Assessing Software Maintainability Based on Class Diagram Design: A Preliminary Case Study

Yao Lu, Xinjun Mao, and Zude Li

Abstract—Can software maintainability be assessed at the early design stage? For a preliminary answer, we conducted a case study. The study adopts a set of metrics for class diagram measurement, and defines three indices for maintainability assessment from the defect-correction perspectives. The dataset under investigation includes the defect repository and corrective maintenance history of Apache Tomcat (maintained from 2006 to 2014). Statistical findings show that some class diagram metrics (such as the number of class association across packages, the number of classes, the inheritance depth for a class et al.) are significantly correlated with the maintainability assessment in this software. The result can guide maintenance-oriented software design, and also motivates us to do a stronger empirical evaluation.

Index Terms—Software maintainability, class diagram metrics, change history.

I. INTRODUCTION

It is known that maintaining a mature software system can consume more than 90% of its lifecycle costs [1], and that costly maintenance usually leads to user satisfaction decreasing [2]. From both the economic and the user-satisfaction perspectives, the system shall stay within a good-enough maintenance status during its lifecycle. For that, a best practice is: first create and implement a maintenance-oriented system design, and then try best to avoid or retard design erosion during maintenance. Our research focus is on the first part, i.e., to assess the system maintainability at the early design stage, in order to guide the improvement of software design. Before that, we have to build approaches to correctly assessing software maintainability based on design artifacts.

Today’s software design usually adopts object-oriented (OO) techniques, in which class diagram modeling is the most frequently used one. A class diagram describes the structure of a system by defining a set of interconnected OO classes, their attributes and operations (methods). This structure represents the conceptual design of the system, which largely determines the maintainability in code. Therefore, class diagram offers a means to assessing the maintainability of OO systems at design stage. A few studies [3]-[13] have been conducted, and most of them are survey-based controlled experiments, which use participants’ subjective opinions and some maintainability indices (e.g., the time for understanding a class diagram) to assess the maintainability of class diagram. One issue with these studies is that: the system maintainability is assessed by participants’ opinions, which can be assessed objectively on design and code artifacts. In fact, such objective studies could add credits to the findings of existing survey-based subjective studies. Motivated by that, we conducted an explorative case study, to objectively assess the maintainability of a system based on the class diagram.

In this case study, we follow the Goal-Question-Metric modeling process to define a set of metrics for class diagram measurement, and define three indices for maintainability assessment. These maintainability indices examine three aspects of defect correction, respectively: time span, number of modified lines of code, and physical impact span. The dataset under investigation includes the defect repository and corrective maintenance history of Apache Tomcat releases 6.0-8.0.

At the late study stage, we conducted statistical analysis, which affirms the significant correlation between the class diagram measurement and maintainability assessment. Particularly, the findings show that: 1) more across-package class associations, more children of a class, and lower class inheritance depth indicate bigger time span for a defect correction; 2) more classes and lower class inheritance depth indicate more source lines of code modified and bigger impact span for a defect correction. The findings can guide software design and maintenance practice. For example, it is better to control the number of class association (i.e., coupling) across packages, and reduce the number of classes and the number of children of a class, towards a more maintainable system.

The paper is structured as follows. Section II discusses related work. Section III describes the case study design. Section IV gives and interprets the findings of the study. Section V analyzes the implications of the findings. Section VI discusses the threats to validity of the study. Finally, Section VII concludes the paper and describes the future work.

II. RELATED WORK

OO software maintainability can be assessed at both code level and at design level. This section outlines existing literatures on maintainability assessment based on code and design artifacts.

