Using Simulation to Build Inspection Efficiency Benchmarks for Development Projects

Lionel Briand, Khaled El Emam, Oliver Laitenberger
Fraunhofer IESE
Sauerwiesen 6
D-67661 Kaiserslautern
Germany
{briand,elemam,laiten}@iese.fhg.de

Thomas Fussbroich
Department of Computer Science
University of Kaiserslautern
D-67653 Kaiserslautern
Germany
fussbroi@informatik.uni-kl.de

International Software Engineering Network Technical Report ISERN-1997-21

Using Simulation to Build Inspection Efficiency Benchmarks for Development Projects

Lionel Briand, Khaled El Emam, Oliver Laitenberger
Fraunhofer IESE
Sauerwiesen 6
D-67661 Kaiserslautern
Germany
{briand,elemam,laiten}@iese.fhg.de

Thomas Fussbroich
Department of Computer Science
University of Kaiserslautern
D-67653 Kaiserslautern
Germany
fussbroi@informatik.uni-kl.de

Abstract

It is difficult for organizations introducing and using software inspections to evaluate how efficient they are. However, it is of practical importance to determine whether they have been effectively implemented or whether corrective actions are necessary to bring them up to standard. We present in this paper a procedure for building inspection efficiency benchmarks based on simulation and typical inspection data. Based on most of the data published in the literature, we build an industry-wide benchmark which intends to capture the current practice regarding inspection efficiency. Moreover, we discuss how this benchmark construction procedure can be used to build enterprise specific benchmarks. Last, we assess how robust we can expect them to be in varying conditions by distorting their input distributions.

Keywords
Software Inspection, Inspection efficiency, Quality modeling, Simulation.

1 Introduction

There exists a compelling business case for early defect detection using inspections. For example, some authors state that technical document reviews can reduce the number of defects reaching testing by ten times [13]. Furthermore, the ratio of the cost of fixing defects during inspections to fixing them during formal testing at JPL was found to range from 1:10 to 1:34 [21]. At the IBM Santa Teresa Lab the cost ratio was 1:20 [31], at the IBM Rochester Lab it was 1:13 [20], and the cost ratio for design inspections to testing at TRW was 1:6.25 [3]. The business case is even more impressive when one considers the cost of fixing defects postrelease. Other nonquantifiable benefits of inspections have been reported, such as promoting team spirit, the transfer of skills and facilitating on-the-job training [8].

As with the introduction of any other new technology, the introduction of inspections ought to be properly evaluated [22]. Furthermore, after initial implementation, inspections can be continuously evaluated and optimized to improve their performance.

In this paper, our focus is on evaluating the *efficiency* of inspections. Efficiency characterizes the cost-effectiveness of detecting defects by inspections. Cost effectiveness is defined as the extent to which the savings achieved are worth the costs.

From a practical perspective, the evaluation of efficiency implies two requirements. First, a measure of efficiency is necessary. Second, and more importantly, we need a means to interpret the measured value (i.e., is the obtained efficiency value good or bad?).

In the measurement literature, the interpretation of measured values can be *norm-referenced* or *criterion referenced* [1]. With norm-referenced evaluation, one compares the obtained value of inspection efficiency with the efficiency values obtained in other inspections. With criterion-referenced evaluation, one compares the obtained value with a threshold to determine whether it is above or below (i.e., better or worst).

Since there is no clear general basis for defining an efficiency threshold, the criterion-referenced approach would be difficult to operationalize. With the norm-referenced approach, one effectively defines a benchmark. An industry-wide benchmark can be constructed using data from the published literature. Published inspection data have been used in the past to evaluate inspections (see for example [15][26]), but no attempt to consolidate this information into an industry-wide benchmark has been made.

Using the efficiency measure developed by Kusumoto [24] as the basis, we construct three different 5 level industry-wide design and code inspections benchmarks for for three different defect detection life cycles within software development projects. The benchmarks use data from the published literature and represent companies that have successfully implemented inspections. The construction of the benchmark employs Monte Carlo simulation to take into account the uncertainty in the data obtained from the literature (e.g., the variation of application contexts and precision in measurement). Furthermore, we evaluate the benchmarks' robustness to violations of some important assumptions made during their construction, and demonstrate that most of the time they are sufficiently robust.

.

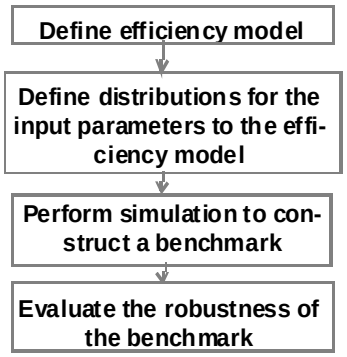

Figure 1: Benchmark development process.

In summary, then, we have packaged the current state of published knowledge about the efficiency of design and code inspections into a form that is usable for industry-wide benchmarking. Furthermore, the general procedure that we have followed can be used by organizations to develop their own internal enterprise-wide benchmarks.

The overall process that we follow is depicted in Figure 1. In Section 2 we review existing efficiency measurement models and identify the one most suitable for benchmarking. Section 3 presents the data set we used as the basis of our inspection efficiency benchmarks, and the distributions for the input parameters in the efficiency model. Section 4 describes the results of the simulation from which we derive benchmarks, and examines the robustness of the benchmarks. In Section 5 we conclude the paper with an overall summary and directions for future research.

2 Efficiency Models for Software Inspections

2.1 Notation

Below we define a general notation for characterizing defect detection phases in a project. This allows us to subsequently describe existing efficiency models in a consistent and unambiguous manner.

We assume that there are n defect detection phases that are performed sequentially[1]. Our assumed unit of analysis is the development project.

We define the effort consumed to detect and fix all defects during a defect detection phase f:

ε_f = effort spent on finding and fixing a defect in defect detection phase f [person-hours]

We define the set of unique defects existing in a document prior to a defect detection phase f as:

α_f = { x | x is a defect that exists in the document prior to defect detection phase f }

1.This does note exclude cycles during defect detection (e.g., inspecting a document for the second time) since each iteration can be considered a sequence.

The $|\alpha_f|$ value is difficult to measure accurately in practice. One approach is to estimate this by the total number of defects found by the time of completion of all defect detection phases f to n. An extension of this approach is to have $|\alpha_f|$ only equal the subset of total defects found by the time of completion that should have been caught by defect detection phase f. This is determined by an analysis of all defects and determining the source of the defect in the development life cycle.

We define the set of defects found during a defect detection phase f as:

λ_f = { x | x is a defect detected during defect detection phase f }

In the case of inspections, $|\lambda_f|$ would be the number of defects logged during a collection meeting for example.

