

# Brand New Information

*What modern literature has to say about code review; what studies do and don't agree on.*

An Amazon search for books on “code inspection” turns up only one item<sup>1</sup>: The 1974, out-of-print, 29-page article by Michael Fagan of IBM. In that year, IBM sold the model 3330-11 disk drive for \$111,600. A megabyte of RAM would set you back \$75,000. The PDP-11 was still the best-selling minicomputer.

Everything has changed since then: programming languages, development techniques, application complexity and organization, levels of abstraction, and even the type of person who decides to enter the field.

But there hasn't been much change to the accepted wisdom of how to conduct proper code inspections. Some of Fagan's

---

<sup>1</sup> Ignoring two “technical articles” and books on home and construction inspections.

ideas are as applicable as ever, but surely there must be something new. Inspecting assembly code in OS/360 is nothing like running down the implications of a code change in an object-oriented interpreted language running in a 3-tier environment. Calling inspection meetings with 5 participants doesn't work in the world of remote-site development and agile methodologies.

This essay is a survey of relatively recent studies on peer review and inspection techniques. We point out results common to all studies and results that vary widely between studies.

There is an emphasis here on the timeliness of the study. You won't see the seminal works of Fagan, Gilb, and Wiegers<sup>2</sup>. Some of the original ideas are still as applicable as ever, but of course some things have changed. We don't want to parrot the accepted wisdom of the great men who started the theory of code reviews, but instead to survey what appears to be the state of affairs in the modern world of software development.

---

<sup>2</sup> Nothing here should be construed as a slight against the excellent work, tradition, and success established by these men. The author of this essay highly recommends Wiegner's 2002 *Peer Reviews in Software* as the most readable, practical guide to formal reviews.

### **Votta 1993<sup>3</sup>, Conradi 2003<sup>4</sup>, Kelly 2003<sup>5</sup>: Are review meetings necessary?**

One of the most controversial questions in code review is: Does every inspection need a meeting? Michael Fagan, the father of code inspection, has insisted since 1976 that the inspection meeting is where defects are primarily detected, but research in intervening thirty years has not been so strongly conclusive.

The first, most famous attack on the value traditionally associated with meetings came from Lawrence Votta from AT&T Bell Labs in 1993. He identified the five reasons most cited by both managers and software developers in support of inspection meetings:

1. Synergy. Teams find faults that no individual reviewer would be able to find.
2. Education. Less experienced developers and reviewers learn from their more experienced peers.
3. Deadline. Meetings create a schedule that people must work towards.
4. Competition. Ego leads to personal incentive to contribute and improve.

---

<sup>3</sup> Lawrence G. Votta, Jr., Does every inspection need a meeting?, *Proceedings of the 1st ACM SIGSOFT symposium on Foundations of software engineering*, p.107-114, December 08-10, 1993, Los Angeles, California, United States

<sup>4</sup> Reidar Conradi, Parastoo Mohagheghi, Tayyaba Arif, Lars Christian Hegde, Geir Arne Bunde, and Anders Pedersen; Object-Oriented Reading Techniques for Inspection of UML Models – An Industrial Experiment. In *European Conference on Object-Oriented Programming ECOOP'03*. Springer-Verlag, Darmstadt, Germany, pages 483-501

<sup>5</sup> Kelly, D. and Shepard, T. 2003. An experiment to investigate interacting versus nominal groups in software inspection. In *Proceedings of the 2003 Conference of the Centre For Advanced Studies on Collaborative Research* (Toronto, Ontario, Canada, October 06 - 09, 2003). IBM Centre for Advanced Studies Conference. IBM Press, 122-134.

5. Process. Inspections simply require meetings. That's the official process.

However, in his 1993 seminal paper based on his own research and that of others, Votta argued that:

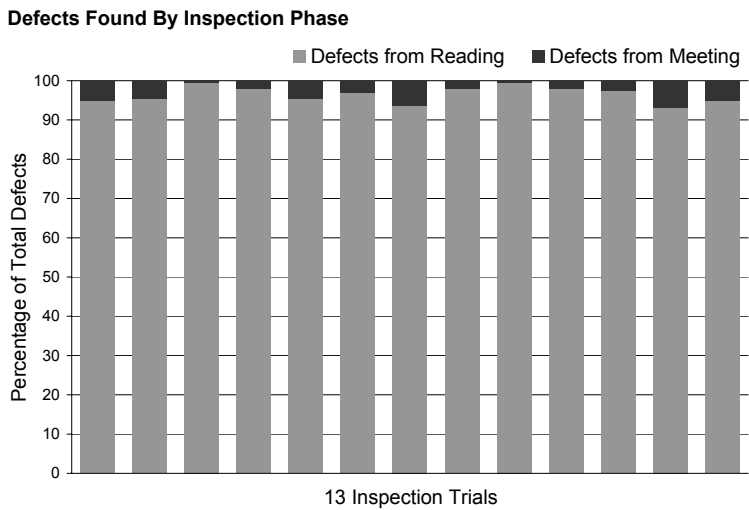
1. Synergy. Meetings tend to identify false-positives rather than find new defects. (More below.)
2. Education. Education by observation is usually unsuccessful; some researchers condemn it completely.
3. Deadlines. Process deadlines are important but could be enforced without meetings per se, or at least without heavy-weight meetings.
4. Competition. Competition is still achieved with any peer review. Some competition destroys teamwork, e.g. between designers and testers.
5. Process. Process is important but facts, not “tradition,” should be used to form and improve the process.

Furthermore, although Votta agreed with the prevailing claims that code inspections save time by detecting defects early in the development process, he pointed out that the founders of inspection did not properly consider the amount of time consumed by the inspection meeting. For example, one study of formal inspection showed that 20% of the requirements and design phase was spent just waiting for a review to start! The time spent in preparing, scheduling, and waiting for reviews is significant and grows with the number of meeting participants, yet this time is ignored in the usual cost-benefit analysis.

Recall that “meeting synergy” was cited most often by both developers and managers as well as by the literature as the primary advantage of inspection meetings. Here “synergy” refers to the team effect that a group of people performs better than any of its

members; in particular, that a group-inspection will necessarily uncover more defects than the reviewers individually.

Votta set out to test this hypothesis by measuring the percentage of defects found in inspection meetings as opposed to the private code readings that precede those meetings. His findings are summarized in Figure 3.



*Figure 3: Votta's results demonstrating that in-spection meetings contribute only an additional 4% to the number of defects already found by private code-readings.*

As it turned out, meetings contributed only 4% of the defects found in the inspections as a whole. Statistically larger than zero, but Votta asks “Is the benefit of  $\sim 4\%$  increase in faults found at the collection meeting (for whatever reason) worth the cost of  $T_{\text{collection}}$  [wasted time<sup>6</sup>] and the reviewer’s extra time? The answer is no.”

Strong words! But surely there are other benefits to inspection meetings besides just uncovering defects?

In 2003, Diane Kelly and Terry Shepard at the Royal Military College of Canada set up an experiment comparing reviewers in isolation versus group meetings. Would the results support or contradict Votta? And besides the quantity of defects, would there be a difference in other important factors such as the rate at which defects were uncovered or a reduction in false-positives that waste authors’ time?

In Kelly’s case, groups of developers read code individually to detect as many defects as possible. Then each group got together in an inspection meeting. If proponents of traditional inspections are correct, significant numbers of defects will be found during the meeting phase, especially compared with the reading phase. If Votta’s conclusions are correct, we should expect to see few defects detected in the meeting but some thrown out during the meeting (i.e. removal of false-positives or confusing points in the code).

---

<sup>6</sup> Votta identifies three components to wasted time: (1) hazard cost of being later to market, (2) carrying cost of development when developers are in meetings instead of writing code, and (3) rework cost when authors continue to develop the work product only to have the work later invalidated by faults found in inspection.

In total, 147 defects were found during the reading phases<sup>7</sup>. Of these, 39 (26%) were discarded during meetings. Although some of these were due to false-positives (i.e. the reviewer was incorrect in believing there was a defect) there were also a significant number of cases where poor documentation or style in the code lead the reviewer to believe there was a problem. Kelly suggests that these should probably be considered “defects” after all.

So the meeting did throw out false-positives – a useful thing – but what about uncovering new defects? Votta would guess that very few new defects would be found. With Kelly the meeting phases added only 20 new defects to the existing 147. Furthermore, of those 20, two-thirds were relatively trivial in nature. So not only did the meeting phases not contribute significantly to overall defect counts, the contribution was generally of a surface-level nature rather than logic-level or algorithmic-level.

Perhaps we should not be surprised by all this. Detecting problems in algorithms generally requires concentration and thought – a single-minded activity that isn’t encouraged in the social milieu of a meeting. Are you more likely to discover the bug in a binary-search algorithm by debate or by tracing through code-paths by yourself?