A. Code-Level Metrics for OO Software Maintainability

Henry and Wake proposes a multiple regression model based on the defect correction history of a software system [14]. This model described the equation between complexity metrics and maintainability, which can be used to assess the
system’s maintainability during all the life-circle. Li and Henry [15] conducted a case study on two industrial systems written in Ada, and found that software size is an important predictor for maintenance effort. Using the measurement data of Li and Henry’s study, Van Koten and Gray [16] defines a Bayesian network maintainability prediction model, which has higher accuracy than regression-based model. Dagpinar and Jahnke [17] conducted an empirical study on two OO systems, and found that the size and import direct coupling metrics are significant predictors for the maintainability of OO classes, but the inheritance, cohesion, and indirect/export coupling metrics are not. Lim et al. [18] collected maintenance data from two real-world mission-critical systems -one NOO (non-object oriented) written in C and the other OO written in C++. They found that the OO group took less time to maintain software artifacts than the NOO group, the reason for which they thought were due to the usefulness of UML for impact analysis of the OO version. Orenyi et al. presented an extensive review on OO software maintainability measurement in the past decade [19]. They found that Regression Analysis methods have been predominantly used in the development of OO software maintainability models, and suggested that maintainability measurement model should use objective measurement method and Software Quality Models should be encouraged.

**B. Design-Level Metrics for OO Software Maintainability**

OO software maintainability can be assessed at the early design stage, but there are fewer work done on this topic and no consolidated model has been presented yet [6]. In a series of studies [3-6], Genero et al. proposed three size metrics for OO classes and eight metrics for OO design. They conducted two controlled experiments with university teachers and students. Using fuzzy regression and decision tree, they found that metrics concerning OO class aggregation and generalization relationships are highly related to the understanding time, modification correctness and completeness of system.

Zhou and Xu investigated the relationships between 15 design metrics and maintainability of 148 Java open source software [8]. They found that the size and complexity metrics are strongly related to the software maintainability, while class cohesion and coupling are not. In another study, they proposed a method to transform a class diagram into a weighted class dependence graph, and presented a structure complexity measurement for class diagrams based on entropy distance [9]. Yi and Wu conducted a controlled experiment which is similar to Genero’s. The study confirms that Zhou’s metric is significantly correlated with human beings’ intuition [10]. They further compared two typical class-diagram complexity metrics based on statistics and entropy-distance [11]. They investigated the understandability and analyzability of 27 class diagrams by means of algorithm C5.0 and found that metric based on statistics has higher classification accuracy than that based on entropy distance.

Alshayeb [12] conducted an empirical study to evaluate the relationship between four stability metrics and indices of maintenance effort and found that classes with higher values of Class Stability Metrics (CSM) are associated with lower values of perfective maintenance effort measured by hours, while none of the stability metrics is correlated with maintainability measured by the number of changed lines.

### III. CASE STUDY DESIGN

**A. GQM Model**

The case study is organized with the Goal-Question-Metric (GQM) paradigm [20], which is a well-known approach and has been used in a number of empirical studies [21]. First, we define the goal below:

**Goal:** To assess the software maintainability based on class diagram design.

The question corresponding to this goal is: *What factors of class diagram would associate software maintainability?*

This question is further refined into three sub-questions:

**Sub-question 1:** Does large scale class diagram indicate low maintainability?

This question examines the correlation between class diagram size and software maintainability. It is associated with three metrics:

1. **NC:** The total number of classes in the class diagram;
2. **NA:** The total number of class attributes;
3. **NM:** The total number of class methods.

**Sub-question 2:** Does strong inter-class coupling indicate low maintainability?

This question examines the correlation between class coupling strength and software maintainability. It is associated with four metrics:

1. **NAssIP:** The total number of class associations in the same packages;
2. **NAssAP:** The total number of class associations across packages;
3. **NC_Attr:** The total number of coupling relationships which one class is used as an attribute of another class;
4. **NC_Par:** The total number of coupling relationships which one class or interface is used as a parameter type of another class.

**Sub-question 3:** Does complex class inheritance indicate low maintainability?

This question examines the correlation between class inheritance and software maintainability. It is associated with two metrics:

1. **NCh:** The total number of children of each class;
2. **NDIT:** The total number of inheritance depth of each class.

