Available online at www.sciencedirect.com
ScienceDirect
The Journal of Systems and Software 81 (2008) 1868–1882
The Journal of Systems and Software
www.elsevier.com/locate/jss

The effectiveness of software metrics in identifying error-prone classes in post-release software evolution process

Raed Shatnawi [a,*], Wei Li [b]

[a] Computer Information Systems, Jordan University of Science & Technology, Irbid 22110, Jordan
[b] Computer Science Department, The University of Alabama in Huntsville, Huntsville, AL 35899, United States

Received 18 September 2007; received in revised form 23 December 2007; accepted 27 December 2007
Available online 8 January 2008

Abstract

Many empirical studies have found that software metrics can predict class error proneness and the prediction can be used to accurately group error-prone classes. Recent empirical studies have used open source systems. These studies, however, focused on the relationship between software metrics and class error proneness during the development phase of software projects. Whether software metrics can still predict class error proneness in a system's post-release evolution is still a question to be answered. This study examined three releases of the Eclipse project and found that although some metrics can still predict class error proneness in three error-severity categories, the accuracy of the prediction decreased from release to release. Furthermore, we found that the prediction cannot be used to build a metrics model to identify error-prone classes with acceptable accuracy. These findings suggest that as a system evolves, the use of some commonly used metrics to identify which classes are more prone to errors becomes increasingly difficult and we should seek alternative methods (to the metric-prediction models) to locate error-prone classes if we want high accuracy.

Keywords: Object-oriented metrics; Class error proneness; Error-severity categories; Design evolution; Open source software; Empirical study

1. Introduction

A common problem in large and complex software systems is that they have errors (Myers et al., 2004). Preventing errors from being introduced into software systems proves to be a difficult, if not an impossible, task. If we cannot completely prevent errors, the next best thing that we can try is to find errors and remove them. Software testing, in conjunction with code inspection and walkthrough, is a widely used method to find errors; the debugging process that follows testing removes errors from a program (Myers et al., 2004). However, software testing is very costly in terms of time and resources. Any information that can help software testers focus their effort on the part of a system that is likely to have errors can increase the efficiency of testing. Measuring software quantitatively and using the measures to predict where errors are likely to occur in a system is one technique that can make the work of software testers more effective and efficient.

Software metrics have been proposed and empirically studied in the object-oriented (OO) paradigms. These empirical studies have studied the relationship between OO metrics and error proneness (Basili et al., 1996; Briand et al., 1998, 2000; Gyimothy et al., 2005; Subramanyam and Krishnan, 2003), maintenance effort (Li and Henry, 1993; Alshayeb and Li, 2003), design effort (Chidamber and Kemerer, 1994) and project progress (Li et al., 2000; Alshayeb and Li, 2005) during the development phase of the systems.

Basili et al. (1996) validated the CK metrics as predictors of class error proneness in a medium-sized management information system specification (Basili et al., 1996). The study showed that the CBO, RFC, WMC, DIT, and NOC metrics were significant predictors of class error proneness.

* Corresponding author.
E-mail addresses: raedamin@just.edu.jo (R. Shatnawi), wli@cs.uah.edu (W. Li).

0164-1212/$ - see front matter © 2008 Elsevier Inc. All rights reserved.
doi:10.1016/j.jss.2007.12.794

Chidamber et al. (1998) investigated the relationship between the CK metrics and various quality factors: software productivity, rework effort, and design effort. The study also showed that the WMC, RFC, CBO metrics were highly correlated. Therefore, Chidamber et al. did not include these three variables in the regression analysis to avoid generating coefficient estimates that would be difficult to interpret. The study concluded that there were associations between the high CBO metric value and lower productivity, more rework, and greater design effort. In another study, (Wilkie and Kitchenham, 2000) validated the relationship between the CBO metric and change ripple effect in a commercial multimedia conferencing system (Wilkie and Kitchenham, 2000). The study showed that the CBO metric identified the most change-prone classes, but not the classes that were most exposed to change ripple effect.

