Master of Science in Software Engineering
June 2020

An empirical assessment of the predictive quality of internal product metrics to predict software maintainability in practice

Maike Zhang
Xinhao Wu

Faculty of Computing, Blekinge Institute of Technology, 371 79 Karlskrona, Sweden

This thesis is submitted to the Faculty of Computing at Blekinge Institute of Technology in partial fulfilment of the requirements for the degree of Master of Science in Software Engineering. The thesis is equivalent to 20 weeks of full time studies.

The authors declare that they are the sole authors of this thesis and that they have not used any sources other than those listed in the bibliography and identified as references. They further declare that they have not submitted this thesis at any other institution to obtain a degree.

Contact Information:
Author(s):
Maike Zhang
E-mail: mazh18@student.bth.se

Xinhao Wu
E-mail: xiau18@student.bth.se

University advisor:
Nauman bin Ali
Department of Software Engineering

Faculty of Computing
Blekinge Institute of Technology
SE–371 79 Karlskrona, Sweden

Internet : www.bth.se
Phone : +46 455 38 50 00
Fax : +46 455 38 50 57

Abstract

Background. Maintainability of software products continues to be an area of importance and interest both for practice and research. The time used for maintenance usually exceeds 70% of the whole period of software development process. At present, there is a large number of metrics that have been suggested to indicate the maintainability of a software product. However, there is a gap in validation of proposed source code metrics and the external quality of software maintainability.
Objectives. In this thesis, we aim to catalog the proposed metrics for software maintainability. From this catalog we will validate a subset of commonly proposed maintainability indicators.
Methods. Through a literature review with a systematic search and selection approach, we collated maintainability metrics from secondary studies on software maintainability. A subset of commonly metrics identified in the literature review were validated in a retrospective study. The retrospective study used a large open source software "Elastic Search" as a case. We collected internal source code metrics and a proxy for maintainability of the system for 911 bug fixes in 14 version (11 experimental samples, 3 are verification samples) of the product.
Results. Following a systematic search and selection process, we identified 11 secondary studies on software maintainability. From these studies we identified 290 source code metrics that are claimed to be indicators of the maintainability of a software product. We used mean time to repair (MTTR) as a proxy for maintainability of a product. Our analysis reveals that for the "elasticsearch" software, the values of the four indicators LOC, CC, WMC and RFC have the strongest correlation with MTTR.

Conclusions. In this thesis, we validated a subset of commonly proposed source code metrics for predicting maintainability. The empirical validation using a popular large-scale open source system reveals that some metrics have shown a stronger correlation with a proxy for maintainability in use. This study provides important empirical evidence towards a better understanding of source code attributes and maintainability in practice. However, a single case and a retrospective study are insufficient to establish a cause effect relation. Therefore, further replications of our study design with more diverse cases can increase the confidence in the predictive ability and thus the usefulness of the proposed metrics.

Keywords: software metrics, maintainability, predict, measurements, mining software repositories, retrospective study, mean time to repair, MTTR

Acknowledgments

We want to thank the professor. Nauman bin Ali has been extremely helpful in guiding our thesis, whether it is in the thesis selection, weekly report, or timely answers to the problems, He has provided us with great help. In the course of our research, we encountered many issues that were difficult for us to deal with. He provided us with many alternative solutions and even helped us check related papers. We sincerely thank the professor for giving us enormous help.

Contents

Chapter 1

Introduction

Maintainability of software products continues to be an area of importance and interest both for practice and research. Because maintenance can account for 40% to 60% resources and usually exceeds 70% time of the whole period of software development process [18].

The importance of maintainability is well established. According to the definitions proposed in ISO standards ISO9126 and ISO25010 [54], software maintainability refers to the ability to modify software products and is composed of analyzability, changeability, stability, testability, and maintainability compliance. To be able to measure the maintainability of software, we need external maintainability metrics that should be able to measure attributes such as the behavior of the maintainer, user, or system (including the software) when maintaining or modifying the software during testing or maintenance. And Internal maintainability metrics are used for predicting the level of effort required for modifying the software product. External quality means the extent to which a product satisfies stated and implied needs when used under specified conditions.Internal quality means the totality of attributes of a product that determine its ability to satisfy stated and implied needs when used under specified conditions. According to the definition of maintainability in IEEE24765 [32], maintainability mainly refers to the changeability and modifiability of the product. For software maintainability, it covers the entire development process: how easy it is to modify the software and its components; the ability to maintain the software product; and the time required to repair the fault and the speed of correcting the program. Based on this, we classify and summarize the internal quality and external quality in some of the relevant literature collected and briefly summarize their impact on maintainability [47] [35].

At present, it is found that there are many metrics including internal metrics and external metrics for measuring maintainability, such internal metrics as Number of Code(NOC), Coupling between Object(CBO),etc[49]. Such external metrics as Mean Time To Repair(MTTR) [6], However, there is a lack of study on software metrics in terms of the how related between internal metrics and maintainability. So we will focus on this part.

The primary purpose of our paper is to study the relationship between source code metrics and maintainability, which metrics can better predict maintainability.

For the theoretical part: we first consulted the relevant literature and first gave a sufficient definition of metrics and maintainability to ensure that readers and we can better understand the meaning of the two. Second, we will introduce the relationship between metrics and maintainability according to the ISO9126 model— How to

influence each other. What's more, because there have been many papers that have shown that there are a large number of metrics related to maintainability, but there is no mention of which can better predict maintainability, so we also summarized these basic data.
For the retrospective study: Based on our collated data, we selected 8 of the most frequent metrics as our experimental variables, and found metrics that can better predict the maintainability. We found 14 iterative versions of the "elasticsearch" software on GitHub as test samples (11 experimental samples, 3 are verification samples), and calculated 911 "bug fix"'s MTTR to calculate the maintainability of the software. We also check all 2198 files that reported in the "bug fix". We get the data of metrics of all these files by CKJM and JavaDesignite. And finally, use the Pearson algorithm to compare its correlation and finally draw a conclusion.

The remainder of the thesis is structured as follows: Chapter 2 starts with the theoretical grounds for the study i.e., presenting both the definition of maintainability and also discussing it in the context of quality in software engineering. Chapter 3 presents the research questions and the methodology pursued in the thesis. Chapter 4 presents the results and their analysis, Chapter 5 discusses the results and their implication for both research and practice. Chapter 6 concludes the thesis and presents directions for future research.

Chapter 2

Background and related work

In this chapter, we present an overview of the prominent definitions of maintainability. The definitions together with a discussion of the relation between internal, external and quality in use provides the theoretical basis for our work on maintainability.

2.1 Background

2.1.1 Definition of maintainability

We found two quality standards with definitions of maintainability, which are:ISO9126 /25010 [2] and IEE24765 [32].

According to the definition of ISO9126/25010 [2], software maintainability refers to the ability to modify software products and is composed of analyzability, changeability, stability, testability, and maintainability compliance.

In IEEE24765, there is a detailed definition of maintainability [32]: 1. The difficulty of modifying the software system or component to change or add functions, correct faults or defects, improve performance or other attributes, or adapt to changing environments 2. Easily a state where a hardware system or component can be used to retain or restore it to an executable state. Its required functions 3. The ability to modify software products. Software Engineering-Software Life Cycle Process-Maintenance. 4. Average the work required to find and repair software malfunctions. 5. Speed and ease of correction procedure or change [32].

It means that maintainability mainly refers to the changeability and modifiability of a product. Software maintainability covers the entire development process: how easy it is to modify the software and its components; the ability to maintain the software product; and the time required to repair the fault and the speed of correcting the behaviour of the product.

To be able to measure the maintainability of software, we need external maintainability metrics that should be able to measure attributes such as the behavior of the maintainer, user, or system (including the software) when maintaining or modifying the software during testing or maintenance. And Internal maintainability metrics are used for predicting the level of effort required for modifying the software product. External quality means the extent to which a product satisfies stated and implied needs when used under specified conditions. Internal quality means the totality of attributes of a product that determine its ability to satisfy stated and implied needs when used under specified conditions.

2.1.2 Aspect of maintainability:

After understanding the basic definition of maintainability, we found that in the ISO25010 standard [2], it also divided the maintainability into five aspects in detail, modularity, reusability, analyzability, modifiability and testability.

Modularity	Modularity means that the ability to modify at the same time without affecting other modules, and because it is modular, you can use aggregation or coupling indicators to better find the relationship between different modules.
Reusability	Reusability refers to the usability of code for another system, and we chose to analyze a single system without analyzing the code reuse of the same software.
Analyzability	Analyzability is to analyze the impact of code modification on the system or component. Good analysis can help users better choose the corresponding position of the code that needs to be modified in the program, and better help improve maintainability.
Modifiability	Modifiability means that no new problems or bugs are introduced in the code, which is a very important part of maintainability.
Testability	Testability test whether the modified code can meet the user's needs, so for the testability test requires a separate test code module, and our choice of code is limited to the historical version information of different software in GitHub, so Unable to complete testability test.

2.1.3 Relationship between maintainability

After understanding the definition of maintainability, we need to understand what factors will affect it to analyze it in detail in this thesis, and the following three classifications of type of metrics are made in ISO9126: the internal metrics, the external metrics and the quality in use metrics.

Internal metrics are used during the development phase of the software product (for example, proposal request, requirement definition, design specification, or source code). During this period, the software cannot be used without being shaped. The internal indicators provide users with the ability to measure the quality of intermediate delivery products so that the quality of the final product can be predicted. This can help users discover quality problems and take corrective actions as early as possible in the development life cycle [2].

The external measurement is used in the testing phase or any operation phase of the software product. The external measurement mainly measures the quality of the software product by measuring the behavior of the system to which the software product belongs. In this process, the software function has been completed, or part of the function is already operable, and the measurement is performed when the software product is executed in the system environment in which it is to be run [2].

The use of quality measurement can only be used in a real system environment.

Its main measurement standard measures whether the product meets the needs of specific users under the specified use environment to meet specific target needs (such as effectiveness, productivity, safety, and satisfaction). This can only be achieved in a realistic system environment [2].

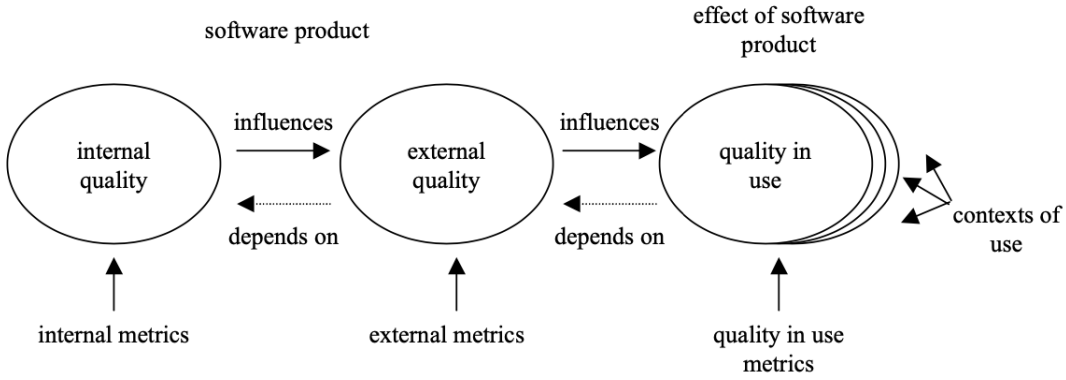

Figure 2.1: Relationship between maintainability

Through the ISO9126 definition for the type of metric, we found that the three indicators of "internal quality," "external quality," and "quality in use" are all related to each other.

First of all, "internal quality" is an indicator to measure the internal code of the software. It is used in the earliest development and design stage of software. It can measure the quality of the internal code of software very well, even in the early stage of software development. The product stage has not yet been formed. And a good "internal quality" can largely guarantee the "external quality" of the software. A good "internal quality" means that there are better software basic design and code advantages, which can improve its performance in the testing phase.

Secondly, "external quality" is an indicator to measure the intuitive feeling displayed by software to users. It is mainly used in the testing phase of the software development process. A good "external quality" will be affected by "internal quality," but it will also affect "quality in use." Because each function is a part of the final recognition software, when each part of the function is complete and meets the requirements, it will meet the needs of the user in the actual situation.

Finally, "quality in use" is to measure whether the product meets the needs of specific users in the specified use environment, mainly in the actual use, to measure the most real product situation of the software product.

Therefore, for maintainability, the best manifestation should be the "external quality" or "use quality" part, and the final evaluation standard is the key to determine the maintainability. Only after the software product enters the maintenance phase of its life cycle can we obtain the "external quality" or "use quality" metrics, so we want to study the "internal quality" metrics to find metrics that can better predict maintainability in order Improve development efficiency and product quality during the development process.

