This article discusses evidence of the Pareto principle as it relates to the distribution of software defects in code. The authors look at evidence in both the context of the software test team and the software users. They also investigate two related principles. The first principle is that the distribution of defects in code relates to the distribution of complexity in code. The second hypothesis discusses how one defines complexity and whether that definition relates to the distribution of complexity in code. The authors present this work as an empirical study of three general hypotheses investigated for large production-level software. They show that the essence of the principle holds while precise percentages do not.

Key words: complexity, defects, empirical software engineering, errors, faults, Pareto principle, quantitative study, software metrics, software quality, testing, usage testing

SQP References
Predict Defect Test Densities Using Test Executions as Time Scale
  vol. 8, issue 1
  Dean F. Miller
Applying Quantitative Methods to Software Maintenance
  vol. 3, issue 1
  Ed Weller

INTRODUCTION
In 1906 Italian economist Vilfredo Pareto observed that 20 percent of the Italian population owned 80 percent of property in Italy (Pareto 1906). Then, in the late 1940s, management leader J. M. Juran suggested that a small number of causes determine most of the results in any situation (Juran 1951). Hence, the 80:20 rule was born. Since then, Juran’s assumption, the 80:20 rule, has been attributed (incorrectly) to the economist as “Pareto’s Principle” and applied to everything from assumptions about advertising effectiveness to the behavior of defects in software (Boehm and Basili 2001). In fact, Thomas McCabe investigated the rule for many project parameters and their impact on quality. He looked at the classes of defects that occurred most frequently and would lead to the errors that would have the highest severity impact (Schulmeyer and McManus 1997). Does this, however, mean that one can apply the principle to every software context? The concern is that this principle is used too liberally, and the evidence for its generalization in the software domain is not always shown. Many people have adopted such principles as conventional wisdom.

To investigate this principle the authors look at two categories of hypotheses related to the Pareto principle for software. These refer to the defects and their relationship to code characteristics, specifically code complexity, code size, and the number of files. They are:

1. Approximately 80 percent of defects occur in 20 percent of the code.
2. Approximately 80 percent of complexity occurs in 20 percent of the code.

The second category of hypotheses seeks to explain the findings of the first category. Such claims propose that minor variances in the parameters used to define code characteristics do not affect the defect-concentration relationships. Of this second category, the authors are particularly interested in statements of relationships between defect distribution and code complexity. This is because such statements usually do not define complexity. They therefore investigate the following hypothesis:

3. The relationship between the distribution of defects and the complexity of code does not depend on the complexity measure in use.

The authors analyze a large industrial production-level application. There are few industrially validated empirical studies on large production-level systems. Ostrand and Weyuker (2002) theorized that the reasons for this relate to: 1) the high degree of difficulty associated with locating and gaining access to large systems; 2) the time-consuming nature and the expense of collecting and analyzing the data for studies; and 3) the difficulty in finding personnel with the appropriate skills to perform empirical studies. The authors agree with these reasons and add that the nature of studies in testing and defect analysis adds a measure of difficulty since the data that are dealt with require confidentiality. However, they have managed these concerns with the cooperation of an organization that recognizes the need to understand how the characteristics of developed code affect software quality. Ostrand and Weyuker themselves conducted an industrial case study on a system of 500,000 lines of code (LOC) and more than 13 releases. The authors’ study looks at more than 200,000 LOC that operate in an overall system of more than 19 million LOC in one release. The size of this system and the plethora of interactions between the modules of interest (invoked by a broad base of users) and the entire system give the findings of this case study at least the strength of other studies that are conducted on smaller laboratory systems of a few hundred LOC.

The issue of replication also arises when considering studies on one application. In the domain of industrially based empirical research, one cannot consider the applicability of one study on one release in isolation. The approach taken here is to consider the work and studies of others and implement replication by conducting the study in an additional domain. Hence, the authors collectively achieve repeated investigation of a single hypothesis. Briand and Labiche (2004) discuss this approach when they explore the topic of replication in empirical studies. The point made (Briand and Labiche 2004) is that replication in the empirical research domain does not require exact repetition. Many factors may vary, including the design of the study, the number of observations, and the characteristics of those involved in the studies.