The estimated cost of finding and fixing a defect in a defect detection phase f across all instances of f is defined as:

$$\hat{\varepsilon}_f = \overline{\left(\frac{\varepsilon_f}{|\lambda_f|}\right)} \quad \left[\frac{\text{person-hours}}{\text{defect}}\right]$$

where the bar on top indicates an average. The estimate can be calculated based on data from other projects, or can be calculated a posteriori for the project under study.

The *defect detection effectiveness* of a defect detection phase f across all instances of f is given by:

$$\hat{p}_f = \overline{\left(\frac{|\lambda_f|}{|\alpha_f|}\right)}$$

This can also be interpreted as the probability of finding a defect during phase f.

2.2 Literature Review

We now briefly present most of the efficiency models presented in the literature that we deemed relevant for our purpose. It should be noted that we only consider efficiency models that explicitly take cost into account. Therefore, even though Fagan [10], Jones [19], Remus [31] and Collofello and Woodfield [6] introduce models of *error detection efficiency* or *defect removal efficiency*, these models are not efficiency models according to our definition since they do not consider costs.

Basic Model

A simple model for the evaluation of efficiency is presented in [14]. This is a measure of how well effort consumed is made use of, and is defined as defects found per some unit of time (e.g., work-hour). In our notation, it is:

$$\hat{A}_f = \frac{1}{\hat{\varepsilon}_f}$$

However, this model does not account for defect detection

phase f compared to subsequent defect detection phases in the development life cycle (i.e., what is the relative advantage of detecting defects earlier). Therefore, only the costs of phase f are taken into account, but not any potential savings.

Collofello and Woodfield's Efficiency Model
Collofello and Woodfield [6] defined efficiency in general as:

$$\hat{Bf} = \frac{\text{Costs saved by the defect detection phase f}}{\text{Costs consumed by the defect detection phase f}}$$

Assuming a defect detection phase f detected and removed defects from a software artifact, we consider that if these defects had not been removed during f, they would have been removed in a later defect detection phase. Therefore, the potential costs associated with detecting and correcting the defects in later phases are saved by conducting f.

Based on this assumption, the costs saved by some phase f is calculated as the sum of the costs that would be incurred with having to use phases $(f+1)$ to n to handle the defects detected by phase f. Since the model was not formalized in the original paper, in the following we interpret and formalize this model

The full equation for efficiency can be expressed as:

$$\hat{Bf} = \frac{\sum_{i=f+1}^{n} \hat{\varepsilon}_i \times (\hat{p}_i \times |\gamma_i|)}{\hat{\varepsilon}_f \times |\lambda_f|}$$

where γ_i is the subset of defects that were detected by f, but that would escape detection in phases $f+1$ to $i-1$ if f was not performed.

For each phase i subsequent to phase f, where $f < i \leq n$, we calculate the costs saved as the product of the estimated effort per defect for phase i multiplied by the number of defects expected to be detected by phase i. The number of defects expected to be detected by phase i is calculated as the product of the effectiveness of phase i and $|\gamma_i|$.

The value of $|\gamma_i|$ is given by:

$$|\gamma_i| = \begin{cases} |\lambda_f| & , i = f+1 \\ |\lambda_f| \times \prod_{k=f+1}^{i-1} (1-\hat{p}_k) & , i > f+1 \end{cases}$$

This is the product of the number of defects that were detected and removed during phase f and the probability that the defect would not have been found during defect detection phases preceding i.

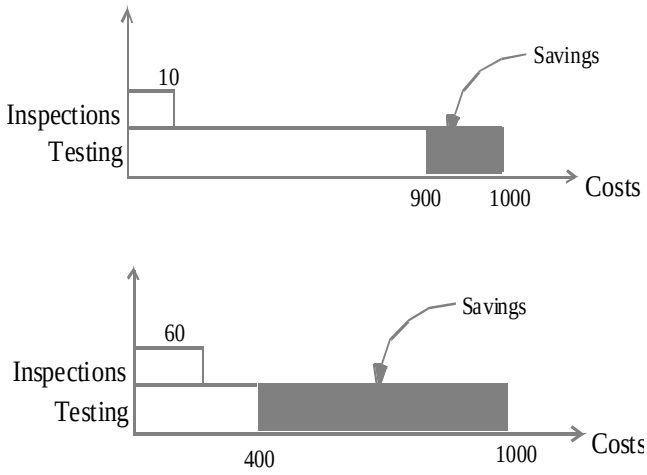

Figure 2: Comparing two different inspections.

ROI-model
An alternative model that builds on the Collofello and Woodfield work is the Return on Investment model used at HP [12]. In particular, the value of costs saved in the Collofello and Woodfield model does not consider the cost of defect detection phase f itself in calculating the savings. Therefore, the costs saved numerator is changed by subtracting the costs consumed by f. This is an improvement in that now we consider the real costs saved. Although in practice this correction involves the addition of a constant and therefore preserves the same order and distance amongst values of efficiency, it is a more valid formulation. Such a model can be expressed as follows:

$$\hat{C}_f = \frac{\left(\sum_{i=f+1}^{n} \hat{\varepsilon}_i \times (\hat{p}_i \times |\gamma_i|)\right) - (\hat{\varepsilon}_f \times |\lambda_f|)}{\hat{\varepsilon}_f \times |\lambda_f|}$$

Kusumoto's Efficiency Model
Kusumoto [24] noticed a discrepancy in the application of models such as C_f above. The discrepancy can de demonstrated with reference to the two projects in Figure 2. These projects only have two defect detection phases. In both projects, if inspections had not been done, the cost of testing would be 1000 units. The first inspection consumes 10 units of cost, and saves 100 units. Therefore, the total cost is 910. The second inspection costs 60 units, and saves 600. The total cost is 440. In the second case, inspections saved much more of total defect detection costs than the first, therefore one would expect it to be more efficient. However, using the ROI model, for example, would give them both an efficiency of 9.

To address this problem, Kusumuto proposes a new model.

Efficiency for the CT life cycle:

$$\hat{Ef} = \hat{p}_{\text{Code Inspection}} \times \left(1 - \frac{\hat{\varepsilon}_{\text{Code Inspection}}}{\hat{\varepsilon}_{\text{Test}}}\right)$$

Efficiency for the DT life cycle:

$$\hat{Ef} = \hat{p}_{\text{Design Inspection}} \times \left(1 - \frac{\hat{\varepsilon}_{\text{Design Inspection}}}{\hat{\varepsilon}_{\text{Test}}}\right)$$

Efficiency for the DCT life cycle:

$$\hat{Ef} = \hat{p}_{\text{Design Inspection}} \times \left(1 - \frac{\hat{\varepsilon}_{\text{Design Inspection}}}{(\hat{\varepsilon}_{\text{Code Inspection}} \times \hat{p}_{\text{Code Inspection}}) + (\hat{\varepsilon}_{\text{Test}} \times (1 - \hat{p}_{\text{Code Inspection}}))}\right)$$

Figure 3: Operational equations for the efficiency of the three life cycles.