2.1.4 Relevant maintainability indicators

Find relevant external indicators that can measure maintainability in relevant metrics, so it can help us measure the maintainability of the software. We consulted relevant literature and papers and found that there are two main aspects to measure maintainability.

The first is to measure maintainability through a Dynamic measurement standard or indicator(Quality in use), such as Mean Time Between Failures(MTBF) [55], & Mean Time To Repair(MTTR) [5]. When it comes to time, it is usually expressed by three indicators: MTBF, MTTF, and MTTR. These three indicators were used early to measure the reliability of a product (especially repairable products such as appliances). The unit is "hour." They reflect the time quality of the product and are a reflection of the ability of the product to remain functional for a specified period of time. A software system is also a product in a certain sense, so it is also appropriate to use these three indicators to measure the reliability and maintainability of a software system. Among these three indicators, especially MTTR, it represents mean time to restoration. MTTR is the expected value of the recovery time of the random variable. It includes the time required to confirm the failure and the time required for maintenance. So it can measure the maintainability of the software.

The second aspect is to evaluate the maintainability with relevant internal quality attributes (Static measurement standard). ISO9126 [24] software quality model divides maintainability into 1. Analyzability: how easy it is to analyze and locate problems. 2. Changeability: the ability of a software product to enable specified modifications to be implemented. 3. Stability: Prevent accidental modification to cause program failure. 4. Testability: the ability to make modified software can be confirmed. We can measure the performance of these aspects of the software separately, and finally, draw the maintainability of the system. Immediately following are some examples:

Cyclomatic Complexity: Measures the structural complexity of the code. Maintainability Index MI: Calculates index values between 0 and 100. Represents the relative ease of maintaining code, and a high value means better maintainability.

Depth of Inheritance: The depth of inheritance tree (DIT) is a code metric that is specific to object-oriented programming. It measures the maximum length between a node and the root node in a class hierarchy.

Class Coupling: Measure coupling to parameters by parameters, local variables, return types, method calls, generic or template instantiations, base classes, interface implementations, field and property modifications defined on external types Unique class.

2.2 Related work

2.2.1 For correlation metrics

In this part, we mainly look for metrics that have been shown to be related to maintainability, because our RQ is to find which metrics can better predict the maintainability, so we have to find those metrics that have been shown to be relevant to maintainability (see Chapter 3 for details of our approach to identify the following

research).
A few of these indicators are mentioned in many papers, such as: Depth of inheritance tree (DIT) [22] [20] [15] [31] [40] [37];Weighted methods per class (WMC) [22] [15] [33] [31] [20]; Number of children (NOC) [15] [31] [37] [20];

Most metrics are only mentioned in 1, 2 papers, for example: Weighted Operations in Module(WOM) [33];Lack of Cohesion in Operations (LCoo) [33].
Complete results are avaialable at the following link: https://docs.google.com/spreadsheets/d/1J3XhIwym07964UerGULRAttCgi28cynPnXznk42sWxg/edit?usp=sharing
We summarize all the metrics in the paper and summarize them in the table named "Summary of metrics" in the manuscript, and sort them according to the cited article, abbreviation, and full name, although in some papers, only the abbreviation or full name is mentioned. But we still aggregated 290 data.
But these mentioned frequencies do not evaluate which metrics are more predictive of maintainability, but only show that it is more likely to be related to maintainability, and this is the problem we will solve in the next chapter of the retrospective study.

Chapter 3

Method

3.1 Research Question

RQ1: Which internal source code metric can better predict maintainability in use?

Motivation: By answering this question, we can find out which metric and maintainability are high correlation, then in the process of software development can provide developers with a way to improve the maintainability of software and improve the quality and efficiency of software development. If developers can notice the internal code indicators related to software maintainability in the early stage of project development, and continue to pay attention to them in subsequent work to maintain them within an acceptable range, then the software in the later stage The time spent due to bug fixes will also be significantly reduced. We hope to conduct retrospective research by studying existing secondary research and observe the correlation between different code indicators and maintainability for specific software.

In order to answer this research question, the first thing we have to do is to find the metrics that may be related to the maintainability of the software, and then further analyze them to observe their correlation with the maintainability. The research question subRQ2 is a detailed introduction about this step.

subRQ1: Which metrics (internal, external and in-use) have been proposed to evaluate maintainability of object-oriented programs?

Motivation: For subRQ1, if we want to study which metrics can help predict maintainability, then first we need to know how to measure the maintainability of the software. Only when we have a method to measure the maintainability can the maintainability be used for comparison. We search and collect secondary literature, trying to find the quality in use that most of the currently included are used to measure the maintainability of the software. We only chose the object-oriented system because this is the most common system at present.

Moreover, object-oriented programs will have the following advantages to help us complete our Retrospective studies compared with Functional Programming.

Reason1: For object-oriented programs, data, and methods (methods can be understood as functions) are encapsulated together. When this is done, the change has little effect on the entire program [8]. But for Functional Programming, each version update of it considers as an essential Re-examine development as the whole process [8]. The maintainability of our calculation software is mainly to calculate the "bug fix" of each updated version separately (see 3 for details). Choosing object-oriented programs will significantly improve our work efficiency.

Reason2: For the development of the software itself, object-oriented programs re-

duce the code redundancy through inheritance, which can reduce unnecessary code smells in the software development process, and can make our calculation of maintainability more reliable [8].

Reason3: object-oriented programs are more comfortable to expand, and they are more suitable for version update iterations and are liked by developers [8]. For us, it's more easier to find object-oriented programs as our test software.

subRQ2: Which internal metrics correlate with maintainability in use?

Motivation: For subRQ2, we know that there are thousands of software indicators in the software engineering discipline. We cannot find all software indicators and use them to perform correlation analysis with maintainability. That is unrealistic and unwise. Therefore, we need to find those indicators that are related to maintainability or may be related to maintainability among these large numbers of software indicators. And focus on analyzing and studying the relationship between them and maintainability. In order to solve this problem, first of all, we will find the internal indicators of the code that have been proposed and are related to maintainability through secondary studies. Based on the internal metrics of these codes, we conduct a retrospective study to verify the correlation and degree of correlation.

3.2 Systematic literature review

In the systematic literature review, we can use different databases and a variety of retrieval and analysis techniques to comprehensively and accurately grasp the progress of research in a particular topic of interest. Our purpose is to study which metric can help predict the maintainability. In the end, our research result should be a correlation table between software internal metrics and software maintainability. We think that the retrospective study may be a better choice than SLR. But for subRQ1 and subRQ2, we can use the SLR method to solve.

3.2.1 Motivation

Compared with studying how to measure software maintainability and which internal software metrics will be related to software maintainability, there has been some literature that has studied the relationship between software indicators and software maintainability, although their results are not so general. But if we conduct a systematic literature review on their basis and count their research results, it will be more convincing and effective.

3.2.2 Data Sources and Search Strategy

Keywords

The keywords we searched are as follows: Search string: TITLE-ABS-KEY (("Systematic Literature Review" OR "Systematic Review" OR "Systematic mapping" OR "Systematic map") AND ("maintainability")) AND (LIMIT-TO (SUBJAREA , "COMP")) First, determine that our goal is a systematic literature review because our goal is to make a comparison table of internal maintainability indicators and external quality attributes and quality in use. It is best to record as many records

as possible in this table, so if we do not put Systematic Literature Review into keywords, we will need to consult too much literature so that the subsequent work cannot be carried out. Regarding the research area of the selected paper, we have made a restriction limited to software engineering. Although the literature review based on systematic literature review seems a bit repetitive, after our preliminary research, the current systematic literature review does not have a good review of internal indicators, quality in use and external quality attributes, most of them Only two aspects are analyzed, so we think our work is necessary.

Data Sources

The databases used to search for relevant literature was Scopus. Scopus was chosen because it claims to be the largest abstract database of peer-reviewed papers[33]. And the date of search is February 25, 2020.

3.3 Retrospective studies

In retrospective studies, the results of interest have already appeared at the beginning of the study[46]. For our research, there have been many secondary studies and they have collected various metrics related to software maintainability. We want to further study the relationship between these metrics and maintainability and the degree of correlation between them. Secondly, with better pertinence, we can directly explore the answers to the questions we need, and we can better explore the relationship between metrics and maintainability. For the three RQs, it is actually different RQs that further explore our results, so we will use retrospective studies methods to verify our thesis.

For RQ1, which metrics can better predict maintainability, using retrospective research methods, we can use previous research results to focus on those metrics that have been shown to be related to maintainability.

For our research object, we only chose one target software for analysis, so our research is not universal. We only chose one software for analysis because we believe that if multiple software is selected for analysis, the scope of research may be too broad and lead to too much work to obtain data, and for different types of software of different difficulty, it may be The most relevant metrics are also different. In this case, we need to select multiple software with the same type and little difference in difficulty for analysis. Use some of the software to analyze the metrics that are most relevant to maintainability. The rest is used. To verify, observe whether the metric we finally obtained can be used to help us predict the maintainability of the software. Doing so will cause us to spend a lot of time in finding suitable software for analysis. Therefore, our goal will be different versions of the software. Our goal is to analyze the correlation between the MTTR of different modules of different versions of the same software and the average metric of the corresponding modules of the same version. These metrics were collected in the literature review.

It should be noted here that the metric corresponding to the corresponding MTTR is not in the same version, but in the previous version. For example, a certain software has version 1 and version 2, and the MTTR measured by version 2 actually

represents the maintainability of the version 1 code. Because when the developer completes the version 1 code, the maintainability of the code cannot be measured immediately. We need to measure the maintainability of the software by the average time it takes developers to maintain bugs in this version of the code. When the developer completes the maintenance, this information will be released in version 2, along with the new features.

3.3.1 Motivation

With better pertinence, we can directly explore the answers to the questions we need, and we can better explore the relationship between metrics and maintainability by using exciting secondary studies.

3.3.2 Target software

For the target software we analyze, the first thing is the java file, because most of the current code analysis software is java-oriented. Then the selected software should record the time spent on each update on GitHub (some software version records are different from the actual time spent, such as the issue that the developer has modified the code to create, this It will result in a large change of code, but the record only shows that it took a short time), and for each issue, the development team needs to classify it, such as whether it is a bug type or improvement or new features. Based on the above requirements, we found the following qualified software on GitHub:

https://github.com/jenkinsci/jenkins
https://github.com/mybatis/mybatis-3
https://github.com/alibaba/Sentinel
https://github.com/apache/dubbo
https://github.com/elastic/elasticsearch

After further screening, we chose elasticsearch as our analysis object. Elasticsearch is an open source, distributed, RESTful search engine developed by elastic. The software currently has 273 releases and 23690 issues. The latest version is currently 7.7.0, and we can trace it back to the earliest version of 6.8.1. Earlier versions were excluded because they could not collect version update information. The reason we choose elasticsearch is compared with other software, this software has more versions, and the problem record is also very detailed, and there is a development team to discuss this in the issue details interface In the issue process, we believe that if there is a discussion, the probability that the time spent in the record and the actual time spent are different is relatively small.

3.3.3 Tools

According to our previous literature review, we have established a total of 9 metrics, including WMC, DIT, NOC, CBO, RFC, LCOM, LOC, DAC, NC, and CC. To meet this requirement, we consulted the relevant literature [22] and found that CKJM can measure WMC, NOC, RFC. The remaining metrics can be measured in Designitejava.

DesigniteJava [51]: DesigniteJava is a code quality assessment tool for code written in Java. It detects many design and implementation odors that indicate maintainability issues in the analyzed code. It also calculates many commonly used object-oriented indicators. It can help you reduce the technical burden and improve the maintainability of the software. [reference] It can mainly measure three aspects of java software: Design smells, Implementation smells, and Metrics. Design smells: DesigniteJava detects 17 design smells violating one of the object-oriented design principles. Implementation smells: DesigniteJava detects ten Implementation smells to keep your software readable and less complicated [52]. Metrics: DesigniteJava computes object-oriented design metrics that are helpful to gauge the structural health of the software.

CKJM [53]: The program ckjm calculates the object-oriented metrics of Chidamber and Kemerer by processing the bytecode of compiled Java files. The program calculates the following six indicators proposed by Chidamber and Kemerer for each class. He can help us measure the remaining metrics [53].

Difficulties encountered: The main reason is that the two versions of Java required by the two software are different, so I encountered a lot of problems during debugging. The jdk11.0 version is installed on the computer, and I have been prompted when calling the jar file with the console. Error calling class file, and finally found the problem and repaired after searching the user instruction manual, and the JDK version files required by the two software are different. DesigniteJava requires jdk11.0, while ckjm requires jdk1.5.0. The two software need to be switched differently during the test, so there will be some troubles.

