Contents lists available at ScienceDirect

Information and Software Technology

journal homepage: www.elsevier.com/locate/infsof

An empirical analysis of package-modularization metrics: Implications for software fault-proneness

Yangyang Zhao [a,b], Yibiao Yang [a,b], Hongmin Lu [a,b], Yuming Zhou [a,b,*], Qinbao Song [c], Baowen Xu [a,b]

[a] *State Key Laboratory for Novel Software Technology, Nanjing University, China*
[b] *Department of Computer Science and Technology, Nanjing University, China*
[c] *Department of Computer Science and Technology, Xi'an Jiaotong University, China*

ARTICLE INFO

Article history:
Received 29 May 2014
Received in revised form 12 September 2014
Accepted 15 September 2014

Keywords:
Modularization
Metrics
Package
Fault-proneness
Prediction
Object-oriented

ABSTRACT

Context: In a large object-oriented software system, packages play the role of modules which group related classes together to provide well-identified services to the rest of the system. In this context, it is widely believed that modularization has a large influence on the quality of packages. Recently, Sarkar, Kak, and Rama proposed a set of new metrics to characterize the modularization quality of packages from important perspectives such as inter-module call traffic, state access violations, fragile base-class design, programming to interface, and plugin pollution. These package-modularization metrics are quite different from traditional package-level metrics, which measure software quality mainly from size, extensibility, responsibility, independence, abstractness, and instability perspectives. As such, it is expected that these package-modularization metrics should be useful predictors for fault-proneness. However, little is currently known on their actual usefulness for fault-proneness prediction, especially compared with traditional package-level metrics.
Objective: In this paper, we examine the role of these new package-modularization metrics for determining software fault-proneness in object-oriented systems.
Method: We first use principal component analysis to analyze whether these new package-modularization metrics capture additional information compared with traditional package-level metrics. Second, we employ univariate prediction models to investigate how these new package-modularization metrics are related to fault-proneness. Finally, we build multivariate prediction models to examine the ability of these new package-modularization metrics for predicting fault-prone packages.
Results: Our results, based on six open-source object-oriented software systems, show that: (1) these new package-modularization metrics provide new and complementary views of software complexity compared with traditional package-level metrics; (2) most of these new package-modularization metrics have a significant association with fault-proneness in an expected direction; and (3) these new package-modularization metrics can substantially improve the effectiveness of fault-proneness prediction when used with traditional package-level metrics together.
Conclusions: The package-modularization metrics proposed by Sarkar, Kak, and Rama are useful for practitioners to develop quality software systems.

1. Introduction

The last decades have seen a considerable increase of large-scale object-oriented software systems consisting of thousands of classes. In such a system, classes are inappropriate to be considered as units of software modularization [1]. Instead, it is common to use packages to group related classes together to provide well-identified services to the rest of the system [1]. In other words, packages indeed play the roles of modules, which enable developers to manage the complexity of a large-scale software system. The importance of packages has been well recognized in object-oriented software development. If well designed, packages will significantly reduce the complexity of a system and thus make it easier to understand, maintain, and extend. However, as software evolves, even for a well-modularized system, the quality of its software packages may gradually degrade over time with code changes. For example, classes may be placed in unsuitable

* Corresponding author at: State Key Laboratory for Novel Software Technology, Nanjing University, China.

http://dx.doi.org/10.1016/j.infsof.2014.09.006
0950-5849/© 2014 Elsevier B.V. All rights reserved.

packages, which makes software maintenance increasingly difficult and expensive [2]. Consequently, in order to reduce the cost of quality assurance, it is necessary to assess the quality of package organization. If we are able to measure the modularization quality of packages in a quantitative way, we can identify potentially problematic packages and take remedial measures to enhance their quality in time. This is especially important for legacy large-scale object-oriented software systems.

Despite the importance of package measurement, little effort has been devoted to develop metrics for measuring the modularization quality of packages and to empirically evaluate their usefulness for software development in practice. In [3], Martin proposed a metrics suite, including afferent couplings, efferent couplings, abstractness, and instability, to quantify the modularization quality of packages in terms of their extensibility, reusability, and maintainability. In [4], Elish et al. found that most metrics in Martin's suite were significantly related to the number of pre-release/post-release faults in packages. Recently, Sarkar et al. proposed a new metrics suite to characterize the modularization quality of a package [5]. Compared with previous work, Sarkar et al.'s suite characterizes the quality of modularization from more comprehensive perspectives such as inter-module call traffic, state access violations, fragile base-class design, programming to interface, and plugin pollution. These package-modularization metrics are quite different from traditional package-level metrics, which measure software quality mainly from the perspectives of size, extensibility, responsibility, independence, abstractness, and instability. In their study, Sarkar et al. argued that the proposed suite was able to reveal the real quality of modularization by manually inspecting six large object-oriented systems. Furthermore, they reported that the proposed suite was able to reveal the quality degradation in a system caused by novice programmers via a simulation experiment. However, little is currently known on the implications of these package-modularization metrics for software fault-proneness, especially compared with traditional package-level metrics.

