Copyright
by
Gary Lynn Wilson Jr.
2013

The Report Committee for Gary Lynn Wilson Jr.
certifies that this is the approved version of the following report:

**An Empirical Study on Software Quality: Developer Perception of Quality, Metrics, and Visualizations**

APPROVED BY

SUPERVISING COMMITTEE:

______________________________
Miryung Kim, Supervisor

______________________________
Herbert Krasner

# An Empirical Study on Software Quality: Developer Perception of Quality, Metrics, and Visualizations

**by**

**Gary Lynn Wilson Jr., B.S.E.E.**

**REPORT**

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

**Master of Science in Engineering**

The University of Texas at Austin

May 2013

# Abstract


## An Empirical Study on Software Quality: Developer Perception of Quality, Metrics, and Visualizations



Gary Lynn Wilson Jr., M.S.E.
The University of Texas at Austin, 2013

Supervisor: Miryung Kim



Software tends to decline in quality over time, causing development and maintenance costs to rise. However, by measuring, tracking, and controlling quality during the lifetime of a software product, its technical debt can be held in check, reducing total cost of ownership. The measurement of quality faces challenges due to disagreement in the meaning of software quality, the inability to directly measure quality factors, and the lack of measurement practice in the software industry. This report addresses these challenges through both a literature survey, a metrics derivation process, and a survey of professional software developers. Definitions of software quality from the literature are presented and evaluated with responses from software professionals. A goal, question, metric process is used to derive quality-targeted metrics tracing back to a set of seven code-quality subgoals, while a survey to software professionals shows that despite agreement that metrics and metric visualizations would be useful for improving software quality, the techniques are underutilized in practice.

# Table of Contents


**List of Tables** vii

**List of Figures** viii

**Chapter 1. Introduction** 1

1.1 Why Quality? . . . 1
1.2 Maintenance . . . 2
1.3 Challenges . . . 4

**Chapter 2. Software Quality** 6

2.1 Defining Quality . . . 6
2.1.1 Customer Satisfaction . . . 6
2.1.2 Quality Factors . . . 8
2.1.3 Defects . . . 9
2.2 Managing and monitoring quality . . . 10

**Chapter 3. Goal, Question, Metric** 12

3.1 The business goal . . . 14
3.2 Subgoals . . . 14
3.3 Measurement goals . . . 16
3.4 Metrics . . . 21
3.4.1 Metric definitions . . . 21

**Chapter 4. Developer Survey** 29

4.1 Background . . . 30
4.2 Quality . . . 31
4.3 Metrics . . . 36
4.4 Visualization . . . 39

**Chapter 5. Conclusion** 47

**Appendices** 50

**Appendix A. GQM** 51

A.1 Subgoals, questions, entities, and attributes . . . . . . . . . . . . . 51

**Appendix B. Developer Survey Questions** 59

B.1 Introduction . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 59

B.2 Background . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 59

B.3 Quality . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 60

B.4 Metrics . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 63

B.5 Visualization . . . . . . . . . . . . . . . . . . . . . . . . . . . . 65

B.6 Thank You . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 66

**Appendix C. Developer Survey Responses** 67

C.1 Responses for question: Please describe the methods your team uses for improving quality, including how and why you use them. . . . 67

C.2 Responses for question: If your team does not use any methods for improving quality, please describe why not. . . . . . . . . . . . . . 71

**Bibliography** 73

# List of Tables


3.1 Steps for applying the Goal-Driven Process [57]. . . . . . . . . . . 13
3.2 Software quality factors, as defined by McCall et al. [52, 53]. . . . 15
3.3 Summary of subgoals identified for GQM exercise. . . . . . . . . . 16
3.4 Summary of measurement goals chosen for GQM exercise . . . . . 17
3.5 Metrics and the measurement goal(s) they address. . . . . . . . . 28

4.1 Country distribution of the 75 survey responders. . . . . . . . . . 31
4.2 Company-size and team-size distributions of the 75 survey responders. 31
(a) Size of company. . . . . . . . . . . . . . . . . . . . . . . . . 31
(b) Size of team. . . . . . . . . . . . . . . . . . . . . . . . . . 31
4.3 Weighted-average ranks of how important each quality factor is in representing quality. . . . . . . . . . . . . . . . . . . . . . . . . 33
4.4 The use of methods during development and/or release processes for improving quality, sorted from most used to least used. . . . . 34
4.5 Frequency that teams make use of metrics to address measurement goals presented in GQM exercise. . . . . . . . . . . . . . . . . . 36
4.6 The knowledge and use of 28 different metrics, sorted by most frequently used and well-known. . . . . . . . . . . . . . . . . . . . 44
4.7 Level to which responders agree with using software metrics to evaluate employee performance, broken down by manager and non-manager roles. . . . . . . . . . . . . . . . . . . . . . . . . . . . 45
4.8 Percentage of responders by how useful it would be to visualize metrics about software projects over time. . . . . . . . . . . . . . 45
4.9 Percentage of responders by how likely they would be to use a software metrics visualization tool to improve the quality of software they write. . . . . . . . . . . . . . . . . . . . . . . . . . . . . 45

# List of Figures


4.1 Ranks given to attributes for importance in representing code quality. 41

4.2 Rank distributions of schedule, quality, and cost influence for a typical software project. . . . . . . . . . . . . . . . . . . . . . . . . 42

4.3 Distributions of the frequency that teams make use of metrics to address measurement goals presented in GQM exercise. . . . . . . 43

4.4 Number of responses on how subjects would like to use a tool for calculating and visualizing metrics describing their software. . . . 46

# Chapter 1

# Introduction

Our goal is to understand developers' perception of software quality, their utilization of development methods for improving quality, and their awareness and use of software metrics and metric visualizations for monitoring code quality during software evolution.

## 1.1 Why Quality?

The environment in which software exists is constantly in flux, pressuring software to change with it in order to remain useful and relevant [16]. Failure to adapt to the changing environment leads to software aging [58], while software change itself leads to increased complexity and software decay [45].

Thus, it is typical to see software decline in quality as it ages [47]. The decline in quality becomes visible through several symptoms, including increased number of defects and increased development and maintenance effort. This cost of poor and/or degrading quality can be summed up by the notion of technical dept [17, 20]. In order to keep technical debt in check, quality must be managed throughout software lifecycles. However, the maintenance phase, in particular, deserves special attention.

It is well understood that the maintenance phase of software systems is

the most costly part of the entire software lifecycle [47, 49, 65]. Therefore, it is prudent that we explicitly target this phase of the software lifecycle in order to reduce overall software system cost. However, reduced maintenance cost is not the only benefit from improved quality. Further cost savings can be seen through:

- Reduced number of bugs
- Bugs found earlier in the product's lifecycle
- Shorter development times
- Reduced time-to-market

A reduced number of bugs, as well as the catching of bugs earlier in the lifecycle, can save future development time that would have otherwise been spent correcting operational faults and fixing the bugs themselves. Reduced development time, both during project development and after product launch, leads to shorter time-to-market and quicker maintenance releases. Meanwhile, these benefits can lead to improved customer satisfaction, as well as competitive advantages over slower, less quality-focused competitors. High quality reduces software product cost on many fronts; however, none are more important than the products' maintenance phase of their lifecycle.

## 1.2 Maintenance

Not only is the majority of cumulative software expenditure spent on maintenance (as opposed to development) [47], but also the majority of a software's existence lies in the maintenance lifecycle. In fact, there have even been proposed

models of the software lifecycle in which maintenance is described as an iterative stage following initial development, for example the "evolution" stage of Bennet and Rajlich's staged software lifecycle model [12].

Why do software products have such a problem with maintenance? From Lientz and Swanson [48], we see that the "maintenance" phase of software is composed of the following classes of maintenance activities:

- Adaptive – modifications due to changes in the software environment
- Perfective – implementation of new or changed user requirements
- Corrective – modifications to fix errors
- Preventive – modifications for improving future maintainability or preventing future problems

It turns out that these maintenance tasks may themselves experience a range of issues—some no different than normal development tasks—including, but not limited to, high complexity and lack of readability, poor code structure that resists change, tightly coupled components that require scattered changes, etc. In other words, it is difficult to "[perform] maintenance on a system which was not designed for change" [65], a side-effect of a system that lacks quality. Some have suggested that maintenance itself should instead be thought of as subsequent development iterations [12, 54]. Such a viewpoint reiterates the notion that maintenance tasks can be as difficult as the "development" phase of a software system, while also accepting that maintenance typically experiences longer durations.

In fact, according to its cost, one could even argue that the maintenance phase is *more difficult* than a product's initial development. This is why high-quality software is so important, as it addresses the undesirable code attributes which make development and change more expensive. Even though the software industry began studying quality nearly four decades ago [14], the reality is that software quality monitoring is underutilized in practice. Many development teams still struggle with the "laws of software evolution" that software systems grow more complex over time, while declining in quality [46, 47].

## 1.3 Challenges

Depending on the organization and/or software project's primary requirements, one or more different attributes may be given higher importance during the project's development. *Quality*, though, is a desirable attribute for any software project. Quality's beneficial effects are seen across all project phases, leading to a reduced total cost of ownership. However, the measurement, and thus improvement, of quality faces several challenges for researchers, developers, and managers alike. For example:

- What is software quality?
- Is there agreement on which attributes compose software quality?
- How does one measure software quality or its attributes?
- How well do developers know available software quality metrics?
- How often do developers utilize software quality metrics in practice?

- How can tools and visualizations be used by developers to improve software quality?

The remainder of this report explores these questions further through a literature survey on software quality and a survey of software professionals. First, Chapter 2 explores the definition of quality. Next, Chapter 3 presets a goal, question, metric exercise that derives a set of metrics for improving software quality. Finally, Chapter 4 presents the evaluation of a survey to software professionals, with the goal of understanding their perception of software quality, their utilization of development methods for improving quality, and their awareness and use of software metrics and metric visualizations.

# Chapter 2

# Software Quality

## 2.1 Defining Quality

Is quality the answer to the maintenance problem [65], and for reducing the cost of software? It may not be the sole answer, but monitoring software quality is certainly one solution to the maintenance problem. Before we can begin to improve quality, we must first answer the struggling question: *What is software quality?*

In the art of software engineering, we have long attempted to define quality. Over the decades we have collected several different understandings of what quality represents [41], including:

- Customer satisfaction
- Quality factors (e.g. maintainability, reliability, etc.)
- Defects

### 2.1.1 Customer Satisfaction

Voas [69] and Denning [25] have argued that quality depends on your viewpoint, and that the software industry should put more focus on satisfying our customers with systems that meet their needs.

There is no question that the industry should ensure that the software products are meeting the needs of the customers; however, this so called "fit of need" for software only makes up a marginal portion of what *quality* truly represents. Take, for example, a scenario where a company is looking to purchase a software system to handle their payment processing. If the company was presented a system that did not have the features they needed, would the product be considered to have low quality? It is likely that, instead, the company would simply view the product as not meeting their needs and move along to another product. In other words, it is possible for a software product to be of good quality, even if it does not fit the needs of a particular customer.

The above scenario highlights the importance of point of view when defining quality. Regarding maintenance, counting the addition of features due to changing requirements is one thing, but counting the addition of features due to missed or inaccurate requirements as maintenance is completely different. Inaccurate requirements will, indeed, ultimately resurface in later phases of a software's lifecycle, e.g. the maintenance phase, when they are more costly to correct [6]. However, this should be classified as a lack of quality in the requirements-gathering and development processes, not a lack of quality in the code itself.

While the establishment of good planning and development processes may reduce overall software cost, when analyzing the code itself we must assume that the proper processes were in place for the phases prior to the product having actual code to study. Thus, this report focuses on software quality from viewpoint of the developer, pertaining to the quality of the code product itself.

### 2.1.2 Quality Factors

Another take on quality embraces the notion that quality is a broad concept that should be broken down and analyzed through a set of quality characteristics. The International Standards Organization (ISO) provides quality vocabulary in ISO 8402-1986, which defines quality as "the totality of features and characteristics of an entity that bears on its ability to satisfy stated and implied needs" [4, 41]. Furthermore, there are several standards addressing quality, for example the ISO 9000 series and the and the British TickIT standards; however, research has shown that many organizations who have implemented these standards are either unsatisfied or have not shown improvements in product quality [21, 67]. This is not surprising given the fact that even the ISO 9000-3 software quality standard states that "[t]here are currently no universally accepted measures of software quality" [41].