3.3.4 MTTR

MTTR is an indicator that we have found to measure the maintainability of software. In this section, we will introduce how MTTR is collected and why we do so. First, we enter the GitHub interface of elasticsearch. After viewing the version update information, we select those issues that are marked as bugs to collect information and analyze. The reason for choosing only bug issues is that according to the definition of MTTR, we should calculate the time from when a bug occurs to the time when the bug is resolved, that is, from the time each bug issue is created to the time it is submitted. The calculation of the date starts from the day the issue is created, and it needs to be increased by one after the two dates are subtracted in the calculation process.

After collecting the time spent on all bug issues in a version, we classify the collected data by module and then calculate the average time from bug discovery to resolution for each module, which is the MTTR of each module.

The reasons for choosing module level instead of class level are as follows:

- Conducive to more accurate data, some bug modification records involve multiple modules, and some of them have no actual effect, such as docs, in order to make the data more accurate, we need to remove it and recalculate MTTR. So we classify according to modules.

- It is easy to calculate and reduce data loss, because the bug repair involves multiple files in a single package, and there may be changes to the name of variable

and other small changes that affect the accuracy of the data (for example, the system has both login and verification functions, There are hundreds of lines of code in both logo.java and verify.java. We added a permission function (selec table user type) in the bug modification. Although this function is not a bug modification of the verification function, it will be verified. There is one more return value in the return value of, so the MTTR of verify.java will be very small, and the metrics of verify.java will be very large, which will affect our results.).

- In some modifications, only a small part of the code has been modified, but because the smallest unit of time recorded on Github is one day, in fact, the maintenance staff may only solve it in two or three hours. If we choose class-level, The impact of this part will be great, and module-level can help us reduce some inaccuracies.

We encountered some problems in actual operation, here is their description and solutions:

- The software selected is too large, too many versions are updated, we need to deal with the number of bugs and calculate the MTTR value of more than 1400 in a week.

- Part of the MTTR calculation will have problems because we use the calculation time and merged time on GitHub to calculate MTTR, these are manual operations, so sometimes there will be problems such as untimely update or being ignored, so for For this problem, we removed the bugs whose repair time was more than 60 days.

- A large part of all bug issues in elasticsearch not only modify a module, so there may be one MTTR corresponding to multiple modules, we need to divide the MTTR in this bug issue. The judgment basis for the division is the number of lines of code that have changed in each module. We know that just changing the number of lines of code does not accurately divide the time spent by developers on different modules, but this is the most intuitive and the only information we can get from the GitHub bug issue page.

- In actual operation, the smallest unit recorded on GitHub is one day, which is also a limitation of our retrospective study. In some cases, developers only modify a small part of the code, and they spend less than an hour, Which may lead to inaccurate results. As discussed in the selection of software, although we have screened the software used for analysis to avoid the fact that the time it takes to actually fix the bug is different from the time recorded on GitHub, we still cannot guarantee to record on Github The time is completely accurate.

3.3.5 Calculation of metrics

After obtaining the MTTR of each module of different versions, all we need to do is to obtain the internal indicators of the different modules of the corresponding version. This step is the same as obtaining MTTR, which requires a lot of time to

Pearson range	Relevance
r <= 0.1	Unrelated
0.1 <r <= 0.3	Small
0.3 <r <= 0.5	Medium
0.5 <r <= 0.7	Big
0.7 <r <= 0.9	Very large
0.9 <r <= 1.0	Almost perfect

Table 3.1: Pearson correlation

obtain manually. On the bug issue page, we can know the file name of the code file specifically changed for each issue. In this step, we extract all the changed files in the bug issue. As mentioned above, we have to go to the previous version of the source code to find the specific files that have changed and put them together according to the module. We save the changed file name in the Excel table and then use Python to read the Excel table, automatically go to the previous version of the source code to find the file and copy to the specified directory. Then use DesigniteJava software to analyze the obtained file, and then take the average value according to the module to obtain the metric value of each module of the final different version.

3.3.6 Correlation analysis

After obtaining the data required for analysis, it is necessary to verify the correlation between the selected different metrics and MTTR. There are several quantitative statistical techniques that can be applied. In this work, Pearson correlation is selected [42] because it has been cited in other studies related to maintainability, such as Wang et al. [34] Lincke et al. [36] He Zhuo et al. [59] and N. Barbosa Jr. et al [14].

We used Python's pearsonr function to analyze each metric and MTTR of different versions of Elasticsearch. We saved the calculated MTTR and metric values in different excel files according to the module, and then read the excel file with Python and call the "pearsonr()" function to obtain the Pearson value of each module for different metric. The larger the absolute value of the result, the more relevant this metric and MTTR are. If the result is negative, it means that they are negatively correlated. To be able to generate this result, we need to summarize the collected data, create an Excel table for each module, record the MTTR of each version of this module and the average value of each metric.

The following Table3.1 is proposed by Zhou and Leung [58] and shows the range of values of the correlation between the two metric. For positive values, it can be used for negative values of correlation.

3.4 Alternative method

3.4.1 Survey

Surveys is a kind of understanding of the existing status and people's knowledge. For our topic,we need to perform a quantitative and accurate analysis of the question of whether we have new metrics or the correlation between internal metrics and maintainability. So, survey does not apply.

3.4.2 Case study

Case study has specificity and uncertainty for the analysis of experiments. The cases themselves are cases, and each case has its own characteristics that cannot be copied. For the topics mentioned in our thesis, pure case analysis may influence the analysis of metrics. For example,a certain metric will have an advantage for that particular case. It is not possible to compare all metrics fairly, so case study is not suitable for our method.

Chapter 4

Results and Analysis

4.1 Literature review results

According to the scoups we mentioned in the method chapter (Search string: TITLE-ABS-KEY (("Systematic Literature Review" OR "Systematic Review" OR "Systematic mapping" OR "Systematic map") AND ("maintainability")) AND (LIMIT-TO (SUBJAREA, "COMP")) A total of 45 related papers were used as keywords. After browsing their abstract and articles, we selected a total of 17 papers as our research objects. The detailed information of the selected paper is in the Table 4.1. Other papers that do not meet the requirements can be found in appendix.

Table 4.1: Literature review results

Authors	Title	Year
Elmidaoui S., Cheikhi L., Idri A.	Towards a Taxonomy of Software Maintainability Predictors [22]	2019
Elmidaoui S., Cheikhi L., Idri A.	Software product maintainability prediction: A survey of secondary studies [21]	2017
Santos D., Resende A., Junior P.A., Costa H.	Attributes and metrics of internal quality that impact the external quality of object-oriented software: A systematic literature review [48]	2017
Malhotra R., Chug A.	Software Maintainability: Systematic Literature Review and Current Trends [38]	2016
Jabangwe R., Börstler J., Šmite D., Wohlin C.	Empirical evidence on the link between object-oriented measures and external quality attributes: A systematic literature review [33]	2015
Saraiva J., Soares S., Castor F.	Towards a catalog of Object-Oriented Software Maintainability metrics [50]	2013
Abílio R., Teles P., Costa H., Figueiredo E.	A systematic review of contemporary metrics for software maintainability [7]	2012
Montagud S., Abrahão S., Insfran E.	A systematic review of quality attributes and measures for software product lines [41]	2012
Burrows R., Garcia A., Taïani F.	Coupling metrics for aspect-oriented programming: A systematic review of maintainability studies [1]	2010
Riaz M., Mendes E., Tempero E.	A systematic review of software maintainability prediction and metrics [45]	2009
Elmidaoui S., Cheikhi L., Idri A., Abran A.	Empirical studies on software product maintainability prediction: A systematic mapping and review [23]	2019

[No author name available]	ICSOFT 2019 - Proceedings of the 14th International Conference on Software Technologies	2019
García-Mireles G.A., Moraga M.Á., García F., Calero C., Piattini M.	Interactions between environmental sustainability goals and software product quality: A mapping study [27]	2018
Malhotra R., Khanna M., Raje R.R.	On the application of search-based techniques for software engineering predictive modeling: A systematic review and future directions [39]	2017
[No author name available]	Proceedings - 8th International Conference on Advanced Software Engineering and Its Applications, ASEA 2015	2016
Bastos C., Afonso P., Costa H.	Detection techniques of dead code: Systematic literature review [Técnicas para detecção de código morto: Uma revisão sistemática de literatura] [15]	2016
García-Mireles G.A., Moraga M.Á., García F., Piattini M.	Approaches to promote product quality within software process improvement initiatives: A mapping study [28]	2015

4.1.1 Metric

After selecting relevant papers, we find the metrics that have been shown to be related to maintainability, and classify them according to the mentioned paper, the abbreviation of metrics (if mentioned in the article), and the full name of metrics. A total of 291 metrics related to or related to maintainability were mentioned in these 17 papers. The following Table 4.2 describes all metrics that occur more than once and their sources.

Due to the limitation of the length of the paper, we cannot quote all the data, complete results are avaialable at the following link: https://docs.google.com/spreadsheets/d/1J3XhIwym07964UerGULRAttCgi28cynPnXznk42sWxg/edit?usp=sharing

We summarize all the metrics in the paper and summarize them in the table named "Summary of metrics" in the manuscript, and sort them according to the cited article, abbreviation, and full name, although in some papers, only the abbreviation or full name is mentioned. But we still aggregated 290 data.

Based on this table, we chose The following metrics: WMC, DIT, NOC, CBO, RFC, LCOM, LOC, DAC, NC, CC

It can be found that most of our selected metrics are from the C & K metric suite. Based on the number of occurrences of these metrics in the papers we collected, we selected WMC, DIT, NOC, CBO, RFC, LCOM, LOC, DAC, NC, and these metrics appeared most frequently, all of which reached more than four times. CC can be selected because we think that the complexity of the code will greatly affect the readability of the code and thus the maintainability of the software, although this effect is largely due to the familiarity of the maintainers with the code Different and different.In the data analysis and acquisition part, we removed the two metrics of DAC, because most of the current java metric analysis software can not get it.

- WMC: Its value is obtained by counting the methods implemented in the class or by adding the complexity of the method, and this complexity is measured by the "ring complexity" developed by McCabe [33]. It directly links to the definition of the object and the properties of the object.

- DIT: It is the value corresponding to the maximum distance between the root class and the farthest node. The deeper this class is in the hierarchy, the more methods it may inherit, which makes it more complicated to predict its behavior. It directly links to the definition of the object.

- NOC: Equal to the number of direct descendants of the class. It indicates the potential impact of the class on the entire system. It directly links to the definition of the object.

- CBO: Its value is calculated by counting the number of classes that are coupled as classes of customers and information providers. The greater the number of couplings, the greater the sensitivity of other parts of the project to changes, making maintenance more difficult. It is directly linked to the communication between objects.

- RFC: Involves combining the complexity of the class and communicating with other objects through multiple methods. Its value is calculated by counting

Metric	Full name	Source
DIT	Depth of inheritance tree	[22] [20] [15] [31] [40] [37]
NODBC	Number of data base connections	[22] [15] [31] [37] [20]
WMC	Weighted methods per class	[22] [15] [33] [31] [20]
NOC	Number of children	[15] [31] [37] [20]
LOC[2]	Line of code	[15] [33] [20] [40]
RFC	Response for a class	[15] [33] [31] [22]
NC	Number of classes	[22] [20] [31] [40]
CBO	Coupling between object	[22] [20] [15] [31]
DAC	Data Abstraction Coupling	[22] [20] [31] [40]
NAA	Number of attributes added	[22] [20] [40]
MPC	Message Passing Coupling	[22] [31] [40]
NM	Number of methods	[22] [33] [20]
LOC[1]	Lack of cohesion	[15] [31]
CAE	Coupling on Advice Execution	[4] [37]
CBC 2	Coupling Between Components	[4] [37]
CDC	Concern Diffusion Over Components	[20] [4]
EC	Efferent Coupling	[31] [4]
LCC 2	Loose class cohesion	[22] [20]
NA	Number of attributes	[22] [20]
NAgg	Number of aggregations	[22] [40]
NAggH	Number of aggregations hierarchies	[22] [40]
NGen	Number of generalisation	[22] [40]
NGenH	Number of generalisation hierarchies	[22] [40]
NOT		[22] [31]
CAMC	Cohesion among methods in a class	[22] [31]
CBO_U	CBO Using by the methods of class	[22] [40]
CC	Cyclomatic complexity	[22][39]
LCOM	Lack of Cohesion over Operations	[22] [4]
MI	maintainability index	[22][39]
POF	Average weighted methods per class	[22][39]
TCC	Total cyclomatic complexity/Tight class cohesion	[22] [31]
TWPR	Tight class cohesion	[22] [20]

Table 4.2: Metrics

the number of methods that can be executed in response to the received message. The more methods an object calls through a message, the greater the complexity of its class. Generally, the larger the coupling, the higher the cost of maintenance activities. It directly links to the properties of objects and the communication between objects.

