The Optimal Class Size for Object-Oriented Software

Khaled El Emam, Saïda Benlarbi, Nishith Goel, Walcelio Melo, Hakim Lounis, and Shesh N. Rai

Abstract—A growing body of literature suggests that there is an optimal size for software components. This means that components that are too small or too big will have a higher defect content (i.e., there is a U-shaped curve relating defect content to size). The U-shaped curve has become known as the "Goldilocks Conjecture." Recently, a cognitive theory has been proposed to explain this phenomenon and it has been expanded to characterize object-oriented software. This conjecture has wide implications for software engineering practice. It suggests 1) that designers should deliberately strive to design classes that are of the optimal size, 2) that program decomposition is harmful, and 3) that there exists a maximum (threshold) class size that should not be exceeded to ensure fewer faults in the software. The purpose of the current paper is to evaluate this conjecture for object-oriented systems. We first demonstrate that the claims of an optimal component/class size (1) above) and of smaller components/classes having a greater defect content (2) above) are due to a mathematical artifact in the analyses performed previously. We then empirically test the threshold effect claims of this conjecture (3) above). To our knowledge, the empirical test of size threshold effects for object-oriented systems has not been performed thus far. We performed an initial study with an industrial C++ system and repeated it twice on another C++ system and on a commercial Java application. Our results provide unambiguous evidence that there is no threshold effect of class size. We obtained the same result for three systems using four different size measures. These findings suggest that there is a simple continuous relationship between class size and faults, and that, optimal class size, smaller classes are better and threshold effects conjectures have no sound theoretical nor empirical basis.

Index Terms—Object-oriented metrics, software quality, quality models, quality prediction, software size, optimal size.

1 Introduction

An emerging theory in the software engineering literature suggests that there is an optimal size for software components. This means that components that are of approximately this (optimal) size are least likely to contain a fault, or will contain relatively less faults than other components further away from the optimal size.[1] This optimal size has been stated to be, for example, 225 LOC for Ada packages [49], 83 total Ada source statements [12], 400 LOC for Columbus-Assembler [34], 877 LOC for Jovial [20], 100-150 LOC for Pascal and Fortran code [31], 200-400 LOC irrespective of the language [25], 25-80 Executable LOC [1], or 50-160 SLOC [1]. Card and Glass [8] note that military standards for module size range from 50 to 200 executable statements. Kan [30] plotted a curve of the relationship between LOC and fault density for IBM's AS/400, and concluded that there was indeed an optimal component size. This U-shaped curve is depicted in Fig. 1 as it is often presented in the literature. It shows that fault density is at a minimal value at a certain component size. Recently, Hatton [25] has articulated this theory forcefully and proposed a cognitive mechanism that would explain it. Fenton and Neil [17] term this theory the "Goldilocks Conjecture."

Indeed, if the "Goldilocks Conjecture" is demonstrated to be an accurate description of reality, this would have wide implications on software engineering design. Designers should endeavor to decompose their systems so that most of their components are at or near optimal size. For instance, Withrow [49] has stated "software designers may decrease how error-prone their products are by decomposing problems in a way that leads to software modules that are neither too large nor too small," and [12] "This study suggests that software engineers working with intermediate-size packages have an increased probability of producing software that is minimally defect-prone." Kan [30] states "when an empirical optimum is derived by reasonable methods (for example, based on the previous release of the same product, or based on a similar product by the same development group), it can be used as a guideline for new module development."

1. Note that the optimal size theory is stated in terms of component size, which could be functions in the procedural paradigm and classes in the object-oriented paradigm. It is not stating that there is an optimal program size, where a program consists of many components. Also note that the optimal size theory is stated and demonstrated in terms of fault minimzation, rather than other outcome measures such as maximizing maintenance productivity.

- K. El Emam is with National Research Council of Canada, Institute for Information Technology, Building M-50, Montreal Road, Ottawa, Ontario, Canada K1A OR6. E-mail: Khaled.El-Emam@nrc.ca.
- S. Benlarbi is with the Carrier Internetworking Division, Alcatel Networks Corp., 600 March Road, Kanata, Ontario K2K 2E6. E-mail: saida.benlarbi@alcatel.com.
- N. Goel is with Cistel Technology, 210 Colonnade Road, Suite 204, Nepean, Ontario, Canada K2E 7L5. E-mail: ngoel@cistel.com.
- W. Melo is with Oracle Brazil, SCN Qd. 2 Bl. A, Ed. Corporate, S. 604, 70712-900 Brasilia, DF, Brazil. E-mail: wmelo@br.oracle.com.
- H. Lounis is with CRIM, 550 Sherbrooke West, Suite 100, Montreal, Quebec, Canada H3A 1B9. E-mail: hlounis@crim.ca.
- S.N. Rai is with the Department of Biostatistics & Epidemiology, St. Jude Children's Research Hospital, 332 N. Lauderdale St., Memphis, TN 38105-2794. E-mail: shesh.rai@stjude.org.

Manuscript received 11 Mar. 2000; revised 15 Mar. 2001; accepted 16 Mar. 2001.
Recommended for acceptance by S. Pfleeger.
For information on obtaining reprints of this article, please send e-mail to: tse@computer.org, and reference IEEECS Log Number 111719.

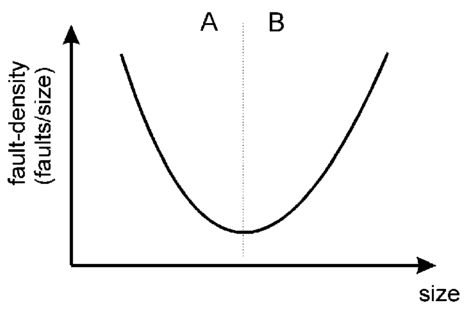


Fig. 1. U-shaped curve relating fault density to size that exemplifies the optimal size theory. The curve can be broken up into two parts: A and B. The evidence that exists sometimes supports the whole curve, part A, or part B.

The evidence that does exist does not always support the U-shaped curve depicted above. Some studies demonstrate only part A in Fig. 1. Others only support part B.

The first part of the Goldilocks Conjecture argues that smaller components have higher fault proneness than larger components. There are a number of empirical observations that demonstrate this. For example, Basili and Perricone [2] observed that smaller Fortran components are likely to have greater fault density, but did not identify an optimal component size. A similar observation was made by Davey et al. for C modules [13] and Selby and Basili [42] for routines written in a high-level language similar to PL/I used at the NASA GSFC. Shen et al. [43] analyzed data from three systems[2] at IBM and found that fault density was higher for smaller components, but again did not identify a point where the fault density would start to rise again for larger components. Moller and Paulish [34] found that components less than 70 LOC tended to have a higher fault density than larger components. These results mean that reducing the size of components is detrimental to software quality. As noted by Fenton and Neil [17], this is counter to one of the axioms of software engineering, namely program decomposition. In fact, this is one of the more explicit and controversial claims made by proponents of the optimal size theory [23], [24], [26].

Part B of Fig. 1 illustrates another element of the Goldilocks Conjecture: Beyond a certain size, fault proneness increases rapidly. This is in essence a *threshold effect*. For instance, Card and Glass [8] note that many programming texts suggest limiting component size to 50 or 60 SLOC. A study by O'Leary [35] of the relationship between size and faults in knowledge-based systems found no relationship between size and faults for small components, but a positive relationship for large components; again suggesting a threshold effect. A number of standards and organizations had defined upper limits on components size [5], for example, an upper limit of 200 source statements in MIL-STD-1679, 200 HOL executable statements in MIL-STD-1644A, 100 statements excluding annotation in RADC CP 0787796100E, 100 executable source lines in MILSTAR/ESD Spec, 200 source statements in MIL-STD-SDS, 200 source statements in MIL-STD-1679(A), and 200 HOL executable statements in FAA ER-130-005D. Bowen [5] proposed component size thresholds between 23-76 source statements based on his own analysis. After a lengthy critique of size thresholds, Dunn and Ullman [15] suggest two pages of source code listing as an indicator of an overly large component. Woodfield et al. [50] suggest a maximum threshold of 70 LOC.

The evidence for the Goldilocks Conjecture and its parts is not unequivocal, however. In fact, there exists some evidence that is completely contradictory. For instance, Card and Glass [8] reanalyze a data set using nonparametric techniques that on the surface suggested a higher fault density for smaller components. The reanalysis found no relationship between size and fault density. Fenton and Ohlsson [18] also found no obvious relationship between size and fault density[3] for a system at Ericsson.[4]

Some researchers have extended the argument for optimal component size to the object-oriented paradigm. For instance, Compton and Withrow [12] argue "We believe that the shape of the defect density curve [..] is fundamentally a reflection of optimal software decomposition, whether it was decomposed functionally or as objects." Hatton [27] applies a memory model from cognitive psychology and concludes that there is a U-shaped relationship between fault density and the class size of object-oriented software.

However, if smaller components are more likely to contain a fault than "medium-sized" components, this implies major problems for object-oriented programming.[5] The object-oriented strategies of limiting a class' responsibility and reusing it in multiple contexts results in a profusion of small classes in object-oriented systems [48]. For instance, Chidamber and Kemerer [11] found in two systems studied[6] that most classes tended to have a small number of methods (0-10), suggesting that most classes are relatively simple in their construction, providing specific abstraction and functionality. Another study of three systems performed at Bellcore[7] found that half or more of the methods are fewer than four Smalltalk lines or two C++ statements, thus suggesting that the classes consist of small methods [48]. According to the Goldilocks Conjecture, this practice is detrimental to the quality of object-oriented applications.

Thresholds have been derived for the size of object-oriented classes. For instance, Lorenz and Kidd [32] recommend a maximum of 20 methods per class for non-UI

2. The systems were implemented in Pascal, PL/S, and assembler, respectively.

3. This conclusion is based on the pooling of prerelease and postrelease faults.

4. Interestingly, Card and Glass [8] initially examined a scatter plot of size versus fault density and concluded that it was misleading. They subsequently grouped their data and based on that concluded that there was no relationship. On the other hand, Fenton and Ohlsson [18] started off with the data grouped and concluded that this way of analysis was misleading. They then examined a fault density versus size scatter-plot, from which they concluded that there was no relationship.