The problem of defining quality has been evident even in the early literature, where quality has been shown to be a broad concept lacking opportunities for direct measurement. For example, Boehm et al. in 1976 defined several tens of quality metrics [14], which were further consolidated down to eleven quality factors (discussed further in Chapter 3) by McCall et al. [52, 53].

With the overwhelming candidates of quality factors to choose from, there is certainly no shortage of potential measurements to collect. However, most of the documented quality factors lack the ability to be directly measured, as they are just as broad and vague as *quality* itself. For example, how would one directly measure the factors of usability or flexibility? Chapter 3 expands on this idea through a goal, question, metric exercise, which is able to drill down to directly-

measurable code artifacts from several indirectly-measurable quality factors.

### 2.1.3 Defects

Number of defects is a common way to measure software quality [29], perhaps because this metric is more directly measurable than the broad, *-ility* attributes utilized by quality factors and customer satisfaction alternatives presented in the previous sections. However, while defects may be a direct measure, the number of defects alone does not fully represent a software product's code quality. Based on the quality factors concept introduced in Section 2.1.2, and later detailed in Chapter 3, defects only relate to a few of the quality factors, e.g. *reliability* and *maintainability*.

Additionally, though a defect count measurement may be easy to obtain, complications do arise in exactly *how* defects should be counted. Examples of this can been seen with defect types and defect severity [64]. For example, should one major defect count more than one minor defect? Should a defect in requirements be treated the same as a defect in code?

While the measurement of defects may not be absolute, it is straightforward to see how defects directly affect the cost of a software product. As has been shown in several studies [6], the cost of fixing a defect increases throughout a software product's lifecycle, with the highest cost—exponentially more than the pre-code-development stages—occurring after the product has been deployed. Thus, the maintenance phase, which follows a product's launch, experiences the highest cost for fixing defects.

## 2.2 Managing and monitoring quality

Like the software industry, the manufacturing industry also faces issues with product quality and the costs associated with bad quality. Given the higher maturity of the manufacturing industry, though, there is much the software industry can learn from the former's experiences with managing quality. With traditional manufacturing, it can be very expensive (and sometimes impossible) to revise a product or correct defects once the product has rolled off the assembly line. On the other hand, many perceive software as a more malleable product that never lacks the opportunity for modification [16]. Perhaps this is a reason why the software industry has not focused on quality as strongly, ignoring or dismissing the cost of bad quality.

Meanwhile, the manufacturing industry has understood the importance of managing quality since decades before computers and software became prevalent. In the 1950s, W. Edwards Deming brought his ideas for quality improvement to th Japanese car manufacturing industry. In his viewpoint, high quality was defined as products with few defects [23, 24]. A major take-away from these ideas was that producing quality products involved managing the processes that went into creating the products.

Several quality management methodologies have appeared since—some specific to the software industry and some generally applicable to any industry—including:

- Total Quality Management [61]
- Six Sigma [36]

- Continuous Improvement [39]
- ISO/IEC 15504, or Software Process Improvement and Capability Determination (SPICE) [68]
- Capability Maturity Model (CMM) [59]

...all of which focus on improving and/or maturing the processes which drive the creation and development of products.

Managing the quality of software, though, is more complex than the simplified, defect-only viewpoint of traditional manufacturing. More than just void of defects, software also needs other qualities, such as readability or modularity, for example. To that point, the complexity of software just makes a stronger case for the need to manage the development process and all parts of the software lifecycle. The next chapter will work towards this goal, by deriving metrics explicitly for the purpose of improving software quality through the monitoring and understanding of the development and maintenance processes.

# Chapter 3

# Goal, Question, Metric

As we have learned in the previous section, improving software quality requires improvement of the processes that control the software's development. However, it is *measurement* that provides the information needed to feedback into process improvements, allowing for continuous process—and quality—improvement. This report specifically targets quality improvement from the developer's perspective, since the focus is improving quality of the code itself and since developers are the primary creators of the code product.

Instead of just measuring for measurement sake, quality improvement for a team, organization, and/or software product should purposefully work towards addressing the priorities of the specific organization, product, or situation. This chapter presets artifacts and results from a goal, question, metric (GQM) exercise performed by the author. Each metric derived in this exercise will retain traceability back to goals represented by the quality factors from Section 2.1.2. It is intended that the resulting set of metrics may then act as a framework for future quality-improvement efforts, including the use by software-development organizations or the future development of software-quality visualizations tools.

The GQM technique [8, 10, 11] is a methodology for deriving relevant metrics to collect, based on questions that answer unknown details which are trace-

able back to specific measurement goals. However, making the jump directly to measurement goals can be difficult. Park et al. [57] recommend a "goal-driven" process, which begins at a higher level—at business goals—and works down to the measurement goals as described in the pure GQM process. The complete "goal-driven" process, according to Park et al. [57], is described in Table 3.1.

| **Step** | **Description** |
|---|---|
| 1 | Identify business goals. |
| 2 | Identify knowledge discovery areas about business goals. |
| 3 | Identify subgoals. |
| 4 | Identify entities and attributes related to subgoals. |
| 5 | Identify measurement goals based on review of subgoals, entities, and attributes. (This is the *G* of GQM.) |
| 6 | Identify questions that will help addresses measurement goals. (This is the *Q* of GQM.) |
| 7 | Identify data elements that answer the questions. |
| 8 | Identify and define measures to use. (This is the *M* of GQM.) |
| 9 | Identify actions to take in order to implement the measures. |
| 10 | Prepare plan for implementing the measures. |

Table 3.1: Steps for applying the Goal-Driven Process [57].

The remainder of this chapter presents results from this goal-driven process. First, Section 3.1 defines a generic, quality-improvement-focused business goal (Step 1). Section 3.2 then defines subgoals (Step 3) formulated directly from a list of software quality factors, which act as the knowledge discovery areas (Step 2). Finally, Section 3.3 presents the derived measurement goals (Steps 5–7) and Section 3.4 presets the resulting metrics (Step 8). Note that Steps 9 and 10 involve the application of the process to a particular organization, and thus are outside the scope of this exercise.

## 3.1 The business goal

In general, business goals will be different from organization to organization, and from product to product. Example business goals include "reduce maintenance costs," or "increase product market share." The GQM exercise presented here assumes the overall, driving business goal:

> "Improve code quality from the developer's viewpoint."

Starting with such a general business rule for the focus of this GQM exercise, what follows is a relatively generic, quality-improvement metrics plan. An organization performing the same exercise is recommended to not follow this exact metrics plan directly, but rather to utilize it as a foundation. Organizations are encouraged to add in their own goals, particular to their environment, and follow the same goal-driven process to produce a metrics plan tailored to their precise needs.

## 3.2 Subgoals

Normally, the next step would be to ask questions related to the business goal(s) in order to discover and organize ideas into a set of subgoals. However, the exercise presented here directly positions software quality factors as the subgoals. Starting with the eleven software quality factors outlined by McCall et al. [52, 53] (shown in Table 3.2), the list was then pared down by discarding four factors—Correctness, Integrity, Portability, and Interoperability—that relate more to the functional, security, and deployment-environment requirements of a software product, respectively, rather than to the quality of the code itself.

| Quality Factor | Definition |
|---|---|
| Correctness | Extent to which a program satisfies its specifications and fulfills the user's mission objectives. |
| Reliability | Extent to which a program can be expected to perform its intended function with required precision. |
| Efficiency | The amount of computing resources and code required by a program to perform a function. |
| Integrity | Extent to which access to the software or data by unauthorized persons can be controlled. |
| Usability | Effort required to learn, operate, prepare input, and interpret output of a program. |
| Maintainability | Effort required to locate and fix an error in an operational program. |
| Testability | Effort required to test a program to insure it performs its intended function. |
| Flexibility | Effort required to modify an operational program. |
| Portability | Effort required to transfer a program from one hardware configuration and/or software system environment to another. |
| Reusability | Extent to which a program can be used in other applications. |
| Interoperability | Effort required to couple one system with another. |

Table 3.2: Software quality factors, as defined by McCall et al. [52, 53].

With the remaining seven factors, the GQM process then continued with these factors as its subgoals, listed in Table 3.3 for easy reference. Note, in a GQM process performed for an actual organization, the same subgoals presented here may exist as subgoals mapped from real business goals.

With subgoals identified, the goal-driven process then dictates a mental model be constructed for each subgoal in order to explore and gather more knowledge. Questions related to each subgoal were formulated, each detailed with its associated entities and attributes. Following from the guiding business goal from

| Subgoals | |
|---|---|
| Subgoal 1 | Improve the software's reliability. |
| Subgoal 2 | Improve the software's efficiency. |
| Subgoal 3 | Improve the software's usability. |
| Subgoal 4 | Improve the software's maintainability. |
| Subgoal 5 | Improve the software's testability. |
| Subgoal 6 | Improve the software's flexibility. |
| Subgoal 7 | Improve the software's reusability. |

Table 3.3: Summary of subgoals identified for GQM exercise.

Section 3.1, each subgoal's purpose was to reduce development costs for a software product over its development, testing, and maintenance lifecycles, from the perspective of the developer. For brevity, the fully-detailed subgoals, questions, entities, and attributes are omitted from this section, but can be found in Appendix A.

## 3.3 Measurement goals

Next, measurement goals were derived through analysis of the subgoals and related questions. Note that each measurement goal typically also includes details about the perspective and environment. However, the exercise here assumes that all perspectives are from the developer point of view. The environment details are outside the scope of this exercise, as they include details related to the specific organization and/or product(s) under consideration.

Included in each of the measurement goal descriptions below are the purpose, the object of interest (underlined), the associated subgoals addressed, as well as a set of questions that guided the resulting metrics (to be presented in the next section). For reference, Table 3.4 summarizes the selected measurement

goals.

| Measurement Goals | |
|---|---|
| Measurement Goal 1 | Analyze code size. |
| Measurement Goal 2 | Evaluate issues from issue database. |
| Measurement Goal 3 | Improve test suite and benchmarks. |
| Measurement Goal 4 | Improve documentation. |
| Measurement Goal 5 | Reduce code complexity. |
| Measurement Goal 6 | Reduce maintenance effort. |
| Measurement Goal 7 | Improve code modularity. |

Table 3.4: Summary of measurement goals chosen for GQM exercise

**Measurement Goal 1:** Analyze code size.

**Addresses:** Subgoal 1, Subgoal 2, Subgoal 3, Subgoal 4, Subgoal 5, Subgoal 6, Subgoal 7

**Questions:**

- How many packages, modules, classes, and functions/methods exist?
- How many lines of code exist?
- What is the distribution of lines of code, classes, and functions/methods over packages and modules?
- What are the features and/or use cases implemented by the code?

**Measurement Goal 2:** Evaluate issues from issue database.

**Addresses:** Subgoal 1, Subgoal 3, Subgoal 4, Subgoal 6

**Questions:**

- How many issues exist, and in what proportion to the code size?

- What is the distribution of issues over available statuses (e.g. open, in progress, closed)?
- What is the distribution of issues over available severities (e.g. trivial, normal, important, critical)?
- What is the distribution of issues over available software lifecycle phases (e.g. requirements, development, testing, maintenance)?
- What is the distribution of defects over packages and modules?
- What is the distribution of defect reports over time?
- What is the average time between defect reports?
- How long have issues been open?
- How many defects have been found during code reviews?
- What proportion of issues relate to a previous issue that has already been released/deployed?

**Measurement Goal 3:** Improve test suite and benchmarks.

**Addresses:** Subgoal 1, Subgoal 2, Subgoal 5

**Questions:**

- Are test suite failures tracked and corrected?
- How well is code covered by the test suite?
- How well are the product's requirements, features, and use cases covered by the test suite?
- What proportion of performance-sensitive features are tested with a benchmark test?

- What is the trend for benchmark test results?

**Measurement Goal 4:** Improve documentation.

**Addresses:** Subgoal 3

**Questions:**

- What is the size of the documentation in relation to the code?
- Is the code well commented?
- Are all requirements, features, and use cases documented?

**Measurement Goal 5:** Reduce code complexity.

**Addresses:** Subgoal 3

**Questions:**

- What are the number of code paths that exist within functions and methods?
- What is the distribution of code-path counts over modules, classes, and functions/methods?
- How complex are the class hierarchies?
- What are the sizes of the function/method call hierarchies?
- How complex are the conditional logic depths within functions and methods?

