation, 2008 1st International Conference on*, 2008, S. 248–257

[9] SANTOS CUNHA, Andre A.: *An Empirical Investigation of Source Code Metrics and FindBugs Warnings*, FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO, Diplomarbeit, 2010

[10] DON COLEMAN, Bruce Lowther Paul O. Dan Ash A. Dan Ash: Using Metrics to evaluate software system maintainability.

[11] KAUR, A. ; KAUR, K. ; PATHAK, K.: A proposed new model for maintainability index of open source software. In: *Reliability, Infocom Technologies and Optimization (ICRITO) (Trends and Future Directions), 2014 3rd International Conference on*, 2014, S. 1–6

[12] PAUL OMAN, Jack R. H.: Construction and testing of polynomials predicting software maintainability. In: *Journal of Systems and Software - Special issue of the best papers from the Oregon Workshop on Software Metrics* (1993)

[13] W.LI ; S, Henry: Maintenance metrics for the object oriented paradigm. In: *Software Metrics Symposium, 1993. Proceedings,. First International*, 1993, S. 52 60

[14] DENNIS KAFURA, Geereddy R. R.: The use of software complexity metrics in software Maintenance, 1987

[15] Evaluation of FindBugs - The static analysis tool that finds bugs. (2009)

[16] DAVID HOVEMEYER, William P.: Using FindBugs in Anger / York College. – Forschungsbericht

[17] *SonarQube*. `http://www.sonarqube.org/`

[18] ; Software Engineering Standards Committee of the IEEE Computer Society (Veranst.): *IEEE Standard for Software Maintenance*. 1998

[19] LIENTZ, Bennett P. ; SWASON, E. B.: *Software Maintenance Management*. Addison-Wesley Longman Publishing Co., Inc., 1980

[20] VLIET, Hans van: Software Maintenance. (2008)

[21] GLASS, Robert L.: Frequently Forgotten Fundamental Facts about Software Engineering. In: *IEEE Software* (2001), May/June

[22] MILLS, Everald E.: Software Metrics. (December 1988)

[23] *Definitions of Mc Cabe Cyclomatic complexity*. `http://www.chambers.com.au/glossary/mc_cabe_cyclomatic_complexity.php`

[24] *Efferent Coupling*. `https://en.wikipedia.org/wiki/Efferent_coupling`

[25] *J2EE - Java 2 Platform Enterprise Edition*. http://www.webopedia.com/TERM/J/J2EE.html

[26] Spearman's rank correlation. (2007), December. http://www.mei.org.uk/files/pdf/spearmanrcc.pdf

[27] SEAMAN, Carolyn B.: Qualitative Methods in Empirical Studies of Software Engineering. In: *Software Engineering, IEEE Transactions* 25 (Jul/Aug 1999), S. 557 – 572

[28] *Snapshot*. https://en.wikipedia.org/wiki/Snapshot

[29] *Java Downlaod site*. http://www.oracle.com/technetwork/java/javase/downloads/java-archive-downloads-javase7-521261.html#jdk-7u79-oth-JPR

[30] ORACLE: The Java EE 6 Tutorial. 2013. – Forschungsbericht

[31] *FindBugs Bug Descriptions*. http://findbugs.sourceforge.net/bugDescriptions.html

[32] *Data transfer object*. https://en.wikipedia.org/wiki/Data_transfer_object

[33] POSHYVANYK, D. ; MARCUS, A.: The Conceptual Coupling Metrics for Object-Oriented Systems. In: *2006 22nd IEEE International Conference on Software Maintenance*, 2006. – ISSN 1063–6773, S. 469–478

[34] HEITLAGER, I. ; KUIPERS, T. ; VISSER, J.: A Practical Model for Measuring Maintainability. In: *Quality of Information and Communications Technology, 2007. QUATIC 2007. 6th International Conference on the*, 2007, S. 30–39

[35] OMAN, Hagemeister J. P: Metrics for assessing a software system's maintainability, 1992

[36] OSTBERG, J. P. ; WAGNER, S.: On Automatically Collectable Metrics for Software Maintainability Evaluation. In: *Software Measurement and the International Conference on Software Process and Product Measurement (IWSM-MENSURA), 2014 Joint Conference of the International Workshop on*, 2014, S. 32–37

List of Figures

2.1. Seven axes of SonarQube Code analysis [17] 9
2.2. Distribution of maintenance activities 10
2.3. McCabe's cyclomatic complexity calculation example 13

3.1. Two approaches design 17

4.1. Continous integration deployment in company 24
4.2. SonarQube UI of project metrics 24
4.3. SonarQube UI of class metrics 24
4.4. Data processing step 25
4.5. SonarQube Database structure 26
4.6. Local Database structure 28
4.7. local data gathering system 31
4.8. Web UI - Project A - Project metrics in 2015 35
4.9. Web UI example - Project A - Version A - Bug Category 'CORRECTNESS' 36
4.10. Qualitative Data Collection Process 37
4.11. Designed interview questions 39

5.1. Code example of returning reference to mutable objects 50

6.1. Coding result of maintainability 58
6.2. Designed hierarchy in maintainability 61
6.3. Projects' LOC change in 2015 65
6.4. Projects' CVR change in 2015 66

List of Tables

2.1. Findbugs Category Definition[15, 16] 8

3.1. Brief projects' introduction 16
3.2. Information contained classes in one project case 17
3.3. Correlation coefficients matrix - Project * - Version * 17
3.4. table 'RULES_FAILURES' and 'RULES' 20
3.5. table 'PROJECT_MEASURES' 20

5.1. General information in three versions of Project A 44
5.2. Correlation coefficients ρ - Project A - Version A 44
5.3. Correlation coefficients ρ - Project A - Version B 44
5.4. Correlation coefficients ρ - Project A - Version C 45
5.5. Correlation coefficient differences - Project A - between Version A and Version B 45
5.6. Correlation coefficient differences - Project A - between Version B and Version C 45
5.7. Correlation coefficients ρ - Project A 49
5.8. Correlation coefficients ρ - Project B 49
5.9. Correlation coefficients ρ - Project C 50

6.1. Suitable metrics for maintainability evaluation 63

Declaration

I hereby declare that the work presented in this thesis is entirely my own and that I did not use any other sources and references than the listed ones. I have marked all direct or indirect statements from other sources contained therein as quotations. Neither this work nor significant parts of it were part of another examination procedure. I have not published this work in whole or in part before. The electronic copy is consistent with all submitted copies.

---

place, date, signature