The Journal of Systems and Software 85 (2012) 244–257

Contents lists available at ScienceDirect

The Journal of Systems and Software

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

Identifying thresholds for object-oriented software metrics

Kecia A.M. Ferreira*, Mariza A.S. Bigonha, Roberto S. Bigonha, Luiz F.O. Mendes, Heitor C. Almeida

Dept. Computer Science, Federal University of Minas Gerais (UFMG), Av. Antônio Carlos, 6627, Pampulha, CEP: 31270-010, Belo Horizonte, Brazil

ARTICLE INFO

Article history:
Received 1 March 2010
Received in revised form 15 April 2011
Accepted 23 May 2011
Available online 7 June 2011

Keywords:
Software metric thresholds
Open-source software
Object-oriented software

ABSTRACT

Despite the importance of software metrics and the large number of proposed metrics, they have not been widely applied in industry yet. One reason might be that, for most metrics, the range of expected values, i.e., reference values are not known. This paper presents results of a study on the structure of a large collection of open-source programs developed in Java, of varying sizes and from different application domains. The aim of this work is the definition of thresholds for a set of object-oriented software metrics, namely: LCOM, DIT, coupling factor, afferent couplings, number of public methods, and number of public fields. We carried out an experiment to evaluate the practical use of the proposed thresholds. The results of this evaluation indicate that the proposed thresholds can support the identification of classes which violate design principles, as well as the identification of well-designed classes. The method used in this study to derive software metrics thresholds can be applied to other software metrics in order to find their reference values.

1. Introduction

Software metrics allow measurement, evaluation, control and improvement of software products and processes. Much research has been carried out on the topic of software metrics. Dozens of metrics have been proposed and validated, and a large number of tools have been developed (Fenton and Neil, 2000; Xenos et al., 2000; Baxter et al., 2006; Kitchenham, 2009). Despite the important contribution of such works, the effective use of metrics in Software Engineering is inhibited by the lack of knowledge on software metric thresholds. Few empirical studies have been accomplished in order to derive these reference values (Lanza and Marinescu, 2006). These works are not based properly on statistical properties of the analyzed data. Besides, they are not concerned in investigating thresholds for object-oriented software metrics specifically.

The aim of the present work is to identify thresholds for a set of object-oriented software metrics, namely: LCOM, DIT, coupling factor, afferent couplings, number of public methods and number of public fields. This set of metrics was selected because they are related to relevant software quality factors, such as cohesion, coupling and information hiding.

We performed a study on the structure of a large collection of open-source programs developed in Java, with different sizes and application domains. The thresholds were derived by analyzing the statistical properties of the data and, then, by identifying the values most commonly used in practice. Our analysis concluded that five metrics investigated in this study are modeled by a heavy-tailed distribution, which means that there is no typical value for them. We present in detail the method used to derive thresholds for software metrics, in such way it can be repeated and applied to derive thresholds for other software metrics. We carried out four types of analyses: with the entire data set, which leads to general thresholds; by software system sizes; by application domains; and by types of software systems (tool, library and framework).

In this paper, we carried out two experiments to evaluate the proposed thresholds. In the first experiment, we investigated whether the proposed thresholds can help to identify classes with design problems. The second experiment was performed to assess whether the thresholds can support to identify well-designed classes. The results of this study indicate that the proposed thresholds are useful in helping to evaluate software system designs.

The paper is organised as follows. Section 2 discusses relevant related work. Section 3 provides background on the analysed metrics and describes the methods used in this research. Section 4 presents results of this study and their analysis. Section 5 identifies the software metric thresholds suggested in this work. Section 6 describes two case studies carried out to evaluate the proposed thresholds. Section 7 discusses the limitations of this work. Concluding remarks are presented in Section 8.

2. Related work

A large number of software metrics have been proposed (Abreu and Carapuça, 1994; Chidamber and Kemerer, 1994; Xenos et al., 2000; Kitchenham, 2009). Despite the effort in defining and evaluating software metrics, this research area is challenging. Proper

* Corresponding author. Tel.: +55 31 3409 5860; fax: +55 31 3409 5858.
E-mail address: kecia@dcc.ufmg.br (K.A.M. Ferreira).

interpretation of metric values is essential to characterize, evaluate and improve the design of software systems. However, the typical values of most software metrics are not known yet (Tempero, 2008). Without knowing metric thresholds, software community will not be able to apply software metrics in practice (Lanza and Marinescu, 2006). This section discusses works concerned with characterization of software systems by means of software metrics, and with identification of software metric thresholds.

There has been wide interest in investigating the way modules within a software system connect to each other. A conclusion drawn by those works is that software seems to be governed by power laws (Baxter et al., 2006; Louridas et al., 2008; Potantin et al., 2005; Puppin and Silvestri, 2006; Wheeldon and Counsell, 2003; Ferreira et al., 2009). A power law is a probability distribution function in which the probability that a random variable X takes a value x is proportional to a negative power of x, i.e., $P(X=x) \propto cx^{-k}$. A power law distribution is a heavy-tailed distribution. A characteristic of this type of distribution is that the frequency of high values for the random variable is very low, and the frequency of low values is high. In such distribution, the mean value is not representative, and so, there is no value that can be considered as typical to the random variable (Newman, 2003). Much research has identified power laws in graphs that represent relationships between classes and objects in an object-oriented system. For instance, Potantin et al. (2005) analyzed 60 graphs of 35 software systems and concluded that relationships between objects, in execution time, constitute a scale free graph. A scale free graph is different from a graph with edges distributed randomly. In a random graph, the mean value of node degrees is representative, while in a scale free graph this property is not true because its node degrees distribution follows a power law. Wheeldon and Counsell (2003) identified power laws in the graph of Java program classes. The data analyzed in their study are from three well-known software systems: JDK (*Java Development Kit*), Apache Ant and Tomcat, in a total of 6870 classes. They also identified power law in the following class metrics: number of fields, number of methods and constructors. Louridas et al. (2008) analyzed data of programs developed in C, Perl, Java and Ruby. A set of 11 software systems was analyzed in their work, among them J2SE SDK, Eclipse, OpenOffice and Ruby. The study concluded that, regardless of the programming paradigm, in and out-degree are governed by power law.

Baxter et al. (2006) investigated the structure of a large number of Java programs. The data set used in their study is from 56 open-source software systems, with varying size and from different application domain. They concluded that some of the analyzed metrics follow a power law and others do not. Their study suggests that in-degree and number of subclasses are a power law distribution, but out-degree, number of fields and number of public fields are not. This conclusion diverges from findings of the study of Louridas et al. (2008), which found that out-degree of classes follows a power law. Findings of Baxter et al. (2006) and Louridas et al. (2008) bring information that can allow understanding the shape of open-source programs. However, they did not explore their results in order to identify typical or reference values for the analyzed metrics. Furthermore, those works did not analyze metrics of important quality factors, such as module cohesion.

Lanza and Marinescu (2006) state that there are two sources for thresholds values: statistical information and the widely accepted knowledge. Statistics-based thresholds are derived from statistical analysis of data from a population or a sample of a population. Using statistical analysis, Lanza and Marinescu (2006) suggested thresholds of three software metrics: number of methods per class, lines of code per method, and cyclomatic number per lines of code. They collected these metrics from 37 C++ systems and 45 Java systems. The diversity of size, application domain, and type (open-source and commercial software) was the basis of the sample selection. Thresholds proposed by them are given by: an interval of typical values of the metric; a lower bound and an upper bound of this interval; and a value which can be considered an outlier. Considering a normal distribution for the collected data, they applied average and standard deviation in order to define the thresholds: for each metric, the average is the typical value and the standard deviation is used to define the two bounds of the typical values interval. They consider a value as an outlier if it is 50% higher than the highest value of the interval. The method applied in their work is useful only if the values follow a normal distribution. However, as pointed by the other studies described in this section, a large number of software metrics follow power law. Then, interpreting these metrics in terms of average values can be extremely misleading.

The present work aims to determine thresholds of six software metrics, which have not been studied in this way in previous works: LCOM (lack of cohesion in methods), DIT (depth in inheritance tree) (Chidamber and Kemerer, 1994), COF (coupling factor) (Abreu and Carapuça, 1994), afferent couplings, number of public methods and number of public fields. These metrics are described in Section 3.1. Our analysis shows that values of the metrics evaluated in this work are not fitted by a normal distribution. We identify a probability distribution that well fits values of these metrics. Furthermore, we present a method for derive thresholds of these metrics based on statistical analysis of the data. Using this method, we propose thresholds for the six software metrics evaluated in this study.

3. Methods

In this section, we describe the software metrics analyzed in this study and present the method which we employed to derive metric thresholds.

3.1. Software metrics

There is a large number of object-oriented software metrics. In the scope of this work, we studied six of those metrics, which were selected because they are related to influential factors to software quality, such as coupling, cohesion, encapsulation, information hiding and depth of inheritance tree. The metrics are described as follows:

COF (Coupling Factor) (Abreu and Carapuça, 1994): this metric is calculated for the system level. It is based on the concept of *client–supplier* relationship between classes. Considering this concept, a class A is client of a server class B if A references to at least one feature of B. If A is client of B, then there is a connection from A to B. In a software system with n classes, the maximum possible number of connections is $n^2 - n$. COF is given by $c/(n^2 - n)$, where c is the number of actual connections between the classes in the program. This metric is an indicator of the connectivity level of the system. The higher COF value, the higher the connectivity of the system and the lower its maintainability is thought to be (Abreu and Carapuça, 1994). As asserted by Meyer (1997), in a software architecture, “every module should communicate with as few others as possible”, otherwise changes and errors may widely propagate in the system.

Number of public fields: this metric measures the total number of public fields defined in a class. One of the main concepts of object-oriented paradigm is information hiding, which states that a module should reveal little as possible about its inner workings. In order to achieve this principle, it is desirable to avoid making data public. A public field of an object can be directly accessed and changed by other objects. Hence, the use of public field can lead to strong coupling among classes within a software system, reducing the modularity of the program (Fowler, 1999; Meyer, 1997).

Table 1
Software systems, their application domain, type, size, number of connections between classes, and COF metric.

