A Review of Software Inspections

Adam Porter*, Harvey Siy
Computer Science Department
University of Maryland
College Park, Maryland 20742
{aporter,harvey}@cs.umd.edu

Lawrence Votta
Software Production Research Department
AT&T Bell Laboratories
Naperville, Illinois 60566
votta@research.att.com

June 3, 1996

Abstract

For two decades, software inspections have proven effective for detecting defects in software. We have reviewed the different ways software inspections are done, created a taxonomy of inspection methods, and examined claims about the cost-effectiveness of different methods.

We detect a disturbing pattern in the evaluation of inspection methods. Although there is universal agreement on the effectiveness of software inspection, their economics are uncertain. Our examination of several empirical studies leads us to conclude that the benefits of inspections are often overstated and the costs (especially for large software developments) are understated. Furthermore, some of the most influential studies establishing these costs and benefits are 20 years old now, which leads us to question their relevance to today's software development processes.

Extensive work is needed to determine exactly how, why, and when software inspections work, and whether some defect detection techniques might be more cost-effective than others. In this article we ask some questions about measuring effectiveness of software inspections and determining how much they really cost when their effect on the rest of the development process is considered. Finding answers to these questions will enable us to improve the efficiency of software development.

1 Introduction

For twenty years, software inspections have been described as one of the most cost-effective ways to improve the quality of computer software[5] . Although it's clearly an expensive process, its cost is often justified on the grounds that the longer a defect remains in a software system the more expensive it is to repair; therefore, the cost of finding defects today will always be less than the cost of repairing them in the future. However, this argument is simplistic – for example, it doesn't consider the powerfully negative effect inspections have on schedule.

We have observed that a typical release of AT&T's 5ESS® switch[30] ($\approx$ .5M lines of added and changed code per release on a base of 5M lines) can require roughly 1500 inspections, each with four, five or even more

*This work is supported in part by a National Science Foundation Faculty Early Career Development Award, CCR-9501354. Mr. Siy was also partly supported by AT&T's Summer Employment Program

participants. Scheduling so many meetings causes delays, lengthens cycle time and greatly increases cost. (In the case of one 5ESS release we estimate that inspections alone increased cycle time 10 weeks – from 60 to 70.)

Three expensive but often uncontested assumptions are that inspections must include group meetings, that inspection plus later testing is always much more cost-effective than testing alone, and that every part of every artifact must be inspected.

We have also seen that reviewers spend considerable amounts of time identifying and reporting issues that might be found more easily or prevented altogether with automated tools. As new tools appear, inspections may no longer be cost-effective for finding certain kinds of defects.

Although these are only examples, they reveal two fundamental problems that undermine the cost-effective use of software inspection: (1) the costs and benefits of software inspections haven't been adequately defined and therefore haven't been properly measured; and (2) the *causal agents* responsible for increasing the benefits and/or lowering the cost of inspections haven't been rigorously studied, making it impossible to determine how and when to best use inspections.

1.1 Levels of Analysis

These problems lead to two questions whose answers will help us understand exactly when inspections are justified for desired levels of cost, quality, and interval.

1. How should the costs and benefits of inspections be measured?
2. What factors significantly influence these costs and benefits?

Several studies have addressed these questions, usually at one of two different levels of analysis:

- Local Analysis – comparing inspection methods, but without regard to their effect on the entire development process, or
- Global Analysis – examining the effect of one or more inspection methods on the entire development process.

In this article we survey existing research with the goal of understanding how how well these questions have been answered at each level of analysis. We will also identify areas in which further work is needed.

2 The Software Inspection Process

To eliminate defects, many organizations use an inspection process with at least three steps: Preparation, Collection, and Repair. First, each member of a team of reviewers reads the artifact separately, detecting as many defects as possible. Next, these newly discovered defects are collected and discussed, usually at a team meeting. Then the author repairs them. Under some conditions an artifact can be inspected one or more times.

The several variations on this process are detailed in the following taxonomy of inspection methods.

2.1 Variations Among Different Inspection Methods

We choose to describe inspection methods based on the following attributes: (1) team size, (2) number of sessions, (3) coordination between multiple sessions, (4) collection technique, (5) defect detection method, and (6) use of post-collection feedback. Although other classification schemes could also be used, we believe these attributes represent underlying mechanisms that drive the costs and benefits of inspections.

Team Size. Team sizes can be *large* or *small*. The inspection team is normally composed of several reviewers. Presumably, this allows a wide variety of defects to be found since each reviewer relies on different expertise and experiences when inspecting. Thus, the larger and more varied the team, the better the coverage. However, large teams require more effort since more people analyze the artifact (which is often unfamiliar to them). This also reduces the time they can spend on other development work. In addition, it becomes harder to find a time a suitable meeting time as the number of attendees grows. Finally, it is more difficult for everyone to contribute fully during the meeting because of limited *air time*.

Smaller teams take less effort and are easier to schedule. However, they risk missing more defects and becoming superficial if personnel with required domain expertise are not included.

Number of Sessions. This refers to the number of times the artifact undergoes the inspection process, possibly with different teams of inspectors. Multiple-session inspections will find more defects as long as some important or subtle defects escape detection by any one inspection session. Also, splitting one large team inspection into multiple sessions with smaller teams might be more effective. The main problem with multiple sessions is that inspection effort increases as the number of sessions grows.

Coordination of Multiple Sessions. For multiple-session inspections, there is the additional option of conducting the sessions in *parallel* – with each session inspecting the same version of the artifact – or in *sequence* –

with defects found in one session being repaired before going on to the next session. Parallel sessions will be more effective only if different teams find few defects in common. They should also have nearly the same interval to completion as single-session inspections since the meetings can be scheduled to occur at nearly the same time. In addition, the author can collect all defect reports and do just one pass at the rework. But collecting the reports takes more effort, especially in sorting out which issues from different reports actually refer to the same defect in the artifact. Additionally, there might be conflicting issues which would take time to resolve.

Sequential sessions shouldn't duplicate issues since those found by an earlier team would have already been repaired. More defects may be found, since cleaning out old defects might make it easier to find new ones. However, it does take longer because the author cannot schedule the next phase of the inspection until defects from the first session have been resolved.

Collection Technique. This refers to whether an collection meeting is to be held (group-centered) or not (individual-centered). Although there is almost always some meeting between reviewers and the artifact's author to deliver the reviewer's findings, the goal of group-centered meetings is to find defects. Many people consider the meeting to be the central step of the inspection process because they believe that several people working together will find defects that none of them would find while working separately. This is known as "synergy". Meetings also serve as a way to spread domain knowledge since unfamiliar inspectors interact with more experienced developers. Finally, meetings provide a natural milestone for the project under development. It does however takes time and effort to schedule a meeting and recent studies have shown that meetings do not create as much synergy as previously believed[47] . In addition, the problems of improperly held meetings are well-documented[11, 34] . These include free-riding (one person depending on others to do the work), conformance pressure (the tendency to follow the majority opinion), evaluation apprehension (failure to raise a seemingly "stupid" issue for fear of embarrassment), attention blocking (failure to comprehend someone else's contribution and to build on it), dominance (a single person dominating the meeting), and others.

Individual-centered inspections sidestep these problems by eliminating the inspection meeting or de-emphasizing it (e.g., making it optional, making attendance optional, etc.). However, they risk losing the meeting synergy.

Defect Detection Method. Preparation, the first step of the inspection process, is accomplished through the application of defect detection methods. These are composed of defect detection techniques, individual reviewer responsibilities, and a policy for coordinating responsibilities among the review team. Defect detection techniques range in prescriptiveness from intuitive, nonsystematic procedures (such as *ad hoc* or *checklist* techniques) to explicit and highly systematic procedures (such as *scenarios* or *correctness proofs*).

A reviewer's individual responsibility may be general, to identify as many defects as possible, or specific, to focus on a limited set of issues (such as ensuring appropriate use of hardware interfaces, identifying untestable requirements, or checking conformity to coding standards).

Individual responsibilities may or may not be coordinated among the review team members. When they are not coordinated, all reviewers have identical responsibilities. In contrast, each reviewer in a coordinated team has different responsibilities.

The most frequently used detection methods (ad hoc and checklist) rely on nonsystematic techniques. Reviewer responsibilities are general and identical. However, multiple-session inspection approaches normally require reviewers to carry out specific and distinct responsibilities.

Use of Post-Collection Feedback. In most inspections, the author is left alone after the inspection meeting to analyze the issues raised and deal with the rework. Consequently, the development community may not learn why defects were made, nor how they they could have been avoided. Some authors argue that a brainstorming meeting should be held after the inspection meeting to determine the root cause of each issue recorded in the meeting.

The problems with this are the same as with other meetings: they require more effort and congest schedules as well as suffer from other group-interaction problems.

2.2 Example Inspection Methods

Fagan Inspections. In 1976, Fagan[15] published an influential paper detailing a software inspection process used at IBM. Basically, it consists of six steps. :

1. *Planning.* The artifact to be inspected is checked to see whether it meets certain entry criteria. If so, an inspection team, usually composed of up to four persons, is formed. Inspectors are often chosen from a pool of developers who are working on similar software, software that interfaces with the current artifact. The assumption is that inspectors familiar with the artifact will be more effective than those who aren't.

2. *Overview.* The author meets with the inspection team. He or she provides background on the artifact, e.g., its purpose and relationship to other artifacts.

Method	Team Size	No. of Sessions	Detection Method	Meet	Post
Fagan[15]	Large	1	Ad hoc	Yes	—
Bisant	Small	1	Ad hoc	Yes	—
Gilb[20]	Large	1	Checklist	Yes	Root cause analysis
Meetingless Inspection[47]	Large	1	Unspecified	No	—
ADR[35]	Small	>1 Parallel	Scenario	Yes	—
Britcher[6]	Unspecified	4 Parallel	Scenario	Yes	—
Phased Inspection[27]	Small	>1 Sequential	Checklist (Comp)	Yes (Reconcile)	—
N-fold[44]	Small	>1 Parallel	Ad hoc	Yes	—
Code Reading[33]	Small	1	Ad hoc	Optional	—

Table 1: **Example Inspection Methods.** This table compares the example inspection methods based on the inspection taxonomy.