In this paper, we aim to empirically examine the usefulness of Sarkar et al.'s package-modularization metrics in predicting the fault-proneness of packages in object-oriented software systems. In this context, if at least one post-release fault is detected in a package, the package is considered to be faulty and otherwise not-faulty. We first use principal component analysis to analyze whether Sarkar et al.'s package-modularization metrics capture additional information compared with traditional package-level metrics, including source code size and Martin's metrics suite. Second, we employ univariate prediction models to investigate how Sarkar et al.'s package-modularization metrics are related to package fault-proneness. Finally, we build multivariate prediction models to examine the ability of Sarkar et al.'s package-modularization metrics for predicting fault-proneness. Based on six object-oriented systems, our results show that: (1) package-modularization metrics provide new and complementary views of software complexity compared with traditional package-level metrics; (2) most package-modularization metrics have a significant association with package fault-proneness in an expected direction; and (3) package-modularization metrics can substantially improve the effectiveness of package fault-proneness prediction when used with traditional package-level metrics together. These results provide valuable data in an important area where there is limited experimental data available. These experimental results are critically important to help both researchers and practitioners understand whether package-modularization metrics are indeed of practical value for fault-proneness prediction. We believe that they can guide the development of better fault-proneness prediction models in practice.

The remainder of the paper is structured as follows. Section 2 describes package-modularization metrics and traditional package-level complexity metrics and formulates the research questions investigated in this study. Section 3 presents the modeling technique for fault-proneness prediction models and the data analysis methods for the three research questions under investigation. Section 4 introduces the data source and reports the distribution of package-modularization metrics. Section 5 provides in detail the experimental results. Section 6 discusses our findings. Section 7 analyzes the threats to the validity of our study. Section 8 gives the conclusions and outlines the directions for future work.

2. The investigated metrics and research questions

In this section, we first describe the definitions of Sarkar et al.'s software modularization metrics and traditional package-level complexity metrics. Then, we formulate the research questions relating software modularization metrics to traditional package-level complexity metrics and package fault-proneness.

2.1. Software modularization metrics

In a large object-oriented software system, it is common to organize classes into different packages in order to manage the complexity of the system. In this context, packages are actually used as the units for software modularization and hence their quality has a large influence on the quality of the system.

Recently, Sarkar et al. proposed a set of metrics to measure the quality of software modularization (as shown in Table 1). Overall, these metrics can be classified into to three categories. The first category is to measure the quality of modules (i.e. packages in our study) with respect to inter-module coupling created by method invocation. Ideally, each module should use its APIs (Application Programming Interfaces[1]) to provide well identified services to other modules and these modules should access each other's resources only through the published APIs. In particular, if a module has multiple S-APIs, each S-API should be cohesive from the perspective of a similarity of purpose and should be maximally segregated from the other S-APIs from the perspective of usage [5]. However, in practice, not all software systems strictly follow this modularization principle. Therefore, Sarkar et al. use the following metrics to measure the modularization quality of modules with respect to APIs:

- MII (method interaction index). It measures the extent to which all external calls made to a module are routed though the APIs of the module.
- NC (non-API method closeness index). It measures the extent to which non-API public methods in a module are not called by other modules.
- APIU (API usage index). It measures the extent to which the cohesiveness and segregation properties are followed by the S-APIs of a module.

The second category is to measure the quality of modules with respect to inter-module couplings created by inheritance and association. Inheritance and association are the most important two dependence relationships between classes in a system. The former denotes that a class extends another class, while the latter denotes that a class uses another class either as an attribute or as a parameter in a method definition. In a real system, it is not uncommon to

[1] According to [5], an API in a module is a set of public methods. There exist two different kinds of APIs in a module: S-API (service API) and E-API (extension API). An S-API aims to provide a specific service to other modules. An E-API indeed describes the services that need to be provided by an external plugin for the module. Therefore, an E-API generally consists of a set of abstract methods whose implementation codes are provided by the external plugin. A module may have multiple APIs.