- LCOM: You can calculate its value as the number of method pairs without shared variables minus the number of pairs with shared variables. If the value is negative, it is considered zero. The lack of cohesion adds complexity. It is directly linked to the attributes of the object and is a reverse measure of cohesion.

- LOC: Lines of code simply counts the lines of source code (line break characters) of a certain software entity. It is a simple yet powerful metric to assess the complexity of software entities. Since it is depending on code conventions and format, it is critical to use it in generated codes since it may lack of line breaks. Additionally it can only be measured in the source code itself from the front-end and is therefore a front-end side metric.

- DAC: The DAC measures the coupling complexity caused by Abstract Data Types (ADTs). This metric is concerned with the coupling between classes representing a major aspect of the object oriented design, since the reuse degree, the maintenance and testing effort for a class are decisively influenced by the coupling level between classes.

- NC: Present how many classes are in a java file.

- CC: CCis a measure of the control structure complexity of software. It is the number of linearly independent paths and therefore, the minimum number of independent paths when executing the software.

4.2 Retrospective study

4.2.1 MTTR calculation result

In Chapter 3, we stated that we would calculate the MTTR corresponding to each module of each version according to the module recording time of the bug fix issue on Github. But in the actual operation process, it took us a lot of time, because we need to calculate the total maintenance time of the bug fix issue (as defined by MTTR). In addition, for a bug fix issue that changes multiple modules, we also record the total number of modified lines (including addition, modification, and deletion) and the corresponding modification of different modules to calculated the MTTR proportionally, all this needs to be calculated manually.

The following Figure 4.1 shows all the MTTR we collected and their changes for each version. One thing we need to pay attention to is that not all modules can collect MTTR in every version, only those versions that have bugs and have been fixed have. The abscissa is the version, and the ordinate is the MTTR value. 7.2.0-7.5.1 Version collected 875 bug issues totally. The MTTR data we collected and the calculations are in the data sheet in the following google drive link: https://docs.google.com/spreadsheets/d/1qaTwTt4gQA0OAhbyNc6BmFl95fm_SqaCdTkSctYQW4E/edit?usp=sharing

The second column is the issue number, the third column is the total MTTR, and the following columns are the modules contained in this issue and the total number of modified lines of each module and the divided MTTR.

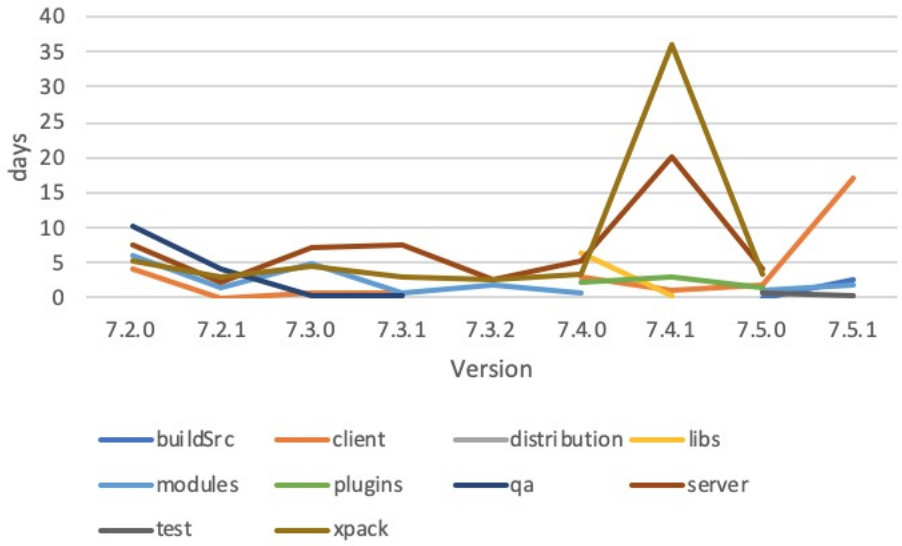

Figure 4.1: MTTR

4.2.2 Metric

According to the data selected by the metrics we obtained in Table 4.2, and according to the relevant test software mentioned in the paper (Empirical studies on software product maintainability prediction- A systematic mapping and review [23]), we used

Version	buildSrc	client	distribution	libs	modules	plugins	qa	server	test	xpack
7.2.0	/	32	/	1	12	8	7	126	4	59
7.2.1	/	1	1	/	3	/	1	31	/	/
7.3.0	/	9	/	16	25	1	12	231	11	121
7.3.1	5	7	2	/	2	/	1	44	/	56
7.3.2	/	/	/	/	2	/	/	12	/	14
7.4.0	2	11	1	10	20	2	1	168	9	199
7.4.1	/	1	/	/	/	4	/	14	/	27
7.5.0	/	21	4	/	9	7	/	168	8	238
7.5.1	1	2	/	1	2	/	2	40	1	46

Table 4.3: Data distribution

DesigniteJava and ckjm to complete the relevant metrics. For calculation, first, we extract all the modified files in the bug fix issue according to the modules, and then use the software to test, and get the following pictures, the results are displayed according to the module. These tables show the average metric value of the files modified by bugs in each version of each module. We need to extract the modified files and analyze them separately before averaging, so it takes a lot of time. In order to show the change of metric more clearly, all the values are standardized, the maximum value is 1, and the minimum value is 0. The abscissa is the version, and the ordinate is the standard value of metric

These data are derived from 1876 files that have been changed due to bug fixes, and their distribution in each module of each version is presented in Table 4.3.

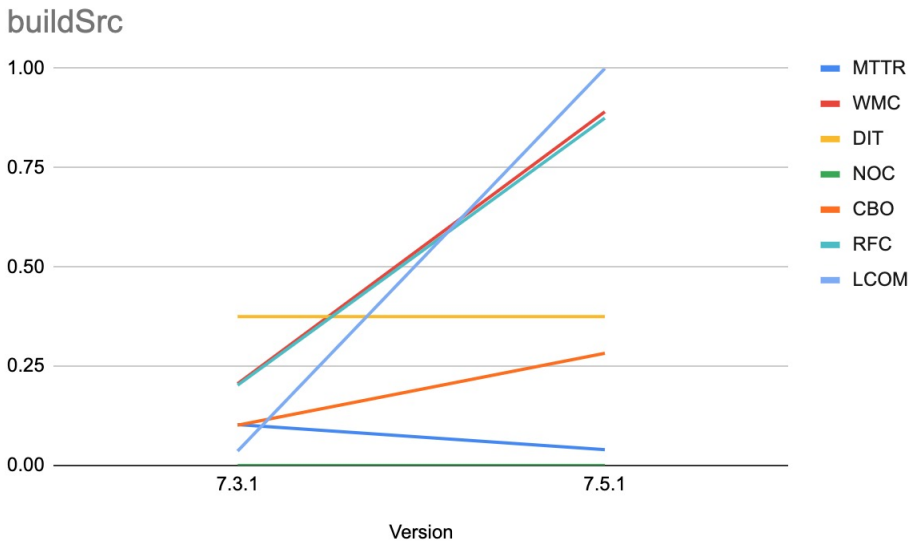

Figure 4.2: buildSrc Metric Value

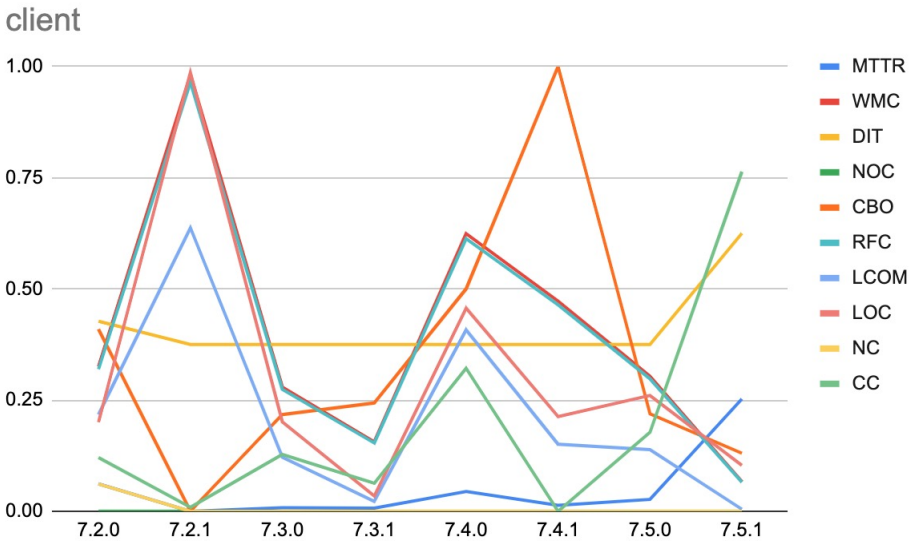

Figure 4.3: client Metric Value

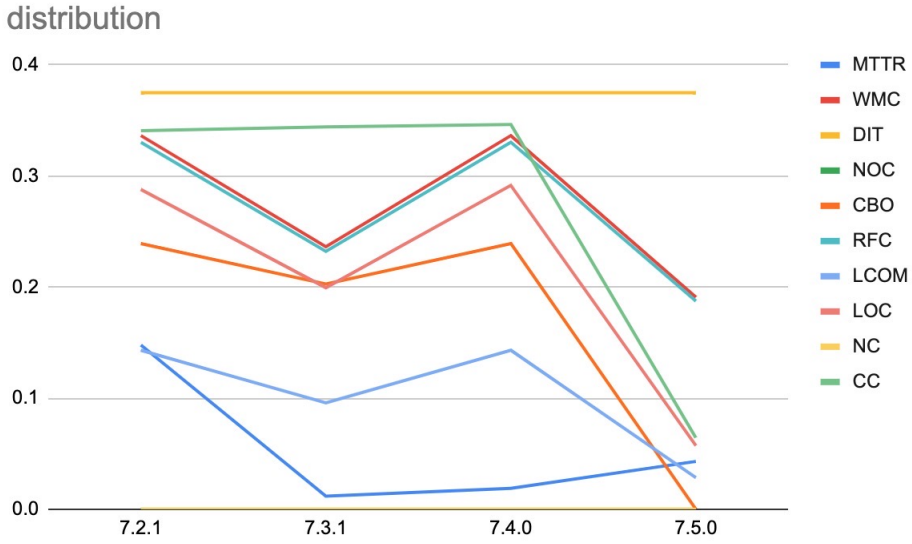

Figure 4.4: distribution Metric Value

4.2.3 Pearson correlation

After getting the MTTR value and average metric value of each module of each version, we used them to perform correlation analysis. The results are shown in the Figure 4.12 and Figure 4.13.

Figure 4.12 shows the correlation between different metrics and MTTR of each module. The ordinate is the Pearson correlation index, and the maximum is 1, indicating a perfect correlation. It should be noted here that we have processed the absolute value of the result because the sign of the value only represents whether it is positive or negative, and the degree of correlation is determined by the value.

Based on the definition of The Pearson correlation coefficient, it only reflects the

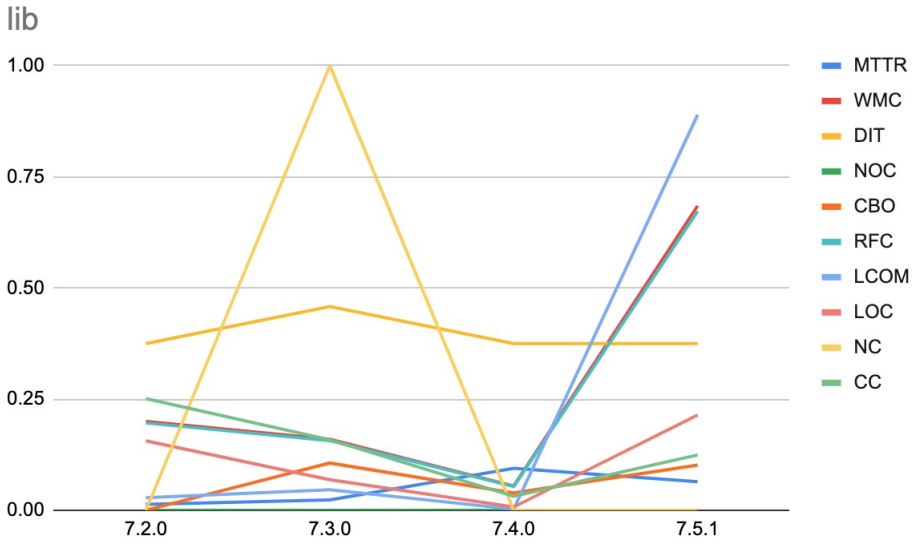

Figure 4.5: lib Metric Value

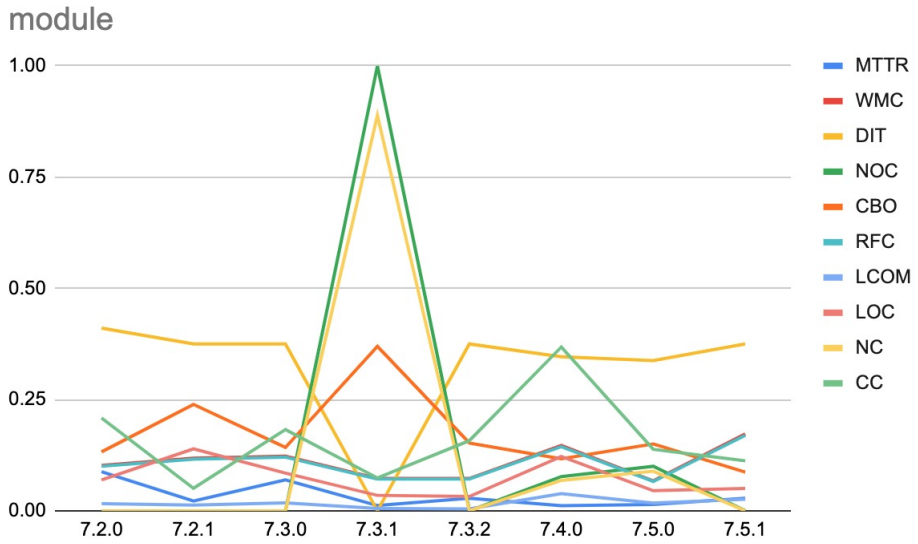

Figure 4.6: module Metric Value

strength of the linear correlation between the two variables. The larger the absolute value of r, the stronger the correlation. The value range of r: -1≤r≤1.

