An Investigation into the Functional Form of the Size-Defect Relationship for Software Modules

A. Güneş Koru, Dongsong Zhang, Khaled El Emam, and Hongfang Liu

Abstract—The importance of the relationship between the size and defect proneness of software modules is well recognized. Understanding the nature of that relationship can facilitate various development decisions related to prioritization of quality assurance activities. Overall, the previous research only drew a general conclusion that there was a monotonically increasing relationship between module size and defect proneness. In this study, we analyzed class-level size and defect data in order to increase our understanding of this crucial relationship. We studied four large-scale object-oriented products, Mozilla, Cn3d, JBoss, and Eclipse. We observed that defect proneness increased as class size increased, but at a slower rate; smaller classes were *proportionally* more problematic than larger classes. Therefore, practitioners should consider giving higher priority to smaller modules when planning focused quality assurance activities with limited resources. For example, in Mozilla and Eclipse, an inspection strategy investing 80 percent of available resources on 100-LOC classes and the rest on 1,000-LOC classes would be more than twice as cost-effective as the opposite strategy. These results should be immediately useful to guide focused quality-assurance activities in large-scale software projects.

Index Terms—Product metrics, software science, size-defect relationship, measurement applied to SQA and V&V, planning for SQA and V&V, open-source software.

1 Introduction

Software quality is becoming increasingly important as our reliance on software is ever increasing. Software defects usually detract from software quality because they cause failures [59]. A software failure occurs when a service delivered by software deviates from fulfilling its intended functionality; error is the discrepancy between the delivered and intended functionality; and the adjudged or hypothesized cause of an error is a fault [37], which is commonly known as a defect (or as a bug) among developers [15].

Size is arguably one of the most important measures in software engineering. It is popularly believed that software size is positively related to defect proneness (i.e., the larger the software modules or systems, the more defects there should be). Size measures are often used to adjust defect counts when evaluating quality, as in measures of defect density. Furthermore, due to its confounding effect [17], size needs to be used as a covariate in models that predict quality using static and dynamic metrics. Indeed, some of those metrics were found to be highly correlated with size [19], [20], limiting their additional benefits in defect prediction models. Therefore, our focus in this paper is on the *size-defect relationship* for software modules.

In order to be able to use size effectively and efficiently in software quality improvement, we need to understand the functional form of the size-defect relationship. If this relationship is linear, smaller and larger modules should include an equal number of defects per line of code (LOC). Therefore, in terms of cost-effectiveness, size should not play a significant role in deciding which groups of modules should be given a higher priority in testing and inspections. However, a different functional form could affect the prioritization strategy given that the inspection and testing resources are often limited. For example, a quadratic form implies that larger modules are proportionally more troublesome and they should receive a higher priority, whereas a logarithmic form implies that smaller modules are proportionally more troublesome and should receive a higher priority.

Despite its significance [15], the nature of the size-defect relationship has not been sufficiently understood yet [16], [20] to provide practitioners with empirically validated and generalizable guidance for detecting defects in their software development and maintenance efforts. In this study, we empirically investigate the functional form of the size-defect relationship by using class-level size and defect data collected from four long-lived object-oriented products, namely Mozilla [43], Cn3d [10], JBoss [30], and Eclipse [14].

Findings about the nature of this fundamental relationship can be more generalizable to different environments as opposed to the complex and overfitted quality prediction models. Consequently, the results can provide a theoretical foundation and empirical guidance to help software

- *A.G. Koru and D. Zhang are with the Department of Information Systems, University of Maryland Baltimore County, 1000 Hilltop Circle, Baltimore, MD 21250. E-mail: {gkoru, zhangd}@umbc.edu.*
- *K. El Emam is with the Faculty of Medicine and the School of Information Technology and Engineering, University of Ottawa, and with the Children's Hospital of Eastern Ontario Research Institute, 401 Smyth Road, Ottawa, Ontario, Canada K1H 8L1. E-mail: kelemam@uottawa.edu.*
- *H. Liu is with the Department of Biostatistics, Bioinformatics, and Biomathematics, Georgetown University, School of Medicine, Suite 180, Building D, 4000 Reservoir Rd. NW, Washington, DC 20057-1484. E-mail: hl224@georgetown.edu.*

Manuscript received 24 Dec. 2007; revised 10 July 2008; accepted 16 Sept. 2008; published online 24 Nov. 2008.
Recommended for acceptance by R. Jeffery.
For information on obtaining reprints of this article, please send e-mail to: tse@computer.org, and reference IEEECS Log Number TSE-2007-12-0354.
Digital Object Identifier no. 10.1109/TSE.2008.90.

practitioners develop more effective and efficient quality assurance and control strategies.

The rest of this paper is organized as follows: We start by introducing the related work on size-defect relationship in Section 2. Then, we describe the research methodology and data collection and analysis methods used in this study in Section 3. Section 4 presents our modeling results. Section 5 discusses the implications of our results by showing the cost-effectiveness of our recommendations in a realistic scenario. We also discuss the potential threats to the validity of our findings in Section 6. Finally, we conclude this paper in Section 7.

2 Related Research

In this section, we provide a comprehensive literature review by categorizing different types of related work and findings. In the previous studies, some researchers explored the size-defect relationship, while others studied size versus defect density. Some methodological problems have been identified with the latter [16].

In this literature review, we also include the studies that associated Halstead's Science metrics [24] and McCabe's metrics [39] with defects because, at the current state of our field, there is a general recognition that those metrics serve as surrogate size measures due to their high correlation with size [19], [20].

2.1 Linear Relationship between Size and Defects

Akiyama [1] built a linear univariate model for size-defect relationship by using the data collected from an assembly-language program with nine modules. It was the first attempt to model this relationship. The data for those modules were published and used by other researchers later (e.g., [22]). However, the functional form of the relationship was not explored and the small number of data points was a limitation.

By using the data published by Akiyama [1], Funami and Halstead [21] provided estimates of the Halstead's software science measures [24] and observed a very high correlation between the obtained measures and defect count. Based on this high correlation, they concluded that there was a linear relationship between the estimated measures and defects. The conclusion of linearity based on high correlation is problematic because a linear relationship also requires proportionality between size and defects. In addition, as pointed out by Hamer and Frewin [25], there were errors in the data reported by Funami and Halstead [21].

Schneidewind and Hoffman [50] examined the relationship between the number of source code statements and defects, and observed high correlations. The correlation between size and defects was higher for the linear model, which somewhat surprised the authors, who expected nonlinearity. The authors did not report the statistical significance of the correlations and stated that the correlations implied some degree of linearity.

Ottenstein [45] built a model in which Halstead's volume metric was linearly related with defects. The model was evaluated with four different data sets including Akiyama's [1] and Thayer et al.'s [55] data sets and achieved reasonable success. Ottenstein compared different models involving various Halstead measures and found that the linear model had the best fit. However, Ottenstein did not investigate the functional form of the relationship.

Shen et al. [53] developed various linear regression models for Halstead's measures and defects. The authors stated that they only developed linear regression models because of better tool support and linear models would serve as satisfactory approximations to the true relationship. Therefore, no other alternative functional forms were explored.

Gyimothy et al. [23] collected size and object-oriented metrics data from the Mozilla project. The main objective of their study was to validate object-oriented metrics but they also fitted a linear regression model for size and defects. The obtained R^2 value was 0.34. The functional form of the size-defect relationship was not explored any further.

2.2 Nonlinear Relationship between Size and Defects

In contrast, other studies have suggested a nonlinear relationship between module size and defects. Based on the analysis of data collected from a NASA Goddard project along with Withrows data, Hatton [27], [28] suggested two different models. For sizes up to 200 lines, Hatton stated that the total number of defects grew logarithmically with module size. For larger modules, a quadratic model was suggested.