Besides the quantity of defects, it is also useful to consider how much time was consumed by each of these phases. After all, if the review meeting is very fast, the elimination of the false-positives would make it worthwhile even if no additional defects are found.

Kelly found that about two-thirds of total person-hours were spent in reading and one-third in meetings. This leads to a defect

---

<sup>7</sup> Developers inspecting code in isolation will find duplicate defects; we probably don’t want to count these in the analysis. The researchers found only 10 of the 147 were duplicates.

discovery rate of 1.7 defects per hour for reading and 1.2 for meeting. Reading is 50% more efficient in finding defects than are meetings.

Yet another direct test of Votta's contentions came from a different angle in 2003 from a joint effort conducted by Reidar Conradi between Ericsson Norway and two Norwegian colleges, NTNU and Agder University. The goal of the experiments was to measure the impact of certain reading techniques for UML model inspections. Votta experimented with design reviews, Kelly with source code, and now Conradi would examine architecture reviews.

The stated goal of the study was to determine the effectiveness of "tailored Object-Oriented Reading Techniques" on UML inspections. They collected separate data on defect detection during the reading phase and the meeting phase. Their purpose was not to support or invalidate Votta's results, but their data can be applied to that purpose. Indeed, in their own paper they causally mention that their data just happens to be perfectly in line with Votta's.

In particular, in 38 experimentally-controlled inspections they found that 25% of the time was spent reading, 75% of the time in meetings, and yet 80% of the defects were found during reading! They were 12 times more efficient at finding defects by reading than by meeting. Furthermore, in their case they had 5-7 people in each meeting – several more than Kelly or Votta or even Fagan recommends – so the number of defects found per man-hour was vanishingly small.



Other research confirms these results<sup>8</sup>. Because the reduction of false-positives appears to be the primary effect of the inspection meeting, many researchers conclude that a short meeting with two participants – maybe even by e-mail instead of face-to-face – should be sufficient to get the benefits of the meeting without the drawbacks. The value of detecting false-positives in the first place is questioned because often this is a result of poorly-written code and so often shouldn't be discarded anyway. Given all this, some even suggest that the extra engineering time taken up by implementing fixes for so-called false-positive defects is still less than the time it takes to identify the defects as false, and therefore we should dispense with meetings all together!

### **Blakely 1991: Hewlett Packard<sup>9</sup>**

Hewlett Packard has a long history of code inspections. In 1988 a company-wide edict required a 10x code quality improvement – a tall order for any development organization, but at least it was a measurable goal. They turned to design and code inspections as part of their effort to achieve this, and management sanctioned a pilot program implemented by a development group in the Application Support Division.

---

<sup>8</sup> For example, see L. Land, C. Sauer and R. Jeffery's convincing 1997 experiment testing the role of meetings with regard to finding additional defects and removing false-positives. Validating the defect detection performance advantage of group designs for software reviews: report of a laboratory experiment using program code. In *Proceedings of the 6th European Conference Held Jointly with the 5th ACM SIGSOFT international Symposium on Foundations of Software Engineering* (Zurich, Switzerland, September 22 - 25, 1997). M. Jazayeri and H. Schauer, Eds. Foundations of Software Engineering. Springer-Verlag New York, New York, NY, 294-309.

<sup>9</sup> Frank W. Blakely, Mark E. Boles, Hewlett-Packard Journal, Volume 42, Number 4, Oct 1991, pages 58-63. Quoting and copying herein is by permission of the Hewlett-Packard Company.

Their conclusion: “Based on the data collected about the use of code inspections, and the data concerning the cost of finding and repairing defects after the product has been released to the customer, it is clear that the implementation of code inspections as a regular part of the development cycle is beneficial compared to the costs associated with fixing defects found by customers.”

This pilot study involved a single project with 30 development hours and 20 review hours – 13 hours in pre-meeting inspection and 7 hours in meetings. They restricted their inspection sizes to 200 lines of code per hour as per the guidelines set out by Fagan and Gilb. 21 defects were uncovered giving the project a defect rate of 0.7 per hour and 100 per thousand lines of code.

This study went further than most to quantify how many defects found in code review would not have been otherwise found in testing/QA. After all, if you’re trying to reduce overall defect numbers, it’s not worth spending all this time in review if testing will uncover the problems anyway.

Because they knew this issue was important from the start, they collected enough information on each defect to determine whether each could have been detected had the testing/QA process been better. In particular, for each defect they answered this question: “Is there any test that QA could have reasonably performed that would have uncovered this defect?” Perhaps it would be more efficient to beef up testing rather than reviewing code.