Domain	Software	Type	#Classes	#Connections	COF
Clustering	Essence	Framework	182	543	0.016
	Gridsim	Tool	214	774	0.017
	JavaGroups	Tool	1061	3807	0.003
	Prevayler	Library	90	137	0.017
	Super (Acelet-Scheduler)	Tool	246	1085	0.018
Database	DBUnit	Framework	289	911	0.011
	ERMaster	Tool	569	2187	0.007
	Hibernate	Framework	1359	5199	0.003
Desktop	Facilitator	Tool	2234	6565	0.001
	Java Gui Builder	Tool	60	126	0.036
	Java X11 Library	Library	318	1146	0.011
	J-Pilot	Tool	142	367	0.018
	Scope	Framework	214	535	0.012
Development	Code Generation Library	Library	226	662	0.013
	DrJava	Tool	2766	9684	0.001
	Find bugs	Tool	1019	3108	0.003
	Jasper reports	Library	1233	5610	0.004
	Junit	Framework	154	353	0.015
	Spring	Framework	2116	7069	0.002
	BCEL	Library	373	2111	0.015

Number of public methods: this metric is the total number of public methods defined in a class. The number of public methods is an indicator of the external size of a class. It is a relevant measure from the viewpoint of the clients of a class because it represents the number of functionalities the class provides to them. As stated by Fowler (1999), a large number of methods in a class is a sign that the class may have too many responsibilities, what makes this class hard to understand and maintain. This code smell, also known as God Class or Blob, refers to classes which perform a large amount of responsibilities in such way they centralize the intelligence of the system (Lanza and Marinescu, 2006). Although the number of public methods can be only a portion of the total number of methods of a class, it can be taken as an indicator of how large a class is. In the present work, we computed this metric by counting the public methods, excluding the protected ones.

LCOM (Lack of Cohesion in Methods) (Chidamber and Kemerer, 1994): this metric measures the cohesion level of a class by considering the concept of similarity of methods of the class. Two methods are similar if they use at least one common field of their class. LCOM is given by the number of pairs of non-similar methods minus the number of pairs of similar methods. When the number of pairs of non-similar methods is less than the number of pairs of similar methods, LCOM is set to zero. According to Chidamber and Kemerer (1994), the higher LCOM value, the lower the class cohesion. However, a zero value does not necessarily mean good cohesion.

There is a large number of cohesion metrics for object-oriented software, and LCOM has been criticised in the literature (Briand et al., 1999; Kitchenham, 2009). In spite of this, we consider LCOM in our study because there is no consensual conclusion about the best way on measuring class cohesion. In addition, some cohesion metrics are based on the same idea of similarity used by LCOM. Here, we make no claim about the validity of LCOM. The scope of this work is to investigate the common values of this metric in practice, providing a threshold for those who use it. A study of Lincke et al. (2008) analyzed ten software metric tools for Java programs; seven of those tools collect LCOM as originally proposed by Chidamber and Kemerer (1994), while only three of them collect a new version of LCOM. Therefore, LCOM seems to be widely used despite the criticism it has been suffered.

DIT (Depth of Inheritance Tree) (Chidamber and Kemerer, 1994): this metric is given by the maximum distance of a class from the root class in the inheritance tree of the system. Inheritance is a powerful technique of software reuse. Nevertheless, Gamma et al. (1994) claim its immoderate use can make software design more complex. They, then, define a principle: *favor object composition over class inheritance*. In the same vein, Sommerville (2000) argues that inheritance introduces difficulties in the comprehension of objects behavior. An empirical study of Daly et al. (1996) shows that deep inheritance trees make software maintenance more difficult. DIT indicates how deep a class is in the inheritance tree. It is considered as an indicator of the design complexity of the system (Chidamber and Kemerer, 1994). The higher the DIT of a class, the higher the number of classes involved in its analysis and the more complex its comprehension.

Afferent couplings: if a class *A* references another class *B*, then there is an afferent coupling in *B* from *A* and there is a corresponding efferent coupling in *A*. A given class *A* references another class *B* if (i) *A* calls a method from *B*, (ii) *A* access a field of *B*, or (iii)*A* is a subclass of *B*. This metric is the number of incoming couplings of a class (Ferreira et al., 2008). It is an indicator of the number of classes which depend upon a given class. Classes with a high number of afferent couplings play an important role in the system, because errors or modifications on them may widely impact other classes. For Java software, we computed this metric also for interfaces. In this case, the number of afferent couplings of an interface is the number of classes which implement the interface or have any reference to the interface. Martin (1994) have defined the metric afferent coupling (Ca) to the package level. By his definition, the number of afferent couplings of a package is the number of classes outside this package which depend upon classes within this package. The metric used in the present work, however, is computed to the class level. In the literature, the term *in-degree* is also commonly used to denote the number of incoming couplings of a given class.

3.2. Data set

The data used in this study are from 40 open-source Java software systems, downloaded from SourceForge (www.sourceforge.net), varying size from 18 to 3500 classes, in their latest version up to June 2008. Program codes are from 11 application domains and three types: tool, library and framework. More than 26,000 classes were analyzed. The software systems and their application domains, types and sizes are described in Tables 1 and 2. The two final columns of Tables 1 and 2 show,

Table 2
Software systems, their application domain, type, size, number of connections between classes, and COF metric.

Domain	Software	Type	#Classes	#Connections	COF
Enterprise	Liferay	Framework	14	14	0.077
	Talend	Tool	2779	3567	0.000822
	uEngine BPM	Framework	708	1774	0.004
	YAWL	Tool	382	1186	0.008
Financial	JMoney	Tool	193	424	0.019
Games	JSpaceConquest	Tool	150	424	0.019
	KoLmafia	Tool	810	5106	0.008
	Robocode	Tool	213	738	0.016
Hardware	Jcapi	Library	21	61	0.145
	LibUSBJava	Library	35	90	0.076
	ServoMaster	Library	55	117	0.039
Multimedia	CDK	Library	3586	14711	0.001
	JPedal	Tool	539	1533	0.005
	Pamguard	Tool	1503	5267	0.002
Networking	BlueCove	Library	142	461	0.023
	DHCP4Java	Library	18	29	0.095
	jSLP	Library	42	156	0.091
	WiKID Strong Authentication	Library	50	27	0.011
Security	JSch	Library	110	226	0.022
	OODVS	Library	171	325	0.011

respectively, the number of connections between classes within the software system, and the value of COF.

A tool, called *Connecta* (Ferreira et al., 2008), was used to collect the metrics. *Connecta* collects object-oriented software metrics from bytecodes of Java programs. For this reason, a criterion to choose the software systems analyzed in this study was the availability of their bytecode.

We carried out four analyses of the data set: (1) we analyzed the data as whole and also by (2) application domain, (3) by size, and (4) by type of software systems. The purpose of these analyses was finding out whether there would be a single probability distribution which could fit values of a metric, regardless of the application domain, type or size of the software systems. Based on the statistical properties of the identified probability distributions, we derived metric thresholds.

3.3. Data fitting

A tool, called EasyFit (Mathwave, 2010), was used to fit the data to various probability distributions, such as Bernoulli, Binomial, Uniform, Geometric, Hypergeometric, Logarithmic, Binomial, Poisson, Normal, *t*-Student, Chi-square, Exponential, Lognormal, Pareto and Weibull. A probability distribution has two main functions: the *probability density function* (*pdf*), $f(x)$, which expresses the probability the random variable takes a value x, and the *cumulative distribution function* (*cdf*), $F(x)$, which expresses the probability the random variable takes a value less than or equal to x. In the experiment of this study, the Poisson and the Weibull distributions showed themselves to be well fitted to the data.

The Poisson distribution has *pdf*, $f_p(x)$, and *cdf*, $F_p(x)$, defined by Eqs. (1) and (2), respectively. The parameter λ of the distribution represents the mean value of the random variable.

$$f_p(x) = P(X = x) = \frac{e^{-\lambda}.\lambda^x}{x!} \tag{1}$$

$$F_p(x) = P(X \le x_0) = \sum_{x=0}^{x=x_0} \frac{e^{-\lambda}.\lambda^x}{x!} \tag{2}$$

The Weibull distribution has *pdf*, $f_w(x)$, and *cdf*, $F_w(x)$, with parameters α and β, defined by Eqs. (3) and (4), respectively. The parameter β is called *scale parameter*. Increasing the value of β has the effect of decreasing the height of the curve and stretching it. The parameter α is called *shape parameter*. If the shape parameter is less than 1, Weibull is a heavy-tailed distribution and it can be applied in cases in which the random variable presents left asymmetry, i.e., when there is a small number of occurrences with high values and a far larger number of occurrences with low values. In this kind of distribution, the mean value is not representative.

$$f_w(x) = P(X = x) = \frac{\alpha}{\beta}\left(\frac{x}{\beta}\right)^{\alpha-1} e^{-(x/\beta)^{\alpha}}, \qquad \alpha > 0, \beta > 0 \tag{3}$$

$$F_w(x) = P(X \le x) = 1 - e^{-(x/\beta)^{\alpha}}, \qquad \alpha > 0, \beta > 0 \tag{4}$$

3.4. Data analysis

For each metric, the data were collected and two graphics were generated: a scatter plot, in order to exhibit the frequency of the metric values, and the same data in doubly logarithmic scale (log–log scale), in order to observe whether the distribution shows itself to be a power law. When plotted in a log–log scale, a power law distribution is right-skewed and has an approximately straight-line form. If values of a metric follow a power law, it means that frequency of high values for the metric is very low, while frequency of low values is too high.

The data were fitted to probability distributions by the tool, which indicated the probability distributions best fitted to the data. For each metric, considering the results given by the tool and the visual analysis of the fitting, we identified the probability distribution which best fit to the data. If probability distribution had a representative mean value, like the Poisson distribution, this value was taken as typical for this metric. Otherwise, we worked with three ranges for the metric values: *good*, *regular* and *bad*. The *good* range corresponds to values with high frequency. It is the most common values of the metric in practice. These values do not express the best practices in Software Engineering necessarily. Nevertheless, they expose a pattern of most software systems. It is desirable that a software system has at least the same quality level of the others. Preferably, a software development should strive for even better quality. The *bad* range corresponds to values with quite low frequency, and the *regular* range is an intermediate one, that corresponds to values that are not too frequent neither have very low frequency.

