Information and Software Technology 57 (2015) 186–203

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
Available online 11 October 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.

see that classes having such dependencies are distributed in different modules. In particular, if a class and its subclass exist in two different modules, a phenomenon known as fragile base-class problem (i.e. a seemingly safe modification to the base class may cause the subclass to break) may occur when these modules are maintained by independent programmers. In practice, these

Table 1
Definition of package modularization metrics.

Category	Metric	Definition	Description
Coupling by method invocation	MII (module interaction index)	$MII(p) = \frac{\lvert \bigcup_{\imath \in \mathrm{I(p)}} ExtCall\mathrm{Rel}(\imath) \rvert}{\lvert ExtCallRel(p) \rvert}$ $ExtCallRel(\imath) = \{\langle m_f, m_t \rangle \mid Call(m_f, m_t) \wedge m_t \in M(\imath) \wedge m_f \notin M(p)\}$ $ExtCallRel(p) = \{\langle m_f, m_t \rangle \mid Call(m_f, m_t) \wedge m_t \in M(p) \wedge m_f \notin M(p)\}$	The extent to which all external calls made to a module are routed though the APIs of the module
	NC (non-API method closedness index)	$NC(p) = \frac{\lvert M_{na}(p) \rvert}{\lvert M_{pub}(p) - \left\{ \bigcup_{i \in I(p)} M(i) \right\} \rvert}$	The extent to which non-API public methods in a module are not called by other modules
	APIU (API usage index)	$APIU(p) = \frac{APIUS(p) + APIUC(p)}{2}$ $APIUC(p) = \frac{\sum_{\imath \in I^S(p)} (\alpha(\imath) \times APIUC(\imath))}{\lvert I^S(p) \rvert}$ $APIUC(\imath) = \frac{\sum_{p' \in CallingMod(\imath)} \lvert M(\imath, p') \rvert}{\lvert CallingMod(\imath) \rvert \times \lvert M(\imath) \rvert} \quad \alpha(\imath) = \frac{\lvert M(\imath) \rvert}{\lvert \bigcup_{\imath_j \in I^S(p)} M(\imath_j) \rvert}$ $CallingMod(\imath) = \left\{ p \in P \mid \exists_{m_f \in M(p)} \exists_{m_t \in M(\imath)} Call(m_f, m_t) \wedge p \neq p' \right\}$ $M(\imath, p') = \left\{ m \in M(\imath) \mid p' \in CallingMod(\imath) \wedge \exists_{m_f \in M(p')} Call(m_f, m) \right\}$ $APIUS(p) = 1 - \left(\frac{\lvert \bigcup_{\imath_1, \imath_2 \in APIUSet(p)} CallingMod(\imath_1) \cap CallingMod(\imath_2) \rvert}{\lvert \bigcup_{\imath \in APIUSet(p)} CallingMod(\imath) \rvert} \right) = 0 \ if \ APIUSet(p) = \emptyset$ $APIUCSet(p) = \left\{ \imath \in I^S(p) \mid APIUC(\imath) \geqslant \lambda \right\}$	The extent to which the cohesiveness and segregation properties are followed by the S-APIs of a module
Coupling by inheritance or association	BCFI (base-class fragility index)	$BCFI(p) = 1 - \frac{1}{\lvert BCVSet(p) \rvert} \sum_{c \in C(p)} BCVMax(c)$ $BCVSet(p) = \{ c \in C(p) \mid \exists_{m \in M(c)} (UpCallBy(m) \neq \emptyset) \}$ $BCVMax(c) = \max_{m \in M(c)} BCVMax(m)$ $BCVMax(m) = \max_{m' \in UpCallby(m)} \{BCVio(m')\} = 0 \ when \ UpCallby(m) = \emptyset$ $UpCallby(m) = \{ m'_{conc} \mid Call(m, m'_{conc}) \wedge (m'_{conc} \in M(c) \vee m'_{conc} \in M_{anc}(c)) \}$ $BCVio(m') = \frac{\lvert \{ m'_o \in Ovride(m') \mid Module(m') \neq Module(m'_o) \} \rvert}{\lvert Ovride(m') \rvert} = 0 \ when \ Ovride(m') = 0$ $M_{anc}(c) = \{ m'_{conc} \in M(c') \mid c' \in Anc(c) \wedge (\forall_{m \in M(c) \neg Ovride(m'_{conc}, m)}) \}$	The extent to which the methods that have overridden the concrete methods called in a module exist in the same module
	IC (inheritance-based inter-module coupling index)	$IC(p) = \min(IC_1, IC_2, IC_3)$ $IC_1(p) = 1 - \frac{\lvert \{p' \in P\} \mid \exists_{d \in C(p')} \exists_{c \in C(p)} (Child(c, d) \wedge p \neq p') \rvert}{\lvert P \rvert - 1} = 1 \ when \ \lvert P \rvert = 1$ $IC_2(p) = 1 - \frac{\lvert \{ d \in C \mid \exists_{c \in C(p)} (Child(c, d) \wedge Module(d) \neq p) \} \rvert}{\lvert C - C(p) \rvert} = 1 \ when \ \lvert P \rvert = 1$ $IC_3(p) = 1 - \frac{\lvert \{ c \in C(p) \mid \exists_{d \in Par(c)} (Module(d) \neq p) \} \rvert}{\lvert C(p) \rvert}$	The extent to which there are no inheritance-based inter-module dependencies between a module and other modules
	AC (association-induced coupling index)	$AC(p) = \min\{AC_1, AC_2, AC_3\}$ $AC_1(p) = 1 - \frac{\lvert \{ p' \in P \mid \exists_{c \in C(p')} \exists_{d \in C(p)} (Uses(c, d) \wedge p \neq p') \} \rvert}{\lvert P \rvert - 1} = 1 \ when \ \lvert P \rvert = 1$ $AC_2(p) = 1 - \frac{\lvert \{ c \in C \mid \exists_{d \in C(p)} (Uses(c, d) \wedge Module(c) \neq p) \} \rvert}{\lvert C - C(p) \rvert} = 1 \ when \ \lvert P \rvert = 1$ $AC_3(p) = 1 - \frac{\lvert \{ c \in C(p) \mid \exists_{d \in Uses(c)} (Module(d) \neq p) \} \rvert}{\lvert C(p) \rvert}$	The extent to which there are no association-based inter-module dependencies between a module and other modules

Table 1 (*continued*)

Category	Metric	Definition	Description
Other modularization practices	NPII (not-programming-to-interfaces index)	$NPII(p) = \frac{\lvert AbsCallby(p)\rvert}{\lvert AbsCallby(p)\rvert + \lvert InhConcCallby(p)\rvert} + \frac{\lvert InhConcCallby(p) - BadConcCallby(p)\rvert}{\lvert AbsCallby(p)\rvert + \lvert InhConcCallby(p)\rvert}$ $AbsCallby(p) = \bigcup_{m \in M(p)} AbsCallby(m)$ $InhConcCallby(p) = \bigcup_{m \in M(p)} InhConcCallby(m)$ $BadConcCallby(p) = \bigcup_{m \in M(p)} BadConcCallby(m)$ $AbsCallby(m) = \{m_{abs} \mid m_{abs} \in Callby(m)\}$ $InhConcCallby(m) = \{m_{conc} \mid m_{conc} \in Callby(m) \wedge \exists_{c' \in C}(m_{conc} \in M(c,)) \wedge Anc(c') \neq \emptyset\}$ $BadConcCallby(m) = \{m' \mid m' \in InhConcCallby(m) \wedge (DoesOvride(m') \vee DoesImpl(m'))\}$	The extent to which the method calls in a module are calls to abstract methods or are not bad calls
	SAVI (state access violation index)	$SAVI(p) = \frac{1}{\lvert C(p)\rvert} \sum_{c \in C(p)} SAVI(c)$ $SAVI(c) = 1 - \left(\begin{array}{c} \varpi_1 \frac{\lvert InterStAcc1(c)\rvert}{\lvert A(c)\rvert} + \varpi_2 \frac{\lvert InterStAcc2(c)\rvert}{\lvert P\rvert} \\ + \varpi_3 \frac{\lvert IntraStAcc1(c)\rvert}{\lvert A(c)\rvert} + \varpi_4 \frac{\lvert IntraStAcc2(c)\rvert}{\lvert C(p)\rvert} \end{array} \right)$ $InterStAcc1(c) = \{a \in A(c) \mid \exists_{d \in Uses(a)} Module(c) \neq Module(d)\}$ $InterStAcc2(c) = \{p \in P \mid \exists_{a \in A(c)} \exists_{d \in C(p')} (Module(c) \neq p' \wedge uses(a,d))\}$ $IntraStAcc1(c) = \{a \in A(c) \mid \exists_{d \in used(a)} Module(c) = Module(d)\}$ $IntraStAcc1(c) = \{d \in C(p) \mid \exists_{a \in A(c)} (Module(c) = Module(d) \wedge uses(a,d))\}$	The extent to which the attributes defined in the classes in a module are not directly accessed by other classes
	CU$_m$ (size uniformity index)	$CU_m(p) = \frac{\mu_m(p)}{\mu_m(p) + \sigma_m(p)}$	The extent to which the classes in a module are different in terms of the number of methods
	CU$_l$ (size uniformity index)	$CU_l(p) = \frac{\mu_l(p)}{\mu_l(p) + \sigma_l(p)}$	The extent to which the classes in a module are different in terms of lines of code
	PPI (plugin pollution index)	$PPI(p) = \frac{\left\lvert \bigcup_{m \in ImplExtn(p)} \{ModuleClosure(m,p)\} \right\rvert}{\lvert M(p)\rvert}$ $ImplExtn(p) = \left\{ m \in M(p) \mid \forall_{p' \in P} \exists_{\imath \in I^E(p')} Abstract(m) \in \imath \right\}$ $ModuleClosure(m,p) = \left\{ Callby^{+}(m) \bigcup \{m\} \right\} \bigcap M(p)$	The extent to which the methods that are needed to define extension APIs exists in a plugin module

cross-module-boundaries dependencies created by inheritance and association make the modules more difficult to maintain and extend. In their study, Sarkar et al. use the following metrics to measure the modularization quality of modules with respect to such inter-module dependencies[2]:

- BCFI (base-class fragility index). It measures the extent to which the methods that have overridden the concrete methods called in a module exist in the same module.
- IC (inheritance-based inter-module coupling index). It measures the extent to which there are no inheritance-based inter-module dependencies between a module and other modules.
- AC (association-induced coupling index). It measures the extent to which there are no association-based inter-module dependencies between a module and other modules.

The third category is to measure the quality of modules with respect to other modularization practices. In the literature, many other modularization principles are recommended to enhance the quality of the modules in a system [5]. These principles include adherence to programming-to-interface, free of state-access-violation, maximum of module size uniformity, and avoidance of plugin code bloat. In their study, Sarkar et al. use the following metrics to measure the modularization quality of modules with respect to these principles[3]:

- NPII (not-programming-to-interfaces index). It measures the extent to which the method calls in a module are calls to abstract methods or are not bad calls (a call is bad if the called method is an overriding method or an implementation of an abstract method).
- SAVI (state access violation index). It measures the extent to which the attributes defined in the classes in a module are not directly accessed by other classes.
- CU$_m$ (size uniformity index). It measures the extent to which the sizes of the classes in a module are different in terms of the number of methods.
- CU$_l$ (size uniformity index). It measures the extent to which the sizes of the classes in a module are different in terms of the number of lines of code.

[2] Note that BCFI, IC and AC are inverse measures, i.e., a large BCFI value indicates a small possibility of the existence of the fragile base-class problem and a large IC/AC value indicates a small inter-module coupling.

[3] Note that NPII, SAVI, and PPI are inverse measures, i.e., a large NPII value indicates a small possibility of the existence of the not-programming-to-interfaces problem, a large SAVI value indicates a small possibility of the existence of the state-access-violation problem, and a large PPI value indicates a small possibility of the existence of the plugin code bloat problem.

Table 2
The notations used in Table 1.