Therefore, this case study is not the final word on the value of the Pareto principle. However, findings made with a widely used and mature application are valuable additions to the body of knowledge in defect analysis. This work contributes to other relevant studies (including Ostrand and Weyuker 2002; Adams 1984; Fenton and Ohlsson 2000; Ostrand, Weyuker, and Bell 2004; Gittens et al. 2002; Daskalantonikis 1992; Khostgoftaar et al. 1996). It is also notable that this article explores new ground in comparisons of complexity metrics, since others have proposed such metrics to quantify code but empirical verification is sparse (Kan 2003). The authors presented some of this work in Gittens, Kim, and Godwin (2005); however, they have expanded that work in this article with more detailed discussions of the relationships between complexity measures and the concentration of defects.

Related Work

An empirical study is a test that compares perceptions to observations (Perry, Porter, and Votta 2000). Such studies outline hypotheses and seek to prove or disprove them. The results of empirical studies can therefore influence software development practice.

For the reasons noted in Ostrand and Weyuker (2002) and the need to protect sensitive facts, there are few publicly available, industrially based empirical studies. There are even fewer focused on the distribution of defects in large production-level software, and the impact of complexity in its various forms. The authors, however, note a few relevant examples.

Fenton and Ohlsson (2000) did a pivotal empirical study. In this work, the researchers also examined the Pareto principle. The authors’ work strongly relates to this study. Fenton and Ohlsson made several unexpected discoveries, and, in a few cases, their studies
confirmed expectations. For example, they found weak support for the hypothesis that a higher number of defects found in function-level testing correlates to a higher number of defects found in system-level testing. In contrast, they refuted the hypothesis that a higher number of errors found in-house translates to a higher number of defects found by customers. An interesting statement by these researchers is that one cannot make statements about software engineering that will apply to all situations, and that many of the results in these hypotheses and beliefs appear to vary with the project. So, it is imperative that software practitioners continue studies such as these in industrial circumstances and observe whether particular patterns are consistent.

Adams (1984) did another critical study that documented a broad analysis of IBM® software. The study showed that defects found by users just after the release of software products cause the most pervasive failures in the field. Adams found that the defects that cause the most repeated problems in the field are relatively few in relation to the total number of defects found eventually. Therefore, Adams was able to motivate preventive maintenance at IBM to pre-empt the redundant and costly rediscoveries of tenacious problems.

More recently, Graves et al. (2000) studied a telephone switching system and examined the relationship among defects, code size, LOC changed, module age, and complexity. Module age is derived by multiplying the number of lines changed by the period elapsed since those changes were made. Their overall findings were that complexity and code size have a poor relationship with software defects and, hence, cannot be used to predict the occurrence of the defects.

Alternatively, Ostrand and Weyuker (2002) present a case study of 13 releases of an inventory tracking system also to investigate the distribution of software defects (or faults) and fault proneness (Khostgoftaar et al. 1996) of code. They investigated whether defects are concentrated in small percentages of code. They found this to be true. They found that defects tend to be concentrated more in certain files as the software evolves over releases. More specifically, they found that, as the releases progressed, all of the defects were concentrated in 4 percent of the code by the last release studied. This finding applied to defects found in-house and did not apply to defects found by users. Users found defects in 7 percent of the code and this persisted throughout releases.

In the authors’ previous work (Gittens et al. 2002), they investigated the relationships among defects found by an in-house test team, defects found by users in the field, and coverage achieved by system testing. As part of this work, they investigated if, in the case where more defects were found in a module by the in-house test team, customers will find fewer defects in that module after software release. They found that counter-intuitively this is not necessarily so, since the same files where users discover a concentration of defects, the in-house test team discovers a concentration of defects. As a distributed replication of another study, the authors’ work added consideration to the findings in Fenton and Ohlsson (2000). This is because their findings were different from Fenton and Ohlsson (2000) on the topic of defects found in-house giving indication of defects in the field, and found that (with respect to files) where there are defects found in-house, there are also defects found in the field. Since the focus (Gittens et al. 2002) was to analyze the effects of testing on defect occurrence, the authors did not investigate the Pareto principle then. This complementary work is done here.