3. *Preparation.* The inspection team independently analyzes the artifact and any supporting documentation and record potential defects.

4. *Inspection.* The inspection team meets to analyze the artifact with the sole objective of finding errors. The meeting is held on the assumption that a group of people working together finds defects that the members, working alone, would not.

   Before the meeting, one person is designated as the team leader or moderator, who orchestrates the meeting. Another person, designated as the reader, paraphrases the artifact. Defects are found during the reader's discourse and questions are pursued only to the point that defects are recognized. The issues found are noted in an inspection report and the author is required to resolve them. (Extensive solution hunting is discouraged during inspection.) The inspection meeting lasts no more than two hours to prevent exhaustion.

5. *Rework.* All issues noted in the inspection report are resolved by the author.

6. *Follow-up.* The resolution of each issue is verified by the moderator. The moderator then decides whether to reinspect the artifact depending on the quantity and quality of the rework.

Many software organizations have adopted this process (or a variation) for their own review procedures. The term "software inspection" is now almost exclusively associated with some form of this method and its variations.

Table 1 describes Fagan's method. It uses a large team of three or more persons; there is one session; the preparation uses ad hoc techniques; and there is a meeting.

Two-person Inspections. Bisant and Lyle[4] proposed reducing the inspection team to two persons: the author and one reviewer.

Table 1 describes Bisant and Lyle's method. It uses a small team of one reviewer; there is one session; the preparation uses *ad hoc* techniques; and there is a meeting between the sole reviewer and author.

Gilb Inspections. Gilb[20] inspections are similar to Fagan inspections, but introduces a *process brainstorming* meeting right after the inspection meeting. This step enables process improvement through studying and discussing the causes of the defects found at the inspection to find positive recommendations for eliminating them in the future. These recommendations may affect the technical, organizational, and political environment in which the developers work.

Table 1 describes Gilb's method. It uses a large team usually varying between four and six persons; there is one session; the preparation uses checklists, and there is an inspection meeting which is immediately followed by a root cause analysis meeting.

Meetingless Inspections. Many people believe that most defects are identified during the inspection meeting. However, several recent studies have indicated that most defects are actually found during the preparation step[39, 47] . Humphrey[22] states that "three-quarters of the errors found in well-run inspections are found during preparation." Votta[47] suggests replacing inspection meetings with *depositions*, where the author and, optionally, the moderator meet separately with each of the reviewers to get their inspection results.

Table 1 describes meetingless inspection. It uses many small (one-person) teams; there are multiple sessions (one per reviewer); the preparation technique is left unspecified; and there are no team meetings. Instead, the author meets with each reviewer separately.

Active Design Reviews. Parnas and Weiss[35] present *active design reviews* (ADR). The authors believe that in conventional design reviews, reviewers are given too much information to examine, and they must participate in large meetings which allow for limited interaction between reviewers and author. In ADR, the authors provide questionnaires to guide the inspectors. The questions are designed such that they can only be answered by careful study of the document. Some of the questions force the inspector to take a more active role than just reading passively. For example, he or she may be asked to write a program segment to implement a particular design in a low-level design document being reviewed.

Each inspection meeting is broken up into several smaller, specialized meetings, each of which concentrates on one attribute of the artifact. An example is checking consistency between assumptions and functions, i.e., determining whether assumptions are consistent and detailed enough to ensure that functions can be correctly implemented and used.

Table 1 compares ADR to the rest of the example methods. It uses small teams usually varying between 2–4 persons; there is more than 1 session; sessions are held in parallel with each examining one aspect of the artifact; the preparation uses questionnaires, a form of scenarios; and each session has a meeting.

Inspecting for Program Correctness. Britcher[6] takes ADR one step further by incorporating correctness arguments into the questionnaires. The correctness arguments are based on four key program attributes: Topology (whether the hierarchical decomposition into subproblems solves the original problem), Algebra (whether each successive refinement remains functionally equivalent), Invariance (whether the correct relationships among variables are maintained before, during, and after execution), and Robustness (how well the program handles error conditions).

By applying formal verification methods informally through inspections, this approach makes a compromise between the difficulty of scaling formal methods to large systems and the benefit of using systematic detection techniques in inspection.

Table 1 describes Britcher's method. The team size is left unspecified; there are four sessions, which may be held in parallel, with each session examining one aspect of the artifact; the preparation uses scenarios; and each session has a meeting.

Phased Inspections. Knight and Myers[27] present *phased inspections*, where the inspection step is divided into several mini-inspections or "phases." Standard inspections check for many types of defects in a single examination. With phased inspections, each phase is conducted by one or more inspectors and is aimed at detecting one class of defects. Where there is more than one inspector, they will meet just to reconcile their defect list. The phases are done in sequence, i.e., inspection does not progress to the next phase until rework has completed on the previous phase.

Table 1 describes phased inspections. It uses small teams usually varying between one and two persons; there is more than one session; sessions are held in sequence and each examines one aspect of the artifact; the preparation uses checklists; and each session with more than one reviewer includes a team meeting, held just to reconcile and consolidate the reviewer's defect lists.

N-fold Inspections. Schneider, et al.[44] , developed the *N-fold* inspection process. This is based on the hypotheses that a single inspection team can find only a fraction of the defects in an artifact and that multiple teams will not significantly duplicate each others efforts. In an N-fold inspection, N teams each carry out parallel, independent inspections of the same artifact. The results of each inspection are collated by a single moderator who removes duplicate defect reports.

Table 1 describes the N-fold inspections to the rest of the example methods. It uses large teams (three reviewers per team in their study); there is more than one session; sessions are held in parallel, with each session looking at all aspects of the artifact; the preparation uses ad hoc techniques; and each session includes a team meeting.

Code Reading. Code reading has been proposed as an alternative to formal code inspections[33] . In code reading, the inspector simply focuses on reading source code and looking for defects. The author hands out the source listings (1K-10K lines) to two or more inspectors who read the code at a typical rate of 1K lines per day. This is the main step. The inspectors may then meet with the author to discuss the defects, but this is optional. Removing the emphasis on meetings allows for more emphasis on individual defect discovery. In addition, the problems associated with meetings automatically disappear (including scheduling difficulties and inadequate air time).

Table 1 describes code reading. It uses a small teams; there are multiple sessions; the preparation uses ad hoc techniques; and holding a meeting is optional.

Code Reading by Stepwise Abstraction. Code reading by stepwise abstraction[2] is a code-reading technique. The inspector decomposes the program into a set of proper subprograms where a proper subprogram is a chunk of code that performs a single function that can be conveniently documented. A proper subprogram implementing a function that cannot be decomposed further is known as a prime subprogram. The program is decomposed until only prime subprograms remain. Then their functions are composed together to determine a function for the entire program. This derived function is then compared to the original specifications of the program.

3 Measuring the Costs and Benefits of Inspections

Software inspections are one of many techniques for improving the quality of software artifacts. Consequently, before choosing to perform inspections we should ascertain (1) the costs and benefits of individual inspection methods and (2) how the use of a given inspection method affects the costs and benefits of the entire software development process. This section discusses models for measuring the costs and benefits of software inspections and then presents examples of cost-benefit analyses from previous studies.

3.1 Local Analysis of Inspection Costs and Benefits

To measure the local costs and benefits of one or more inspection methods we can construct two models: one for calculating inspection interval and effort, and another for estimating the number of defects in an artifact. These models are depicted in Figure 1.

3.1.1 Modeling Local Cost

Two of the most important inspection costs are interval and effort. The inspection process begins when an artifact is ready for inspection and ends when the author finishes repairing the defects found. The elapsed time between these events is called the inspection interval.

The length of this interval depends on the time spent working (preparing, attending collection meetings, and repairing defects) and the time spent waiting (time during which the inspection is held up by process dependencies, higher priority work, scheduling conflicts, etc).

In order to measure inspection interval and its various subintervals, we devised an inspection time model based on visible inspection events [50] . Whenever one of these events occurs it is timestamped and the event's participants are recorded.

These events occur, for example, when the artifact is ready for inspection, or when a reviewer starts or finishes his or her preparation. This information is entered into a database, and inspection intervals are reconstructed by performing queries against the database. Inspection effort can also be calculated using this information.

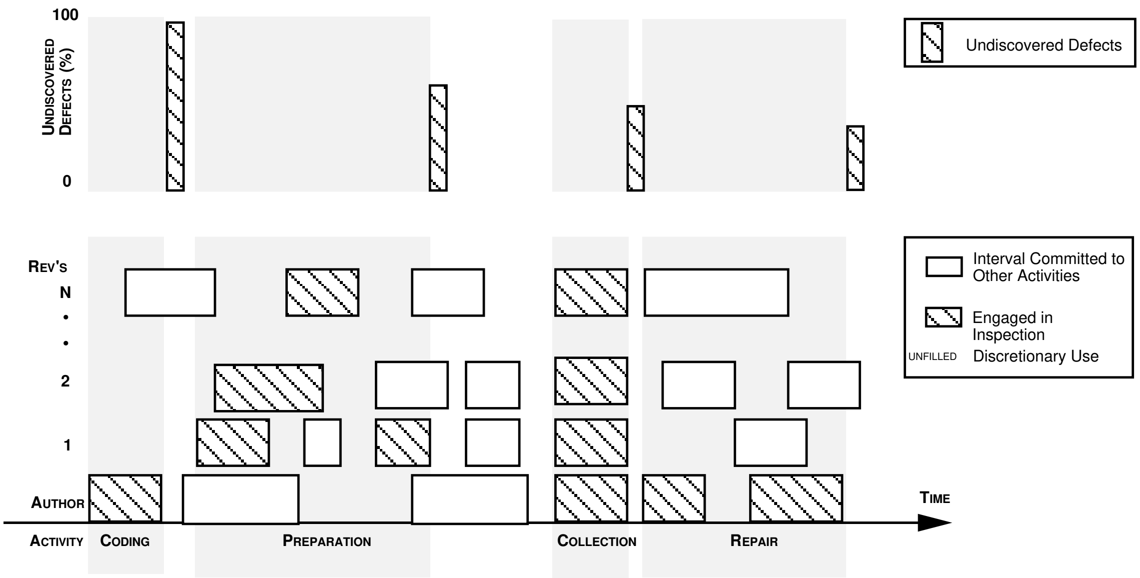

Figure 1: This figure depicts a simple model of the inspection process. The figure's lower panel summarizes the inspection's time usage. Specifically, it shows the inspection's participants (an author and several reviewers), the activities they perform (coding, preparation, collection, repair, and other), the interval devoted to each activity (denoted by the shaded areas), and the total inspection interval (from end of coding to completion of repair). It also shows how inspections must compete with other development processes for limited time and resources. The upper portion of the figure shows when and to what extent inspections remove defects from the artifact.