He first introduces the concept of *virtual testing cost*. This is the total cost of testing if no inspections were conducted at all. He suggests that instead of using the costs consumed as the denominator, we should use the *total* potential cost that would be consumed had inspections not been conducted (i.e., the virtual testing cost).

Using our notation, Kusumoto defined efficiency as:

$$\hat{D}_{f=} \frac{|\lambda_f| \times \hat{\varepsilon}_{\text{testing}} - |\lambda_f| \times \hat{\varepsilon}_{\text{inspection}}}{|\alpha_f| \times \hat{\varepsilon}_{\text{testing}}}$$

This model can be generalized to any sequence of defect detection phases as follows:

$$\hat{Ef} = \frac{\left(\sum_{i=f+1}^{n} \hat{\varepsilon}_i \times (\hat{p}_i \times |\gamma_i|)\right) - (\hat{\varepsilon}_f \times |\lambda_f|)}{\sum_{j=f+1}^{n} \hat{\varepsilon}_j \times (\hat{p}_j \times |\alpha_j|)}$$

where:

$$|\alpha_j| = \begin{cases} |\alpha_f| & , j = f+1 \\ |\alpha_f| \times \prod_{k=f+1}^{j-1} (1 - \hat{p}_k) & , j > f+1 \end{cases}$$

The interpretation of this model is quite intuitive. It can be interpreted as the proportion of defect detection costs that are saved due to the introduction of phase f. For example, if the value is 0.25, this means that 25% of the defect detection costs are saved, and now only 75% of defect detection costs are consumed to find the same number of defects.

2.3 Discussion

Each successive model we presented above represents a further step in sophistication in capturing the efficiency of inspections. The most appropriate model is E_f, since it is addresses the weaknesses of previous efficiency models. However, there remains a set of assumptions embedded within this model. The assumptions are necessary to obtain an efficiency model that can be easily operationalized in practice. The assumptions are presented below:

1. No new defects are introduced between phases f and n.
2. A defect found in a phase i would result in a single defect in subsequent phases.

In practice, every model must have a predefined scope. The the scope of the efficiency model is defined by the sequence of defect detection phases. For any scope, it is necessary to assign the defect detection effectiveness of phase n to 1.

In our formalization above we do not take into account different defect types. While this would be a simple extension of the efficiency model, we do not do so because for the purposes of constructing an industry-wide benchmark, consistent data that would make use of such an extension was not found in the literature

3 Industry Data

In this section, we present the inspection data that match the model's input variables and published in the software engineering literature. It is expected that these data come from companies that have had some success in implementing inspections.

We consider three different defect detection life cycles for *development* projects. The first is code inspection, and testing (life cycle CT). The second is design inspections, code inspections, and testing (life cycle DCT), and the third design inspections and testing (life cycle DT). The

operationalization of efficiency for each of these three life cycles is presented in Figure 3.

The five values that we wish to obtain industrial values for are those that are based on historical data in our model. These are: $\hat{\varepsilon}_{\text{Design Inspection}}$, $\hat{\varepsilon}_{\text{Code Inspection}}$, and $\hat{\varepsilon}_{\text{Test}}$ (the average effort per defect for the three defect detection phases), and $\hat{p}_{\text{Design Inspection}}$ and $\hat{p}_{\text{Code Inspection}}$ (the effectiveness of design and code inspections).

3.1 Effectiveness of Defect Detection Phases ($\hat{p}_f$)

In the following, the articles that give data on effectiveness are discussed.

Fagan [10] presents data from a development project at Aetna Life and Casualty, Hartford, Connecticut, USA. An application program of eight modules (4439 non-commentary source statements) was written in `Cobol` by two programmers. Design and code inspections were introduced into the development process, the number of inspection participants ranged between three and five. After 6 months of actual usage, 46 defects had been detected during development and usage of the program. Fagan reports that 38 defects had been detected by design and code inspections together, yielding a defect detection effectiveness for inspections of 82%. The remaining 8 defects had been found during unit test and preparation for acceptance test.

In another article, Fagan [11] publishes data from a project at IBM Respond, United Kingdom. A program of 6271 LOC in `PL/1` was developed by 7 programmers. Over the life cycle of the product, 93% of all defects were detected by inspections. He also mentions two projects of the Standard Bank of South Africa (143 KLOC) and American Express (13 KLOC of system code), each with a defect detection effectiveness for inspections of over 50% without using trained inspection moderators.

Weller [34] presents data from a project at Bull HN Information Systems which replaced inefficient C code for a control microprocessor with Forth. After system test had been completed, code inspection effectiveness was around 70%.

Grady and van Slack [16] report on experiences from achieving widespread inspection use at HP. In one of the company's divisions inspections (focusing on code) typically found 60 to 70% of the defects.

Shirey [32] states that defect detection effectiveness of inspections is typically reported to range from 60 to 70%.

Barnard and Price [5] cite several references and report a defect detection effectiveness for code inspections varying from 30 to 75%. In their environment at AT&T Bell Laboratories, the authors achieved a defect detection effectiveness for code inspections of more than 70%.

McGibbon [26] presents data from Cardiac Pacemakers Inc. where inspections are used to improve the quality of life critical software. They observed that inspections removed 70 to 90% of all faults detected during development.

Collofello and Woodfield [6] evaluated reliability-assurance techniques in a case study - a large real-time software project that consisted of about 700,000 lines of code developed by over 400 developers. The project was performed over several years recording quality-assurance data for design, coding, and testing. The respective defect detection effectiveness are reported to be 54% for design inspections, 64% for code inspections, and 38% for testing. Although the authors state that testing efforts are normally identifiable as unit testing, integration testing, and acceptance testing, they do not provide more detail on the testing procedures in the project under examination.

Kitchenham et al. [23] report on experience at ICL, where 57.7% of defects were found by software inspections. The total proportion of development effort devoted to inspections was only 6%.

Gilb and Graham [14] include experience data from various sources in their discussion of the benefits and costs of inspections. IBM Rochester Labs publish values of 60% for source code inspections, 80% for inspections of pseudocode, and 88% for inspections of module and interface specifications.

Grady [15] performs a cost/benefit analysis for different techniques, among them design and code inspections. He states that the average percentage of defects found for design inspections is 55%, and 60% for code inspections.

Jones [27] discusses defect-removal effectiveness in the context of evaluating current practices in US industry. He gives approximate ranges and averages of defect detection effectiveness for various activities.

Franz and Shih [12] present data from code inspection of a sales and inventory tracking systems project at HP. This was a batch system written in COBOL. Their data indicate that inspections had 19% effectiveness for defects that could also be found during testing.