**Measurement Goal 6:** Reduce maintenance effort.

**Addresses:** Subgoal 4

**Questions:**

- Does the code conform to style guidelines?
- What proportion of code is unused or duplicated?
- What amount of code is changed for completed issues (i.e. defects and new features)?
- How much development effort has been spent on issues, and what is the distribution of effort over time?
- What is the distribution of issue requests over time?
- What is the total effort backlog (i.e. total amount of effort required for all open issues)?
- Is the rate of issue requests decreasing over time?
- How many defects are being introduced?
- How long does it take for defects to be fixed?

**Measurement Goal 7:** Improve code modularity.

**Addresses:** Subgoal 6, Subgoal 7

**Questions:**

- What amount of code is duplicated?
- What is the distribution of module dependencies, function/method calls, and class instantiations over the available modules, functions/methods, and classes?
- Do classes effectively encapsulate their data and behavior?
- Do classes effectively make use of their attributes and methods?
- Which packages, modules, and classes change together in completed issues (i.e. defects and new features)?

## 3.4 Metrics

From the measurement goals and related questions above, metrics were then identified to provide information for answering the questions and providing the information necessary for helping achieve each goal. Table 3.5 lists all of the derived metrics, in alphabetical order, along with markings to signify which measurement goal(s) each metric addresses.

Note that the metrics listed in Table 3.5, and later defined in Section 3.4.1, are not an exhaustive list of metrics needed for analyzing and controlling code quality. They were selected through the guidance of the GQM process performed in this report, which assumed a particular viewpoint for analysis and a particular set of business and measurement goals.

### 3.4.1 Metric definitions

The definitions for all metrics from Table 3.5 are included below, in alphabetical order:

**Benchmark coverage:** The percentage of performance-sensitive features that are tested with a benchmark test (for measuring performance) [70, 71]. It is assumed that a list of performance-sensitive features is known and documented.

**Benchmark performance:** The resulting value for each available performance benchmark test [70, 71]. Resulting value may be a duration of time, an execution rate, or any other measurement that allows comparison to previous and future benchmark results for the same test.

**Class cohesion:** How well the attributes and methods of a class are related, or belong together. Lack of cohesion (LCOM) measured as $LCOM = \frac{\frac{1}{a}\sum_{j=1}^{a} A_j - m}{1-m}$, where $a$ is the number of class attributes, $m$ is the number of class methods, and $A_j$ is the number of methods that access the $j$th attribute [15].

**Class inheritance breadth:** The maximum number of subclasses at any level of a class hierarchy [62].

**Class inheritance depth:** The number of levels in a class hierarchy [18, 62].

**Code review defects found:** The average number of defects found per code review [40].

**Code churn ratio:** The ratio of source lines of code changed (sum of net added and removed lines) to total source lines of code over a past duration of time, where duration may be different granularities (e.g. one month, six months, one year, etc.) [55].

**Comment ratio:** The ratio of Comment lines of code to Source lines of code (see *Lines of code*) [30].

**Cyclomatic complexity:** The number of different execution paths within the code [51], including:

**Aggregate cyclomatic complexity:** The total sum of all complexity values from the entire source code.

**Average cyclomatic complexity:** The average complexity over packages, modules, classes, and functions/methods.

**Maximum cyclomatic complexity:** The maximum complexity across all code entities of a particular kind (e.g. maximum package, module, class, or function/method complexity).

**Documentation coverage:** The percentage of classes, functions/methods, and features/use-cases that are documented or have associated comments or docstrings (depending on language features).

**Duplicated code percentage:** The percentage of duplicated source lines of code to total source lines of code, e.g. as reported by a code-duplication or code-clone detection tool [63].

**Fan-in:** For a given module, class, or function/method, the number of other modules, classes, or functions/methods, respectively, that access or call the given object [37].

**Fan-out:** For a given module, class, or function/method, the number of other modules, classes, or functions/methods, respectively, that the given object accesses or calls [37].

**Issue age:** The average duration an issue has been active, i.e. from time of request to time of close [34].

**Issue density:** The number of issues per thousands of source lines of code [27, 33]. Can be measured at different granularity, including by package or module. Counts may also be limited to certain issue types, e.g. defect density, new-feature density, etc.

**Issue effort:** The total amount of development time (e.g. person-hours) recorded against all issues [33]. May also be limited to certain issue types, e.g. defect effort, new-feature effort , etc.

**Issue effort backlog:** The total amount of development time (e.g. person-hours) estimated for completion of all open issues, not counting development time already accrued [33]. May also be limited to certain issue types, e.g. defect effort backlog, new-feature effort backlog, etc.

**Issue effort rate:** The amount of development time (e.g. person-hours) spent over a time [33], aggregated with a certain duration granularity, e.g. one week, one month, etc. Rates may also be limited to certain issue types, e.g. defect effort rate, new-feature effort rate, etc.

**Issue rate:** The number of issues reported over a time [33], aggregated with a certain time duration, e.g. one week, one month, etc. Rates may also be limited to certain issue types, e.g. defect rate, new-feature rate, etc.

**Lines of code (LOC) and thousands of lines of code (KLOC):** Total lines of code, including source lines of code, documentation lines of code, and comment lines of code [2, 29]. Can also be measured at different granularity, including by package, module, class, or function/method. LOC metrics include:

**Source lines of code (SLOC):** A count of the number of lines, excluding blank lines, documentation lines, and comment lines.

**Documentation lines of code:** A count of the number of lines within documentation files, excluding blank lines and comment lines.

**Comment lines of code:** A count of the number of lines that are comments only, including documentation constructs within source code (e.g. class or function docstrings in supported languages) and excluding blank lines or lines that also contain code.

**Mean time between issue:** The average duration between issue reports [34]. Values may also be limited to certain issue types, e.g. mean time between defect, mean time between new feature, etc.

**Nesting depth:** The number of nested conditional logic or looping constructs within a portion of code [60]. Also includes:

**Maximum nesting depth:** The maximum level of nesting. May be measured at different granularity, e.g. overall, per class, or per function/method.

**Average nesting depth:** The average level of nesting amongst similar code entities, e.g. module, class, or function/method.

**Number of code entities:** The number of entities (i.e. namespace, object, or function) within the code [26]. Depending on language features, this may include the metrics:

**Number of classes:** The number of class definitions within the code.

**Number of functions/methods:** The number of function and/or method definitions within the code (depending on language features).

**Number of modules:** The number of modules present within the code, i.e. a single file that contains one or more classes and/or functions

(depending on language features).

**Number of packages:** The number of packages present within the code, where a package is defined to be a collection of modules and/or classes (depending on language features) that has a namespace. For example a namespace of `X.Y.Z`, where `Z` is a module or class, would count as two packages: `X` and `X.Y`.

**Number of issues:** The total number of issues present in the issue database [33]. Counts also measured across various issue attributes, including:

- Status (e.g. open, in progress, closed, invalid, duplicate)
- Active vs. non-active (e.g. open or in progress vs. closed or invalid)
- Severity (e.g. trivial, normal, important, critical)
- Type (e.g. defect, feature)
- Lifecycle reported (e.g. requirements, development, testing, quality assurance, maintenance)
- Whether not issue was caused by a previous corrective action (i.e. defect fix)

**Number of features and use cases:** The number of features and/or use cases that are implemented within the code, e.g. from requirements and specifications [44].

**Style errors:** The number of instances of code that does not conform to style standards, e.g. as reported by an automated style checker [13].

**Test coverage:** The percentage of SLOC, branches, or requirements/features/use-cases covered by a run of the full test suite [74].

**Test suite failures:** The number of failures reported by a run of the full test suite [74].

| Metric | Measurement Goal | | | | | | |
|---|---|---|---|---|---|---|---|
| | **1** | **2** | **3** | **4** | **5** | **6** | **7** |
| Benchmark coverage | | | • | | | | |
| Benchmark performance | | | • | | | | |
| Class cohesion | | | | | | | • |
| Class inheritance breadth | | | | | • | | |
| Class inheritance depth | | | | | • | | |
| Code review defects found | | • | | | | | |
| Code churn ratio | | | | | | • | |
| Cyclomatic complexity | | | | | • | | |
| Documentation coverage | | | | • | | | |
| Duplicated code percentage | | | | | | • | • |
| Fan-in | | | | | • | | • |
| Fan-out | | | | | • | | • |
| Issue age | | • | | | | • | |
| Issue density | | • | | | | • | |
| Issue effort | | | | | | • | |
| Issue effort backlog | | | | | | • | |
| Issue effort rate | | | | | | • | |
| Issue rate | | • | | | | • | |
| Lines of code | • | • | • | • | | • | • |
| Mean time between issue | | • | | | | • | |
| Nesting depth | | | | | • | | |
| Number of code entities | • | • | | | • | | • |
| Number of issues | • | • | | | | • | |
| Number of features and use cases | • | | | • | | | |
| Style errors | | | | | | • | |
| Test coverage | | | • | | | | |
| Test suite failures | | | • | | | | |

Table 3.5: Metrics and the measurement goal(s) they address.

# Chapter 4

# Developer Survey

From the literature review in Chapter 2 and the GQM exercise in Chapter 3, it was shown that software quality is a broad topic covering several factors, requiring a diverse set of metrics to adequately evaluate and control. A survey to software professionals was administered in order to compare their opinions of the definition of software quality with the definitions found in the literature, and to understand how well-known and utilized software methods, metrics, and visualizations are within software teams. Guiding questions for the survey included:

- How do software professionals define quality?
- What methods do teams most frequently use for improving quality?
- How often do teams/companies manage quality through metrics and visualizations?
- How well do developers know available software quality metrics?
- How likely are software professionals to make use of metric visualization tools for monitoring software quality during software evolution?

## 4.1 Background

The survey consisted of 23 questions, and can be found in its entirety within Appendix B. The survey was administered online using kwiksurveys[1], and was announced via my personal blog[2], Twitter[3], Google+[4], word-of-mouth, and the Students in Software Engineering[5] mailing list.

In total, 75 responses across 21 countries and more than 17 companies—not all subjects specified their place of work—were accumulated. See Table 4.1 for a full breakdown of the frequency of responses by country. Responders averaged 10.2 years of software industry experience and identified themselves into the following breakdown of job roles (approx.): 79% developers, 16% managers, 4% Test/QA engineers, and 1% business analysts.

Over half (50.7%) of the responders identified as working for a company with 6–50 employees, while another significant portion (30.7%) identified as working for companies with greater than 1,000 employees. The vast majority of responders (89.3%) identified as working within teams of 10 or less, almost evenly split between a size of 1–5 (49.3%) and a size of 6–10 (40.0%). See Table 4.2 for the full distribution of responders' company (a) and team (b) sizes.

[1] http://kwiksurveys.com/
[2] http://thegarywilson.com/
[3] https://twitter.com/
[4] https://plus.google.com/
[5] http://www.edge.utexas.edu/sse/

| Count | Countries |
|---:|---|
| 33 | United States |
| 6 | Germany |
| 5 | Poland |
| 5 | United Kingdom |
| 4 | Australia |
| 3 | France |
| 2 | Argentina, Austria, Canada, India |
| 1 | Denmark, Hungary, Italy, Lithuania, Netherlands, Romania, Singapore, South Africa, Spain, Sweden, Switzerland |

Table 4.1: Country distribution of the 75 survey responders.

| Size | Count | % |
|---|---:|---:|
| up to 5 | 9 | 12.0 |
| up to 50 | 29 | 38.7 |
| up to 250 | 8 | 10.7 |
| up to 1,000 | 6 | 8.0 |
| over 1,000 | 23 | 30.7 |

(a) Size of company.

| Size | Count | % |
|---|---:|---:|
| up to 5 | 37 | 49.3 |
| up to 10 | 30 | 40.0 |
| over 10 | 8 | 10.7 |

(b) Size of team.

Table 4.2: Company-size and team-size distributions of the 75 survey responders.

## 4.2 Quality

When asked which definition best defines software quality, responders overwhelmingly choose “a broad mix of factors” (77%) over “fit of need” (17%) and “lack of defects” (5%); however, the latter two definitions weighted heavily in the responders’ perceptions of quality. When asked to rank a set of eight quality factors in how important the factors represent code quality, *Correctness* and *Reliability* were predominantly ranked first and second, respectively. These results confirm the quality definitions found in the literature, with both the *Correctness*

(i.e. "fit of need") and *Reliability* (i.e. "defects") attributes standing out from the others. With quality in manufacturing very much tied to defects and reliability, it is interesting that, here, "lack of defects" was thought to be less important than "fit of need."