3.1.2 Modeling Local Benefit

The most important benefit of an inspection is its effectiveness, and one important measure of an inspection's effectiveness is its defect detection ratio – the number of defects found during the inspection divided by the total number of defects in the artifact. Because we never know exactly how many defects an artifact contains, it is impossible to make this measurement directly, and therefore we are forced to approximate it.

Several methods can provide these approximations. Each differs in their accuracy (how close they come to the true measure), and their availability (how early in the software development process they can be applied).

- **Observed detection ratio:** Assume that total defect density is constant for all artifacts of the same type and that we can compare the observed defect densities. This is always available, but very inaccurate.

- **Partial estimation of detection ratio:** Statistical methods such as capture-recapture estimation can be used to estimate pre-inspection defect content[14, 46] . This method can be used when there are at least two reviewers and they discover some defects in common. Under these conditions this method can be more accurate than the observed detection ratio and is available immediately after every inspection.

- **Complete estimation of detection ratio:** Track the artifact through testing and field deployment, recording new defects as they are found. This is the most accurate method, but is not available until well after the project is completed.

3.1.3 Assessing Local Costs and Benefits

In this section we survey previous work, showing how each study justified the costs and benefits of its proposed inspection method.

Anecdotal Studies. The cost-effectiveness of a method may be described anecdotally. Parnas and Weiss[35] applied ADR on an actual review of the design document for the operational flight program of one of the Navy's aircraft.

Case Studies. An implied requirement of inspections is understanding the artifact being reviewed. Rifkin and Deimel[41] suggest teaching program comprehension techniques during code inspection training classes in order to improve program understanding during preparation and inspection. Using historical data they argued that while inspections reduced the number of defects discovered by testing, they did not significantly decrease the number of customer-identified defects.

Rifkin and Deimel hypothesized that introducing inspections have had little effect on reducing customer-identified defects because, although reviewers were being thoroughly trained in the group aspects of the inspection process, they were being given little guidance how to analyze a software work product.

To test this hypothesis, they collected data from three software development groups, each composed of 30–35 professionals. Everyone was familiar with the inspection process. One group was given 1.5 days training in program reading comprehension. The variable being measured was the number of customer-identified defects reported to each group per day.

The data showed that the number of customer-reported defects dropped by 90% after the reviewers received reading comprehension training, while results of the other two groups of reviewers showed no change.

Controlled Experiments. Bisant and Lyle[4] ran an experiment using two sets of student projects in a programming language class to study the effects of using a two-person inspection team, with no moderator, on programmer productivity, or time to complete the project. The experiment used a pretest-posttest, control group design. The students were divided into an experimental group, which held inspections, and a control group, which did not. There were 13 students in the experimental group and 19 students in the control group. Both groups did not inspect their design or code during the first project. For the second project, the members of the experimental group were asked to inspect, along with a classmate, each other's design or code. The results showed that the programming speed of the experimental group improved significantly in the second project.

Knight and Myers[27] carried out an experiment involving 14 graduate students and using a phased inspection with four phases. Each student was involved in exactly one of the phases. The artifact was a C program with more than 4,000 lines and 45 seeded defects, whose types were distributed across those which the four phases are expected to find. The inspections raised a total of 115 issues. (Of these, only about 26 appear to affect the execution of the program.) The inspectors also found 30 of the 45 seeded defects. The amount of effort totaled 66 person-hours. This was determined from the usage of the inspection tool, and from the meeting times of the of the phases using more than one inspector.

Acknowledging that they cannot make definitive comparisons, Knight and Myers found it interesting to compare their results to Russell[43] ; which are also described in Section 3.2. They show that while Russell found 1 defect per hour, the phases found 1.5 to 2.75 defects per hour.

Mathematical Modeling. To test the cost-effectiveness of meetingless inspections, Votta[47] collected data from 13 inspections with meetings. He modeled the effort needed to hold depositions by the following formula:

$$E_{depositions} = 3ka_d + t * 3Sum(p_i)$$

where

k = number of reviewers (apart from the moderator and recorder)

a_d = overhead time of starting and stopping a deposition (assume 10 minutes)

p_i = the fraction of faults found by the i^{th} reviewer

t = inspection time (assume 2 hours).

The model suggests that depositions would always take less effort than an inspection meeting, as long as the number of reviewers is not greater than 20. Their actual data showed that foregoing inspection meetings would however reduce the percentage of defects found by only 5%.

3.2 Global Analysis of Inspection Costs and Benefits

The rationale most often used to justify inspections is that it's cheaper to find and fix defects today than it is to do it later. Several studies have evaluated this conjecture by (1) measuring the costs and benefits of inspections (local analysis) and by (2) estimating the effect of inspections on the rest of the development process (global analysis).

Global analysis usually involves evaluating alternative scenarios (i.e., if we hadn't found those 20 defects during the inspection, how much more testing and rework would we have had to do?) This information is normally extrapolated from historical data and requires that the analyst make strong assumptions about its representativeness. As a result any analysis of the global cost-benefits of inspections must be examined critically.

3.2.1 Modeling Global Cost

The costs of performing inspections include the local costs described in Section 3.1 as well as any costs that stem from including inspections in the development process, for example, duplicating inspection artifacts and maintaining inspection reports. Another significant cost comes from increasing schedule. Inspections, like other labor-intensive processes, require group meetings, which can cause delays and increase interval. Since longer intervals may incur substantial economic penalties, this cost must be considered. Extra interval can lead to:

- late market entry - products that enter the market when there are few competitors often do better than technically superior products that enter later when there are more competitors;
- opportunity costs - resources devoted to one product can't be used on others;
- carrying costs - the longer it takes to build a product, the higher the cost of maintaining hardware labs, office space, etc.

Since these costs are difficult to quantify, we believe that the cost of inspections is often underestimated.

3.2.2 Modeling Global Benefit

Inspections provide the direct benefit of finding defects. Many people believe that they also positively affect later stages of development by reducing rework, testing, and maintenance. As we mentioned earlier, measuring these benefits directly is impossible and therefore they must be estimated. Of course, any attempt to do this will involve making certain assumptions about how observed data relates to the values being estimated. This section examines several commonly made assumptions and explains why the some studies may be overstating the benefits of inspections.

A1. **All defects not found at an inspection would be shipped with the delivered system.** Several articles compute the benefit of finding a defect during an inspection by equating it with the observed cost of finding and fixing defects that appear in the field. However, some of these defects would be found by another means prior to system release.

A2. **Inspections find the same type and distribution of defects as testing.** Other authors calculate the benefit of finding a defect during inspection by equating it with the average cost to find defects during testing. For example, if it was determined that finding and repairing a defect during testing costs an average of 10 hours per defect, then the cost of finding and repairing a defect found in inspection is also equated to 10 hours per defect.

One of the problems with this approach is that inspections may not find the same classes of defects as testing. For example, inspections turn up many issues which do not affect the operational behavior of the system. Figure 2 shows that in an industrial case study of more than 100 inspections, 60% of all issues recorded during an inspection meeting fall into this class[40] . These defects will never be found by testing. In another example, some studies have shown that almost half the defects found in testing are interface defects[38] , suggesting that inspections are not effectively finding this class of defects, even though effort is spent looking for them.

A3. **Each defect found during inspection results in a linear reduction in testing effort.** Another problem with equating the benefit of finding defects at inspections with the average cost of testing is that finding defects during inspection does not proportionately reduce the effort spent in testing. For the example given in A2, if 40 defects were found during inspection, it is usually estimated that $40 \times 10 = 400$ hours of testing will be saved. However, in our experience, testers make no assumptions on the reliability of pretested code and will run the same test suites whether the code was inspected or not. The amount of time spent testing depends more on the resources that are available and the desired reliability than the exact number of defects. Also, as testing progresses and defects are removed, it often takes longer to find

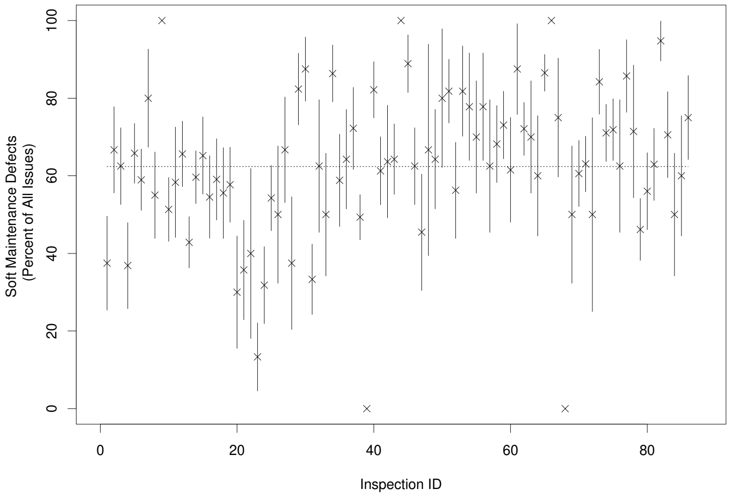

Figure 2: Percentage of soft maintenance defects recorded per inspection meeting. The term "soft maintenance defects" refers to defects which do not affect the operational behavior of the software system. The dashed line shows the mean percentage.

new defects. Therefore defects found later in testing may disproportionately increase the mean number of hours to find and fix defects in testing.