Meyer [27] performed an experiment to compare program testing to code walkthroughs and inspections. The subjects were 59 highly experienced data processing professionals testing and inspecting a PL/I program. Myers reports an average effectiveness value of 0.38 for inspections.

3.2 Average Effort per Defect($\hat{\varepsilon}_f$)

In this section, the data on the average effort per defect for various defect detection techniques (design inspections, code inspections, testing) is summarized.

Ackerman et al. [2] present data on different projects as a sample of values from the literature and from private reports. As the inspection process is described in the article as a six-step process including rework and follow-up, the data should mirror the cost of finding and fixing defects.

The development group for a small warehouse-inventory system used inspections on detailed design and code. For detailed design, they reported 3.6 hours of individual preparation per thousand lines, 3.6 hours of meeting time per thousand lines, 1.0 hours per defect found, and 4.8 hours per major defect found (major defects are those that will affect execution). For code, the results were 7.9 hours of preparation per thousand lines, 4.4 hours of meetings per thousand lines, and 1.2 hours per defect found.

A major government-systems developer reported the following results from inspection of more than 562,000 lines of detailed design and 249,000 lines of code: For detailed design, 5.76 hours of individual preparation per thousand lines, 4.54 hours of meetings per thousand lines, and 0.58 hours per defect found. For code, 4.91 hours of individual preparation per thousand lines, 3.32 hours of meetings per thousand lines, and 0.67 hours per defect found.

Two quality engineers from a major government-systems contractor reported 3 to 5 staff-hours per major defect detected by inspections showing a surprising consistency over different applications and programming languages.

A banking computer-services firm found that it took 4.5 hours to eliminate a defect by unit testing compared to 2.2 hours by inspection (probably code inspections).

An operating-system development organization for a large mainframe manufacturer reported that the average effort involved in finding a design defect by inspections is 1.4 staff-hours compared to 8.5 staff-hours of effort to find a defect by testing.

Weller [34] reports data from a project that performed a conversion of 1200 lines of C code to Fortran for several timing-critical routines. While testing the rewritten code, it took 6 hours per failure. It was known from a pilot project in the organization that they had been finding defects in inspections at a cost of 1.43 hours per defect. Thus, the team stopped testing and inspected the rewritten code detecting defects at a cost of less than 1 hour per defect.

McGibbon [26] discussed software inspections and their return on investment as one of four categories of software process improvements. For modeling the effects of inspections, he uses a sample project of an estimated size of 39.967 LOC. It is assumed that if the cost to fix a defect during design is 1X, then fixing design defects during test is 10X and in post-release is 100X. Thus, the rework effort per defect for different phases is assumed to be 2.5 staff hours per defect for design inspections, 2.5 staff hours for code inspections, 25 staff hours for testing, and 250 staff hours for maintenance (customer-detected defects).

Collofello and Woodfield [6] discuss a model for evaluating the efficiency of defect detection. In order to conduct a quantitative analysis, they needed to estimate some factors for which they had not enough data. They performed a survey among many of the 400 members of a large real-time software project who were asked to estimate the effort needed to detect and correct a defect for different techniques. The results were 7.5 hours for a design error, 6.3 hours for a code error, both detected by inspections, 11.6 hours for an error found during testing, and 13.5 hours for an error discovered in the field.

Kitchenham et al. [23] report on experience at ICL where the cost of finding a defect in design inspections was 1.58 hours.

Gilb and Graham [14] include experience data from various sources in their discussion of the benefits and costs of inspections. A senior software engineer describes how software inspections started at Applicon. In the first year, 9 code inspections and 39 document inspections (other documents than code) were conducted and an average effort of 0.8 hours was spent to find and fix a major problem. After the second year, a total of 63 code inspections and 100 document inspections had been conducted and the average effort to find and fix a major problem was 0.9 hours.

Bourgeois [4] reports experience from a large maintenance program within Lockheed Martin Western Development Labs (LMWDL) where software inspections replaced structured walk-throughs in a number of projects. The analyzed program is staffed by more than 75 engineers who maintain and enhance over 2 million lines of code. The average effort for 23 conducted software inspections (6 participants) was 1.3 staff-hours per defect found and 2.7 staff-hours per defect found and fixed. Bourgeois also presents data from Jet Propulsion Laboratory which is used as an industry standard. There, the average effort for 171 software inspections (5 participants) was 1.1 staff-hours per defect found and 1.4 to 1.8 staff-hours per defect found and fixed.

Franz and Shih's data [12] indicate that the average effort per defect for code inspections was 1 hour and for testing was 6 hours.

In presenting the results of analyzing inspections data at JPL, Kelly et al. [21] report that it takes up to 17 hours to fix defects during formal testing, based on a project at JPL. They also report approximately 1.75 hours to find and fix defects during design inspections, and approximately 1.46 hours during code inspections.

3.3 Modelling the Uncertainty of Data

One advantage of using data from the literature is that it represents a considerable number of project experiences with defect detection phases. This wide range of experiences, however, also introduces variation in the values relevant for our benchmarks. This variability is due to a combination of factors. We consider below the sources of this variability and how to model it.

Potential Causes of Variability in Literature Data

Three classes of factors can cause this variability: inherent variation, different implementations of defect detection techniques, and data errors.

It is to be expected that any phenomenon involving social

systems will exhibit some natural variability. This is due to the fact that all variation in such systems cannot be explained through some causal mechanism [28].

Different implementations of inspections can be characterized by:

1. Differences in the level of experience of the staff performing defect detection.
2. Differences in the defect detection process that is followed (e.g., inspections can be conducted with or without collection meetings with different amounts of rigour), differences in the number of participants in the inspections, differences in the number of inspection sessions, and the existence of tool support [30].
3. The data from the literature comes from different application contexts (e.g., different application domains).
4. Different types of artifacts of different levels of quality go through defect detection in the literature reports (e.g., some design documents may be harder to read than others).

In general, using data from the literature requires some caution due to the possible introduction of data errors [17]. In the case of inspections, in addition to clerical errors, the data errors can be characterized by:

1. Data from the literature may have different levels of precision due to different organizations exhibiting different amounts of rigour in their data collection.
2. Defect detection in different phases need not be independent from each other. For example, if earlier phases detect many defects, the defect detection probability might decrease in later phases because the remaining defects may be harder to find than the average defect. Therefore the effectiveness values obtained from the literature will sometimes include the effect of dependence on the previous defect detection phase (which reduces effectiveness), and sometimes not (if the defect detection phase was the first in the life cycle).
3. Imprecision in the reporting of data in the literature. For example, some papers report the average effort per defect. However, it is not always clear whether these refer to the effort to detect only or to detect and fix the defect.
4. The unit of analysis of calculations reported in the literature also can vary. For example, in some reports, defect detection effectiveness is calculated across all artifacts for a project. In others, the effectiveness for each artifact is calculated, then the overall effectiveness is computed as the average over all effectiveness values.