Figure 4.1 shows a graph of the rank selection distribution for how important the subjects felt each attribute represents quality, and Table 4.3 shows the attributes sorted by their weighted-average ranking. By far, the most important attribute was *Correctness*, which was placed in the first rank by nearly 70% of the subjects. Second, third, and fourth ranks were also clearly shown to be *Reliability*, *Usability*, and *Maintainability*, respectively. *Reusability* was clearly the lowest ranked factor, on average, while the remaining three factors (*Testability*, *Efficiency*, and *Flexibility*) were tightly bunched between the fourth and eight ranked factors. While most factors display a single peak within the rank positioning, *Efficiency* was the only factor that demonstrated two distinct local-maxima (centered around ranks three and seven), suggesting that this factor may have importance in only certain environments or applications.

Subjects were then asked to rank three popular trade-off factors (cost, quality, and schedule) in how influential the factors were for a typical software project at their company. The responses predominantly showed *schedule* ranked first, *quality* ranked second, and *cost* ranked third, with resulting weighted-rank averages of 1.69, 1.93, and 2.32 (out of 3), respectively. Figure 4.2 shows the full rank distribution for all three factors. In a follow-up question to responders who ranked either schedule or cost as more influential than quality, (summarized) reasons given for their selections included:

| Quality factor | Weighted-average rank (out of 8) | Median rank (out of 8) | Mode (out of 8) |
|---|---|---|---|
| Correctness | 1.76 | 1 | 1 |
| Reliability | 2.61 | 2 | 2 |
| Usability | 3.57 | 3 | 3 |
| Maintainability | 4.13 | 4 | 5 |
| Testability | 5.41 | 6 | 6 |
| Efficiency | 5.56 | 6 | 7 |
| Flexibility | 5.93 | 6 | 7 |
| Reusability | 6.97 | 7 | 8 |

Table 4.3: Weighted-average ranks of how important each quality factor is in representing quality.

- Typical projects involve fast-paced, exploratory prototyping.
- Schedule is typically driven by outside factors, such as dependent projects, partner relationships, or regulatory changes.
- Cost and/or schedule are typically the primary focus within contracts.
- Deadlines and schedule receive a high level of focus from management.
- Focus on low cost leads to situations such as the use of less experienced developers who lack the skills to improve quality, or a lack of proper management resources.

Although quality was not ranked as a top consideration for software projects, it was reassuring to learn that the majority of subjects did report on their teams' regular use of multiple methods during the development and/or release process for improving quality. The complete tally of responses can be seen in Table 4.4. The most popular method used was "informal discussion with colleagues" (84.0%),

followed closely by "automated testing" (78.7%) and "human quality assurance (QA) testing" (77.3%). The least used method was "software metrics", which received marks from only 17.3% of responders. The two responses marked "Other" both made mention of static code-analysis tools. The minimal use of metrics as a method for improving quality is disconcerting since it shows that most development teams do not quantitatively know if, or to what degree, their current quality improvement methods are helping them.

| **Method** | **Count (of 75)** | **%** |
|---|---|---|
| Informal discussion with colleagues | 63 | 84.0 |
| Automated testing | 59 | 78.7 |
| Human QA testing | 58 | 77.3 |
| Code reviews | 42 | 56.0 |
| Pair programming | 24 | 32.0 |
| Formal meetings | 21 | 28.0 |
| Software metrics | 13 | 17.3 |
| Other | 2 | 2.7 |

Table 4.4: The use of methods during development and/or release processes for improving quality, sorted from most used to least used.

When asked to elaborate on methods used for improving quality, including how and why they are used, the result was a wide range of responses, with most describing choices that were provided in the previous question (see Table 4.4). The full text of all responses can be seen in Appendix C. Notable answers not included in the answer choices included: refactoring, bug monitoring, coding guidelines, stress-test tools, branching strategies (e.g. feature branching), and continuing education. Additional insights gathered on choices that were presented include:

- Related to automated testing were mentions of test-driven development

(TDD) and continuous integration.

- Several responses mentioned the fact that automated testing was either under-utilized within their software products, new to their environment, and/or undergoing active improvement.
- Those who mentioned pair programming indicated use of the method for complex or critical bugs and code changes.
- Informal discussions were utilized for completing complex bugs or features, identifying potential issues, learning from problems teammates were facing, and completing changes requiring quick turn-around.
- Code reviews were utilized during new projects, release/sprint iterations, and mentorship situations.
- Human QA testing was used for verifying software correctness and usability, validating releases, and building test plans.
- Responses mentioning formal meetings involved project reviews or retrospectives.

Out of the 75 total subjects, 10 (13.3%) responded that their team does not use any methods for improving quality. Reasons cited included: tight deadlines, lack of time, lack of experience, bad tool support, non-production software, too many projects, lack of management, and team/company culture. The multiple mentions of time constraints here align with the earlier results that show *schedule* as a more influential trade-off factor on software projects over *quality*.

## 4.3 Metrics

In the previous section, it was shown that software metrics was the least utilized method for improving quality as part of development processes. However, when asked if their team currently utilizes any metrics for the intentional purpose of improving code quality, just over half of the responders (38 of 75) answered "yes."

The subjects were then asked how regularly their team used metrics to evaluate seven different areas, which represented a one-to-one mapping to the measurement goals derived in the GQM exercise (Section 3.3) and summarized in Table 3.4. Answer choices were a frequency-type Likert scale with the following labels: Never, Rarely, Sometimes, Regularly, Constantly. Figure 4.3 depicts the choice distributions for all seven measurement areas, and Table 4.5 shows the measurement areas sorted by most frequently used (a weighted-average ordering assuming approximately equal intervals of the frequency-type Likert scale, with values 1 to 5 representing "Never" to "Constantly", respectively).

| **Object of measurement** | **Mode** | **Median** | **Weighted rank (out of 5)** |
|---|---|---|---|
| Issues in issue database | Regularly | Sometimes | 2.99 |
| Test suites or benchmarks | Never | Sometimes | 2.67 |
| Maintenance effort performed | Never | Rarely | 2.39 |
| Modularity of software | Never | Rarely | 2.20 |
| Adequacy of documentation | Never | Rarely | 1.89 |
| Complexity of software | Rarely | Rarely | 1.88 |
| Size of software | Never | Rarely | 1.87 |

Table 4.5: Frequency that teams make use of metrics to address measurement goals presented in GQM exercise.

While over half of the responders answered “yes” to currently utilizing metrics for improving code quality, they did so at a very low frequency in each of the measurement areas questioned. Only two areas (*issues in issue database* and *test suites or benchmarks*) had a median frequency above “Rarely,” and were also not much more utilized themselves with a median frequency of “Sometimes.” Likewise, only two areas had a most-selected frequency above “Never”—*complexity of software* had a mode of “Rarely” and *issues in issue database* had a mode of “Regularly.” *Issues in issue database* was the most-frequently utilized metric area, clearly seen as on outlier within the “Regularly” frequency choice in Figure 4.3. In fact, aside from that single outlying point, there is no metric area for the “Regularly” and “Constantly” frequencies that had more selections than the least-selected metric area for the “Rarely” and “Never” frequencies.

Next, the subjects were asked to rate their knowledge and use of 28 different metrics on a Likert scale consisting of the four choices: Never heard of or do not know, Know of but have not used, Have used before, and Use regularly. The list of metrics used for the survey was primarily taken from the GQM derivation in Chapter 3, and was found throughout a range of topics in the software engineering literature, including software sizing and effort [1, 2, 35, 44], complexity [5, 32, 51], defects [6, 27, 34, 55], code duplication [42], testing [3, 22, 38, 70–72, 74], evolution [30], coupling [31], cohesion and object-oriented design [7, 15, 18, 26, 62], information flow [37], people and organizational structure [56], and various combinations of the proceeding topics [9, 28, 29, 40, 50, 75].

Table 4.6 shows the full count (and percentage) breakdown of responses for each metric. The data is sorted by highest weighted average (assuming ap-

proximate equal intervals of a Likert scale, weighted from 1 to 4 representing the selections "Never heard of..." to "Use regularly"), which allows identification of the central tendency of selections for each metric. From the data collected, several insights were captured:

- No metric was used regularly by a majority of the responders. The only metric that came close (42.7%) was *Unit test failures or pass/fail ratio*.

- Only four metrics have been used before or were used regularly by a majority of responders: *Unit test failures or pass/fail ratio*, *Line coverage*, *Lines of code*, and *Number of classes/functions/files/modules*. These four metrics were also the only ones that were both unknown to less than 10% of responders and used regularly by more than 10% of responders.

- Aside from the four metrics mentioned in the previous bullet, *Branch/path coverage* was the only other metric that was used regularly by more than 10% of the responders; however, a higher percentage of responders, at 19%, had never heard of this metric (about three times more than the four metrics mentioned above).

- Several metrics were highly known, yet not used before or used regularly, including: *Mean time between defect/error*, *Time-to-fix defect*, *Defect density*, *Code churn*, *Depth of inheritance tree*, *Depth of nesting*, and *Defect count or distribution*. It us unknown, however, if the high levels of knowledge of these metrics are due to familiarity of their use within the software industry or rather due to the ease of which their definitions can be gleaned from their names.

- Nearly one-third of the metrics presented were completely unknown by a majority of responders, with most metrics in this group used regularly by zero responders and used before by about 5% or less of responders. Metrics in this group consisted of *Fan-in and fan-out*, *Defect slippage ratio*, *Halstead metrics*, *Mutant killing percentage*, and several of the people and organizational metrics described by Zimmermann et. al [75].

The final two questions in the survey related to metrics were on the topic of the use of metrics to evaluate employee performance. About 15% of responders indicated that their company uses software metrics for evaluating employee performance. Surprisingly, nearly twice that amount (28.0%) either agreed or strongly agreed with the use of metrics for evaluating employee performance. The full results can been seen in Table 4.7, broken down by manager and non-manager roles (as identified earlier in the survey, see Section 4.1).

From the manager vs. non-manager results, we expectedly see that those in manager roles were much more likely to agree (50.0%) or strongly agree (8.3%) than were those in non-manager roles. Those in non-manager roles were most likely to be neutral (32.0%) or to disagree (23.8%). Both manager and non-manager roles differed by less than 1% in the strongly disagree segment and, in aggregate, very few (4.0%) responders identified as strongly agreeing with the use of metrics for employee performance evaluation.

## 4.4 Visualization

The survey to software professionals also included four questions on the topic of visualizing software metrics. Overall, responders overwhelmingly felt

that it would be useful to visualize metrics about their software projects over time, with over 81% who either agreed or strongly agreed with this statement and less than 6% who either disagreed or strongly disagreed. However, when asked if their team currently uses any tools to visualize software metrics over time, less than 30% answered "yes." In a similar question, subjects were asked how likely they would be to use a software metrics visualization tool to help improve the quality of software they write. Here, responders were also positive, albeit slightly less enthusiastic. The majority of responders fell into the likely (42.7%) or neutral (25.3%) segments, with an equal percentage (14.6%) in both the unlikely and very likely segments. The full results for these two questions can be seen in Table 4.8 and in Table 4.9.

From these results, combined with the earlier result that just over half of the responders answering "yes" to their team currently utilizing metrics for improving code quality, there clearly exists a gap between those who use metrics and those who use metrics visualizations. Additionally, the positive responses to the perceived usefulness of metrics visualizations, and willingness to use metrics visualizations, suggests there is opportunity and demand for such tools.

On the topic of visualization tools, subjects were asked how they would like to use such a tool (see Figure 4.4 for a chart of all responses). The most popular choices, selected by about 61–71% of the responders, were: *integrated into build/test server*, *integrated into issue tracker*, and *a self-hosted, stand-alone product with web interface*. Least popular were: *integrated into editor/IDE* and *a hosted service with web interface*. Four responders filled in a selection for *other*, mentioning a dedicated build server, as well as a script or command-line interface.