A4. **Inspection costs and benefits aren't affected by changing technology.** Several early studies of inspections studied the cost and benefits they provide. In the intervening 20 years, changes in technology have changed these tradeoffs. For example, Perry and Evangelist[37] suggest that there are significant savings in finding and repairing interface defects when formal semantic information is added to subprogram interfaces and then the software is analyzed using tools like Inscape[36], App[42] and Aspect[23]. Also, for code, fast machines make extensive unit testing possible which again changes the benefits of inspecting. Finally, several early articles equate machine effort with human effort. Clearly one hour of human effort may be more expensive than one hour of machine effort.

3.2.3 Assessing Global Costs and Benefits

In this section, we present examples of cost benefit analyses from previous papers on software inspections. We evaluate each one in the context of the four assumptions stated in Section 3.2.2.

The reader must be cautioned that claims on improvement cited by each study occurred within specific development environments, under the influence of many factors not directly related to inspection such as design notation, programming language, development processes, available hardware, process maturity, artifact size, etc. Also, units of measurement may have differing operational definitions.

Fagan[15] studied the use of design and code inspections on an IBM operating system component. The data was compared against that for similar components which did not use inspections. The results showed an increase in productivity, attributed to a minimized overall amount of error rework. For instance, there was a 23% increase in coding productivity compared to projects which did not use inspections. Design and code inspections resulted in a net savings of 94 and 51 person-hours per KNCSL, respectively. This included the cost of defect rework, which was 78 and 36 person-hours per KNCSL for design and for code inspections, respectively. It should be noted that this data is 20 years old! As explained in assumption A4, the advertised benefits may have diminished over the years as technology, defect prevention methods, and software development skills improved.

In a follow-up study, Fagan[16] summarized several industrial case studies of inspection performance. His conclusions were that inspections of a 4,000 line program at AETNA Life and Casualty and a 6,000 line program at IBM detected 82% and 93%, respectively, of all defects detected over the entire life cycle of the programs; that the inspection of a 143,000 line software project at Standard Bank of South Africa reduced corrective maintenance costs by 95%; and that inspection of test plans and test cases for a 20,000 line program at IBM saved more than 85% of programmer effort by detecting major defects through inspection instead of testing.

Russell[43] observed inspections for a two year period at Bell-Northern Research. These inspections found about one defect for every man-hour invested in inspections. He also concluded that each defect found before it reached the customer saved an average of 33 hours of maintenance effort. As the following excerpt shows, the article assumes that the benefit of finding a defect during inspection equals the cost of fixing it after the software has been released.

> Here's some more perspective on this data. Statistics collected from large BNR software projects show that each defect in software released to customers and subsequently reported as a problem requires an average of 4.5 man-days to repair. Each hour spent on inspection thus avoids an average of 33 hours of subsequent maintenance effort, assuming a 7.5-hour workday.

Using assumption A1 Doolan[13] calculated that inspecting requirements specifications at Shell Research saved an average of 30 hours of maintenance work for every hour invested in inspections (not including rework).

Bush[8] related the first 21 months of inspection experience at the Jet Propulsion Laboratory. In that time 300 inspections had been conducted over 10 projects. She calculated that inspections cost \$105 per defect. (The effort to find, fix, and verify the correction of a defect varies between 1.5 and 2.1 hours, corresponding to a cost between \$90 and \$120 or an average of \$105.) But this saved them \$1,700 per defect in costs which would have been incurred by testing and repair. (It was not explained how this value was calculated.) The papers assumes that finding and fixing a defect during inspection costs the same as finding and fixing a defect during test (assumption A2).

Kelly, et al.[26] , report on 203 inspections at the Jet Propulsion Laboratory. They showed that inspections cost about 1.6 hours per defect, from planning, overview, preparation, meeting, root cause analysis, rework, and follow-up. This is less than the 5 to 17 hours required to fix defects found during formal testing. Although this calculation requires assumption A2, many of the defects found didn't affect the behavior of the software and wouldn't have been caught by testing.

Weller[49] relates 3 years of inspection experience at Bull HN. In one case study, data at the end of system test showed that inspections found 70% of all defects detected up to that point. In the same project, which was to replace C code with Forth, the developers had initially decided not to do any inspections on the rewritten code, but found that testing was taking six hours per failure. After inspections were instituted, they began to find defects at the cost of less than one hour per defect. In another case study, inspections of fixes dropped the number of defective fixes to half of what it had been without inspections.

Franz and Shih[18] report the effects of using inspection on various artifacts of a sales and inventory tracking project at Hewlett-Packard. They calculate that inspections saved a total of 618 hours (after taking into account the 90 hours needed to perform the inspections). The total time saved by inspection is the time saved in system test plus the time saved by reduced maintenance. System test time is the estimated black box testing effort needed to find each critical defect. Maintenance effort is the estimated effort saved for noncritical defect These savings are subtracted from the cost of performing inspections – the time to do preparation, meeting, causal analysis, discussion, rework, and followup. In this particular project, inspections found 12 critical and 78 noncritical defects. Based on an estimated black box testing time of 20 hours per defect and 6 hours of maintenance for each noncritical defect, the total time saved amounted to $20 \times 12 + 6 \times 78 - 90 = 618$ hours. The estimated black box testing time and noncritical defect maintenance time seem to be loose upper bounds, based also on assumptions

A1, A2 and A3. Also, unit and module testing found and fixed another 51 defects at a cost of 310 hours, or ≈ 6 hours per defect. This shows that it would take far less than 20 hours to find and fix the critical defects from inspections if they happen instead to be discovered before system testing.

Discovering defects in unit and module testing saved an estimated 710 hours in subsequent maintenance. While testing seemed to give a lower return on investment ($\frac{710}{310} \approx 230\%$ as compared to $\frac{618}{90} \approx 685\%$ for inspections), it should be noted again that the farther along the test stage, the longer it takes to find defects. Also note that the 310 hours included machine time (which may be less expensive than people time) to execute the test cases, as explained in assumption A4.

Another interesting point is that noncritical defects comprised 85% of the defects found at inspection. It is not clear how much of the 90 hours invested in inspections were spent looking for and fixing these – they might be dealt with using automated tools, as explained in assumption A4. Also, the return on investment comparison between inspection and testing might be more accurate if only the savings and costs from critical defects found at inspection were considered.

Grady and van Slack[21] discuss nearly 20 years of inspection experience at Hewlett-Packard. In one 50,000-line project, they report that design inspections saved at total of 1759 engineering hours in defect-finding effort. (It was not explained how this value was calculated.) The cost was 169 engineering hours in training and start-up. (The cost of performing the actual inspections was not given.) The inspections also shortened the estimated development interval by 1.8 months. Overall, they estimated that inspections saved HP $21.4 million dollars in 1993.

Fowler[17] summarizes the results of several studies on the use of inspections in industry. In one study, a major software organization increased its productivity by 14% from one release to the next after introduction of improved project phasing and tracking mechanisms, including inspections. It also showed a tenfold improvement in quality. Fowler acknowledges, however, that these results cannot be attributed solely to inspections. Another study gave the results of using inspections in the AT&T 5ESS switch project. It claimed that defects detected in inspections cost ten times less to fix than defects found during other development phases (assumption A2). Another study gave the results of using inspections in a project within AT&T's network services. These results showed that inspections are twenty times more effective than testing in finding bugs and make up only 2% of the total cost of testing[1].

4 Underlying Mechanisms

Having looked at how inspection costs and benefits are measured, we now look at studies that investigate the underlying mechanisms driving those costs and benefits.

Some of the studies use student subjects to inspect nonindustrial artifacts (*in vitro* – in the laboratory) while others are conducted with professional software developers using industrial projects (*in vivo* – in the industry). Typically, it is more economical to use students subjects, but results may be more generalizable with industrial subjects. Nevertheless, using student subjects is an important first step towards eventually replicating the experiment with professional subjects because the design and instrumentation can then be refined and improved as experience is gained.

4.1 Investigating Underlying Mechanisms – Local Analysis

Earlier we described the attributes of different inspection methods. Supposedly, different values for these attributes produce different cost-benefit tradeoffs ("How many reviewers should we use?", "Do we need a collection meeting?", etc.). In this section we describe several empirical studies that investigate some of these tradeoffs.

4.1.1 Does Every Inspection Need a Meeting?

Votta surveyed software developers in AT&T's 5ESS project to find out what factors they believed had the largest influence on inspection effectiveness[48] . The most frequent reason cited was *synergy* (mentioned by 79% of those polled).

Informally, *synergy* allows a team working together to outperform any individual or subgroup working alone. The *Subarctic Survival Situation* exercise [28] dramatically shows this effect. (groups outperform individuals unless the individual is an arctic survival expert.)

If synergy is fundamental to the inspection process, we would expect to see many inspection defects found only by holding a meeting. That is that few defects are found in preparation (before the meeting), but many

[1] The reader should realize that this huge cost-benefit advantage of inspection over testing is in part due to an exceedingly costly system test lab.

