Empir Software Eng (2008) 13:473–498
DOI 10.1007/s10664-008-9080-x

Theory of relative defect proneness

Replicated studies on the functional form of the size-defect relationship

A. Güneş Koru · Khaled El Emam · Dongsong Zhang · Hongfang Liu · Divya Mathew

Published online: 5 September 2008
© Springer Science + Business Media, LLC 2008
Editor: Tim Menzies

Abstract In this study, we investigated the functional form of the size-defect relationship for software modules through replicated studies conducted on ten open-source products. We consistently observed a power-law relationship where defect proneness increases at a slower rate compared to size. Therefore, smaller modules are proportionally more defect prone. We externally validated the application of our results for two commercial systems. Given limited and fixed resources for code inspections, there would be an impressive improvement in the cost-effectiveness, as much as 341% in one of the systems, if a smallest-first strategy were preferred over a largest-first one. The consistent results obtained in this study led us to state a theory of relative defect proneness (RDP): In large-scale software systems, smaller modules

A. G. Koru (✉) · D. Zhang · D. Mathew
Department of Information Systems, UMBC, 1000 Hilltop Circle, Baltimore, MD 21250, USA
e-mail: gkoru@umbc.edu

D. Zhang
e-mail: zhangd@umbc.edu

D. Mathew
e-mail: dmathew2@umbc.edu

K. El Emam
Childrens Hospital of Eastern Ontario, CHEO Research Institute,
E-Health Information Laboratory, 401 Smyth Road,
Ottawa, Ontario K1H 8L1, Canada
e-mail: kelemam@uottawa.ca

K. El Emam
Faculty of Medicine and School of Information Technology,
University of Ottawa, Ottawa, Ontario, Canada

H. Liu
Department of Biostatistics, Bioinformatics, and Biomathematics, School of Medicine,
Georgetown University, Suite 180, Building D, 4000 Reservoir Rd, NW,
Washington, DC 20057-1484, USA
e-mail: hl224@georgetown.edu

will be proportionally more defect-prone compared to larger ones. We suggest that practitioners consider our results and give higher priority to smaller modules in their focused quality assurance efforts.

Keywords Software metrics · Software science · Size–defect relationship · Planning for software quality assurance · Open–source software · Software inspections · Software reviews · Software testing

1 Introduction

Size is one of the most important measures frequently used by the software engineering practitioners and researchers. For example, it is used to estimate the effort and schedule for developing software modules and products; it is associated with defect proneness – larger software modules and products typically have higher number of defects. Because of its significance, size has been used as an adjustment factor when evaluating quality (e.g. defect density measures), and it needs to be incorporated in quality prediction models because of its significant confounding effect (El Emam et al. 2001). However, the functional form of the crucial relationship between size and defect proneness for software modules (henceforth, size-defect relationship) has not been well investigated or understood thus far (El Emam et al. 2002; Fenton and Neil 1999; Fenton and Ohlsson 2000).

In the PROMISE (2007) Workshop, we reported that the defect-proneness of software classes in Mozilla[1] increased by 44% with one unit of increase in the natural logarithm of size (Koru et al. 2007). This finding suggests a nonlinear size-defect relationship where defect proneness monotonically increases with module size but *at a slower rate*. Consequently, smaller modules should be proportionally more troublesome compared to larger ones. In an earlier study (Basili and Perricone 1984), defect densities of smaller and larger FORTRAN modules were compared, which gave surprising results to the authors at that time because smaller modules had higher defect densities. Later, it was shown that studying size versus its inverse, defect density, inherently gives higher defect densities for smaller modules even when there is no correlation in the original data set (El Emam et al. 2002). Nevertheless, it turns out that the main conclusion drawn by Basili and Perricone was similar to ours in the sense that smaller modules were judged to be proportionally more problematic compared to larger ones.

The study reported in Koru et al. (2007) was conducted with the initial intent of estimating the effect of module size on defect proneness. The insights gained from that study motivated us to investigate the functional form of the size-defect relationship by pursuing a much more extensive study with significant internal replication of the results. The reason is that the observations presented in Koru et al. (2007), and expressed earlier in Basili and Perricone (1984), suggest that under certain plausible conditions software managers and developers should assign a higher priority to smaller modules in their focused quality assurance activities, such as software testing and inspections.

[1] Webcite link: http://www.webcitation.org/5RqqbCKKm (cached Sep. 14, 2007)

So far, recommendations from contemporary studies in this area have been that the practitioners should focus on more complex and larger modules, as discussed further in Section 2. Even some researchers assessed the net benefits of their defect prediction models by taking a largest-first simplified prediction approach as their baseline, e.g., (Briand et al. 2002; Ostrand et al. 2005), after seeing that size by itself was the strongest explanatory variable in their data sets. Therefore, it is important to ground the findings presented by Koru et al. (2007) with strong empirical evidence, preferably from multiple products, before practitioners start to adjust their quality assurance practices accordingly.

Building upon the earlier results and the observations made in Koru et al. (2007), we further investigated the functional form of the size-defect relationship and tested the hypothesis that smaller modules are proportionally more defect prone. For this purpose, we collected and analyzed module-level size and defect data from ten open-source software products (Koru et al. 2007). The open-source data used in this research will be publicly available in the PROMISE data repository (Promise 2007). Therefore, the analysis results will be reproducible and verifiable.

In the rest of this paper, we start by discussing the related work. Then, we explain the methodology used in the study. After that, we present our results followed by a discussion of the cost effectiveness of giving higher priority to smaller modules. Then, we demonstrate the benefits of our recommendations by using size and defect data from two commercial systems. We also discuss the limitations of the study and mention the future research directions, after which we conclude the paper.

2 Related Work

There is a significant and growing body of work that predicts defect-proneness using size and other structural measures. However, the studies in this area have not explicitly examined the functional form of the size-defect relationship. In this section, we discuss the other related studies to provide context.

A large number of studies directly built regression models by using size as the predictor variable and defect count as the response variable. Sometimes, instead of size, other code measures, e.g. McCabe's cyclomatic complexity (McCabe 1976) and Halstead measures (Halstead 1977), were used as predictors. Later, those measures were found to be strongly correlated with size (Fenton and Pfleeger 1996; Fenton and Neil 1999). The earlier studies often built linear regression models, where the linearity of the size-defect relationship was implicitly accepted (Akiyama 1971; Funami and Halstead 1976; Shen et al. 1985). This approach was sometimes defended for its practicality and sometimes for the availability of tool support (Shen et al. 1985).

Some researchers derived models analytically first and then fit the data to validate those models. Lipow's logarithmic model (Lipow 1982) and Gaffney's models (Gaffney 1984) are such examples. However, in (Lipow 1982) defect density and size were included together in the same model, which suffers from artificial ratio correlations (Chayes 1971). In Gaffney (1984), the significance of the deviation from linearity was not shown, and alternative (non-linear) models were not explored.

Another group of studies used defect density (defect count divided by size) in order to adjust for size and reported that there was a U-shaped curve when defect density was plotted against size (Compton and Withrow 1990; Hatton 1997, 1998; Withrow 1990). Defect density was observed to be high for smaller and larger module sizes, and an optimal medium size minimizing defect density was reported. Such observations are generally called the Goldilock's conjecture (El Emam et al. 2002). As a result, developers were advised to produce medium size modules: not too small, not too big. The early study of Basili and Perricone (1984) can be put into this category too because the authors studied defect density against size when they made quality comparisons. Recently, this group of studies was criticized because of their analysis approach of plotting or studying size against defect density (El Emam et al. 2002). Such an approach masks the original relationship between size and defects, resulting in superior correlations and misleading conclusions, as also stressed by Rosenberg (1997) and Fenton and Neil (1999). The reason is that, when plotted against size, defect density will be almost always high (caused by using a small denominator in the division operation); then, it will sharply drop hitting a minimum value at a medium size and then, it will start to raise again. Consequently, the U-shaped curves, threshold effects, and optimum module sizes observed by following this approach are arithmetic artifacts (El Emam et al. 2002). Such observations could be reported even if there was no particular relationship between size and defects.