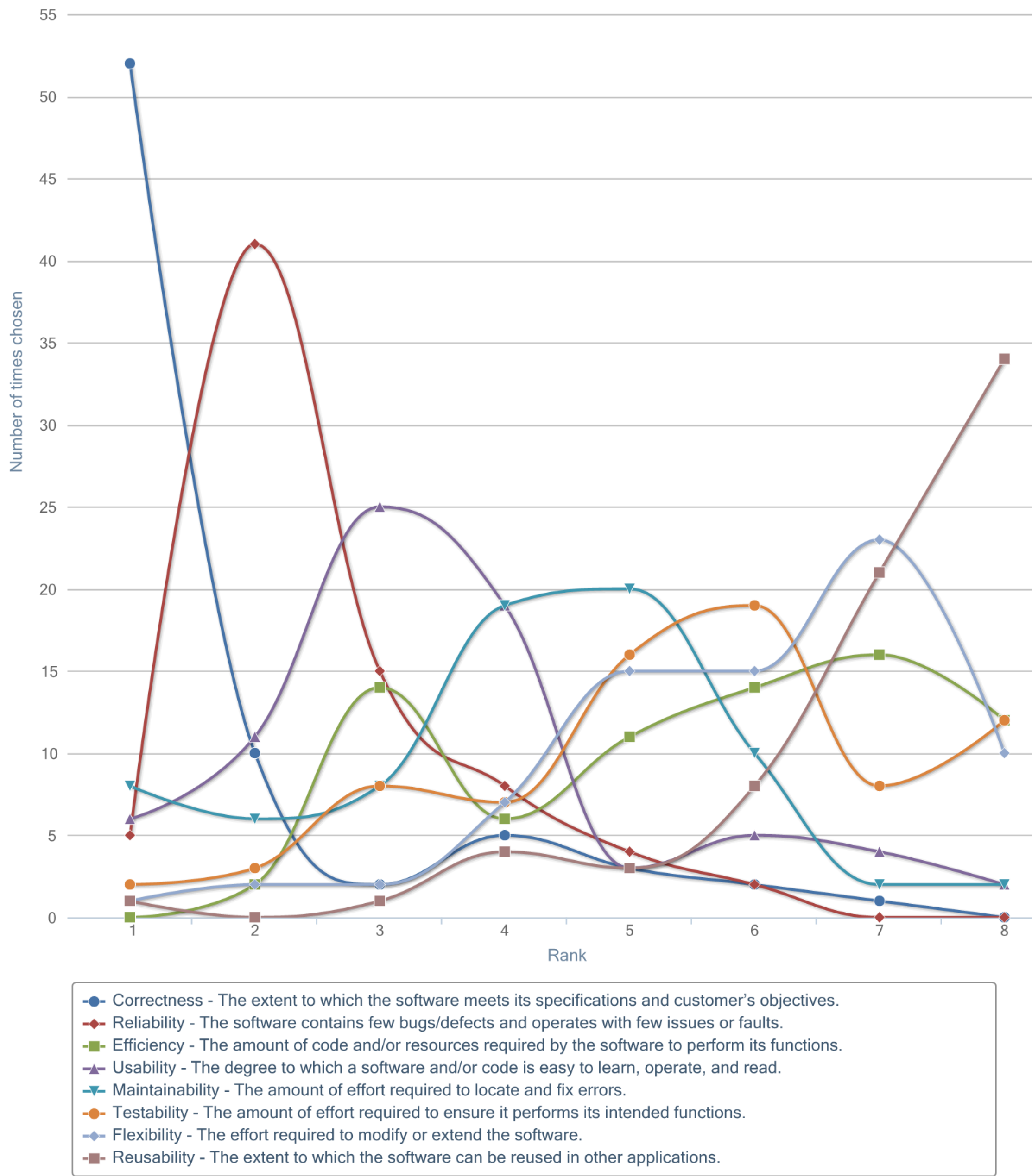


Figure 4.1: Ranks given to attributes for importance in representing code quality.

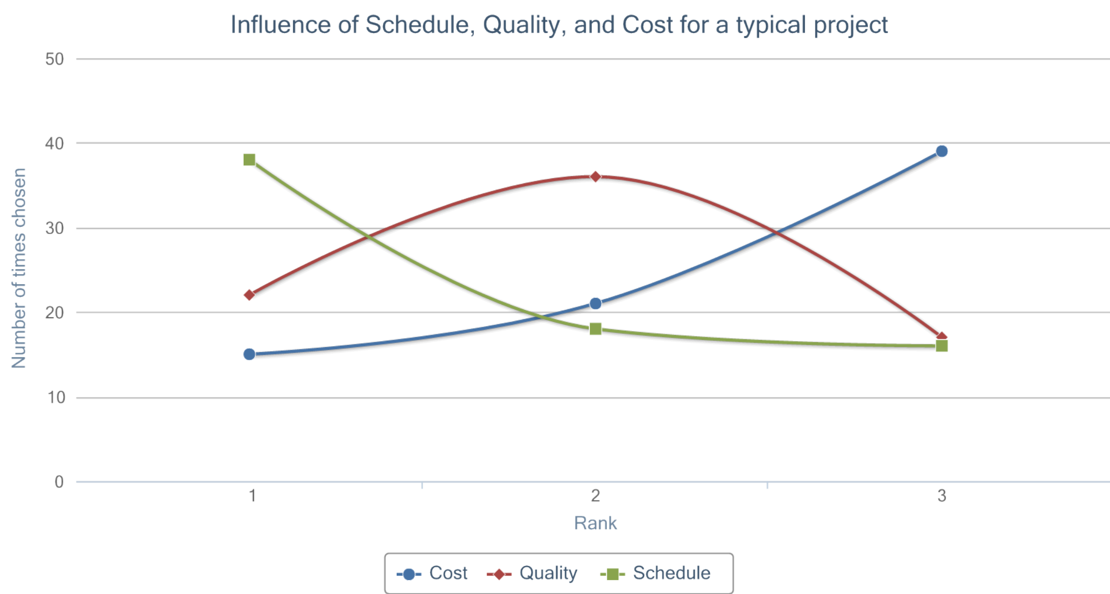


Figure 4.2: Rank distributions of schedule, quality, and cost influence for a typical software project.

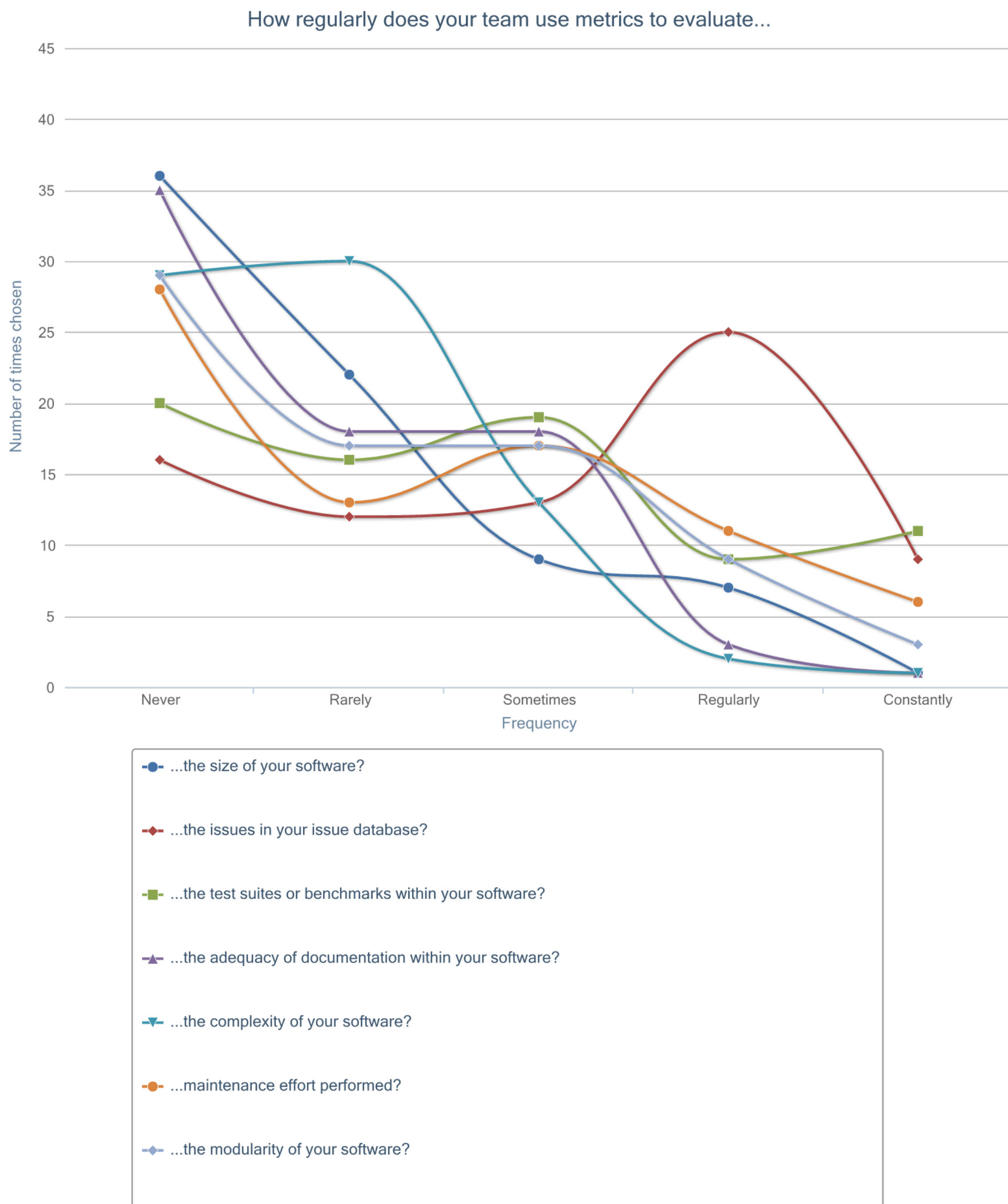


Figure 4.3: Distributions of the frequency that teams make use of metrics to address measurement goals presented in GQM exercise.

| Metric | Never heard of or do not know | | Know of but have not used | | Have used before | | Use regularly | | Weighted average |
|---|---|---|---|---|---|---|---|---|---|
| | Count | % | Count | % | Count | % | Count | % | (out of 4) |
| Unit test failures or pass/fail ratio | 4 | 5.3 | 10 | 13.3 | 29 | 38.7 | 32 | 42.7 | 3.19 |
| Line coverage | 5 | 6.7 | 24 | 32.0 | 30 | 40.0 | 16 | 21.3 | 2.76 |
| Lines of code | 2 | 2.7 | 29 | 38.7 | 32 | 42.7 | 12 | 16.0 | 2.72 |
| Number of classes/functions/files/modules | 5 | 6.7 | 32 | 42.7 | 27 | 36.0 | 11 | 14.7 | 2.59 |
| Branch/path coverage | 14 | 18.7 | 25 | 33.3 | 26 | 34.7 | 10 | 13.3 | 2.43 |
| Duplicated code (or code clone) percentage | 9 | 12.0 | 33 | 44.0 | 27 | 36.0 | 6 | 8.0 | 2.40 |
| Defect count or distribution | 14 | 18.7 | 38 | 50.7 | 16 | 21.3 | 7 | 9.3 | 2.21 |
| Time-to-fix defect | 13 | 17.3 | 44 | 58.7 | 13 | 17.3 | 5 | 6.7 | 2.13 |
| Cyclomatic complexity | 22 | 29.3 | 26 | 34.7 | 23 | 30.7 | 4 | 5.3 | 2.12 |
| Depth of nesting | 15 | 20.0 | 38 | 50.7 | 21 | 28.0 | 1 | 1.3 | 2.11 |
| Function points or similar | 29 | 38.7 | 24 | 32.0 | 17 | 22.7 | 5 | 6.7 | 1.97 |
| Coupling | 26 | 34.7 | 29 | 38.7 | 17 | 22.7 | 3 | 4.0 | 1.96 |
| Depth of inheritance tree | 20 | 26.7 | 39 | 52.0 | 15 | 20.0 | 1 | 1.3 | 1.96 |
| Cohesion (or lack of cohesion) | 27 | 36.0 | 31 | 41.3 | 14 | 18.7 | 3 | 4.0 | 1.91 |
| Defect density | 24 | 32.0 | 40 | 53.3 | 6 | 8.0 | 5 | 6.7 | 1.89 |
| Mean time between defect/error | 18 | 24.0 | 49 | 65.3 | 6 | 8.0 | 2 | 2.7 | 1.89 |
| Code churn, turnover ratio, or edit frequency | 26 | 34.7 | 39 | 52.0 | 10 | 13.3 | 0 | 0.0 | 1.79 |
| Number of engineers | 30 | 40.0 | 34 | 45.3 | 8 | 10.7 | 3 | 4.0 | 1.79 |
| Fan-in and fan-out (class/module dependencies) | 38 | 50.7 | 24 | 32.0 | 11 | 14.7 | 2 | 2.7 | 1.69 |
| Number of ex-engineers | 35 | 46.7 | 34 | 45.3 | 4 | 5.3 | 2 | 2.7 | 1.64 |
| Defect slippage ratio | 41 | 54.7 | 30 | 40.0 | 4 | 5.3 | 0 | 0.0 | 1.51 |
| Percentage of org contributing to development | 48 | 64.0 | 23 | 30.7 | 4 | 5.3 | 0 | 0.0 | 1.41 |
| Overall organization ownership | 54 | 72.0 | 17 | 22.7 | 4 | 5.3 | 0 | 0.0 | 1.33 |
| Level of organizational code ownership | 56 | 74.7 | 15 | 20.0 | 4 | 5.3 | 0 | 0.0 | 1.31 |
| Depth of master ownership | 59 | 78.7 | 12 | 16.0 | 4 | 5.3 | 0 | 0.0 | 1.27 |
| Halstead metrics | 58 | 77.3 | 16 | 21.3 | 1 | 1.3 | 0 | 0.0 | 1.24 |
| Mutant killing percentage | 60 | 80.0 | 13 | 17.3 | 2 | 2.7 | 0 | 0.0 | 1.23 |
| Organization intersection factor | 61 | 81.3 | 11 | 14.7 | 3 | 4.0 | 0 | 0.0 | 1.23 |