It is important to capture this variability because it reflects the uncertainty we have in the values that have been obtained from the literature. This can be achieved by modeling each of the variables as a distribution rather than a single value. By employing distributions, we account for the uncertainty introduced by using published data. Even within single enterprises, it is expected that there will be variation due to some of these factors.

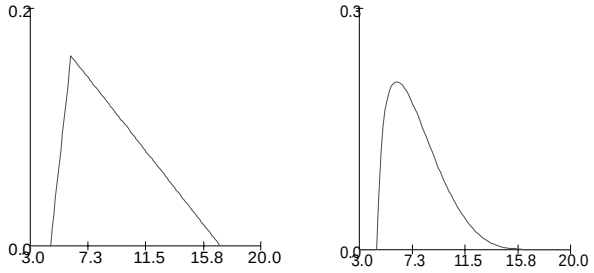

Figure 4: The shape of the prbability distribution for a triangular and a BetaPert distributions having a minimum of 4.5, a most likley value of 6, and a maximum of 17.

Modelling Variation

In order to model a variable's uncertainty, we chose to use the triangular distribution defined by a minimum, most likely, and maximum value triplet (see [9]). As discussed in [33], when little is known about the shape of the actual distributions, there is no compelling reason to use a distribution more complex than a triangular one. Although the triangular distribution gives more weight to the minima and maxima when compared, for example, to a BetaPert distribution (see Figure 4), the literature review provided us with what is referred to as "practical" minima/maxima [33], that is extreme but plausible situations. For prudence, we also evaluated the robustness of our benchmarks to usage of the BetaPert distribution. This is presented later.

In constructing distributions, not all the data available in the literature was used and only information resulting from precise data collection performed on actual defect detection activities was considered (i.e., we did not consider approximations or estimates, or data whose origins were unclear).

We applied the following rationale to use the published data. For data that comes from actual projects:

- the highest value is taken to be a potential maximum for the triangular distribution
- the lowest value is taken to be a potential minimum

If there are multiple values provided for the same project (e.g., different defect types), we took their average before applying the above two rules. For data that comes from many projects, we take the maximum and minimum of their range. If these are larger/smaller than these from individual projects above, then the multiple project maximum/ minimum become the values for the triangular distribution. The average of all of the remaining project values was taken as the most likely value.

Effectiveness of Design Inspections

The maximum value for the effectiveness of design inspections is reported in [14] from IBM Rochester Labs. This was 0.84 (average for two types of design document). The minimal value was reported by Jones as 0.25 for

informal design reviews [18]. The most likely value is the mid-point of the data from [6] and the industry mean reported in [18]. The final set of values are summarized in Table 1.

Effectiveness of Code Inspections
For the minimum value, only the data from [12] is used (0.19). The maximum value of 0.7 was obtained in [34]. For the most likely value, the data from [16][6][14][18][27] was used to produce an average. The final set of values are summarized in Table 1.

Defect detection technique	Minimum value	Most likely value	Maximum value
Design Inspections	0.25	0.57	0.84
Code inspections	0.19	0.57	0.70

Table 1: Probability distribution parameters for the effectiveness of different defect detection techniques

Average Effort per Defect for Design Inspections
Ackerman et al. [2] provide the maximum value of 2.9 hours on average per defect for different design documents on a project. The same article also provides the minimum value obtained from another project. The most likely value was the average of another project in [2], and [23][21]. The final set of values are summarized in Table 2.

Average Effort per Defect for Code Inspections
The maximum value for code inspections of 2.7 hours per defect was reported in [4]. The minimal value was reported in [2]. The most likely value was the mean of values reported in [2][34][12][21]. The final set of values are summarized in Table 2.

Average Effort per Defect for Testing
The maximum value of 17 was obtained from Kelly et al. based on a project at JPL [21]. The minimum of 4.5 was obtained from Ackerman et al. [2]. The most likely value was the mean for projects reported in [12][34] The final set of values are summarized in Table 2.

Defect detection technique	Minimum value	Most likely value	Maximum value
Design inspections	0.58	1.58	2.9
Code inspections	0.67	1.46	2.7
Testing	4.5	6	17

Table 2: Probability distribution parameters for the average effort using different defect detection techniques

4 The Benchmarks and Their Robustness

At this point, we have an efficiency model and a probabliity distribution function (PDF) for each of its input variables. Since these PDFs are taken to be somewhat representative of the current inspection practices in the software industry, we can produce an efficiency distribution which should be itself representative of industry performances. This is performed through Monte-Carlo simulations.

With Monte Carlo simulation, we sample independently a value from each input distribution, and then calculate the efficiency value according to our model. This procedure is repeated 2000 times, and we end up with a distribution of efficiency. We use the @RISK tool [29] to perform the simulations.

4.1 Simulation Results

Efficiency of Code Inspections
Figure 5 shows the simulation results for code inspection efficiency, and Table 3 summarizes the parameters for that distribution. As expected, it can be concluded that code inspections have a positive impact on software development cost. Compared to a defect detection life cycle of testing alone, the introduction of code inspections would save on average 39% of the defect detection costs.

Efficiency of Design Inspections
Figure 6 and Figure 7 show the probability density functions and Table 3 summarizes the distribution parameters for design inspections in two defect detection life cycles.

For the DT life cycle, it would be expected that design inspections would save on average 44% of the defect detection costs when compared to a testing only life cycle. Design inspections would also save, on average, 37% of costs when compared to a life cycle of code inspections and testing.

Summary
The simulation results indicate that the introduction of inspections to any existing defect detection life cycle during development will rarely if ever not result in cost savings. The maximum obtainable savings from introducing a single inspection phase, according to our simulations, will rarely if ever exceed 80%. We can calculate that, compared to a testing only life cycle, having design and code inspections results in savings, on average, of more than 60% of defect detection costs. The maximum that can be gained with the introduction of the two inspection types is approaximately 90% of the cost os using testing alone.

4.2 Setting up the Benchmarks

One common and easily understandable approach for interpreting scores through benchmarking is to use percentiles [25]. The results from the 2000 simulation runs are used to produce the percentile values. We use 20% intervals to get a 5-level benchmark. The values for these are shown in Table 4 for our three life cycles. One advantage of percentiles is that, since they use the same unit, they are comparable for different defect detection life cycles.

The interpretation of the values in Table 4 is relatively easy. For example, the 60% value indicates that 60% of the organizations will have larger inspection efficiency values

than 0.37 for the CT life cycle. This is equivalent to a level 3 on our benchmark. Therefore, if you have an efficiency value x such that $0.37 < x \leq 0.43$, then you are at level 3 on the benchmark. The benchmark is constructed such that higher levels are better (i.e., lower percentage of organizations will have greater efficiency values). The lowest level, level 1, is reserved for situations where efficiency is less than 80% of other inspections.