Gaffney [22] used the same data set as Lipow [38] and Akiyama [1] to develop a nonlinear model in which defects were proportional to size raised to the power of 4/3. Gaffney justified the model by some analytical derivations from Halstead's equations [24] first and then supported it empirically. The supportive statistics in [22] came from a comparison of six equations that included various Halstead measures either raised to the power of 2/3 or to the power of 4/3. Other functional forms were not explored and there was no evidence showing that deviation from linearity was statistically significant.

2.3 Relationship between Size and Defect Density

A larger number of studies (e.g., [6], [27], [28], [42], [52], [53]) have concentrated on how defect density changed with size.

Lipow [38] derived a model analytically from Halstead's equations [24]. In his study, defect density was predicted from a logarithmic transformation of size using the data published by Thayer et al. [55]. That approach suffers from ratio correlations [9] caused by using size and its inverse, defect density, in the same model. Lipow also calculated an optimal module size that minimized defect density.

Basili and Perricone [6] also examined the size-defect density relationship. They observed that defect density was lower in larger modules, and concluded that the reason could be that a large number of interface defects were spread equally over smaller and larger modules. Moller and Paulish [42] performed a similar analysis and also observed that defect density was significantly higher in smaller modules. Compton and Withrow [11] also suggested that there be an optimal module size that would minimize the defect density. Similar observations were later commonly called Goldilock's conjecture, which appeared as a U-shaped curve on a plot showing size (on the x-axis) against defect density (on the

y-axis). Hatton [27], [28] also observed a U-shaped curve and advised developers to produce modules of intermediate size, not too small, not too big.

El Emam et al. [16] conducted an extensive survey of similar studies and reported that there were also studies that validated only the left (e.g., [6], [42], [52], [53]) or only the right half of the U-shaped curve [8]. They argued that the main reason behind the U-shape observations was a methodological artifact, namely, plotting size against its reciprocal. The authors stated that studying defect density versus size causes ratio correlations [9], masks the true nature of the relationship between size and defects, and misleads us by showing some threshold values and optimal module sizes. Similar concerns about studying size versus defect density were actually raised earlier by Fenton and Neil [19] and even earlier by Rosenberg [48].

2.4 No Relationship between Size and Defects

Some researchers examined size-defect plots and did not see any relationship between the two.

Indeed, Basili and Perricone [6] plotted size-defect density plots when they could not visualize a relationship in the plain size-defect plots. Fenton and Ohlsson [20] also observed no relationship between size and defects in their study of a system at Ericsson. They stated that the only general conclusion that could be drawn from the previous studies in the field was that defect count would generally increase with size. It was stressed that a linear relationship between size and defect count was not always guaranteed. In Fenton and Ohlsson's study [20], the authors only presented scatter plots to support their arguments; they did not try to find alternative functional forms using special techniques as done in this study.

Recently, Andersson and Runeson [4] replicated the Fenton and Ohlsson's study [20] using size and defect data collected from three software projects and found that there was some evidence showing that size and defects were correlated; however, the evidence collected from all three projects was mixed. Andersson and Runeson reported only R^2 values and no alternative functional forms were explored.

2.5 Data Mining and Machine Learning Techniques

Finally, there are also a number of studies that employed various data mining and machine learning techniques to predict defect-prone modules, such as neural networks [31], principal component analysis [32], tree-based models [46], and optimized set reduction OSR [7]. These approaches do not assume a functional form for the size-defect relationship, which is preferable, and try to detect patterns for defect proneness.

However, it is often difficult to understand the nature of the plain size-defect relationship from data mining and machine learning models. Understanding this main relationship is important because its nature can be generalizable to different environments, which is one of the important needs in this research area [19], [58].

The main contribution of this study is to bridge this knowledge gap by empirically examining the functional form of the essential size-defect relationship with the rich data obtained from different large-scale software products.

3 Methods

This section introduces the empirical research methods used in this study. Our main objective was to measure class size in LOC (excluding blank and comment lines) and relate the size measurements to defect proneness of classes.

In this study, the assessment of defect proneness was achieved by looking at the corrective changes (i.e., defect fixes) in the histories of version control systems. This approach has been commonly used in the empirical studies of software quality as further explained in Appendix A. It is suitable for the long-lived open-source products examined in this study, which were used by a large community of open-source users and developers who reported problems, detected defects, and fixed them.

In the rest of this section, we first start by discussing why the use of a conventional data analysis method could create internal validity problems in this study. Then, we introduce a novel data analysis method that applies Cox proportional hazards modeling for recurrent events, which is appropriate for analyzing the data obtained from open-source products. After that, we will describe the size and defect data used in the study and discuss our data collection method.

3.1 Concerns with Using a Conventional Analysis Approach

Conventionally, in the empirical studies of software quality, software modules in a certain system snapshot are measured, and the future defect counts for those modules are obtained by observing the modules for a time period (e.g., [18], [31], [33])—preferably, for a time period long enough for defects to be revealed. Then, the conventional approach uses the measurements to characterize and predict defect-prone modules. For this approach to produce results with high validity, the observed phenomena (i.e., defect fixes) should occur after the time of measurement.

The conventional approach can be appropriate for a plan-driven project where planning, analysis, and design phases are designated, and the outputs from those phases guide the implementation phase. In that case, the source code could be measured at a snapshot, e.g., right after implementation, and the future defects could be counted. On the other hand, open-source products are usually developed in an evolutionary manner by continuously changing, adding, and removing modules [5], [40], [47]. Defects are detected and fixed in a continuous manner too [34]. In this case, there are two main concerns with using the conventional approach:

1. **Deleted modules**. Deleted modules are problematic because their defect counts could be smaller than nondeleted ones. They could be excluded from data analysis but such an approach would reduce the number of data points.
2. **Size changes**. Ignoring size changes can bias the analysis results in unpredictable ways. For example, a module measured when it was small can become larger and more defect prone over time. However, its measurement values in the data set will remain small.

Fig. 1 presents empirical evidence for the existence of these concerns. Fig. 1a shows the cumulative number of

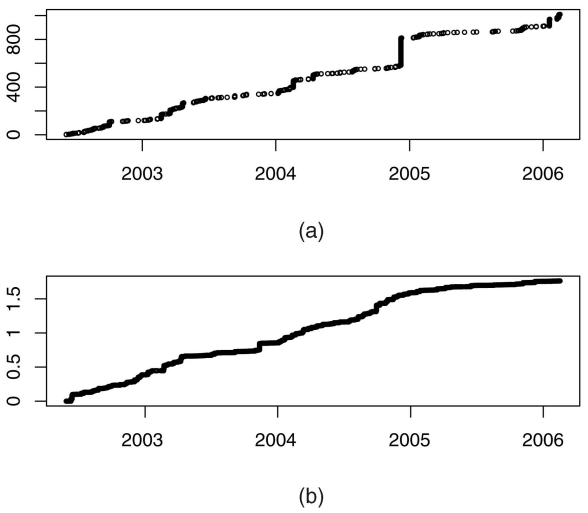

Fig. 1. Evidence for concerns with using a conventional approach for Mozilla 1.0. (a) The cumulative number of deleted Mozilla 1.0 classes over time. (b) The cumulative code churn for Mozilla 1.0 classes.

deleted Mozilla 1.0 classes over time (a total of 1,010 deleted classes). Fig. 1b shows the code churn for Mozilla 1.0 classes over time. Code churn is the sum of added and deleted code lines (excluding those only blank or comments). Both plots show a steady increase over time. The average code churn for Mozilla classes was 90.75 LOC. Therefore, the size of a class at the time of measurement can be quite different from its size when its defects get revealed and fixed.