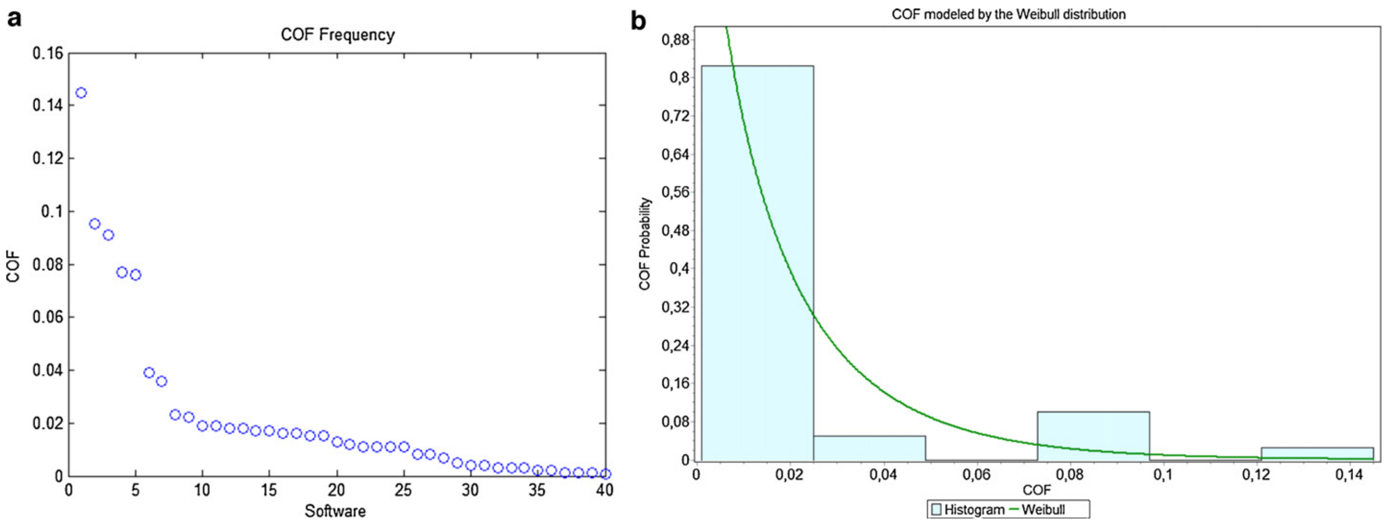

Fig. 1. COF – (a) frequency and (b) fitting to the Weibull distribution.

4. Results

In this section, we describe the findings of our study. First, results of the entire data set are described. Then, we discuss results of the analysis performed on application domains, types and sizes of the software systems.

4.1. Data fitting of metrics in the entire data set

4.1.1. COF

The COF scatter plot in Fig. 1a shows that values less than 0.02 are far more frequent than higher values. COF can be modeled by the Weibull distribution, with parameters $\alpha = 0.91927$ and $\beta = 0.01762$. Fig. 1b shows values of COF modeled with the Weibull distribution. More than 80% of the programs have COF less than 0.02. The probability that COF will fall into the interval 0.02–0.14 is quite low, and the probability that COF takes values higher than 0.14 tends to zero. This result points out that, in most cases, open-source software system is low connected, what may contribute to its maintainability.

4.1.2. Afferent couplings

The scatter plot for afferent couplings, shown in Fig. 2a, suggests a heavy-tailed distribution. Fig. 2b shows the same data plotted in logarithmic scale (log–log scale). In this plot, distribution shows itself to be linear that is the characteristic signature of a power law. There is a small number of classes with high number of afferent couplings and a far higher number of classes with few afferent couplings. As shown by Fig. 2c, values of this metric can be modeled by the Weibull distribution, with parameters $\alpha = 0.78986$ and $\beta = 3.2228$. Afferent couplings distribution is detailed in Fig. 2d. Almost 50% of classes have at most one afferent coupling. The probability that a class has 1–20 afferent couplings is low, and the probability that a class has more than 20 afferent coupling tends to zero. This observation points out that most classes affect directly only one class at most. This can contribute to software maintainability since a modification or an error in a class will impact in a low number of classes.

4.1.3. LCOM

LCOM also is fitted by a heavy-tailed distribution. Fig. 3a shows a scatter plot of the data set, and Fig. 3b shows the same data plotted in log–log scale. This graphic indicates that LCOM values follow a power law. Values of LCOM can be modeled by the Weibull distribution, as shown in Fig. 3c, with parameters $\alpha = 0.23802$ and $\beta = 1.465$. Fig. 3d details LCOM distribution. Approximately 50% of classes have LCOM equals to zero. There are classes with LCOM between 0 and 20 in a low frequency, less than 12%. The probability that a class has LCOM greater than 20 tends to zero.

4.1.4. DIT

The scatter plot in Fig. 4a shows the distribution of values of DIT, and Fig. 4b shows the same data in a log–log scale. These plots do not suggest power law characteristics. In fact, DIT values can be fitted to the Poisson distribution, as shown in Fig. 4c, with parameters $\lambda = 1.6818$. In the Poisson distribution, λ is the mean value of the random variable. By this finding, in open-source software systems, the largest distance from a class to the root in the inheritance tree is 2, in general. Shallow inheritance tree contributes to software quality by decreasing software complexity, as asserted by Gamma et al. (1994), Daly et al. (1996) and Sommerville (2000).

4.1.5. Public fields

The scatter plot of number of public fields, shown in Fig. 5a, reveals that classes with a large number of public fields are few. In most cases, the number of public fields in a class is near to zero. Fig. 5b shows the data plotted in a log–log scale, which indicates that number of public fields in classes also follows a power law. This metric can be modeled by the Weibull distribution with parameters $\alpha = 0.71008$ and $\beta = 4.4001$, what is shown in Fig. 5c. Fig. 5d details the same distribution. Most of 75% of the classes have no public field, and the probability that a class has more than 10 public fields tends to zero. This observation reveals that most of software systems apply the information hiding principle appropriately.

4.1.6. Public methods

The frequency of number of public methods is shown in Fig. 6a and, in a log–log scale, in Fig. 6b. These plots show that number of public methods follows power law. This metric can be modeled by the Weibull distribution, with parameters $\alpha = 0.85938$ and $\beta = 5.6558$, as shown in Fig. 6c. The frequency of number of public methods is detailed in Fig. 6d. There is a low portion of classes with a large number of public methods, and most classes have few public methods. Most classes have 0–10 public methods. Classes with 10–40 public methods are rare, and the probability that a class has more than 40 public methods is quite low. By these findings, it could be concluded that, in most of cases, classes provide few services.

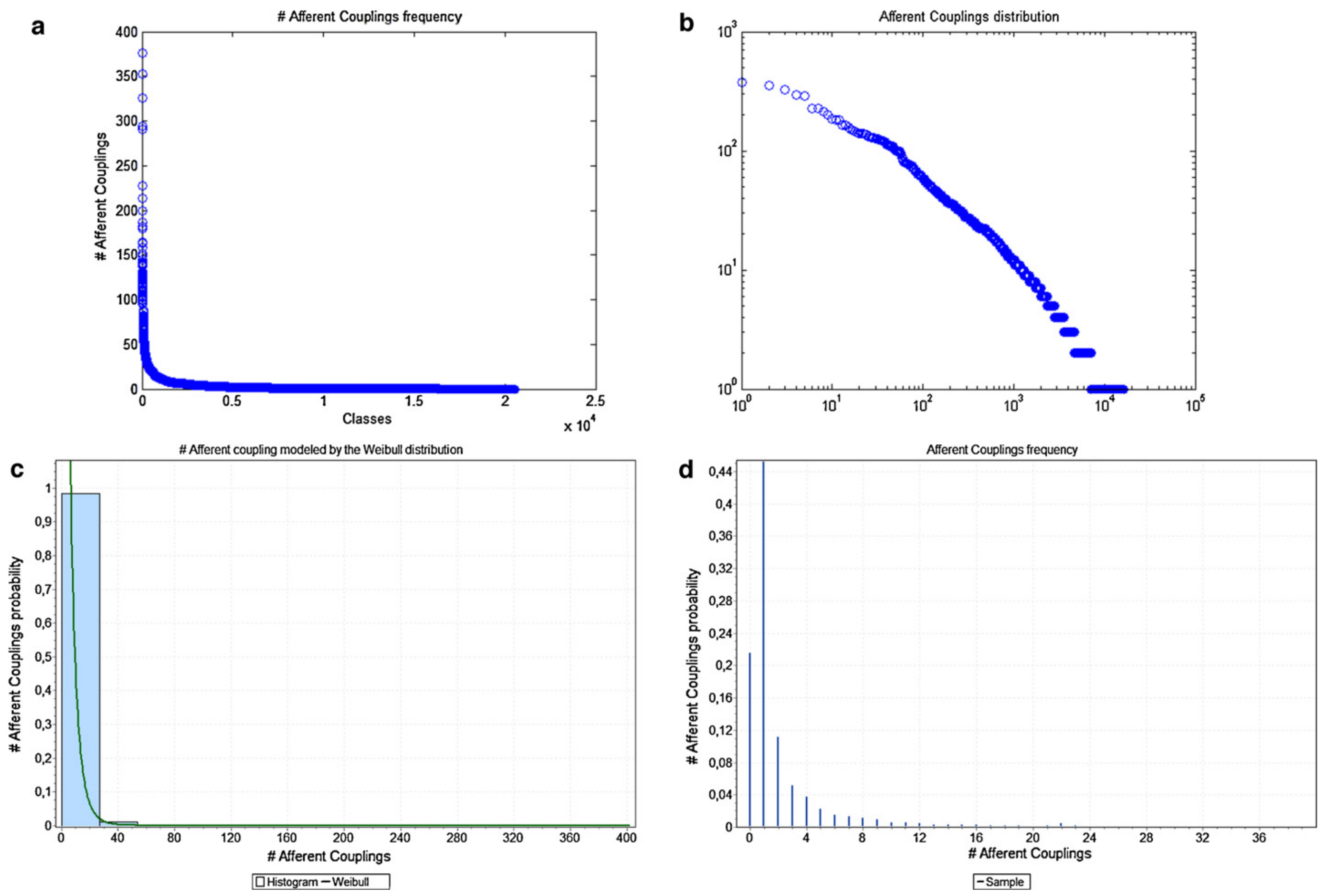

Fig. 2. Afferent couplings – (a) frequency, (b) frequency in log–log scale, (c) fitting to the Weibull distribution and (d) frequency detailed.