The System

The software used in this work is a production-level system. The entire product is more than 19 million lines of C, C++, and Java code. It also has portions written in shell script, Perl, makefile, and other languages. In this study, the authors only consider the C/C++ portion of the system for their analysis. The components used in the study are typical of the facilities of interest to the system test team for the software under test (SUT). This software allows users to store, access, and manipulate their information using system resources—including CPU, memory, disk space, and communications. The SUT is available for several operating systems and platforms, and progresses through several versions. The authors did this study using one recent version of the SUT with a focus on one platform and one operating system. Future work will explore other components on other platforms and other versions.
The two software components of the product comprise 221 files. Table 1 summarizes the size of the files. The product is highly complex and evolves through several releases.

THE EMPIRICAL STUDY

The metrics examined in this study include pre-release defects and post-release defects, since concentrations of faults both in the field and in-house indicate if trends persist after release. Other metrics studied include: 1) code size measured by number of files; 2) complexity measured by McCabe’s complexity; 3) Henry-Kafura fan out (FO); and 4) the number of functions.

This study presents a hypothesis and investigates its validity. The authors present facts related to perceptions held in software development. Therefore, they present their hypotheses, background discussion on these hypotheses by other researchers, and their findings.

**TABLE 1** Summary of application file sizes

<table>
<thead>
<tr>
<th>Lines of code</th>
<th>Number of files</th>
</tr>
</thead>
<tbody>
<tr>
<td>200 and below</td>
<td>38</td>
</tr>
<tr>
<td>201–400</td>
<td>51</td>
</tr>
<tr>
<td>401–600</td>
<td>28</td>
</tr>
<tr>
<td>601–800</td>
<td>31</td>
</tr>
<tr>
<td>801–1000</td>
<td>15</td>
</tr>
<tr>
<td>1001–2000</td>
<td>34</td>
</tr>
<tr>
<td>2001–3000</td>
<td>14</td>
</tr>
<tr>
<td>3001–4000</td>
<td>4</td>
</tr>
<tr>
<td>4001–5000</td>
<td>2</td>
</tr>
<tr>
<td>5001–6000</td>
<td>1</td>
</tr>
<tr>
<td>6001–7000</td>
<td>1</td>
</tr>
<tr>
<td>7001–8000</td>
<td>2</td>
</tr>
<tr>
<td>Total</td>
<td>221</td>
</tr>
</tbody>
</table>

**FIGURE 1** Histogram of files vs. proportion of defects pre-release

- Hypothesis 1: Approximately 80 percent of defects occur in 20 percent of code.

This hypothesis is a concrete statement of proposals made in the work of others. Ostrand and Weyuker (2002) stated that there is widespread belief that: “... we can expect a small percentage of files will contain a large percentage of the faults detected at every stage of the life cycle.”

They investigated their expectation for their inventory-tracking case study and found strong evidence that a small percentage of files accounted for a large percentage of faults. This occurred for all of their releases, as well as the stages of development before release and after release.

The authors’ study with the two components in both the pre-release and post-release stages gave the following results. Two hypotheses that are more specific arise from the more general hypothesis 1.

- Hypothesis 1a: Approximately 80 percent of the defects found in testing occur in 20 percent of the code.

The authors observed a pattern where a small number of files contained most of the defects. Figure 1, which is a histogram of the 10 percentiles of defects that occur in the two components, shows that approximately 30 percent of the files contain all of the defects found before software release. The files grouped in Figure 1 are shown in detail in Figure 2. In Figure 2, the same pre-release defects are presented against the number of files that occur in the software, with the files being arranged in order of less to more in-house defects.

**FIGURE 2** Graph of files vs. proportion of in-house defects
defects found. The defects are shown to occur from file no. 159 through file no. 221. That is, they found 100 percent of the pre-release defects in 28 percent of the files. So in Figure 1, the authors divided these 221 files into 10 percentiles and ordered the files in those percentiles from those with the least number of defects to those with the most (left to right). So the last three percentiles in the 10 contain all of the defects.