In order to avoid the potential internal validity threats, we used Cox proportional hazards modeling for recurrent events (Cox modeling, henceforth), which is explained next.

3.2 Cox Modeling

Cox proportional hazards modeling (henceforth *Cox modeling*) is one of the most commonly used techniques to create time-to-event models [29], [56]. This technique is associated with the counting process and Martingale theory [2], [3], which makes it suitable for the analysis of recurrent events. Recurrent events, also called repeated measurements, multiple events, or recurring events, refer to the events that recur at intervals [29].

Using Cox modeling [13], the *complete size* histories of software classes can be related to the instantaneous risk of having a defect fix, which is an indicator of defect proneness. At any moment, all subjects that are followed constitute a *risk set*. The size history of a class consists of a set of observations obtained by measuring its size right after it is created and then right after each subsequent modification made to it.

In this study, each defect fix made to a class is considered an *event*. There is a single time-dependent covariate, *size*, denoted by $x(t)$. We specify the hazard function, which is the instantaneous risk of an event for a class i at time t, as follows:

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

β is the regression coefficient for $x_i(t)$ and $\lambda_0(t)$ is an *unspecified* nonnegative function of time called the *baseline hazard function*. It is the instantaneous hazard of having an event without any covariate effect (i.e., when $\beta = 0$).

Cox modeling is semiparametric because its baseline hazard function is not described. The time-to-event values are used in a comparative sense and the emphasis is on understanding the effect of the covariate (size) differences on the instantaneous *relative risk* of having an event. This relative risk is also called hazard ratio. Cox modeling assumes proportionality meaning that the hazard ratio for two subjects should depend on the differences in their covariate values. If one writes the hazard for two subjects, classes j and k, at a time t, using (1) and calculates their ratio, the baseline hazard will be eliminated and the instantaneous relative risk will be

$$\lambda_j(t)/\lambda_k(t) = e^{\beta\left(x_j(t)-x_k(t)\right)}. \quad (2)$$

Note that β should be constant over time. Otherwise, the hazard functions for two classes will not be proportional. We checked the validity of the proportional hazards assumption in our models, as discussed in Section 4.2.

Equation (2) requires that the hazard ratios of any class pairs with the same size difference be the same. In practice, this may or may not hold true when a covariate is used in its original form; therefore, looking for a transformation of the covariate might become necessary [56]. In our case, we need to understand whether module size can be directly used in our models and, if not, what kind of transformation of size should be used. This transformation function is also called link function.

Taking the natural log of the both sides of (1), one could obtain

$$\ln \lambda_i(t) = \ln \lambda_0(t) + \beta x_i(t). \quad (3)$$

As shown in (3), $x_i(t)$ should be *linearly* related to log-hazard. If it is not and if there is a link function, $f(x_i(t))$, that is linearly related to the log-hazard, we want to find that link function and use it instead of $x_i(t)$. For this purpose, we relaxed the linearity assumption by fitting restricted cubic spline functions [26] and by visually examining the emerging overall shape. This approach identifies a number of knots on the entire size range (x-axis) and fits a cubic function of $x_i(t)$ for each subrange. Using cubic splines to understand the shape of the link function in Cox models is appropriate when there are multiple observations per subject [57].

The Cox models built in this study are *conditional Cox models*. A nice characteristic of any Cox model is that it can incorporate observation categories (also called strata) and use a different baseline hazard for each category. For example, in Mozilla, a class (subject) starts in the nondefective state (State 0). It cannot enter the defective state (State 1) and be at risk in that state before experiencing an event. Each stratum corresponds to a different risk set. This stratification is useful in preserving proportionality in Cox models. Note that, still, a single coefficient estimate ($\hat{\beta}$) is produced for size, which applies to all baseline hazards.

To summarize, the ability to model recurrent events and the ability to take the changing characteristics of software modules into account make Cox modeling an appropriate choice for statistical modeling for open-source projects.

TABLE 1
Summary of Data Used for Cox Modeling

Product	Language	Number of Classes	Number of Observations	Size data over all observations						Number of Events
				Min	25^{th} percentile	Median	Mean	75^{th} percentile	Max.	
Mozilla	C++	4,089	15,545	1	36	126	434.0	418	12,360	8,828
Cn3d	C++	136	2,134	5	108	251	341.6	509	1,473	339
Eclipse	Java	1,134	11,903	1	35	91	211.2	253	1,507	4,191
JBoss	Java	1,703	9,428	1	42	102	173.0	242	1,479	1,265

3.3 Data Collection and Description

In this section, we first discuss our data collection procedures. Then, we describe the size and defect data used in our study.

3.3.1 Data Collection

We used our own Perl scripts to extract data from the CVS repositories of the four open-source products chosen for this study, namely Mozilla, Cn3d, JBoss, and Eclipse. For each project, we extracted class-level size and defect data for a long time period called *observation period*.

For each CVS commit, our programs detected the changed files, the changed lines in those files, and then checked which existing classes were affected by those changes. Each change made to a class created an observation for Cox modeling. When drawing the boundaries of a class, we took into account that a class could spread over header and implementation files (for C++). During this process, we also kept track of the boundary changes during successive modifications of a class,and made sure that the most recent boundary information was used at all times. For each CVS commit, our programs updated the project database kept by a reverse engineering tool, Understand [51], and extracted the size values of the changed classes. By following the above procedure, we obtained the size measurements for the entire change history of every single class added during our observation period. To our best knowledge, no prior study in the literature obtained and used such detailed measurement and defect data from large-scale open-source projects.

The defect data, called event data in Cox modeling, were obtained by automatically parsing the log portions of CVS commits with the regular expressions made using the keywords "defect," "fix," and "bug" in a non-case-sensitive manner. Once a CVS commit was classified as a defect fix, the affected classes were identified together with their most recent observations. The event field of those observations was set to 1. This automated approach was preferred because of the large number of CVS commits that took place during our long observation periods. For example, in Mozilla, there were 10,459 CVS commits. To examine the effectiveness of this approach, we randomly sampled 100 CVS logs. We went through them manually and classified them as either corrective (defect fix) or non-corrective changes (not defect fix). Then, we compared our manual classification results with those of the automated approach. This comparison showed that the classification accuracy of the automated approach was 98 percent.

The first examined product, Mozilla, is a Web browser. Some of the previous research studies [23], [35], [41] also extracted and analyzed various structural measures and defect data from this project. We extracted our own data from the Mozilla project for the time period between 30 May 2002, right after the release date of Mozilla 1.0, and 22 February 2006, the date of our data collection. During this observation period, 4,089 classes were added, for which we obtained the complete size measurement history. At the end, we created a data set in a certain format (see Section 3.3.2) to be used for Cox modeling. There were a total of 15,545 observations that belonged to 4,089 classes created after the Mozilla 1.0 release.

Since Mozilla was a large-scale product, we picked Cn3D, which is a smaller product. Cn3d is a bioinformatics application for Web browsers written in C++. It visualizes 3D structures from the National Center for Biotechnology Information's (NCBI) *Entrez* retrieval service. Its data set included 2,134 observations that belonged to 136 classes observed between 27 June 2000 and 26 September 2005. Unlike Mozilla, which is a general-purpose product, Cn3d is a domain-specific product.

We also wanted to work on the products written in another programming language. Since Java is another popular object-oriented language, we chose Eclipse and JBoss that are almost entirely written in Java. Both are large-scale and long-lived software products providing a large number of data points. Eclipse is a Java Integrated Development Environment (IDE) that includes a large suite of subproducts; therefore, we decided to work on its core component that provides the basic Eclipse platform structure. The Eclipses core component had 1,134 Java classes and 11,903 observations made between 2 May 2001 and 9 November 2007.