- When r> 0, it indicates that the two variables are positively correlated, that is, the higher the value of one variable, the higher the value of the other variable;
- When r <0, it indicates that the two variables are negatively correlated, that is, the larger the value of one variable, the smaller the value of the other variable;
- When r = 0, it indicates that the two variables are not linearly related.

Therefore, the larger the absolute value is, the higher the correlation is, and the correlation is not related to the sign.

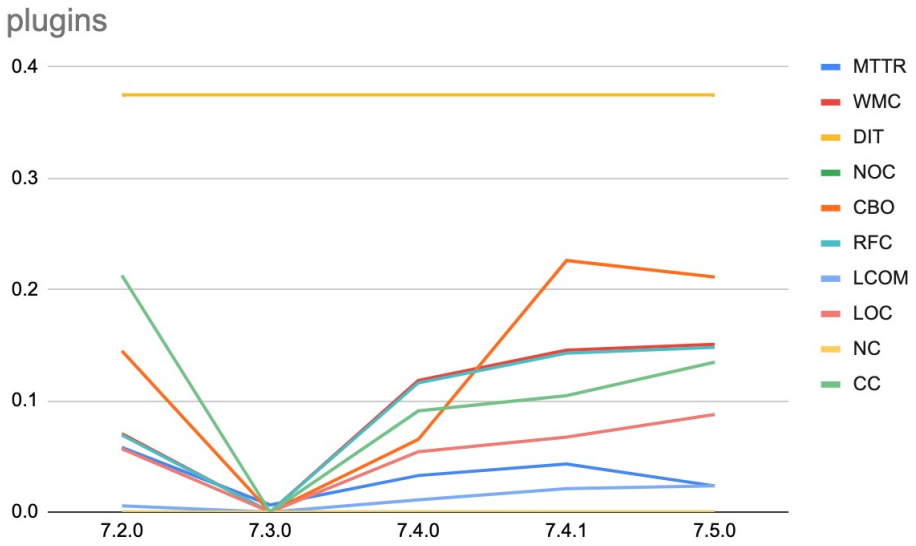

Figure 4.7: plugins Metric Value

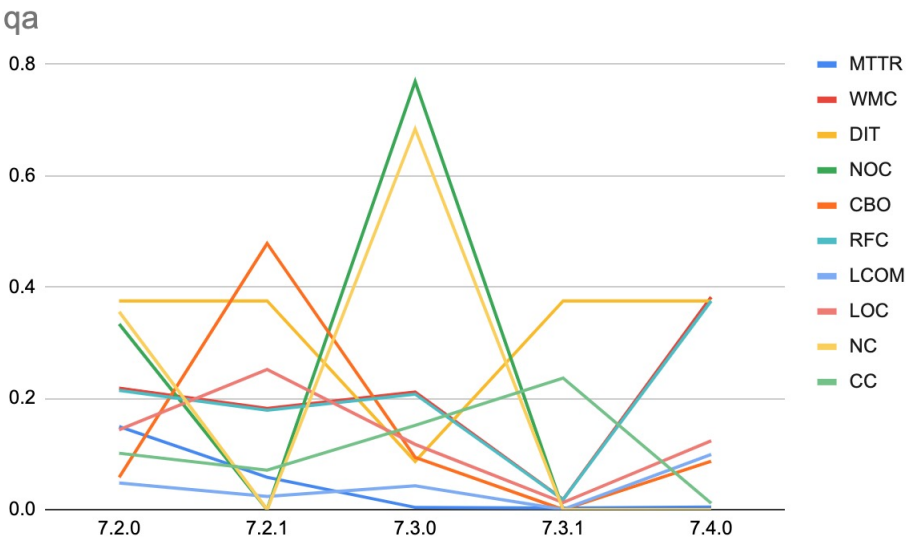

Figure 4.8: qa Metric Value

According to the results of Figure 4.12 Pearson calculation, we know that the same metrics have a gap in the correlation between different modules and MTTR. According to the explanation of table 3.3.6 Pearson coefficient, we found that LOC, CC, WMC, and RFC have a strong correlation (Pearson coefficient appears more than 0.7 twice).

At the same time, as an indicator, it must have strong applicability and compatibility rather than accidental uniqueness [56], which means that it should be applied to all modules in all software. Therefore, We averaged the results again, trying to analyze the correlation between different metrics and the MTTR of the entire software. It can be seen from Figure 4.13: The correlation among these four metrics is relatively

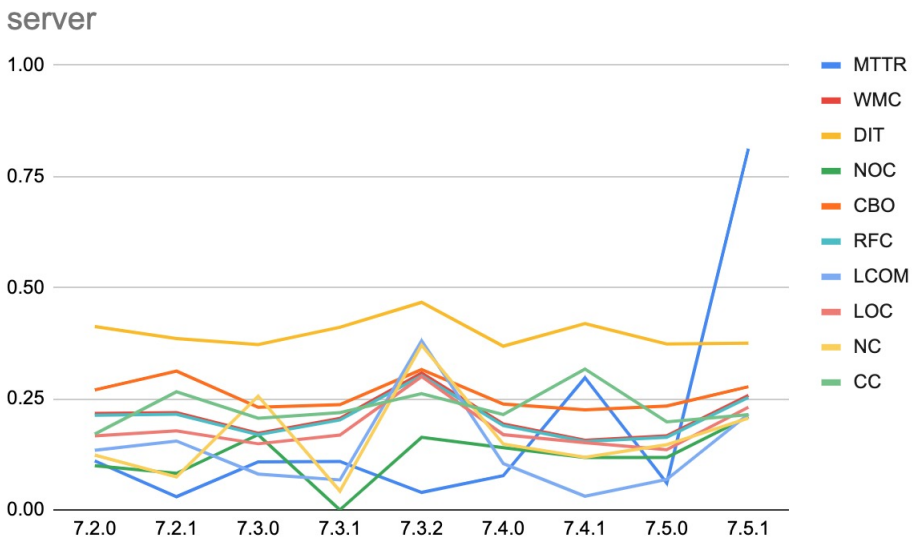

Figure 4.9: server Metric Value

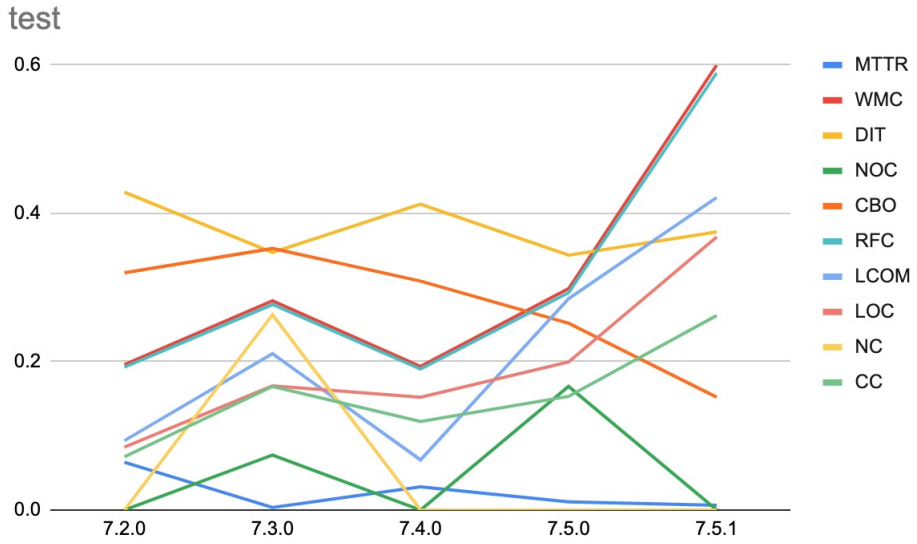

Figure 4.10: test Metric Value

high, in order: CC, RFC, WMC, LOC(Match the previous result). The two are relatively low, in order: DIT, NC.

4.2.4 Verification

According to the relevant results of different metrics and MTTR of different modules in the previous section, we chose the xpack module to verify its correlation with MTTR. The main reason for choosing the xpack module is that compared to other modules, each version of the xpack module has been modified more due to errors, which can minimize errors. As you can see from Figure 4.12, although the server

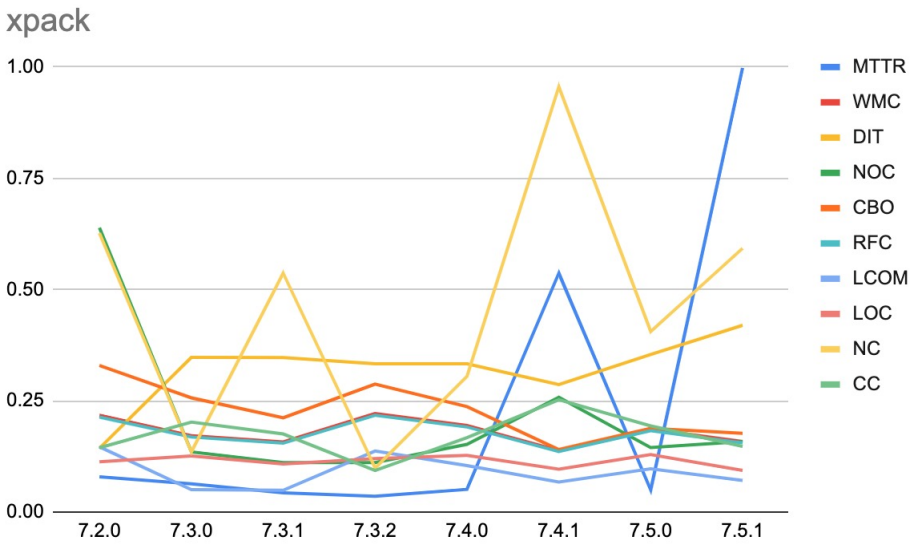

Figure 4.11: xpack Metric Value

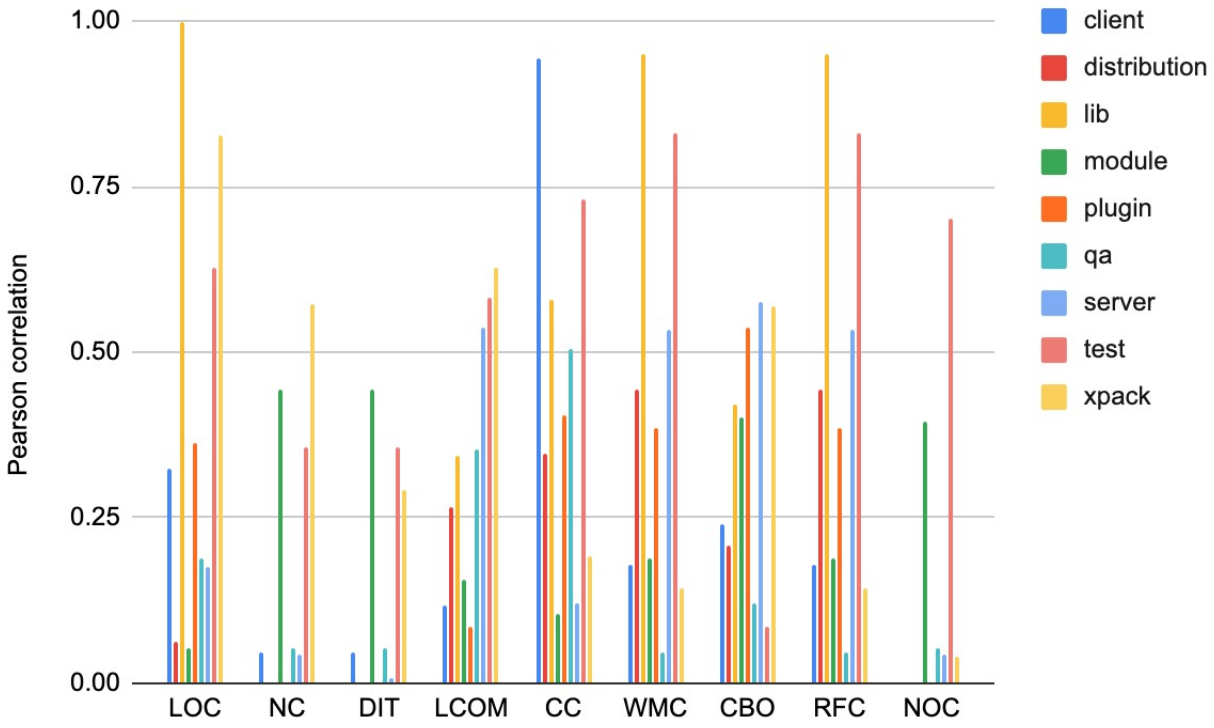

Figure 4.12: Pearson result

module of each module has modified many files, its correlation with metric is not very good. After selecting the prediction module, we use the LOC indicator to make a simple prediction of MTTR because it has the highest correlation.