The observation of 100 percent of the pre-release defects in 28 percent of the files is important, since this shows that if a test team tests 100 percent of the code, only 28 percent of their efforts discover defects. This indicates that perhaps, on limited schedules, one can better focus testing effort on the vital few. Further to the Pareto principle, 80 percent of the defects occurred in 26 percent of the files. The findings do not confirm an adherence to the 80:20 hypothesis, but do confirm that most defects are found in a small percentage of the code.

• Hypothesis 1b: Approximately 80 percent of the defects that users find occur in 20 percent of the code.

There is a similar pattern in Figures 3 and 4 to that observed in Figures 1 and 2. The difference is that the defects found in the field hold more closely to the 80:20 rule. Figure 3 shows the post-release defects against the files in the software divided into 10 percentiles, with the percentiles of files arranged in order of less to more field defects found. As a result, the last percentile contains that tenth of the files with the largest number of field defects. Figure 4 is similar to Figure 3, but shows the post-release defects against the actual number of files in the software, with the files arranged in order of less to more field defects found. The defects occur in approximately 23 files, which is 100 percent of the defects occurring in 10 percent of the code. This is again a case where most of the defects occur in a few files.

Others such as Fenton and Ohlsson (2000) have confirmed the concentration of defects in a few application files. They found that in the case of in-house testing, 60 percent of the defects are concentrated in 20 percent of the files. They also found 100 percent of the operational faults in 10 percent of the code. This shows a range in defect concentration, yet concentration nevertheless.

• Hypothesis 2: The distribution of defects in code files relates to the complexity in the files.

This hypothesis is often the basis that practitioners and researchers use to explain why most of the defects occur in a small percentage of the code. It is intuitive that the more involved the code, the more prevalent the defects. The authors investigate this hypothesis with a solid statement of the belief. They further decompose this hypothesis into two related hypotheses.

• Hypothesis 2a: Approximately 80 percent of complexity occurs in 20 percent of code.

Figure 5, which is a histogram that orders the code files on the x-axis from those with the least number of defects found in-house to those with the most, shows that varied complexity extends across all of the files.

There is no evidence of concentration of complexity in a few files. It is therefore unlikely that complexity provides a reason for the concentration of defects in

FIGURE 3 Histogram of files vs. defects found in the field

FIGURE 4 Graph of files vs. defects found in the field
A Closer Look at the Pareto Principle for Software

code files. This leads to further investigation of the relationship between defects and complexity. This is done in Hypothesis 2b.

- Hypothesis 2b: There is a relationship between defects, both field defects and in-house defects, and code complexity.

In Ostrand, Weyuker, and Bell (2004) the authors found that there is no significant relationship between defects and the complexity of the code files. The authors’ own work made the same discovery. This counters the findings of Troster (1992), who summarized discussions with developers of the IBM SQL/DS™ product where the developers perceived that more complexity led to increased defects. If one shows a similar graph to that in Figures 2 and 4 and includes the McCabe complexity measures on the graph, as in Figure 6, then one can see that complexity is not concentrated in a few files. It also shows that there is no pattern of complexity similar to that of field defects in the files.

In addition, a scatter plot of McCabe’s complexity against field defects confirms that there is no distinct relationship between the two variables (see Figure 7). There is similarly no relationship between complexity and in-house defects, as can be seen from the lack of a discernable pattern in Figure 8. This means that 80 percent of the complexity of the SUT is not attributed to 20 percent of the code, and hypothesis 2b and, more generally, hypothesis 2, do not hold. Even with reasonable relaxation of the hypothesis to consider the vital few and the trivial many, there is no application in the case of complexity. Therefore, the distribution of defects in code files does not relate to complexity in files. The authors have confirmed the findings of Ostrand and Weyuker (2002) in a different software domain.

- Hypothesis 3: The relationship between the distribution of defects and the complexity of code does not depend on the complexity measure in use.