Life cycle	Minimum value	Median value	Mean value	Maximum value
CT	0.13	0.40	0.39	0.64
DT	0.16	0.44	0.44	0.75
DCT	0	0.37	0.37	0.71

Table 3: Statistical parameters of probability distribution functions for inspection efficiency

Percentile	**CT**	**DT**	**DCT**
5 (20%)	> 0.48	> 0.53	> 0.46
4 (40%)	> 0.43	> 0.47	> 0.39
3 (60%)	> 0.37	> 0.41	> 0.33
2 (80%)	> 0.31	> 0.35	> 0.27
1 ---	<= 0.31	<= 0.35	<= 0.27

Table 4: Inspection efficiency benchmarks for the three defect detection life cycles.

Being based on percentiles, the benchmark scale is strictly ordinal. We chose 5 levels to provide a reasonable amount of discriminatory power, but also to emphasize that this is an approximate benchmark (i.e., more than 5 levels would imply more precision than is warranted).

4.3 Robustness of the Benchmarks

General Approach

In developing our benchmarks we have made a number of assumptions. It is then prudent to evaluate the impact violations of these assumptions would have on the benchmarks. We consider two assumptions: the appropriateness of the triangular distribution to model the input variables, and the representativeness of the values from the literature.

For each assumption, we consider its violation(s). For each violation, we construct a new benchmark (referred to as the violation benchmark). There will be differences in the level classifications made by the violation benchmarks and the benchmarks in Table 4. If the violation is more concordant with reality, then we consider that the level assigned with the violation benchmark as the more correct one. Consequently,

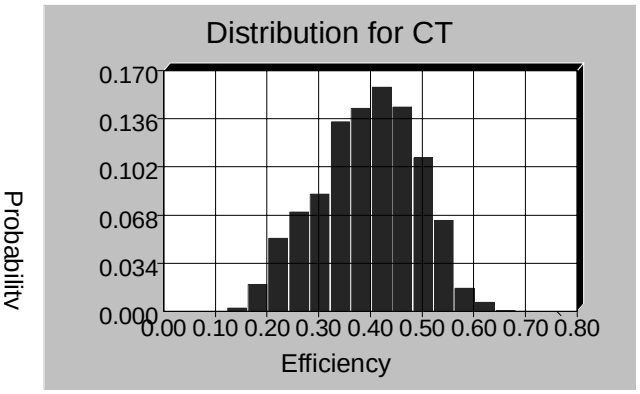

Figure 5: Simulation results for the CT life cycle.

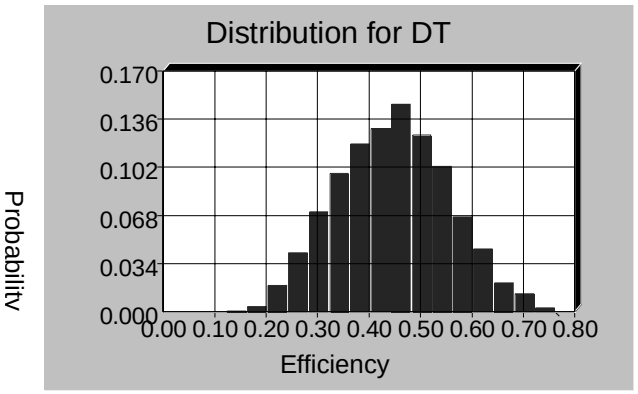

Figure 6: Simulation results for the DT life cycle.

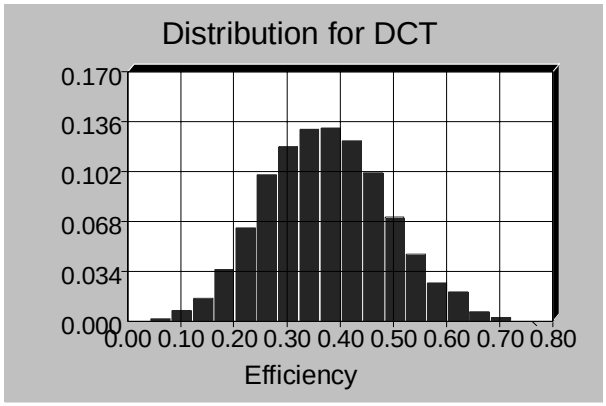

Figure 7: Simulation results for the DCT life cycle.

a difference between the level classification using a violation benchmark and the equivalent benchmark in Table 4 represents an error in the benchmark of Table 4. We then evaluate robustness by the percentage correct classifications. The higher this percentage, the more robust the benchmark of Table 4 to the violation...

To operationalize this procedure, we simulate 2000 inspections using each violation benchmark and compare the

violation level classification with that in Table 4. The percentage of correct classifications is calculated across these 2000 runs.

This percentage correct value gives the reader an indication of how tolerant our benchmarks are to increasing levels of violation of their assumptions. However, it is also necessary at some point to make a decision as to whether the benchmarks are sufficiently robust to be used. This then raises the issue of how high does the percentage have to be for it to be good enough.

While, for example, in statistics there are generally agreed-to threshold values for accepting or rejecting a null hypothesis, there are no equivalent threshold precedents in software engineering for benchmarks. Therefore, we can take as a guideline an accepted threshold from the arena of cost estimation: that 75% of the predicted values should be within 25% of their actual observations [7]. To interpret this in our context, we consider our benchmark to be robust if 75% of its classifications are within 1 level of the classification given by the new (violation) benchmark.

Using the above criterion to evaluate the robustness results of our benchmarks results in obtaining 100% of all classifications of our benchmarks within 1 level of the classification given by the violation benchmarks. Hence, it is a lax criterion to apply because it does not give an indication of how performance deteriorates as the violations become more extreme. We therefore consider 75% of classifications of our benchmarks that are *at the same level* of classification given by the violation benchmark as an acceptable threshold.

Robustness to Shape of Distribution

To evaluate the robustness of our benchmarks to the shape of the distribution, we evaluate the extent to which benchmark levels would be different had we used BetaPert distributions.

We found that we obtain 75% similar classifications for the CT life cycle, 91% for the DT life cycle, and 66% for the DCT life cycle. Therefore, it would seem that the DCT benchmark is sensitive to the shape of the distribution that is used, but the CT and DT life cycles are robust according to our criterion.

Robustness to Under-Representation of Low Performers

It can be argued that only organizations that have implemented inspections with reasonably encouraging result have publicized them. In addition, they would be the ones who are likely to have actually collected sufficient data to report. Therefore, it is plausible that projects at the low end of the inspection effectiveness scale are under-represented in the literature and hence are under-represented in our benchmark. Similarly, those projects at the high end of the inspections cost scale would be under-represented. The ensuing implication then is that the benchmark is too stringent since it is composed largely of successful implementations, which is not representative of the current industry-wide status.