Notation	Description	Notation	Description
P	The set of modules in a system	$Anc(c)$	The set of ancestor classes of class c
p	A module in P	$Par(c)$	The set of parent classes of class c
$I(p)$	The set of APIs in module p	$Par(c, d)$	Whether class d is a parent of class c
$I^S(p)$	The set of S-APIs in module p	$Child(c, d)$	Whether class d is a child of class c
$I^E(p)$	The set of E-APIs in module p	$Uses(c)$	The set of classes that class c uses
$C(p)$	The set of classes in module p	$Uses(c, d)$	Whether class c uses class d
$M(p)$	The set of methods in module p	$Uses(a, d)$	Whether class d uses attribute a
$M_{pub}(p)$	The set of public methods in module p	m_{abs}	An abstract method
$M_{na}(p)$	The actual non-API public methods in module p that are not being called by other modules	m_{conc}	A method that is not abstract
$\mu_m(p)$	The average class size in module p in terms of the number of methods	$Abstract(m)$	The abstract method implemented by method m
$\mu_l(p)$	The average class size in module p in terms of the number of lines of code	$DoesImpl(m')$	Whether method m' is an implementation of some abstract method
$\sigma_m(p)$	The standard deviation of the class size in module p in terms of the number of methods	$Ovride(m)$	The set of methods that override method m
$\sigma_l(p)$	The standard deviation of the class size in module p in terms of lines of code	$DoesOvride(m')$	Whether method m' overrides a method
$M(\imath)$	The set of methods in API $\imath$	$Call(m_f, m_t)$	Whether method m_f call method m_t
C	The set of classes in a system	$Callby(m)$	The set of method called by method m
$Module(c)$	The module to which class c belongs	$Callby^+(m)$	The transitive closure of the method calls
$M(c)$	The set of methods in class c	λ	A user-defined threshold for determining whether an S-API is cohesive
$A(c)$	The set of attributes in class c	$\varpi_1, \varpi_2, \varpi_3, \varpi_4$	The user-defined weights (their sum = 1)

- PPI (Plugin pollution index). It measures the extent to which the methods that are needed to define extension APIs exists in a plugin module.

Table 1 summarizes the formal definitions and descriptions of the above-mentioned software modularization metrics that will be investigated in this study and Table 2 lists the notations used in Table 1. For each of these software modularization metrics, its value always ranges between 0 and 1. In particular, the larger the value of a metric is, the better the modularization quality of the module under investigation is. From Table 1, we can see that MII, NC, APIU, and PPI for a module are based on the concepts of S-APIs and E-APIs. However, in a large object-oriented software system, the S-APIs and E-APIs for the modules are generally not explicitly declared. In [5], to deal with this problem, Sarkar et al. propose the following heuristics to identify the S-APIs and E-APIs for a given module p, respectively:

- The identification of the S-APIs. At the first step, identify the set of public abstract methods in the classes in module p. At the second step, remove those public abstract methods whose implementation codes are not in the classes in module p. At the third step, remove those public abstract methods that are never called by the classes in the other modules. At the fourth step, for the remaining abstract methods, add those concrete methods that are frequently called by the other modules.
- The identification of the E-APIs. At the first step, identify the set of public abstract methods in the classes in module p. At the second step, remove those public abstract methods whose implementation codes are in the classes in module p. At the third step, retain those public abstract methods that are only called by the classes in module p. In other words, a public abstract method is identified as a method in the E-API of module p only if its declaration is in p, its implementation code is outside p, and it receives only internal calls from p.

2.2. Traditional package-level complexity metrics

In this paper, we aim to investigate the ability of Sarkar et al.'s package-modularization metrics for fault-proneness prediction compared with traditional package-level metrics. Table 3 lists the traditional package-level metrics investigated in this study, in which the former seven metrics were developed by Martin [3]. *N*, *Ca*, *Ce*, and *A* are respectively the indicators of a package's extensibility, responsibility, independence, and abstractness. The higher the value of *N* (*Ca*/*A*) is, the higher the package's extensibility (responsibility/abstractness) is. The lower the value of *Ce* is, the higher the package's independence is. *I* is a composite metric of *Ca* and *Ce*, which indicates a package's resilience to change. The value of *I* ranges between 0 and 1, with $I = 0$ indicating a completely stable package and $I = 1$ indicating a completely instable package. *D* is an indicator of a package's balance between abstractness and stability. An ideal package is either completely abstract and stable ($I = 0$, $A = 1$) or completely concrete and instable ($I = 1$, $A = 0$). The smaller the distance *D* is, the more balance the package is. *isCycle* is an indicator whether a package directly participates in a package dependence cycle or depends on packages directly participating in a cycle. In addition to the Martin's metrics suite, we also include the most commonly used *SLOC* in the traditional package-level metrics. The size metric *SLOC* simply counts the non-blank, non-commentary source lines of code in a package. These traditional package-level metrics can be automatically and cheaply collected from source codes, even for large-scale object-oriented software. Currently, they have been incorporated into many development tools such as JDepend,[4] NDepend,[5] RefactorIT,[6] and the DMS software reengineering Toolkit.[7] In particular, existing experimental results show that they are useful predictors for identifying potentially faulty packages and hence are very helpful for software quality enhancement [4].

2.3. Research questions

We use these traditional package-level metrics as the baseline metrics to investigate the practical value of Sarkar et al.'s package-modularization metrics in the context of fault-proneness

[4] JDepend. http://www.clarkware.com/software/JDepend.html.
[5] NDepend. http://www.ndepend.com.
[6] RefactorIT. http://refactorit.sourceforge.net.
[7] DMS software reengineering Toolkit. http://www.semdesigns.com/Products/DMS/DMSToolkit.html.

Table 3
Definition of traditional package-level metrics.

Metric	Description
N (number of classes and interfaces)	The number of concrete and abstract classes (and interfaces) in a package
Ca (afferent couplings)	The number of other packages that depend upon classes within a package
Ce (efferent couplings)	The number of other packages that the classes in a package depend upon
A (abstractness)	The ratio of the number of abstract classes (and interfaces) in a package to the total number of classes in the package
I (instability)	The ratio of efferent coupling (*Ce*) to total coupling (*Ce* + *Ca*), i.e. $I = Ce/(Ce + Ca)$
D (distance from the main sequence)	The normalized perpendicular distance of a package from the idealized line $A + I = 1$ (called main sequence), i.e. $D = \|A + I - 1\|$
isCycle (package dependency cycles)	Whether a package participates in package dependency cycles
SLOC	The source lines of code in a package

prediction. More specifically, we will investigate the following three research questions:

- *RQ1 (Relationship with traditional metrics):* Do these new package-modularization metrics capture different aspects of complexity that are not captured by traditional package-level complexity metrics?
- *RQ2 (Association with package fault-proneness)*: Are these new package-modularization metrics statistically negatively related to package fault-proneness?
- *RQ3 (Ability to improve fault-proneness prediction):* Can these new package-modularization metrics significantly improve the performance of package fault-proneness prediction when used with traditional package-level complexity metrics together?

These research questions are of critical importance for software practitioners when developing high-quality object-oriented systems. The objective of RQ1 is to determine whether Sarkar et al.'s package-modularization metrics are redundant with respect to traditional complexity metrics. If the experimental results show that they are redundant in nature, there is no need to collect these package-modularization metrics at all in practice. The objective of RQ2 is to determine whether Sarkar et al.'s package-modularization metrics are statistically negatively related to the occurrence of faults in packages. As described in Section 2.1, for each package-modularization metric, a larger value indicates a better modularization quality. This leads us to naturally conjecture that there is a negative association between modularization metrics and package fault-proneness. If the result from RQ1 shows that these modularization metrics are not redundant, we need to further examine our conjecture in order to determine whether they are useful predictors for package fault-proneness. The objective of RQ3 is to determine whether we should employ these modularization metrics to build fault-proneness prediction models in practice. This research question is indeed raised by RQ1 and RQ2. If the results from RQ1 and RQ2 show that Sarkar et al.'s modularization metrics are not redundant and useful fault-proneness predictors, we still need to investigate whether their combination with traditional metrics can provide a significantly better fault-proneness prediction performance. If the answer is no, these package-modularization metrics are still of no practical value for fault-proneness prediction. Currently, no empirical results are available on these research questions. Our study attempts to fill this gap by an in-depth investigation into the actual usefulness of these new package-modularization metrics in the context of fault-proneness prediction.

3. Experimental methodology

In this section, we describe the technique for modeling fault-proneness prediction models and the data analysis methods for the three research questions under investigation.

3.1. Modeling technique

We build logistic regression models to predict the probability of a package being faulty. In the literature, logistic regression is the most commonly technique to build fault-proneness prediction models as it is easy to use and understand. Recently, Hall et al. performed a systematic literature review on fault prediction performance in software engineering [6]. In the review, one of their three research questions was to investigate which modeling technique performed best when used in fault prediction. Based on 208 fault prediction studies published from 2000 to 2010, they presented a synthesis of current knowledge on the impact of a wide variety of modeling techniques on model performance. Consequently, they concluded that logistic regression performed well compared with other complex modeling techniques when predicting fault-proneness. A similar conclusion was obtained in [7], in which no significant performance difference was detected between logistic regression and the other 16 classifiers. In particular, Lessmann et al. concluded that a simple modeling technique such as logistic regression was sufficient to model the relationship between static code attributes and software defect [7].

In the logistic regression, the dependent variable Y can take on only one of two different values: 0 or 1. In our study, if there is at least one post-release fault detected in a package, then the dependent variable Y is set to 1, and 0 otherwise. Given the independent variables $X_1, X_2, \ldots$, and X_n (i.e. the modularization metrics and traditional complexity in this study), let $\Pr(Y = 1 \mid X_1, X_2, \ldots, X_n)$ represent the corresponding probability that the dependent variable $Y = 1$ (i.e. the extent of a package being faulty). A multivariate logistic regression model assumes that $\Pr(Y = 1 \mid X_1, X_2, \ldots, X_n)$ is related to $X_1, X_2, \ldots, X_n$ by the following equation:

$$\Pr(Y = 1 | X_1, X_2, \ldots, X_n) = \frac{e^{\alpha+\beta_1 X_1+\beta_2 X_2+\cdots+\beta_n X_n}}{1 + e^{\alpha+\beta_1 X_1+\beta_2 X_2+\cdots+\beta_n X_n}}$$

where α and β_is are the regression coefficients and can be estimated through the maximization of a log-likelihood. The univariate logistic regression model is a special case of the multivariate logistic regression model, where there is only one independent variable.

3.2. Principal component analysis

In order to investigate RQ1, we use Principal Component Analysis (PCA) to determine whether Sarkar et al.'s package-modularization metrics capture additional complexity dimensions than traditional complexity metrics. PCA is a statistical technique that uses orthogonal transformation to convert a set of possibly correlated independent variables into a set of linearly uncorrelated variables (i.e. dimensions) called principal components (PCs). This enables us to identify the underlying, orthogonal dimensions that explain the relations between the independent variables in a data set. In our study, following in [8], we only retain the PCs that have eigenvalues greater than one for interpretation. In particular, we use PCA with the varimax rotation to make the mapping of the independent variables to PCs clearer where the variables have either a very low or very high loading. This helps to identify which independent variables indeed measure the same property, though they may purport to capture different properties. In our context, package-modularization metrics are considered not redundant if the result of PCA shows that they define new PCs of their own compared with traditional complexity metrics.

3.3. Univariate logistic regression analysis

In order to investigate RQ2, we use univariate logistic regression to examine whether each package-modularization metric is significantly negatively related to package fault-proneness. We use the Cook's distance to estimate the influence of an observation on the regression coefficients. If an observation has a Cook's distance equal to or larger than 1, it is regarded as an influential observation and hence should be excluded from the univariate logistic regression analysis [9]. For each of Sarkar et al.'s package-modularization metrics, a larger value indicates a better modularization. Therefore, it is expected that each package-modularization metric is negatively related to package fault-proneness. In our study, we will investigate whether each package-modularization metric is statistically significant at the significance level of 0.05, 0.10, or 0.20. Furthermore, we use the commonly used ΔOR, the odds ratio associated with one standard deviation increase, to examine the magnitude of association between each metric and fault-proneness [10]. This enables us to compare the relative magnitude of the associations of individual metrics with fault-proneness. For a given independent variable, ΔOR is the exponent of the product of the logistic regression coefficient and its standard deviation. Indeed, ΔOR corresponds to the ratio of the odds of being faulty after and before one standard deviation increase in the independent variable. Here, the odds of being faulty is the probability of being faulty divided by the probability of being not-faulty. If ΔOR is greater than 1, then it indicates a positive association between the independent variable and fault-proneness. If ΔOR is less than 1, it indicates a negative association. If ΔOR is equal to 1, it indicates no association.