5. An object-oriented component is defined as a class in this paper.

6. One system was in developed in C++, and the other in Smalltalk.

7. The study consisted of analyzing C++ and Smalltalk systems and interviewing the developers for two of them. For a C++ system, method size was measured as the number of executable statements and, for Smalltalk, size was measured by uncommented nonblank lines of code.

classes,[8] nine message sends per method, six LOC for Smalltalk methods, and 24 LOC for C++ methods. Rosenberg et al. [38] and French [19] present a number of thresholds for class size. If the existence of thresholds can be supported through systematic empirical study, then that could greatly simplify quality management for object-oriented projects.[9] However, none of these three studies demonstrated that indeed classes that exceed the thresholds are more fault prone (i.e., the thresholds were not empirically validated).

From the above exposition, it is clear that the implications of the optimal size theory on the design of object-oriented systems are significant. Even if only parts of the theory are found to be supportable, this would suggest important changes to current design or quality management practices. As noted above, however, the combination of evidence gives a confusing picture. This confusion makes acting on any of the above claims premature.

In this paper, we present a theoretical and empirical evaluation of this conjecture for object-oriented applications. We first demonstrate that the claim of smaller components being more fault prone than larger components to be a consequence of a mathematical artifact. We then test the threshold theory on three object-oriented applications. Our results provide clear evidence that there is no size threshold effect whatsoever. Hence, at least for object-oriented systems, the optimal size theory and its subparts are without support. We then state a simpler theory that matches our results.

In the next section, we review the literature on the Goldilocks Conjecture and its parts. In Section 3, we describe in detail the research method that we used for testing the threshold effect theory and our results are described in Section 4. We conclude the paper in Section 5 with an overall summary and the implications of our findings.

2 Background

The Goldilocks Conjecture stipulates that there exists an optimal component size, S. As component size increases or decreases away from S, the relative number of faults, or alternatively the probability of a fault, increases. This theory can be broken down into two parts that match A and B in Fig. 1:

1. For any set of components with size S, S', and S'', where $S'' < S' < S$, then $D(S'') > D(S') > D(S)$, where $D(\cdot)$ is the fault density given the size $(\cdot)$.
2. For any set of components with size S, S', and S'', where $S'' > S' > S$, then $D(S'') > D(S') > D(S)$, where $D(\cdot)$ is the fault density given the size $(\cdot)$.

The evidence and arguments that support the whole of the U-shaped curve phenomenon and the first part are similar and, therefore, they are addressed jointly in the following subsection. We then consider the second part of the Goldilocks Conjecture.

8. User-Interface classes.

9. Thresholds provide a straight forward way for flagging classes that are high risk. Hence, they can be targeted for special defect detection activities, such as more extensive testing.

2.1 The High Fault-Density of Small Components and The U-Shaped Curve

A number of studies found a negative relationship between fault density and some measure of size for $S' < S$. For example, Basili and Perricone [2] analyzed testing and maintenance fault data for a Fortran project developed at the Software Engineering Laboratory. They found that smaller modules tended to have a larger fault density than larger modules. Size was measured in terms of executable lines of code. Davey et al. [13] found a similar pattern for C modules, Selby and Basili [42] observed the same phenomenon for routines written in a high-level language similar to PL/I, Shen et al. [43] reported the same pattern for three systems at IBM, and Moller and Paulish for systems at Siemens [34].

Other studies reported a U-shaped curve between fault density and size. For example, Withrow [49] analyzed Ada software for command and control of a military communications system developed by a team of 17 professional programmers. Faults were logged during the test and integration phases. In total, 362 components were examined, whereby a component is an Ada package. The results showed a U-shaped curve and gave a clear pattern of higher fault density for smaller components below an identified optimal component size. A subsequent study by Compton and Withrow [12] on an Ada system, whereby they also collected faults during the first year postrelease, found a similar pattern. Defect density tended to increase as component size decreased below the optimal size. Kan [30] plotted a curve of the relationship between LOC and fault density for IBM's AS/400 development and concluded that there was indeed an optimal component size. When looking at components developed using the same programming language across releases, Moller and Paulish [34] identified the U-shaped curve for Columbus assmbler. Lind and Vairavan [31] identified the U-shaped curve for a real-time medical imaging system written mainly in Pascal and Fortran. They [31] explain the increase in fault density after the minimal optimal value by stating that the complexity becomes too excessive for the programmers.

Fenton and Neil [17] provide an explanation for the above observations which they illustrate using a Bayesian Belief Network. They modeled a causal system of variables whereby a solution to a difficult problem was being designed. This leads to a larger design. Furthermore, their model specified that little testing effort was allocated. This means that few faults were detected during testing. Therefore, the larger the component, the smaller the prerelease faults to size ratio; the smaller the component, the larger the faults to size ratio. This is the behavior observed in the studies above. A similar explanation can be derived for the case where testing was extensive and postrelease faults. If testing was extensive, then few faults will be found during operation. Therefore, the postrelease faults to size ratio will be smaller for larger components, and larger for smaller components. However, this explanation fails somewhat in that the studies above were not focused solely on testing faults nor solely on postrelease faults. In fact, a number of them incorporated both prerelease and postrelease faults and the same effect was observed. The explanation is

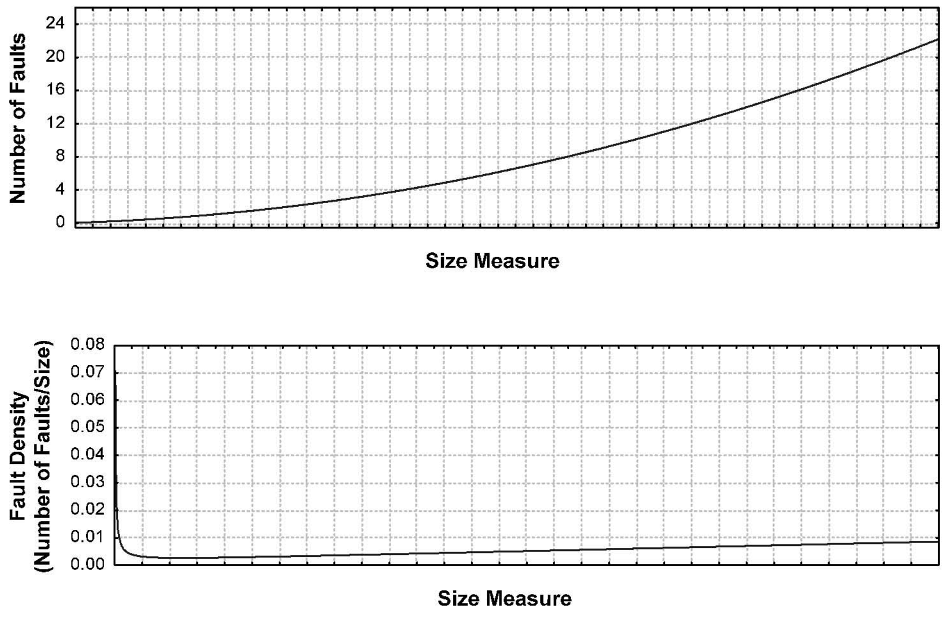


Fig. 2. Relationship between component size and faults according to Compton and Withrow [12].

therefore not feasible since it is contradictory about the intensity of testing effort to explain the pre- and postrelease phenomena. For example, the Selby and Basili [42] study included faults due to Trouble Reports, which are problems reported against working, released code. The studies of Basili and Perricone [2], Shen et al. [43], Moller and Paulish [34], and Compton and Withrow [12] included faults reported during the maintenance phase as well as testing.

A number of further explanations for finding that smaller components have a larger fault content have been put forward:

- When just one part of a module is tested, other parts are to some degree also tested because of the connectivity of all the parts. Therefore, in a larger module, some parts may get "free testing" [49].
- Larger modules contain undiscovered faults because of inadequate test coverage [2], [49] and, therefore, their fault density is low. Alternatively, faults in smaller modules are more apparent (e.g., easier to detect) [2].
- Interface faults tend to be evenly distributed among modules and, when converted to fault density, they produce larger values for small modules [2], [49].
- In some data sets, the majority of components examined for analysis tended to be small, hence biasing the results [2].
- Larger components were coded with more care because of their size [2].

While the above explanations seem plausible, we will show that the observed phenomenon of smaller components having a higher fault density is due to an arithmetic artifact. First, note that the above conclusions were drawn based exclusively on examination of the relationship between *fault density* versus size.

A plot of the basic relationship that Compton and Withrow [12] derived is shown in the top panel of Fig. 2. If we represent exactly the same curve in terms of fault density, we get the curve in the lower panel. The top panel shows that, as size increases, there will be more faults in a component. This makes intuitive sense. The bottom panel shows that smaller components tend to have a higher fault density up to the optimal size (the lowest point in the curve), at which point fault density is at a minima. However, by definition, if we model the relationship between any variable X and 1/X, we will get a negative association as long as the relationship between size and faults is growing at most linearly. Rosenberg [37] has demonstrated this further with a Monte Carlo simulation, whereby he showed that the correlation between fault density and size is always negative irrespective of the distributions of size and faults and their correlation. This is in fact a mathematical artifact of plotting a variable against its own reciprocal.

Similarly, the relationship between size[10] and faults as derived by Hatton [25] is shown in the top panel in Fig. 3. The bottom panel shows the relationship between fault density and size. Again, the plotting of 1/X verus X will, by definition, gives you a negative association, whatever the variables may be.

Chayes [10] derives a formula for the correlation between $\frac{X_1}{X_2}$ and X_2 when there is a zero correlation between the two variables:

$$r \approx \frac{-\mu_1\sigma_2}{\sqrt{\mu_2^2\sigma_1^2 + \mu_1^2\sigma_2^2}}, \qquad (1)$$

where μ_i is the mean of variable X_i and σ_i its standard deviation. It is clear that there will be a negative correlation and possibly a large one (it depends on the variances and means of the "raw" variables), even if the two "raw" variables are not associated with each other. Therefore, drawing conclusions from a fault density versus size

10. The actual analysis used static path count. However, Hatton interprets the results to indicate a relationship between size and faults.