Since it has been shown that there is no reliance on complexity in the concentration of defects in certain files, several questions arise. The first question is whether the relationship, or lack thereof, between complexity and defects and defect concentration relates to the complexity measure chosen. As a result, with this hypothesis, the authors look at the relationship between two measures of the complexity of code other than McCabe’s cyclomatic measure: 1) the number of functions in each file, and 2) the FO measure of each file.
Functions This is the number of functions in the code studied, as identified by function type, name, arguments, return type, and parentheses. Functions encapsulate and modularize distinct tasks. One uses this measure to find the number of tasks that the software under study is expected to carry out. One may consider it more suitable to use the function point measure, which encapsulates external inputs, external outputs and interfaces, files, and queries supported by the application. The authors offer that the intention of using function points to measure complexity is to provide a more attainable and general instrument that can be used to compare, standardize, and approximate the size of software developed in many programming languages (Kan 2003). However, because the authors are aware of the precise programming languages for their SUT, their focus is not comparison, and because they are familiar with the code of the SUT, they do not need to estimate the functions. The authors do not deny that the function point surrogate is useful to standardize function counting, but for this case, the number of functions in the code is a reasonable approximation of code complexity.

The authors found that there is no distinct relationship between the number of functions in each file and the defects found in the files. Figure 9 (which is a scatter plot of the relationship between the number of functions and the defects found by users) and Figure 10 (which is a scatter plot of the relationship between the number of functions and the defects found by testing) show the somewhat random nature of this relationship.

FIGURE 7 Scatter plot of the relationship between McCabe complexity and field defects

FIGURE 8 McCabe complexity and defects found by testing

FIGURE 9 The relationship between the number of functions and the defects found by users

FIGURE 10 The relationship between the number of functions and the defects found by testing
An interesting pattern emerges, however, between Figures 7 and 8 and Figures 9 and 10. There is a tendency for more defects to be found in-house than in the field. In addition, there are 39 distinct points on the plot of defects found in the field. In the plot of defects found in-house, one can see that there are 70 distinct points. This is because even though there are 221 distinct files, there are files in which there are the same number of defects and the same number of functions, so there is some overlap. The larger number of points for in-house defects, however, shows wider coverage and overall, more in-house defects than in the field. Kim (2003) discusses this in more detail.

**Fan out (FO)** Henry-Kafura FO provides an alternative measure of complexity that incorporates the data containment and management in the form of data structures. Therefore, it accounts for programs that are data focused. The software quality team can also derive this measurement during the design stage. In the case where the organization is attempting projections, this metric is useful. In the authors’ study, they found that again there is no notable relationship among the FO complexity metric and either defects found in-house or in the field.

The scatter plots for these relationships are very similar to Figures 7-10. This suggests a possible relationship among the complexity measures that are generally used. Perhaps investigation of relationships between defects and the various types of complexity can answer this question. The authors observed the truth in this posit when they conducted correlation analysis between these measures and derived a line graph to illustrate the trends between the measures for the software under study (as shown in Figure 11). There is a definite similarity of trends among McCabe complexity, the number of functions in the code, and the FO value.

Table 2 shows the relationships among the complexity measures, quantified by R. The authors note all strong positive relationships, where each complexity measure increases with the others or decreases similarly. A study by O’Neal (1993) found that the complexity measures are related. This study, however, looked at McCabe complexity, Halstead complexity, and LOC (Kan 2003). In addition, the correlations are shown to be strong since the $R^2$ values are all greater than 0.5. The authors therefore have strong evidence to support their hypothesis. The relationship between the distribution of defects and code complexity does not depend on the complexity measure in use.

**CONCLUSIONS AND FUTURE INVESTIGATION**

In this study, the authors learned many useful lessons regarding the Pareto principle for software. Many of them are important because of their empirical validation of

**TABLE 2** The relationships among complexity measures

<table>
<thead>
<tr>
<th>Relationships</th>
<th>Pearson</th>
<th>$R^2$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Correlation Between McCabe and Fan out:</td>
<td>0.783671</td>
<td>0.614141</td>
</tr>
<tr>
<td>Correlation Between McCabe and Number of Functions:</td>
<td>0.838928</td>
<td>0.703801</td>
</tr>
<tr>
<td>Correlation Between Number of Functions and Fan out</td>
<td>0.743336</td>
<td>0.552549</td>
</tr>
</tbody>
</table>

**FIGURE 11** Correlation of complexity metrics
previous findings in other domains. They used distributed replication, which adds breadth to software quality principles on the behavior of defects. The authors' other observations also are important, since they measure previously undocumented relationships between metrics.