According to our further data collection and analysis, we use the same method to obtain the MTTR of the xpack modules 7.6.0, 7.6.1, and 7.6.2. After normalizing

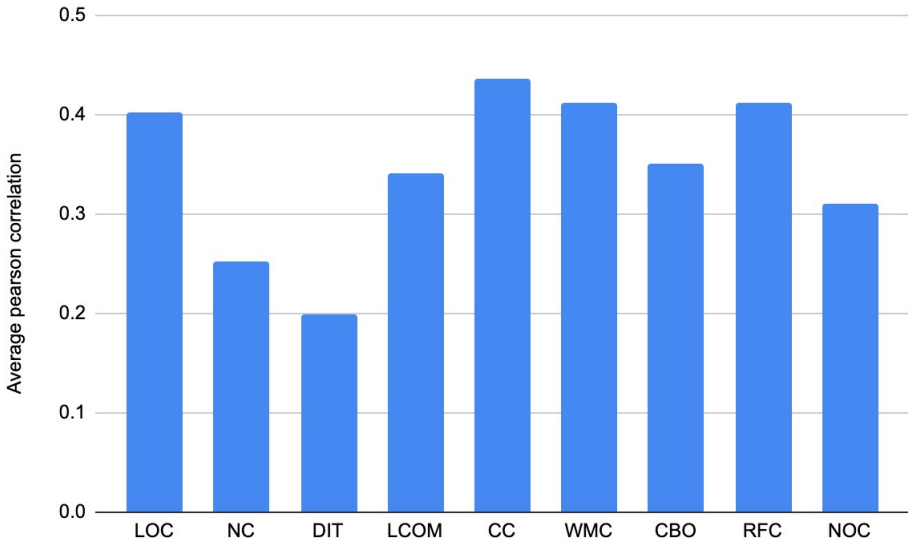

Figure 4.13: Average Pearson result

them, the values are 0.04139, 0.05113, and 0.04028, respectively. And their LOC metric values are 136.3386, 127.1505 and 145.8864. We used the collected 3 sets of data and the previous version of the xpack module's LOC metric and MTTR data to perform a correlation analysis, and obtained their correlation index of 0.6893.

Chapter 5

Discussion

5.1 Research question

- First of all, for our Research Question, for RQ1: Which internal metric can help predict maintainability better? The retrospective study results finally indicate that for elasticsearch, the four indicators of LOC, CC, WMC and RFC have the most values and MTTR correlation. However, their correlation is not very high, both are around 0.4, which is a moderate correlation. NC, DIT, LOCM, CBO, and other indicators are related to MTTR to a small extent. Among the selected metrics, DIT has the smallest correlation, and the correlation index is only 0.2. That is to say, for this RQ, according to our results, the most helpful indicators for our maintainability are LOC, CC, WMC and RFC.

- For subRQ1: subRQ1: Which measurement is used to evaluate the maintainability of object-oriented programs? Our conclusion is MTTR (mean time to repair), we mentioned in the literature review that there is mean time to repair (MTTR) [38], mean time to correct maintenance (MCMT) [38], mean time to preventive maintenance (MPMT) [38] and the longest time to corrective maintenance (MaCMT) [38] four external metrics to test the maintainability, but according to what we found about maintainability definitions and understanding of these definitions, we feel that MTTR is more in line with our requirements and can better measure maintainability.

- For subRQ2: subRQ2: Which internal metrics are related to maintainability? This is the result obtained after our SLR. We collected and sorted 291 related metrics. Although we only selected 8 of the most mentioned in the paper, all 291 of these metrics are also the answer to our question.

  Although we got corresponding answers in the course of our retrospective study, more importantly, we have gained a lot in the whole process of the study.

5.2 Literature Review

In this part, we will briefly introduce the patterns and principles we found. First of all, for our basic background and principles, such as understanding the definition of maintainability, factors affecting maintainability and related metrics, we use the method of literature review, Introduced in detail in chapter 4 of the paper, we searched for it in Scoups and selected the category of software engineering major,

and screened the abstract of the resulting paper after abstracting and statistically obtained the relevant background and content.

In order to solve the problems mentioned in our paper, we first solve the choice of the paper. In this part, as we mentioned in chapter "Related Work," we must first figure out what is the definition of maintainability, what are the metrics have been shown to be related to maintainability, and in this regard, What research has been carried out. And we also found two quality attributes with maintainability definition; they are: ISO9126 / 25010 and IEE24765 [2], and explained and explained it in detail in chapter "Related Work." In this part, we have a better understanding of what is maintainability, and use the definition as a standard and combine our own understanding to correctly define and use maintainability in the paper.

After that, we systematically analyzed the factors that affect maintainability. We need to understand which factors will affect the maintainability, so that we can analyze them in detail in future studies and found three aspects in ISO9126: "Internal Quality, "External Quality "and" Quality in use. " This also helps us a lot when we are looking for related metrics. It can help us understand the maintainability more intuitively and classify the related indicators found.

Based on ISO9126's classification of impact indicators, we look for relevant indicators to measure maintainability. In this part, we have selected a total of 17 related papers and collected 291 related metrics. In these papers, they only show that these metrics have a certain relationship with maintainability, and even some papers use a lot of space. To explain the relationship between the two, but they do not analyze the correlation between the two, so, based on it, we collected related metrics and analyzed their correlation.

We collected all the results during the collection process, so we will have a list of 291 related metrics, but based on our actual working hours and the limitations of manually collecting data, we cannot bring all the metrics into our studies are tested one by one. We still choose the most appearing metrics to carry out our studies as much as possible within the constraints of time and workload, and try our best to meet our needs.

5.3 Retrospective study

For the change of the MTTR value of different modules in each release, we can see from Figure 4.1. In the nine versions we selected, the MTTR values of the xpack and server modules have always been relatively large compared to other modules. These two modules also have the most changes, and most of the modifications occurred in these two modules. We guess that the changes that happened in these two modules are all about the system's main function. Compared with other modules, they will spend more time to fix the bug, so the value of MTTR will be higher than other modules.

In the 7.4.1 version, the xpack and server modules have high MTTR values. After excluding the impact of extreme data, we guess that the reason may be that in these two modules in the 7.4.1 version, more serious bugs have occurred. It takes a lot of time for the development team to repair.

In general, the overall trend of MTTR remains unchanged, but extreme data will

appear in some versions.

From Figure 4.12, we can easily find that the correlation between different modules of the same metric will be different. We speculate that it may be related to the functions implemented by different modules. It is possible that for different functional modules, there are different metrics to measure their maintainability better. For example, for some software, developers may spend a lot of time fixing bugs in security modules, while other functions may take little time to fix. Therefore, we guess how to better measure the maintainability may be related to the function of the module. Our other conjecture is that the reason for these differences is the size of each module. It may be that for modules of different sizes, the measures that can better measure their maintainability are also different.

From the perspective of results, the values of the four indicators LOC, CC, WMC and RFC are most related to MTTR, but the correlation between them is not very high, all around 0.4, which is a medium correlation. In the subsequent verification of the LOC metrics of the xpack module, the correlation between the internal metrics and MTTR is again shown. We guess that for different software, the correlation between different metrics and MTTR may be different, but there will always be a set of metrics related to MTTR. The degree of correlation depends on various aspects, such as the function of the software, the familiarity of the development team with the code, and so on. For the software selected in our experiment, it is understandable that the correlation between LOC and CC and maintainability is higher. Because as the number of lines of code and the cyclomatic complexity of the code increase, the development team needs more time to read and understand the previous code, which makes the time spent fixing bugs longer.

From Figure 5.1 we also noticed that the correlation between different metrics and MTTR of the lib module is very high, while on the other hand, the correlation between different metrics and MTTR of the qa module is very low (Figure 5.2). We guess this may be related to the functions implemented by different modules. In addition, from the results, we can see that the highest correlation is CC. We think this result is reasonable, because the higher the complexity of the code, the more time the developer needs to understand and develop the code, although this code may be written by himself.

Our retrospective study also have a lot of potential weaknesses and deficiencies, and these deficiencies are likely to be important reasons that affect our results. Some are the shortcomings of the experiment itself, we have tried our best to make up for them, but we thought of some methods that might better solve them.

- MTTR can be calculated more accurately: because we calculate it manually, we cannot calculate the time in the release note very accurately. Days are the smallest unit on Github, but in fact, there may be many issues from discovery to resolution which took only a few hours. These seemingly few errors may reach 30, 40%, which will affect our results. If we want to solve this problem, we may need to work with a professional development team to track the development cycle and record the time when the bug appeared and completed in detail and accurately. However, this kind of retrospective study that tracks the development cycle of the project takes much more time than we expected, and it is also difficult to find a professional development team and let them promise

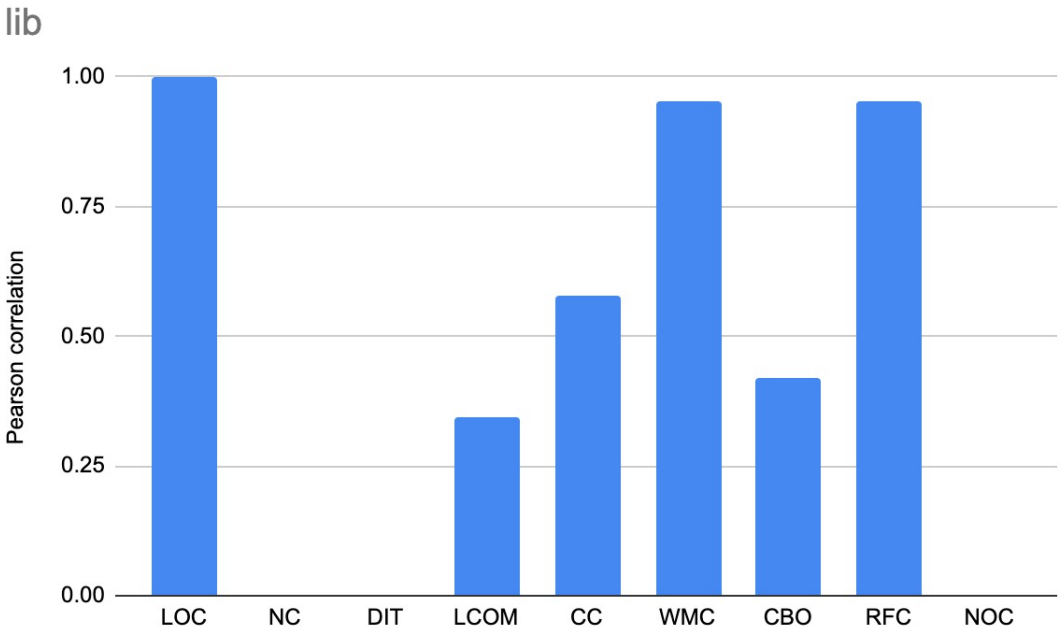

Figure 5.1: lib module correlation

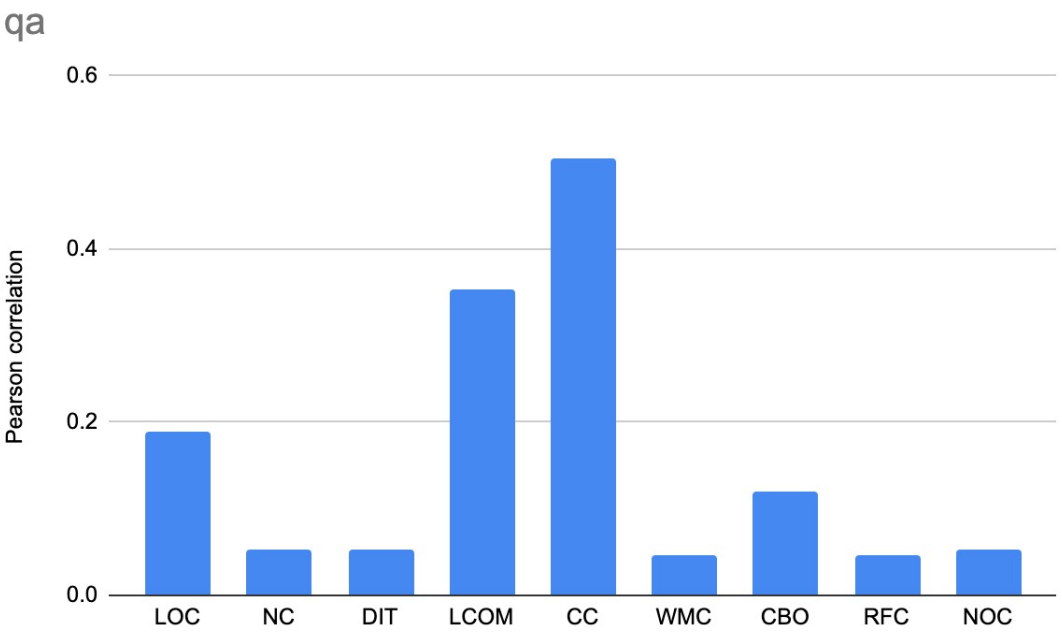

Figure 5.2: qa module correlation

us to participate in project development.