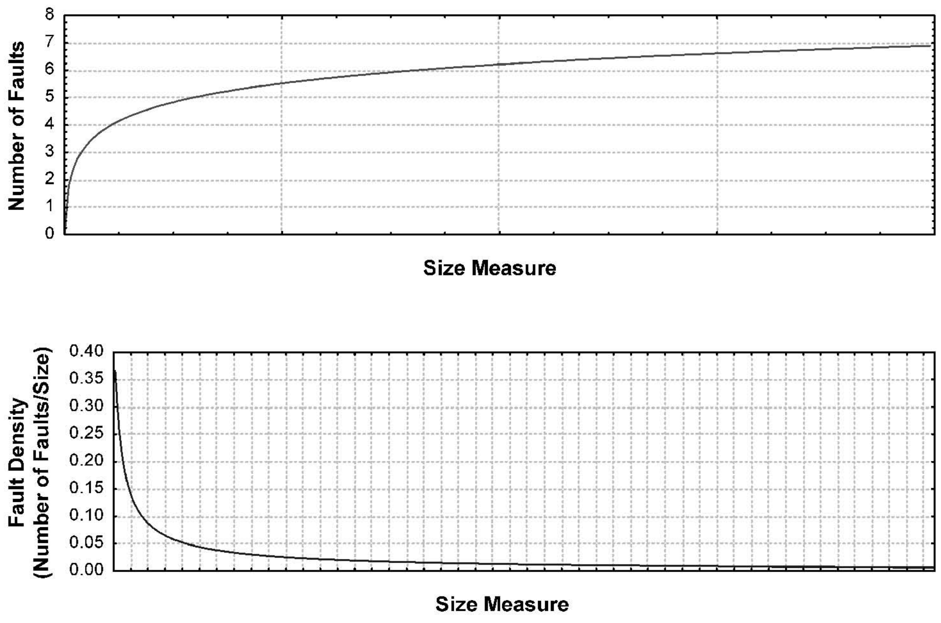


Fig. 3. Relationship between component size and faults according to Hatton [25].

association is by no means an indication that there is an association at all, nor of it being a strong or weak one. On the other hand, as Takahashi and Kamayachi [45] demonstrate, if the relationship between faults and size is linear, then the association between fault density and size will be almost zero, indicating that there is no impact of increased size when in fact there is a strong relationship between size and faults.[11] Therefore, studying fault density versus size can actually mask and mislead us about the true relationship between size and fault content. By a medical analogy, we further show below how relationships can be easily distorted when you plot a compound measure with 1/X versus X.

Consider a medical study, whereby we want to look at the relationship between say exposure to a carcinogen and the incidence of cancer. For example, consider the data in the left panel in Fig. 4. This shows the dose-response relationship between smoking intensity and mortality rates from lung cancer. It is clear that as the extent of smoking increases, the mortality rate increases. This is analogous to plotting the size versus the number of faults in our context. Now, if we plot the mortality per extent of smoking versus the smoking intensity,[12] we get the right panel in Fig. 4. This shows a clear U-shaped curve, suggesting, according to the practice outlined above, that there is an optimal smoking intensity of around 0.75 packs/day. It follows that low levels of smoking (around 0.25 packs/day) is associated with the highest risk of mortality from lung cancer![13]

As is seen from the above example, the conclusions that are drawn defy common sense, and are clearly incorrect. This should illustrate that drawing conclusions from plots of fault density against size is a practice that is not recommended.

The above exposition makes clear that the basic relationship is that as the size of the component grows, it will have more faults (or, alternatively, the likelihood of a fault increases). When one models a compound measure, such as fault density, against one of its components, meaningless conclusions can be easily drawn [37] and, in fact, ones that are in exact opposition to the basic relationship.

2.2 Threshold Effect

Hatton [25] has proposed a cognitive explanation as to why a threshold effect would exist between size[14] and faults.[15] The proposed theory is based on the human memory model, which consists of short-term and long-term memory. Hatton argues that Miller's work [33] shows that humans can cope with around seven (+/- two) pieces of information at a time in short-term memory, independent of information content. He then refers to [28] where they note that the contents of long-term memory are in a coded form and the recovery codes may get scrambled under some conditions. Short-term memory incorporates a rehearsal buffer that continuously refreshes itself. He suggests that anything that can fit into short-term memory is easier to understand and less fault prone. Pieces that are too large overflow, involving use of the more error-prone recovery code mechanism used for long-term storage.

11. The unit of observation for the Takahashi and Kamayachi study was the program rather than the component. However, our interest here is with their analysis which demonstrates that not finding a relationship between fault density and size may be simply due to a linear relationship between size and faults (not due to size having no effect on fault content).

12. Here, we took the midpoint of the smoking intensity variable shown in parantheses in the histogram.

13. Note that in our curve, we did not include the data for nonsmokers. Including that and plotting mortality per smoking rate would give an infinite increase in lung cancer mortality for nonsmokers.

14. Hatton uses the term "complexity" rather than size per se. However, he makes clear that he considers size to be a measure of complexity.

15. Hatton's model also suggests that the size of objects should fill up short-term memory in order to utilize it efficiently, and that failure to do so also leads to increased fault proneness. However, this aspect of his model is based on the types of analyses that we reviewed in Section 2.1 and showed that the conclusions from the observations are not sound. Therefore, we will not consider this aspect of Hatton's model further.

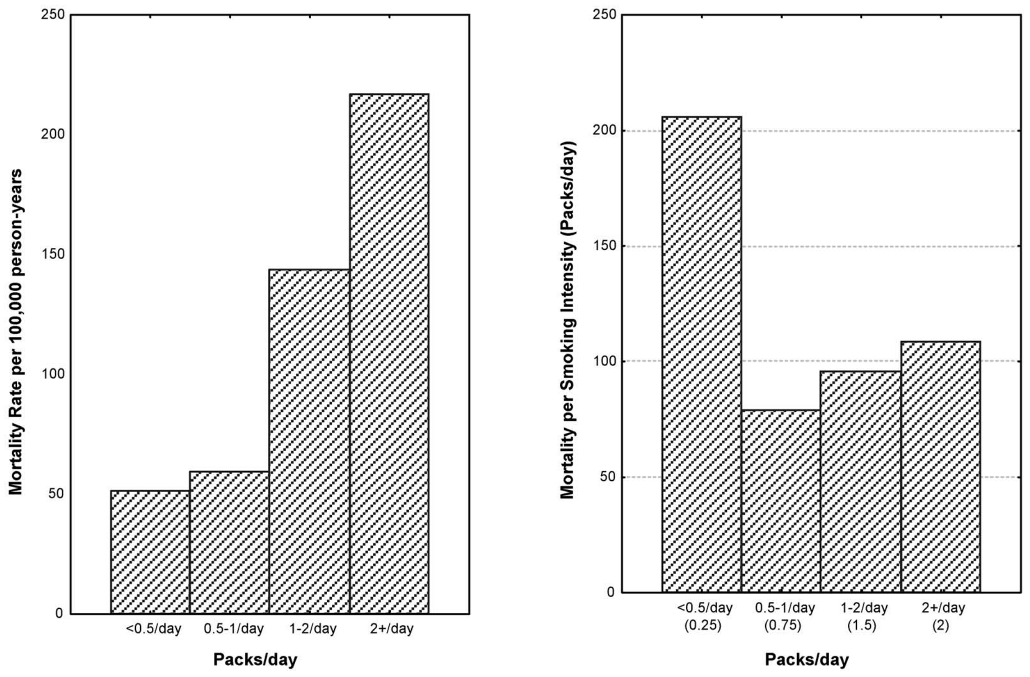


Fig. 4. Histogram showing the relationship between the intensity of smoking (packs/day) and the mortality rate (the source of the data is [21]). The histogram on the left consists of the original data set as presented by the original authors. The histogram on the right represents the data when the *y* variable has been turned into a density.

Hatton's theory states that a component of some size S fits entirely into short-term memory. If the component's size exceeds S, then short-term memory overflows. This makes the component less comprehensible because the recovery codes connecting comprehension with long-term memory break down. He then suggests a quadratic model that relates component size greater than S with the incidence of faults. In fact, Hatton's explanation has some supporters in earlier software engineering work. For instance, Woodfield et al. [50] state "we hypothesize that there is a limit on the mental capacity of human beings. A person cannot manipulate information efficiently when its amount is much greater than that limit."

Not all studies that looked at the relationship between size and faults found a threshold effect. For instance, the studies by Basili and Perricone [2] and Davey et al. [13] mentioned above did note that the components for which they had data tended to be small and, therefore, they could not observe the effect at large component sizes. One can consider the U-shaped curves identified by Withrow [49], Compton and Withrow [12], Moller and Paulish [34], and Kan [30] as identifying a threshold effect. However, as shown in the previous section, plotting fault density against size does not make too much sense.

In a subsequent article, Hatton [27] extends this model to object-oriented development. He argues that the concept of encapsulation, central to object-oriented development, lets us think about an object in isolation. If the size of this object is small enough to fit into short-term memory, then it will be easier to understand and reason about. Objects that are too large and overflow the short-term memory tend to be more fault prone.

There is ample evidence in the software engineering literature that the size of object-oriented classes is associated with fault-proneness. The relationship between size and faults is clearly visible in the study of [9], where the Spearman correlation was found to be 0.759 and statistically significant. Another study of image analysis programs written in C++ found a Spearman correlation of 0.53 between size in LOC and the number of errors found during testing [22] and was statistically significant at an alpha level of 0.05. Briand et al. [7] find statistically significant associations between six different size metrics and fault proneness for C++ programs, with a change in odds ratio going as high as 4.952 for one of the size metrics. None of these studies, however, was concerned with size thresholds and, therefore, did not test for them.

There has been interest in the industrial software engineering community with establishing thresholds for object-oriented measures. This was not driven by theoretical concerns, or a desire to test a theory, but rather with pragmatic ones. Thresholds provide a simple way for identifying problematic classes. If a class has a measurement value greater than the threshold, then it is flagged for investigation.