Previous studies show that module size may have a large confounding effect on the true associations between product metrics and fault-proneness [11,12]. Therefore, in this paper, for a given package metric X, we first use the linear regression based method proposed in [11,12] to remove the potentially confounding effect of package size. More specifically, at the first step, we build a univariate linear regression model with Z against X. At the second step, we subtract the predicted values by the regression model from X to obtain a new variable X'. After that, we perform a univariate logistic regression with X' against fault-proneness. This will give us an understanding on the true association between X and fault-proneness.

3.4. Multivariate logistic regression analysis

In order to investigate RQ3, we build two types of multivariate logistic regression models: (1) the "T" model (using only traditional complexity metrics) and (2) the "M+T" model (using both package-modularization metrics and traditional complexity metrics). Before building these models, for each metric, we first remove the potentially confounding effect of package size using the linear-regression based method proposed in [12]. After that, we use the p-value as the criterion to perform a stepwise variable selection. In particular, we use the variance inflation factor (VIF) to detect the severity of multicollinearity among the independent variables. As a common rule of thumb, VIF should be less than 10 in order to avoid troubles with the stability of the coefficients [13]. In our study, if there are independent variables having a VIF > 10, we remove the independent variable having the maximum VIF to refit the model. This procedure is repeated until all the independent variables in the model have a VIF < 10. Furthermore, we use the Cook's distance to detect the influence observations. In our study, if an observation has a Cook's distance > 1, it is considered as an influential observation and hence is excluded from the model building [14]. Then, we use 30 times 3-fold cross-validation to compare the predictive effectiveness of "M+T" against "T". Consequently, each model has $30 \times 3 = 90$ predictive effectiveness values. Based on these values, we use the Wilcoxon's signed-rank test to examine whether "M+T" significantly outperforms "T" at the significance level of 0.05. In our study, we will report the improvement in percentage using "M+T" model over "T" model in terms of the prediction performance. This will help us to determine whether the improvement caused by package-modularization metrics are important from the viewpoint of practical application, especially when the extra cost involved in data collection is considered.

We investigate RQ3 in the following two typical application scenarios: ranking and classification. In the ranking scenario, packages are ranked in descending order according to their predicted relative risk. With such a ranking list in hand, software developers can simply select as many "high-risk" packages for inspection or testing as available resources will allow. In the classification scenario, packages are first predicted as "high-risk" or "low-risk" according to their predicted relative risk. After that, those packages predicted as "high-risk" will be targeted for inspection or testing. In both scenarios, we take into account the effort to inspect or test those "high-risk" packages when evaluating the predictive effectiveness of a model m. Following previous work [15], we use $SLOC$ in a package p as the proxy of the effort required to inspect or test the package p. In particular, we define the relative risk of the package p as $R(p) = \Pr/SLOC(p)$, where Pr is the probability of p being faulty predicted by the model m. This is the most commonly used relative risk formula in the context of effort-aware fault-proneness prediction [16–18]. In the following, we will describe the effort-aware predictive performance indicators used in this study for ranking and classification. Note that there are many other performance measures to evaluate the effectiveness of predicting models, such as recall and precision. However, these regular measures do not take into account the cost associated with additional quality assurance activities for each module, which is not reasonable in practice [15,16,19]. For this reason, cost and/or effort aware measures recently become a significant strand of interest in prediction measurement, and have been increasingly reported as effective [6,12]. Therefore, we investigate RQ3 using the effort-aware predictive performance indicators.

3.4.1. Effort-aware ranking performance indicator

In the ranking scenario, CE (cost-effectiveness) is the most commonly used performance indicator for evaluating the effort-aware ranking effectiveness of a fault-proneness prediction model [15]. For a given model m, the corresponding CE is computed based on a so-called "SLOC-based" Alberg diagram (as shown in Fig. 1). In this diagram, the x-axis represents the cumulative percentage of SLOC of the packages selected from the package ranking and the y-axis represents the cumulative percentage of faults found in those selected packages. Consequently, the prediction model m corresponds to a curve in the diagram. In addition to the curve corresponding to m, the Alberg diagram also includes two additional curves, which respectively correspond to the "optimal" and "random" models. The "optimal" model denotes the best model in theoretically, in which packages are sorted in decreasing order according to their actual fault densities. The "random" model denotes that packages are randomly selected to inspect or test.

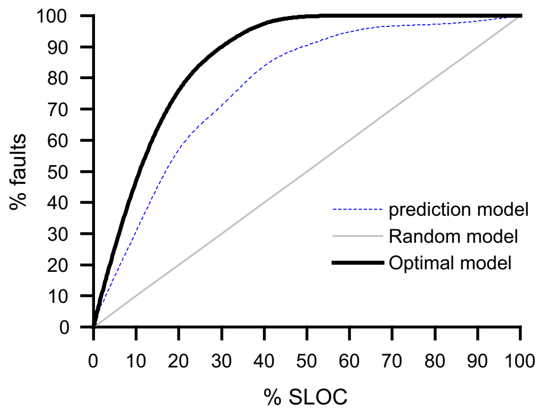

Fig. 1. SLOC-based Alberg diagram.

Based on the "SLOC-based" Alberg diagram, the effort-aware ranking effectiveness indicator CE of the prediction model m is defined as [15]:

$$\mathrm{CE}_\pi(m) = \frac{Area_\pi(m) - Area_\pi(random\ model)}{Area_\pi(optimal\ model) - Area_\pi(random\ model)}$$

In the above definition, $Area_\pi(m)$ denotes the area under the curve corresponding to model m for a given top $\pi \times 100\%$ percentage of SLOC. The cut-off value π varies between 0 and 1, depending on the amount of available resource for inspecting modules. As can be seen, $\mathrm{CE}_\pi(m)$ ranges between -1 and 1. The larger the value of CE is, the better the ranking effectiveness is. In particular, if $\mathrm{CE}_\pi(m)$ is less than 0, it means that the model m is worse than the "random" model. In practice, since software development resource is often limited, it may not be possible to inspect or test all the modules in a system. Consequently, developers have to choose a small number of modules in the system to inspect or test. In this context, it is widely believed that the rank performance of the top fraction of such a ranking list is the most important for practitioners (given limited inspection or testing resource) [15,29,38]. In other words, developers are more interested in the ranking performance of the prediction model m at the top fractions (for example $\pi = 0.1$ or 0.2).

3.4.2. Effort-aware classification performance indicator

In the classification scenario, ER (effort reduction, originally called "LIR") is the most commonly used performance indicator for evaluating the effort-aware classification effectiveness of a fault-proneness prediction model [20]. Indeed, ER denotes the ratio of the reduced source lines of code to inspect or test by using a classification model compared with random selection to achieve the same recall of faults. Assume that the system under analysis consists of n packages. For $1 \leqslant i \leqslant n$, let s_i and f_i respectively be the SLOC and the number of faults in package i. Furthermore, let p_i be 1 if package i is predicted as "high-risk" by the model m and 0 otherwise, $1 \leqslant i \leqslant n$. In the classification scenario, only those packages predicted as "high-risk" will be inspected or tested. In this context, the effort-aware classification effectiveness indicator ER of the prediction model m is defined as:

$$\mathrm{ER}(m) = \frac{Effort(random) - Effort(m)}{Effort(random)}$$

In the above definition, $Effort(m)$ denotes the ratio of the total SLOC in those predicted "high-risk" packages to the total SLOC in the system, i.e. $Effort(m) = \sum_{i=1}^{n} s_i \times p_i / \sum_{i}^{n} s_i$. Effort(random) is the ratio of SLOC to inspect or test to the total SLOC in the system that a random selection model needs to achieve the same recall of faults as the model m, i.e. $Effort(random) = \sum_{i=1}^{n} f_i \times p_i / \sum_{i=1}^{n} f_i$.

The prerequisite of computing ER is the determination of the classification threshold for the prediction model m. In the literature, there are three popular methods to determine the classification threshold on a training set. The first method, called BPP (balanced-pf-pd) method, leverages the ROC curve corresponding to model m to determine the classification threshold. In the ROC curve, pd (probability of detection) is plotted against pf (probability of false alarm) [21]. Indeed, the point (0,1) represents a perfect classification ($\mathrm{pf} = 0$ and $\mathrm{pd} = 1$). The closer a point in the ROC curve is to the point (0,1), the better the model predicts. In this context, the "balance" metric $balance = 1 - \sqrt{(0 - \mathrm{pf})^2 + (1 - \mathrm{pd})^2}/\sqrt{2}$ can be used to evaluate the degree of balance between pf and pd. The BPP method chooses the threshold that has a maximum "balance" value. The second method, called BCE (balanced-classification-error) method, chooses the threshold that roughly equalizes false positive rate and false negative rate. As stated in [22], this indeed has the effect, in imbalanced data sets, of giving more weight to errors from false positives. This is especially important for fault data, as they are typically imbalanced. The third method, called MFM (maximum-F-measure) method, chooses the threshold that has a maximum F-measure [29]. The F-measure is the harmonic mean of precision and recall, i.e. $\text{F-measure} = 2 \times Recall \times Precision/(Recall + Precision)$. Here, *Recall* is the ratio of the number of packages correctly classified as faulty to the total number of faulty packages. *Precision* is the ratio of the number of packages correctly classified as faulty to the total number of packages classified as faulty. In the following, the effort reduction metrics under the BPP, BCE, and MFM thresholds are respectively called "ER-BPP", "ER-BCE", and "ER-MFM".

In addition to "ER-BPP", "ER-BCE", and "ER-MFM", we also include an additional metric called "ER-AVG" for model evaluation. In practice, it is possible that a model is good under the BPP, BCE, and MFM thresholds but is poor under the other thresholds. Consequently, it may be misleading to use the "ER-BPP", "ER-BCE", and "ER-MFM" metrics for model evaluation. In this paper, we use "ER-AVG", the average effort reduction of a model over all possible thresholds on the test data set, to alleviate this problem. The "ER-AVG" metric is indeed independent of specific thresholds and hence can provide a complete picture of the classification performance.

4. Data sets

This study makes use of the data collected from six well-known open-source Java systems, including Cxf 2.1, Flume 1.4.0, Hbase 0.96.0, Hive 0.12.0, JDT Core 3.4, and Lucene 2.4.0. Cxf is an open-source, fully featured Web services framework. Flume is a distributed, reliable, and available service for efficiently collecting, aggregating, and moving large amounts of log data. HBase is a Hadoop database (a distributed, scalable, big data store) and aims to provide a storage system similar to Bigtable in the Hadoop distributed computing environment. Hive is a distributed agent platform, a decentralized system for building applications by networking local system resources. JDT Core is an Eclipse plug-in that implements the Java infrastructure of the Java IDE (Integrated development environment) including the core Java elements and application programming interface. Lucene is a high-performance, full-featured text search engine library, which provides a Java-based indexing and search implementation, spellchecking, hit highlighting, and advanced analysis/tokenization capabilities. We select these systems as the subjects of our study for two reasons. First, the numbers of packages in these systems are large enough to draw statistically meaningful conclusions on the practical value

Table 4
Descriptive information of the six systems under study.

System	Release date	Total size (KSLOC)	Number of packages	Number of classes	Number of interfaces	Number of Methods	Number of S-API methods	Number of E-API methods	Number of non-API public methods	Number of inter-package call	
										Call API	Call non-API
Cxf 2.1	28-04-2008	153	298	1572	208	13,058	285	154	9102	1667	9067
Flume 1.4.0	01-07-2013	38	41	282	59	1827	29	50	1200	66	939
Hbase 0.96.0	11-10-2013	390	88	1015	143	12,876	340	372	9349	909	8085
Hive 0.12.0	10-10-2013	394	160	1856	177	21,294	306	198	17,319	1355	12,155
JDT Core 3.4	17-06-2008	280	52	1045	188	18,046	369	143	11,485	1462	16,325
Lucene 2.4.0	10-10-2008	126	72	934	52	7947	166	118	5278	1155	5654

Table 5
The six data sets used in this study.