- Non-linear correlation research, currently only using Pearson correlation analysis. When our study finally analyzed the correlation between metric and MTTR, only the Pearson correlation analysis was carried out. This has further research space. We guess that the relationship between the metric and MTTR is likely to be logarithmic. If we want to understand the principle behind this particular phenomenon, we still need to invest a lot of work in the future to carry out more in-depth research. But for those linearly related metrics, our results are correct to a certain extent.

- Because manual testing cannot complete a large amount of data at one time, we only classify the files modified by each version of the release according to modules, which may be calculated according to the loc and file, and the results will be more accurate. In our research, some files may have changed only a few lines of code, but the records show that it took a relatively long time. Although we have carried out operations to remove extreme data, the calculation and collection of MTTR at the java file level can undoubtedly help to further remove extreme data.

- Although we have adopted the literature review method to obtain hundreds of related metrics, due to time and workload constraints, we only selected the most frequent ten metrics and did not involve more metrics, and these may be more effective.

- The modified bug file may also have some new feature changes, and the metric changes caused by these changes should not be counted. The solution to this problem is very limited, and our current guess is that all files that have been repaired because of bugs I have to find out, and then find some files that have not changed because of bug fixes, and remove these files. The remaining files are those that have only been modified by bug fixes. This method has two disadvantages: first, it requires a larger amount of manual calculation, and if this step cannot be automated, it is almost impossible; second, it requires a larger amount of data, and the period of the study will change It takes longer, and the selected project also needs a large number of bug fixes in each version to meet the data requirements.

- Regarding the acquisition software of metrics, because the acquisition of metrics is based on the relevant test software, our selected metrics test software is JavaDesignate and CKJM. For the selected metrics, we have one metric not obtained (NOBC). Our results are not precise and perfect. Later work can introduce more software to improve our conclusions.

Chapter 6

Conclusions and Future Work

In this thesis, we validated a subset of commonly proposed source code metrics for predicting maintainability. We conducted a retrospective study of a large-scale open source software and verified that some metrics have a strong correlation with the performance of software maintainability. This study provides important empirical evidence towards a better understanding of source code attributes and maintainability in practice. For our research questions, we collected and sorted out 291 related metrics through literature reviews. Although in this article, we have selected only the eight most frequently occurring metrics, all of these metrics have been proposed to be relevant to software maintainability. In addition, the results of the retrospective study finally showed that for Elasticsearch software, the values of the four metrics of LOC, CC, WMC and RFC are the most relevant to the performance of maintainability, while the metrics of NC, DIT, LOCM, CBO and the maintainability The performance correlation is very small, DIT has the smallest correlation, and the correlation index is only 0.2. That is, for this RQ, according to our results, the most useful metrics for our maintainability are LOC, CC, WMC, and RFC. However, research on a piece of software is not enough to verify the universality of software code internal measurement and software maintainability. Therefore, we need further research and analysis of more software to improve the confidence of our experimental conclusions in order to improve the practicality of the proposed metrics. In terms of assessing the maintainability of the object-oriented degree, although the article[38] mentions the mean time to repair (MTTR), mean time to repair (MCMT), mean time to preventive repair (MPMT) and the longest corrective repair time (MaCMT). By discovering the definition of maintainability and understanding of these definitions, we believe that MTTR is more in line with our requirements and can better measure maintainability.

6.1 Limitations and Future Work

For the involvement and implementation of our retrospective study, we also encountered the following problems and deficiencies.

For our calculation of the maintainability indicator: MTTR, it can be calculated more accurately. According to the definition of MTTR, the calculation unit of MTTR is "h," but because we are calculating manually, we use the "bug fix" corresponding to the GitHub software. "The time recorded in the log is calculated. There are two reasons for this. One is the problem of time recording. The developer does not modify the record in real-time as the update time but uploads the recording time.

Have a certain impact. The second is that even if the upload time is accurate, we cannot calculate the time in the "release note" very accurately. The time unit of the GitHub document update record uses days as the unit, but in fact, many releases may be 16h or 24h on the same day. These seemingly few errors have reached 30, 40%, which will affect our results. Therefore, in the future work, using "hour" as the minimum unit when calculating MTTR can reduce the error (in the case of data with hour units).

Regarding the problem of data collection, because we did not find relevant papers or literature on automatic research this time, we used manual testing. We can also see in our chapter "result" that our results collection is quite large, We can use automated data collection to improve our problems, we can expand the scope to all versions of multiple software, and study the general law, not the law for a single software. Therefore, we should expand the content of the retrospective study and add more programming language software or more versions to get more perfect constraints.

Regarding the correlation algorithm, currently, only Pearson correlation analysis is used. Based on the limitations of the Pearson algorithm, the correlation analysis of our paper this time can only perform linear analysis on the data, and we have not performed on the nonlinear model. Analysis and summary. Therefore, in the following work, the analysis framework should be further improved, and more accurate results should be obtained.

Regarding the choice of data level, this problem is also caused by manual testing. Because it is a manual collection of sentences, we cannot complete a large amount of data at one time, so for the files modified by each version of the release, we follow the modules It is classified and calculated according to loc and file, so if automatic data collection is used, we can analyze the results at other different levels, and the results will be more accurate.

Regarding the choice of metrics, although we adopted the literature review method to obtain hundreds of related metrics, due to time and workload constraints, we only selected the most frequent ten metrics and did not involve more metrics, and these other metrics may also have correlation and better results, which may be more effective.

Regarding the acquisition software of metrics, because the acquisition of metrics is based on the relevant test software, our selected metrics test software is JavaDesignate and CKJM. For the selected metrics, we have one metric not obtained (NOBC). Our results are not precise and perfect. Later work can introduce more software to improve our conclusions.

Finally, in addition to the problem of the shortcomings of our current retrospective study, we can also take our study one step closer. Regarding our study, we only analyzed and studied the correlation between metrics and maintainability. We only used Pearson's algorithm. Analyzed the correlation between the two (linear correlation), which means the magnitude of the correlation between the two about y = kx + b, but we did not use linear regression to develop models that predict MTTR. In the following work, We may have more research on this part, so that we can directly calculate the final value of MTTR or other related maintainability measures based on the results of the metrics.

References

[1] Coupling metrics for aspect-oriented programming: A systematic review of maintainability studies. In *Proceedings of the 4th International Conference on Evaluation of Novel Approaches to Software Engineering*. SciTePress - Science and and Technology Publications, 2009.

[2] ISO/IEC/IEEE International Standard - Systems and software engineering – Vocabulary. *ISO/IEC/IEEE 24765:2010(E)*, pages 1–418, December 2010.

[3] A SYSTEMATIC LITERATURE REVIEW ON SOFTWARE PRODUCT LINE QUALITY. In *Proceedings of the 6th International Conference on Software and Database Technologies*. SciTePress - Science and and Technology Publications, 2011.

[4] Refactoring business process models - a systematic review. In *Proceedings of the 7th International Conference on Evaluation of Novel Approaches to Software Engineering*. SciTePress - Science and and Technology Publications, 2012.

[5] Mean time to repair, April 2019. Page Version ID: 894653864.

[6] Pearson correlation coefficient, December 2019. Page Version ID: 933310309.

[7] Ramon Abilio, Pedro Teles, Heitor Costa, and Eduardo Figueiredo. A systematic review of contemporary metrics for software maintainability. In *2012 Sixth Brazilian Symposium on Software Components, Architectures and Reuse*. IEEE, sep 2012.

[8] D. Alic, S. Omanovic, and V. Giedrimas. Comparative analysis of functional and object-oriented programming. In *2016 39th International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO)*, pages 667–672, 2016.

[9] Khalid Alkharabsheh, Yania Crespo, Esperanza Manso, and José A. Taboada. Software design smell detection: a systematic mapping study. *Software Quality Journal*, 27(3):1069–1148, oct 2018.

[10] Hadeel Alsolai and Marc Roper. A systematic review of feature selection techniques in software quality prediction. In *2019 International Conference on Electrical and Computing Technologies and Applications (ICECTA)*. IEEE, nov 2019.

[11] Apostolos Ampatzoglou and Ioannis Stamelos. Software engineering research for computer games: A systematic review. *Information and Software Technology*, 52(9):888–901, sep 2010.

[12] Marco Antônio Pereira Araújo, Vitor Faria Monteiro, and Guilherme Horta Travassos. Towards a model to supportin silicostudies of software evolution. In *Proceedings of the ACM-IEEE international symposium on Empirical software engineering and measurement - ESEM '12*. ACM Press, 2012.

[13] Maria Teresa Baldassarre, Danilo Caivano, Simone Romano, and Giuseppe Scanniello. Software models for source code maintainability: A systematic literature review. In *2019 45th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*. IEEE, aug 2019.

[14] Nelson Barbosa and Kechi Hirama. Assessment of software maintainability evolution using c&k metrics. *IEEE Latin America Transactions*, 11(5):1232–1237, sep 2013.

[15] Camila Bastos, Paulo Afonso Junior, and Heitor Costa. Técnicas para detecção de código morto: Uma revisão sistemática de literatura. In *Anais do Simpósio Brasileiro de Sistemas de Informação (SBSI)*. Sociedade Brasileira de Computação, may 2016.

[16] Aloisio Cairo, Glauco Carneiro, and Miguel Monteiro. The impact of code smells on software bugs: A systematic literature review. *Information*, 9(11):273, nov 2018.

[17] Ivica Crnkovic, Ivano Malavolta, Henry Muccini, and Mohammad Sharaf. On the use of component-based principles and practices for architecting cyber-physical systems. In *2016 19th International ACM SIGSOFT Symposium on Component-Based Software Engineering (CBSE)*. IEEE, apr 2016.

[18] Surender Singh Dahiya, Jitender Kumar Chhabra, and Shakti Kumar. Use of genetic algorithm for software maintainability metrics' conditioning. In *15th International Conference on Advanced Computing and Communications (ADCOM 2007)*, pages 87–92, Guwahati, Assam, India, December 2007. IEEE.

[19] Jakob Danielsson, Nandinbaatar Tsog, and Ashalatha Kunnappilly. A systematic mapping study on real-time cloud services. In *2018 IEEE/ACM International Conference on Utility and Cloud Computing Companion (UCC Companion)*. IEEE, dec 2018.

[20] Jane Dirce Alves Sandim Eleuterio, Felipe Nunes Gaia, Andrea Bondavalli, Paolo Lollini, Genaina Nunes Rodrigues, and Cecilia Mary Fischer Rubira. On the dependability for dynamic software product lines: A comparative systematic mapping study. In *2016 42th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*. IEEE, aug 2016.

[21] Sara Elmidaoui, Laila Cheikhi, and Ali Idri. Software product maintainability prediction: A survey of secondary studies. In *2017 4th International Conference on Control, Decision and Information Technologies (CoDIT)*. IEEE, apr 2017.