Lorenz and Kidd present a catalogue of thresholds [32] based on their experiences with C++ and Smalltalk projects. Most notable, for our purposes, are the thresholds they propose for size measures. For example, they set a maximum threshold of 20 methods in a class and three variables for nonuser-interface classes. Rosenberg et al. [38] present a series of thresholds for various object-oriented metrics, for instance, they suggest a size threshold of 40 methods per class. French [19] presents a procedure for computing thresholds and applies it to object-oriented projects in Ada95 and C++. In none of the above three works was a systematic validation performed to demonstrate that classes that exceeded the threshold size were indeed more problematic than those that do not, e.g., that they were more likely to contain a fault.

2.3 Summary

To our knowledge, there have been no studies that compute and validate size thresholds for object-oriented applications. This means that the human memory model proposed by Hatton and the practical thresholds derived by other researchers have not been empirically validated. Therefore, the purpose of our study was to test the size threshold effect theory.

It should be noted that if the size threshold theory is substantiated, this could have important implications. Given that its premise is cognitive, it would be expected that similar thresholds will hold across professional programmers and designers, and hence have broad generalizability.

3 Research Method

3.1 Measurement

3.1.1 Size Measures

The size of a class can be measured in a number of different ways. Briand et al. [7] have summarized some common size measures for object-oriented systems, and these consist of:[16]

- **Stmts:** The number of declaration and executable statements in the methods of a class. This can also be generalized to a simple SLOC count.
- **NM (Number of Methods):** The number of methods implemented in a class.
- **NAI (Number of Attributes):** The number of attributes in a class (excluding inherited ones). Includes attributes that are basic types such as strings and integers

As noted earlier, our initial study was performed on one C++ telecommunications system and repeated twice; once on another C++ system and once on a Java application. The studies were performed seperately over a period of 18 months and were performed under different constraints. We were therefore not able to collect all of the size measures for all systems. For instance, for the Java system the objective was to focus on measures that could be collected at design time only, which excludes Stmts and SLOC. For each of the three systems, we present below the subset of size measures that we collected.

3.1.2 Fault Measurement

In the context of building quantitative models of software faults, it has been argued that considering faults causing field failures is a more important question to address than faults found during testing [4]. In fact, it has been argued that it is the *ultimate* aim of quality modeling to identify postrelease fault proneness [16]. In at least one study, it was found that prerelease fault proneness is not a good surrogate measure for postrelease fault proness, the reason posited being that prerelease fault proneness is a function of testing effort [18].

Therefore, faults counted for all the systems that we studied were due to field failures occuring during actual usage. For each class, we characterized it as either *faulty* or *not faulty*. A faulty class had at least one fault detected during field operation. Distinct failures that are traced to the same fault are counted as a single fault.

3.2 Data Sources

Our study was performed on three object-oriented systems. It was initially done on C++ System 1 and then repeated on the subsequent two. These data sources are described below.

3.2.1 C++ System 1

This is a telecommunications system developed in C++ and has been in operation for approximately seven years. This system has been deployed around the world in multiple sites. In total six different developers had worked on its development and evolution. It consists of 83 different classes, all of which we analyzed.

Since the system has been evolving in functionality over the years, we selected a version for analysis where reliable fault data could be obtained. Fault data was collected from the configuration management system. This documented the reason for each change made to the source code and, hence, it was easy to identify which changes were due to faults. We focused on faults that were due to failures reported from the field. In total, 53 classes had one or more fault in them that was attributed to a field failure.

3.2.2 C++ System 2

Our data set comes from a telecommunications framework written in C++ [39]. The framework implements many core design patterns for concurrent communication software. The communication software tasks provided by this framework include event demultiplexing and event handler dispatching, signal handling, service initialization, interprocess communication, shared memory management, message routing, dynamic (re)configuration of distributed services, and concurrent execution and synchronization. The framework has been used in applications such as electronic medical imaging systems, configurable telecommunications systems, high-performance real-time CORBA, and web servers. Examples of its application include in the Motorola Iridium global personal communications system [40] and in network monitoring applications for telecommunications switches at Ericsson [41]. A total of 174 classes from the framework that were being reused in the development of commercial switching software constitute the system that we study. A total of 14 different programmers were involved in the development of this set of classes.[17]

For this product, we obtained data on the faults found in the library from actual field usage. Each fault was due to a unique field failure and represents a defect in the program

16. The number of methods and attributes can be further decomposed into private and public.

17. This number was obtained from the different login names associated with each class within the version control system.

TABLE 1
Size Measures Collected for the Different Systems

System	Size Measures			
	Stmts	**LOC**	**NM**	**NAI**
C++ System 1	X			
C++ System 2		X		
Java System			X	X

that caused the failure. Failures were reported by the users of the framework. The developers of the framework documented the reasons for each delta in the version control system and it was from this that we extracted information on whether a class was faulty. A total of 192 faults were detected in the framework at the time of writing. These faults occurred in 70 out of 174 classes.

3.2.3 Java System

This system is a commercial Java application. The application implements a word processor that can either be used stand-alone or embedded as a component within other larger applications. The word processor provides support for formatted text at the word, paragraph, and document levels, allows the definition of groupings of formatting elements as styles, supports RTF and HTML external file formats, allows spell checking of a range of words on demand, supports images embedded within documents or pointed to through links, and can interact with external databases.

This application was fielded and feedback was obtained from its users. This feedback included reports of failures and change requests for future enhancements. The application had a total of 69 classes. The size measures were collected from an especially developed static analysis tool. No Java inner classes were considered. For each class it was known how many field failures were associated to a fault in that class. In total, 27 classes had faults.

3.2.4 Size Measures

The size measures that were collected for each of the above three systems are summarized in Table 1.

Across the three systems, we covered different types of size measures. The measures STMTS and LOC are commonly only available from the source code. However, NM and NAI can be easily obtained from design documents. If we obtain consistent results across the systems and measures, then we can have greater confidence in the generalizability of our conclusions.

3.3 Analysis Method

The method that we use to perform our analysis is logistic regression. Logistic regression (LR) is used to construct models when the dependent variable is binary, as in our case. The general approach we use is to construct a LR model with no threshold, and a LR model with a threshold, and then compare the two models. This is a standard technique for evaluating LR models [29].

3.3.1 Logistic Regression

The general form of an LR model is:[18]

$$\pi = \frac{1}{1 + e^{-(\beta_0 + \beta_1 size)}}, \qquad (2)$$

where π is the probability of a class having a fault. The β parameters are estimated through the (unconditional)[19] maximization of a log-likelihood [29].

Collinearity. Collinearity is traditionally seen as being concerned with dependencies amongst independent variables. The models that we build in our study all involve only one independent variable, namely, size. However, it is known that collinearities can exist between the independent variables and the intercept [44]. Belsley et al. [3] propose the *condition number* as a collinearity diagnostic for the case of ordinary least-squares regression. This diagnostic has been extended specifically to the case of LR models [14], [47] by capitalizing on the analogy between the independent variable cross-product matrix in least-squares regression to the information matrix in maximum likelihood estimation and, therefore, it would certainly be parsimonious to use the latter. Based on a number of experiments, Belsley et al. suggest that a condition number greater than 30 indicates mild to severe collinearity.

Hypothesis Tests. The next task in evaluating the LR model is to determine whether the regression parameter is different from zero, i.e., test $H_0 : \beta_1 = 0$. This can be achieved by using the likelihood ratio G statistic [29]. One first determines the log likelihood for the model with the constant term only and denotes this l_0 for the "null" model. Then, the log likelihood for the full model with the size parameter is determined and denoted as l_S. The G statistic is given by $2(l_s - l_0)$ which has a X^2 distribution with one degree of freedom.[20]

Goodness of Fit. In previous studies with object-oriented measures another descriptive statistic has been used, namely, an R^2 statistic that is analogous to the multiple coefficient of determination in least-squares regression [29]. It should be recalled that this descriptive statistic will in

18. We also evaluated log and quadratic models in the logit. The conclusions were the same and, therefore, we present the results for the linear model only.

19. Conditional logistic regression is used when matching of observations was performed during the study and each matched set is treated as a stratum in the analysis [6].

20. Note that we are not concerned with testing whether the intercept is zero or not since we do not draw substantive conclusions from the intercept. If we were, we would use the log-likelihood for the null model which assigns a probability of 0.5 to each response.

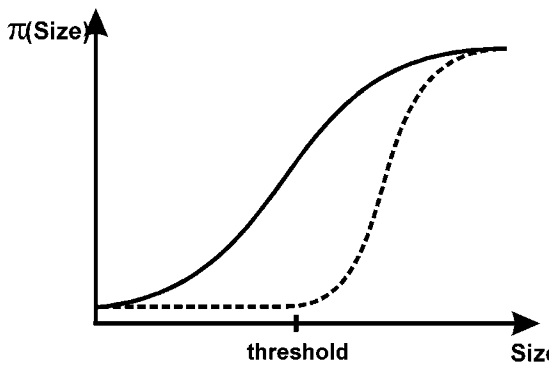


Fig. 5. Relationship between size and the probability of a fault for the threshold and no-threshold models.

general have low values compared to what one is accustomed to in a least-squares regression.

Influence Analysis. Influence analysis is performed to identify influential observations (i.e., ones that have a large influence on the LR model). Pergibon has defined the $\Delta\beta$ diagnostic [36] to identify influential groups in logistic regression. We use the $\Delta\beta$ diagnostic in our study to identify influential groups of observations. For groups that are deemed influential we investigate this to determine if we can identify substantive reasons for them being overly influential. In all cases in our study where a large $\Delta\beta$ was detected, its removal, while affecting the estimated coefficients, did not alter our conclusions.

3.3.2 Model With A Threshold

A LR model with a threshold can be defined as [46]:

$$\pi = \frac{1}{1 + e^{-(\beta_0 + \beta_1 (size-\tau) I_+(size-\tau))}}, \quad (3)$$

where

$$I_+(z) = \begin{cases} 0 & if\ z \leq 0 \\ 1 & if\ z > 0 \end{cases}, \quad (4)$$

and τ is the size threshold value. The difference between the no-threshold and threshold model is illustrated in Fig. 5. For the threshold model, the probability of a fault only starts to increase once the size is greater than the threshold value, τ.[21]

To estimate the threshold value, τ, one can maximize the log likelihood for the model in (3). Ulm [46] presents an algorithm for performing this maximization.