Cartwright and Shepperd (2000) also investigated the relationship between a subset of CK metrics in a real-time system. The study showed that the parts of the system that used inheritance were three times more error prone than the parts that did not use inheritance (Cartwright and Shepperd, 2000). Subramanyam and Krishnan (2003) validated the WMC, CBO, and DIT metrics as predictors of the error counts in a class in a business-to-consumer commerce system. Their results indicated that the CK metrics could predict error counts.

Alshayeb and Li (2003) conducted a study on the relationship between some OO metrics and the changes in the source code in two client–server systems and three Java Development Kit (JDK) releases. Three of the CK metrics (WMC, DIT, and LCOM) and three of the Li metrics (NLM, CTA, and CTM) were validated. Alshayeb and Li found that the OO metrics were effective to predict design effort and source lines of code added, changed, and deleted in short-cycled agile process (client–server systems); however, the metrics were ineffective predictors of those variables in long-cycled framework evolution process (JDK).

These studies shows that the use of metrics to predict design quality during development still equivocal. On the other hand, as systems evolve, significant amounts of resources must be dedicated to maintain the quality of the systems. It is not clear whether software metrics can still predict class error proneness with reasonable accuracy in the post-release system, because a post-release system has been through rigorous quality-assurance procedures (which include code inspections, walkthroughs, and testing) and they should have fewer errors than they had before the release. Research conducted by Alshayeb and Li in 2003 raised doubts that the metrics' predictive capabilities would carry from the development phase through the post-release evolution phase (Alshayeb and Li, 2003). Their research results, obtained from analyzing several releases of an open source system, indicated that software metrics could not predict source code changes in the post-release evolution of the system. Another issue that has not been addressed sufficiently by the previous research is whether software metrics can predict class error proneness if we split the errors into several severity categories.

Differentiating errors into different severity categories can help software engineers narrow down the areas in the design to focus their efforts of testing or refactoring. For example, if software testers have only a very limited amount of time left to conduct testing, knowing where the most severe errors are likely to occur in a system is certainly more helpful than just knowing where errors are likely to occur, because the knowledge can help them narrow down the testing areas further. Some previous studies used different error groups. For example, Basili et al. (1996) divided classes from several student projects into two groups: the group of classes that had errors and the group of classes that did not. Szabo and Khoshgoftaar (1995) classified functions/procedures from a system into three groups: High-, Medium-, and Low-risk groups. However, these studies grouped modules (classes or functions/procedures) based on the number of errors, not the severity of errors. Recently, Zhou and Leung (2006) used the Chidamber and Kemerer (C&K) metrics suite (Chidamber and Kemerer, 1994) to predict two levels of error severity in a storage management system. They used the error data (code named KC1) that were collected, managed, and posted by the Metrics Data Program repository at the National Aeronautics and Space Agency (NASA). The KC1 data were collected throughout the lifecycle of the project, but mainly from the development phase of the project.

Software systems continue to evolve after they are released. The post-release evolution process of a system is different from the development process because the system has been through (presumably) some rigorous quality assurance checks (e.g., inspections/walkthroughs and testing). Although the post-release systems tend to have fewer errors than the systems that are under development, they are still not free of errors. Software maintenance effort, which is what we spend to keep the post-release systems functioning properly, has been reported to be the largest share of the entire software cost; Estimates of the maintenance cost may range from 60% to 80% of all efforts expended by a development organization and the figure continues to rise (Hanna, 1993; McConnell, 1996; Hatton, 1998; Erlikh, 2000; Sommerville, 2007). The challenge of locating and fixing errors in the post-release systems still exists, if not more pressing than in the development phase. Knowing where the errors are likely to occur, or more importantly, where the errors of various severity categories are likely to occur in the post-release evolution of a system is equally important as the same knowledge in the development phase.

In this study, we intend to empirically investigate three aspects of using metrics to predict class error probability in the post-release evolution process. First, we want to know whether software metrics can predict class error probability (also known as error proneness in previous