**TABLE 1: THREE EXAMPLE DEFECT RECORDS**

<table>
<thead>
<tr>
<th>ID</th>
<th>Version</th>
<th>Importance</th>
<th>Report Time</th>
<th>Committing Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>50306</td>
<td>7.0.4</td>
<td>enhancement</td>
<td>2010/11/19 18:09</td>
<td>2012/6/20 8:55</td>
</tr>
<tr>
<td>55500</td>
<td>7.0.34</td>
<td>critical</td>
<td>2013/08/29 12:21</td>
<td>2013/09/3 9:38</td>
</tr>
</tbody>
</table>

It is worth noting that there are some subtle differences between NCh and NDIT. For instance, as shown in Fig. 1, the

---

1. The associations in this study include all kinds of associations such as plain, aggregate, composite associations et al.
B. Data-Set Description

In this study, we investigate the whole corrective maintenance history (more than 7 years) of Apache Tomcat project from release 6.0.0 to 8.0.5. Tomcat is an open-source web server written in Java. It has been managed with SVN and Bugzilla since 2006 from release 6.0.0, and has 100 versions and 825 valid defect-correction records so far.

Table I shows three real defect records (key attributes only). For example, the first record is interpreted as below: defect #43944 occurred in the Version 6.0.14. It was reported as the normal-level defect at 19:23 on Nov. 22th 2007, and committed at 21.22 on Dec. 28th. We exclude 53 versions from the 100 ones, because their valid defect-correction records are less (<5) for statistical analysis.

![Fig. 1. Counting methods for NDIT and NCh.](image)

C. Process Description

As Fig. 2 shows, the case study has three main steps: maintainability assessment, class diagram measurement and statistical analysis. These steps are described as below:

![Fig. 2. Experiment process.](image)

1) Maintainability Assessment. We assess software maintainability with the following three indices:

**Average Physical Impact Span (APIS)** is defined to assess the impact scope of change for a tag version. It counts the average number of OO code files (such as .java, .sh, .bin) and packages fixed for a defect correction. It is calculated below:

\[
APIS = \frac{\sum_{i=1}^{n}(NCF + NCP)}{NR}
\]

where, \(NCF\) is the number of changed code files, \(NCP\) is the number of changed OO packages, and \(NR\) is the total number of defect correction records.

**Average Time Span for a Defect Correction (ATSDC)** approximately assesses the time span for a defect correction in a version. It is calculated below:

\[
ATSDC = \frac{\sum_{i=1}^{n}(CT - RT)}{NR}
\]

where, \(CT\) is the committing time recorded in the SVN change log, and \(RT\) is the report time recorded in the Bugzilla.

**Average number of Source Lines of code for a Defect Correction (ASLDC)** counts the valid changed code lines to fix a defect for a version. The valid changed lines here is calculated by the added lines and the modified lines, without counting the blank lines, comment lines and configuration file lines. We developed a tool SVNCLCounter which extract SVN diff file according to the inputted revision number and counts the changed lines based on the notation in front of the lines.

2) Class Diagram Measurement. To get the class-level metrics, we first use Enterprise Architect to reverse the java code of each versions to class diagrams. Enterprise Architect is a CASE (Computer Aid Software Engineering) tool which can reverse source code to class diagram of UML 2.0. After the reversing process, we export the class diagrams to xmi files. Having the xmi files as input, we use SDMetrics to caculate the value of the metrics.

3) Statistical Analysis. To study the correlationship between the class diagram metrics and the dependent maintainability indices, we analyze the data in the following two methods:

**Partial Correlation Analysis (PCA).** For the maintainability indexes are influenced by many factors, we try to exclude the influence of the other variables when considering the correlation between two variables. We use PCA to analyze the correlation between each class-diagram metrics and each maintainability indices with controlling the other metrics’ influence. In this paper, the significance level is set at \(\alpha = 0.05\).

**Canonical Correlation Analysis (CCA).** Compared with PCA analyzing a single maintainability indice and class-diagram metric, CCA is to analyze the correlation between two variable sets: the maintainability indice set and the class-diagram metric set. Its fundamental principle is to use the correlations between canonical varibles to reflect the correlations between the two variable-sets.
shown below.