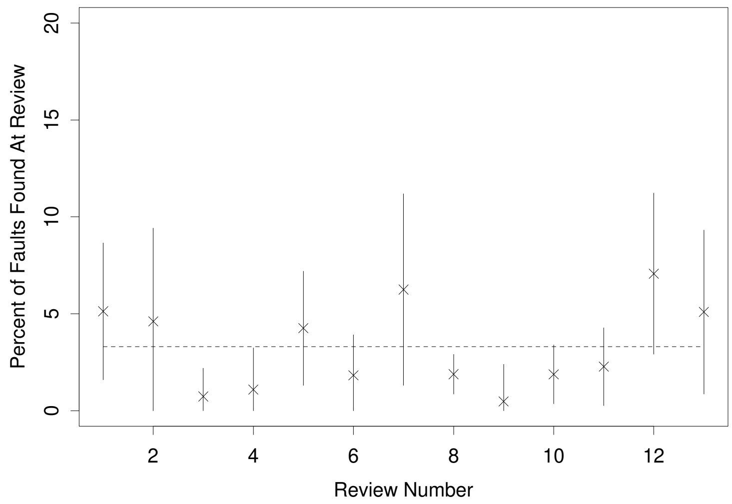

Figure 3: **Measured Synergy for Low Level Design Inspections.** Each point represents the synergy rate for a particular collection meeting, i.e., the number of defects which went undetected by reviewers in their preparation (before the meeting was held to collect the inspection results), divided by the total number of defects recorded at the meeting. This rate is marked with an $\times$. The vertical line segment through each $\times$ marks one standard deviation in the estimate of the rate (assuming each defect was a Bernoulli trial). Thus the more defects used for the rate estimate, the shorter the line segment, and hence, the more precise the estimate of the rate. This provides information on the significance of any one rate measurement. The average synergy rate is about 4% (the dashed line) for these 13 inspections.

are found (during the meeting). Votta made this measurement as part of a study of capture-recapture sampling techniques for estimating the number of defects remaining in a design artifact after inspection[14] . Figure 3 displays data showing that *synergy* is not responsible for inspection effectiveness (it only accounted for 5% of the defects found by inspections).

4.1.2 The Effect of Different Inspection Approaches

Inspection approaches are usually evaluated according to the number of defects they find. As a result, some information is available about the effectiveness of different approaches, but very little about their costs. Porter et al. argued that cost is as important as effectiveness. In particular, they believed that longer inspection intervals result in longer development intervals. They hypothesized that different approaches make significantly different tradeoffs between inspection interval and detection effectiveness. Specifically, that (1) inspections with large teams have longer inspection intervals, but find no more defects than smaller teams; (2) collection meetings do not significantly increase detection effectiveness; and (3) multiple-session inspections are more effective than single-session inspections, but at the cost of significantly increasing the inspection interval.

To evaluate these hypotheses they conducted a controlled experiment to compare the tradeoffs between the minimum interval and effort and the maximum effectiveness of several inspection approaches[40] .

They ran this experiment at AT&T on a project that is developing a compiler and environment to support developers of the AT&T 5ESS telephone switching system. The finished system contained 45 lines of C++ code, of which about 8K is reused.

The subjects were all of the team's six members plus five other developers. All were experienced, and all had received training on inspections within five years of the experiment. The project conducted more than 100 code inspections.

The experiment manipulated three independent variables:

1. the team size (one, two, or four members, in addition to the author);
2. the number of inspection sessions (one session or two sessions);
3. the coordination between sessions. (In two-session inspections the author either repaired or did not repair known defects between sessions).

For each inspection they measured four dependent variables:

1. inspection intervals,
2. estimated defect detection ratio,
3. the percentage of defects first identified at the collection meeting (meeting gain rate),
4. the percentage of potential defects reported by an individual, but not recorded at the collection meeting (meeting suppression rate).

They also captured repair statistics for every defect.

This experiment used a $2^2 \times 3$ partial factorial design to compare the interval and effectiveness of inspections with different team sizes, different numbers of inspection sessions, and different coordination strategies. They chose a partial factorial design because some treatment combinations were considered too expensive (e.g., two-session-four-person inspections with and without repair).

The results showed the following:

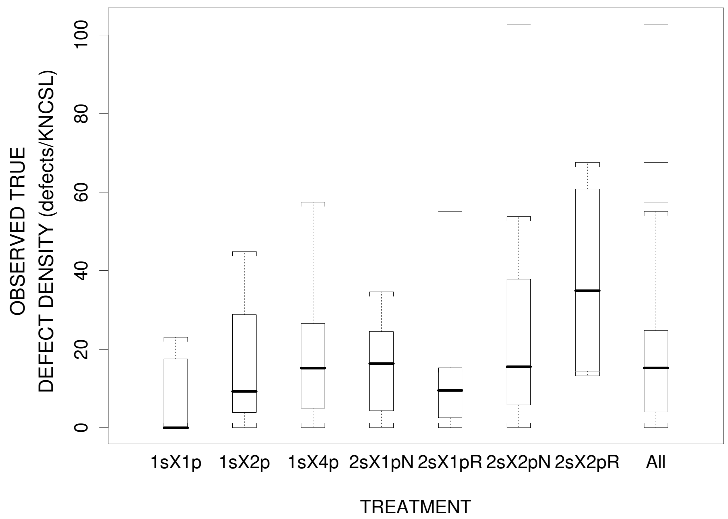

Figure 4: **Observed Defect Density by Treatment.** This plot shows the observed defect density for each inspection treatment. Across all inspections, the median defect detection rate was 24 defects per KNCSL.

1. There was no difference in either effectiveness or inspection interval between small teams and large teams.

2. 2-session, 2-reviewer inspections were more effective than 1-session, 4-reviewer inspections, but 2-session, 1-reviewer inspections were not more effective than 1-session, 2-reviewer inspections. Also, 2-session inspections held in parallel have no difference in inspection interval when compared to 1-session inspections. (See Figure 4.)

3. There was no difference in effectiveness between 2-session inspections held in parallel and those held in sequence. But those held in sequence had significantly longer intervals. (See Figure 5.)

4. Meeting gain rates (33%) were higher than in previous, recent studies[22, 47] .

4.1.3 Comparing Meetings and Their Alternatives

Votta [47] evaluated the importance of meetings in a case study of 13 design inspections at AT&T. To quantify the usefulness of inspection meetings, he determined the proportion of defects found during the inspection that were originally discovered at the meeting (the meeting gain rate). He reported that the average meeting gain rate for these inspections was $\approx 5\%$. This would mean that if 20 defects were discovered during the inspection, 19 were already known before the meeting ever started!

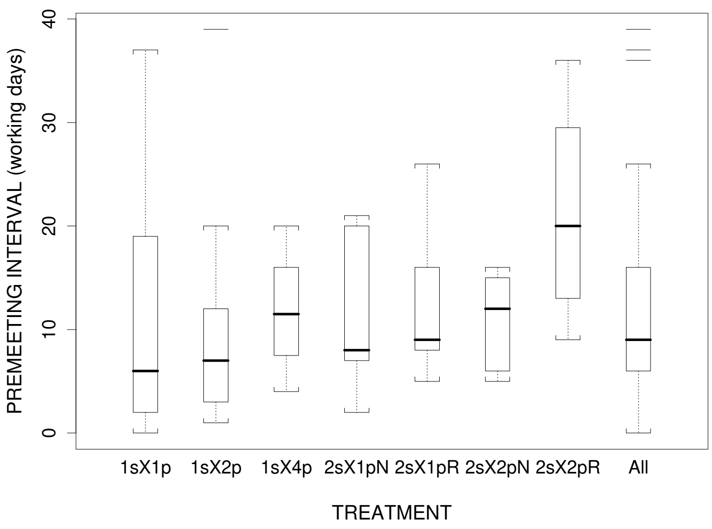

Figure 5: **Interval by Treatment.** This plot shows the observed pre-meeting interval (availability of inspection module up to the inspection meeting) for each inspection treatment. Across all treatments, the median interval is 8.5 working days.

This result was striking, but later data seems to contradict it. Porter et al.[40] conducted another study, also at AT&T, involving > 100 code inspections. Although their primary goal was not to study inspection meetings, they collected data on meeting gains in much the same way that Votta's earlier study had.[2] This time the average meeting gain rate was 33%, with considerable variance in the observations (i.e., many meetings produced no gains at all, while some had rates as high as 80%.)

This situation illustrates that every empirical study is at best an approximation, needs to be checked against previous observations, and differences resolved through continued experimentation.

Porter et al.[32] attempted to to resolve the conflicting results of two earlier industrial case studies. While doing this they uncovered anecdotal evidence that pointed to two possible explanations: (1) differences in the type of artifact being inspected (design documents vs. code units) led to the use of different "implicit" inspection processes, and (2) defects found at the meeting might be explained by factors other than meeting synergy or teamwork. Initially they are concentrating on the second explanation.

They hypothesized that inspection meetings are not nearly as cost-beneficial as many people believe; and that inspection methods that eliminate meetings are at least as cost-effective than methods that rely heavily on

[2] We strongly believe that empirical research must be replicated. This experience illustrates an economical way to do this. We instrumented the study so that it provided not only the data we were immediately interested in, but also the data needed to replicate Votta's earlier study.

them, and probably more so. They expected to see this result because they expected that benefit of additional individual analysis to be equal to or greater than the benefit of holding inspection meetings.

To evaluate these hypotheses they designed and conducted a controlled experiment. The goals of this experiment were twofold: to characterize the behavior of existing approaches, and to assess the potential benefits of meetingless inspections. They ran the experiment in the spring of 1995 with 21 subjects – students taking a graduate course in software engineering – who acted as reviewers.

Three inspection methods were used in this experiment.