JBoss is one of the most commonly used Java application servers. Our JBoss data set included 9,428 observations that belonged to 1,703 Java classes. JBoss observations came from the time period between 4 May 2000 and 27 July 2006. The ending date was the time the project changed its source code control repository from CVS [12] to Subversion [54].

The overall summary of the size and event data collected from the products studied can be seen in Table 1.

3.3.2 Data Format Description

The data format used in our study is shown in Table 2, which presents hypothetical observations for demonstration purpose. We created a distinct data set with this layout for each one of our four products.

The subjects Y and Z in Table 1 represent new classes added during an observation period. The events of interest here are defect fixes. Each new class introduced to the system during our observation period is followed till the observation period ends or the class gets deleted. Modifications made

TABLE 2
Data Layout Used in the Study (with Hypothetical Data)

class name	start	end	event	size	state
Y	0	50	0	75	0
Y	50	100	1	200	0
Y	100	200	0	300	1
Z	0	200	1	250	0
Z	200	800	0	180	1
Z	800	1400	1	400	1
Z	1400	1800	0	300	1
.	.	.	.	.	
.	.	.	.	.	

during the follow-up time are entered as observations, which correspond to the rows in Table 2. Each modification creates a new observation with a (start, end] time interval, where start is a time infinitesimally greater than the modification time and end is either the time of the next modification or the end of the observation period or the time of deletion, whichever came first.

Open and closed brackets enclosing *start* and *end* time enable us to model the consecutive observations in a nonoverlapping manner. For example, at $t = 100$, the second row in Table 2 would be used in the internal computations of the Cox model for class Y, not the third one.

It is worth distinguishing between calendar time and analysis (also called study) time. The *start* and *end* values shown in Table 2 are analysis times. Classes were usually integrated into a system at different calendar times. However, for analysis purposes, those different integration times were all considered time *zero*. As a result, the end times of the last observations were usually different too. From an analysis viewpoint, they corresponded to the time when follow up ended. The *start* and *end* times were computed in minutes based on the time tags of the CVS commits.

To create data sets in this format, when a class was introduced into a system, a new observation was entered with $start = 0$. The event attribute was set to 1 if an event (i.e., defect fix) took place at the time represented by *end* or zero otherwise. With this approach, the class deletions were taken into account easily by setting the *end* time of the last observation to the time of deletion and by setting the event attribute of the last observation to 1 if the class was deleted as a part of defect fix or 0 otherwise. *size* is a time-dependent covariate and its column shows the LOC values of the classes at time *start*. *size* might change in successive observations or remain constant. The state column in Table 2 was used to create a stratified model (see Section 3.2). For any class, state was initially set to 0, which became 1 after the class experienced an event.

Let us use the hypothetical data presented in Table 2 as an example. Class Y has 75 LOC when it is first introduced. It is modified at time 50. However, this change is not a corrective one, which is indicated by a zero in the *event* column of the first observation. With the change at time 50, the size of Y becomes 200 LOC and a new observation for this class starts. A corrective change happens at time 100, represented by a 1 in the corresponding cell of the event column. This corrective change increases *size* to 300 LOC. Then, one of the two things might happen at time 200: Either our observation period ends or the class is deleted without an event. The history of Class Z can be interpreted in a similar manner.

4 Modeling and Results

In this section, we discuss the details of our modeling and results. We first present and explain the Cox models. After that, we discuss the diagnostic tests applied to the models in order to assess their adequacy. Finally, we explain what the modeling results mean in terms of size-defect relationship.

4.1 Building Cox Models

First, for each product, we examined the link function by using restricted cubic spline functions. The cubic splines provided a plot for the relationship between size and the instantaneous relative log-hazard of defect fix. Note that the link function is investigated to understand which transformation of size should be used in Cox models (see Section 3.2). The form of the relationship between size and defect proneness is discussed later in Section 4.3 after obtaining the coefficients from Cox models.

Fig. 2 shows the plots generated for all products for the link functions. Note that the plotting function omits the highest 10 and lowest 10 values to avoid a potential bias for the purpose of presenting a more accurate form. The dashed lines represent the 95 percent confidence intervals. Overall, the plots consistently showed a logarithmic form for the link function of size across different products. Surely, the F-statistic obtained when testing the deviation from linearity was highly significant $(p < .0001)$ for all products, supporting that the link function was not linear. Hence, we used the natural logarithm as the link function in our models.

The generated Cox models obtained for Mozilla, Cn3d, Eclipse, and JBoss are shown in Table 3. All Cox models show that size is highly significant with a very large z statistic and a very small or zero p value when entered using log transformation. For example, in the Mozilla project, the coefficient estimate, $\hat{\beta}$, for $\ln(size)$ was 0.368 and its standard error estimate was 0.00732. The robust sandwich estimate of the standard error, which took the intrasubject correlation into account, was 0.018. Both of those standard error estimates were quite small; therefore, we safely accepted $\hat{\beta} = 0.368$.

Using the Mozilla model, the effect of size can be interpreted as follows:

> *For Mozilla classes, one unit of increase in the natural log of size caused the rate of defect fix to be multiplied by 144 percent.*

For the other products, Cn3d, Eclipse, and JBoss, their modeling results in Table 3 can be interpreted in a similar manner. For all four products, the rate of defect fix is multiplied by the number given in the third column $(e^{\hat{\beta}})$ of Table 3 with one unit of increase in the natural log of size.

4.2 Model Diagnostics

Since R^2 is not an appropriate measure for Cox models because of censored observations and nonnormal distribution of residuals [29], [49], there are other fitness checks for Cox models. The first step is to check whether a Cox model satisfies the proportional hazards assumption. A

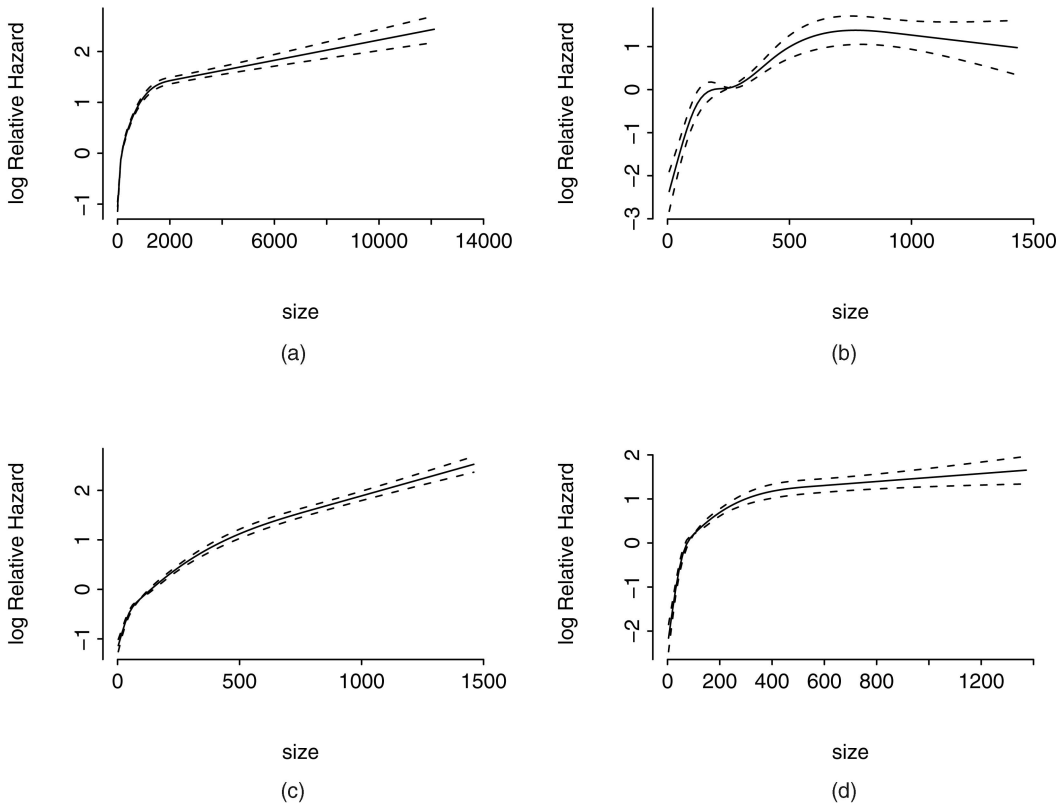

Fig. 2. The link functions for all products fitted using cubic spline functions. (a) Mozilla. (b) Cn3d. (c) Eclipse. (d) Jboss.