A. Partial Correlation Analysis

In the results of partial analysis studying the relationship between a single class diagram metric and maintainability indice, we find that the correlation value between NAAss_AP and ATSDC is 0.323 and the p value is 0.048 (< 0.05). That means, the correlation relationship between NAAss_AP and ATSDC is statistically significant. Therefore, we can make an inference that:

Finding 1: The more across-package class associations in a class diagram, the bigger time span needed for a defect correction.

B. Canonical Correlation Analysis

The results of canonical correlation analysis are shown in TABLE II. The significance testing suggests that only the former two groups (U1-V1, U2-V2) of canonical variables are significantly correlated (p value>0.05) and their canonical correlations are 0.750 and 0.636. The standard canonical coefficients and canonical loadings reflect the correlations between the raw variables and canonical variables, we can see from Table II that for U1, NCh and NDIT have high correlations with their variates, while for V1 ATSDC is the dominating factor (presented in Fig. 3.), and NCh is positively correlated with ATSDC while NDIT is negatively correlated. That is to say,

Finding 2: The more children and lower inheritance depth for a class in a class diagram, the bigger time span needed for a defect correction.

In the same way, U2 is determined by NDIT and NC, and V2 is determined by APIS and ATLFB. Therefore, the following finding holds:

Finding 3: The more classes and lower inheritance depth for a class in a class diagram, the more code lines modified and more documents and packages fixed for a defect correction.

Redundancy coefficients can be used as a measure of possible predictive ability and the results shows that U1 and U2 can accumulatively explain 65.6 percent of the metric set, and V1 and V2 can accumulatively explain 79.1 percent of the maintainability indice set. As a result, the two canonical variables can explain the most of the raw variable sets while there are still a small part they cannot explain, the reason for which we think is that there are still some other factors we haven’t considered.

![Fig. 3. Relationship between key variables and canonical variates.](image)

V. IMPLICATIONS

A. Class Diagram Design

From the findings in Section IV, we can see that all the three factors have some correlations with maintainability indices: scale factors for NC, coupling factors for NAAssAP and inheritance factors for NDIT and NCh. The following subsections discuss the implications on class diagram design from these three factors.

1) Scale Factor. The findings show that NC is positively correlated with the average number of modified lines of code, which means that with other requirements satisfied, we should try to keep the number of classes small. For example, when two classes have the same or similar responsibility, we should consider merging the two classes. It is important to note that we should consider other factors and make tradeoffs. An example is that class A and B inherit from class C, although A and C have the similar responsibility, they cannot be merged into a class.

2) Coupling Factor. The findings show that NAAss_AP is positively correlated with average time span, which is consistent with the low coupling principle. If the value of NAAss_AP is excessive, it means that some mutually-associated classes are put in different packages. In this case, we should use Common Reuse Principle\(^3\) and refactor the classes, and put the classes closely connected into one package. Furthermore, another important principle is Common Closure Principle\(^4\), which tells us the package should not have more than one reason to change. If change were to happen in an application dependent on a number of packages, ideally we only want changes to occur in one package, rather than in a number of them. But in practice, it is impossible to keep every change to software inside one package. That is to say, we should try to keep the common changes we have experienced closed inside one package.

3) Inheritance Factor. We can see from the findings that NCh is positively correlated with average time span, while NDIT is negatively correlated with all three indices. So from this, we can see that increasing the depth of inheritance helps to improve maintainability from these three indices. It confirms the theory that the inheritance hierarchy should be as deep as possible, which helps new derived classes to derive from deeper parent classes. While in practice, the depth of an inheritance hierarchy should not exceed people’s short-term memory’s capacity, otherwise programmers would be ‘lost’ and requires much more time for understanding and modification. Empirically, the commonly accepting depth of inheritance is around 6, and practitioners can depend on their own situation to decide the proper inheritance depth.