The result was conclusive: Only 4 of the 21 defects could conceivably been caught during a test/QA phase. They further postulate that it would have taken more total engineering hours to find and fix those 4 in QA rather than in inspection.

## Dunsmore 2000: Object-Oriented Inspections<sup>10</sup>

What inspection techniques should be used when reviewing object-oriented code? Object-oriented (OO) code has different structural and execution patterns than procedural code; does this imply code review techniques should also be changed, and how so?

Alastair Dunsmore, Marc Roper, and Murray Wood sought to answer this question in a series of experiments.

The first experiment with 47 participants uncovered the first problem with traditional code inspections: Understanding a snippet of OO code often requires the reader to visit other classes in the package or system. Indeed, a large number of defects were rarely found by the reviewers because the defect detection required knowledge outside the immediate code under inspection. With traditional sit-down with code-in-hand inspections the readers didn't have the tools to investigate other classes, and therefore had a hard time finding the defects.

They explored a way to address this problem in the second experiment. The reviewers were given a reading plan that directed their attention to the code in a certain order and supplied additional information according to a systematic set of rules. The rules were a rough attempt at pulling in related code given the code under review. The theory was that, if this technique was better, one could conceivably make a tool to collect the information automatically. This "systematic review" was performed by 64 reviewers and the results compared with those from the first study.

The systematic review was better. Some defects that weren't found by anyone in the first test were found in the second. Furthermore, both reviewers and the creators of the reading plan

---

<sup>10</sup> Dunsmore, A., Roper, M., Wood, M. Object-Oriented Inspection in the Face of Delocalisation, appeared in Proceedings of the 22nd International Conference on Software Engineering (ICSE) 2000, pp. 467-476, June 2000.

reported that they enjoyed creating and having the plan because it led to a deeper understanding of the code at hand. Indeed, the plans could be used as documentation for the code even outside the context of a code review. Reviewers also reported feeling more comfortable having a strict reading plan rather than having to wade through a complex change and “wander off” into the rest of the system.

In the third experiment, the researchers compared three different approaches to the review problem in a further attempt to identify what techniques work best in the OO context:

1. The “checklist review” gives the reviewers a specific list of things to check for at the class, method, and class-hierarchy levels. The checklist was built using the experience of the first two experiments as a guide for what types of problems reviewers should be looking for.
2. The “systematic review” technique of the second experiment, with more refinement.
3. The “use-case review” gives the reviewers a set of ways in which we expect the code to be used by other code in the system. This is a kind of checklist that the code behaves in documented ways, “plays nice” when under stress, and works in a few specific ways that we know will be exercised in the current application.

The result of this experiment is shown in Figure 4. Clearly the checklist method was the most successful, uncovering more defects in less time than the other two techniques, 30% better than the worst in the rate at which defects were uncovered. However it should be mentioned that the defects found in each of the three techniques didn’t overlap completely. The authors therefore suggested using more than one approach to cover the most ground,

although the amount of pre-review time it would take to prepare for all these techniques is probably prohibitive.

	Checklist	Systematic	Use-Case
Defects (of 14)	7.3	6.2	5.7
False-Positives	3.4	3.2	2.9
Inspection Time	72.1	77.0	81.9
Defect Rate	6.07	4.83	4.18

*Figure 4: Comparing results from three types of reviews. Inspection time is in minutes. Defect rate is in defects per hour.*

In this third experiment they also kept track of the exact time that each of the defects were found during the inspection. Are most defects found quickly? Is there a drop-off point after which defects are no longer found? Is there a difference between the three types of review?

The results are shown in Figure 5.