More recently, machine learning and data mining techniques such as neural networks (Khoshgoftaar et al. 1997), tree-based modeling (Koru and Tian 2003), and optimal set reduction (Briand et al. 1993), were employed to create quality prediction models. These models often use size as a predictor variable along with others. Such models do not assume a functional form for the size-defect relationship. However, it is often difficult to have a good understanding of the nature of the size-defect relationship from the resulting models (e.g., from a neural network model).

Therefore, if we put our study in the context of the existing literature of software engineering, focusing this research on the basic size-defect relationship is important for several reasons:

1. As defect prediction models get larger and more complex (e.g. with many predictor variables and rules) and customized to specific environments, it becomes more difficult to interpret them and generalize them to different environments. Instead, smaller models can inform practitioners better about what kind of functional relationships should be expected; they can become generalizable and gain widespread use in practice.
2. Many earlier studies focused on predicting defects and improving the prediction accuracy; however, they did not consider how the resulting models would be used in a development context, which involves making business decisions about how to allocate the limited quality assurance resources. For example, many studies suggest that the practitioners focus the quality assurance activities on more complex modules, which are also larger because of the now well-known correlation between complexity metrics and size (El Emam et al. 2001; Fenton and Pfleeger 1996; Meine and Miguel 2007). However, it is often overlooked that inspecting or testing those modules will also consume more time and effort. Knowing the nature of the size-defect relationship will enable the practitioners to invest their resources in a more efficient and effective manner. Such benefits will be further discussed in Section 5.
3. Even if it is only for descriptive purposes, understanding the size-defect relationship is very important. Still, many practitioners are using defect density

to make quality comparisons among software modules, which assumes a linear size-defect relationship. However, if smaller modules are always proportionally more defect-prone than larger modules, and if this phenomenon can be repeatedly observed by looking at different products, then using defect density for quality comparison among software modules without considering the nature of size-defect relationship will always penalize smaller modules. Therefore, better understanding of the plain size-defect relationship is important because it will enable us to make more informed quality comparisons among software modules.

3 Methods

The main objective of this study was to investigate the functional form of the size-defect relationship and test the hypothesis that smaller modules are proportionally more defect prone. We measured size in LOC (lines of code), and related it to defect-proneness at the module level.

The size and defect data came from ten different products in the KOffice[2] suite, which were developed by different programmer teams with the support of an open-source community surrounding them. The KOffice suite is an open-source alternative to closed-source commercial office suites. All of the KOffice products studied were written in C++. Therefore, rather than using files as modules, we considered each class a module because, in object-oriented programs, a class represents a logically cohesive software unit.

The development process for the KOffice products was similar to those in many other open-source projects, which typically involves the identification of tasks (e.g., various enhancement and maintenance tasks), identification of volunteers, and task execution (Mockus et al. 2002). Typically, open-source development processes are loosely organized, where products are continuously evolved (Raymond 1999) as opposed to better defined processes traditionally seen in the closed-source development world, which can involve designated planning, analysis, and design phases and activities.

In this section, we start with an overview of Cox proportional hazards modeling (Cox modeling, henceforth), which is the analysis method used. We explain the application of this method in this study where a different Cox model was created for each product analyzed. Then, we explain the data used in this study and the specific data collection procedures.

3.1 Cox Modeling

Cox modeling was proposed to understand the effects of covariates on the instantaneous relative risk for an event of interest (Cox 1972; Harrell 2001; Hosmer and Lemeshow 1999; Therneau and Grambsch 2000). It is used frequently in epidemiology to understand the effects of certain patient characteristics (e.g., age, medication, sex) on the relative risk of death at any given time, which can be also subject to some other unknown or uncontrollable effects. Data used for Cox models are called

[2]Webcite link: http://www.webcitation.org/5Rqr0BSz8 (cached Sep. 14, 2007)

censored data because the event of interest may or may not occur for an observed subject. Therefore, the event information must be attached to observations. The Cox model was later connected to the counting process theory to accommodate recurrent events (Andersen et al. 1993). Accommodating recurrent events (also called multiple events) enables analysts to take the changes in the predictor variables, which can take place over time, into account. In our research, we use this technique to model recurrent defect fixes and changes in module size.

Cox modeling with recurrent events is especially suited to software quality modeling for non-traditional software development environments, where software modules evolve, change in size, and experience non-corrective and corrective changes in a concurrent and continuous fashion. Open-source and agile development projects are good examples of such development environments.

In those environments, measuring software modules in a specific system snapshot and relating those measurements to future defects could pose serious internal validity threats mainly due to the size changes in modules and due to the modules deleted over time. Indeed, the frequent daily changes in the KOffice projects studied in this paper have been well recognized by the KOffice developers and documented by researchers (Askari and Holt 2006). In this situation, we cannot make an assumption that a module's size will be fixed during its life time when it undergoes various corrective, perfective, and adaptive maintenance work. One could obtain a sample of non-deleted classes whose size does not change. However, this would significantly reduce the amount of available data. Therefore, Cox modeling becomes necessary.

We obtained a complete size and change history of every class (module) introduced to the KOffice products during our observation period. Each change made to a class resulted in a new snapshot for that class. For each such change, we determined the new class size, date and time of change, and whether the change was a corrective change or not. An individual observation (data point) was created for each snapshot of a class with the size and time data and added to the data set. For deleted classes, no new observation was added after the last one.

In our research, an event corresponded to a defect fix made to a C++ class. Since it is impossible to know defects ahead of time, defect fixes have been traditionally used as an indicator of defect proneness in many studies in this area, e.g., (Basili and Perricone 1984; Munson and Khoshgoftaar 1992; Troster and Tian 1995). The instantaneous risk (probability) of experiencing an event, which is also called hazard, at any time t was used as the indicator of defect-proneness. This hazard is modeled by the following hazard function:

$$\lambda_i(t) = \lambda_0(t)e^{\beta x_i(t)} \tag{1}$$

where,

- $x_i(t)$: Size of class i at time t (class size is a time-dependent covariate)
- β: The coefficient for size
- λ_0: Baseline hazard function, which is the hazard when there is no covariate effect

Taking the natural logarithm of both sides of (1), it can be seen that $x_i(t)$ should be linearly related to log hazard. If not, a link function $f(x_i(t))$, which provides a transformation that is linearly related to the log hazard should be found and used instead of $x_i(t)$.

The baseline hazard is an unknown function, and it does not have to be specified. This function is cancelled out when the relative risk between two classes is calculated, which is the emphasis of Cox modeling. The relative risk (also called hazard ratio or relative hazard) for two classes, *i* and *j*, at any time *t*, can be written as (using the link function for the sake of generality):

$$\lambda_i/\lambda_j = e^{\beta(f(x_i(t))-f(x_j(t)))} \tag{2}$$

Therefore, the relative risk at any given time should depend only on the covariate values. The coefficient, β, should be a constant over time. A Cox model must satisfy this condition, which is called the proportional hazards assumption. As discussed in Section 4.3, we checked all of our models to make sure that this assumption was satisfied.

A nice characteristic of any Cox model is that it can accommodate multiple baseline hazards. Still, a single coefficient estimate is produced for each covariate, which applies to all baseline hazards. We take advantage of this characteristic and create different baseline hazards for different observation categories. This stratification helps preserve the proportionality in Cox models. The idea behind stratification in Cox modeling is similar to the idea of stratification in other statistical methods. If there is a factor that needs to be controlled because it is of secondary or no interest, it can be used to divide the data set. As a result, the effect of more important covariate(s) can be observed better.

The specific Cox models produced in this study using stratification are called conditional models (Therneau and Grambsch 2000), which are a specialized form of recurrent Cox models. In the conditional Cox models, the subjects start at an initial state and can make certain state transitions. Each state corresponds to a different risk set with a different baseline hazard. At any given time, a subject belongs to only one particular risk set. The states in our study were identified according to the number of prior events (defect fixes) experienced. We used four states corresponding to 0, 1–5, 6–25, and >25 *prior* events, labeled from 1 to 4, respectively. These states were determined empirically by examining the range of the number of events per class across products and by testing whether using those states preserve proportionality for all models. A class starts at state 1, and can make a transition from state n to state $n+1$ $(0 < n < 4)$ according to the number of events it has experienced. The state information was also attached to each observation as a numeric field.