System	Number of packages	Number of faulty packages	Percent of faulty packages	Total size (KSLOC)
Cxf 2.1	294	64	21.8	152
Flume 1.4.0	41	14	34.1	38
Hbase 0.96.0	88	23	26.1	390
Hive 0.12.0	160	44	27.5	394
JDT Core 3.4	45	32	71.1	277
Lucene 2.4.0	71	15	21.1	125

of package-modularization metrics for predicting fault-proneness. Second, the fault information about the packages in these systems is publicly available. Therefore, it is relatively easy for other researchers to externally validate our empirical results.

Table 4 summarizes the descriptive information for these systems. The first to seventh columns respectively report the system name, the release date, the total size, the number of packages, the number of classes, the number of interfaces, and the number of methods. The eighth to ninth columns respectively report the number of S-API methods and the number of E-API methods for these systems. For each system, the number of identified E-APIs is larger than 0. This indicates that all the systems do come with plug-in packages. The tenth to eleventh columns report the number of inter-package method-calls, including the number of inter-package calls to API methods and the number of inter-package calls to non-API methods. For each system, the number of API method calls is much smaller than the number of non-API method calls, thus indicating that most inter-package call traffics are not directed through the API methods.

We collected six data sets from these six systems. Each data point in each data set corresponds to one Java package and consists of: (1) 11 package-modularization metrics; (2) 8 traditional package-level complexity metrics; and (3) a binary variable indicating whether the package had a post-release fault. To collect the 19 above-mentioned metrics, for each system, we first used a well-known commercial program understanding tool called "Understand"[8] to generate a corresponding database. This database provided information about entities (such as packages, classes, and methods) and references (such as method call, inheritance relations, and attribute references) in the system. After that, we used the Understand's Perl interface to develop a Perl script to collect the 19 metrics for each package in each system by assessing the corresponding database. In this paper, we used a binary variable indicating whether the package was likely to be fault-prone or not fault-prone. The former four systems (i.e. Cxf 2.1, Flume 1.4.0, Hbase 0.96.0, and Hive 0.12.0) used Github as the version control system and used JIRA as the issue tracking system. For these four systems, we developed a toolkit to extract the package-level fault data from the website https://issues.apache.org/jira/. In particular, we used JIRA's issue filter to obtain the post-release fault data that existed in the current version and were fixed in the later versions. For the latter two systems (i.e. JDT Core 3.4 and Lucene 2.4.0), we used the fault data provided in the website http://bug.inf.usi.ch/. We found that not all the packages in these systems had the corresponding fault information. In this study, we excluded those packages in which the fault information was unavailable. Table 5 summarizes, for each data set, the number of packages, the number of faulty packages, the percentage of faulty packages, and the total size in KSLOC.

Fig. 2 employs the box-plot to describe the distribution of the investigated 11 package-modularization metrics in the collected six data sets. For each metric, there are six box-plots, indicating the distribution of the metric in each data set. Each box-plot shows the 25th and 75th percentiles (the lower and upper sides of the box) as well as the median (the horizontal line in the middle of the box). From Fig. 2, we can see that all the data sets have a very low median MII value. This means that most inter-package call traffics are not routed through the identified S-APIs, which is consistent with the inter-package call traffics presented in Table 4. Furthermore, we observe that all the data sets have a median NC value larger than 0.7. This means that only a small percentage of non-API public methods actually participate in inter-package method calls. Compared with the other five data sets, JDK core 3.4 has a lower median BCFI and hence exhibits a larger extent of the fragile base-class problem. When looking at the distributions of IC and AC, we can see that association relationship makes a larger contribution to inter-package couplings than inheritance relationship. The distribution of NPII shows that the principle of programming-to-interfaces has been sufficiently adhered to in these systems. Of all the systems, JDK core 3.4 has the lowest median SAVI. This reveals that there is a more serious violation of the principle of not directly accessing the state of a class in JDK core 3.4. From the distributions of CU_m and CU_l, we can observe that JDK core 3.4 has the lowest median values. This indicates that packages in JDK core 3.4 have the largest class size variations, regardless of whether class size is measured by the number of methods or SLOC. The distribution of PPI shows that the extension modules in all the systems may have a low code quality. The median APIU for all the data sets are around 0.5, indicating a good API

[8] Available from http://www.scitools.com.

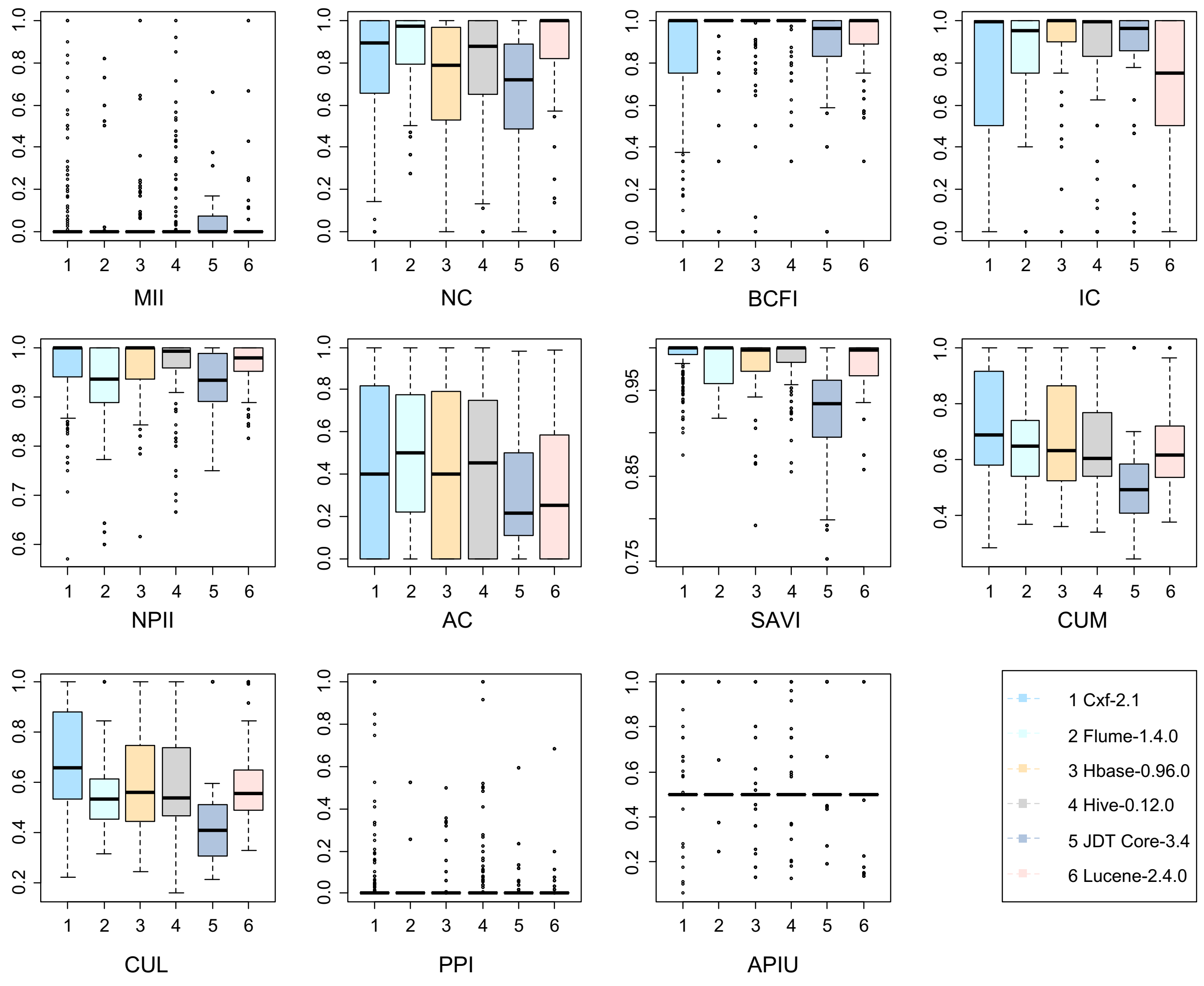

Fig. 2. Distributions of package-modularization metrics for the six data sets.

cohesion and segregation. Overall, for all the six systems, the distributions of most package-modularization metrics vary with different systems.

5. Experimental results

In this section, we present in detail the experimental result for software modularization metrics. In Section 5.1, we report the results from examining their redundancy with traditional complexity metrics (RQ1). In Section 5.2, we describe the results from examining their correlations with fault-proneness (RQ2). In Section 5.3, we give the results from examining their ability for improving the performance of fault-proneness prediction models (RQ3).

5.1. Relationship with traditional complexity metrics

To answer RQ1, we employ Principal Component Analysis (PCA) to investigate the relationship between package-modularization metrics and traditional complexity metrics. Fig. 3 shows the rotated components obtained from PCA for the six data sets (see more details in Tables 11–16 in Appendix A). For each data set, the corresponding sub-figure consists of three parts: (1) the top part reports the name of each PC; (2) the bottom part reports the metrics in each PC; and (3) the middle part reports the cumulative percentage of variances explained by these PCs. In particular, for the bottom part, the dashed line distinguishes between traditional metrics and package-modularization metrics in each PC.

From Fig. 3, we can see that the metrics (package-modularization metrics + traditional complexity metrics) are clustered into six to seven distinct orthogonal principle components (PCs). These PCs explain around 65–85% of the variances in the data sets. Furthermore, we can observe that there are three types of PCs: (1) type I: consisting of only traditional metrics; (2) type II: consisting of only modularization metrics; and (3) type III: consisting of both metrics. As can be seen, for each data set, there are at least two type II PCs. In particular, for Cxf 2.1, Flume 1.4.0, Hbase 0.96.0 and Hive 0.12.0, the number of type II PCs is even equal to or larger than the total number of type I and type III PCs. These results clearly indicate that Sarkar et al.'s package-modularization metrics are complementary to traditional complexity metrics. Therefore, the PCA analysis results, from six different data sets, consistently support that Sarkar et al.'s package-modularization metrics capture different aspects of complexity that are not captured by traditional package-level complexity metrics.

5.2. Association with package fault-proneness

To answer RQ2, we use univariate logistic regression to analyze the association between Sarkar et al.'s package-modularization metrics and fault-proneness. Table 6 shows for each metric ΔOR

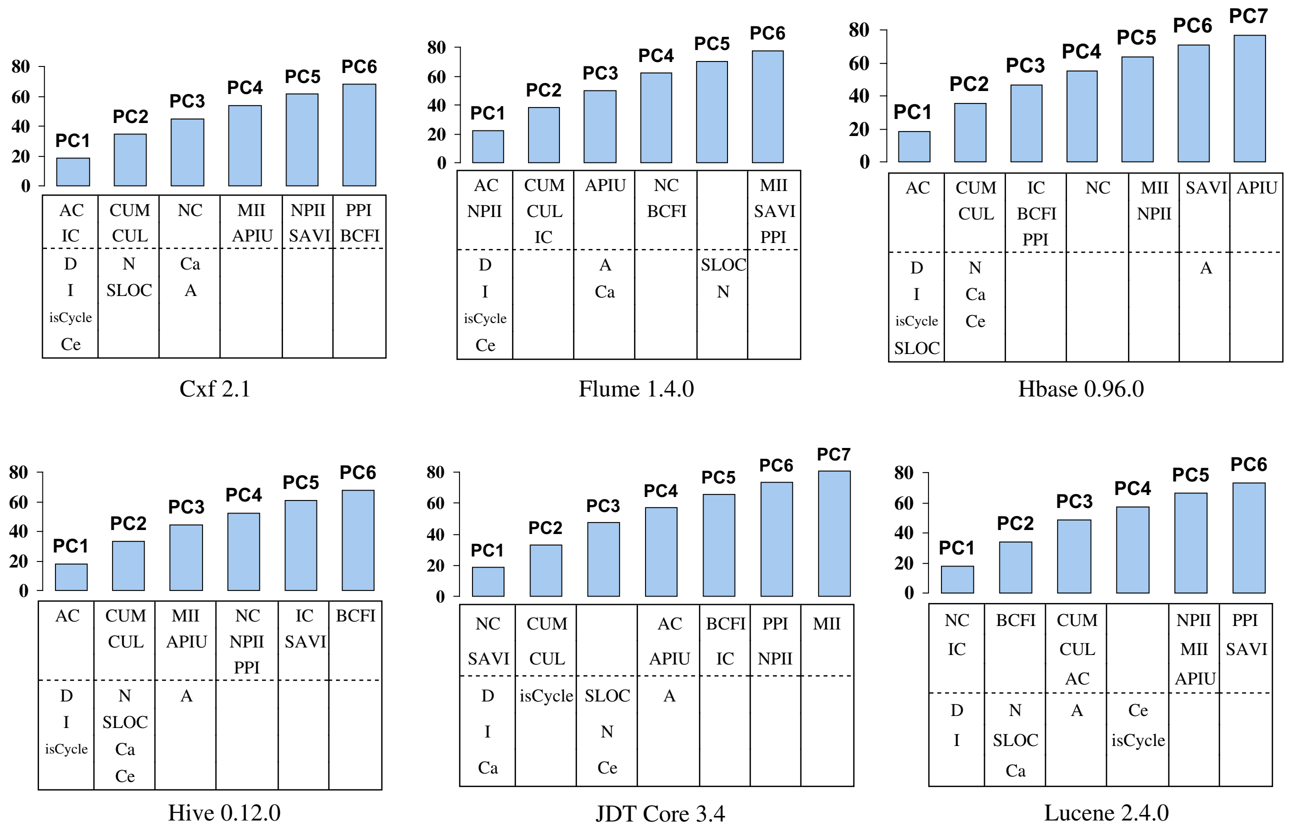

Fig. 3. Rotated components from PCA.