The first hypothesis explored the relationship between defect distributions over software files. This case study did not support the hypothesis of an adherence to the 80 percent of the defects in 20 percent of the code. The general idea, however, is accurate. The test team found 100 percent of the defects in 28 percent of the code, and 80 percent of the defects in 26 percent of the code. Users also found defect concentration in a small percentage of the files. They sought to explain this by examining the concentration of complexity in the code. Authors such as Troster (1992) have discussed intuitions about complexity and defects with developers, and have collected qualitative information that shows belief that defects occur where there is more structural complexity. As a result, these developers are reluctant to modify highly complex code files. The authors’ findings and the findings of Adams (1984); Fenton and Ohlsson (2000); and Graves et al. (2000) did not support this theory. There is, however, a definite trend toward more defects existing in a few files. The authors’ study is on a different application from that in Adams (1984); Fenton (2000); and Graves (2000) and therefore supports the breadth of application of the evidence.

With regard to the lack of an observable relationship between the occurrence of defects and complexity, they then sought an explanation of defect concentration in files, and considered the possibility that the chosen complexity measure may be a hindrance to defining a relationship. It was therefore worthwhile to investigate whether another measure of complexity would highlight a preponderance of defects in the case of high levels of complexity where complexity is defined differently. They surveyed the literature and found that O’Neal (1993) noted a relationship among McCabe’s complexity, Halstead’s complexity, and LOC. They decided to complement O’Neal’s work by investigating whether this was borne out for other complexity measures. They looked at the correlations among McCabe’s, FO, and the number of functions. The literature does not document the relationships among these simple complexity measures, which are accessible to most organizations. These are all structural code metrics, since they count physical characteristics of the code, which may explain why the authors found a strong correlation among them.

This means that if they substitute most of the structural complexity metrics such as those outlined in this study and those in O’Neal’s study, one probably would not find a relationship that indicates that complexity is the reason for the concentration of defects. The authors propose to do further work to examine correlations between logical/functional/design complexity and defects and their concentrations. They have started this process by using function counts. Function points, however, which evaluate logical and physical measures, may show a different relationship.

This leads to the concept of problematic modules, which was first observed in Gittens et al. (2002). This confirmed the existence of rogue modules also mentioned in Fenton (2000). In Gittens et al. (2002), the authors noted that there are modules that tend to have fault concentration. In discussion with experts in the software domain, this appears related more to functionality and project factors than to structural elements of the code. The work in Khostgotaar et al. (1996) uses the fault-prone nature of some code to predict the overall behavior of the product. Leszak, Perry, and Stoll (2000) found that defects are largely a factor of changes in functionality, architectural concerns, usage, and other factors not quantified by structural measures. As a result, in future work, the authors will investigate correlations among defects, root causes, and project factors that are quantified through parameters such as module maturity. Maturity can be evaluated using modifications of models such as the Constructive Cost Model (COCOMO) as discussed in Boehm (1981) and Gittens, Bennett, and Ho (1997).

ACKNOWLEDGMENTS
The authors would like to thank Yong Woo Kim for his work in their previous work in Pareto principle analysis. His contributions and feedback proved invaluable in the development of previous related research.

REFERENCES
A Closer Look at the Pareto Principle for Software


*IBM and SQL/DS are trademarks or registered trademarks of International Business Machines Corporation in the United States, other countries, or both.

*Java and all Java-based trademarks are trademarks of Sun Microsystems, Inc. in the United States, other countries, or both.

*Other company, product, or service names may be trademarks or service marks of others.

*The views contained here are the views of the authors and not necessarily IBM Canada.