B. Software Maintenance

In maintenance practice, maintainers should pay more attention to obeying the design principles while modifying the software and trying not to break the design. For example, they should control the impact scope of change. On the other hand, it is better to invite maintainers to participate in the design process, which guarantees the design is maintenance-oriented and helps maintainers are more aware

---

\(^3\)The classes in a package are reused together. If you reuse one of the classes in a package, you reuse them all [22].

\(^4\)Classes that change together, belong together [22].
of design architecture.

VI. THREATS TO VALIDITY

A. Data Quality

The data-set under investigation is the change logs and defect repository of an open source system. For the developers are more free and decentralized around the world, it is possible that some defects were fixed but were not recorded in the defect-tracking database and some changes were made in the code base but were not recorded in the version control system [21]. In addition, the information in the change logs is not fully accurate. For example, in some cases, we find that the same author committed two revisions for different defect correction in few minutes, which is possible that he made a mistake on the commit message in the first committing action.

B. Experiment Process

In the case study, the metric ATSDC reflects the time cost for a defect correction. However, as Section IV mentions, ATSDC is not the real time cost for a defect correction. Because ATSDC contains other time costs such as email communication time and sleeping time. On the other hand, the time span under calculation is an average value for a version, and in most cases the fixing records are more than 10, so it is reasonable to use it to reflect maintainability to some extent.

C. Tools

The study tried multiple reverse-engineering tools, and they outputted different results. Furthermore, these tools act differently. For example, Rational Software Architect would stop the reversing process if it meets an unrecognized notation, while Enterprise Architect would ignore. What’s more, ArgoUML are unable to extract interfaces while the others could. Even if they can, the results may be different. Similar phenomenons are found in [23]. As well as for metric tools, measurements are usually tool-dependent [24], and their usage could lead to discrepant implications.

VII. CONCLUSION AND FUTURE WORK

This paper describes a preliminary case study for assessing software maintainability based on class diagram design. The findings of the study conclude that the software maintainability in terms of time span, number of modified code lines, and impact span for a defect correction can be assessed based on class diagram measurement with size, coupling and inheritance metrics. This gives an insight into maintenance-oriented software design, also motivates us to do a stronger empirical evaluation.

In terms of future work, we will scale up this study in two dimensions. First, duplicate the case study process with more real software systems. Second, assess the maintainability based on the complete maintenance history (as in the current study only corrective maintenance records are examined). As mentioned in [11], relationships among different classes affect the complexity of class diagrams with different extents, so different types of associations should be examined for a stronger empirical evaluation.

ACKNOWLEDGMENT

We want to thank SDMetrics group for providing their excellent tool freely, and their kind help on the tool use.

REFERENCES

Yao Lu is a master's candidate of the College of Computer at National University of Defense Technology (NUDT), Changsha, China. He obtained his B.S. degree in computer science from the NUDT in 2012. His research interests are in the fields of software evolution, quality and software data analysis.

Xinjun Mao received the B.S. degree in computer engineering from the Institute of Engineering and Technology, Zhengzhou, China, in 1992, and the M.S. and Ph.D. degrees in computer science from the National University of Defense Technology (NUDT), Changsha, China, in 1995 and 1998, respectively.

He is a full professor and PhD supervisor with the College of Computer, National University of Defense Technology, Changsha, P.R. China. He was a visiting scholar with the Department of Computer Science, University of Toronto, Toronto, Canada, from 2003 to 2004. He was a lecturer, and an associate professor at National University of Defense Technology, Changsha, China, from 1998 to 2005. He has published more than 100 research papers, 2 books, and 2 book chapters. His research interests include agent theory and technology, agent-oriented software engineering, software architecture and pattern.

Zude Li is an associate professor at Central South University, China. He obtained his Ph.D. degree in 2010 from The Univ. of Western Ontario, Canada. His research interests are in the fields of software architecture, evolution and quality. He collaborates with IBM Canada and is appointed as a software expert in skane Soft and lionsun data.