- **Preparation-Inspection (PI).** Each reviewer individually analyzes the artifact in order to become familiar with it. The goal is not to discover defects but only to prepare for the inspection meeting. After all reviewers have completed this Preparation the team holds an Inspection meeting to find as many defects as possible.
- **Detection-Collection (DC).** Each reviewer individually analyzes the artifact with the goal of Detecting as many defects as possible. As with the PI approach, the team then meets (the Collection phase) to inspect the document. The results of the collection phase will, of course, contain many defects already found during the detection phase.
- **Detection-Detection (DD).** Each reviewer individually analyzes the artifact with the goal of Detecting as many defects as possible. After all reviewers complete this first Detection phase, each is asked to conduct defects Detection a second time, again individually, and again with the goal of detecting as many defects as possible. This approach does not involve a meeting, and instead the time is used by the reviewers to continue working individually.

The experiment manipulated four independent variables:

1. the inspection method used by each reviewer (PI, DC, or DD);
2. the inspection round (each reviewer participated in two inspections during the experiment);
3. the specification to be inspected (two were used during the experiment);
4. the order in which the specifications were inspected. (Either specification could be inspected first.)

For each inspection they measured three dependent variables:

1. the Individual Defect Detection Rate,
2. the Team Defect Detection Rate, [3]
3. the Gain Rate, i.e., the percentage of defects initially identified during the second phase of the inspection.

The results of this study showed

1. that the inspection method used can't be ignored as a significant source of variation in the meeting gain rates,
2. that meetingless inspections detected more new defects in the second phase of the inspection than did inspections using the other methods,
3. that meetingless inspections found more total defects than did inspections with meetings.

These results suggest that defects found at inspection meetings might be explained by factors other than meeting synergy or teamwork. Because of the small number of data points, further replications of this experiment are needed.

4.1.4 The Effect of Different Detection Methods

Two types of defect detection methods are most frequently used, Ad Hoc and Checklist. Ad Hoc reviewers use nonsystematic techniques and are assigned the same general responsibilities. Checklist reviewers are given a list of items to search for. Checklists embody important lessons learned from previous inspections within a specific environment or domain.

Porter et al.[39] , hypothesized that an alternative approach which assigned individual reviewers separate and distinct detection responsibilities and provided specialized techniques for meeting them would be more effective. This hypothesis is depicted in Figure 7.

To explore this alternative they prototyped a set of defect-specific techniques called Scenarios – collections of procedures for detecting particular classes of defects. Each reviewer executes a single Scenario and all reviewers are coordinated to achieve broad coverage of the document.

The experiment manipulated five independent variables:

---

[3] The Team and the Individual Defect Detection Rates are the number of defects detected by a team or individual divided by the total number of defects known to be in the specification. The closer these values are to 1, the more effective the detection method. No defects were intentionally seeded into the specifications. All defects were naturally occurring.

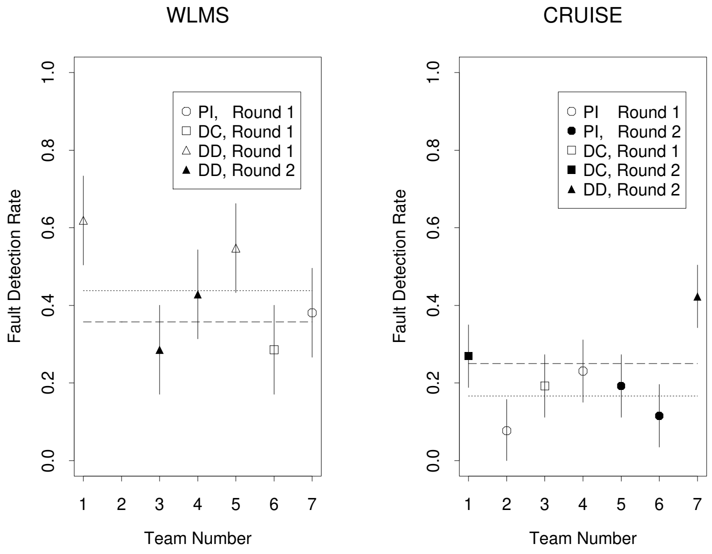

Figure 6: **Defect detection by inspection method.** The observed defect detection rates are displayed above. The unfilled symbols indicate observations from Round 1; the filled symbols those from round 2. The vertical line through each point indicates one standard deviation in the rate's estimate (modeling defect detection as Bernoulli trials). The dashed (dotted) lines display the average detection rates for Round 1 (Round 2).

1. the detection method used by a reviewer (Ad Hoc, Checklist, or Scenario);
2. the experimental replication (they conducted two replications);
3. the inspection round (each reviewer participated in two inspections during the experiment);
4. the specification to be inspected (two were used during the experiment);
5. the order in which the specifications are inspected.

For each inspection they measured four dependent variables:

1. the individual defect detection rate;
2. the team defect detection rate;
3. the percentage of defects first discovered at the collection meeting (meeting gain rate);
4. the percentage of defects first discovered by an individual but never reported at the collection meeting (meeting loss rate).

They evaluated this hypothesis in a controlled experiment, using a 3×2^4 partial factorial, randomized experimental design [39] . Forty-eight graduate students in computer science participated in this experiment.

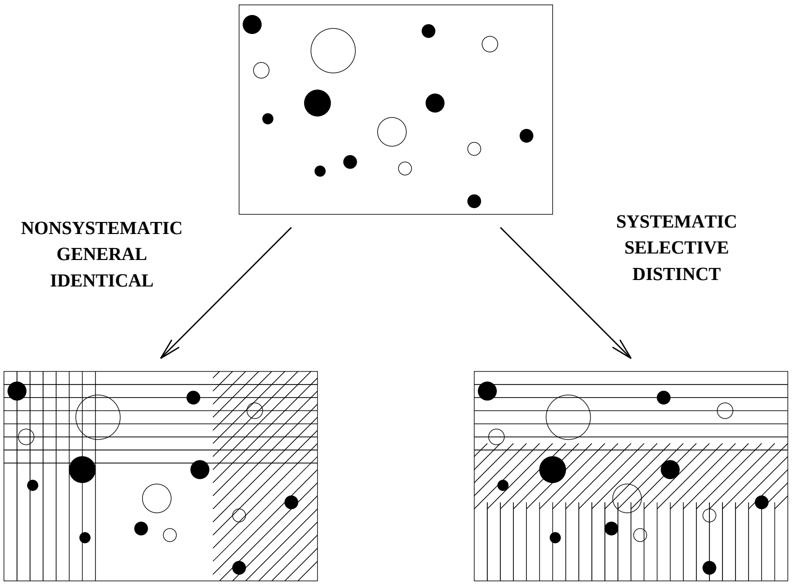

Figure 7: **Systematic Inspection Research Hypothesis.** This figure represents a software requirements specification before inspection (top) and after an inspection using *nonsystematic* techniques with *general* and *identical* responsibility assignments (bottom left), and an inspection using *systematic* techniques with *specific* and *distinct* responsibility assignments (bottom right). The points and holes represent various defects and the line-filled regions indicate the coverage achieved by different inspectors. Our hypothesis is that inspections using systematic techniques with specific and coordinated responsibilities achieve broader coverage and minimize reviewer overlap, resulting in higher defect detection rates and greater cost-benefits than do nonsystematic methods.

They were assembled into 16 three-member teams. Each team inspected two software requirements specifications (SRS) using some combination of ad hoc, checklist and scenario methods.

The experimental results showed

1. that the scenario method had a higher defect detection rate than either the ad hoc or the checklist methods (see Figure 8),
2. the scenario reviewers were more effective at detecting the defects their scenarios were designed to uncover, and were no less effective at detecting other defects,
3. that checklist reviewers were no more effective than ad hoc reviewers,
4. that regardless of the method used, collection meetings produced no net improvement in the defect detection rate – meeting gains were offset by meeting losses.

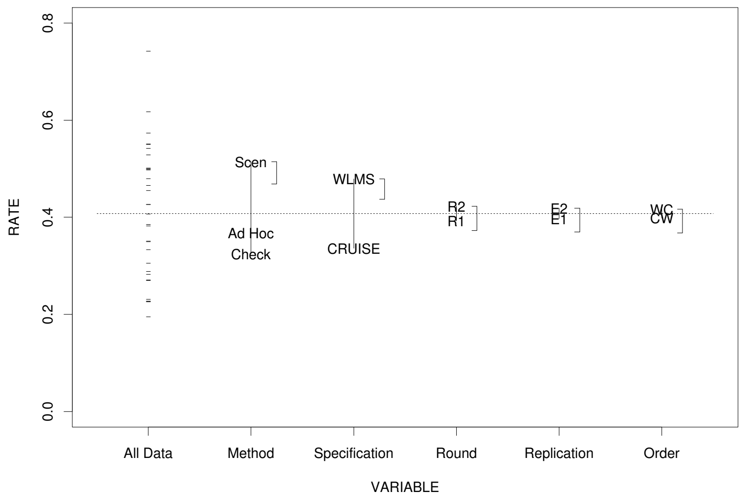

Figure 8: **Defect Detection Rates by Independent Variable.** The dashes in the far left column show each team's defect detection rate for the WLMS and CRUISE. The horizontal line is the average defect detection rate. The plot demonstrates the ability of each variable to explain variation in the defect detection rates. For the Specification variable, the vertical location of WLMS (CRUISE) is determined by averaging the defect detection rates for all teams inspecting WLMS (CRUISE). The vertical bracket,], to the right of each variable shows one standard error of the difference between two settings of the variable. The plot indicates that both the Method and Specification are significant; but Round, Replication, and Order are not.

4.1.5 Indicators of Quality Inspections

The number of defects found in an inspection is not an adequate indicator because it is influenced by the quality of the artifact being inspected. Buck [7] conducted a study at IBM by Buck[7] to identify a variable, other than the number of defects found, that would differentiate high quality inspections from low quality ones.

He collected data from 106 code inspections of a single piece of COBOL source code. Next he examined several potential indicators.