BIographies

Mechelle Gittens has worked and carried out research in software engineering since 1995. She is a research staff member at the IBM Toronto Lab and has a doctorate from the University of Western Ontario (UWO) where she is now a research adjunct professor. Her work is in software quality, testing, empirical software engineering, software reliability, and project management, and she has published at international forums in these areas. Her work has included consulting for the Institute for Government Information Professionals in Ottawa and software development for an Internet start-up. Gittens’ research previously led to work with the IBM Centre Advanced Studies (CAS) where she was a faculty fellow. Her work with IBM is in customer-based software quality where she works to understand how customers use software to create intelligence for more reflective testing, development, and design. She can be reached by e-mail at mgittens@ca.ibm.com and mgittens@csd.uwo.ca.

David Godwin graduated from York University in 1985. Since joining IBM in 1989, he has held positions in system verification test and product service. He started with IBM as a testing contractor and subsequently accepted a full-time position with IBM in 1989. He held the position of team lead for system testing until 1996, and continued his career with IBM in the roles of service analyst and team lead from 1996 to 2000. In 2000, he returned to system verification testing and is presently a quality assurance manager responsible for system testing for three major departments within IBM.
A Closer Look at the Pareto Principle for Software

APPENDIX

Terminology
This section defines the important terms used in this article.

- **Defect**: This is a variance from a desired software attribute. A software defect is also a software fault. There are two types of defects: 1) defects from product specification that occur when the product deviates from the specifications; and 2) defects from customer expectation that occur when the product does not meet the customer’s requirements, which the specifications did not express. The development organization in this case considers a defect to be from product specifications because the product is not custom built and requirements are determined in house.

- **In-house defects**: These are defects found by a test team prior to the software release.

- **Field defects**: These are defects found by users post release.

- **Pearson correlation coefficient**: This is a measure (denoted by R) of the strength of the linear relationship between two random variables x and y. It ranges between 1, for a strong positive relationship, and -1 for a strong negative relationship. Its square, R², indicates the strength of the fit of the data as it quantifies the sum of squares of deviation.

- **Component**: This is part of a system, such as a software module.

- **Error**: This is an incorrect or missing action by a person or persons causing a defect in a program.

- **Product**: This is a software application sold to customers.

- **Pre-release**: This refers to any activity that occurs prior to the release of the software product. The pre-release stage is synonymous with the in-house stage of the product life cycle.

- **Post-release**: This refers to any activity that occurs after the release of the software product. The post-release stage is synonymous with the field stage of the product life cycle.

- **McCabe’s complexity**: This is measured using cyclomatic complexity, defined for each module as e – n + 2, where e and n are the number of edges and nodes, respectively, in the control flow graph.

- **Henry-Kafura fan out (FO)**: For a module M, the fan out is the number of local flows that emanate from M, plus the number of data structures that are updated by M.

- **Number of functions**: This is the number of C, C++, or Java™ functions in the code as identified by function type, name, arguments, return type, and parentheses.

- **SUT**: This refers to the software under test.

Thanks to Reviewers
In addition to members of our Editorial Board (whose names are listed in each issue), each submission to SQP is examined by up to three review panelists. These professionals labor anonymously to evaluate and recommend improvements to all our manuscripts.

As we reach the end of Volume 9, Software Quality Professional wishes to acknowledge the contribution of those individuals who have reviewed submissions in the past year:

| Karla Alvarado | Uma Ferrell | Leonard Jowers | Rupa Mahanti | Marc Rene |
| John Arce      | Christopher Fox | Jim Ketchman | Linda McInnes | Malcolm Stiefel |
| David Blaine   | Eva Freund   | Derek Kozikowski | Subhas Misra | John Suzuki |
| Leo Clark      | Tammy Hoelting | Carla Langda Sivik | Joe McConnell | Rajan Thiyyagarajan |
| Pat Delohery   | Nels Hoenig  | Gloria Leman   | Fred Parsons   | Larry Thomas |
| David Dills    | Trudy Howles | Scott Lindsay  | Shari Pfeleger | Joe Wong |
| Tom Ehrhorn    | Alyce Jackson | Paulo Lopes    | Uma Reddi     |           |