To summarize, we followed the steps below in our modeling for each product analyzed:

1. We found the appropriate link function for size by using the restricted cubic splines (Harrell 2001). This approach relaxes the linearity assumption for the link function and plots size against the log-relative hazard by dividing size into multiple ranges identified by knots and by fitting a cubic polynomial for each individual size range. The advantage of using cubic splines over linear ones is that they provide better fits by curving at the knot points (Harrell 2001).
2. We built a conditional Cox model using the identified link function of size, and examined the significance of the covariate. We interpreted the models and what they meant in terms of the size-defect relationship.
3. We applied a number of model diagnostics, which included checking whether the model satisfied the proportional hazards assumption or not, examining the overly influential points, and checking the overall fitness of the model.

4. Finally, we obtained the functional form of the size-defect relationship and tested whether smaller modules are proportionally more defect prone. This step is further discussed in the results section because it uses the particular link function identified in this study.

3.2 Data Description

The C++ classes in our study were those created during our observation window, which was from April 18, 1998, when developers started to add source code to the public code repository of the KOffice products, to January 19, 2006, the end date of our data collection. For each of the ten KOffice products, a distinct data set was created. All of the data sets had the same format shown in Table 1, which includes hypothetical data for demonstration purposes.

The data was prepared according to the following rules:

1. A new observation was created when a class was first created, or whenever it was modified. Therefore, each observation corresponding to a row in Table 1 belongs to a class snapshot with the creation time, *Start*. For the first observation of any class, *Start* was set to zero. Note that, in modeling, we always use study times, which are different from calendar times; two classes introduced at different calendar times would both have zero as their Start time.
2. The time *End* was set with either the class' next modification time, or the time our observation period ended, or the deletion time, whichever applied or came first.
3. *Event* was set to 1 if the change taking place at time *End* was a defect fix, 0 otherwise.
4. The *State* column included the state of a class (explained in Section 3.1) at the time of *Start*.
5. *Size* was the class size at the time *Start*, as measured by LOC excluding blank and comment lines.

The above rules allow modeling of class deletions and the changes in class size, which is difficult to deal with using the traditional quality modeling methods. The traditional approaches usually measure the modules in a single or limited number of identified system snapshots, and associate the measurements with the future defect count or with a binary variable.

Table 1 Format of the data for KOffice products (hypothetical data used)

Class name	Size	Start	End	Event	State
C	100	0	20	1	1
C	300	20	40	0	2
D	250	0	30	1	1
D	150	30	40	1	2
D	100	40	80	1	2
D	170	80	100	1	2
D	200	100	130	1	2
D	300	130	180	1	2
D	450	180	200	0	3
...	...	...	...	...	...

The data in the format shown in Table 1 was fed to the statistical analysis environment, R (R Development Core Team 2003) for Cox modeling. More specifically, we used the Design Package (Harrell 2005) to obtain the Cox models. Then, we used the Survival (Therneau 1999) package to obtain the robust standard error estimates of β (not available in the Design package). These two packages require us to put the data in the format seen in Table 1 (also known as conditional counting process format).

3.3 How Cox Modeling Works

At this point, it is useful to discuss how a data set that looks like Table 1 is used for Cox recurrent event modeling purposes because this format is different from what is typically seen in ordinary regression modeling. In ordinary regression, each observed unit (in this case, each software class) corresponds to only one observation. The data for each unit includes a response variable and a set of predictor variables, which are represented across the row for each observed unit. In Cox recurrent event modeling, each observed unit might have one or more observations, each observation with its own corresponding row. Each observation includes time stamps (*Start* and *End*), event data (*Event*), and covariates (in this case only one covariate, *Size*). The covariate(s) are used as predictors; and, the time stamps and event columns are used to derive the measure of interest, relative risk.

In order to see how Cox recurrent events modeling works, imagine that a new table of event times was built from Table 1 by taking the distinct *End* values where *Event* is 1. This new format can be seen in Table 2.

To simplify this presentation, let us assume that *Size* is transformed to a dichotomized variable *Large* by using a threshold value. That is:

$$Large = \left\{ \begin{array}{l} 1, \text{if } Size > Threshold \\ 0, \text{Otherwise} \end{array} \right\} \tag{3}$$

In this case, directly following (2), the relative risk of a large class, L (having $Large = 1$), compared to a small one, S (having $Large = 0$), at any time can be simply modeled with:

$$\lambda_L/\lambda_S = e^{\beta} \tag{4}$$

In Table 2, at each event time, the number of classes experiencing an event and those that are at risk can be noted for both large and small categories. Then, for both categories, we can divide the number of classes experiencing an event by those that are at risk. This will give us the hazard rates for large and small classes, λ_L and λ_S, respectively for each event time. The estimate of the relative risk can be obtained

Table 2 Table of event times (fictitious data is included for demonstration purposes)

Event time	Large classes		Small classes		Point estimates		Relative risk ($\hat{\lambda}_L/\hat{\lambda}_S$)
	Number having an event	Number at risk	Number having an event	Number at risk	Hazard for large classes ($\hat{\lambda}_L$)	Hazard for small classes ($\hat{\lambda}_S$)	
Time 1	10	450	20	2,500	0.02	0.008	2.5
Time 2	7	400	14	2,800	0.0175	0.005	3.5
…	…	…	…	…	…	…	…

by calculating λ_L/λ_S at each event time. Finally, we can empirically obtain a point estimate of β by using (4) as the model and Table 2 as the data set. Note that, in a table like Table 2, the number of classes at risk in each risk set (corresponding to a *State*) will be most probably different at different times because of the added and deleted classes.

In this study, a continuous covariate, *Size*, is used rather than a dichotomous one (*Large*). The estimation of β is based on similar principles but is achieved by identifying the value that maximizes a partial likelihood function. The explanation of this method is substantially longer. Readers can find more information in Cox's original paper (Cox 1972) and (Hosmer and Lemeshow 1999; Therneau and Grambsch 2000).

3.4 Data Collection

The raw data, used to obtain our data set in the format shown in Table 1, were collected from the CVS[3] databases of the analyzed products. We used a reverse engineering tool, Understand for C++ (Scientific Toolworks 2003), for measurement purposes. We developed PERL[4] scripts to examine the CVS check-ins that took place during our observation period one by one. For each CVS check-in, our programs asked Understand for C++ to identify the changed, added, and deleted classes, and exported the size measurements for them. The lines that were only comment lines or blank lines were excluded in size measurement. The *Start* and *End* fields were calculated in minutes from the time tags of the CVS check-ins. The events were detected by searching for the words 'bug', 'fix', and 'defect' in the CVS logs in a non-case sensitive manner. Descriptive information about the products analyzed in this study and about their data sets is shown in Table 3.

After the product name and functionality, Table 3 first presents the number of classes created, observations made, and the number of events that took place during our observation period. The number of observations (n) per product is equal to the number of rows (in Table 1) for each data set. To give an idea about the size of the products, Table 3 also shows the number of classes and the total size for each product as they appeared in the KOffice 1.4.2, which was the last release before our observation period ended. The sizes of those products ranged roughly from seven to 132 KLOC (thousand lines of code). Therefore, the products we studied provided a large number of data points for our analysis. The external libraries and the libraries shared among the KOffice products were excluded from the analysis.

4 Results

In this section, we present the results of the analysis steps enumerated in our Cox modeling section (Section 3.1).

[3]CVS was the source code control system used by the KOffice developers. Webcite link: http://www.webcitation.org/5RrT2BaV1 (cached Sep. 14, 2007)

[4]Perl is a stable, cross platform programming language. Webcite link: http://www.webcitation.org/5RrTDEdYV (cached Sep. 14, 2007)

Table 3 Descriptive information about the data sets used in the study

Product	Functionality	During the observation period (Apr 18, 1998–Jan 19, 2006)			In KOffice 1.4.2 (Oct. 11, 2005)	
		Number of classes created	Number of observations made (n)	Number of events (defect fixes)	Number of classes	Total LOC
Karbon	Scalable vector drawing	382	5,072	1,242	179	24,219
KChart	Chart drawing	112	406	98	77	26,787
Kexi	Integrated data management	250	613	106	507	60,219
KFilter	Conversion among various file formats	1,131	5,045	1,142	882	132,226
Kivio	Diagramming and flowcharting	191	1,431	377	138	22,184
KPresenter	Presentation	409	2,380	608	175	47,348
Krita	Painting and image editing	1,210	9,149	2,961	513	47,612
KSpread	Scriptable spreadsheet	587	5,339	1,789	231	65,373
Kugar	Tool for generating business quality reports	129	602	112	58	7,257
KWord	A frame-based word-processing and desktop publishing program	802	5,953	1,932	193	42,900

4.1 Identifying the Link Function

We started by identifying the appropriate link function for each product which is achieved by plotting each covariate against log-relative risk (Therneau and Grambsch 2000; Harrell 2001). Figure 1 shows the relationship between size and log-relative hazard for each product using four knots placed at quartile size values and restricted cubic splines. Usually, using four knots results in adequate fits as stressed by Harrell (2001).