Once a threshold is estimated, it should be evaluated. This is done by comparing the no-threshold model with the threshold model. Such a comparison is, as is typical, done using a likelihood ratio statistic (for example see [29]). The null hypothesis being tested is:

$$H_0 : \tau \leq size^{(1)} = \min size, \quad (5)$$

21. This type of model has been used in epidemiological studies, for example, to evaluate the threshold of dust concentrations in coal mines above which miners develop chronic bronchitic reactions [46]. In fact, the general approach can be applied to investigate any dose-response relationship that is postulated to have a threshold.

where $size^{(1)}$ is the smallest size value in the data set. If this null hypothesis is not rejected, then it means that the threshold is equal to or below the minimal value. In the latter case, this is exactly like saying that the threshold model is the same as the no-threshold model. In the former case, the threshold model will be very similar to the no-threshold model since only a small proportion of the observations will have the minimal value. Hence, one can conclude that there is no threshold. The alternative hypothesis is:

$$H_1 : \tau > size^{(1)}, \quad (6)$$

which would indicate that a threshold effect exists.

The likelihood ratio statistic is computed as $-2(ll(H_0) - ll(H_1))$, where $ll(\cdot)$ is the log-likelihood for the given model. This can be compared to a X^2 distribution with one degree of freedom. We use an α level of 0.05. Ulm [46] has performed a Monte Carlo simulation to evaluate this test and subsequently recommended its use.

It should be noted that, if $\hat{\tau}$ is equal or close to $size^{(n)}$ (the largest size value in the data set), this would mean that most of the observations in the data set would have a value of zero, making the estimated model parameters unstable. In such a case, we conclude that no threshold effect was found for this data set. Ideally, if a threshold exists then it should not be too close to the minimum or maximum values of size in the data set.

4 Results

4.1 Descriptive Statistics

The plots in Figs. 6, 7, and 8 show the dispersion and central tendency of size for the three systems (a description of what these box and whisker plots mean is provided in the appendix). One thing that is noticeable from these plots is that C++ System 1 tends to have larger classes than C++ System 2. This is further emphasized when we consider that a statement is sometimes over multiple source lines. Hence, if we measure size for both systems on the same scale, it would become more obvious that System 1 is much larger.

Another point to note is that, for each of the three systems, there are extreme outliers. Although the outliers are not necessarily influential observations, they do provide support for being prudent and performing diagnostics for influential observations.

The box and whisker plots in Figs. 9, 10, 11, and 12 contrast the size of faulty and not-faulty classes. It is clear from these plots that faulty classes tend to be larger than not-faulty classes. In some cases, as in the Java system, more substantially so.

4.2 Testing For Threshold Effects

The results of testing for threshold effects are presented in Table 2 for the four size measures across the three systems. The table shows the parameters for the no-threshold model (2), the threshold model (3), and their comparison. The final set of columns show the estimated threshold value, the chi-square value from comparing the two models, and its p-value. Even though a threshold value was estimated, the threshold model must have a

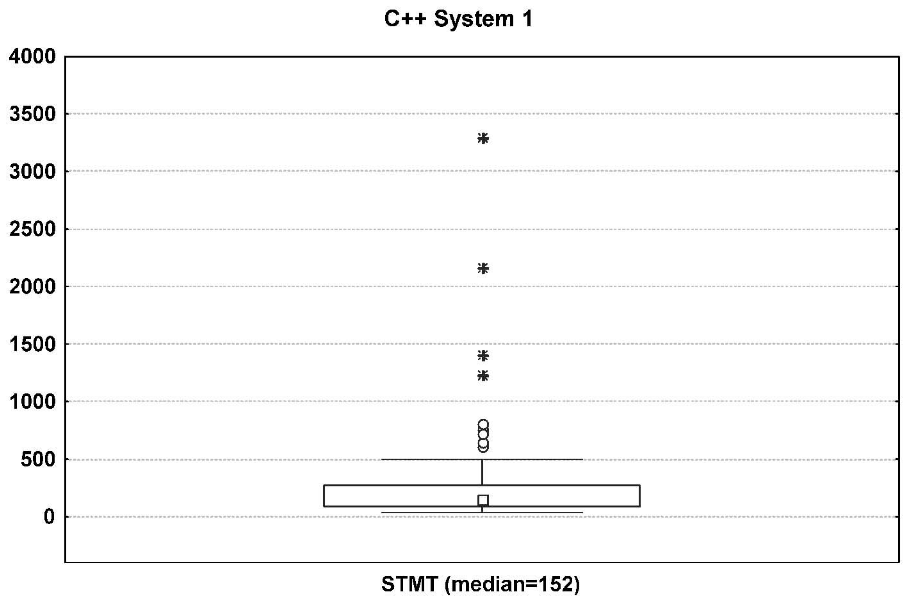


Fig. 6. Dispersion and central tendency of the size measure for C++ System 1.

statistically significant difference from the no-threshold model in order to reject the null hypothesis. Note that the condition numbers are not shown, although they never exceeded three, indicating that there are no collinearity problems in any of the models.

A number of general trends can be observed from this table. First, for all the threshold and no-threshold models, size has a positive parameter and is always statistically significant. This is not surprising and is consistent with previous results. Second, the R^2 measures tend to run low. However, as noted earlier for logistic regression, this is common and also one should not expect that a simple model with only size will provide substantial explanatory power (for our purposes of testing the threshold hypothesis, however, this size only model is all that is required). Finally, in none of the comparisons performed was there a difference between the threshold and no-threshold models. This indicates that there are no threshold effects.

4.3 Discussion

The results that we have presented above are unambiguous. We obtained consistent results across three different industrial object-oriented systems developed by different teams in different locations. Furthermore, the results were for four different measures of class size.

It is clear from the evidence that there is no class size threshold effect on postrelease fault-proneness. Therefore, the theory that states that the fault proneness of classes remains stable until a certain size and then increases due to limitations in short-term memory is without support. The

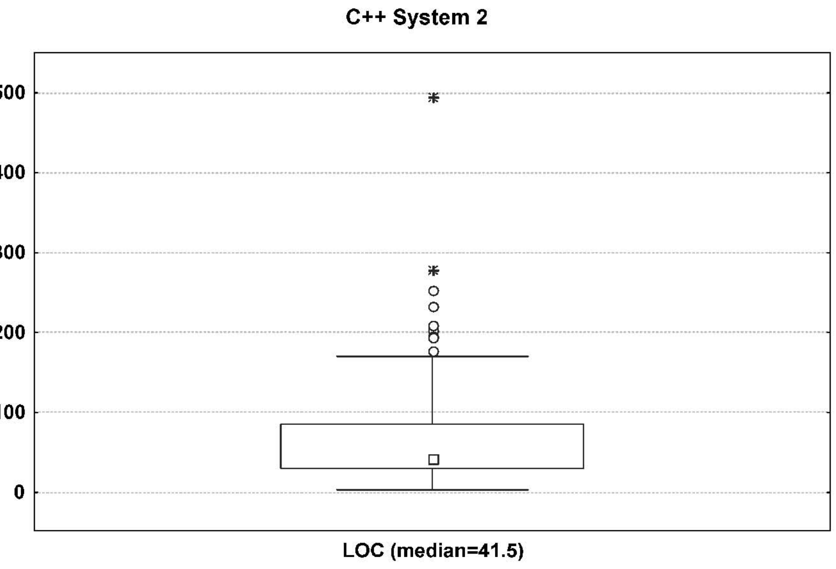


Fig. 7. Dispersion and central tendency of the size measure for C++ System 2.

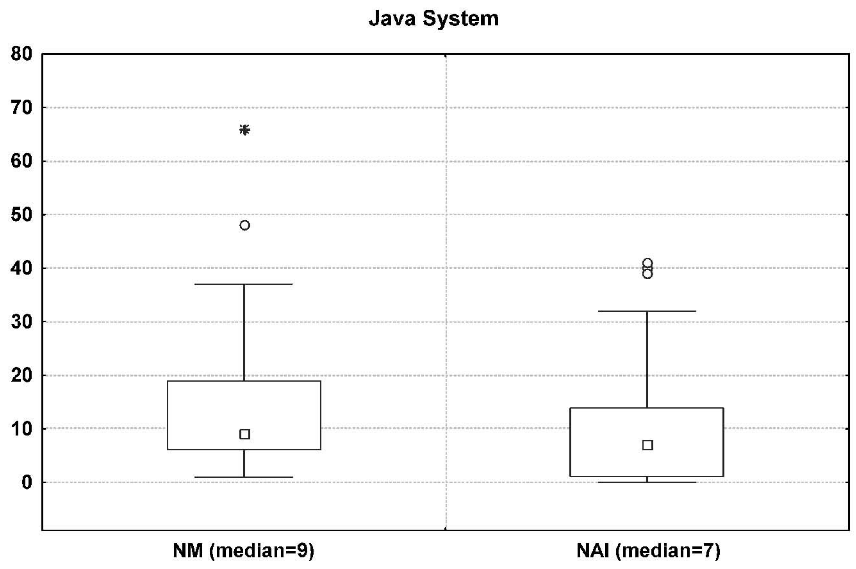


Fig. 8. Dispersion and central tendency of the size measure for the Java system.

only evidence that we can present is that there is a continuous relationship between size and fault proneness. This means that, as class size increases, so will its fault proneness.

This latter conclusion needs to be qualified, however. It has been argued that the relationship between size and postrelease faults is a function of test intensity [18]. If components that are larger are subjected to more testing, then there will be a positive relationship between size and prerelease faults. However, the larger components will have few faults postrelease, thus suggesting that the relationship between size and postrelease faults would be considerably weakened.

For the three object-oriented systems studied, we found a consistent positive relationship between size and postrelease fault proneness (as evidenced by the positive regression parameters for the no-threshold models). The same positive association was found in a recent study [9]. Therefore, we can only speculate that, for these systems, extra care was *not* given to large classes during development and testing. To the contrary, due to their size, it was not possible to attain high test coverage for the larger classes.