(the odds ratio associated with one standard deviation increase), the magnitude of the association with fault-proneness, in the six data sets. Table 7 shows ΔOR for each metric after removing the confounding effect of package size in the six data sets. In both tables, each cell with a ΔOR value means that the metric in the corresponding data set is associated with fault-proneness at the significant level of 0.05 (denoted by $***$), 0.10 (denoted by $**$), or 0.20 (denoted by $*$). The blank cell means that the metric is not significant at the significance level of 0.20. The second and third columns respectively count for each metric the number of positive associations (i.e. $\Delta OR > 1$) and the number of negative associations (i.e. $\Delta OR < 1$) over different data sets.

As described in Section 2.1, for each package-modularization metric, a larger value means a better modularization quality. Therefore, it is expected that there is a negative association between each modularization metric and fault-proneness. From Table 6, we can see that software modularization metrics can be classified into three categories. The first category of metrics,

Table 6
The magnitude of association with fault-proneness before removing the confounding effect of package size in the six data sets.

Metric	+	–	Cxf 2.1	Flume 1.4.0	Hbase 0.96.0	Hive 0.12.0	JDT Core 3.4	Lucene 2.4.0
MII	3	0			1.721**	1.288*		54.363***
NC	1	1	0.826*	2.566**				
BCFI	0	4	0.661***	0.707*	0.780*		0.425***	
IC	0	2			0.526***	0***		
NPII	0	2			0.699*			0.342***
AC	0	3	0.707***	0.625**	0.722**			
SAVI	0	3	0.836*			0.740**	0.588*	
CUM	0	5	0.333***	0.245***	0.405***	0.088***	0.384***	
CUL	0	5	0.368***	0.337***	0.601***	0.023***	0.411**	
PPI	0	2	0.702*			0.327*		
APIU	0	2	0.716**		0.305***			
N	6	0	1.594***	3.001***	4.577***	1.875***	216.5***	52.835***
Ca	5	0	2.047***	3.280***	1.799***	2.530**	7.621***	
Ce	6	0	2.839***	2.594***	6.117***	2.963***	11.090***	2.833***
A	1	0						1.523*
I	1	1		1.936**				0.515***
D	1	1		0.528*				1.846***
isCycle	4	0	3.766***	2.079**	1.876***	1.876***		
SLOC	6	0	4.784***	6.391***	678.595***	21.767***	77400.531***	198.65***

Table 7
The magnitude of association with fault-proneness after removing the confounding effect of package size in the six data sets.

Metric	+	–	Cxf 2.1	Flume 1.4.0	Hbase 0.96.0	Hive 0.12.0	JDT Core 3.4	Lucene 2.4.0
MII	3	0			1.720**	1.293*		43.166***
NC	0	1	0.832*					
BCFI	0	4	0.662***		0.707*	0.776*		0.541***
IC	0	2			0.524***		0***	
NPII	0	2			0.698*			0.002***
AC	0	4	0.746***	0.571*	0.619**	0.710**		
SAVI	0	1				0.741**		
CUM	0	5	0.438***		0.245***	0.415***	0.652***	0.907*
CUL	0	4	0.480***		0.315***	0.617***	0.241***	
PPI	1	2		0***		0.376*		47.642**
APIU	0	3	0.706**		0.307***			0.330***
N	5	0	1.203*	1.730*	5.149***	1.985***		6.561***
Ca	4	0	1.843***		3.686***	1.816***		5.510***
Ce	6	0	2.582***	2.240***	6.121***	2.913***	2.906***	2.292***
A	1	0		1.612*				
I	1	1		2.136***				0.609**
D	1	1		0.463**				1.596**
isCycle	5	1	2.237***	2.250*	2.348**	1.974***	0***	4.855e+24***
SLOC	6	0	4.784***	6.391***	678.595***	21.767***	77400.531***	198.65***

consisting of only MII, exhibits a positive association with fault-proneness in three data sets. The second category of metrics, consisting of only NC, exhibits a negative association in one data set and a positive association in another data set. The third category of metrics, consisting of the remaining nine modularization metrics, exhibits a negative association in at least two data sets. What is more, of these nine modularization metrics, CUM and CUL are negatively significant in almost all the data sets. In contrast, except I and D, all of the traditional complexity metrics show a positive association with fault-proneness. Similar results can be seen from Table 7. Particularly, we observe that, there are two main changes after removing the potential confounding effect of package size: (1) some metrics become more significantly associated with fault-proneness, including A, I and D in Flume 1.4.0, BCFI, AC, and Ca in Hive 0.12.0, IC, CUL and isCycle in JDT Core 3.4, and BCFI, CUM, APIU, Ca and isCycle in Lucene 2.4.0 and (2) some metrics become not significantly associated with fault-proneness any more, including SAVI and PPI in Cxf 2.1, NC, BCFI, CUM, CUL and Ca in Flume 1.4.0, IC in Hive 0.12.0, BCFI, SAVI, N and Ca in JDT Core 3.4, and A in Lucene 2.4.0.

Overall, the univariate analysis results, from six different data sets, show that most of software modularization metrics have a significantly negative association with fault-proneness. This is in contrast to traditional complexity metrics, most of which have a significantly positive association with fault-proneness.

5.3. Ability to predict package fault-proneness

In order to answer RQ3, we first use the procedure described in Section 3.4 to build the "T" model and the "M+T" model for each data set. Then, we compare the predictive effectiveness of the "T" and "M+T" models with respect to both ranking and classification.

5.3.1. Model fitting

Table 8 summarizes the detailed descriptive information of each multivariate fault-proneness prediction model. We build two types of multivariate logistic regression models (i.e. "T" model and "M+T" model) for each data set. In Table 8, the third and fourth columns respectively list for each model the R^2 and the number of influential observations. Note that the R^2 is a measure of goodness of fit for a logistic regression model [10]. A higher R^2 indicates a more accurate model. In practice, the R^2s for logistic regression are not very high compared with the least-square regression R^2's. There are three rows for each model in the last column which show the metrics (in the first row), the regression coefficients (in the second row), and the corresponding p-value (in the third row) for each model.

From Table 8, we make the following two observations. First, for each data set, many package-modularization metrics are included in the "M+T" model. This again suggests that Sarkar et al.'s package-modularization metrics are complementary to traditional package complexity metrics. Second, for most data sets, the "M+T" model has a considerably larger R^2 than the "T" model. This indicates that Sarkar et al.'s package-modularization metrics have the potential to improve the performance of package fault-proneness prediction when used with traditional package-level complexity metrics together. In Sections 5.3.2 and 5.3.3, we will investigate the extent of improvement by package-modularization metrics.

5.3.2. Ranking performance

In practice, given the possibly limited resource, only the packages at the top fraction of a fault-proneness ranking list might be chosen for inspection or testing. Therefore, practitioners are more interested in the ranking performance of a prediction model at the top fraction. Table 9 summarizes the CEs at the cut-off $\pi = 0.1$ and 0.2 obtained from 30 times 3-fold cross-validation for different fault-proneness prediction models. In each table, the second column reports the average CE and the corresponding standard deviation from 30 times 3-fold cross validation for "T" model. The third column not only reports similar information for "M+T" model, but also shows how often "M+T" model performs significantly better (denoted by $\surd$) or worse (denoted by $\times$) than "T" model by the Wilcoxon's signed-rank test. The fourth column reports the improvement in percentage using "M+T" model over "T" model in terms of the CE. The row "Average" reports the average CE (under the second and third columns) and the average improvement (under the fourth column) over the six data sets. In particular, the row "Win/tie/loss" reports: (1) the number of data sets for which "M+T" model obtains a better, equal, and worse CE than "T" model (under the third column); and (2) the number of data sets for which the "% improved CE" is greater than, is equal to, and is less than zero (under the last column).

From Table 9 (a), we can see that "M+T" model significantly outperforms "T" model in all the six data sets. In terms of CE at $\pi = 0.1$, the average improvement is more than 30% over "T" model.

Table 8
"T" and "M+T" models in the six data sets.

System	Model	R^2	cooks'd > 1	Metrics										
Cxf 2.1	T	0.268	1	Constant	Ca	Ce	isCycle	SLOC						
				−4.367	0.032	0.104	1.892	0.001						
				<0.001	0.002	0.003	0.074	<0.001						
	M+T	0.340	0	Constant	MII	NC	BCFI	CUM	PPI	Ca	Ce	D	isCycle	SLOC
				−4.260	−2.626	2.237	−0.946	−3.775	−5.839	0.033	0.158	2.307	2.978	<0.001
				0.051	0.070	0.034	0.068	0.005	0.082	0.029	<0.001	0.041	0.023	<0.001
Flume 1.4.0	T	0.395	0	Constant	N	I	SLOC							
				−6.681	0.213	4.429	0.002							
				0.003	0.032	0.018	0.006							
	M+T	0.463	5	Constant	NC	BCFI	N	SLOC						
				−10.894	14.260	−7.503	0.457	<0.001						
				0.036	0.017	0.057	0.019	0.067						
Hbase 0.96.0	T	0.339	0	Constant	Ce									
				−3.084	0.298									
				<0.001	<0.001									
	M+T	0.549	1	Constant	IC	APIU	Ce	D						
				15.492	−3.998	−36.453	0.543	3.952						
				0.066	0.006	0.036	<0.001	0.038						
Hive 0.12.0	T	0.176	0	Constant	Ca	Ce								
				−2.217	0.034	0.152								
				<0.001	0.048	<0.001								
	M+T	0.244	0	Constant	CUM	CUL	PPI	Ce						
				−0.302	−6.431	4.086	−10.777	0.173						
				0.771	0.008	0.047	0.033	<0.001						
JDT Core 3.4	T	0.484	0	Constant	SLOC									
				−2.234	0.001									
				0.020	0.004									
	M+T	0.725	0	Constant	NC	AC	CUL	SLOC						
				20.985	−9.603	−8.228	−29.398	0.002						
				0.086	0.091	0.091	0.077	0.036						
Lucene 2.4.0	T	0.435	0	Constant	Ca	isCycle								
				−168.049	0.121	164.977								
				0.010	0.046	0.011								
	M+T	0.582	0	Constant	NC	N	Ca	isCycle						
				−246.370	23.452	0.168	0.347	219.505						
				0.022	0.076	0.036	0.047	0.026						

Table 9
Ranking performance of logistic regression models in terms of CE.