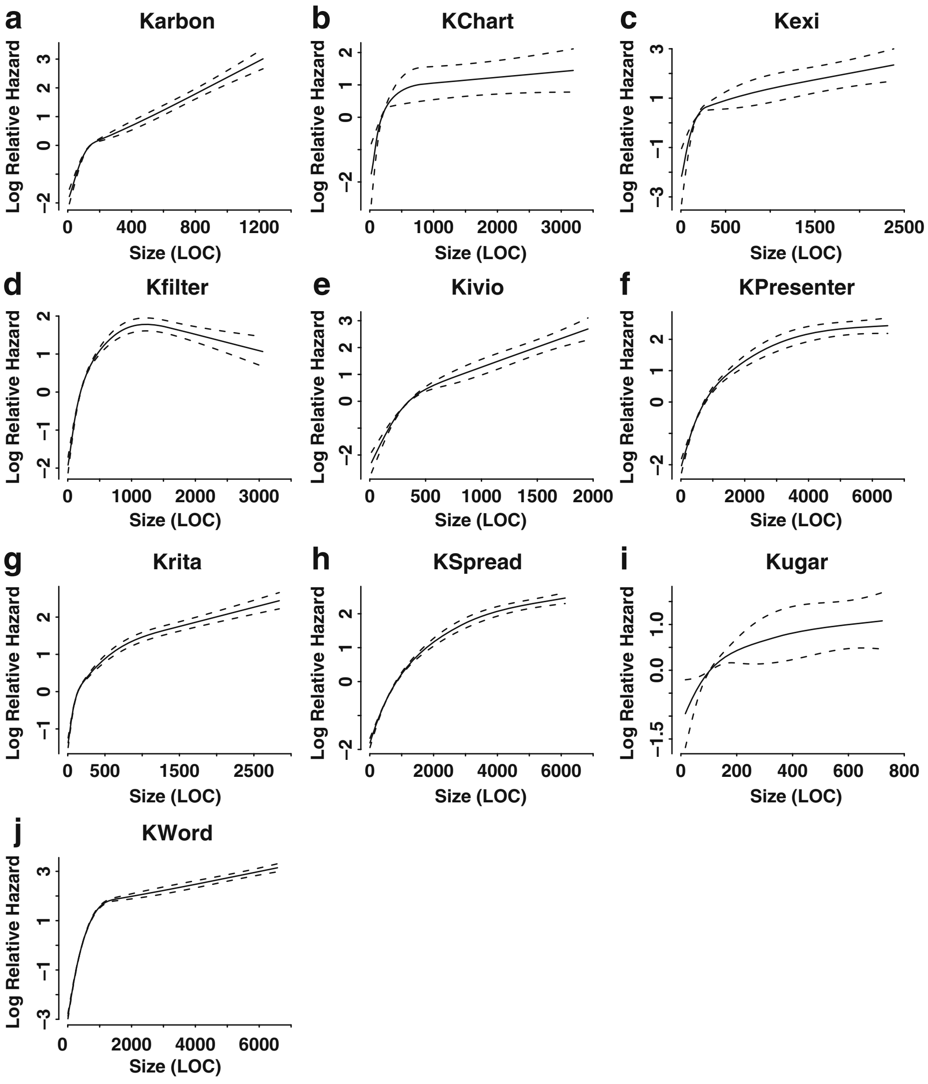

Fig. 1 Link functions for all KOffice products studied

These plots show both the main curve and the 95% confidence intervals (dashed curves). A log-relative hazard of zero was arbitrarily assigned to a class whose observations were close to the mean size when all observations were taken into account. The width between the dashed curves becomes zero at the point where the log-hazard is zero. To have a plot less affected by the outliers, the smallest and largest 10 observations were omitted. The assignment of zero log-relative hazard and the omission of the extreme values are standard procedures performed automatically by the plotting function in the Design package (Harrell 2005).

After plotting Fig. 1, we visually examined the plots to find a generally applicable link function as usually done (Therneau and Grambsch 2000). The nonlinearity of the curves in Fig. 1 is very obvious. Generally speaking, the link function looks similar to a logarithmic curve. Only in KFilter, there is a curve down showing less risk for very large classes (for about 1% of the total number of classes). Because of this commonly observed shape, we used natural logarithm as the link function. In our models, $\ln Size$ was used as the predictor variable instead of Size.

Note that the shape of this link function reflects the relationship between size and *log-relative risk*. The main purpose of identifying the link function is to find a transformation of size so that the same differences in the transformed covariate values will have the same multiplicative effect on the hazard, helping preserve the proportionality assumption in the Cox models. Section 4.4 discusses how we used this link function to obtain the functional form of the size-defect relationship and test our hypothesis that smaller modules are proportionally more defect prone.

In some of the medical problems, the important covariates affecting the hazard function and their transformation are already known through prior research. For example, for liver transplantation survival problems, the link function for bilirubin (a substance related to liver function read from blood values) is known to take a roughly logarithmic form (Therneau and Grambsch 2000). The shapes of the link function observed in this paper and also in Koru et al. (2007) have started to accumulate similar evidence in this domain showing that, in Cox models, the link function for size is logarithmic.

4.2 Building Cox Models

Our next step was to build Cox models by using the logarithmic transformation of size as the link function. We built ten different Cox models, each corresponding to one of the ten products included in our study. The modeling results are summarized in Table 4.

It can be noted that the point estimates of β were always above 0, showing the positive effect of size on defect proneness. We also calculated the estimates of the standard errors for $\hat{\beta}$. To take the intra-subject correlations into account (multiple observations per class), we also calculated the robust sandwich estimates for the standard error of $\hat{\beta}$ using grouped Jackknife as shown in Table 4. The models presented in this table can be interpreted as follows:

One unit of increase in the natural logarithm of class size multiplies the rate of experiencing a defect fix by $e^{\hat{\beta}}$.

Table 4 Models for KOffice products

Product	$\hat{\beta}$	Standard error (SE) of $\hat{\beta}$	Robust estimate of SE for $\hat{\beta}$	Non-proportionality test (p-value)	Spearman's ρ between actual and expected event count
Karbon	0.592	0.041	0.069	0.817	0.86
KChart	0.656	0.105	0.109	0.339	0.80
Kexi	0.843	0.107	0.100	0.691	0.73
KFilter	0.583	0.026	0.040	0.770	0.84
Kivio	0.786	0.067	0.075	0.780	0.92
KPresenter	0.590	0.040	0.051	0.402	0.93
Krita	0.414	0.019	0.026	0.061	0.87
KSpread	0.474	0.024	0.033	0.738	0.93
Kugar	0.555	0.096	0.091	0.492	0.74
KWord	0.740	0.026	0.037	0.285	0.96

4.3 Model Diagnostics

After creating the Cox models, we performed a test to see whether the proportionality assumption of the Cox models was satisfied. This test can be performed by checking whether time has a significant interaction with the effect of Size (i.e. $p < .05$). If a covariate has disproportionate effects on hazard over time, it can be detected by this test. A good example could be a drug, which is more effective in the first weeks of a treatment. As noted in Therneau and Grambsch (2000), this test also gives significant results when a wrong link function is used.

As seen in Table 4, the p-values for the time interaction test for all products are well above 0.05. Therefore, there is no evidence for non-proportionality at all, which shows that the proportional hazards assumption is satisfied for all Cox models developed for all products.