5. Software metric thresholds

A large number of object-oriented, open-source programs were evaluated by means of six software metrics in this study. Findings of our analysis lead to identify thresholds of those metrics. We propose to use as thresholds the values of software metrics found in practice. From the achieved results, we identified three ranges of reference values for the metrics: *good*, which refers to the most common values of the metric; *regular*, which is an intermediate range of values with low frequency, but not irrelevant; and *bad*, that refers to values with quite rare occurrences. For instance, LCOM, whose frequency is shown in Fig. 3d, is 0 for more than 44% of the classes, values between 1 and 20 occur in a very low frequency, and values greater than 20 are rare. Therefore, we derived the LCOM threshold: 0 (good cohesion), 1–20 (regular cohesion) and greater than 20 (bad cohesion).

These thresholds do not necessarily denote the best practices in Software Engineering. Nevertheless, they represent the quality level which developers usually apply in software, providing a reference in evaluation of software by means of metrics. For instance, if COF of a software system is 0.5 and the *good* value of this metric is 0.02, it means that the classes within the software system are much more coupled than in most of systems. Hence, if the developers want their systems as good as others in terms of coupling, they should keep the COF of their systems below 0.02. Thresholds are highly important in interpreting values of a metric. Knowing reference values of software metrics might strongly contribute to make them useful in practice. However, metrics can support decision-making, but cannot substitute the judgment of the specialist.

The same analysis was performed for the other metrics. The reference values suggested for COF, LCOM, DIT, afferent couplings, number of public methods and number of public fields are summarized in Table 3. We also carried out similar analyses by application domains, type and size of the software systems. The results of these analyses are described as follows. These analyses, however, were not performed to COF because this metric is given to the system level, and each resultant subset of those analyses has just a few software systems. Having a few programs in these resultant sub-

Table 3
General thresholds for OO software metrics.

Factor	Level	Metric	Reference values
Connectivity	System	COF	Good: up to 0.02; regular: 0.02–0.14; bad: greater than 0.14
	Class	# Afferent couplings	Good: up to 1; regular: 2–20; bad: greater than 20
Information hiding	Class	# Public fields	Good: 0; regular: 1–10; bad: greater than 10
Interface size	Class	# Public methods	Good: 0–10; regular: 11–40; bad: greater than 40
Inheritance	Class	DIT	Typical value: 2
Cohesion	Class	LCOM	Good: 0; regular: 1–20; bad: greater than 20

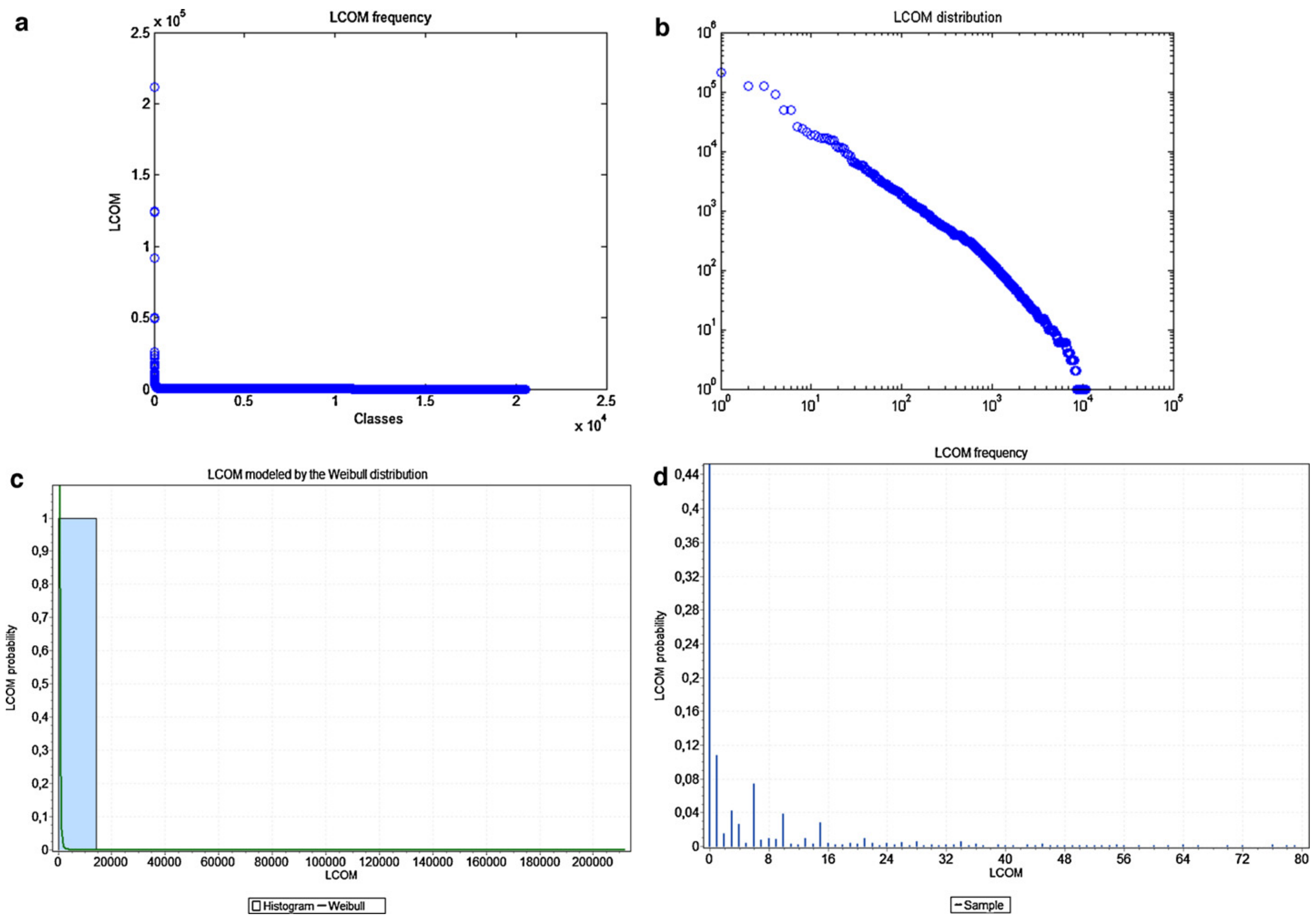

Fig. 3. LCOM (a) frequency, (b) frequency in log–log scale, (c) fitting to the Weibull distribution and (d) frequency detailed.

sets restricts any conclusion about the behavior of COF regarding size, type, and application domain in this study.

5.1. Thresholds for OO software metrics by application domains

The software metrics studied in this work were analyzed for each application domain of the software systems listed in Table 1. We found that values of a software metric, in a specific application domain, can be modeled by the same probability distribution which fitted the metric in the entire data set. Using the same analysis performed to the entire data set, we derived the reference values by application domains. The results are reported in Table 4. There is a slight difference between the results of the application domains and those of the entire data set. Nevertheless, this difference does not mean a disagreement. This observation is true even for the *Financial* application domain, which has only one software system analyzed in this work. Thus, we believe that the general result, listed in Table 3, can be used to software systems in general.

5.2. Thresholds for OO software metrics by software types

The software systems analyzed in this work are of three types: tool, framework and library. We carried out the analysis of values of the software metrics studied in this work for those three categories. The results reveal that the distribution probability which modeled values of a metric in the whole data set is also applicable to different

Table 4
Thresholds for OO software metrics by application domains.

Application domain	Afferent coupling (good/regular/bad)	# Public fields (good/regular/bad)	# Public methods (good/regular/bad)	DIT (typical value)	LCOM (good/regular/bad)
Clustering	0–1/1–20/>20	0/1–7/>7	0–20/20–45/>45	2	0/1–20/>20
Database	0–1/2–20/>20	0/1–8/ >8	0–20/21–50/>50	2	0/1–20/ >20
Desktop	0–1/2–20/>20	0/1–8/ >8	0–20/21–55/ >55	2	0/1–15/>15
Development	0–1/2–25/>25	0/1–8/>8	0–20/21–35/>35	2	0/1–25/>25
Enterprise	0–1/2–22/>22	0/1–11/>11	0–15/16–35/>35	1	0/1–35/>35
Financial	0–1/2–16/>16	0/1–4/>4	0–13/14–32/>32	2	0/1–25/>25
Games	0–1/2–22/>22	0/1–9/>9	0–20/21–32/>32	1	0/1–35/>35
Hardware	0–1/2–11/>11	0–1/2–6/>6	0–20/21–36 />36	2	0/1–80/>80
Multimedia	0–1/2–20/>20	0/1–9/>9	0–35/36–60/>60	2	0/1–60/>60
Networking	0–1/2–20/ >20	0/1–5/>5	0–15/16–40/>40	2	0/1–40/>40
Security	0–1/2–16/>16	0/1–8/ >8	0–25/26-50/>50	1	0/1–45/>45