Table 4.6: The knowledge and use of 28 different metrics, sorted by most frequently used and well-known.

| Role | Count | Percentage of Responders | | | | |
|---|---|---|---|---|---|---|
| | | Strongly Disagree | Disagree | Neutral | Agree | Strongly Agree |
| Managers | 12 | 16.7 | 16.7 | 8.3 | 50.0 | 8.3 |
| Non-managers | 63 | 17.5 | 23.8 | 36.5 | 19.1 | 3.2 |
| Total | 75 | 17.3 | 22.7 | 32.0 | 24.0 | 4.0 |

Table 4.7: Level to which responders agree with using software metrics to evaluate employee performance, broken down by manager and non-manager roles.

| Strongly Disagree | Disagree | Neutral | Agree | Strongly Agree |
|---|---|---|---|---|
| 2.7 | 2.7 | 13.3 | 54.7 | 26.7 |

Table 4.8: Percentage of responders by how useful it would be to visualize metrics about software projects over time.

| Very Unlikely | Unlikely | Neutral | Likely | Very Likely |
|---|---|---|---|---|
| 2.7 | 14.7 | 25.3 | 42.7 | 14.7 |

Table 4.9: Percentage of responders by how likely they would be to use a software metrics visualization tool to improve the quality of software they write.

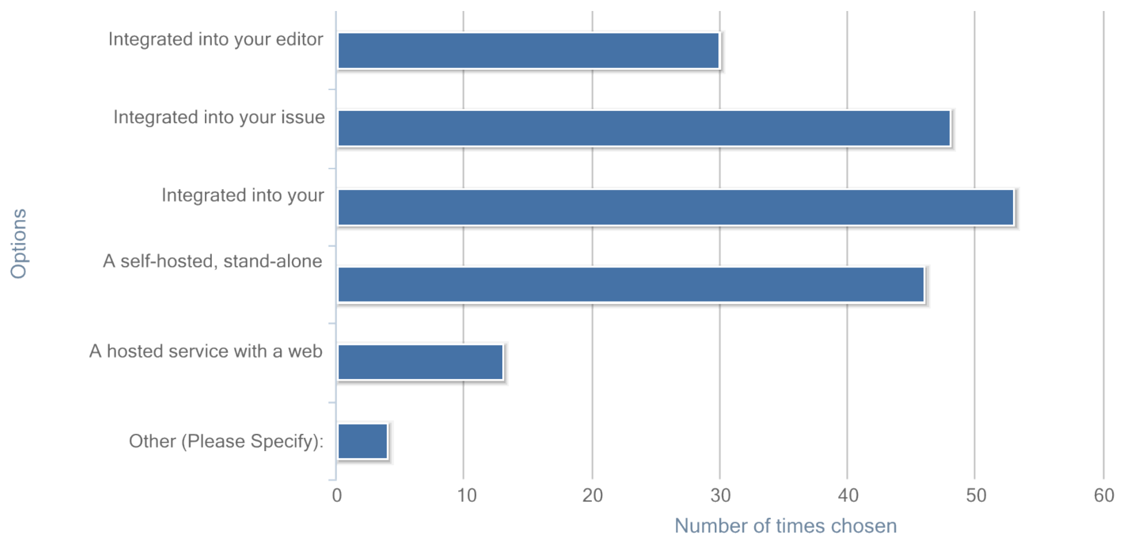


Figure 4.4: Number of responses on how subjects would like to use a tool for calculating and visualizing metrics describing their software.

# Chapter 5

# Conclusion

The quality of software is important because it lowers the product's total cost of ownership. However, quality's broad scope and nested layers of ambiguity provide challenges in understanding how to define, analyze, and manage software quality. Literature definitions of quality were reviewed, and a survey of software professionals showed that the multi-faceted idea of quality factors was most popular in practice. Meanwhile, the other quality definitions from the literature aligned with the most important quality factors (*Correctness* and *Reliability*), as ranked individually by the responders.

Even though this report set out to understand the quality of code itself, many professionals did consider the "fit of need" aspect an important representation of quality. Perhaps, it would be beneficial for future studies to split focus into two orthogonal quality concerts: quality of the requirements process (i.e. focused on requirements/correctness) and quality of the the development process (i.e. focused on code, defects, and other direct attributes of the code).

From the quality factors definition of quality, a GQM exercise was performed that resulted in the derivation of 27 targeted metrics which help to understand and analyze each quality factor for a given software product. These metrics may act as a starting point for future quality-driven improvement efforts that

share the same goals represented by the seven quality factors that were explored.

While research and case studies have shown the usefulness of metrics in improving software quality, the actual use of metrics in software development practice is underutilized (less than 18% of responders). In fact, metrics was the least utilized of all quality-improvement methods presented, behind more popular methods such as automated testing and code reviews. The use of software metrics visualization was also found to be lacking in practice, even though over 81% of software industry professionals agreed it would be useful and over 57% said they would be likely to use such a tool to help improve their code quality.

Results clearly showed that there is much opportunity in the industry for bringing both metrics and metrics visualizations to practice. With it would also come scientific results for quality improvement within development practices. Quality was also shown to take a second seat behind schedule demands on a typical software project. Current perception, as seen in several survey responses from development teams not utilizing any methods specifically for quality improvement, was that there was not enough time for quality. Perhaps the software industry would be wise to adopt quality management mantras from traditional manufacturing, such as "quality is free," "zero defects," and "continuous improvement." These notions revolve around the idea that the additional effort needed to improve quality is outweighed by the benefits seen from the improvement efforts [19, 24].

Within the software industry, research tells us that improved quality can lead to less defects, less rework, reduced cost, and quicker development cycles. These, in turn, can lead to improved customer satisfaction and increased market share. Thus, if software teams were to just focus more on quality, perhaps they

could get their desired schedule and cost benefits for free.

# Appendices

# Appendix A

# GQM

## A.1 Subgoals, questions, entities, and attributes

Full mental model of subgoals (see Section 3.2).

**Subgoal 1:** Improve the software's reliability.

**Question 1-1:** Which modules/packages are most error prone?

**Entities:**

- Code modules/packages
- Issue tracker

**Attributes:**

- Distribution of defects over modules/packages
- Number of defects
- Size of code, size of modules/packages

**Question 1-2:** How often are errors found?

**Entities:**

- Issue tracker
- Developers

**Attributes:**

- Distribution of defect reports over time
- Amount of time between defect/fault reports

**Question 1-3:** What is the current state of defects?

**Entities:**

- Issue tracker
- Code

**Attributes:**

- Number of defects - open, closed, and total
- Defect state, open or closed
- Ratio of open to total defects
- Defect impact and/or severity
- Amount of time defect has been open
- Size of code

**Question 1-4:** Are defects often caught before code is released or deployed?

**Entities:**

- Issue tracker
- Code commits
- Developers
- Test suite

**Attributes:**

- Test suite failures

- Number of issues affecting code already released/deployed
- Test coverage for lines changed in each commit
- Number of defects found during each code review

**Subgoal 2:** Improve the software's efficiency.

**Question 2-1:** How much code is used to provide the product's current features?

**Entities:**

- Code
- Functionality requirements and specifications

**Attributes:**

- Size of code
- Number of features
- Amount of unused code
- Amount of duplicated code

**Question 2-2:** Is performance improving?

**Entities:**

- Functionality requirements and specifications
- Performance benchmarks (by use case)

**Attributes:**

- Percentage of requirements covered by performance benchmarks

- Amount of time to complete benchmarks

**Subgoal 3:** Improve the software's usability.

**Question 3-1:** Is the code well documented?

**Entities:**

- Code
- Code comments
- Documentation

**Attributes:**

- Size of code
- Amount of code comments
- Size of documentation
- Ratio of comments to code

**Question 3-2:** How easy is the code to read and understand?

**Entities:**

- Code
- Developers
- Issue tracker
- Style checker

**Attributes:**

- Size of code
- Distribution of code size amongst classes, functions, and modules

- Code complexity
- Nesting depth
- Depth of call hierarchies
- Percentage of defect corrections found to have their own defects
- Amount of code not conforming to style guidelines

**Subgoal 4:** Improve the software's maintainability.

**Question 4-1:** How much effort is spent on maintenance?

**Entities:**

- Code
- Issue tracker (e.g. features, defects)
- Developers

**Attributes:**

- Amount of code changed during maintenance efforts
- Development time required to fix defects
- Development time required to add features

**Question 4-2:** Are maintenance costs decreasing?

**Entities:**

- Issue tracker (e.g. defects, features)
- Developers

**Attributes:**

- Distribution of maintenance effort over time
- Amount of newly reported defects/features over time
- Average duration to fix reported defects or add new features (e.g. time from request to completion)

**Subgoal 5:** Improve the software's testability.

**Question 5-1:** How complete is the test suite?

**Entities:**

- Code
- Test suite

**Attributes:**

- Size of code
- Code coverage of executed test suite

**Subgoal 6:** Improve the software's flexibility.

**Question 6-1:** How long does it take to perform code modifications?

**Entities:**

- Code
- Issue tracker
- Developers

**Attributes:**

- Effort required to implement new features
- Duration between request and completion of features

**Question 6-2:** How isolated are code changes?

**Entities:**

- Code files, modules, classes
- Issue tracker (e.g. bugs, features)

**Attributes:**

- Distribution of the number of files/modules affected by each change (i.e. feature or defect correction)
- Distribution of class/module coupling

**Question 6-3:** How modular is the code?

**Entities:**

- Code modules and classes

**Attributes:**

- Inter-package/module/class dependencies/coupling
- Class cohesion

**Subgoal 7:** Improve the software's reusability.

**Question 7-1:** How often are existing code elements used?

**Entities:**

- Code classes, functions, methods

**Attributes:**

- Distribution of the number of references/calls/instantiations of classes, functions, and methods.

**Question 7-2:** Are there any missed opportunities for reuse?

**Entities:**

- Code

**Attributes:**

- Size of code
- Amount of duplicated code
- Ratio of duplicated code to overall code size

# Appendix B

# Developer Survey Questions

The sections in this appendix contain a text version of the online survey that was distributed to software professionals and presented in Chapter 4.

## B.1 Introduction

The goal of this survey is to evaluate how software industry professionals define software quality, and to understand how software methods, metrics, and visualizations are used within teams for software quality improvement.

**Who should take this survey?**

This survey is meant for individuals who are part of an organization or team whose primary function is to manage, plan, develop, test, and/or maintain software products.

## B.2 Background

1. Which of the following roles best fits your primary job responsibilities?

    - Manager
    - Business Analyst, Requirements Engineer
    - Developer, Software Engineer

- Test/QA Engineer

2. How many years of experience do you have in the software industry?

3. (Optional) What is the name of your company?

4. What is the size of your company?

    - up to 5
    - up to 50
    - up to 250
    - up to 1,000
    - over 1,000

5. What is the size of your team?

    - up to 5
    - up to 10
    - over 10

## B.3 Quality

1. Of the choices below, which do you feel best defines software quality?

    - Lack of defects, i.e. bugs, operational faults, etc.
    - "Fit of need", or how well the software meets customer requirements and/or expectations.