TABLE 3
Cox Models Obtained for $\ln(size)$ (Each Line Corresponds to a Model Built for One of the Products)

Product	$\hat{\beta}$	$e^{\hat{\beta}}$	Standard Error (SE) of $\hat{\beta}$	Robust SE of $\hat{\beta}$	Significance of $\ln(size)$		Significance of non-proportionality (p)	Spearman's ρ between expected and actual number of events
					z	p		
Mozilla	0.368	1.44	0.007	0.018	20.4	0	0.84	0.77
Cn3d	0.755	2.13	0.058	0.096	7.86	3.7e-15	0.07	0.86
Eclipse	0.287	1.33	0.015	0.026	10.9	0	0.38	0.79
JBoss	0.623	1.86	0.027	0.040	15.6	0	0.73	0.91

covariate-time interaction is one of the common reasons for nonproportionality [56] (e.g., similar to a drug being effective only in the first hour). Using a wrong link function causes nonproportionality in Cox models too [56].

As seen in Table 3, the result of the test for nonproportionality was statistically insignificant for all products ($p > 0.05$). Therefore, the proportionality assumption is satisfied; we can accept a primarily linear relationship between $\ln(size)$ and log-hazard.

We also examined whether there were overly influential observations (outliers) in our data sets. The purpose of this step was to see whether those observations were valid and to examine what would happen if we were to build another model after removing those observations. For each model, we plotted the dfbeta residuals [56] for size against the observations in the respective data set. An example can be seen in Fig. 3 for Mozilla, where the residuals were generally close to each other; and, we found two overly influential points. Neither outlier point was an erroneous entry. When we constructed another model excluding those data points, we obtained $\beta = \hat{0}.372$, which was only slightly higher than $\hat{\beta}$ of the original model, resulting in $e^{\hat{\beta}} = 1.45$. For all four products, the influential points were valid data points. Also considering that we had a very large number of observations in our data sets, we decided to keep our original data sets and the original models.

In order to check the overall adequacy of our models, for each model, we first obtained the number of estimated events for the classes with noncensored observations. The detailed information about the technique used for this purpose can be seen in [36]. Then, we examined the Spearman's correlation values between the actual and expected number of events for those classes. This strategy was suggested by Schemper and Stare [49]. Spearman's correlation is appropriate to use due to its semiparametric nature. The correlation values obtained were high as shown in the last column of Table 3. Therefore, we concluded that the models developed were adequate.

4.3 Findings about Size-Defect Relationship

We noted that the proportional hazards assumption is satisfied when the logarithmic transform of size is used. This means that there is no statistical evidence for the nonlinearity of the log-size and log-hazard relationship. As

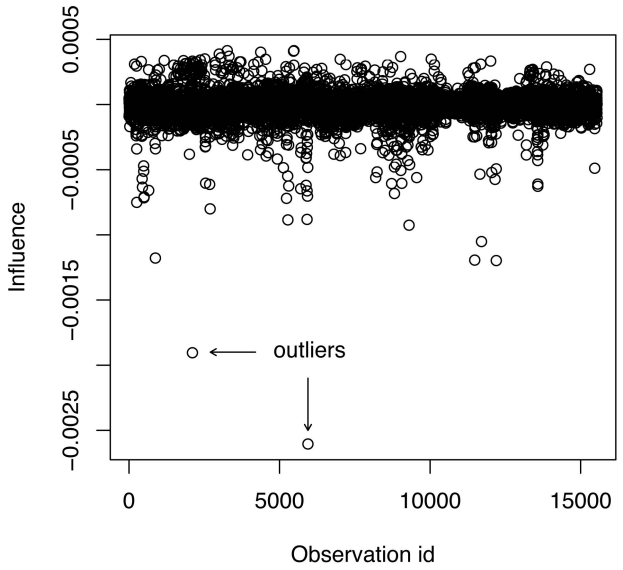

Fig. 3. Influences of the individual data points in the Mozilla model.

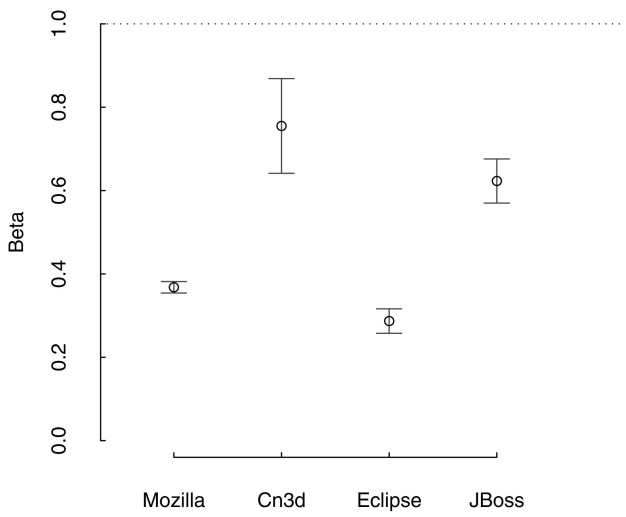

Fig. 4. Coefficient (beta) values in all models (point estimates with 95 percent confidence intervals calculated using the robust sandwich estimate of the standard error of beta).

a result, the size-defect relationship can be viewed as a power-law relationship, whose shape is further investigated in this section.

Without loss of generality, for two classes j and k with sizes $x_j > x_k$, at any time t, the relative defect proneness (RDP) is (using (2) with logarithmic link function but omitting time t to simplify the notation):

$$\lambda_j/\lambda_k = e^{\beta(\ln x_j - \ln x_k)} = e^{\beta\left(\ln(x_j/x_k)\right)} = (x_j/x_k)^{\beta}. \qquad (4)$$

According to this equation, $\hat{\beta} = 1$ would imply that the RDP of class j compared to class k was equal to their size ratio x_j/x_k. $\hat{\beta} > 1$ would mean that defect proneness grows faster than size. However, as seen in Fig. 4, $\hat{\beta} < 1$ was observed with 95 percent confidence for all examined products. Indeed, except Cn3d, which provided less data points and a larger confidence interval because of its smaller size, $\hat{\beta} < 1$ was observed with even more than 99 percent confidence. Since $\hat{\beta} < 1$ was obtained for all products studied, the RDP of class j compared to class k was actually less than their size ratio, x_j/x_k.

To summarize, we observed that defect proneness increased with module size but at a smaller rate. Therefore, smaller modules were *proportionally* more defect prone compared to larger ones.

5 Discussion

Our findings have important practical implications. They imply that, when carrying out focused quality assurance activities such as testing and code inspections, it will be useful to consider giving higher priority to smaller modules.

To clarify, we are not suggesting that practitioners rely on a strict ordering of module size in their prioritizations for inspections and testing. Surely, there will be other concerns such as the business values of modules, which modules are logically related to be inspected or tested together, the operational profile, which modules are library modules (having high fan-in), etc. Instead, we are suggesting that practitioners choose among different prioritization strategies by also considering that smaller modules are proportionally more defect prone.