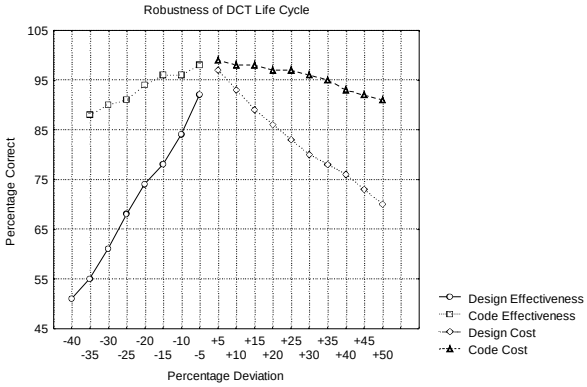

Figure 8: Sensitivity of the DCT life cycle.

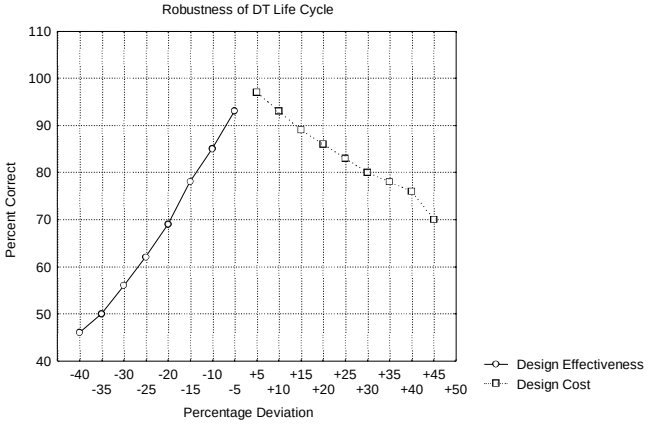

Figure 9: Sensitivity of the DT life cycle.

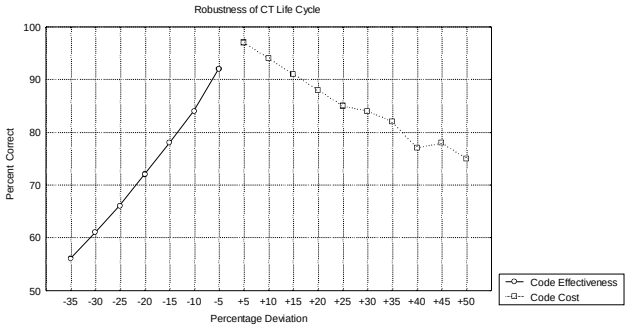

Figure 10: Sensitivity of the CT life cycle.

It is informative then to evaluate the robustness of the benchmark if the above situation was true. The approach we use follows the logic of the previous robustness evaluation. Assuming that the published data under-represents low performers, then we construct new distributions with incrementally decreasing performance that may be more representative. We did this separately for the three defect detection life cycles. We incrementally reduced the effectiveness of design and cost inspections until just above zero, and incrementally increased the costs of design and

code inspections for 10 increments, with each increment at 5% of the total range.

We start by presenting the results for the DCT life cycle since it is the most general. These are shown in Figure 8. The benchmark is not sensitive to reductions in the effectiveness of code inspections (since the percent similar classifications is above 75% down to an effectiveness of zero). As long as design inspection effectiveness is not in violation by more than 15%, then the percent similar classifications remains above 75%). It is not sensitive to the cost of code inspections increasing by up to 50%. Also, as long as the cost of design inspections is not in violation by more than 40%, then the percent similar classifications remains above 75%.

For the DT life cycle (see Figure 9), the percent similar classifications are above 75% if design effectiveness is not in violation by more than 15%, and if the cost of design inspections is not in violation by more than 40%. For the CT life cycle (see Figure 10), the percent similar is above 75% if the effectiveness of code inspections is not in violation by more than 15%, and if the cost of code inspections is not in violation by more than 50%.

In general then, as long as the effectiveness measures of potentially under-represented low-performing projects does not reduce the minima of our distributions by more than 15%, and their cost measures do not increase our maxima by 40%, then we can consider our benchmarks to be robust.

We contend that the literature would have to be quite biased against low-performing projects for violations as extreme as those above to be plausible. However, this remains an open question, and the plausibility of such violations can only be demonstrated by future empirical examples.

4.4 Improving the Benchmarks

While we have presented the method for constructing an efficiency benchmark, there remain a number of opportunities for improving the benchmark. Improving the benchmark means reducing the variability of the efficiency distribution.

The variability in the efficiency distribution is due to the variability of the input parameters. The reasons for this variability have been discussed earlier. It is then obvious that reducing the variability of input parameters would lead to reducing the variability of the efficiency distribution.

To reduce variability in the input parameters, one ought to collect more precise input data that give a more accurate representation of the past experiences that you want to base the benchmark on. For instance, one can develop distributions using data exclusively from similar projects (i.e., within the same organization in the same application domain). Also, improvement in the consistency of data collection and reporting could increase the precision of the $\hat{\varepsilon}_f$ and $\hat{p}_f$ distributions.

It may be difficult to collect more accurate and more relevant data for *all* of the input variables. The question then becomes: what input variable uncertainty should in priority be reduced to lower significantly the overall uncertainty of the benchmark?

To identify those input variables of the efficiency model that have the greatest impact on the uncertainty of the simulation outcome, we can consider the amount of variation in efficiency that is explained by each of the input variables. This form of sensitivity analysis can use bivariate correlation coefficients [33]. This can be defined intuitively as a way of computing the association between an input variable values and the simulation outcome values during the Monte-Carlo process. It can therefore be used to determine which input variable seems to influence most the efficiency during simulation.

We use both the Pearson product moment correlation and Spearman's rank order correlation, and obtained similar results. We therefore present the results only for the former in Table 5.

	CT Efficiency	DT Efficiency	DCT Efficiency
Effectiveness of Design Inspections	--	0.89	0.71
Effectiveness of Code Inspections	0.91	--	-0.22
Effort per Defect for Design Inspections	--	-0.29	-0.46
Effort per Defect for Code Inspections	-0.25	--	0.008
Effort per Defect for Testing	0.27	0.29	0.38

Table 5: Sensitivity analysis results.

These results indicate that, for each defect detection life cycle, the input variable that has the biggest impact on efficiency variation is the effectiveness of the defect detection phase under evaluation. For example, for the CT life cycle, the variable with the greatest correlation is the effectiveness of code inspections. Similarly, for the DT life cycle, it is the effectiveness of design inspections. In second position, with a more moderate correlation, is the effort per defect. For the DCT life cycle, the variables with the greatest influence are the effectiveness of design inspections and the effort per defect for design inspections.