Our results should not imply that size is the only variable that can be used to predict fault proneness for object-oriented software. To the contrary, while size seems to be an important variable, other factors will undoubtedly also have an influence. Therefore, for the purpose of building comprehensive models that predict fault proneness, more variables than size should be used.

Furthermore, we do not claim that the existing size thresholds that have been derived from experiential knowledge, such as those of Lorenz and Kidd [32] and Rosenberg et al. [38], are of no practical utility in light of our findings.

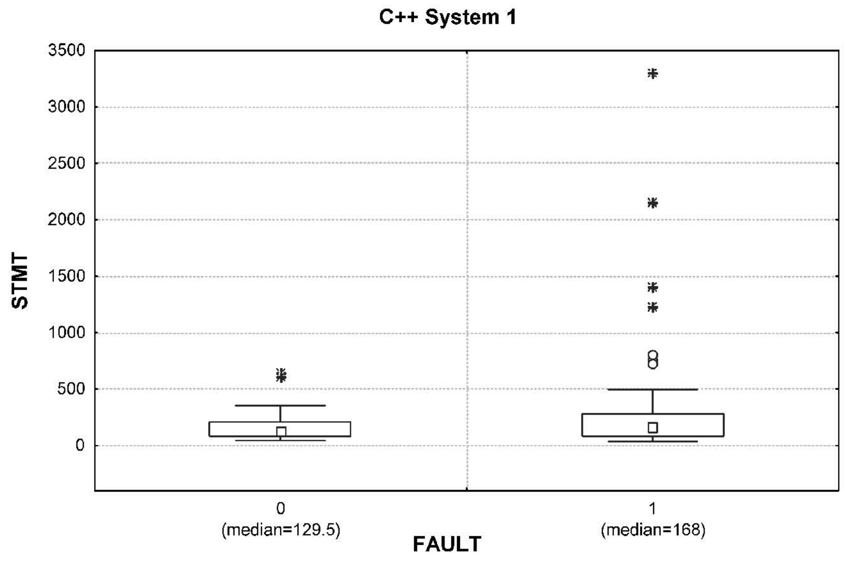


Fig. 9. Dispersion and central tendency for faulty (1) and nonfaulty (0) classes in C++ System 1.

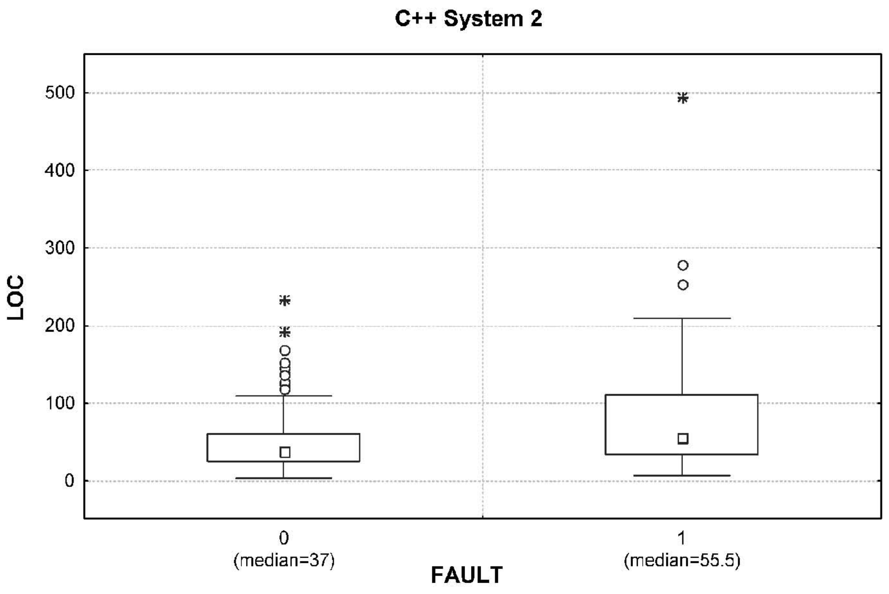


Fig. 10. Dispersion and central tendency for faulty (1) and nonfaulty (0) classes in C++ System 2.

Even if there is a continuous relationship between size and fault proneness as we have found, if you draw a line at a high value of size and call this a threshold, classes that are above the threshold will still be the *most fault-prone*. This is illustrated in the left panel of Fig. 13. Therefore, for the purpose of identifying the most fault-prone classes, such thresholds will likely work. But, it will be noted that class size below the threshold can still mean high fault proneness, just not the highest.

Had a threshold effect been identified, then class size below the threshold represents a "safe" region whereby designers deliberately restricting their classes within this region can have some assurance that the classes will have, everything else being equal, *minimal fault proneness*. This is illustrated in the right panel of Fig. 13.

4.4 Limitations

It is plausible that the three systems we studied had class sizes that were always larger than a true threshold and, hence, we did not identify any threshold effects even though they exist. While the strength of this argument is diluted because it would have to be true for all three systems developed by three different teams in different countries, it cannot be discounted without further studies.

In our study, we utilized a specific threshold model. With no prior work, this seems like a reasonable threshold model to use since it captures the theoretical claims made for threshold effects. However, we encourage other researchers to critique and improve this threshold model. Perhaps with an improved model of a threshold effect, thresholds will be identified. Therefore, while our results

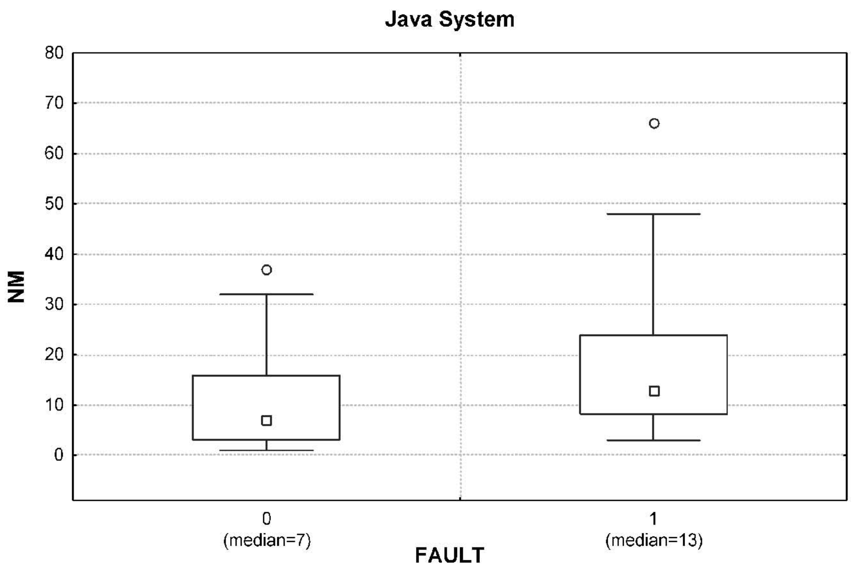


Fig. 11. Dispersion and central tendency for faulty (1) and nonfaulty (0) classes in the Java System and the NM size measure.

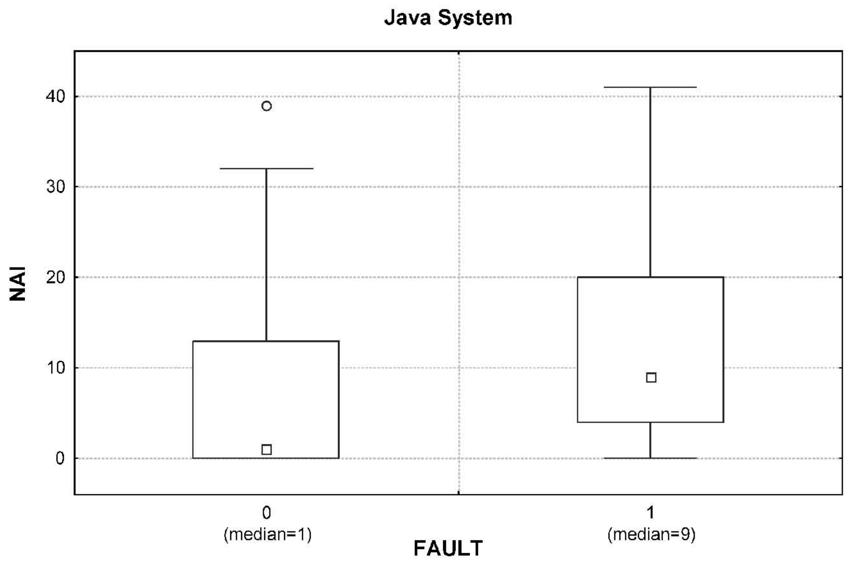


Fig. 12. Dispersion and central tendency for faulty and nonfault classes in the Java System and the NAI size measure.

are clear, we do not claim that this is the last word on size thresholds for object-oriented software. Rather, we hope this study will catalyze interest in size thresholds. After all, the relationship between size and fault proneness can be considered as one of the basic ones in software engineering. We should build a solid understanding of it if we ever hope to have a science of software quality.

5 Conclusions

Every scientific discipline develops theories to explain phenomena that are observed. Theories with strong empirical support also have good predictive power, in that their predictions will match reality closely. Software quality theories are important in that they can help us understand the factors that have an impact on quality, but also they can have considerable practical utility in providing guidance for ensuring reliable software.

The theory that was the focus of our study concerned the optimal size of object-oriented classes, the so-called "Goldilocks Conjecture." This conjecture was posited initially for procedural software based on empirical observations by a number of researchers. It states that components below the optimal size are more likely to contain a fault and, so, are above the optimal size. In fact, the conjecture has been extended to state that this is applicable irrespective of the programming language that is used.