Let us discuss a hypothetical scenario to explain why using our principle is useful. Assume that the Mozilla team has a limited and fixed amount of resources for code inspections, denoted by R. There are two inspection strategies proposed, each spending the same R but focusing on a different selection of modules. The team has to make a decision by comparing the estimated defect proneness of the modules selected by two strategies. This RDP is the cost-effectiveness of one strategy over the other because both strategies use the same amount of resources.

Here, we assume that all modules have an equal business value and all defects are equally detectable. For the sake of simplicity, we also assume that there is a linear positive relationship between R spent for a module and module size. Therefore, we will use LOC as a metric for R rather than the actual dollar amount, say $R = 10,000$ LOC.

The first proposed strategy chooses 80 modules of 100 LOC and 2 modules of 1,000 LOC. The second proposed strategy chooses 20 modules of 100 LOC and 8 modules of 1,000 LOC. These classes are imaginary and they are only used to demonstrate what the models presented in Table 3 would tell us about RDP.

Using our model for developed Mozilla, the RDP of the modules selected by the first strategy with respect to the modules selected by the second strategy can be estimated as (using $\hat{\beta} = 0.368$ with 95 percent confidence interval, [0.333,0.403] found for Mozilla):

$$RDP = \frac{(80 * e^{0.368*\ln 100}) + (2 * e^{0.368*\ln 1,000})}{(20 * e^{0.368*\ln 100}) + (8 * e^{0.368*\ln 1,000})} = 218.96\%\ [95\%\ CI\ 211.19\%, 226.70\%]. \qquad (5)$$

The detailed equation to calculate RDP and its derivation are presented in Appendix B. Note that this equation can be used for *any two portfolios* of modules.

TABLE 4
Relative Cost-Effectiveness of Choosing the First Strategy Compared to Choosing the Second Strategy for All Products Studied

Product	RDP (95% Confidence Interval)
Mozilla	2.19 (2.16 – 2.22)
Cn3d	1.40 (1.09 – 1.77)
Eclipse	2.36 (2.25 – 2.47)
JBoss	1.65 (1.50 – 1.81)

In Mozilla, choosing the first strategy would be almost 119 percent more cost-effective than choosing the second one, providing considerable savings in real-life software development projects. Table 4 shows the analysis results for the same scenario for all products studied, where the estimates are shown with their 95 percent confidence intervals. These results show that there would be considerable gains in all projects if the first strategy were preferred over the second one.

Our models point to the effectiveness of the strategy of giving higher priority to smaller modules for focused quality assurance activities when the resources are limited and fixed, which is usually the case in many software development projects.

6 Threats to Validity

Since this is an empirical study, in this section, we will discuss the potential threats to the validity of our results.

Construct validity. In this research, defect proneness was operationalized by using the instantaneous risk of receiving a defect fix for classes. We identified defect fixes from the CVS logs entered by developers. Surely, some potential issues can be raised, such as 1) some defects may not surface, 2) some defects may surface but not get fixed, and 3) some defect fixes may not be recorded in CVS logs.

Still, in many studies of software quality, defect fix data have served useful purposes (see Appendix A). Since we know that a large community of users and developers used the open-source products studied here, we are confident that the defects were revealed and fixed adequately. In an earlier study [34], we had found that there had been established processes for defect tracking in medium and large-scale open-source projects.

Internal validity. Some other variables may have an influence on defect proneness. Many other structural measures have been found to be strongly correlated with size, but their additional benefit in characterizing defect-prone modules have been questioned (e.g., see [17], [19], [20]). Therefore, the internal validity threat caused by not using other structural measures should be minor. Obviously, developers' skills, expertise, and training could affect defect proneness. Similarly to many earlier studies, such data were not available in this study. At this moment, we have no reason to believe that the distribution of those factors over size was nonuniform. In addition, an advantage of Cox modeling over other approaches is that it assumes a baseline hazard, which is cancelled out when one is interested in the relative hazard. Therefore, Cox modeling is appropriate in this context when some factors are unknown or uncontrollable (a similar situation occurs in epidemiology, especially in cancer research, where Cox modeling is used intensively).

While calculating the RDP in Section 5, we made the simplifying assumption that the inspection resources spent for a module is linearly related to its size. If the empirical evidence shows otherwise, the main findings about the nature of the size-defect relationship will not change; however, the cost-effectiveness results should be revised accordingly.

External validity. In this study, we validated our findings from Mozilla by using the size and defect data collected from three other open-source products with different characteristics. By doing so, we have gained more confidence in the external validity of the results reported in this paper. However, one could still claim that the functional form observed in this study will be different in different systems developed for different domains with different complexity. Surely, the replicated studies examining this relationship on other software products will be useful to assess the generalizability of our findings.

Conclusion validity. We showed that the assumptions of the statistical models and tests used in the study were valid. For example, when identifying the link function for size in the Cox model, we took the intrasubject correlations into account and used cubic spline functions; we validated the proportional hazards assumption of the Cox model; we examined the effects of overly influential observations; in the estimation of the standard error of β, we again considered intrasubject correlations and used Jackknife to obtain a robust estimate.

Overall, even though there might be potential threats to the validity of this study just like any other empirical study, we have confidence that our study and findings have adequate validity.

7 Conclusion

The functional form of the relationship between software size and defect proneness is important for various development decisions regarding how to prioritize focused testing and inspection activities with limited amount of resources for quality assurance.

Our results for four different products consistently demonstrated a statistically significant relationship between size and defect proneness. Defect proneness increased with size but at a slower rate. These findings imply that, contrary to common perception, smaller modules are proportionally more problematic compared to the larger ones.

The results of this study can be immediately useful to software practitioners when they make decisions about which modules should be chosen for focused testing and inspection activities. As demonstrated in our hypothetical example, choosing a larger percentage of smaller modules is a considerably advantageous strategy. Therefore, practitioners should consider giving higher priority to smaller modules in order to improve the effectiveness of their quality improvement plans and to use their limited resources more efficiently.

In the future, replicated studies on other open and closed-source software products will be highly useful to

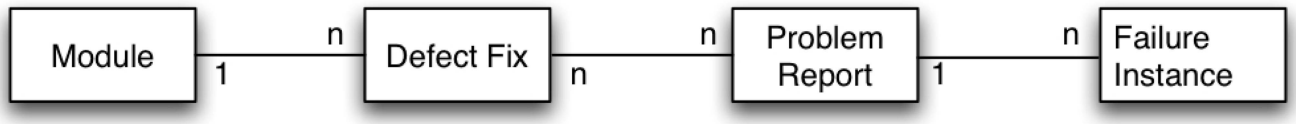

Fig. 5. Basic entity-relationship model for concepts related to the assessment of defect proneness for software modules.

assess the generalizability of our findings and to see whether the observed phenomenon can be stated as a theory. Future studies can also investigate the mechanisms behind our findings. Earlier, Basili and Perricone [6] reported that the interface defects, those (defects) that were associated with structures existing outside the modules local environment but which the module used, could explain why smaller modules are proportionally more troublesome because, in the system they studied, the interface defects were equally distributed over smaller and larger modules. This is an important speculation for further investigation because, if validated, it can provide an explanation about the mechanisms behind the size-defect relationship discovered in this study.

Appendix A

Assessment of Defect Proneness for Software Modules

In this appendix, we discuss how defect proneness can be assessed in the context of a software development project. Fig. 5 shows a basic entity-relationship diagram for the related concepts and the relationships among them.