1. inspection rate,
2. team size (3, 4, or 5, including the author),
3. major defects found per hour of inspection,
4. preparation rate

The collected data showed

1. that code inspections conducted at a rate of less than 125 NCSL per hour found significantly more defects,

2. that there was no difference in defect detection capability between 3-, 4-, and 5-member teams,

3. that effectiveness was also independent of major defects found per hour,

4. that additional preparation resulted in more defects being found. [4]

Thus, the study suggests that quality inspections are a result of following a low inspection rate.

4.1.6 Using Multiple Inspection Teams

The N-fold inspection method[44] is based on the idea that no single inspection team can find all the defects in a software requirements document, that N separate inspection teams do not significantly duplicate each others' efforts, and therefore that N inspections will be significantly more effective than one. Replicating the inspection process 5 or 10 times will certainly be expensive, but it might be acceptable for critical systems if the detection rate increased significantly.

To evaluate the hypothesis, they designed and ran an experiment with 27 students who were taking a graduate course in software engineering as subjects. The subjects were divided into 9 inspection teams of 3 persons each. An attempt was made to form evenly matched teams based on background experiences. These teams inspected a single requirements document that was seeded with 99 defects. After the inspections, each recorded defect was to be checked to see if it was one of the 99 seeded defects. If so it was entered into the defect database. The authors then calculated the number of defects found by exactly x teams, where $x = 0...N$.

The results show that the 9 teams combined found a little more than twice as many of the seeded defects as the average found by any single team (78% compared to 35%). Also, no single defect was found by every team. The authors suggest that this supports their claims that parallel teams do not duplicate each other's work. The inspection took 1.5 weeks, from distribution of the document to completion of the meetings, and used 324 person-hours.

4.1.7 Computer-Aided Inspections

Computer support adds a new dimension to the inspection process. By automating some parts of the process and providing computer support for others, the inspection process can possibly be made more effective and efficient[29]

[4] The study concludes with the unsatisfying result that you can always spend more preparation time and find more defects. There is no discussion of what a practical limit may be.

. For example, during preparation computer support allows artifacts to be inspected, inspector comments to be recorded and project management reports to be handled online. This eliminates much of the bulky printed materials and the forms normally generated by inspections.

Software tools can also perform automated detection of simple defects, freeing inspectors to concentrate on major defects. Using such tools required that artifacts are specified with some formal notation, or programming language. For example, a C language-specific inspection tool called ICICLE[29] uses `lint`[25] , to identify C program constructs that may indicate the presence of defects. It also checks the C program against its own rule-based system.

Computer support for meetings can reduce the cost of meetings. With videoconferencing, inspectors in different locations can easily meet. Computer support can also mitigate the group-interaction-related problems by allowing meetings to be held in "nominal" fashion, where inspectors do not actually have to meet, but can just place their comments in a central repository which others can read at their convenience and extend [11] .

The main disadvantage is inadequate technological support. Most computer-aided inspection systems are still in the research labs and not yet ready for industrial use. In addition, some special equipment may be needed for videoconferencing.

Collaborative Software Inspection. Mashayekhi, et al.[31] , discuss a case study on the use of Collaborative Software Inspection (CSI), a software system to support inspections. Computer support is provided for the preparation and meeting steps. CSI assists with online examination of the artifact and recording of inspector comments. In addition, CSI collates the comments into a single list. The main feature of CSI is that it allows the meeting to be geographically distributed, with the artifact being displayed on each inspector's screen and a voice connection that allows people to talk to each other.

This case study was conducted with 9 student volunteers from a software engineering class and compared the effectiveness of using CSI with face-to-face inspection meetings. The participants were divided into 3 teams, each of which inspected the same 4 pieces of code for a total of 12 inspections. Of these, 5 inspection meetings were randomly selected to use CSI while the rest met face-to-face. The results showed that in only one of the 4 pieces did CSI find more defects. However, because the teams retained their relative rankings across all modules inspected (i.e., Team 1 was always first in each module, Team 2 was always second, Team 3 was always last), the authors concluded that the use of CSI did not have any positive or negative effect on any of them.

FTArm. Johnson[24] presents the *Formal Technical Asynchronous review method* (FTArm) implemented on Collaborated Software Review System (CSRS). CSRS is a software inspection environment whose aim is not to specify inspection policy, but only to automate the support functions required for various inspection methods. FTArm is geared towards asynchronous software inspections. All comments by reviewers are kept online. The inspection consists primarily of a private review step and a public review step. During the private review step, reviewers cannot see each other's comments. In the public review step, all comments become public and reviewers can build on each other's suggestions. They then vote on whether they agree or disagree with the comments made about each section of the artifact being inspected. If unresolved issues remain, they are handled in a face-to-face group review meeting. Evaluation of the effectiveness of FTArm is under way.

4.2 Investigating Underlying Mechanisms – Global Analysis

Holding, or not holding, inspections has an effect on the cost of the overall software development process. Several factors influence the relationship between inspections and the rest of the software development process.

4.2.1 Inspection Versus Testing

Testing is traditionally the most widespread method for validating software. The tester prepares several test cases and runs each on through the program, comparing actual output with expected output. Testing puts theory into practice: a program thought to work by its creator is applied to a real environment with a specific set of inputs, and its behavior is observed. Defects are normally found one at a time. When the program behaves incorrectly on certain inputs, the author carries out a debugging procedure to isolate the cause of the defect.

Inspections have an advantage over testing in that they can be performed earlier in the software development process, even before a single line of code is written. Defects can be caught early and prevented from propagating down to the source code. In terms of the amount of effort to fix a defect, inspections are more efficient since they find and fix several defects in one pass as opposed to testing, which tends to find and fix one defect at a time [1] . Also, there is no need for the additional step of isolating the source of the defect because inspections look directly at the design document and source code. It may be argued that this additional step in testing is offset by inspection preparation and meeting effort, but testing also requires effort in preparation of test cases and setting up test environments. However, testing is better for finding defects related to execution, timing, traffic, transaction rates, and system interactions[43] . So inspections cannot completely replace testing (although some case studies argue that unit testing may be removed)[1, 49] .

The following two studies compare inspection methods with testing methods. The first is a controlled experiment while the second is a retrospective case study.

Comparing the Effectiveness of Software Testing Strategies. Basili and Selby[3] investigated the effectiveness of 3 program validation techniques: functional (black box) testing, structural (white box) testing, and code reading by stepwise abstraction (described in Section 2.2).

The goals of the study were to determine which of the 3 techniques detects the most faults in programs, and which detects faults at the highest rate, and to find out if each technique finds a certain class of faults.

A controlled experiment was conducted in which both students and professionals validated 4 different pieces of software, labeled P_1, P_2, P_3, and P_4. Three independent variables were manipulated: (1) testing technique (functional testing, structural testing, code reading), (2) software type (P_1, P_2, P_3, P_4), and (3) level of expertise (advanced, intermediate, junior).

The dependent variables measured included (1) number of faults detected, (2) percentage of faults detected (the total number of faults was predetermined), (3) total fault detection time, and (4) fault detection rate.

The experiment was carried out in three phases, the first two with student subjects and the third with professional developers. Each phase validated three of the four programs. The experiment employed a partial factorial design, assigning each subject to validate all three programs using a different technique on each. The sequence of programs and techniques was randomized.

The most interesting result is that code reading was more effective than functional and structural testing at finding faults in the first and third phases and was equally good in the second phase. With respect to fault detection rate, code reading achieved the highest rate in the third phase and the same rate as the testing techniques in the other two phases. Finally, code reading found more interface faults.

Evaluating Software Engineering Technologies. Card, et al.[9] describe a study measuring the importance of certain technologies (practices, tools, and techniques) on software productivity and reliability.

Eight technologies were assessed:

1. quality assurance (reviews, walkthroughs, configuration management, etc.),

2. software tool use (use of design language, static analysis tools, precompilers, etc.),

3. documentation,

4. structured programming,

5. code reading,

6. top-down development,

7. chief programmer team (a team organized around a technical leader who delegates programming assignments and reviews finished work),

8. design schedule (putting more weight on the design phase).

A non-random sample of 22 software projects from NASA Goddard Space Flight Center was chosen. The selection criteria were chosen to minimize the effects of the programming language and the development environment. Variation in the sizes of projects was also minimized. The effects of nontechnological factors were removed – productivity was corrected for computer use (amount of time spent using computers) and programmer effectiveness (development teams' years of experience), while reliability was corrected for programmer experience and data complexity (number of data items per subsystem).

The results showed that no technological factor explained any of the remaining variation in productivity. However, variation in software reliability was reduced using code reading and quality assurance. The authors conclude that since reliability and productivity are positively correlated, improving reliability improves productivity.

5 Conclusions and Future Work

We have presented a survey of existing research, paying attention to how each study measured the costs and benefits of holding inspections and how they explained the factors that influence these measurements, at either a local or a global level.

At the global level, we see that software inspection is still an effective method for detecting and removing defects. However, whether it is cost-effective remains to be seen. The literature contains little solid empirical evidence. Many studies have focused on the benefits of inspections and made cost assumptions that seldom hold in actual practice. Future studies should take a more realistic view. The results (or lack of them) to be found in existing research indicate that, while it is relatively easy to measure the global benefits of holding inspections, it is very difficult to measure the global cost incurred by inspections, especially the cost of greater development intervals, which we believe is significantly higher than has been realized. This could have serious economic consequences, especially in a highly competitive environment where being the first to introduce a new (even poorly implemented) feature to the market may mean the difference between success and failure of a product[45] . Obviously, it would be expensive and impractical to replicate entire development projects to see how removing inspections from the process affects the development interval. Future research will need to find more economical ways to estimate this cost.