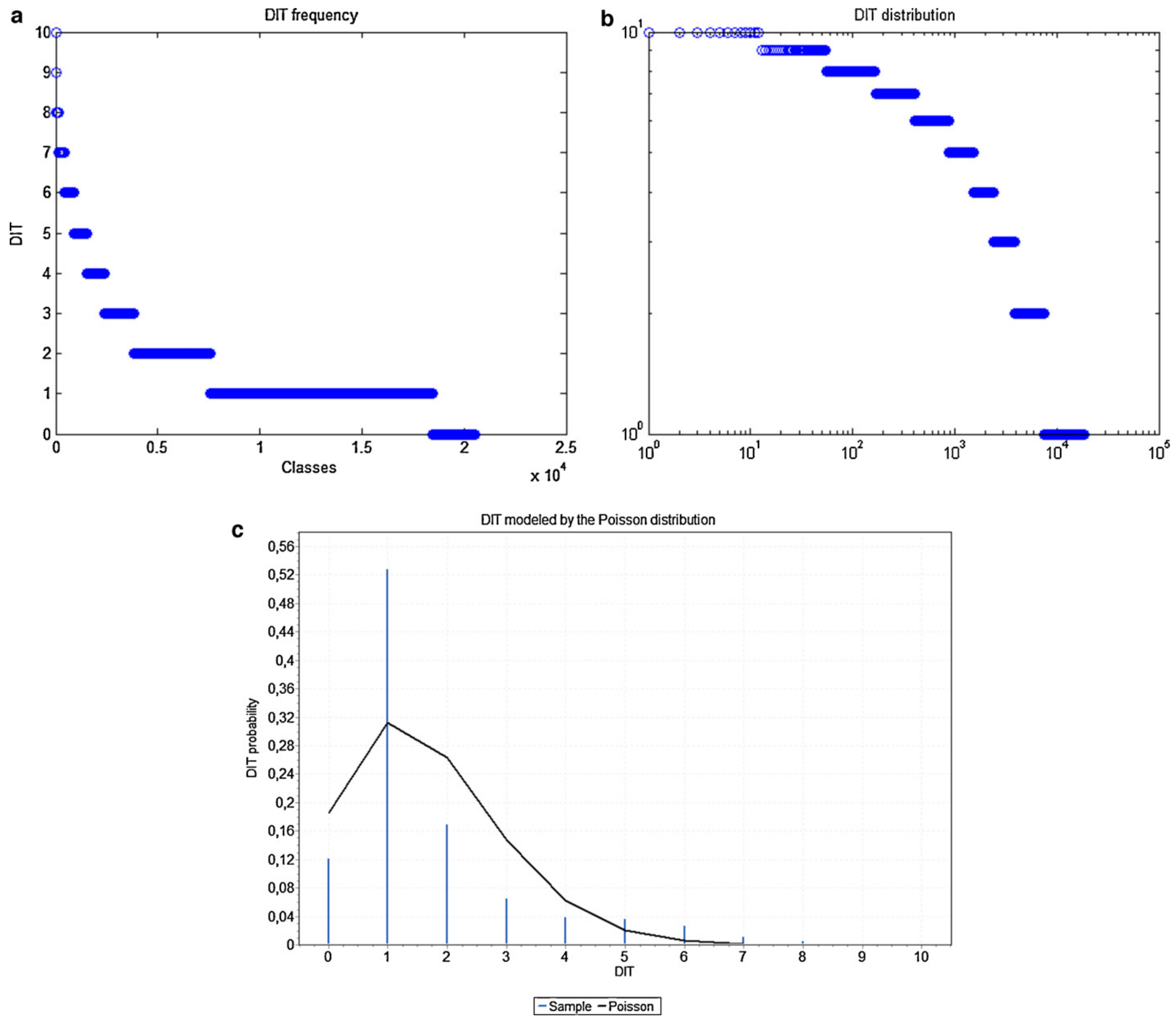

Fig. 4. DIT – (a) frequency, (b) frequency in log–log scale, (c) fitting to Poisson's distribution.

categories of software. Table 5 summarizes the thresholds derived in this analysis. We discuss such results as follows.

- *Public fields*: in the three cases, more than 80% of classes have no public fields, frequency of classes with 1–8 public fields is very low, and frequency of classes having more than 8 public fields is near to zero.
- *Public methods*: there is a slight difference among the distributions of values of this metric in frameworks, libraries and tools. The distribution of values in tools is a little more left concentrated, indicating that tools have less number of public methods than frameworks and libraries.
- *LCOM*: values of this metric have a very similar distribution in the three cases. Thresholds of this metric in the three categories of software system are basically the same found in the analysis of the whole data set.
- DIT: this metric can be modeled by the Poisson distribution in the three cases. There is a little difference in mean values: 1.68 in frameworks, 1.74 in tools and 1.96 in libraries.
- *Afferent couplings*: results found in the analysis of data from frameworks, libraries and tools are basically the same. Thresholds

Table 5
Thresholds for OO software metrics by software types.

Type	Afferent coupling (good/regular/bad)	# Public fields (good/regular/bad)	# Public methods (good/regular/bad)	DIT (typical value)	LCOM (good/regular/bad)
Tool	0–1/2–20/>20	0/1–8/>8	0–20/21–50/>50	2	0/1–20/>20
Framework	0–1/2–20/>20	0/1–10/>10	0–25/26–50/>50	2	0/1–20/>20
Library	0–1/2–25/>25	0/1–8/>8	0–25/26–40/>40	2	0/1–25/>25

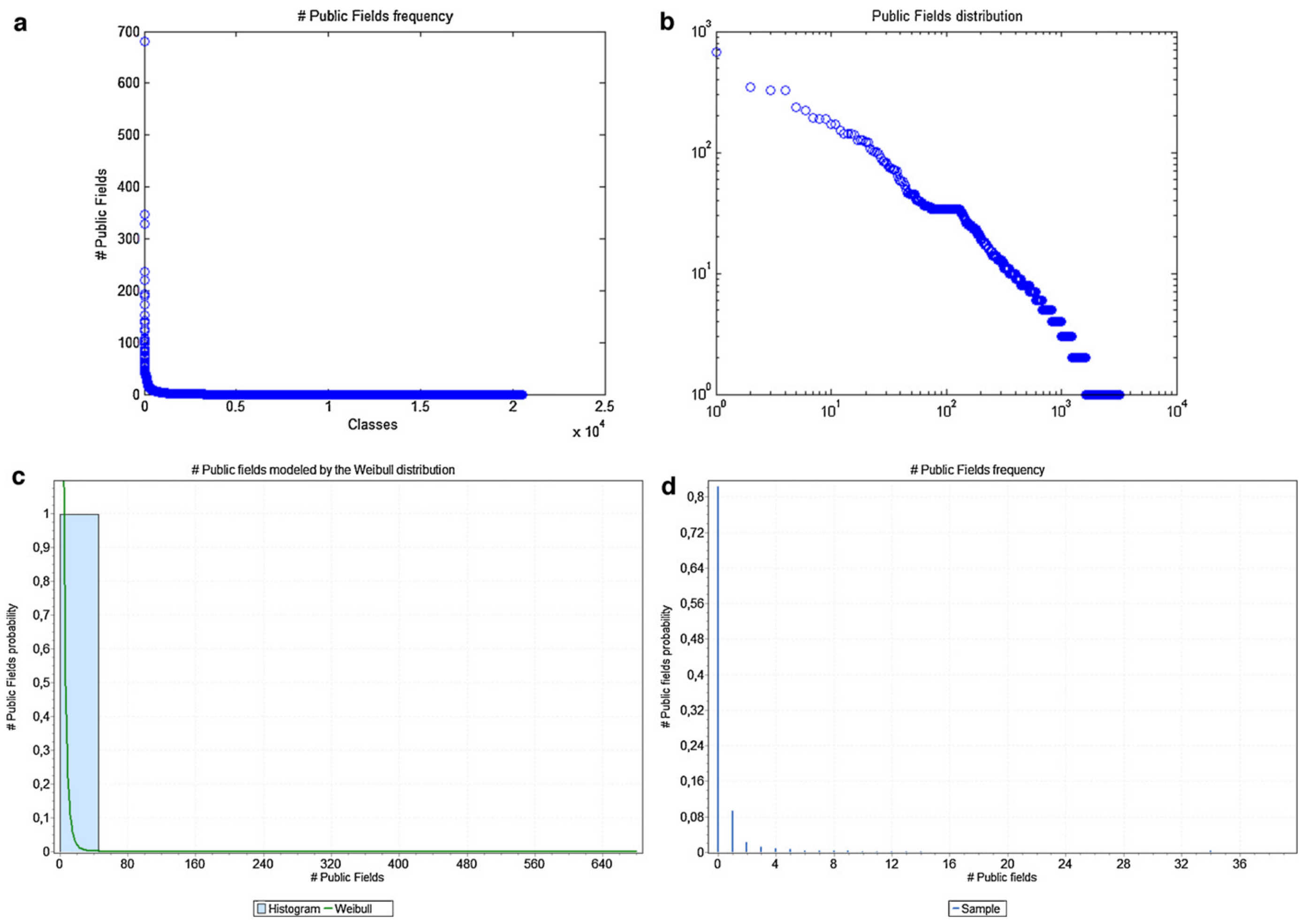

Fig. 5. Public fields – (a) frequency, (b) frequency in log–log scale, (c) fitting to the Weibull distribution and (d) frequency detailed.

identified for this metric in the three cases are compatible with those found in the entire data set.

5.3. Thresholds for OO software metrics by software size

We grouped the software systems analyzed in this work in three sets according their size: up to 100 classes, from 101 to 1000 classes and more than 1000 classes. The software metrics studied in this work were analyzed for each set. We found that a single probability distribution can model values of a software metric, regardless of the software system size. Such probability distribution is the same which modeled values of the metric in the whole data set.

Using the same analysis performed to the entire data set, we derived the thresholds for the software metrics, by size of software system. The results are reported in Table 6. There is a slight difference among the results found for the sets. An interesting difference is about the number of public fields: the higher the software system size, the lower the number of public fields. It leads to conclude that public fields are strongly avoided in large software system. Hence, software developers seem to be more concerned about information hiding in large systems.

The results of this analysis do not disagree with those of the entire data set. Therefore, we believe that the general result, listed in Table 3, can be used to software systems in general, regardless of the software system size.

6. Evaluation and discussion

Thresholds can aid specialists to apply metrics in their tasks. Medicine is an example of field in which the work of the specialist is strongly supported by metrics and their thresholds. In that field, many clinical laboratory tests are interpreted by using reference values. Without knowing those values, diagnosing and treating diseases would be extremely difficult. In Software Engineering, thresholds of metrics can aid specialists in their tasks of evaluating products and software processes, for instance. The use of a metric without knowing its reference values is very restricted because it may be difficult to identify whether a situation

Table 6
Thresholds for OO software metrics by software size.