In many organizations and open-source projects, whenever a failure or a problem is observed (through testing or from customers or users) a Problem Report (PR) is opened. A PR typically describes the symptoms of the problem and how to recreate them. It is useful to distinguish between a PR and a failure instance. Consider the relationships in Fig. 5 for a development environment where problems are consistently recorded. Each failure instance is associated with only a single PR, but that a PR can be associated with multiple failure instances. A PR can be resolved with multiple defect fixes, possibly in the same module or across multiple modules. Also, the same defect fix can help resolve multiple PRs if the defect has multiple symptoms. A defect fix occurs in a single module, and each module may have multiple defect fixes.

In real-life software development settings, the actual number of defects in a module cannot be known with certainty [59]. Therefore, the number of defect fixes has been traditionally used as a proxy for defect proneness in many empirical studies of software quality and served to many useful purposes (e.g., [6], [44], [60]). Defect proneness is usually assessed by looking at the corrective changes in the version control system used by developers. Each change to a module is examined and, if the change was to make a fix, then this indicates that there was a defect in the module. By observing the number of changes implementing fixes to a module (i.e., defect fixes), a proxy measure of defect proneness for that module is obtained.

Appendix B

Derivation of the Equation for Relative Defect Proneness

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) $s_1, s_2, \ldots, s_m$ and the second one chooses n modules having sizes $S_1, S_2, \ldots, S_n$.

First, let us take a reference module with size C. Since we identified a logarithmic 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 $e^{\beta(\ln s - \ln C)} = e^{\beta \ln \frac{s}{C}}$. For each inspection strategy, we calculate the sum of the RDP of the selected individual modules with respect to the reference module. To find RDP, we simply take the ratio of these sums:

$$
\begin{aligned}
RDP &= \frac{\sum_{i=1}^{m} e^{\beta \ln \frac{s_i}{C}}}{\sum_{i=1}^{n} e^{\beta \ln \frac{S_i}{C}}} = \frac{\sum_{i=1}^{m} e^{\beta \ln s_i - \beta \ln C}}{\sum_{i=1}^{n} e^{\beta \ln S_i - \beta \ln C}} \\
&= \frac{\sum_{i=1}^{m} \frac{e^{\beta \ln s_i}}{e^{\beta \ln C}}}{\sum_{i=1}^{n} \frac{e^{\beta \ln S_i}}{e^{\beta \ln C}}} = \frac{\cancel{\frac{1}{e^{\beta \ln C}}} \sum_{i=1}^{m} e^{\beta \ln s_i}}{\cancel{\frac{1}{e^{\beta \ln C}}} \sum_{i=1}^{n} e^{\beta \ln S_i}} \\
&= \frac{\sum_{i=1}^{m} e^{\beta \ln s_i}}{\sum_{i=1}^{n} e^{\beta \ln S_i}}. \qquad (6)
\end{aligned}
$$

Acknowledgments

The authors are grateful to Dr. Frank E. Harrell for his help and advice, to the associate editor, Ross Jeffery, and to the anonymous reviewers for their careful review of this paper and for their useful and constructive comments.

References

[1] F. Akiyama, "An Example of Software System Debuggings," *Proc. Int'l Federation of Information Processing Societies Congress,* vol. 1, pp. 353-359, 1971.

[2] P.K. Andersen, O. Borgan, R.D. Gill, and N. Keiding, *Statistical Models Based on Counting Processes.* Springer-Verlag, 1993.

[3] P.K. Andersen and R.D. Gill, "Cox's Regression Model for Counting Processes: A Large Sample Study," *Annals of Statistics,* vol. 10, pp. 1100-1120, 1982.