After checking the proportional hazards assumption, we examined the overly influential data points by plotting the dfbeta residuals for each product. An example plotted for KWord can be seen in Fig. 2, where one observation at the bottom can be considered overly influential. For all products, the influential data points were identified, examined, and found to be valid data points. Removing them did not change our coefficient estimates significantly due to the large number of data points used in model building. Therefore, we decided to keep our original models.

Finally, we assessed model fitness by examining the Spearman correlation between the actual and expected number of events for classes. The expected number of events for a class was calculated by taking the minus log of the value of the cumulative hazard at the end of the period in which the class was observed. As seen in Table 4, the Spearman's correlation values obtained are high.

Therefore, each one of the produced models passed all of the diagnostic tests for a good-fitting Cox model.

4.4 Obtaining the Functional Form and Hypothesis Testing

As discussed in Section 4.1, our visual examinations identified the link function to be logarithm; therefore we used $\ln Size$ in our models (we denote that with $\ln x$ below

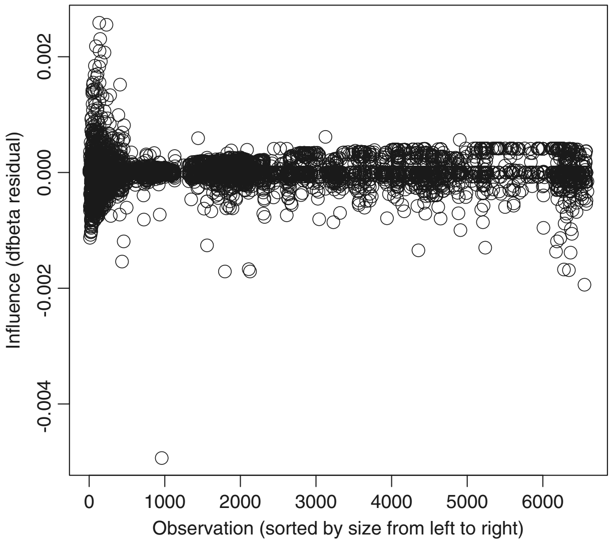

Fig. 2 Influences for the KWord data set

where x represents size). Note that the logarithmic transformation of size satisfied the proportionality assumption in our Cox models seen in Table 4. In this section, we first obtain the functional form of the size-defect relationship, and then test the hypothesis that smaller modules are proportionally more defect prone.

Without loss of generality, following (2), the relative defect proneness (RDP) for two classes j and k having $x_j > x_k$ at any given time t can be written as (here we exclude t for simplicity):

$$e^{\beta(\ln x_j - \ln x_k)} = e^{\beta(\ln(x_j/x_k))} = (x_j/x_k)^{\beta} \tag{5}$$

The equation in (5) means that defect proneness (D) is related to module size (s) raised to the power β. This functional form is a power law (Newman 2005) and it can be generically written as:

$$D = \alpha\, s^{\beta} + \epsilon \tag{6}$$

where α and ϵ represent a positive constant and a deviation term, respectively. As a result, when the logarithm of defect-proneness (y-axis) is plotted against the logarithm of size (on x-axis) using empirical data, it will result in a primarily linear relationship.

To test our main hypothesis, i.e., smaller modules are proportionally more defect prone, we need to examine the value of β because:

- $\beta = 1$ implies that size-defect relationship is linear, and smaller and larger modules are, proportionally speaking, equally defect prone
- $\beta > 1$ implies that defect proneness increases faster than size, and larger modules are proportionally more defect prone
- $\beta < 1$ implies that defect proneness increases slower than size, and smaller modules are proportionally more defect prone

Our main hypothesis can be supported only if the first two cases are rejected strongly. We plotted the β estimates obtained from our models seen in Table 4 with their 95% confidence intervals. The resulting plot is shown in Fig. 3.

Firstly, Fig. 3 shows that there is a positive effect of size on defect proneness for all products ($\hat{\beta} > 0$ significantly). Therefore, as size increases, defect proneness increases too, which shows that size-defect relationship is a monotonically increasing relationship. Secondly, concerning the main hypothesis of this study, $\hat{\beta}$ values and their 95% confidence intervals are significantly smaller than 1 (shown by the dashed horizontal line) for nine products. For Kexi, the same argument can be made with 85% confidence. Therefore, our hypothesis that smaller modules are proportionally more defect prone is strongly supported. This means that when size-defect relationship is plotted on a log-log graph, the slope of the linear fit will be less than 1.

Based on this observation, we can further refine the functional form of the size-defect relationship: there is a power-law relationship between module size and defect proneness where the latter increases at a slower rate.

5 Discussion

Our results based on the analysis of ten open-source products provide strong evidence that smaller modules are proportionally more defect prone. The evidence obtained supports the main hypothesis of this study.

Our findings have important implications on how to allocate resources for focused quality assurance, such as testing and inspections, and how to prioritize those activities. In real settings, there will be additional concerns in making such resource allocation decisions, for example, which modules have higher business value, which modules will be executed more frequently, which modules were referred to more (e.g. library modules with higher fan-in), or which modules are logically related to be examined together. Nevertheless, practitioners should also consider the results presented in this paper. Let us demonstrate the implications of the models presented in Table 4 on a hypothetical example.

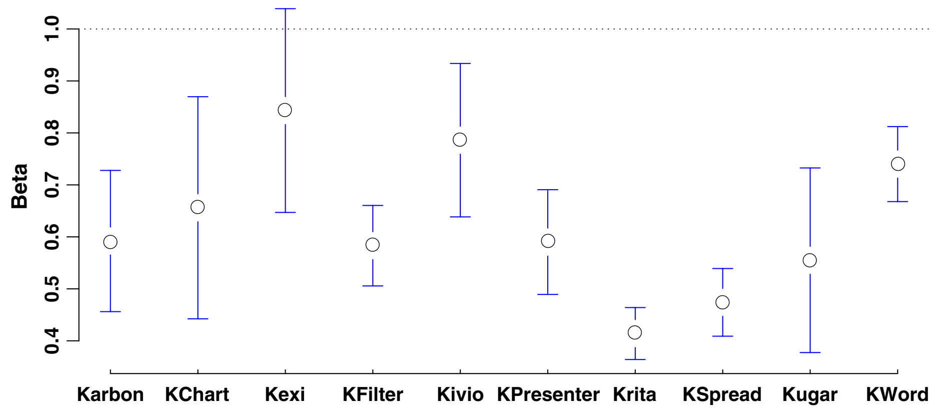

Fig. 3 Coefficient estimates ($\hat{\beta}$'s with their 95% confidence intervals)

Imagine a scenario where each KOffice project is given a limited and fixed amount of resources for code inspections, which is only enough to inspect 10,000 LOC. Let us assume that the developers discussed their business concerns, operational profile, etc., and decided that there are two inspection strategies each identifying a different portfolio of C++ classes for inspection. The first strategy includes 80 classes of 100 LOC and 2 classes of 1,000 LOC; and, the second strategy includes 20 classes of 100 LOC and 8 classes of 1,000 LOC. The classes in this scenario are imaginary and they are only used to demonstrate what information would be obtained from the models presented in Table 4.

In this scenario, both strategies entirely consume the available inspection resources. Understandably, the developers are trying to figure out which strategy would be more cost effective in terms of detecting more defects. Here, for the sake of simplicity, we also assume that there is a linear relationship between the inspection resources spent and size for software modules, and all defects are equally detectable (within and across modules).

According to our results, smaller modules are proportionally more defect prone. Therefore, the first strategy including more classes of smaller sizes will be more cost effective. To quantify the relative cost effectiveness of choosing the first strategy over the second one, we can compare the defect proneness of the classes in the first category to those included in the second category in a relative manner. The Cox models developed in the study allow us to make such relative comparisons. Therefore we define RDP as the relative risk (probability) of experiencing a defect fix. In this case:

$$RDP = \frac{\text{Probability of defect fix for all classes included in the first strategy}}{\text{Probability of defect fix for all classes included in the second strategy}} \tag{7}$$

An RDP value of 1 means that the risk of experiencing a defect fix is equal for the classes included in both strategies (no benefit of choosing one strategy over the other one). If the RDP is greater than 1, this means that the risk of experiencing a defect fix is greater for the classes included in the first strategy. If it is smaller than 1, this means that the classes in the second strategy have a greater risk of experiencing a defect fix. The derivation of the complete formula used to calculate the RDP is provided in the Appendix. Note that, that formula can be used for any two portfolios of modules. This flexibility enables practitioners to give higher priority to some other concerns important for their business, and but still calculate the RDP with respect to module size.