- A broad mix of factors including reliability, maintainability, reusability, etc.

2. If you feel that none of the definitions above accurately describe your definition of quality, please briefly describe your definition of quality in the space below.

3. Rank the code attributes below according to how important you feel the attribute weights in representing code quality. Place the most important attribute at the top and the least important attribute at the bottom.

    - Correctness - The extent to which the software meets its specifications and customers objectives.
    - Reliability - The software contains few bugs/defects and operates with few issues or faults.
    - Efficiency - The amount of code and/or resources required by the software to perform its functions.
    - Usability - The degree to which a software and/or code is easy to learn, operate, and read.
    - Maintainability - The amount of effort required to locate and fix errors.
    - Testability - The amount of effort required to ensure it performs its intended functions.
    - Flexibility - The effort required to modify or extend the software.
    - Reusability - The extent to which the software can be reused in other applications.

4. Does your team use any methods regularly as part of your development/release process for improving quality? Select all that apply from below:
   - Automated testing (Test-driven-development, unit testing, integration testing, UI testing, etc.)
   - Human QA testing
   - Code reviews
   - Pair programming
   - Formal meetings
   - Informal discussion with colleagues
   - Software metrics (complexity, mean-time-between-failure, coupling, etc.)
   - Other (please describe)
5. Please describe the methods your team uses for improving quality, including how and why you use them.
6. If your team does not use any methods for improving quality, please describe why not.
7. For a typical software project within your company, how influential are the factors below? Rank the following factors by placing the most important factor at the top and the least important factor at the bottom.
   - Cost
   - Quality
   - Schedule

## B.4 Metrics

1. Does your team currently utilize any software metrics for the intentional purpose of improving code quality?

    - Yes
    - No

2. How regularly does your team use metrics to evaluate...

| | Never | Rarely | Sometimes | Regularly | Constantly |
|---|---|---|---|---|---|
| ...the size of your software?<br>E.g. Lines of code, function points, number of files, modules, classes, functions, etc. | | | | | |
| ...the issues in your issue database?<br>E.g. number of open tickets, ticket age, mean time between issue, etc. | | | | | |
| ...the test suites or benchmarks within your software?<br>E.g. code coverage, benchmark timing, etc. | | | | | |
| ...the adequacy of documentation within your software?<br>E.g. number of comment lines, comment ratio, use case coverage, etc. | | | | | |
| ...the complexity of your software?<br>E.g. cyclomatic complexity, dependencies, call hierarchies, etc. | | | | | |
| ...maintenance effort performed?<br>E.g. time tracked on issues, issue age, issue backlog, code size changes, etc. | | | | | |
| ...the modularity of your software?<br>E.g. coupling, cohesion, dependencies, code duplication, etc. | | | | | |

3. For each metric listed below, select the choice that best describes your knowledge of and/or use of the metric.

| | Never heard of or do not know | Know but have not used | Have used before | Use regularly |
|---|---|---|---|---|
| Lines of code | | | | |
| Function points or similar | | | | |
| Number of classes/functions/files/modules | | | | |
| Halstead metrics | | | | |
| Defect count or distribution | | | | |
| Defect density | | | | |
| Mean time between defect/error | | | | |
| Time-to-fix defect | | | | |
| Defect slippage ratio | | | | |
| Unit test failures or pass/fail ratio | | | | |
| Line coverage | | | | |
| Branch/path coverage | | | | |
| Mutant killing percentage | | | | |
| Coupling | | | | |
| Cohesion (or lack of cohesion) | | | | |
| Cyclomatic complexity | | | | |
| Depth of inheritance tree | | | | |
| Depth of nesting | | | | |
| Fan-in and fan-out (class/module dependencies) | | | | |
| Duplicated code (or code clone) percentage | | | | |
| Code churn, turnover ratio, or edit frequency | | | | |
| Number of engineers | | | | |
| Number of ex-engineers | | | | |
| Depth of master ownership | | | | |
| Percentage of org contributing to development | | | | |
| Level of organizational code ownership | | | | |
| Overall organization ownership | | | | |
| Organization intersection factor | | | | |

4. In your company, are any software metrics used for evaluating employee performance?

    - Yes
    - No

5. Do you agree with employers using software metrics to evaluate employee performance?

| Strongly disagree | Disagree | Neutral | Agree | Strongly agree |
|---|---|---|---|---|
| | | | | |

## B.5 Visualization

1. Do you feel it is, or would be, useful to visualize metrics about your software projects over time?

| Strongly disagree | Disagree | Neutral | Agree | Strongly agree |
|---|---|---|---|---|
| | | | | |

2. Does your team currently use any tools to visualize software metrics over time?

    - Yes
    - No

3. How likely would you be to use a software metrics visualization tool to help improve the quality of software you write?

| Very unlikely | Unlikely | Neutral | Likely | Very Likely |
|---|---|---|---|---|
| | | | | |

4. If you were to use a tool to calculate and visualize metrics describing your software, how would you like to use the tool? Select all that apply.

    - Integrated into your editor or IDE
    - Integrated into your issue tracker
    - Integrated into your build/test server
    - A self-hosted, stand-alone product with a web interface and API access
    - A hosted service with a web interface and API access

- Other (please specify)

## B.6 Thank You

- (Optional) If it is OK to contact you for further research and clarifications on your answers to this survey, please fill in your email address below.
- (Optional) Use the space below to submit any comments to the researcher about this survey or related topics.

# Appendix C

# Developer Survey Responses

The sections in this appendix contain responses for free-form questions in the developer survey from Appendix B (evaluated in Chapter 4).

## C.1 Responses for question: Please describe the methods your team uses for improving quality, including how and why you use them.

- “Refactoring”
- “Use feature branches, one feature one branch, and never work alone, even if you have to force someone to look at your code. TDD when possible (aka convenient).”
- “TDD (jenkins) running after each push. Human QA for releases. Code reviews each iteration. Pair programming for complex tickets. Informal discussion when needed.”
- “We use also Sentry to monitor possible bugs”
- “Software Metrics, customer feedback, product throughput visualization”
- “Mostly human methods: code reviews, pair programming, and human QA testing. The code reviews and pair programming is intended to take care

of maintainability, bugs, code quality; human QA testing (based on formal, repeatable test plans) is to make sure the software is correct and usable. Automated testing is new for us; only about 25% of our products have it."

- "We start a task looking at the question "how do we know we are done?" That helps with coming up with Testing parameters as the languages we work in most of the time do not have unit test frameworks. This enables us to determine with the help of those who use the systems what is an acceptable fix or what problem is being solved."
- "rewrite code often"
- "Refactoring, bug tracking"
- "I personally use Test Driven Development. My closest team mates... not so much. They basically [...] make messes with the code.
- "Informal peer-design reviews."
- "There is a development process using established guidelines. This provides a clear path for development."
- "Historically we have used code reviews and pair programming, coupled with "human QA testing." We are just beginning to introduce automated integration tests. Technically, our "code coverage" is fairly low but we have still found the integration tests to be valuable."
- "Using the software ourselves, direct contact of developers with the customers (providing support)."

- "Automated build and quality check (valgrind for memory issue[s], cppcheck for static code analysis, Sonar for code duplication). Both end to end testing and unit testing. Component's bug heatmap: if a component [h]as many bugs, it likely means something: bad implementation, too complex (making it hard to think about all test cases...)."
- "Updating recommended coding practises, mentoring, project retrospectives"
- "Our human QA is performed by a small QA team who review requirements in order to build test cases and execute them. Code reviews are done by contractors assisting with the building of a new product."
- "We have CI that runs tests on every commit to ensure we're always ready to release (we do continuous deployment on most of our projects). We pair developers when doing changes to complicated or critical parts of the systems in development. We discover quite a lot of potential issues by simply discussing problems we fix each day. Having a whole team in a single room helps a lot. We measure test coverage (by lines and branches), SLOC and the usual CI build-breaker scores for each developer (chief build-breaker here)."
- "Sprint Reviews, Retrospectives, Code Reviews, Formal project reviews"
- "Software review"
- "jenkins and unit tests are the core, gerrit [for] reviews, and informal discussions when changes require fast deploy[ment] or are big changes"
- "Unit and regressi[on] testing and code reviews."

- "Spot-check testing by devs, starting to use some automated tests"
- "Pair programming and the occasional informal code review are the most important ways. Code reviews generally only occur when a senior member of the team has noticed a problem with a piece of work and wants the team to learn from it."
- "Our main method is unit testing and functional testing. On most new things we write, we include unit tests and functional tests. We're also trying to add unit testing to everything else that we touch. We've also started writing docs for those that test to give scenarios for testing."
- "TDD, Refactoring, adhering to "Clean Code" principles"
- "Load testing an order of magnitude beyond projected users Direct access (paid support) to the developers of the tools we use (e.g. database servers)"
- "Testing (automated and not, because some things can't be automated properly), bug trackers, http://www.slideshare.net/coordt/documentation-driven-development, etc."
- "ongoing education, unit tests"
- "We encourage each other to do things "the right way." We ensure requirements do not contain implementation details. Automated unit testing. Code reviews."

## C.2 Responses for question: If your team does not use any methods for improving quality, please describe why not.

- "Very tight deadlines. It's release tomorrow or never release."
- "I personally use Test Driven Development. My closest team mates... not so much. They basically [...] make messes with the code.
- "Software is developed by students working part time. There is no real experience in software quality and there are no guidelines. A new student looks at the code and starts to develop."
- "Formal meeting[s]: do not fit our culture. Software metrics: bad tool support at the time being."
- "We know the right things to do, but we don't have time to do them. Publishing dates and investor meetings always usurp good practices in the priority list."
- "We've done unit testing in the past. We don't do automated testing now because our contractors have indicated that it's quite difficult to perform such automated testing on code written on the framework to which we are moving. We've occasionally had formal and informal meetings in the past, but those who advocated for such have left the company recently. We've never experimented with pair programming or software metrics."
- "We don't have regular human QA testers because we're a small company. We only do informal code reviews. Teams are too small for formal reviews

and the code gets looked at anyway. No formal meetings, everything gets discussed on IRC or by the water-cooler."

- "We don't do much production software, it's mostly prototypes and proof-of-concept work that gets handed off to others, so our processes tend to be informal."
- "I've had to drive these quality efforts, often with great resistance. The culture is into getting features out the door rather than making them usable or maintainable. I suppose a lack of education retards the team."
- "More projects than time, no real management"

# Bibliography


[1] Allan J. Albrecht. Measuring application development productivity. In *Proceedings of the Joint SHARE/GUIDE/IBM Application Development Symposium*, volume 10, pages 83–92. SHARE Inc. and GUIDE International Corp. Monterey, CA, 1979.

[2] Allan J. Albrecht and John E. Gaffney. Software function, source lines of code, and development effort prediction: A software science validation. *Software Engineering, IEEE Transactions on*, SE-9(6):639–648, 1983.

[3] J. H. Andrews, L. C. Briand, and Y. Labiche. Is mutation an appropriate tool for testing experiments? In *Software Engineering, 2005. ICSE 2005. Proceedings. 27th International Conference on*, pages 402–411, 2005.

[4] Danielle G. T. Arts, Nicolette F. De Keizer, and Gert-Jan Scheffer. Defining and improving data quality in medical registries: A literature review, case study, and generic framework. *Journal of the American Medical Informatics Association*, 9(6):600–611, 2002.

[5] Rajiv D. Banker, Srikant M. Datar, Chris F. Kemerer, and Dani Zweig. Software complexity and maintenance costs. *Commun. ACM*, 36(11):81–94, Nov. 1993.

[6] Victor Basili, Roseanne Tesoriero, Patricia Costa, Mikael Lindvall, Ioana Rus, Forrest Shull, and Marvin Zelkowitz. Building an experience base for software engineering: A report on the first cebase eworkshop. In Frank Bomarius and Seija Komi-Sirviö, editors, *Product Focused Software Process Improvement*, volume 2188 of *Lecture Notes in Computer Science*, pages 110–125. Springer-Verlag Berlin Heidelberg, 2001.

[7] Victor R. Basili, Lionel C. Briand, and Walcélio L. Melo. A validation of object-oriented design metrics as quality indicators. *Software Engineering, IEEE Transactions on*, 22(10):751–761, 1996.