At the local level, we have almost the opposite problem when measuring costs and benefits. While it is often easy to tell if one inspection method costs more than another (for example, inspections using several sessions are clearly costlier than inspections using one session), it is very difficult to tell if one method is actually better than another at detecting defects (paired studies are expensive, we have to get the same artifact and the same set of reviewers to try out each inspection method). One problem comes in comparing the resulting defect detection ratios – the number of defects found in the inspection divided by the total number of defects in the artifact. A fundamental technical problem is that we can never know exactly how many defects are originally in an artifact, unless we follow the product through its life cycle. Even then, we don't know for sure; there may still be defects left undiscovered. Also, it is very hard to trace a certain failure in the field to a defect that was missed in the inspection of a certain artifact. One solution is to estimate the pre-inspection defect content using statistical methods. One such method is capture-recapture[14, 46] , which is based on the intuitive premise that if reviewers are finding many of the same defects in an inspection, then it is likely that there are few defects to be found in the first place. Conversely, if reviewers are finding few defects in common with one another, then it is likely that there are many more defects to be found. However, experience has shown that capture-recapture does not work well when the overall number of defects found by each reviewer is small. Future research should look further into this and other estimation methods.

References

[1] A. Frank Ackerman, Lynne S. Buchwald, and Frank H. Lewski. Software inspections: An effective verification process. *IEEE Software*, pages 31–36, May 1989.

[2] Victor R. Basili and Harlan D. Mills. Understanding and documenting programs. *IEEE Trans. on Software Engineering*, SE-8(3):270–283, May 1982.

[3] Victor R. Basili and Richard W. Selby. Comparing the effectiveness of software testing strategies. *IEEE Trans. on Software Engineering*, SE-13(12):1278–1296, Dec. 1987.

[4] David B. Bisant and James R. Lyle. A two-person inspection method to improve programming productivity. *IEEE Trans. on Software Engineering*, 15(10):1294–1304, Oct. 1989.

[5] Barry Boehm. Verifying and validating software requirements and design specifications. *IEEE Software*, 1(1):75–88, January 1984.

[6] Robert N. Britcher. Using inspections to investigate program correctness. *IEEE Computer*, pages 38–44, Nov. 1988.

[7] F. O. Buck. Indicators of quality inspections. Technical Report 21.802, IBM, Kingston, NY, Sep. 1981.

[8] Marilyn Bush. Improving software quality: The use of formal inspections at the Jet Propulsion Laboratory. In *Proceedings of the 12th International Conference on Software Engineering*, pages 196–199, 1990.

[9] David N. Card, Frank E. McGarry, and Gerald T. Page. Evaluating software engineering technologies. *IEEE Trans. on Software Engineering*, SE-13(7):845–851, July 1987.

[10] Jarir K. Chaar, Michael J. Halliday, Inderpal S. Bhandari, and Ram Chillarege. In-process evaluation for software inspection and test. *IEEE Trans. on Software Engineering*, 19(11):1055–1070, Nov. 1993.

[11] Alan R. Dennis and Joseph S. Valacich. Computer brainstorms: More heads are better than one. *Journal of Applied Psychology*, 78(4):531–537, April 1993.

[12] James H. Dobbins. Inspections as an up-front quality technique. In *Handbook of Software Quality Assurance*, pages 137–177. Van Nostrand Reinhold, 1987.

[13] E. P. Doolan. Experience with Fagan's inspection method. *Software Practice and Experience*, 22(2):173–182, Feb. 1992.

[14] Stephen G. Eick, Clive R. Loader, M. David Long, Scott A. Vander Wiel, and Lawrence G. Votta. Estimating software fault content before coding. In *Proceedings of the 14th International Conference on Software Engineering*, pages 59–65, May 1992.

[15] Michael E. Fagan. Design and code inspections to reduce errors in program development. *IBM Systems Journal*, 15(3):182–211, 1976.

[16] Michael E. Fagan. Advances in software inspections. *IEEE Trans. on Software Engineering*, SE-12(7):744–751, July 1986.

[17] Priscilla J. Fowler. In-process inspections of workproducts at AT&T. *AT&T Technical Journal*, 65(2):102–112, March-April 1986.

[18] Louis A. Franz and Jonathan C. Shih. Estimating the value of inspections and early testing for software projects. *Hewlett-Packard Journal*, pages 60–67, Dec. 1994.

[19] Daniel P. Freedman and Gerald M. Weinberg. *Handbook of Walkthroughs, Inspections, and Technical Reviews*. Little, Brown and Company, 3rd edition, 1982.

[20] Tom Gilb and Dorothy Graham. *Software Inspection*. Addison-Wesley Publishing Co., 1993.

[21] Robert B. Grady and Tom Van Slack. Key lessons in achieving widespread inspection use. *IEEE Software*, pages 46–57, July 1994.

[22] Watts S. Humphrey. *Managing the Software Process*, chapter 10. Addison-Wesley Publishing Company, 1989.

[23] Daniel Jackson. Aspect: An economical bug-detector. In *Proceedings of the 13th International Conference on Software Engineering*, pages 13–22, 1991.

[24] Philip M. Johnson. An instrumented approach to improving software quality through formal technical review. In *Proceedings of the 16th International Conference on Software Engineering*, pages 113–122, Sorrento, Italy, May 1994.

[25] S. C. Johnson. A C program checker. In *UNIX(TM) Time-Sharing System - UNIX Programmer's Manual*. Holt, Rinehart and Winston, New York, 7th edition, 1982.

[26] John C. Kelly, Joseph S. Sherif, and Jonathan Hops. An analysis of defect densities found during software inspections. *Journal of Systems and Software*, 17:111–117, 1992.

[27] John C. Knight and E. Ann Myers. An improved inspection technique. *Communications of the ACM*, 36(11):51–61, Nov. 1993.

[28] C. Lafferty. *The Subarctic Survival Situation*. Synergistics, Plymouth, MI, 1975.

[29] F. MacDonald, J. Miller, A. Brooks, M. Roper, and M. Wood. A review of tool support for software inspection. Technical Report RR-95-181, University of Strathclyde, Glasgow, Scotland, Jan. 1995.

[30] K.E. Martersteck and A.E. Spencer. Introduction to the 5ESS(TM) switching system. *AT&T Technical Journal*, 64(6 part 2):1305–1314, July-August 1985.

[31] Vahid Mashayekhi, Janet M. Drake, Wei-Tek Tsai, and John Riedl. Distributed, collaborative software inspection. *IEEE Software*, pages 66–75, Sep. 1993.

[32] Patricia McCarthy, Adam Porter, Harvey Siy, and Lawrence G. Votta. An experiment to assess cost-benefits of inspection meetings and their alternatives. Technical Report CS-TR-3520, University of Maryland, College Park, MD, September 1995.

[33] Steve McConnell. *Code Complete*, chapter 24. Microsoft Press, 1993.

[34] J.F. Nunamaker, Alan R. Dennis, Joseph S. Valacich, Douglas R. Vogel, and Joey F. George. Electronic meeting systems to support group work. *Communications of the ACM*, 34(7):40–61, July 1991.

[35] David L. Parnas and David M. Weiss. Active design reviews: Principles and practices. In *Proceedings of the 8th International Conference on Software Engineering*, pages 215–222, Aug. 1985.

[36] Dewayne E. Perry. The Inscape environment. In *Proceedings of the 11th International Conference on Software Engineering*, pages 2–12, May 1989.

[37] Dewayne E. Perry and W. Michael Evangelist. An empirical study of software interface faults — an update. In *Proceedings of the Twentieth Annual Hawaii International Conference on Systems Sciences*, volume II, pages 113–126, Jan. 1987.

[38] Dewayne E. Perry and Carol S. Stieg. Software faults in evolving a large, real-time system: a case study. In *4th European Software Engineering Conference – ESEC93*, pages 48–67, Sept. 1993. Invited keynote paper.

[39] Adam Porter, Lawrence G. Votta, and Victor Basili. Comparing detection methods for software requirement inspections: A replicated experiment. *IEEE Transactions on Software Engineering*, 21(6):563–575, June 1995.

[40] Adam A. Porter, Lawrence G. Votta, Harvey P. Siy, and Carol A. Toman. An experiment to assess the cost-benefits of code inspections in large scale software development. In *The Third Symposium on the Foundations of Software Engineering*, Washington, D.C., Oct. 1995. To appear.

[41] Stan Rifkin and Lionel Deimel. Applying program comprehension techniques to improve software inspections. In *Proceedings of the 19th Annual NASA Software Engineering Laboratory Workshop*, Greenbelt, MD, Nov. 1994.

[42] David S. Rosenblum. Towards a method of programming with assertions. In *Proceedings of the 14th International Conference on Software Engineering*, pages 92–104, Melbourne, Australia, May 1992.

[43] Glen W. Russell. Experience with inspection in ultralarge-scale developments. *IEEE Software*, pages 25–31, Jan. 1991.

[44] G. Michael Schneider, Johnny Martin, and Wei-Tek Tsai. An experimental study of fault detection in user requirements documents. *ACM Trans. on Software Engineering and Methodology*, 1(2):188–204, Apr. 1992.

[45] George Stalk, Jr. and Thomas M. Hout. *Competing Against Time: How Time-Based Competition is Reshaping Global Markets*. The Free Press, 1990.

[46] Scott A. Vander Wiel and Lawrence G. Votta. Assessing software design using capture-recapture methods. *IEEE Trans. Software Eng.*, SE-19:1045–1054, November 1993.

[47] Lawrence G. Votta. Does every inspection need a meeting? In *Proceedings of ACM SIGSOFT '93 Symposium on Foundations of Software Engineering*, pages 107–114. Association for Computing Machinery, December 1993.

[48] Lawrence G. Votta. Does every inspection need a meeting? *ACM SIGSoft Software Engineering Notes*, 18(5):107–114, Dec. 1993.

[49] Edward F. Weller. Lessons from three years of inspection data. *IEEE Software*, pages 38–45, Sep. 1993.

[50] Alexander L. Wolf and David S. Rosenblum. A study in software process data capture and analysis. In *Proceedings of the Second International Conference on Software Process*, pages 115–124, February 1993.