Table 5 shows what would be the RDP of the classes chosen by the first strategy compared to those chosen by the second strategy by using the models presented in Table 4. The 95% confidence intervals are also shown. As seen in Table 5, the models strongly favor the first strategy over the second one. For example, in Krita, choosing the first strategy is 2.09 times effective, in other words 109% more effective, compared with the second one. This scenario demonstrates a potential for substantial and important savings in real-life development efforts considering that software developers usually work under tight deadlines, and they are typically in a rush to deliver their products to market.

In real-life settings, developers might not have existing models similar to those presented in Table 4. Could they simply use the general result of this study, which is, smaller modules are proportionally more defect prone? In other words, would it

Table 5 Relative cost-effectiveness of the first strategy choosing mostly smaller classes compared with the second one choosing mostly larger classes

Product	*RDP* (95% Confidence interval)
Karbon	1.71 (1.45–2.00)
KChart	1.58 (1.20–2.03)
Kexi	1.24 (0.95–1.60)
KFilter	1.73 (1.57–1.90)
Kivio	1.34 (1.10–1.62)
KPresenter	1.72 (1.56–1.88)
Krita	2.09 (2.01–2.17)
KSpread	1.96 (1.82–2.10)
Kugar	1.79 (1.43–2.17)
KWord	1.42 (1.29–1.56)

be advantageous to choose the smallest modules instead of the largest ones without using any statistical or machine learning techniques? If such an approach were useful, developers would only need a program implementing a simple sorting algorithm or a spreadsheet program (e.g. Excel) to sort their classes according to their LOC value.

Note that, so far, practitioners have been always advised to focus on more complex and larger modules because defect proneness generally increase with module size and the other complexity metrics used in the prediction models are highly correlated with module size (El Emam et al. 2001; Fenton and Pfleeger 1996; Meine and Miguel 2007). In fact, the perceived benefits of focusing on larger modules has been so prevalent that even the researchers in this area assessed the benefits of their defect prediction models by taking a simplified largest-first prediction approach as their baseline, e.g., (Briand et al. 2001; Ostrand et al. 2005).

To externally validate the application of our results, we will use data from two commercial software systems:

- The first system is called ACE, which is a telecommunications system developed in C++, and remained in operation for more than fifteen years (Schmidt 1995). The version we used consists of 174 different C++ classes each corresponding to a data point in the ACE data set. Each data point includes the number of non-comment and non-blank source LOC, and the number of post-release defects. In the ACE data set, the total LOC was 11,195, and the total number of defects was 192. This data set was used in a previous study (El Emam et al. 2002).
- The second system is an IBM relational database management system (called IBM-DB for short), which was developed using C++ at the IBM Software Solutions Toronto Laboratory. Each data point in the IBM-DB data set corresponds to a single class, and it includes the number of non-comment and non-blank source LOC, and the number of all defects (pre and post release). In this data set, the total LOC is 185,755 and the total number of defects is 7,824. The IBM-DB data set was also used in the previous studies (Koru et al. 2003; Tian and Troster 1998).

It is useful to first note that the ACE and IBM-DB data sets confirmed our finding that there is a linear relationship between the logarithms of size and defects; the Harvey-Collier test (Harvey and Collier 1977) found no evidence for nonlinearity (p=0.48 for ACE and p=0.22 for IBM-DB).

Now, using the ACE and IBM-DB data, we can understand whether a smallest-first or a largest-first strategy would potentially detect more defects for a limited and fixed amount of resource expenditure. We can assume that both projects are given resources only enough to inspect around 20% of their code (it is generally known that such resources are limited (El Emam 2005)). Then, we can compare the defect proneness of the modules covered by both strategies using defect count as a measure.

The results presented in Fig. 4 show that the advantage of focusing the inspections on the smallest modules is tremendous. Figure 4a shows that in ACE, given enough resources to inspect 2,400 LOC, following the smallest-first strategy would cover a group of modules that have 160% more defects compared to those covered by the largest-first approach. The net gain of this approach, which is the extra number of defects covered by the smallest-first strategy, is plotted in Fig. 4c. It is observable that the benefit of the smallest-first strategy quickly increases even when a small amount of code is inspected. Normally, the rate of increase in this benefit diminishes as more code gets inspected. As seen in Fig. 4b, for IBM-DB, given enough resources

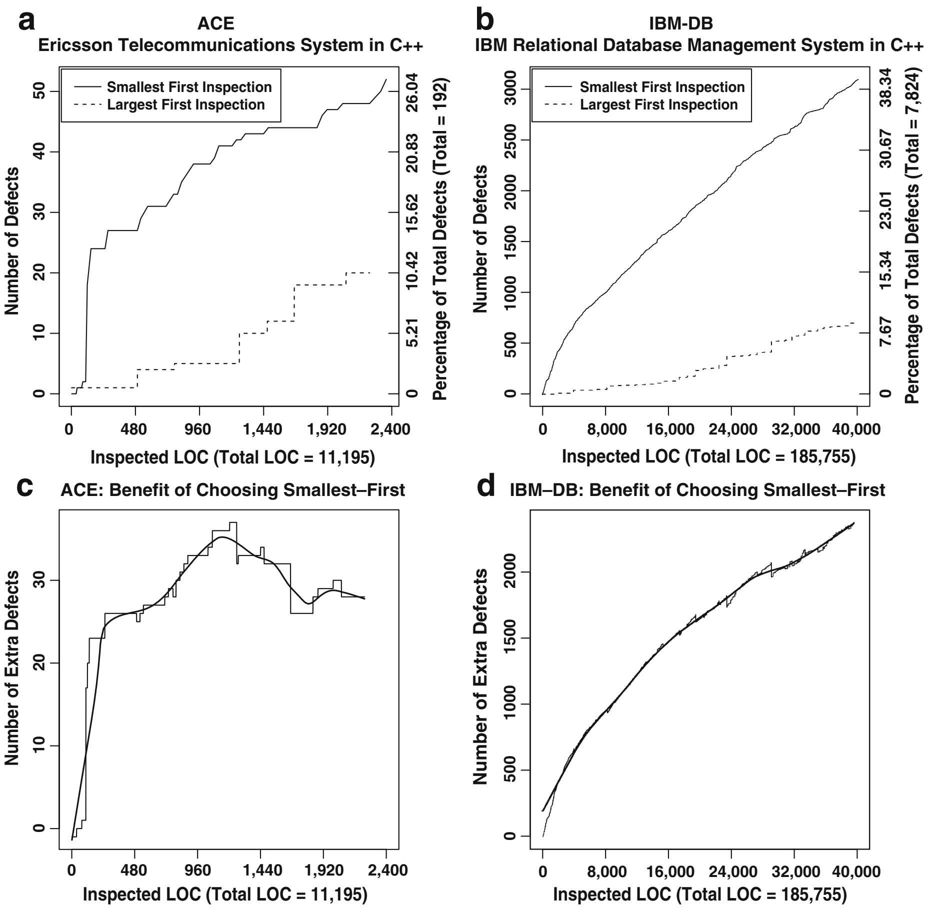

Fig. 4 Comparing the smallest-first and largest-first strategies by using ACE and IBM data. In **a** and **b**, the number and percentage of defects covered by both strategies are plotted against the amount of inspected code. In **c** and **d**, the benefit of the smallest-first strategy is shown in terms of the number of extra defects it covers

to inspect 40,000 LOC, the smallest-first strategy would be 441% as effective as the largest-first strategy with the benefit of covering 2,394 extra defects. Again, the smallest-first strategy would quickly start to pay off from the beginning of the inspections.

To reiterate, in real settings there will be other criteria in prioritizing the testing and inspection efforts. We are not suggesting that the smallest-first approach should be strictly followed. However, the benefits demonstrated in Fig. 4 are huge gains in cost effectiveness. Therefore, practitioners should consider giving a higher priority to smaller modules in their focused quality assurance efforts.

6 Limitations

Our study has a number of limitations, some of which are similar to those typically encountered in the empirical studies of software quality modeling.