Size (# classes)	Afferent coupling (good/regular/bad)	# Public fields (good/regular/bad)	# Public methods (good/regular/bad)	DIT (typical value)	LCOM (good/regular/bad)
≤100	0–1/2–20/>20	0/1–10/>10	0–20/21–30/>30	2	0/1–25/>25
101–1000	0–1/2–20/>20	0/1–8/>8	0–25/6–50/>50	2	0/1–20 />20
>1000	0–1/2–15/>15	0/1–5/>5	0–30/30–60/>60	2	0/1–20/ >20

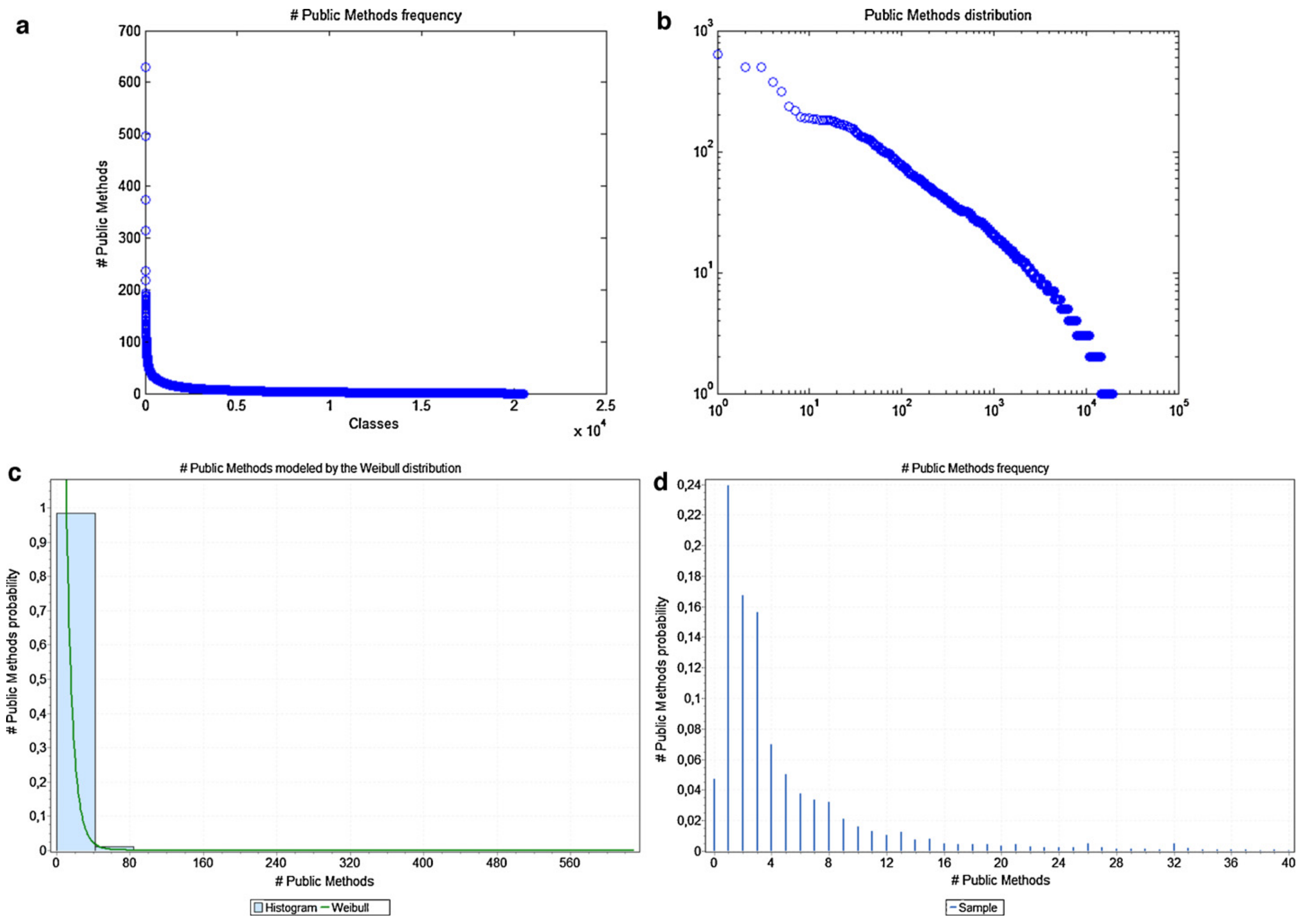

Fig. 6. Public methods – (a) frequency, (b) frequency in log–log scale, (c) fitting to the Weibull distribution and (d) frequency detailed.

represents a risk or a problem to the project, having as basis only a number without its proper interpretation.

In this section, we evaluate and discuss the reference values identified in this work. We carried out two experiments to evaluate the usefulness of the proposed thresholds. The reference values considered in these experiments are the general ones, which are listed in Table 3. In the first experiment, we evaluate whether the thresholds support the identification of problematic classes. In the second study, we evaluate whether the thresholds support the identification of well-designed classes.

Applying a reference value can lead two results: or the reference value evaluates the property accurately, or it fails in the evaluation. The evaluation is right if the range in which the metric value falls corresponds to the actual assessment of the property. In the experiments carried out in this work, we consider two possibilities of failure by using a threshold: *false positive*, when the range of the metric evaluates the class as *bad*, but the manual inspection of the class does not identify problems in its structure; and *false negative*, when the range of the metric does not evaluate the class as *bad*, but the manual inspection of the class identifies problems in its structure.

The first experiment was performed on the entire set of classes from which the reference values proposed in this work were derived. In this experiment, we considered the classes whose measures correspond to the *bad* range of each evaluated metric. Such classes were manually inspected in order to evaluate the effectiveness of the reference values in helping to identify classes with structural problems. It is possible to identify occurrences of *false positive* in this way. However, the identification of *false negatives* is impractical because it would demand the manual inspection of a large number of classes.

In the second experiment, we analyzed JHotDraw.[1] We chose to analyze this program because its design has been considered as having high quality, and many other studies have used this program as a case study (Seng et al., 2006; Czibula and Czibula, 2008; Jancke, 2010; Kessentini et al., 2010). JHotDraw was initially developed by Erich Gamma and Thomas Eggenschwiler in order to be an exercise of design patterns application, but has already a good design (Riehle, 2000). Based on the assumption that JHotDraw possesses a good design, the aim of the experiment is to explore how well the proposed thresholds can be used to identify good designs. It is expected that most of the measures of JHotDraw will be classified in the *good* or *regular* ranges. Having this result will suggest that the thresholds can evaluate a software design as good when it really is, i.e., the negative result (a measure in a good range, indicating the absence of a design problem) is trustful. The results of the second case suggest that when a measure falls in the *good* range it is a sign of absence of design problems, and, hence, the proposed thresholds do not lead to occurrences of false negatives in general.

[1] http://www.jhotdraw.org/.

Table 7
First experiment – classes and their metrics #AF (afferent couplings), LCOM, DIT, #PF (public fields) and #PM (public methods).

Software	Class	#AF	LCOM	DIT	#PF	#PM
JasperReports	`net.sf.jasperreports.engine.xml.JRXmlConstants`	51	957	1	347	44
KolMafia	`net.sourceforge.kolmafia.textui.DataTypes`	27	261	1	70	27
JavaX11Library	`gnu.x11.Display`	63	67	1	54	31
KolMafia	`net.sourceforge.kolmafia.request.UseItemRequest`	29	67	4	41	18
KolMafia	`net.sourceforge.kolmafia.KoLmafia`	163	2826	1	38	75
Hibernate	`org.hibernate.Hibernate`	61	91	1	31	17
JavaX11Library	`gnu.x11.Window`	53	1345	3	29	96
KolMafia	`net.sourceforge.kolmafia.KoLAdventure`	25	147	3	25	20
JavaX11Library	`gnu.x11.extension.glx.VisualConfig`	32	878	1	20	45
KolMafia	`net.sourceforge.kolmafia.request.GenericRequest`	119	872	3	19	38
Jpilot	`org.jpilotexam.ui.util.UIUtilities`	24	21	1	18	4
CodeGeneration	`net.sf.cglib.core.CodeEmitter`	59	2175	1	16	98
KolMafia	`net.sourceforge.kolmafia.persistence. ConcoctionDatabase`	39	328	1	16	20
Bcel	`org.apache.bcel.generic.Type`	55	24	1	16	10
YAWL	`org.yawlfoundation.yawl.engine.interfce. WorkItemRecord`	44	2725	1	15	75
KolMafia	`net.sourceforge.kolmafia.AdventureResult`	144	46	1	15	31
KolMafia	`net.sourceforge.kolmafia.request.UseSkillRequest`	26	285	4	15	19
KolMafia	`net.sourceforge.kolmafia.KoLCharacter`	124	15507	2	13	188
JasperReports	`net.sf.jasperreports.engine.util.JRProperties`	37	223	1	13	26
JasperReports	`net.sf.jasperreports.engine.design.JRDesignChart`	31	4449	3	12	128
KolMafia	`net.sourceforge.kolmafia.request.EquipmentRequest`	36	62	5	12	14
KolMafia	`net.sourceforge.kolmafia.persistence.SkillDatabase`	26	206	3	11	22
KolMafia	`net.sourceforge.kolmafia.swingui.panel.ItemManagePanel`	28	53	7	11	15
Super	`com.acelet.lib.CommonPanel`	36	75	5	11	8

6.1. Experiment 1

In this experiment, we inspected classes whose measures were classified as *bad*, in order to compare this result with a qualitative evaluation.

Initially, we selected each class which has a number of afferent couplings greater than 20. This first filter resulted in 526 classes. In this resultant set, we selected the classes with LCOM greater than 20, obtaining 275 classes. Because the manual inspection of all those classes would be extremely laborious, we applied other three distinct filters to the resultant set: number of public methods greater than 40, which resulted in 59 classes; DIT greater than 2, which resulted in 39 classes; and number of public fields greater than 10, which resulted in 24 classes. We chose to work with this last set due to the small enough number of classes. Hence, the classes analyzed in this experiment have number of afferent couplings, LCOM and number of public fields classified in the *bad* range. The data of those classes are shown in Table 7. In this experiment, we did not evaluate the bad range of COF because this is a metric for the system level and, hence, the evaluation of this metric will demand the manual inspection of each system as whole. Following, we indicate the structural problems that we identified in the classes.