[4] C. Andersson and P. Runeson, "A Replicated Quantitative Analysis of Fault Distributions in Complex Software Systems," *IEEE Trans. Software Eng.,* vol. 33, no. 5, pp. 273-286, May 2007.
[5] D.L. Atkins, T. Ball, T.L. Graves, and A. Mockus, "Using Version Control Data to Evaluate the Impact of Software Tools: A Case Study of the Version Editor," *IEEE Trans. Software Eng.,* vol. 28, no. 7, pp. 625-637, July 2002.
[6] V.R. Basili and B.T. Perricone, "Software Errors and Complexity: An Empirical Investigation," *Comm. ACM,* vol. 27, no. 1, pp. 42-52, 1984.
[7] L.C. Briand, V.R. Basili, and C.J. Hetmanski, "Developing Interpretable Models with Optimized Set Reduction for Identifying High-Risk Software Components," *IEEE Trans. Software Eng.,* vol. 19, no. 11, pp. 1028-1044, Nov. 1993.
[8] D.N. Card and R.L. Glass, *Measuring Software Design Quality.* Prentice Hall, 1990.
[9] F. Chayes, *Ratio Correlation: A Manual for Students of Petrology and Geochemistry.* Univ. of Chicago Press, 1971.
[10] *Cn3D Project,* Archived by Webcite at http://www.webcitation.org/5TuzAf1KC, 2007.
[11] B.T. Compton and C. Withrow, "Prediction and Control of Ada Software Defects," *J. Systems and Software,* vol. 12, no. 3, pp. 199-207, 1990.
[12] *Concurrent Versions Software,* p. Archived by Webcite at http://www.webcitation.org/5TuzMgxuG, 2007.
[13] D.R. Cox, "Regression Models and Life Tables," *J. Royal Statistical Soc.,* vol. 34, pp. 187-220, 1972.
[14] *Eclipse Project,* Archived by Webcite at http://www.webcitation.org/5TuyrW4y4, 2007.
[15] K. El Emam, *The ROI from Software Quality.* Auerbach Publications, Taylor and Francis Group, LLC, 2005.
[16] K. El Emam, S. Benlarbi, N. Goel, W. Melo, H. Lounis, and S.N. Rai, "The Optimal Class Size for Object-Oriented Software," *IEEE Trans. Software Eng.,* vol. 28, no. 5, pp. 494-509, May 2002.
[17] K. El Emam, S. Benlarbi, N. Goel, and S.N. Rai, "The Confounding Effect of Class Size on the Validity of Object-Oriented Metrics," *IEEE Trans. Software Eng.,* vol. 27, no. 7, pp. 630-650, July 2001.
[18] K. El Emam, W. Melo, and J.C. Machado, "The Prediction of Faulty Classes Using Object-Oriented Design Metrics," *J. Systems and Software,* vol. 56, no. 1, pp. 63-75, 2001.
[19] N.E. Fenton and M. Neil, "A Critique of Software Defect Prediction Models," *IEEE Trans. Software Eng.,* vol. 25, no. 5, pp. 675-689, Sept./Oct. 1999.
[20] N.E. Fenton and N. Ohlsson, "Quantitative Analysis of Faults and Failures in a Complex Software System," *IEEE Trans. Software Eng.,* vol. 26, no. 8, pp. 797-814, Aug. 2000.
[21] Y. Funami and M.H. Halstead, "A Software Physics Analysis of Akiyama's Debugging Data," *Proc. MRI XXIV Int'l Symp. Computer Software Eng.,* pp. 133-138, 1976.
[22] J.E. Gaffney, "Estimating the Number of Faults in Code," *IEEE Trans. Software Eng.,* vol. 10, no. 4, pp. 459-465, 1984.
[23] T. Gyimothy, R. Ferenc, and I. Siket, "Empirical Validation of Object-Oriented Metrics on Open Source Software for Fault Prediction," *IEEE Trans. Software Eng.,* vol. 31, no. 10, pp. 897-910, Oct. 2005.
[24] M.H. Halstead, *Elements of Software Science.* Elsevier, 1977.
[25] P.G. Hamer and G.D. Frewin, "M.H. Halstead's Software Science —A Critical Examination," *Proc. Sixth Int'l Conf. Software Eng.,* pp. 197-206, 1982.
[26] F.E. Harrell, *Regression Modeling Strategies: With Applications to Linear Modes, Logistic Regression, and Survival Analysis.* Springer-Verlag, 2001.
[27] L. Hatton, "Reexamining the Fault Density-Component Size Connection," *IEEE Software,* vol. 14, no. 2, pp. 89-97, Apr. 1997.
[28] L. Hatton, "Does OO Sync with the Way We Think," *IEEE Software,* vol. 15, no. 3, pp. 46-54, May/June 1998.
[29] D.W. Hosmer and S. Lemeshow, *Applied Survival Analysis: Regression Modeling of Time to Event Data.* John Wiley & Sons, 1999.
[30] *JBoss Project,* Archived by Webcite at http://www.webcitation.org/5TuyyIF4R, 2007.
[31] T.M. Khoshgoftaar, E.B. Allen, J. Hudepohl, and S. Aud, "Applications of Neural Networks to Software Quality Modeling of a Very Large Telecommunications System," *IEEE Trans. Neural Networks,* vol. 8, no. 4, pp. 902-909, 1997.
[32] T.M. Khoshgoftaar and R.M. Szabo, "Using Neural Networks to Predict Software Faults during Testing," *IEEE Trans. Reliability,* vol. 45, no. 3, pp. 456-462, 1996.
[33] A.G. Koru and J. Tian, "An Empirical Comparison and Characterization of High Defect and High Complexity Modules," *J. Systems and Software,* vol. 67, no. 3, pp. 153-163, 2003.
[34] A.G. Koru and J. Tian, "Defect Handling in Medium and Large Open Source Projects," *IEEE Software,* vol. 21, no. 4, pp. 54-61, July/Aug. 2004.
[35] A.G. Koru and J. Tian, "Comparing High Change Modules and Modules with the Highest Measurement Values in Two Large-Scale Open-Source Products," *IEEE Trans. Software Eng.,* vol. 31, no. 8, pp. 625-642, Aug. 2005.
[36] A.G. Koru, D. Zhang, and H. Liu, "Modeling the Effect of Size on Defect Proneness for Open-Source Software," *Proc. Third Int'l Workshop Predictor Models in Software Eng.,* 2007.
[37] J.-C. Laprie and K. Kanoon, "Software Reliability and System Reliability," *Handbook of Software Reliability Engineering,* M.R. Lyu, ed., vol. 1, pp. 27-69, IEEE CS Press-McGraw Hill, 1996.
[38] M. Lipow, "Number of Faults per Line of Code," *IEEE Trans. Software Eng.,* vol. 8, no. 4, pp. 437-439, 1982.
[39] T.J. McCabe, "A Complexity Measure," *IEEE Trans. Software Eng.,* vol. 2, no. 6, pp. 308-320, 1976.
[40] A. Mockus, R.T. Fielding, and J. Herbsleb, "A Case Study of Open Source Software Development: The Apache Server," *Proc. 22nd Int'l Conf. Software Eng.,* pp. 263-272, 2000.
[41] A. Mockus, R.T. Fielding, and J. Herbsleb, "Two Case Studies of Open Source Software Development: Apache and Mozilla," *ACM Trans. Software Eng. and Methodology,* vol. 11, no. 3, pp. 309-346, 2002.
[42] K. Moller and D. Paulish, "An Empirical Investigation of Software Fault Distribution," *Proc. First Int'l Software Metrics Symp.,* pp. 82-90, May 1993.
[43] *Mozilla Project,* Archived by Webcite at http://www.webcitation.org/5RqqbCKKm, 2007.
[44] J.C. Munson and T.M. Khoshgoftaar, "The Detection of Fault-Prone Programs," *IEEE Trans. Software Eng.,* vol. 18, no. 5, pp. 423-433, May 1992.
[45] L.M. Ottenstein, "Quantitative Estimates of Debugging Requirements," *IEEE Trans. Software Eng.,* vol. 5, no. 5, pp. 504-514, 1979.
[46] A.A. Porter and R.W. Selby, "Empirically Guided Software Development Using Metric-Based Classification Trees," *IEEE Software,* vol. 7, no. 2, pp. 46-54, Apr. 1990.
[47] E.S. Raymond, *The Cathedral and the Bazaar: Musings on Linux and Open Source by an Accidental Revolutionary.* O'Reilly and Assoc., 1999.
[48] J. Rosenberg, "Some Misconceptions about Lines of Code," *Proc. Fourth Int'l Symp. Software Metrics,* pp. 137-142, 1997.
[49] M. Schemper and J. Stare, "Explained Variation in Survival Analysis," *Statistics in Medicine,* vol. 15, pp. 1999-2012, 1996.
[50] N.F. Schneidewind and H.-M. Hoffmann, "An Experiment in Software Error Data Collection and Analysis," *IEEE Trans. Software Eng.,* vol. 5, no. 3, pp. 276-285, 1978.
[51] *Understand for C++: User Guide and Reference Manual,* I. Scientific Toolworks, Jan. 2003.
[52] R.W. Selby and V.R. Basili, "Analyzing Error-Prone System Structure," *IEEE Trans. Software Eng.,* vol. 17, no. 2, pp. 141-152, Feb. 1991.
[53] V.Y. Shen, T.J. Yu, S.M. Thebaut, and L. Paulsen, "Identifying Error-Prone Software—An Empirical Study," *IEEE Trans. Software Eng.,* vol. 11, no. 4, pp. 317-324, 1985.
[54] *Subversion,* Archived by Webcite at http://www.webcitation.org/5U90RHRqb, 2007.
[55] R. Thayer, M. Lipow, and E. Nelson, *Software Reliability.* North-Holland, 1978.
[56] T.M. Therneau and P.M. Grambsch, *Modeling Survival Data: Extending the Cox Model.* Springer-Verlag, 2000.
[57] T.M. Therneau, P.M. Grambsch, and T.R. Fleming, "Martingale Based Residuals for Survival Models," *Biometrika,* vol. 77, pp. 147-160, 1990.
[58] J. Tian, "Quality Assurance Alternatives and Techniques: A Defect-Based Survey and Analysis," *Software Quality Professional,* vol. 3, no. 3, pp. 6-18, 2001.
[59] J. Tian, *Software Quality Engineering: Testing, Quality Assurance, and Quantifiable Improvement.* John Wiley & Sons, 2005.
[60] J. Troster and J. Tian, "Measurement and Defect Modeling for a Legacy Software System," *Annals of Software Eng.,* vol. 1, pp. 95-118, 1995.

A. Güneş Koru received the BS degree in computer engineering from Ege University, Izmir, Turkey, in 1996, the MS degree in computer engineering from Dokuz Eylül University, Izmir, in 1998, and the MS degree in software engineering and the PhD degree in computer science from Southern Methodist University (SMU), Dallas, in 2002 and 2004, respectively. He is an assistant professor in the Department of Information Systems at the University of Maryland, Baltimore County. His research interests include software quality, measurement, maintenance, and evolution, open source software, bioinformatics, and healthcare informatics.

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

Khaled El Emam received the PhD degree from the University of London. He is an associate professor on the Faculty of Medicine and the School of Information Technology and Engineering at the University of Ottawa. He is a Canada Research Chair in Electronic Health Information at the University of Ottawa. Previously, he was a senior research officer for the National Research Council of Canada and, prior to that, he was the head of the Quantitative Methods Group at the Fraunhofer Institute, 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. His lab Web site is http://www.ehealthinformation.ca/.

Hongfang Liu received the BS degree in applied mathematics and statistics from the University of Science and Technology of China in 1994, the MS degree in computer science from Fordham University in 1998, and the PhD degree in computer science from the City University of New York in 2002. She is currently an assistant professor in the Department of Biostatistics, Bioinformatics, and Biomathematics (DBBB) at 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.