To improve the benchmarks, the most important action would be to collect more precise data on the effectiveness of inspections. For industry-wide benchmarks, this would necessitate greater detail and consistency in the reporting of values in the literature. For enterprise-wide benchmarks, the

literature-derived cost values we present here can still be used, but the organization ought to focus on the collection of better effectiveness measures to obtain substantially more precise benchmarks.

5 Summary and Conclusions

We have constructed three efficiency benchmarks for design and code inspections based on most of the publicly available industry data. We have evaluated the robustness of these benchmarks to assumptions made during their construction, and found them to be robust. Although far from being fully satisfactory, these benchmarks can be used as a comparison baseline by companies using and especially introducing formal inspections. Such benchmarks should help them determine whether their inspection processes have a satisfactory efficiency level or whether corrective actions are needed.

The benchmark construction procedure presented in this paper can be used to guide organizations in building their own enterprise-wide benchmarks, which would in turn help them conduct internal comparisons of inspection processes.

Also, as a lesson learned, many of the data reported in the literature were not presented in a manner that would allow straightforward comparison and analysis. If, as a community, we are to make progress regarding the modeling of inspection efficiency, we need to be able to reuse published data, perform comparisons across organizations, and attempt some degree of meta-analysis. For that purpose, efficiency models need to be clearly defined, along with their underlying assumptions, and the raw data need to be presented at a level of granularity allowing their use for meta-analysis purposes.

REFERENCES

[1] M. Allen and W. Yen: *Introduction to Measurement Theory*. Brooks/Cole Publishing Company, 1979.

[2] A. Ackerman, L. Buchwald, and F. Lewski: "Software Inspections: An Effective Verification Process". *IEEE Software*, 6(3):31–36, May 1989.

[3] B. Boehm: *Software Engineering Economics*. Prentice-Hall, 1981.

[4] K. Bourgeois: "Process Insights from a Large-Scale Software Inspections Data Analysis." In *Cross Talk, The Journal of Defense Software Engineering*, 9(10):17–23, October 1996.

[5] J. Barnard and A. Price: "Managing Code Inspection Information". *IEEE Software*, 11(2):59–69, March 1994.

[6] J. Collofello and S. Woodfield: "Evaluating the Effectiveness of Reliability-Assurance Techniques". *Journal of Systems and Software*, 9(3):191–195, 1989.

[7] S. Conte, H. Dunsmore, and V. Shen: *Software Engineering Metrics and Models*. Benjamin Cummings, 1986.

[8] E. Doolan: "Experience with Fagan's Inspection Method". In *Software - Practice and Experience*, 22(2):173-182, February 1992.

[9] M. Evans, N. Hastings, and B. Peacock: *Statistical Distributions*. John Wiley & Sons, Inc., 1993.

[10]M. Fagan: "Design and Code Inspections to Reduce Errors in Program Development". In *IBM Systems Journal*, 15(3):182–211, 1976.

[11]M. Fagan: "Advances in Software Inspections". In *IEEE Transactions on Software Engineering*, 12(7):744–751, July 1986.

[12]L. Franz and J. Shih: "Estimating the Value of Inspections and Early Testing for Software Projects". In *Hewlett-Packard Journal*, pages 60-67, December 1994.

[13]D, Freedman and G. Weinberg: *Handbook of Walkthroughs, Inspections, and Technical Reviews*. Dorset House, 1990.

[14]T. Gilb and D. Graham. *Software Inspection*. Addison-Wesley Publishing Company, Wokingham, England, 1993.

[15]R. Grady. *Practical Software Metrics for Project Management and Process Improvement*. Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1992.

[16]R. Grady and T. Van Slack: "Key Lessons In Achieving Widespread Inspection Use". In *IEEE Software*, 11(4):46–57, July 1994.

[17]H. Jacob: *Using Published Data: Errors and Remedies*. Sage Publications, 1984.

[18]C. Jones: "Software Defect Removal Efficiency". In *IEEE Computer*, 29(4):94-95, April 1996.

[19]C. Jones: *Applied Software Measurement*, McGraw-Hill, 1991.

[20]S. Kan: *Metrics and Models in Software Quality Engineering*. Addison-Wesley Publishing Company, Inc., Reading, Massachusetts, 1995.

[21]J. Kelly, J. Sherif, and J. Hops: "An Analysis of Defect Densities found During Software Inspections". In *Proceedings of the Fifteenth Annual Software Engineering Workshop*, Goddard Space Flight Center, 1990.

[22]B. Kitchenham, L. Pickard, and S-L Pfleeger: "Case Studies for Method and Tool Evaluation". In *IEEE Software*, pages 52-62, July 1995.

[23]B. Kitchenham, A. Kitchenham, and J. Fellows: "The Effects of Inspections on Software Quality and Productivity". In *ICL Technical Journal*, 5(1):112–122, May 1986.

[24]S. Kusumoto. *Quantitative Evaluation of Software Reviews and Testing Processes*. PhD. Dissertation, Osaka University, September 1993.

[25]H. Lyman: *Test Scores and What They Mean*. Prentice-Hall, 1963.

[26]T. McGibbon: "A Business Case for Software Process Improvement". A DACS State-of-the-Art Report, September 1996. URL: http://www.dacs.com/techs/roi.soar/soar.html.

[27]G. Meyer: "A Controlled Experiment in Program Testing and Code Walkthroughs/Inspections". In *Communications of the ACM*, 21(9):760-768, September 1978.

[28]M. Morgan and M. Henrion: *Uncertainty: A Guide to Dealing with Uncertainty in Quantitative Risk and Policy Analysis*. Cambridge University Press, 1990.

[29]Palisade Corporation, Newfield, NY. *Guide to Using @RISK*, September 1996.

[30]A. Porter, H. Siy, and L. Votta: *A Review of Software Inspections*. Technical Report CS-TR-3552, Institute for Advanced Computer Studies, Department of Computer Science, University of Maryland, College Park, MD 20742, October 1995.

[31]H. Remus: "Integrated Software Validation in the View of Inspections/Reviews". In *Proceedings of the Symposium on*

Software Validation, H. L. Hausen (ed.), Elsevier Science Publishers, pages 57-64, 1984.

[32]G. Shirey: “How Inspections Fail”. In *Proceedings of the Ninth International Conference on Testing Computer Software*, pages 151–159, 1992.

[33]D. Vose. *Quantitative Risk Analysis: A Guide to Monte Carlo Simulation Modelling*. John Wiley & Sons Ltd, Chicester, England, 1996.

[34]E. Weller: “Lessons from Three Years of Inspection Data”. In *IEEE Software*, 10(5):38–45, September 1993.