[8] Victor R. Basili, Gianluigi Caldiera, and H. Dieter Rombach. The goal question metric approach. *Encyclopedia of software engineering*, 2:528–532, 1994.

[9] Victor R. Basili and Barry T. Perricone. Software errors and complexity: an empirical investigation. *Communications of the ACM*, 27(1):42–52, Jan. 1984.

[10] Victor R. Basili and H. Dieter Rombach. Tailoring the software process to project goals and environments. In *Proceedings of the 9th international conference on Software Engineering*, ICSE '87, pages 345–357, Los Alamitos, CA, USA, 1987. IEEE Computer Society Press.

[11] Victor R. Basili and H. Dieter Rombach. The tame project: towards improvement-oriented software environments. *Software Engineering, IEEE Transactions on*, 14(6):758–773, 1988.

[12] Keith H. Bennett and Václav T. Rajlich. Software maintenance and evolution: a roadmap. In *Proceedings of the Conference on The Future of Software Engineering*, ICSE '00, pages 73–87, New York, NY, USA, 2000. ACM.

[13] Gerald M. Berns. Assessing software maintainability. *Commun. ACM*, 27(1):14–23, Jan. 1984.

[14] B. W. Boehm, J. R. Brown, and M. Lipow. Quantitative evaluation of software quality. In *Proceedings of the 2nd international conference on Software engineering*, ICSE '76, pages 592–605, Los Alamitos, CA, USA, 1976. IEEE Computer Society Press.

[15] Lionel C. Briand, John W. Daly, and Jürgen Wüst. A unified framework for cohesion measurement in object-oriented systems. *Empirical Software Engineering*, 3(1):65–117, 1998.

[16] Frederick P. Brooks. No silver bullet: Essence and accidents of software engineering. *IEEE Computer*, 20(4):10–19, April 1987.

[17] Nanette Brown, Yuanfang Cai, Yuepu Guo, Rick Kazman, Miryung Kim, Philippe Kruchten, Erin Lim, Alan MacCormack, Robert Nord, Ipek Ozkaya, Raghvinder Sangwan, Carolyn Seaman, Kevin Sullivan, and Nico Zazworka. Managing technical debt in software-reliant systems. In *Proceedings of the*

*FSE/SDP workshop on Future of software engineering research*, FoSER '10, pages 47–52, New York, NY, USA, 2010. ACM.

[18] Shyam R. Chidamber and Chris F. Kemerer. A metrics suite for object oriented design. *Software Engineering, IEEE Transactions on*, 20(6):476–493, 1994.

[19] Philip B. Crosby. *Quality is free: The art of making quality certain*. 1979.

[20] Ward Cunningham. The wycash portfolio management system. In *Addendum to the proceedings on Object-oriented programming systems, languages, and applications*, volume 18, pages 29–30, 1992.

[21] Tomi Dahlberg and Janne Jarvinen. Challenges to is quality. *Information and Software Technology*, 39(12):809–818, 1997.

[22] Richard A. DeMillo, Richard J. Lipton, and Frederick G. Sayward. Hints on test data selection: Help for the practicing programmer. *Computer*, 11(4):34–41, 1978.

[23] W. Edwards Deming. *Quality, productivity, and competitive position*. Massachusetts Institute of Technology, Center for Advanced Engineering Study, 1982.

[24] W. Edwards Deming. *Out of the Crisis*. Massachusetts Institute of Technology, Center for Advanced Engineering Study, 1986.

[25] Peter J. Denning. Editorial: what is software quality? *Communications of the ACM*, 35(1):13–15, Jan. 1992.

[26] Fernando Brito e Abreu and Rogério Carapuça. Candidate metrics for object-oriented software within a taxonomy framework. *Journal of Systems and Software*, 26(1):87–96, 1994.

[27] Norman E. Fenton and Martin Neil. A critique of software defect prediction models. *Software Engineering, IEEE Transactions on*, 25(5):675–689, 1999.

[28] Norman E. Fenton and N. Ohlsson. Quantitative analysis of faults and failures in a complex software system. *Software Engineering, IEEE Transactions on*, 26(8):797–814, 2000.

[29] Norman E. Fenton and Shari Lawrence Pfleeger. *Software metrics*, volume 1. Chapman & Hall, 1991.

[30] Beat Fluri, Michael Würsch, and Harald C. Gall. Do code and comments co-evolve? on the relation between source code and comment changes. In *Reverse Engineering, 2007. WCRE 2007. 14th Working Conference on*, pages 70–79, 2007.

[31] Harald Gall, Karin Hajek, and Mehdi Jazayeri. Detection of logical coupling based on product release history. In *Software Maintenance, 1998. Proceedings., International Conference on*, pages 190–198, 1998.

[32] Renato R. Gonzalez. A unified metric of software complexity: Measuring productivity, quality, and value. *Journal of Systems and Software*, 29(1):17–37, 1995.

[33] Robert B. Grady. Measuring and managing software maintenance. *Software, IEEE*, 4(5):35–45, 1987.

[34] Jim Gray. Why do computers stop and what can be done about it? Technical Report 85.7, Tandom Computers, June 1985.

[35] Maurice H. Halstead. *Elements of Software Science (Operating and programming systems series)*. Elsevier Science Inc., New York, NY, USA, 1977.

[36] Mikel Harry and Richard Schroeder. *Six Sigma: The breakthrough management strategy revolutionizing the world's top corporations*. Crown Business, 2006.

[37] Sallie Henry and Dennis Kafura. Software structure metrics based on information flow. *Software Engineering, IEEE Transactions on*, SE-7(5):510–518, 1981.

[38] Joseph R. Horgan, Saul London, and Michael R. Lyu. Achieving software quality with testing coverage measures. *Computer*, 27(9):60–69, 1994.

[39] Masaaki Imai. *Kaizen: The key to Japan's competitive success*. Random House Business Division, 1986.

[40] Alice L. Jacob and S. K. Pillai. Statistical process control to improve coding and code review. *Software, IEEE*, 20(3):50–55, 2003.

[41] M. Jørgensen. Software quality measurement. *Advances in Engineering Software*, 30(12):907–912, 1999.

[42] Bruno Laguë, Daniel Proulx, Jean Mayrand, Ettore M. Merlo, and John Hudepohl. Assessing the benefits of incorporating function clone detection in a development process. In *Software Maintenance, 1997. Proceedings., International Conference on*, pages 314–321, 1997.

[43] Michele Lanza. The evolution matrix: recovering software evolution using software visualization techniques. In *Proceedings of the 4th International Workshop on Principles of Software Evolution*, IWPSE '01, pages 37–42, New York, NY, USA, 2001. ACM.

[44] Dean Leffingwell and Don Widrig. *Managing software requirements: a use case approach*. Addison-Wesley, 2003.

[45] Meir M. Lehman and Laszlo A. Belady. *Program evolution: processes of software change*. Academic Press Professional, Inc., San Diego, CA, USA, 1985.

[46] Meir M. Lehman and Juan F. Ramil. Rules and tools for software evolution planning and management. *Annals of Software Engineering*, 11:15–44, 2001.

[47] Mier M. Lehman. Programs, life cycles, and laws of software evolution. *Proceedings of the IEEE*, 68(9):1060–1076, Sept. 1980.

[48] B. P. Lientz and E. B. Swanson. *Software Maintenance Management: A Study of the Maintenance of Computer Application Software in 487 Data Processing Organizations*. Addison-Wesley Pub (Sd), 1980.

[49] B. P. Lientz, E. B. Swanson, and G. E. Tompkins. Characteristics of application software maintenance. *Communications of the ACM*, 21(6):466–471, June 1978.

[50] M. Lipow. Number of faults per line of code. *Software Engineering, IEEE Transactions on*, SE-8(4):437–439, 1982.

[51] Thomas J. McCabe. A complexity measure. *Software Engineering, IEEE Transactions on*, SE-2(4):308–320, 1976.

[52] J. A. McCall, P. K. Richards, and G. F. Walters. Factors in software quality - concept and definitions of software quality. Technical Report RADC-TR-77-369, Vol. I, General Electric, Nov. 1977.

[53] J. A. McCall, P. K. Richards, and G. F. Walters. Factors in software quality - preliminary handbook on software quality for an acquisition manager. Technical Report RADC-TR-77-369, Vol. III, General Electric, Nov. 1977.

[54] James R. McKee. Maintenance as a function of design. In *Proceedings of the July 9-12, 1984, national computer conference and exposition*, AFIPS '84, pages 187–193, New York, NY, USA, 1984. ACM.

[55] Nachiappan Nagappan and Thomas Ball. Use of relative code churn measures to predict system defect density. In *Software Engineering, 2005. ICSE 2005. Proceedings. 27th International Conference on*, pages 284–292, 2005.

[56] Nachiappan Nagappan, Brendan Murphy, and Victor Basili. The influence of organizational structure on software quality: an empirical case study. In *Proceedings of the 30th international conference on Software engineering*, ICSE '08, pages 521–530, New York, NY, USA, 2008. ACM.

[57] Robert E. Park, Wolfhart B. Goethert, and William A. Florac. Goal-driven software measurement—a guidebook. Technical Report CMU/SEI-96-HB-002, Carnegie Mellon University, Software Engineering Institute, August 1996.

[58] David Lorge Parnas. Software aging. In *Proceedings of the 16th international conference on Software engineering*, ICSE '94, pages 279–287, Los Alamitos, CA, USA, 1994. IEEE Computer Society Press.

[59] Mark C. Paulk, Charles V. Weber, Bill Curtis, and Mary Beth Chrissis. *The capability maturity model: Guidelines for improving the software process*, volume 441. Addison-Wesley Reading, MA, 1995.

[60] Paul Piwowarski. A nesting level complexity measure. *SIGPLAN*, 17(9):44–50, Sept. 1982.

[61] Thomas C. Powell. Total quality management as competitive advantage: A review and empirical study. *Strategic Management Journal*, 16(1):15–37, 1995.

[62] Linda H. Rosenberg and Lawrence E. Hyatt. Software quality metrics for object-oriented environments. *Crosstalk Journal*, April 1997.

[63] Chanchal K. Roy and James R. Cordy. A survey on software clone detection research. Technical Report 2007-541, School of Computing, Queen's University at Kingston, Ontario, Canada, Sept. 2007.

[64] P. Runeson, C. Andersson, T. Thelin, A. Andrews, and T. Berling. What do we know about defect detection methods? *Software, IEEE*, 23(3):82–90, 2006.

[65] N. F. Schneidewind. The state of software maintenance. *Software Engineering, IEEE Transactions on*, SE-13(3):303–310, March 1987.

[66] N. F. Schneidewind. Methodology for validating software metrics. *Software Engineering, IEEE Transactions on*, 18(5):410–422, May 1992.

[67] Dirk Stelzer, Werner Mellis, and Georg Herzwurm. A critical look at iso 9000 for software quality management. *Software Quality Journal*, 6:65–79, 1997.

[68] Han Van Loon. *Process Assessment and ISO/IEC 15504: a reference book*, volume 775. Springer, 2004.

[69] J. Voas. Software's secret sauce: the "-ilities" [software quality]. *Software, IEEE*, 21(6):14–15, Nov.-Dec. 2004.

[70] Filippos I. Vokolos and Elaine J. Weyuker. Performance testing of software systems. In *Proceedings of the 1st international workshop on Software and performance*, WOSP '98, pages 80–87, New York, NY, USA, 1998. ACM.

[71] Elaine J. Weyuker and Filippos I. Vokolos. Experience with performance testing of software systems: issues, an approach, and case study. *Software Engineering, IEEE Transactions on*, 26(12):1147–1156, 2000.

[72] T.W. Williams, M.R. Mercer, J.P. Mucha, and R. Kapur. Code coverage, what does it mean in terms of quality? In *Reliability and Maintainability Symposium*, pages 420–424, 2001.

[73] J. H. Yahaya, A. Deraman, and A. R. Hamdan. Software certification model based on product quality approach. *Journal of Sustainability Science and Management*, 3(2):14–29, December 2008.

[74] Hong Zhu, Patrick A. V. Hall, and John H. R. May. Software unit test coverage and adequacy. *ACM Computing Surveys*, 29(4):366–427, Dec. 1997.

[75] T. Zimmerman, N. Nagappan, K. Herzig, R. Premraj, and L. Williams. An empirical study on the relation between dependency neighborhoods and failures. In *Software Testing, Verification and Validation (ICST), 2011 IEEE Fourth International Conference on*, pages 347–356, 2011.