System	T model	M+T model	%Improved
(a) π = 0.1			
Cxf 2.1	0.123 ± 0.106	0.163 ± 0.105 √	32.520
Flume 1.4.0	0.209 ± 0.371	0.332 + 0.364 √	58.852
Hbase 0.96.0	0.442 ± 0.217	0.529 ± 0.220 √	19.683
Hive 0.12.0	0.468 ± 0.109	0.535 ± 0.110 √	14.316
JDT Core 3.4	0.444 ± 0.217	0.627 ± 0.201 √	41.216
Lucene 2.4.0	0.132 ± 0.296	0.247 ± 0.302 √	87.121
Average	0.303	0.406	33.993
Win/tie/loss	–	6/0/0	6/0/0
(b) π = 0.2			
Cxf 2.1	0.194 ± 0.111	0.249 ± 0.097 √	28.351
Flume 1.4.0	0.331 ± 0.301	0.395 + 0.290	19.335
Hbase 0.96.0	0.530 ± 0.216	0.617 ± 0.213 √	16.415
Hive 0.12.0	0.566 ± 0.103	0.619 ± 0.104 √	9.364
JDT Core 3.4	0.598 ± 0.194	0.661 ± 0.166 √	10.535
Lucene 2.4.0	0.165 ± 0.263	0.283 ± 0.246 √	71.515
Average	0.397	0.471	18.640
Win/tie/loss	–	5/1/0	6/0/0

Similar results can be observed from Table 9(b). In terms of CE at $\pi = 0.2$, the average improvement is more than 18% over "T" model. These results clearly show that, when ranking fault-prone packages, "M+T" model leads to a substantial improvement of inspection or testing effectiveness over "T" model at the top fraction of the fault-proneness ranking list. These results indicate that using Sarkar et al.'s package-modularization metrics and traditional complexity metrics together is substantially more effective than using traditional metrics alone in the fault-proneness ranking scenario.

5.3.3. Classification performance

Table 10 summarizes the ERs obtained from 30 times 3-fold cross-validation for different fault-proneness prediction models. From Table 10(a), we can see that, "M+T" model significantly outperforms "T" model in two of the six data sets, and has a similar performance in the other data sets. In terms of ER-BPP, "M+T" model has an improvement over "T" model in all the six systems, with an average improvement of 23%. On average, "M+T" model results in 42.9% reduction in code inspection or testing, while "T" model only results in 34.6% effort reduction. Similar results can be observed from Table 10(c). What is more, in terms of ER-AVG, "M+T" model significantly outperforms "T" model in five out of the six data sets. In particular, the average improvement is over 42% in terms of ER-AVG.

Table 10
Classification performance of logistic regression models in terms of ER.

System	T model	M+T model	% Improved
(a) ER-BPP			
Cxf 2.1	0.370 ± 0.102	0.406 ± 0.165 √	9.730
Flume 1.4.0	0.249 ± 0.416	0.270 ± 0.399	8.434
Hbase 0.96.0	0.618 ± 0.483	0.700 ± 0.348	13.269
Hive 0.12.0	0.793 ± 0.085	0.803 ± 0.077	1.261
JDT Core 3.4	0.153 ± 0.486	0.176 ± 0.314	15.033
Lucene 2.4.0	−0.106 ± 0.730	0.219 ± 0.638 √	306.604
Average	0.346	0.429	23.988
Win/tie/loss	–	2/4/0	6/0/0
(b) ER-BCE			
Cxf 2.1	0.390 ± 0.175	0.427 ± 0.135 √	9.487
Flume 1.4.0	0.239 ± 0.461	0.286 ± 0.406	19.665
Hbase 0.96.0	0.725 ± 0.302	0.727 ± 0.300	0.276
Hive 0.12.0	0.805 ± 0.078	0.817 ± 0.072	1.491
JDT Core 3.4	0.400 ± 0.540	0.231 ± 0.337 ×	−42.250
Lucene 2.4.0	−0.023 ± 0.724	0.173 ± 0.673 √	854.174
Average	0.423	0.444	4.965
Win/tie/loss	–	2/3/1	5/0/1
(c) ER-MFM			
Cxf 2.1	0.299 ± 0.082	0.377 ± 0.117 √	26.087
Flume 1.4.0	0.196 ± 0.304	0.188 ± 0.394	−4.082
Hbase 0.96.0	0.562 ± 0.212	0.675 ± 0.166 √	20.107
Hive 0.12.0	0.373 ± 0.183	0.466 ± 0.204 √	24.933
JDT Core 3.4	0.104 ± 0.159	0.055 ± 0.203 ×	−47.115
Lucene 2.4.0	0.032 ± 0.591	0.184 ± 0.547 √	475.000
Average	0.261	0.324	24.138
Win/tie/loss	–	4/1/1	4/0/2
(d) ER-AVG			
Cxf 2.1	0.103 ± 0.141	0.203 ± 0.104 √	97.087
Flume 1.4.0	0.123 ± 0.185	0.138 ± 0.188	12.195
Hbase 0.96.0	0.377 ± 0.252	0.520 ± 0.205 √	37.931
Hive 0.12.0	0.405 ± 0.163	0.438 ± 0.144 √	8.148
JDT Core 3.4	0.218 ± 0.146	0.292 ± 0.138 √	33.945
Lucene 2.4.0	−0.009 ± 0.369	0.143 ± 0.343 √	1688.890
Average	0.203	0.289	42.365
Win/tie/loss	–	5/1/0	6/0/0

These results clearly show that, when classifying fault-prone packages, "M+T" model leads to a substantial improvement of inspection or testing effectiveness over "T" model. Therefore, the experimental results suggest that using Sarkar et al.'s package-modularization metrics and traditional complexity metrics together is substantially more effective than using traditional complexity metrics alone in the fault-proneness classification scenario.

Combining the results from Sections 5.3.2 and 5.3.3, we conclude that the combination of Sarkar et al.'s package-modularization metrics with traditional complexity metrics has a better ability to predict fault-proneness than the combinations of traditional metrics. This conclusion is consistent with intuition, as the experimental results in Section 5.1 show that package-modularization metrics are complementary to traditional complexity metrics. As such, using Sarkar et al.'s package-modularization metrics and traditional complexity metrics together should provide a better ability to predict fault-proneness.

6. Discussion

In this section, we further discuss our findings, including the reasons that modularization metrics can improve the effectiveness of fault-proneness prediction, the comparison to prior work, and the implications for producing dependable systems.

6.1. Why package-modularization metrics can improve fault-proneness prediction?

The results in Section 5.3 show that Sarkar et al.'s package-modularization metrics can substantially improve the effectiveness of fault-proneness prediction in most systems when used with traditional package-level metrics together. The reason for this is that, compared with traditional package-level metrics, Sarkar et al.'s package-modularization metrics capture additional information of software complexity that is related to fault-proneness. As described in Section 2, traditional package-level metrics measure software complexity mainly from size, extensibility, responsibility, independence, abstractness, and instability perspectives. However, Sarkar et al.'s package-modularization metrics measure software complexity mainly from the perspective of method call traffic, inheritance/association-induced coupling, state access violations, fragile base-class design, programming to interface, size uniformity, and plugin pollution. Given the nature of the information and counting mechanism employed by package-modularization metrics, it is understandable that they can capture additional information for software complexity compared with traditional package-level metrics. Indeed, their complementariness has been clearly evidenced by the PCA results in Section 5.1, in which package-modularization metrics even have at least two PC of their own in all data sets. Furthermore, as shown in Section 5.3.1, for each data set, package-modularization metrics are included in the "M+T" model. In particular, the "M+T" model has a larger R^2 than the "T" model. This means that the combination of package-modularization metrics and traditional metrics can explain more variations in fault data than traditional package-level metrics alone. This is the exact reason why Sarkar et al.'s package-modularization metrics can improve fault-proneness prediction effectiveness.

6.2. How do our results compared with prior work?

There exist a number of studies that predict the fault-proneness of packages in object-oriented software systems [17,23,31]. In [31], Zimmermann et al. first aggregated many metrics at the method-level, at the class-level, and at the file-level metrics to obtain package-level metrics. Then, they investigated the prediction performance of those package-level metrics in both the ranking and classification scenarios. Compared with Zimmermann et al.'s work, we used package-level metrics rather than the aggregated method/class-level metrics. Furthermore, we used the effort-aware performance indicators to evaluate the effectiveness of fault-proneness but they did not. In [17], Kamei et al. investigated the effort-aware ranking performance of fault-proneness prediction models built with Martin's metrics suite (i.e. traditional metrics in our study). However, it is difficult to directly compare our results with Kamei et al.'s results, as different performance indicators were used. In [23], Hata et al. investigated the effort-aware ranking performance of package-level prediction models. In their study, they used random forest and process-related metrics to build fault-proneness prediction models. In particular, they leveraged the probability returned by the random forest model to rank the fault-proneness of packages. Their results showed that, when inspecting top 20% of SLOC, the median value of the percentage of found bugs was 21.5%, only marginally better than a random selection model. In our study, when inspecting the top 20% SLOC, the median value of the percentage of found bugs is 52% (for "M+T" model). The large performance difference between our study and Hata et al.'s study may be attributed to the following three reasons. First, different metrics were used to build fault-proneness prediction models (traditional complexity metrics + modularization metrics vs. process-related metrics). Second, different modeling techniques were employed to build fault-proneness prediction models (logistic regression vs. random forest). Third, different risk formulae were used to rank packages fault-proneness (the predicted probability of being faulty per SLOC vs. the predicted probability of being faulty per package). To the best of our knowledge, no previous studies have validated the ability of Sarkar et al.'s package-modularization metrics to predict package-level fault-proneness.

The closest work to ours is the work by Kuo [24]. In his study, Kuo empirically investigated the correlations between three software modularization metrics (i.e. MQ [25], IC, and AC) with the fix response time or defect density. Based on forty-four open source Eclipse projects, their results showed that there was no obvious correlation. There are a number of differences between our study and Kuo's study. First, the IC and AC investigated in Kuo's study were system-level modularization metrics, while the IC and AC investigated in our study were package-level modularization metrics. For a given system, the system-level IC(AC) was equal to the average of package-level IC(AC) values over all packages in the system. Second, the dependent variables investigated in Kuo's study were the fix response time and defect density, while the dependent variable investigated in our study was fault-proneness. In particular, we used effort-aware performance indicators to evaluate the ability of package-modularization metrics in both the ranking and the classification scenarios. Our experimental results provide valuable data for better understanding Sarkar et al.'s package-modularization metrics and for guiding the development of better fault-proneness prediction models in practice.

6.3. What are the implications for producing dependable systems?

Software systems are particularly important for modern society, as they are providing critical services in a wide variety of fields and applications such as traffic control, power plants, and healthcare [26–28]. In this context, one important challenge is how to maintain their dependability, i.e. the ability to deliver services that can justifiably be trusted [27], in spite of continuous changes. As stated in [27], dependability can be achieved by the combined utilization of the following four techniques: fault prevention, fault removal, fault tolerance, and fault forecasting.

Our results in this paper have important implications for fault prevention and fault forecasting. First, it is long believed that modularization is one of the key determinants for developing quality complex modern software systems from the viewpoint of fault prevention [27]. However, the relationships between many software modularization practices and software dependability remain poorly understood. Our results in Section 5.2 evidence that software quality could be improved: (1) by minimizing association/inheritance-induced coupling (AC, IC), the base-class fragility (BCFI), state-access-violation (SAVI), and plugin code bloat (PPI); and (2) by maximizing module size uniformity (CU_m, CU_l), the adherence to programming-to-interface (NPII) and the module coherence on the basis of commonality of goals (APIU). However, further research is needed to understand the influence of maximizing API-based inter-module call traffic (MII, NC). Second, our results in Section 5.3 evidence that modularization metrics can substantially improve fault-proneness prediction effectiveness from the viewpoint of fault forecasting. In the ranking scenario, when inspecting or testing 10% SLOC of a system, the average cost-effectiveness (CE) improvement over six systems is more than 33%. This means that, if modularization metrics are used with traditional metrics together, more faults will be identified when the same inspection or testing effort is spent. In the classification scenario, the average effort-reduction (ER-AVG) improvement over six systems is more than 42%. This means that, if modularization metrics are used with traditional metrics together, more effort will be saved compared with random selection to achieve the same recall of faults. In summary, our findings in this paper could guide developers to choose the appropriate modularization practices to prevent faults and to develop cost-effectiveness models to predict faults when producing dependable software systems.