If true, the implications of this theory are important. First, it suggests that smaller components are more likely to contain a fault. It follows that program decomposition, an axiom of software engineering, is bad practice. Second, that designers and developers should ensure that their

TABLE 2
Threshold and No-Threshold Model Results and Their Comparison for the Three Systems and Four Size Measures

System	Size Measure	No Threshold Model			Threshold Model			Comparison of Models	
		R^2	β_1 Coefficient (s.e.)	p-value	R^2	β_1 Coefficient (s.e.)	p-value	Estimated Threshold Value	Chi-Square (p-value)
C++ System 1	STMT	0.040	0.001915 (0.001261)	0.036	0.061	0.1061 (0.3743)	0.01	671	2.256 (0.133)
C++ System 2	SLOC	0.0578	0.01075 (0.003274)	0.00022	0.0578	0.01075 (0.003274)	0.00022	3	–[28]
Java System	NM	0.07	0.0571 (0.02464)	0.0103	0.07	0.0571 (0.02464)	0.0103	1	–[29]
	NAI	0.037	0.04347 (0.02423)	0.0631	0.0416	7.3956 (31.1326)	0.0499	39[30]	0.39 (0.53)

For all models the condition number was found to be always less than three, indicating that collinearity was not a problem. 28. For the threshold model, the threshold value is equal to the minimum class size. Therefore, there will be no difference between the threshold and no-threshold model. 29. The threshold value is equal to the minimum value for NM, therefore there is essentially no difference between the threshold and no-threshold models. 30. For this model, the threshold of 39 was very close to the maximum value of NAI such that there were only two observations that were nonzero. Therefore, this threshold model is rather unstable.

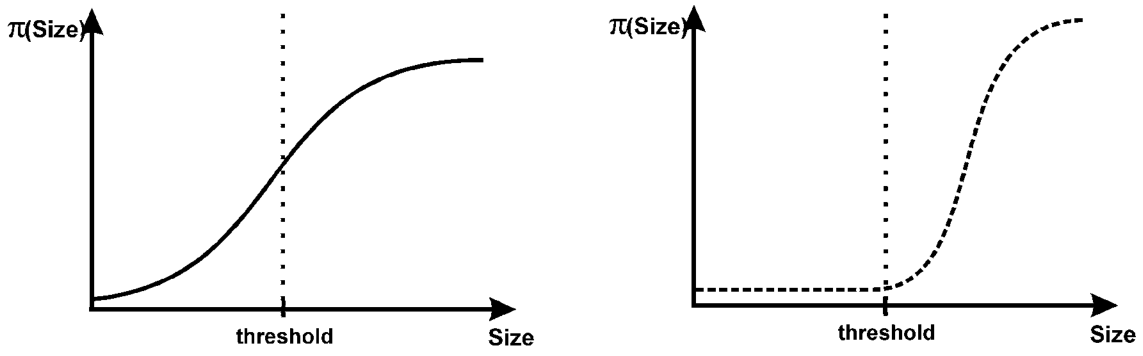


Fig. 13. Different types of thresholds.

components are not too large; otherwise, their reliability will deteriorate. So pursuasive was this conjecture, that a cognitive model was proposed to explain it. Recently, this theory has encroached into the object-oriented area.

We first showed that the claim of smaller components or classes being more fault prone than larger ones is a mathematical artifact; a consequence of the manner in which previous researchers analyzed their data. In fact, if we apply the same logic as those who made such a claim, then our understanding of many medical phenomena would have to be reversed.

We then performed an empirical study to test the claim that there exists a threshold class size, above which the fault proneness of classes increases rapidly. The study was performed on three object-oriented systems using different size measures. Our results provide no support for the threshold theory.

Perhaps most surprisingly, it is clear that even such a basic relationship as the one between size and faults is not well understood by the software engineering community. At least for object-oriented systems, our study may be considered as a contribution to help improve this understanding.

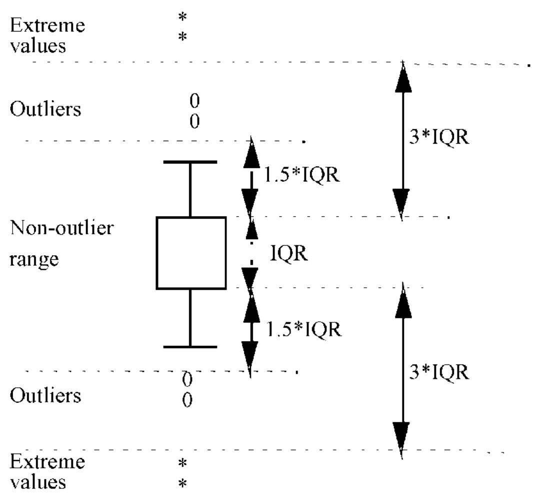


Fig. 14. Description of a box and whisker plot.

Appendix A
Box and Whisker Plots

In this paper, box and whisker plots are used quite frequently. This brief appendix is intended to explain how to interpret such a diagram.

Box and whisker plots are used to show the variation in a particular variable. Fig. 14 shows how such a plot is constructed. The box represents the interquartile range (IQR). The IQR bounds the 25th and 75th percentiles. The 25th percentile is the value of the variable where 25 percent or less of the observations have equal or smaller values. The same is for the 75th percentile. The whiskers are the largest values within 1.5 times the size of the box. This value of 1.5 is conventional. Outliers are within 1.5 times the size of the box beyond the whiskers and extremes are beyond the outliers. Finally, usually there is a dot in the box. This dot would be the median, or the 50th percentile.

Acknowledgments

We wish to thank Hakan Erdogmus, Anatol Kark, David Card, and Janice Singer for their comments on an earlier version of this paper. We also wish to thank the anonymous reviewers of *IEEE Transactions on Software Engineering* for their thoughtful and detailed feedback on an earlier version of this paper.

References

[1] L. Arthur, *Rapid Evolutionary Development: Requirements, Prototyping and Software Creation.* John Wiley and Sons, 1992.
[2] V. Basili and B. Perricone, "Software Errors and Complexity: An Empirical Investigation," *Comm. ACM,* vol. 27, pp. 42-52, 1984.
[3] D. Belsley, E. Kuh, and R. Welsch, *Regression Diagnostics: Identifying Influential Data and Sources of Collinearity.* John Wiley and Sons, 1980.
[4] A. Binkley and S. Schach, "Validation of the Coupling Dependency Metric as a Predictor of Run-Time Failures and Maintenance Measures," *Proc. 20th Int'l Conf. Software Eng.,* 1998.
[5] J. Bowen, "Module Size: A Standard or Heuristic?," *J. Systems and Software,* vol. 4, pp. 327-332, 1984.
[6] N. Breslow and N. Day, *Statistical Methods in Cancer Research—vol. 1—The Analysis of Case Control Studies,* International Agency for Research on Cancer, 1980.
[7] L. Briand, J. Wuest, J. Daly, and V. Porter, "Exploring the Relationships between Design Measures and Software Quality in Object Oriented Systems," *J. Systems and Software,* vol. 51, pp. 245-273, 2000.

[8] D. Card and R. Glass, *Measuring Software Design Quality.* Prentice-Hall, 1990.

[9] M. Cartwright and M. Shepperd, "An Empirical Investigation of an Object-Oriented Software System," *IEEE Trans. Software Eng.,* vol. 26, pp. 786 -796, 2000.

[10] F. Chayes, *Ratio Correlation: A Manual for Students of Petrology and Geochemistry.* The University of Chicago Press, 1971.

[11] S. Chidamber and C. Kemerer, "A Metrics Suite for Object-Oriented Design," *IEEE Trans. Software Eng.,* vol. 20, pp. 476-493, 1994.

[12] B. Compton and C. Withrow, "Prediction and Control of Ada Software Defects," *J. Systems and Software,* vol. 12, pp. 199-207, 1990.

[13] S. Davey, D. Huxford, J. Liddiard, M. Powley, and A. Smith, "Metrics Collection In Code and Unit Test as Part of Continuous Quality Improvement," *Software Testing, Verification, and Reliability,* vol. 3, pp. 125-148, 1993.

[14] C. Davies, J. Hyde, S. Bangdiwala, and J. Nelson, "An Example of Dependencies Among Variables in a Conditional Logistic Regression," *Modern Statistical Methods in Chronic Disease Edpidemiology,* S. Moolgavkar and R. Prentice eds., John Wiley and Sons, 1986.

[15] R. Dunn and R. Ullman, "Modularity Is Not a Matter of Size," *Proc. 1979 Ann. Reliability and Maintainability Symp.,* 1979.

[16] N. Fenton and M. Neil, "Software Metrics: Successes, Failures, and New Directions," *J. Systems and Software,* vol. 47, pp. 149-157, 1999.

[17] N. Fenton and M. Neil, "A Critique of Software Defect Prediction Models," *IEEE Trans. Software Eng.,* vol. 25, pp. 676-689, 1999.

[18] N. Fenton and N. Ohlsson, "Quantitative Analysis of Faults and Failures in a Complex Software System," *IEEE Trans. Software Eng.,* vol. 26, pp. 797-814, 2000.

[19] V. French, "Establishing Software Metrics Thresholds," *Proc. Ninth Int'l Workshop Software Measurement,* 1999.

[20] J. Gaffney, "Estimating the Number of Faults in Code," *IEEE Trans. Software Eng.,* vol. 10, pp. 459-464, 1984.

[21] L. Gordis, *Epidemiology.* W.B. Saunders, 1996.

[22] R. Harrison, L. Samaraweera, M. Dobie, and P. Lewis, "An Evaluation of Code Metrics for Object-Oriented Programs," *Information and Software Technology,* vol. 38, pp. 443-450, 1996.

[23] L. Hatton, "Unexpected (and Sometimes Unpleasant) Lessons from Data in Real Software Systems," *Safety and Reliability of Software Based Systems, 12th Ann. CSR Workshop,* 1995.

[24] L. Hatton, "Is Modularization Always a Good Idea?," *Information and Software Technology,* vol. 38, pp. 719-721, 1996.

[25] L. Hatton, "Re-Examining the Fault Density—Component Size Connection," *IEEE Software,* pp. 89-97, 1997.

[26] L. Hatton, "Software Failures—Follies and Fallacies," *IEE Review,* vol. 43, pp. 49-52, 1997.

[27] L. Hatton, "Does OO Sync with How We Think?," *IEEE Software,* pp. 46-54, 1998.

[28] E. Hilgard, R. Atkinson, and R. Atkinson, *Introduction to Psychology,* Harcourt Brace Jovanovich, 1971.