Firstly, there might be other factors affecting the defect-proneness of software modules, such as developers' skill and training, amount of testing or peer-review applied before source code commits, and reuse. At this point, we do not have such data to include in our models. However, although one can speculate, there is no evidence that such variables are unevenly distributed over size, which is a measure consistently associated with defect proneness in the literature. Fortunately, the modeling approach used in this study, Cox modeling, is a semi-parametric approach, which quantifies the effect of size on relative defect-proneness when there are some unknown or uncontrollable factors represented by the baseline hazard. It should be also noted that we worked on a large number of products providing thousands of data points. Surely, more complete models that include size among other variables can be constructed and tested in the future research studies if the necessary data become available.

Secondly, the defect fixes were used as an indicator of defect proneness. Traditionally, in the empirical studies of software engineering, defect fixes have been frequently used as an indicator of defect-proneness for software modules, e.g., (Basili and Perricone 1984; Munson and Khoshgoftaar 1992; Troster and Tian 1995) because, in real settings, defects cannot be known in advance. This approach is similar to using patient treatment or hospital visits data to understand the extent of certain diseases in epidemiology. It is possible that some defects will never surface, or some will surface but they will not get fixed, or few defects could be fixed multiple times. These are common and inherent problems in all research studies in the software quality area (El Emam 2005), to which there is no general solution. Still, defect fixes are good indicators of module quality, and they are important because of their effect on the corrective maintenance costs. Also, some developers may not write meaningful CVS logs. Nevertheless, in an earlier study (Koru and Tian 2004), we found that defects were recorded with enough consistency and completeness in the KDE projects (KOffice was initially a part of the KDE desktop, which adopted similar defect handling practices). In this study, we identified defect fixes by automatically parsing the CVS logs because a manual procedure would be infeasible and, potentially, error prone. When we sampled 100 logs randomly to test the effectiveness of our automated solution, the automated approach achieved 97% accuracy with the manual approach used as the baseline for comparison.

Thirdly, the hypothetical scenario presented in the discussion section made several assumptions, among which, the most important one is that the inspection resource spent for a module is linearly related to its size. If the future research studies show otherwise, the presented results should be revised accordingly in terms of cost effectiveness. However, the examples will still serve well in order to highlight the advantages of giving priority to smaller modules in terms of being able to focus on more defect-prone parts of a system, which can potentially reveal and fix more defects improving the overall software quality and reliability.

Given that this study gave very consistent results across different products, we have adequate confidence in their validity.

7 Future Work

At this stage, it is useful to speculate about the underlying reasons for our results. As mentioned in our literature review, Basili and Perricone reached a similar conclusion earlier (Basili and Perricone 1984). Their most plausible explanation of the underlying mechanism was that the interface defects could have been distributed over smaller and larger modules equally. The future research studies can investigate the validity of this claim.

It could also prove useful to study whether there is a difference between smaller and larger modules in terms of the resources and skills assigned to them during their design, coding, review, and testing.

The future research studies building prediction models for software quality can also benefit from our results by favoring smaller modules in their prediction mechanism and by taking the cost-effectiveness issues discussed in this paper into account.

8 Conclusion

The nature of the relationship between module size and defect proneness is important because it can affect various decisions about how to apply focused quality assurance activities.

The evidence obtained from this study using the module-level size and defect data from ten different software products consistently showed that there is a power-law relationship between size and defect proneness with the latter increasing at a slower rate. This evidence clearly supports our hypothesis that smaller modules are proportionally more defect prone. The earlier perception among practitioners and researchers about a linear size-defect relationship could have been caused by the fact that the functional form of this relationship was not investigated in depth earlier, or because, they were more interested in or focused on only certain size ranges in the entire size spectrum (e.g., only smaller modules). We suggest that the practitioners take our findings into account in their testing and inspection prioritizations.

We have adequate confidence in the validity of our results considering that ten different products were analyzed in this replicated study, which confirmed the findings from the Mozilla project presented in the PROMISE 2007 Workshop (Koru et al. 2007). In addition, for two commercial systems, we validated the cost effectiveness of focusing on smaller modules given limited and fixed resources for quality assurance. Therefore, we state a theory of RDP for software modules:

In large-scale software systems, smaller modules will be proportionally more defect prone compared to larger ones.

In addition to presenting interesting and significant results, this study also creates a number of interesting future research directions, especially about investigating the underlying reasons for the observed phenomenon and about building new prediction mechanisms for software quality.

Finally, the KOffice results presented in this paper are verifiable. The raw data used is publicly available in the KOffice project repository. The data sets prepared and used for analysis has been made public by the first author in the PROMISE repository. Our study is also repeatable, and it lends itself to future replicated studies considering the current availability of publicly accessible source code, defect, and data repositories.

Acknowledgements We would like to thank Frank E. Harrell for extending and modifying some of the functionality in his Design package for us, Victor R. Basili for his helpful comments, Jeff Tian for providing data, the associate editor, Tim Menzies, for his guidance and suggestions, and the anonymous reviewers of this paper for their helpful and constructive feedback.

Appendix

In this appendix, we explain how to calculate the RDP of the modules chosen by one inspection strategy with respect to those chosen by another inspection strategy. The first inspection strategy chooses m modules having sizes (in LOC), $s_1, s_2, ..., s_m$, and the second one chooses n modules having sizes, $S_1, S_2, ..., S_n$.

First, let us take a reference module with size C. Since we observed a logarithmic shape for the link function, following (2) and omitting the time parameter t to simplify the notation, the RDP of an individual module of size s with respect to this reference module at any time t would be $e^{\beta(\ln s - lnC)}$. For each inspection strategy, we calculate the sum of the RDP of the selected individual modules with respect to the reference module. To find the RDP, we simply take the ratio of these sums:

$$RDP = \frac{\sum_{i=1}^{m} e^{\beta(\ln s_i - \ln C)}}{\sum_{i=1}^{n} e^{\beta(\ln S_i - \ln C)}} = \frac{\sum_{i=1}^{m} \frac{e^{\beta \ln s_i}}{e^{\beta \ln C}}}{\sum_{i=1}^{n} \frac{e^{\beta \ln S_i}}{e^{\beta \ln C}}} = \frac{\cancel{\frac{1}{e^{\beta \ln C}}} \sum_{i=1}^{m} e^{\beta \ln s_i}}{\cancel{\frac{1}{e^{\beta \ln C}}} \sum_{i=1}^{n} e^{\beta \ln S_i}} = \frac{\sum_{i=1}^{m} e^{\beta \ln s_i}}{\sum_{i=1}^{n} e^{\beta \ln S_i}}. \tag{8}$$

References

Akiyama F (1971) An example of software system debuggings. In: Information processing 71, Proceedings of IFIP congress 71, vol 1. IFIP, Amsterdam, pp 353–359

Andersen PK, Borgan O, Gill RD, Keiding N (1993) Statistical models based on counting processes. Springer, Heidelberg

Askari M, Holt R (2006) Information theoretic evaluation of change prediction models for large-scale software. In: Workshop on mining software repositories, MSR 2006, ICSE workshop, Shanghai, 22–23 May 2006

Basili VR, Perricone BT (1984) Software errors and complexity: an empirical investigation. Commun ACM 27(1):42–52

Briand LC, Basili VR, Hetmanski CJ (1993) Developing interpretable models with optimized set reduction for identifying high-risk software components. IEEE Trans Softw Eng 19(11): 1028–1044

Briand LC, Bunse C, Daly JW (2001) A controlled experiment for evaluating quality guidelines on the maintainability of object-oriented designs. IEEE Trans Softw Eng 27(6):513–530

Briand LC, Melo WL, Wüst J (2002) Assessing the applicability of fault-proneness models across object-oriented software projects. IEEE Trans Softw Eng 28(7):706–720

Chayes F (1971) Ratio correlation: a manual for students of petrology and geochemistry. University of Chicago Press, Chicago

Compton BT, Withrow C (1990) Prediction and control of ada software defects. J Syst Softw 12(3):199–207

Cox DR (1972) Regression models and life tables. J Royal Stat Soc 34:187–220

El Emam K (2005) The ROI from software quality. Auerbach Publications, Taylor and Francis Group, LLC, Boca Raton