7. Threats to validity

In this section, we discuss the most important threats to the construct, internal, and external validity of our study. Construct validity is the extent to which the dependent and independent variables accurately measure the concept they purport to measure. Internal validity is the extent to which the conclusions can be drawn about the causal effect of independent variables on the dependent variables. External validity is the extent to which the conclusions can be generalized to the population under study and other research settings.

The most important threat to the construct validity of our study is the accuracy of the dependent and independent variables. The dependent variable in this study is a binary variable which represents the detection or non-detection of a post-release fault in a package. In our study, for the former four systems (i.e. Cxf 2.1, Flume 1.4.0, Hbase 0.96.0, and Hive 0.12.0), we obtained the fault data from apache's issue tracking system JIRA https://issues.apache.org/jira/. According to [29], the fault data in JIRA has a high quality. For the latter two systems (i.e. JDT Core 3.4 and Lucene 2.4.0), we used the fault data provided in the website http://bug.inf.usi.ch/, which are obtained by a so-called SZZ algorithm (designed by Śliwerski et al. [30]). The SZZ algorithm is the most commonly used approach for mining fault data from open-source software and have been widely used in previous software engineering literature [18,23,30–37]. The independent variables used in this study are Sarkar et al.'s package-modularization metrics and traditional complexity metrics. We used a mature commercial tool "Understand" to collect the metric data. "Understand" is a well-known source code analysis tool that is actually being used successfully in practice. In our study, all metric data were collected by a Perl script in conjunction with the corresponding Understand databases. To ensure the reliability of data collection, the fourth author doubly checked the Perl script developed by the first author. Furthermore, we applied the Perl script to a number of packages developed by ourselves and found that the reported results were correct in our test. In this sense, the construct validity of the dependent and independent variables in our study can be considered as satisfactory.

The most important threat to the internal validity of our study is the exclusion of few packages from Cxf 2.1, JDT Core 3.4, and

Table 11
Rotated components from PCA for Cxf 2.1.

	PC1	PC2	PC3	PC4	PC5	PC6
Eigenvalue	3.502	3.048	1.943	1.741	1.466	1.222
Proportion (%)	18.432	16.04	10.225	9.166	7.717	6.429
Cumulative (%)	18.432	34.472	44.697	53.863	61.58	68.009
D	**0.859***	0.021	−0.095	−0.257	−0.014	−0.026
AC	**0.837***	0.043	−0.047	0.261	0.121	0.008
I	**−0.829***	−0.024	0.432	−0.060	0.048	−0.006
isCycle	**−0.793***	0.251	−0.182	−0.024	−0.110	−0.061
Ce	**−0.533***	0.503	−0.069	−0.098	−0.243	−0.110
IC	**0.488***	0.202	−0.189	0.286	0.321	0.426
N	−0.004	**0.832***	−0.048	0.104	0.018	0.092
SLOC	−0.106	**0.803***	0.031	−0.087	0.087	0.121
CUM	−0.029	**−0.781***	0.185	−0.103	0.210	0.180
CUL	0.015	**−0.776***	0.170	−0.190	0.250	0.200
NC	−0.005	−0.020	**0.794***	0.025	−0.025	0.082
Ca	0.115	0.270	**−0.678***	−0.069	−0.088	−0.009
A	0.095	0.017	**−0.591***	0.557	−0.084	0.078
MII	0.040	0.073	−0.134	**0.738***	−0.145	−0.072
APIU	0.015	0.069	0.208	**0.737***	0.018	0.009
NPII	0.237	−0.140	−0.138	−0.018	**0.714***	−0.070
SAVI	−0.043	−0.116	0.244	−0.167	**0.672***	0.040
PPI	−0.163	−0.144	−0.017	−0.117	−0.287	**0.715***
BCFI	0.324	0.032	0.162	0.044	0.283	**0.626***

Lucene 2.4.0 when generating the data sets. As described in Section 4, the reason for exclusion is that the fault data of those excluded packages are unavailable. However, we believe that the exclusion of those packages should not have a substantial influence on our conclusions, as only a very small proportion of packages are excluded.

The most important threat to the external validity of our study is that our conclusions may not be generalized to other systems. In our study, we used six systems to investigate the three research questions. However, all these systems are open-source Java systems. Therefore, our conclusion may not be generalized either to those systems developed by other programming languages or to those systems that are not open-source. This is an inherent problem for all studies in empirical software engineering, not unique to us, as many factors cannot be characterized completely. To mitigate this threat, there is a need to replicate our study using a wide variety of systems in the future work.

8. Conclusion and future work

In this paper, we empirically evaluate the usefulness of a set of newly proposed package-modularization metrics in predicting fault-prone packages in object-oriented software systems. Our study validates these modularization metrics on six open-source Java systems. Our first finding shows that Sarkar et al.'s modularization metrics capture new and complementary views of software complexity compared with traditional metrics. Our second finding shows that most of Sarkar et al.'s modularization metrics exhibit a significantly negative association with fault-proneness. Our third finding, the most important finding, reveals that Sarkar et al.'s package-modularization metrics can substantially improve the effectiveness of fault-proneness prediction when used with traditional metrics together. In the ranking scenario, on average, they lead to more than 33% effectiveness improvement in identifying fault-prone packages when inspecting or testing 10% SLOC of a sys-

Table 12
Rotated components from PCA for Flume 1.4.0.

	PC1	PC2	PC3	PC4	PC5	PC6
Eigenvalue	4.255	2.974	2.299	2.244	1.525	1.442
Proportion (%)	22.396	15.653	12.1	11.809	8.025	7.59
Cumulative (%)	22.396	38.048	50.148	61.958	69.982	77.572
D	**−0.916***	−0.096	−0.043	−0.136	0.019	0.029
I	**0.882***	−0.026	−0.262	0.222	−0.092	−0.066
isCycle	**0.797***	0.139	0.066	−0.320	−0.071	0.166
Ce	**0.781***	0.146	0.156	0.094	0.246	−0.053
AC	**−0.779***	0.272	−0.084	0.171	0.001	0.107
NPII	**−0.613***	0.441	0.008	−0.166	0.013	0.027
CUL	−0.117	**−0.927***	−0.143	0.090	−0.073	−0.073
CUM	0.087	**−0.879***	−0.126	−0.053	−0.082	0.061
IC	−0.464	**0.587***	0.204	0.339	0.221	0.037
APIU	0.060	0.042	**0.819***	0.032	−0.065	0.082
A	0.086	0.489	**0.705***	0.014	0.059	0.172
Ca	−0.106	0.188	**0.658***	−0.596	0.162	−0.004
NC	0.161	−0.076	−0.338	**0.852***	0.063	−0.051
BCFI	−0.060	0.137	0.290	**0.744***	0.110	−0.035
SLOC	−0.106	0.029	−0.133	0.154	**0.926***	−0.047
N	0.286	0.404	0.349	−0.104	**0.692***	0.104
MII	0.019	0.134	0.210	0.230	−0.037	**0.821***
SAVI	−0.108	−0.404	0.354	0.367	0.010	**−0.640***
PPI	0.136	0.176	−0.086	0.156	−0.032	**−0.496***

Table 14
Rotated components from PCA for Hive 0.12.0.

	PC1	PC2	PC3	PC4	PC5	PC6
Eigenvalue	3.373	2.936	2.084	1.598	1.532	1.3
Proportion (%)	17.752	15.453	10.97	8.412	8.063	6.844
Cumulative (%)	17.752	33.204	44.174	52.587	60.649	67.494
I	**−0.880***	−0.112	−0.162	−0.212	0.208	0.157
D	**0.844***	0.147	−0.179	0.100	−0.217	−0.112
isCycle	**−0.837***	0.152	0.040	−0.029	−0.105	−0.166
AC	**0.760***	0.044	0.274	0.000	−0.042	0.180
N	−0.027	**0.823***	0.020	0.077	0.054	0.064
CUL	−0.008	**−0.712***	−0.395	0.171	0.358	0.044
SLOC	0.224	**0.694***	−0.148	0.029	0.242	0.118
CUM	0.116	**−0.623***	−0.392	0.122	0.400	0.000
Ca	0.254	**0.618***	−0.030	0.178	−0.243	−0.389
Ce	−0.479	**0.612***	−0.019	0.043	−0.116	−0.229
MII	0.102	0.062	**0.748***	−0.172	0.010	−0.240
A	0.157	−0.062	**0.682***	0.326	−0.196	0.034
APIU	−0.020	0.059	**0.589***	0.007	0.106	0.169
NC	−0.258	0.060	−0.116	**−0.699***	0.301	0.201
NPII	0.171	0.018	−0.260	**0.681***	0.017	0.128
PPI	0.212	−0.143	−0.340	**−0.584***	−0.191	0.002
SAVI	−0.078	−0.026	0.099	−0.009	**0.775***	0.013
IC	0.247	0.169	0.197	0.172	**−0.476***	0.361
BCFI	0.083	−0.043	−0.010	−0.015	−0.049	**0.840***

Table 13
Rotated components from PCA for Hbase 0.96.0.

	PC1	PC2	PC3	PC4	PC5	PC6	PC7
Eigenvalue	3.434	3.346	2.028	1.625	1.604	1.456	1.138
Proportion (%)	18.076	17.613	10.676	8.551	8.441	7.666	5.99
Cumulative (%)	18.076	35.688	46.364	54.915	63.357	71.022	77.012
I	**−0.893***	−0.109	−0.121	0.266	−0.123	0.157	−0.040
D	**0.870***	0.017	0.127	−0.244	−0.171	0.104	−0.010
isCycle	**−0.825***	0.252	−0.119	0.024	0.177	−0.010	0.085
AC	**0.744***	−0.102	0.192	0.274	0.104	−0.284	0.070
SLOC	**0.460***	0.244	0.003	0.238	−0.053	0.440	0.037
N	0.014	**0.903***	0.015	0.010	0.116	0.077	−0.052
Ca	0.191	**0.851***	0.059	−0.288	0.111	0.078	−0.084
Ce	−0.409	**0.793***	−0.109	0.048	0.001	0.019	−0.077
CUL	−0.084	**−0.693***	0.078	−0.579	−0.066	0.145	−0.027
CUM	0.171	**−0.692***	0.014	−0.458	−0.094	0.297	0.046
IC	0.106	−0.048	**0.852***	0.036	−0.020	−0.121	−0.049
BCFI	0.078	−0.087	**0.800***	−0.043	−0.048	0.141	0.056
PPI	−0.162	−0.088	**−0.743***	0.026	−0.098	0.060	−0.047
NC	−0.160	−0.021	−0.008	**0.820***	0.144	0.239	−0.016
MII	0.035	0.137	0.040	0.057	**0.860***	0.031	0.203
NPII	0.241	−0.094	0.017	−0.172	**−0.615***	−0.050	0.483
SAVI	−0.231	−0.021	−0.031	0.128	0.113	**0.759***	−0.010
A	−0.090	0.117	0.008	0.082	0.544	**−0.591***	0.069
APIU	−0.052	−0.120	0.046	0.026	0.127	−0.014	**0.904***

Table 15
Rotated components from PCA for JDT Core 3.4.