[22] Sara Elmidaoui, Laila Cheikhi, and Ali Idri. Towards a taxonomy of software maintainability predictors. In *Advances in Intelligent Systems and Computing*, pages 823–832. Springer International Publishing, 2019.

[23] Sara Elmidaoui, Laila Cheikhi, Ali Idri, and Alain Abran. Empirical Studies on Software Product Maintainability Prediction: A Systematic Mapping and Review. *e-Informatica Vol. XIII*, page 2019; ISSN 18977979, 2019.

[24] Sinan Eski and Feza Buzluca. An Empirical Studyon Object-Oriented Metrics and Software Evolution in Order to Reduce Testing Costs by Predicting Change-Prone Classes. In *2011 IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops*, pages 566–571, Berlin, Germany, March 2011. IEEE.

[25] Eduardo Fernandes, Johnatan Oliveira, Gustavo Vale, Thanis Paiva, and Eduardo Figueiredo. A review-based comparative study of bad smell detection tools. In *Proceedings of the 20th International Conference on Evaluation and Assessment in Software Engineering - EASE '16*. ACM Press, 2016.

[26] Ana M. Fernández-Sáez, Marcela Genero, and Michel R.V. Chaudron. Empirical studies concerning the maintenance of UML diagrams and their use in the maintenance of code: A systematic mapping study. *Information and Software Technology*, 55(7):1119–1142, jul 2013.

[27] Gabriel Alberto García-Mireles, Mª Ángeles Moraga, Félix García, Coral Calero, and Mario Piattini. Interactions between environmental sustainability goals and software product quality: A mapping study. *Information and Software Technology*, 95:108–129, mar 2018.

[28] Gabriel Alberto García-Mireles, Ma Ángeles Moraga, Félix García, and Mario Piattini. Approaches to promote product quality within software process improvement initiatives: A mapping study. *Journal of Systems and Software*, 103:150–166, may 2015.

[29] Israr Ghani, Wan M.N., and Ahmad Mustafa. Web service testing techniques: A systematic literature review. *International Journal of Advanced Computer Science and Applications*, 10(8), 2019.

[30] Miguel Goulão, Vasco Amaral, and Marjan Mernik. Quality in model-driven engineering: a tertiary study. *Software Quality Journal*, 24(3):601–633, jun 2016.

[31] Milena Guessi, Valdemar V. G. Neto, Thiago Bianchi, Katia R. Felizardo, Flavio Oquendo, and Elisa Y. Nakagawa. A systematic literature review on the description of software architectures for systems of systems. In *Proceedings of the 30th Annual ACM Symposium on Applied Computing - SAC '15*. ACM Press, 2015.

[32] IEE. Iso/iec/ieee international standard - systems and software engineering – vocabulary. *ISO/IEC/IEEE 24765:2010(E)*, pages 1–418, 2010.

[33] Ronald Jabangwe, Jürgen Börstler, Darja Šmite, and Claes Wohlin. Empirical evidence on the link between object-oriented measures and external quality attributes: a systematic literature review. *Empirical Software Engineering*, 20(3):640–693, mar 2014.

[34] Li jin Wang, Xin xin Hu, Zheng yuan Ning, and Wen hua Ke. Predicting object-oriented software maintainability using projection pursuit regression. In *2009 First International Conference on Information Science and Engineering*. IEEE, 2009.

[35] Ahmad Kayed, Nael Hirzalla, Ahmad A. Samhan, and Mohammed Alfayoumi. Towards an Ontology for Software Product Quality Attributes. In *2009 Fourth International Conference on Internet and Web Applications and Services*, pages 200–204, Venice/Mestre, Italy, 2009. IEEE.

[36] Rüdiger Lincke, Tobias Gutzmann, and Welf Löwe. Software quality prediction models compared. In *2010 10th International Conference on Quality Software*. IEEE, jul 2010.

[37] Leszek A. Maciaszek, César González-Pérez, and Stefan Jablonski, editors. *Evaluation of Novel Approaches to Software Engineering*. Springer Berlin Heidelberg, 2010.

[38] Ruchika Malhotra and Anuradha Chug. Software maintainability: Systematic literature review and current trends. *International Journal of Software Engineering and Knowledge Engineering*, 26(08):1221–1253, oct 2016.

[39] Ruchika Malhotra, Megha Khanna, and Rajeev R. Raje. On the application of search-based techniques for software engineering predictive modeling: A systematic review and future directions. *Swarm and Evolutionary Computation*, 32:85–109, feb 2017.

[40] Abid Mehmood and Dayang N.A. Jawawi. Aspect-oriented model-driven code generation: A systematic mapping study. *Information and Software Technology*, 55(2):395–411, feb 2013.

[41] Sonia Montagud, Silvia Abrahão, and Emilio Insfran. A systematic review of quality attributes and measures for software product lines. *Software Quality Journal*, 20(3-4):425–486, aug 2011.

[42] Karl Pearson. *The grammar of science.* Walter Scott Publishing Co, 1892.

[43] Dirk Pfluger, Miriam Mehl, Julian Valentin, Florian Lindner, David Pfander, Stefan Wagner, Daniel Graziotin, and Yang Wang. The scalability-efficiency/maintainability-portability trade-off in simulation software engineering: Examples and a preliminary systematic literature review. In *2016 Fourth International Workshop on Software Engineering for High Performance Computing in Computational Science and Engineering (SE-HPCCSE)*. IEEE, nov 2016.

[44] Saul Melchor Ramirez, Karen Cortes, Jorge Octavio Ocharan-Hernandez, and Angel Juan Sanchez Garcia. Software stability: A systematic literature review. In *2018 6th International Conference in Software Engineering Research and Innovation (CONISOFT)*. IEEE, oct 2018.

[45] Mehwish Riaz, Emilia Mendes, and Ewan Tempero. A systematic review of software maintainability prediction and metrics. In *2009 3rd International Symposium on Empirical Software Engineering and Measurement*. IEEE, oct 2009.

[46] Neil Salkind. *Encyclopedia of Research Design*. SAGE Publications, Inc., 2455 Teller Road, Thousand Oaks California 91320 United States, 2010.

[47] Danilo Santos, Antonio Resende, Paulo Afonso Junior, and Heitor Costa. Attributes and metrics of internal quality that impact the external quality of object-oriented software: A systematic literature review. In *2016 XLII Latin American Computing Conference (CLEI)*, pages 1–12, Valparaíso, Chile, October 2016. IEEE.

[48] Danilo Santos, Antonio Resende, Paulo Afonso Junior, and Heitor Costa. Attributes and metrics of internal quality that impact the external quality of object-oriented software: A systematic literature review. In *2016 XLII Latin American Computing Conference (CLEI)*. IEEE, oct 2016.

[49] Juliana Saraiva, Sergio Soares, and Fernando Castor. Towards a catalog of Object-Oriented Software Maintainability metrics. In *2013 4th International Workshop on Emerging Trends in Software Metrics (WETSoM)*, pages 84–87, San Francisco, CA, USA, May 2013. IEEE.

[50] Juliana Saraiva, Sergio Soares, and Fernando Castor. Towards a catalog of object-oriented software maintainability metrics. In *2013 4th International Workshop on Emerging Trends in Software Metrics (WETSoM)*. IEEE, may 2013.

[51] Tushar Sharma. Designitejava, December 2018. https://github.com/tushartushar/DesigniteJava.

[52] Tushar Sharma. DesigniteJava (Enterprise), September 2019.

[53] D. Spinellis. Tool Writing: A Forgotten Art? *IEEE Software*, 22(4):9–11, July 2005.

[54] M. Sánchez. Assessing the quality of mooc using iso/iec 25010. In *2016 XI Latin American Conference on Learning Objects and Technology (LACLO)*, pages 1–4, 2016.

[55] Amjed Tahir and Rodina Ahmad. An AOP-Based Approach for Collecting Software Maintainability Dynamic Metrics. In *2010 Second International Conference on Computer Research and Development*, pages 168–172, Kuala Lumpur, Malaysia, 2010. IEEE.

[56] P. P. Texel. Measure, metric, and indicator: An object-oriented approach for consistent terminology. In *2013 Proceedings of IEEE Southeastcon*, pages 1–5, 2013.

[57] Aparna Vegendla, Anh Nguyen Duc, Shang Gao, and Guttorm Sindre. A systematic mapping study on requirements engineering in software ecosystems. *Journal of Information Technology Research*, 11(1):49–69, jan 2018.

[58] Yuming Zhou and Hareton Leung. Predicting object-oriented software maintainability using multivariate adaptive regression splines. *Journal of Systems and Software*, 80(8):1349–1361, aug 2007.

[59] F. Zhuo, B. Lowther, P. Oman, and J. Hagemeister. Constructing and testing software maintainability assessment models. In *[1993] Proceedings First International Software Metrics Symposium*. IEEE Comput. Soc. Press.

Appendix A

Supplemental Information

Authors	Title	Year
Guessi M., Neto V.V.G., Bianchi T., Felizardo K.R., Oquendo F., Nakagawa E.Y.	A systematic literature review on the description of software architectures for systems of systems [31]	2015
Mehmood A., Jawawi D.N.A.	Aspect-oriented model-driven code generation: A systematic mapping study [40]	2013
Fernández-Sáez A.M., Genero M., Chaudron M.R.V.	Empirical studies concerning the maintenance of UML diagrams and their use in the maintenance of code: A systematic mapping study [26]	2013
Fernández-Ropero M., Pérez-Castillo R., Piattini M.	Refactoring business process models: A systematic review	2012
Araújo M.A.P., Monteiro V.F., Travassos G.H.	Towards a model to support in silico studies of software evolution [12]	2012
Moraga C., Moraga M.A., Genero M., Piattini M.	A systematic literature review on software product line quality [3]	2011
Gabriel P., Goulão M., Amaral V.	Do software languages engineers evaluate their languages?	2010
[No author name available]	Evaluation of Novel Approaches to Software Engineering - 3rd and 4th International Conferences, ENASE 2008/2009, Revised Selected Papers [1]	2010
Ampatzoglou A., Stamelos I.	Software engineering research for computer games: A systematic review [11]	2010
Baldassarre M.T., Caivano D., Romano S., Scanniello G.	Software Models for Source Code Maintainability: A Systematic Literature Review [13]	2019
Subramaniam H., Zulzalil H.	Software quality assessment using flexibility: A systematic literature review	2012
Céspedes D., Angeleri P., Melendez K., Dávila A.	Software Product Quality in DevOps Contexts: A Systematic Literature Review	2020
Alsolai H., Roper M.	A Systematic Review of Feature Selection Techniques in Software Quality Prediction [10]	2019

Alkharabsheh K., Crespo Y., Manso E., Taboada J.A.	Software Design Smell Detection: a systematic mapping study [9]	2019
Ramirez S.M., Cortes K., Ocharan-Hernandez J.O., Sanchez Garcia A.J.	Software stability: A systematic literature review [44]	2019
Danielsson J., Tsog N., Kunnappilly A.	A systematic mapping study on real-Time cloud services [19]	2019
Ghani I., Wan-Kadir W.M.N., Mustafa A.	Web service testing techniques: A systematic literature review [29]	2019
Cairo A.S., Carneiro G.F., Monteiro M.P.	The impact of code smells on software bugs: A systematic literature review [16]	2018
[No author name available]	Proceedings - International Conference on Software Engineering	2018
Vegendla A., Duc A.N., Gao S., Sindre G.	A systematic mapping study on requirements engineering in software ecosystems [57]	2018
Pflüger D., Mehl M., Valentin J., Lindner F., Pfander D., Wagner S., Graziotin D., Wang Y.	The scalability-efficiency/maintainability-portability trade-off in simulation software engineering: Examples and a preliminary systematic literature review [43]	2017
[No author name available]	Proceedings of SE-HPCCSE 2016: 4th International Workshop on Software Engineering or High Performance Computing in Computational Science and Engineering - Held in conjunction with SC 2016: The International Conference for High Performance Computing, Networking, Storage and Analysis	2017
Eleuterio J.D.A.S., Gaia F.N., Bondavalli A., Lollini P., Rodrigues G.N., Rubira C.M.F.	On the Dependability for Dynamic Software Product Lines: A Comparative Systematic Mapping Study [20]	2016
Goulão M., Amaral V., Mernik M.	Quality in model-driven engineering: a tertiary study [30]	2016
Crnkovic I., Malavolta I., Muccini H., Sharaf M.	On the Use of Component-Based Principles and Practices for Architecting Cyber-Physical Systems [17]	2016
Fernandes E., Oliveira J., Vale G., Paiva T., Figueiredo E.	A review-based comparative study of bad smell detection tools [25]	2016
Sultan A.B.M., Bakar A.D., Zulzalil H., Din J.	Systematic literature review in open source software maintainability	2012

Faculty of Computing, Blekinge Institute of Technology, 371 79 Karlskrona, Sweden