El Emam K, Benlarbi S, Goel N, Rai SN (2001) The confounding effect of class size on the validity of object-oriented metrics. IEEE Trans Softw Eng 27(7):630–650

El Emam K, Benlarbi S, Goel N, Melo W, Lounis H, Rai SN (2002) The optimal class size for object-oriented software. IEEE Trans Softw Eng 28(5):494–509

Fenton N, Pfleeger SL (1996) Software metrics: a rigorous and practical approach, 2nd edn. PWS, Boston

Fenton NE, Neil M (1999) A critique of software defect prediction models. IEEE Trans Softw Eng 25(5):675–689

Fenton NE, Ohlsson N (2000) Quantitative analysis of faults and failures in a complex software system. IEEE Trans Softw Eng 26(8):797–814

Funami Y, Halstead MH (1976) A software physics analysis of akiyama's debugging data. In: Proceedings of MRI XXIV international symposium on computer software engineering. IEEE, Piscataway, pp 133–138

Gaffney JE (1984) Estimating the number of faults in code. IEEE Trans Softw Eng 10(4):459–465

Halstead MH (1977) Elements of software science. Elsevier, Amsterdam

Harrell FE (2001) Regression modeling strategies: with applications to linear modes, logistic regression, and survival analysis. Springer, Heidelberg

Harrell FE (2005) Design: design package. R package version 2.0–12. http://biostat.mc.vanderbilt.edu/twiki/bin/view/Main/Design

Harvey AC, Collier P (1977) Testing for functional misspecification in regression analysis. J Econom 6(1):103–119

Hatton L (1997) Reexamining the fault density-component size connection. IEEE Softw 14(2):89–97

Hatton L (1998) Does oo sync with how we think? IEEE Softw 15(3):46–54

Hosmer DW, Lemeshow S (1999) Applied survival analysis: regression modeling of time to event data. Wiley, New York

Khoshgoftaar TM, Allen EB, Hudepohl J, Aud S (1997) Applications of neural networks to software quality modeling of a very large telecommunications system. IEEE Trans Neural Netw 8(4): 902–909

Koru AG, Tian J (2003) An empirical comparison and characterization of high defect and high complexity modules. J Syst Softw 67(3):153–163

Koru AG, Tian J (2004) Defect handling in medium and large open source projects. Softw IEEE 21(4):54–61

Koru AG, Ma L, Li Z (2003) Utilizing operational profile in refactoring large scale legacy systems. In: WCRE 2003: first IEEE international workshop on refactoring: achievements, challenges, effects, Victoria, November 2003

Koru AG, Zhang D, Liu, H (2007) Modeling the effect of size on defect proneness for open-source software. In: Predictor models in software engineering, PROMISE'07, 20–26 May 2007

Lipow M (1982) Number of faults per line of code. IEEE Trans Softw Eng 8(4):437–439

McCabe TJ (1976) A complexity measure. IEEE Trans Softw Eng 2(6):308–320

Meine JPvdM, Miguel AR (2007) Correlations between internal software metrics and software dependability in a large population of small c/c++ programs. In: The 18th IEEE international symposium on software reliability. IEEE, Trollhattan, pp 203–208
Mockus A, Fielding RT, Herbsleb J (2002) Two case studies of open source software development: apache and mozilla. ACM Trans Softw Eng Methodol 11(3):309–346
Munson JC, Khoshgoftaar TM (1992) The detection of fault-prone programs. IEEE Trans Softw Eng 18(5):423–433
Newman MEJ (2005) Power laws, pareto distributions and zipf's law. Contemp Phys 46:323
Ostrand TJ, Weyuker EJ, Bell RM (2005) Predicting the location and number of faults in large software systems. IEEE Trans Softw Eng 31(4):340–355
Promise (2007) Promise data repository
R Development Core Team (2003) R: a language and environment for statistical computing. ISBN 3-900051-00-3
Raymond ES (1999) The Cathedral and the Bazaar: musings on Linux and open source by an accidental revolutionary. O'Reilly, Sebastopol
Rosenberg J (1997) Some misconceptions about lines of code. In: METRICS '97: Proceedings of the 4th international symposium on software metrics. IEEE Computer Society, Washington, DC, pp 137–142
Schmidt DC (1995) Using design patterns to develop reusable object-oriented communication software. Commun ACM 38(10):65–74
Scientific Toolworks I (2003) Understand for c++: user guide and reference manual, January. I Scientific Toolworks, St. George
Shen VY, Yu TJ, Thebaut SM, Paulsen L (1985) Identifying error-prone software - an empirical study. IEEE Trans Softw Eng 11(4):317–324
Therneau TM (1999) Survival: survival analysis package, including penalized likelihood. R package v. 2.29. http://cran.r-project.org/web/packages/survival/index.html
Therneau TM, Grambsch PM (2000) Modeling survival data: extending the Cox model. Springer, Heidelberg
Tian J, Troster J (1998) A comparison of measurement and defect characteristics of new and legacy software systems. J Syst Softw 44(2):135–146
Troster J, Tian J (1995) Defect characteristics of legacy software: measurement, visualization, regression analysis, and tree-based modeling. Technical report, IBM SWS Toronto Laboratory, March
Withrow C (1990) Error density and size in ada software. IEEE Softw 7(1):26–30

A. Güneş Koru received a B.S. degree in Computer Engineering from Ege University, İzmir, Turkey in 1996, an M.S. degree in Computer Engineering from Dokuz Eylül University, İzmir, Turkey in 1998, an M.S. degree in Software Engineering from Southern Methodist University (SMU), Dallas, TX in 2002, and a Ph.D. degree in Computer Science from SMU in 2004. He is an assistant professor in the Department of Information Systems at University of Maryland, Baltimore County (UMBC). His research interests include software quality, measurement, maintenance, and evolution, open source software, bioinformatics, and healthcare informatics.

Khaled El Emam is an Associate Professor at the University of Ottawa, Faculty of Medicine and the School of Information Technology and Engineering. He is a Canada Research Chair in Electronic Health Information at the University of Ottawa. Previously Khaled was a Senior Research Officer at the National Research Council of Canada, and prior to that he was head of the Quantitative Methods Group at the Fraunhofer Institute in Kaiserslautern, Germany. In 2003 and 2004, he was ranked as the top systems and software engineering scholar worldwide by the Journal of Systems and Software based on his research on measurement and quality evaluation and improvement, and ranked second in 2002 and 2005. He holds a Ph.D. from the Department of Electrical and Electronics, King's College, at the University of London (UK). His labs web site is: http://www.ehealthinformation.ca/.

Dongsong Zhang is an Associate Professor in the Department of Information Systems at University of Maryland, Baltimore County. He received his Ph.D. in Management Information Systems from the University of Arizona. His current research interests include context-aware mobile computing, computer-mediated collaboration and communication, knowledge management, and open source software. Dr. Zhang's work has been published or will appear in journals such as Communications of the ACM (CACM), Journal of Management Information Systems (JMIS), IEEE Transactions on Knowledge and Data Engineering (TKDE), IEEE Transactions on Multimedia, IEEE Transactions on Systems, Man, and Cybernetics, IEEE Transactions on Professional Communication, among others. He has received research grants and awards from NIH, Google Inc., and Chinese Academy of Sciences. He also serves as senior editor or editorial board member of a number of journals.

Hongfang Liu is currently an Assistant Professor in Department of Biostatistics, Bioinformatics, and Biomathematics (DBBB) of Georgetown University. She has been working in the field of Biomedical Informatics for more than 10 years. Her expertise in clinical informatics includes clinical information system, controlled medical vocabulary, and medical language processing. Her expertise in bioinformatics includes microarray data analysis, biomedical entity nomenclature, molecular biology database curation, ontology, and biological text mining. She received a B.S. degree in Applied Mathematics and Statistics from University of Science and Technology of China in 1994, a M.S. degree in Computer Science from Fordham University in 1998, a PhD degree in computer science at the Graduate School of City University of New York in 2002.

Divya Mathew received the BTech degree in computer science and engineering from Cochin University of Science and Technology in 2005 and the MS degree in information systems from the University of Maryland, Baltimore County in 2008. Her research interests include software engineering and privacy preserving data mining techniques.