[29] D. Hosmer and S. Lemeshow, *Applied Logistic Regression.* John Wiley and Sons, 1989.

[30] S. Kan, *Metrics and Models in Software Quality Engineering.* Addison-Wesley, 1995.

[31] R. Lind and K. Vairavan, "An Experimental Investigation of Software Metrics and Their Relationship to Software Development Effort," *IEEE Trans. Software Eng.,* vol. 15, pp. 649-653, 1989.

[32] M. Lorenz and J. Kidd, *Object-Oriented Software Metrics.* Prentice-Hall, 1994.

[33] G. Miller, "The Magical Number 7 Plus or Minus Two: Some Limits on Our Capacity for Processing Information," *Psychological Review,* vol. 63, pp. 81-97, 1957.

[34] K.-H. Moller and D. Paulish, "An Empirical Investigation of Software Fault Distribution," *Proc. First Int'l Software Metrics Symp.,* 1993.

[35] D. O'Leary, "The Relationship Between Errors and Size in Knowledge-Based Systems," *Int'l J. Human-Computer Studies,* vol. 44, pp. 171-185, 1996.

[36] D. Pergibon, "Logistic Regression Diagnostics," *The Annals of Statistics,* vol. 9, pp. 705-724, 1981.

[37] J. Rosenberg, "Some Misconceptions About Lines of Code," *Proc. Fourth Int'l Software Metrics Symp.,* 1997.

[38] L. Rosenberg, R. Stapko, and A. Gallo, "Object-Oriented Metrics for Reliability," *IEEE Int'l Symp. Software Metrics,* 1999.

[39] D. Schmidt, "Using Design Patterns to Develop Reusable Object-Oriented Communication Software," *Comm. ACM,* vol. 38, pp. 65-74, 1995.

[40] D. Schmidt, "A System of Reusable Design Patterns for Communication Software," *The Theory and Practice of Object Systems,* S. Berzuk ed., 1995.

[41] D. Schmidt and P. Stephenson, "Experiences Using Design Patterns to Evolve System Software Across Diverse OS Platforms," *Proc. Ninth European Conf. Object Oriented Programming,* 1995.

[42] R. Selby and V. Basili, "Analyzing Error-Prone System Structure," *IEEE Trans. Software Eng.,* vol. 17, pp. 141-152, 1991.

[43] V. Shen, T.-J. Yu, S. Thebaut, and L. Paulsen, "Identifying Error-Prone Software—An Empirical Study," *IEEE Trans. Software Eng.,* vol. 11, pp. 317-324, 1985.

[44] S. Simon and J. Lesage, "The Impact of Collinearity Involving the Intercept Term on the Numerical Accuracy of Regression," *Computer Science in Economics and Management,* vol. 1, pp. 137-152, 1988.

[45] M. Takahashi and Y. Kamayachi, "An Empirical Study Of A Model For Program Error Prediction," *Proc. Eighth Int'l Conf. Software Eng.,* 1985.

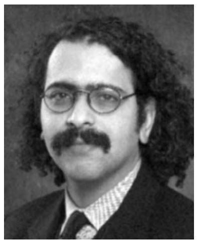
[46] K. Ulm, "A Statistical Method for Assessing A Threshold in Epidemiological Studies," *Statistics in Medicine,* vol. 10, pp. 341-349, 1991.

[47] Y. Wax, "Collinearity Diagnosis for Relative Risk Regression Analysis: An Application to Assessment of Diet-Cancer Relationship in Epidemiological Studies," *Statistics in Medicine,* vol. 11, pp. 1273-1287, 1992.

[48] N. Wilde, P. Matthews, and R. Huitt, "Maintaining Object-Oriented Software," *IEEE Software,* pp. 75-80, 1993.

[49] C. Withrow, "Error Density and Size in Ada Software," *IEEE Software,* pp. 26-30, 1990.

[50] S. Woodfield, V. Shen, and H. Dunsmore, "A Study of Several Metrics for Programming Effort," *J. Systems and Software,* vol. 2, pp. 97-103, 1981.



Khaled El Emam obtained the PhD degree from the Department of Electrical and Electronics Engineering, King's College, the University of London (UK) in 1994. He is currently a research officer at the National Research Council in Ottawa, Canada's primary applied research organization. He is a coeditor of ISO's project to develop an international standard defining the software measurement process (ISO/IEC 15939) and is leading the software engineering process area in the IEEE's project to define the Software Engineering Body of Knowledge. He has also coedited two books on software process, both published by the IEEE CS Press; he is an adjunct professor at both the School of Computer Science at McGill University and the Department of Computer Science at the University of Quebec at Montreal; and is president of a software company specializing in online analytics. Currently, he is a resident affiliate at the Software Engineering Institute in Pittsburgh. Khaled is on the editorial boards of *IEEE Transactions on Software Engineering* and the *Empirical Software Engineering Journal*. Previously, he was the International Trials Coordinator for the SPICE Trials, where he was leading the empirical evaluation of the emerging process assessment international standard, ISO/IEC 15504, world-wide; the head of the Quantitative Methods Group at the Fraunhofer Institute for Experimental Software Engineering in Germany; a research scientist at the Centre de Recherche Informatique de Montreal (CRIM) in Canada; a researcher in the Software Engineering Laboratory at McGill University; and worked in a number of research and development projects for organizations such as Toshiba International Company and Honeywell Control Systems in the UK, and Yokogawa Electric in Japan. Over the last 10 years he has managed over half a dozen software development projects.

Saïda Benlarbi received the PhD degree in computer science from the University of Sherbrooke, Sherbrooke, and the MSc degree in computer science from Laval University, Quebec. She also holds an engineering degree in computer engineering and an associate degree in mathematics and physics, both from the Faculty of Sciences, University of Tunis, Tunisia. She is currently heading the System Reliability Engineering group at Alcatel, Carrier Internetworking Division in Kanata, Canada, where she is leading the reliability engineering activities on Alcatel CID networking products. Previously, she was heading the Software Engineering Division of Cistel Technology Inc. where she led the research and development activities. Previously, she worked as a research scientist at the Centre de Recherche Informatique de Montréal in the Software Engineering Group. Her research interests include networking systems/software dependability evaluation and prediction, statistical and adaptive prediction models for software measurement and software architecture and software reuse; definition and validation of software metrics, neural nets modeling, and supervised and nonsupervised learning in multilayered neural nets.

Nishith Goel received the Bachelor's degree in engineering from the University of Jodhpur, Jodhpur, India (1976), the MASc degree in electrical engineering (1978), and the PhD degree in systems design engineering from University of Waterloo, Canada (1983). He joined Nortel Networks in 1984 as a member of the scientific staff where he worked in various areas of semiconductor development, design, and manufacturing. In 1994, he joined the Software Engineering Assurance Lab (SEAL) at Nortel Networks, where he was engaged in the research and development of software engineering methods and tools for developing large software systems. Dr. Goel is currently president of Cistel Technology Inc., a company he founded in 1995, providing research and consulting in the areas of software engineering and information technology. He was industry cochair for the International Symposium for Software Reliability Engineering in 1999.

Walcelio Melo holds the following degrees in computer science: the BSc degree in 1983 from University of Brasilia, Brazil, the MSc degree in 1988 from the Federal University of Rio Grande do Sul, Brazil, and the PhD degree with high honors from the University of Grenoble—Universite Joseph Fourrier, France. He is a technology manager at Oracle. He leads the development of the Oracle's component-based development approach. He is also a knowledge area leader of Oracle Java professional community. At Oracle, he has acted as a senior technical consultant, software architect, and project manager in several of Oracle's projects in critical environments. He was also the software engineering quality practice manager for Oracle Brazil. He is also a professor in the Computer Science Department at the Catolica University of Brasilia, Brazil, where he teaches software engineering and software quality to undergraduate and graduate students. Before joining Oracle, Dr. Melo had been the software engineering lead researcher at CRIM (Centre de Recherche Informatique de Montreal) and adjunct professor in the School of Computer Science at McGill University, Montreal, Canada. Before that, Dr. Melo was a faculty research associate at the Institute for Advanced Computer Studies at University of Maryland, College Park, Maryland, and a member of the NASA Software Engineering Laboratory. His research interests are software measurement, software reuse, object technologies, and Java-based internet development methods and tools.

Hakim Lounis received the MS and PhD degrees in computer science from Paris-Sud University, Orsay, France. He is currently, professor in the Department of Computer Science of UQAM (Université du Québec à Montréal). After completing his PhD, he joined in 1996 the Computer Research Institute of Montreal (CRIM), where he was a full member of the research staff for the Software Engineering and Knowledge Engineering Group. From 1997 to 2000, he was also an adjunct professor at the University of Laval in Quebec city.

Shesh N. Rai is currently a faculty member at the Department of Biostatistics, St. Jude Children's Research Hospital, Memphis Tennessee. He is also an adjunct professor at the Department of Biostatistics and Community Medicine, the University of Ottawa, Ottawa, Canada. He has provided consulting services to Cistel Technology, Nepean, Canada to improve the software reliability (some of the collaborative results are published in the *Journal of Systems and Software*, *IEEE Transactions on Software Engineering*, and the Proceedings of the International Symposium on Software Reliability Engineering). He has worked at Statistics Canada, Ottawa, Canada where he developed procedures to identify vulnerable populations (at health risk) and he also developed an efficient procedure for the analysis of hierarchical models for complex survey data. Previously, he worked as a postdoctoral fellow (NSERC) at Health Canada, Ottawa, Canada, where he developed methods for characterization and analysis of uncertainty and variability in risk models. The methods proposed also allow for the complex interrelationships between subsets of variables in multivariate models. He has conducted detailed uncertainty analysis on two important practical problems: estimating proportion of the lung cancer burden attributable to radon in homes and estimating cancer risks due to trace levels of chemical contaminants in drinking water. His work is published in *Human and Ecological Risk Assessment* and *Risk Analysis* Journals. In his PhD thesis at the University of Waterloo, he developed methods for the analysis of interval censored failure time data, such as that arising in occult tumor trials (results are published in *Biometrics*, the *Canadian Journal of Statistics*, the *Journal of Royal Statistical Society*, and *Biometrical Journal*).