- *Information hiding violation:* some of the classes have public non constant fields. These fields should be encapsulated. The following classes have this structural problem:

```
- gnu.x11.Display
- gnu.x11.Window
- gnu.x11.extension.glx.VisualConfig
- net.sourceforge.kolmafia.request.GenericRequest
- net.sourceforge.kolmafia.swingui.panel.
  ItemManagePanel
- net.sourceforge.kolmafia.KoLCharacter
- com.acelet.lib.CommonPanel
```

- *Utility class:* three of the evaluated classes have static methods with distinct purposes. Possibly, each of these classes could be split, since it seems they have many responsibilities. The following classes have this problem:

```
- net.sf.jasperreports.engine.xml.JRXmlConstants
- net.sourceforge.kolmafia.textui.DataTypes
- org.jpilotexam.ui.util.UIUtilities
```

- *Data class:* some of the classes can be considered as the code smell *Data class*, since they only have data and methods *get* and *set*. The following classes have this code smell.

```
- gnu.x11.extension.glx.VisualConfig
- org.yawlfoundation.yawl.engine.interfce.
  WorkItemRecord
- net.sf.jasperreports.engine.util.JRProperties
- net.sf.jasperreports.engine.design.JRDesignChart
```

- *Low internal cohesion:* most of the classes are candidate for refactoring because they do not seem to have a single role in the system. The following classes have this problem:

```
- gnu.x11.Display
- net.sourceforge.kolmafia.request.UseItemRequest
- net.sourceforge.kolmafia.KoLmafia
- org.hibernate.Hibernate
- net.sourceforge.kolmafia.KoLAdventure
- net.sourceforge.kolmafia.request.GenericRequest
- net.sf.cglib.core.CodeEmitter
- net.sourceforge.kolmafia.persistence.
  ConcoctionDatabase
- org.apache.bcel.generic.Type
- net.sourceforge.kolmafia.AdventureResult
- net.sourceforge.kolmafia.request.UseSkillRequest
- net.sourceforge.kolmafia.persistence.SkillDatabase
- net.sourceforge.kolmafia.request.EquipmentRequest
```

The metrics number of afferent couplings, LCOM, and the number of public fields of all those classes were classified in a *bad* range of the proposed thresholds. The manual inspection of those classes

Table 8
Second experiment – classes and their metrics #AF (afferent couplings), LCOM, DIT, #PF (public fields) and #PM (public methods).

Class	#AF	LCOM	DIT	#PF	#PM
`org.jhotdraw.geom.Geom`	50	276	1	4	42
`org.jhotdraw.draw.AbstractFigure`	49	1339	2	0	45
`org.jhotdraw.draw.DefaultDrawingView`	34	2573	4	1	58
`org.jhotdraw.draw.BezierFigure`	25	184	4	0	51
`org.jhotdraw.draw.action.ButtonFactory`	23	496	1	8	69

massively revealed structural problems on them. In many cases, the evaluated class does not play a single role in the system, what determines its low internal cohesion.

Although LCOM has not been considered as an ideal way to assess class cohesion, the application of its threshold associated with the threshold of number of afferent couplings led to identify classes with poor cohesion in this experiment.

The threshold of number of public fields led to identify classes which are merely data depositories and classes which use public fields in such way they do not comply with the information hiding principle. Classes which are repositories of constants do not necessarily represent violation of good practices of design. However, classes with public non constant fields should be refactored to encapsulate the fields.

There are eight classes analyzed in this experiment which had their number of public methods classified as *bad* by the threshold of this metric. The inspection of those classes revealed that most of them have many responsibilities and are hard to understand. Some of them are *data classes*. Therefore, applying the threshold of this metric can aid to identify classes with such problems.

Five classes from the analyzed set have DIT greater than 3. A common observation in those classes is that they and their superclasses possess structural problems. In some cases, there are signs of improper use of inheritance. Hence, applying the DIT threshold could aid to identify unsuitable use of inheritance in software systems.

The aim of this experiment was to evaluate the practical use of the proposed thresholds. We explored whether classes with measures classified as *bad* by the proposed thresholds have structural problems. For this purpose, we compared the results of the metrics with a qualitative evaluation of the analyzed classes. The classes analyzed in this experiment had their measures classified as *bad* by the proposed thresholds. The manual inspection of those classes revealed that they have structural problems. Hence, in this experiment, the evaluation of classes by applying the proposed thresholds did not lead to *false positive* occurrences. The observations from this experiment suggest that applying the proposed thresholds can help to identify classes which violate design principles. Therefore, when the manual analysis of classes in large systems would be impractical, metrics with their respective thresholds can be effectively used for guidance in detection of design flaws.

6.2. Experiment 2

In this second experiment, we explored whether the proposed thresholds evaluate rightly software systems which have good design. For this purpose, we used a software system whose design has been qualitatively evaluated as good by many other researchers, JHotDraw. On the assumption that this program has really a good design and that the proposed thresholds are valid, the application of these thresholds should evaluate the program as good or regular at least. JHotDraw has 1095 classes and its COF is 0.003. According to the threshold of this metric, this value is classified as *good*. Following, we describe the evaluation of JHotDraw according to the proposed thresholds of the other metrics.

- *Afferent couplings:* 29 classes have more than 20 afferent couplings, 384 classes have 2–20 afferent couplings, and 682 classes have less than 2 afferent couplings. Hence, 97% of classes from JHotDraw are classified as *good* or *regular* by the thresholds in the aspect of afferent couplings.
- *LCOM:* 215 classes have LCOM greater than 20, 335 classes have LCOM from 1 to 20, and 545 classes have LCOM equals to zero. The application of the threshold identified for LCOM classifies 80% of classes from JHotDraw as *good* or *regular*.
- *DIT:* 344 classes have DIT greater than 2, whereas 751 have DIT up to 2, what corresponds to 70% of the classes. The mean value of DIT in JHotDraw is 2.3, indicating that the use of inheritance in JHotDraw is consistent with the typical value identified for this metric.
- *Number of public fields:* 945 classes have no public fields, 146 classes have 1 to 10 public fields, and only 4 classes have more than 10 public fields. Nearly all classes from JHotDraw are evaluated as *good* or *regular*, by using the threshold of number of public fields.
- *Number of public methods:* 925 classes have up to 10 public methods, 156 classes have 11–40 public methods, and 14 classes have more than 40 public methods. Therefore, the threshold of number of public methods evaluates 99% of classes from JHotDraw as *good* or *regular*.

Considering each metric individually, a very low number of classes were evaluated as *bad* by the proposed thresholds. We also used association of metrics to evaluate the software system in the following way. Only 18 classes have LCOM and number of afferent couplings classified as *bad*. In this resultant set of classes, only one class, named `AttributeKeys`, has the number of public fields classified as *bad*. This class, however, is a repository of constants and it does not necessarily represent a violation of design principles. In that resultant set of classes, six classes have DIT greater than 2, as well six classes have number of public methods classified as *bad*. Data of these classes are shown in Table 8. Considering all metrics, none of the classes is evaluated as *bad*.

The observations of this experiment show that the proposed thresholds evaluate JHotDraw design as good in general. If one considers as correct the evaluation that community has made about JHoDraw, this result indicates that the application of the proposed thresholds does not lead to false negative occurrences, i.e., when a measure is classified in the *good* or *regular* ranges, it is a sign of absence of design problems. Therefore, the observations of this experiment suggest that the proposed thresholds can effectively evaluate a design as good if it really is.

6.3. Discussion

The proposed thresholds were effective in the experiments. The achieved results shows that when a measure is classified as *bad*, it does not mean the class or the system definitely violates a principle design. However, it should be taken as a sign that there might be something wrong. When a measure falls in the *bad* range, it is a symptom which strongly indicates a deeper problem. For DIT, we identified a typical value instead of ranges. Considering that deep inheritance trees might make software maintenance hard, it is desirable to keep DIT low. Hence, classes with DIT higher than the identified typical value should be taken as a symptom that inheritance might be misused in the software system, and, thus, should be viewed as a recommendation to refactor the inheritance tree.

The ranges *good* and *regular* were not evaluated individually because it is difficult to assert if a class has *good* or *regular* quality based on qualitative analysis. JHotDraw, the software system taken to evaluate the *good* and *regular* ranges, is supposed to have a good design, but possibly it could be improved. Hence, we assumed that the target software has quality *regular* at least, and we evaluated the ranges *good* and *regular* in conjunction. For all metrics, most of classes of JHotDraw were ranked as *good*, the second largest group of classes was ranked as *regular*, and only a few classes were ranked as *bad*. This result is consistent with the recognized high internal quality of the target system. Therefore, from the observations of this experiment, the ranges *good* and *bad*, as well the mean value of DIT, showed themselves to be also reliable.

Another insight on this experiment is that the distributions of the metric values of the target program have the same properties which were observed in the applications used to derive the thresholds: the mean value of DIT is compatible with the identified typical value of this metric, and for the other metrics, most of classes presents low metric values, whereas just a few classes have high metric values. This result supports the finding that there is a single distribution which can model values of a metric regardless size, type or application domain of software systems.

The *regular* range can be considered as a sign that the software design should be improved for even better quality, although it does not necessarily represents an anomalous occurrence. However, the frontier of this range with the *bad* range is subtle and is not too accurate. Since the method employed to derive the ranges was based on graphical analysis, one may derive other bounds for the *regular* ranges from the same graphics and employing the same analysis. For instance, one can derive the upper bound of the *regular* range for *afferent couplings* as 15 or 25, instead of 20. Therefore, values of metrics near to the upper bound of the *regular* range should also be interpreted as an indicator of possible design principle violation.

The observations of the study cases carried out in this research showed that the proposed thresholds can aid software engineers to identify design flaws and code anomalies.

7. Limitations

Software metric tools might interpret and implement software metrics differently. From the viewpoint of the application of the results, different interpretations of the software metrics represent a threat to the validity of the study. To avoid this problem, the interpretation to be used must be the one we described in this paper.

The sample of applications used in this work might be itself a threat to the validity of the study. Although the sample contains a large number of applications, it is not possible to ensure that they are the best instances of the common practice. In spite of using some well-known open-source software systems in this study, we are not able to confirm that all programs in the sample possess high-quality. We evaluated two programs qualitatively: BCEL and Robocode. BCEL is a library for manipulation of bytecodes of Java classes we used to implement Connecta (Ferreira et al., 2008). We consider BCEL well constructed, modular and easy to use. Robocode is a popular programming game in which Java is used to program robots to do battle against each other. Our evaluation of source code of Robocode concluded that its classes are well constructed, most of them have no public fields and have a short number of methods. However, qualitative evaluation of software structure is error-prone, especially for large software systems. In addition, we considered only Java software in this study. It is possible that the thresholds are influenced by the program language, what would limit the threshold generalization.