	PC1	PC2	PC3	PC4	PC5	PC6	PC7
Eigenvalue	3.588	2.709	2.683	1.88	1.593	1.503	1.373
Proportion (%)	18.884	14.26	14.122	9.894	8.386	7.91	7.226
Cumulative (%)	18.884	33.145	47.267	57.161	65.546	73.456	80.682
I	**0.942***	−0.067	0.018	−0.120	0.036	−0.180	0.007
NC	**0.845***	−0.007	0.189	−0.086	−0.156	−0.056	0.196
Ca	**−0.719***	−0.308	0.307	−0.100	0.053	0.061	−0.067
D	**−0.713***	0.475	0.057	0.057	0.060	0.052	0.290
SAVI	**0.494***	0.347	−0.161	0.396	0.020	−0.217	−0.336
CUL	0.107	**0.831***	−0.333	−0.185	−0.012	0.120	−0.056
CUM	0.072	**0.809***	−0.313	−0.157	0.005	0.190	−0.050
isCycle	0.322	**−0.710***	0.012	−0.299	−0.077	0.145	0.141
SLOC	−0.102	−0.140	**0.939***	−0.087	−0.072	0.053	0.048
N	−0.051	−0.117	**0.900***	0.108	0.008	0.003	0.078
Ce	0.262	−0.362	**0.711***	−0.077	0.005	−0.351	−0.193
AC	−0.436	0.039	0.005	**0.793***	0.124	0.003	0.054
APIU	0.304	−0.094	0.096	**0.743***	−0.045	0.128	0.185
A	−0.309	−0.471	−0.254	**0.530***	−0.043	0.157	0.057
BCFI	0.099	0.025	−0.016	0.078	**0.901***	−0.144	0.068
IC	−0.267	0.024	−0.033	−0.044	**0.827***	0.187	0.019
PPI	0.115	−0.066	−0.023	−0.092	0.035	**−0.832***	−0.131
NPII	−0.210	0.049	−0.207	0.040	0.141	**0.652***	−0.519
MII	0.020	−0.105	−0.028	0.183	0.116	0.051	**0.854***

Table 16
Rotated components from PCA for Lucene 2.4.0.

	PC1	PC2	PC3	PC4	PC5	PC6
Eigenvalue	3.377	3.029	2.831	1.692	1.68	1.305
Proportion (%)	17.771	15.942	14.899	8.907	8.843	6.87
Cumulative (%)	17.771	33.713	48.613	57.52	66.362	73.233
D	**0.855***	0.315	0.122	−0.213	−0.018	−0.053
I	**−0.840***	−0.330	−0.210	0.224	0.094	0.034
NC	**−0.823***	0.025	0.201	−0.085	−0.176	0.037
IC	**0.603***	−0.304	0.369	0.443	−0.072	0.030
N	0.092	**0.844***	0.195	0.187	−0.151	0.054
SLOC	0.003	**0.822***	0.157	−0.035	−0.056	0.021
Ca	0.323	**0.782***	0.137	0.132	−0.110	−0.020
BCFI	0.068	**−0.614***	0.089	−0.030	0.341	0.022
CUM	−0.015	−0.281	**−0.869***	−0.056	−0.028	0.142
CUL	0.122	−0.289	**−0.794***	−0.102	−0.125	0.242
A	0.212	−0.039	**0.741***	0.096	−0.194	0.166
AC	0.548	−0.130	**0.551***	−0.342	0.039	0.060
Ce	−0.113	0.223	0.146	**0.857***	−0.093	−0.103
isCycle	−0.571	0.098	−0.045	**0.592***	−0.088	−0.021
NPII	0.151	−0.159	0.092	−0.080	**0.765***	−0.044
MII	0.191	0.054	0.273	0.039	**−0.679***	0.176
APIU	0.080	−0.242	0.058	−0.027	**0.602***	0.220
PPI	−0.015	−0.123	−0.091	0.250	0.001	**−0.790***
SAVI	−0.134	−0.100	−0.385	0.203	0.004	**0.684***

tem. In the classification scenario, on average, they lead to more than 42% effectiveness improvement in identifying fault-prone packages. This means that, by using package-modularization metrics, practitioners can detect considerably more fault-prone packages with the same inspection or testing effort. This is particularly important for developing highly dependable software systems, especially when the available inspection or testing resources are limited.

In the future work, we plan to extend our study in the following two directions: (1) replicate this study on large software systems developed by other programming languages such as C++ and C# and (2) replicate this study on large closed-source software systems. The purpose of this is to examine whether our conclusions drawn in this study can be generalized to other systems.

Acknowledgments

The authors would like to thank the editor and anonymous reviewers for very helpful suggestions in greatly improving the quality of this paper. The work in this paper is supported by the National Key Basic Research and Development Program of China (2014CB340702), the National Natural Science Foundation of China (91318301, 61432001, 61321491, 61300051, 61272082), and the National Natural Science Foundation of Jiangsu Province (BK20130014).

Appendix A. Principal component analysis results

Tables 11–16 show the rotated components from PCA for the six data sets. The second to fourth rows in the tables show the eigenvalue, the percentage of variance explained by each PC, and the cumulative percentage of variances. The remaining rows, starting with the fifth row, show the loading of each metric in each PC, which indicates the degree of correlation with the PC. In each row, the loading with the largest absolute value is shown in bold (∗ indicates that it is significant at $p < 0.01$). This enables us to observe which metrics capture the same property and hence belong to the same PC.

References

[1] H. Abdeen, S. Ducasse, H. Sahraoui, Modularization metrics: assessing package organization in legacy large object-oriented software, in: Proceedings of the 18th IEEE Working Conference on Reverse Engineering, 2011, pp. 394–398.
[2] S. Eick, T. Graves, A. Karr, J. Marron, A. Mockus, Does code decay? Assessing the evidence from change management data, IEEE Trans. Softw. Eng. 27 (1) (2001) 1–12.
[3] R. Martin, Agile Software Development: Principles, Patterns, and Practices, Prentice Hall PTR, 2003.
[4] M. Elish, A. Al-Yafei, M. Al-Mulhem, Empirical comparison of three metrics suites for fault prediction in packages of object-oriented systems: a case study of Eclipse, Adv. Eng. Softw. 42 (10) (2011) 852–859.
[5] S. Sarkar, A. Kak, G. Rama, Metrics for measuring the quality of modularization of large-scale object-oriented software, IEEE Trans. Softw. Eng. 34 (5) (2008) 700–720.
[6] T. Hall, S. Beecham, D. Bowes, S. Counsell, A systematic literature review on fault prediction performance in software engineering, IEEE Trans. Softw. Eng. 38 (6) (2012) 1276–1304.

[7] S. Lessmann, B. Baesens, C. Mues, S. Pietsch, Benchmarking classification models for software defect prediction: a proposed framework and novel findings, IEEE Trans. Softw. Eng. 34 (4) (2008) 485–496.
[8] N. Cliff, The eigenvalues-greater-than-one rule and the reliability of components, Psychol. Bull. 103 (2) (1988) 276–279.
[9] D. Belsley, E. Kuh, R. Welsch, Regression Diagnostics: Identifying Influential Data and Sources of Collinearity, John Wiley and Sons, New York, 1980.
[10] L. Briand, J. Wüst, J. Daly, D. Porter, Exploring the relationships between design measures and software quality in object-oriented systems, J. Syst. Softw. 51 (3) (2000) 245–273.
[11] K. Emam, S. Benlarbi, N. Goel, S. Rai, The confounding effect of class size on the validity of object-oriented metrics, IEEE Trans. Softw. Eng. 27 (7) (2001) 630–650.
[12] Y. Zhou, B. Xu, H. Leung, L. Chen, An in-depth study of the potentially confounding effect of class size in fault prediction, ACM Trans. Softw. Eng. Methodol. 23 (1) (2014).
[13] M. Kutner, C. Nachtsheim, J. Neter, Applied Linear Regression Models, fourth ed., McGraw-Hill Irwin, 2004.
[14] D. Belsley, E. Kuh, R. Welsch, Regression Diagnostics: Identifying Influential Data and Sources of Collinearity, John Wiley & Sons, 2005.
[15] E. Arisholm, L. Briand, B. Johannessen, A systematic and comprehensive investigation of methods to build and evaluate fault prediction models, J. Syst. Softw. 83 (1) (2010) 2–17.
[16] T. Mende, R. Koschke, Effort-aware defect prediction models, in: Proceedings of the 14th European Conference on Software Maintenance and Reengineering, 2010, pp. 107–116.
[17] Y. Kamei, S. Matsumoto, A. Monden, K. Matsumoto, B. Adams, A.E. Hassan, Revisiting common bug prediction findings using effort-aware models, in: Proceedings of the 26th IEEE International Conference on Software Maintenance, 2010, pp. 1–10.
[18] Y. Kamei, E. Shihab, B. Adams, A.E. Hassan, A. Mockus, A. Sinha, N. Ubayashi, A large-scale empirical study of just-in-time quality assurance, IEEE Trans. Softw. Eng. 39 (6) (2013) 757–773.
[19] T. Mende, R. Koschke, Revisiting the evaluation of defect prediction models, in: Proceedings of the 5th International Conference on Predictor Models in Software Engineering, 2009.
[20] Y. Shin, A. Meneely, L. Williams, J. Osborne, Evaluating complexity, code churn, and developer activity metrics as indicators of software vulnerabilities, IEEE Trans. Softw. Eng. 37 (6) (2011) 772–787.
[21] T. Menzies, J. Greenwald, A. Frank, Data mining static code attributes to learn defect predictors, IEEE Trans. Softw. Eng. 33 (1) (2007) 2–13.
[22] A. Schein, L. Saul, L. Ungar, A generalized linear model for principal component analysis of binary data, in: Proceedings of the 9th International Workshop on Artificial Intelligence and Statistics, 2003.
[23] H. Hata, O. Mizuno, T. Kikuno, Bug prediction based on fine-grained module histories, in: Proceedings of the 34th International Conference on Software Engineering, 2012, pp. 200–210.
[24] J. Kuo, Experimental Validation of Software Modularization Metrics, Master Thesis, National Central University, 2012.
[25] S. Mancoridis, B. Mitchell, C. Rorres, Y.F. Chen, E.R. Gansner, Using automatic clustering to produce high-level system organizations of source code, in: Proceedings of the 6th International Workshop on Program Comprehension, 1998, pp. 45–52.
[26] P. Alho, Cost-efficient Design of Dependable and Fault Tolerant Control System. <http://www.students.tut.fi/~alhop/review_dependability_Alho.pdf>.
[27] A. Azizienis, J. Laprie, B. Randell, Fundamental Concepts of Dependability, Research Report No. 1145, LAAS-CNRS, 2001.
[28] B. Littlewood, L. Strigini, Software reliability and dependability: a roadmap, in: Proceedings of the 22nd International Conference on Software Engineering, Future of Software Engineering Track, 2000, pp. 175–188.
[29] F. Rahman, D. Posnett, P. Devanbu, Recalling the "imprecision" of cross-project defect prediction, in: Proceedings of the 20th ACM SIGSOFT International Symposium on the Foundations of Software Engineering, ACM, 2012, p. 61.
[30] J. Śliwerski, T. Zimmermann, A. Zeller, When do changes induce fixes? in: Proceedings of the 2005 International Workshop on Mining Software Repositories, 2005, pp. 1–5.
[31] T. Zimmermann, R. Premraj, A. Zeller, Predicting defects for Eclipse, in: Proceedings of the 3rd International Workshop on Predictor Models in Software Engineering, 2007.
[32] S. Neuhaus, T. Zimmermann, C. Holler, A. Zeller, Predicting vulnerable software components, in: Proceedings of the 14th ACM Conference on Computer and Communications Security, 2007, pp. 529–540.
[33] S. Shivaji, E. Whitehead Jr., R. Akella, S. Kim, Reducing features to improve code change based bug prediction, IEEE Trans. Softw. Eng. 39 (4) (2013) 552–569.
[34] G. Bavota, B. Carluccio, A. Lucia, When does a refactoring induce bugs? An empirical study, in: Proceedings of the 12th International Working Conference on Source Code Analysis and Manipulation, 2012, pp. 104–113.
[35] A. Schroter, T. Zimmermann, A. Zeller, Predicting component failures at design time, in: Proceedings of the 2006 ACM/IEEE International Symposium on Empirical Software Engineering, 2006, pp. 18–27.
[36] D. Posnett, R. D'Souza, P. Devanbu, V. Filkov, Dual ecological measures of focus in software development, in: Proceedings of the 35th International Conference on Software Engineering, 2013, pp. 452–461.
[37] T. Mende, Replication of defect prediction studies: problems, pitfalls and recommendations, in: Proceedings of the 6th International Conference on Predictive Models in Software Engineering, 2010.
[38] Y. Kamei, E. Shihab, B. Adams, A.E. Hassan, A. Mockus, A. Sinha, N. Ybayashi, A large-scale empirical study of just-in-time quality assurance, IEEE Trans. Softw. Eng. 39 (6) (2013) 757–773.