One may claim for invalidity of the thresholds of LCOM, since the imperfections of this metric would cause thresholds with no meaning. The main problem with this metric is that when it results in a zero value, it does not necessarily indicate good cohesion. However, this is a limitation of the metric and not of the identified thresholds. The main reason why we chose to evaluate this metric is because it is widely used. Hence, in spite of limitations of this metric, we believe that the identified thresholds for it are still valid and could be useful for developers which use LCOM. Further studies are necessary to identify thresholds for other cohesion metrics in order to provide better software evaluation by means of cohesion.

The experiments we carried out to evaluate the proposed thresholds are based on the qualitative assessment of software systems. In the first experiment, we inspected manually classes from different software systems. This evaluation can be error-prone because, in some cases, the proper assessment of those software systems requires a solid knowledge of the problem domain treated by them. In the second experiment, we evaluated a software system whose design has been considered good in prior works. This premise, however, depends on the validity of those previous works.

8. Conclusions

This work presents a study carried out on a large sample of object-oriented, open-source programs. We analyzed data from 40 programs developed in Java, including tools, libraries and frameworks, of varying sizes and from 11 application domains, in a total of more than 26,000 classes. From the achieved results, we suggest thresholds for six object-oriented software metrics: COF, LCOM, DIT, afferent couplings, number of public methods and number of public fields. The study concluded that values of those metrics, except DIT, can be modeled by a heavy-tailed distribution. This property means that, for most metrics, there is a low number of occurrences of high values and a far higher number of occurrences of low values. Values of DIT can be modeled by the Poisson distribution, having mean value 2.

Based on the most commonly values found in practice, we derived general thresholds for object-oriented software metrics, and thresholds by application domain, size and type (tool, library and framework) of software system. As we did not find relevant difference among them, we believe that the general thresholds can be applied to object-oriented software in general. The identified thresholds were evaluated by means of two experiments. The results of this evaluation indicate that the proposed thresholds can help to identify classes which violate design principles. Moreover, the achieved results suggest that the thresholds can effectively evaluate a design as good when it really is.

The proposed thresholds were derived from common practice. This method could be interpreted as misleading, since the common practice not necessarily represents the good practice. However, the *good* ranges of the derived thresholds showed themselves to be compatible with some well-known design principles. For instance, in a software system, every module should communicate with as few others as possible. The *good* range of COF is 0.02, and the *good* range of *number of afferent couplings* is 1, reflecting that design principle. The *good* range of *number of public fields* is 0, what is compatible with the principle *avoid making data public*. The *good* range of LCOM is 0, what is compatible with the principle that a class should have a high cohesion. A large number of methods in a class is a sign that the class may have too many responsibilities. The *good* range of *number of public methods* is 0–10, reflecting that a class should have few responsibilities. The typical value identified for DIT is 2, what is compatible with the principle *favor object composition over class inheritance*.

The results of this study suggest that the method used to derive the thresholds can lead to identify proper thresholds for software metrics. However, the effectiveness of the method depends on the sample of software systems because if a threshold is obtained by poorly designed software systems, this threshold would not be compatible with design principles.

There are dozens of software metrics. In the scope of this work, we defined thresholds for 6 of them. The results of this work can aid software engineers in evaluating software systems by means of metrics. Nevertheless, effort to define thresholds for other software metrics is necessary in order to encourage the effective application of metrics in Software Engineering. We suggest using the approach of this study in future works in order to define reference values for other software metrics.

Acknowledgments

This work was sponsored by FAPEMIG, as part of the project CONNECTA Process: CEX APQ-3999-5.01/07. Fapemig.

References

Abreu, F.B., Carapuça, R., 1994. Object-oriented software engineering: measuring and controlling the development process. In: Proceedings of 4th International Conference of Software Quality , McLean, VA, USA. October.

Baxter, G., Frean, M., Noble, J., Rickerby, M., Smith, H., Visser, M., Melton, H., Tempero, E., 2006. Understanding the shape of java software. In: OOPSLA'06 , OR, Portland, USA.

Briand, L.C., Daly, J.W., Wüst, J., 1999. A unified framework for cohesion measurement in object-oriented systems. IEEE Transactions on Software Engineering 25, 91–121.

Chidamber, S.R., Kemerer, C.F., 1994. A metrics suite for object oriented design. IEEE Transactions on Software Engineering, 476–493.

Czibula, I.G., Czibula, G., 2008. Clustering based automatic refactorings identification. In: Proceedings of the 2008 10th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing. IEEE Computer Society , Washington, DC, USA, pp. 253–256.

Daly, J., Brooks, A., Miller, J., Roper, M., Wood, M., 1996. An empirical study evaluating depth of inheritance on the maintainability of object-oriented software. Empirical Software Engineering 1, 109–132.

Fenton, N.E., Neil, M., 2000. Software metrics: roadmap. In: ICSE'00: Proceedings of the Conference on the Future of Software Engineering. ACM , New York, NY, USA, pp. 357–370.

Ferreira, K.A.M., Bigonha, M.A.S., Bigonha, R., Mendes, L.F.O., Almeida, H.C., 2009. Reference values for object-oriented software. In: XXIII Brazilian Symposium on Software Engineering , Fortaleza, Ceará, Brazil, pp. 62–72.

Ferreira, K.A.M., Bigonha, M.A.S., Bigonha, R.S., 2008. Reestruturação de software dirigida por conectividade para redução de custo de manutenção. Revista de Informática Teórica e Aplicada 15 (2), 155–179.

Fowler, M., 1999. Refactoring—Improving the Design of Existing Code. Addison Wesley.

Gamma, E., Helm, R., Johnson, R., Vlissides, J., 1994. Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley.

Jancke, S., 2010. Smell Detection in Context. Diploma thesis. University of Bonn. Bonn, Germany. http://dirkriehle.com/computer-science/research/dissertation/.

Kessentini, M., Vaucher, S., Sahraoui, H., 2010. Deviance from perfection is a better criterion than closeness to evil when identifying risky code. In: Proceedings of the IEEE/ACM International Conference on Automated Software Engineering. ASE '10 , ACM, New York, NY, USA, pp. 113–122.

Kitchenham, B., 2009. What's up with software metrics?—A preliminary mapping study. The Journal of Systems and Software 83, 37–51.

Lanza, M., Marinescu, R., 2006. Object-Oriented Metrics in Practice: Using Software Metrics to Characterize, Evaluate, and Improve the Design of Object-Oriented Systems. Springer-Verlag, Germany.

Lincke, R., Lundberg, J., Löwe, W., 2008. Comparing software metrics tools. In: ISSTA '08: Proceedings of the 2008 International Symposium on Software Testing and Analysis , ACM, New York, NY, USA, pp. 131–142.

Louridas, P., Spinellis, D., Vlachos, V., 2008. Power laws in software. ACM Transactions on Software Engineering and Methodology 18 (September 1).

Martin, R., 1994. OO design quality metrics—An analysis of dependencies. http://www.objectmentor.com/resources/articles/oodmetrc.pdf.

Mathwave, 2010. EasyFit. http://www.mathwave.com/products/easyfit.html.

Meyer, B., 1997. Object-oriented Software Construction, 2nd ed. Prentice Hall International Series, USA.

Newman, M.E.J., 2003. The structure and function of complex networks. In: SIAM Reviews. vol. 45 , pp. 167–256.

Potantin, A., Noble, J., Frean, M., Biddle, R., Maio, 2005. Scale-free geometry in oo programs. Communications of the ACM 48 (5), 99–103.

Puppin, D., Silvestri, F., 2006. The social network of java classes. In: SAC'06: Proceedings of the 2006 ACM Symposium on Applied Computing. ACM , New York, NY, USA, pp. 1409–1413.

Riehle, D., 2000. Framework design: a role modeling approach. Dissertation No. 13509, ETH Zürich. http://dirkriehle.com/computer-science/research/dissertation/.

Seng, O., Stammel, J., Burkhart, D., 2006. Search-based determination of refactorings for improving the class structure of object-oriented systems. In: Proceedings of the 8th annual Conference on Genetic and Evolutionary Computation , GECCO '06. ACM, New York, NY, USA, pp. 1909–1916.

Sommerville, I., 2000. Software Engineering, 6th ed. Addison-Wesley.

Tempero, E., 2008. On measuring java software. In: ACSC2008—Conferences in Research and Practice in Information Technology (CRPIT). vol. 74 , Wollongong, Australia.

Wheeldon, R., Counsell, S., 2003. Power law distributions in class relationships. In: Proceedings of 3rd International Workshop on Source Code Analysis and Manipulation (SCAM) , September.

Xenos, M., Stavrinoudis, D., Zikouli, K., Christodoulakis, D., 2000. Object-oriented metrics—a survey. In: FESMA 2000 , Madrid, Spain.

Kecia Aline Marques Ferreira received her Ph.D. in Computer Science from Federal University of Minas Gerais (Brazil) in 2011. She is Professor and researcher of Computer Science at Federal Center for Technological Education of Minas Gerais. Her main research interest is software measurement and its applications in software development and maintenance. Her most recent research has focused on software metrics, software evolution and software maintenance.

Mariza Andrade da Silva Bigonha received her Ph.D. in Computer Science from Pontifícia Universidade Católica do Rio de Janeiro (Brazil) in 1994. She is Professor and researcher of Computer Science at Federal University of Minas Gerais (Brazil) since 1994. Her main research interests are programming languages, compilers and code optimization.

Roberto da Silva Bigonha received his Ph.D. in Computer Science from University of California, Los Angeles (1981). He is Professor and researcher of Computer Science at Federal University of Minas Gerais (Brazil) since 1974 and a member of the Brazilian Computer Society. His main research interests are programming languages, compilers and formal semantics.

Luiz F.O. Mendes received his bachelor degree in Computer Science from Federal University of Minas Gerais (Brazil) in 2009. His current activities are related to software development and social networks analysis.

Heitor C. Almeida received his bachelor degree in Computer Science from Federal University of Minas Gerais (Brazil) in 2009. His current activities are related to Software Engineering.