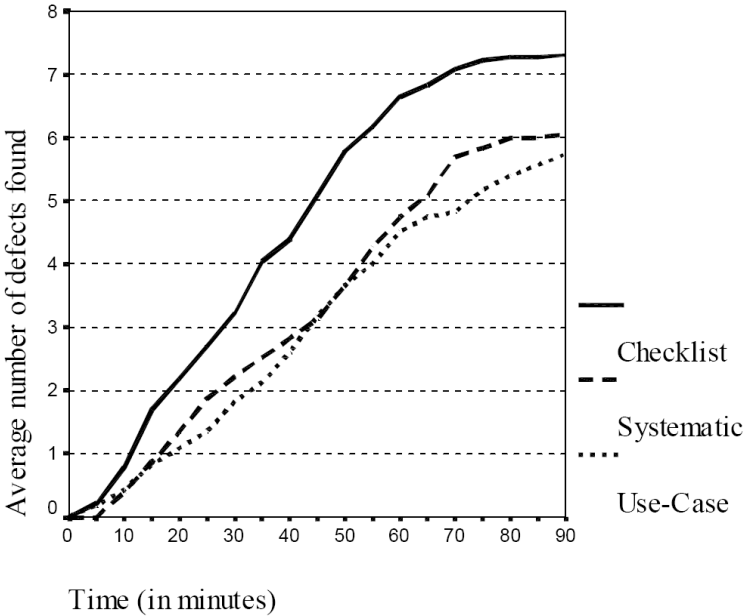


Figure 5: Elapsed time versus cumulative number of defects found for each of the three types of review.

The defect rate is constant until about 60 minutes into the inspection at which point it levels off with no defects found at all after 90 minutes.

In all three review types the pattern is the same. Defects are found at relatively constant rates through the first 60 minutes of inspection. At that point the checklist-style review levels off sharply; the other review styles level off slightly later. In no case is a defect discovered after 90 minutes.

This is direct and conclusive evidence that reviews should be limited to around one hour, not to exceed two hours.

**Uwano 2006: Analysis of eye movements during review<sup>11</sup>**

Four researchers at the Nara Institute of Science and Technology have completed a unique study of the eye movements of a reviewer during a code review. It's always both fascinating and eerie to get a glimpse into our subconscious physiological behaviors.

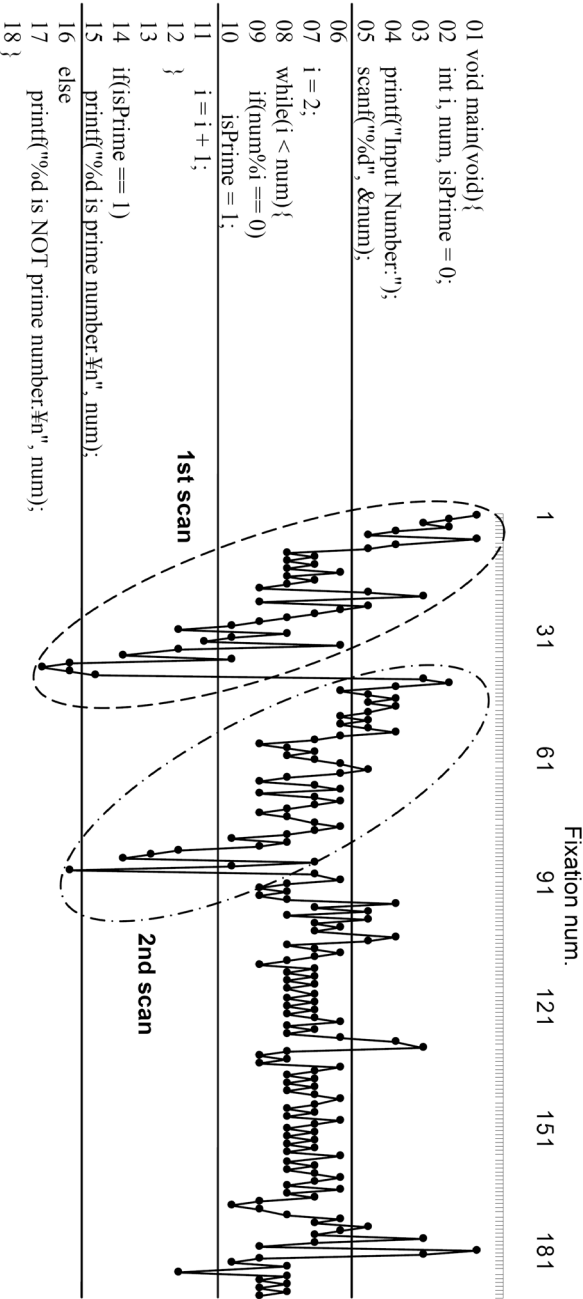
It turns out that certain eye scanning patterns during review correlate with being better at finding the defect. Although this is not really something you can teach someone, it does point out ways in which source code could be organized to facilitate comprehension. That is, specific coding standards could make it easier for developers to understand code in general and for reviewers to find defects in particular.

The researchers used a custom-built system that displayed a short C-language program on a screen while an eye scanner recorded all “fixations” – times when the eye stayed within a 30 pixel radius for longer than 1/20th of a second. Furthermore, because they controlled the display of the source code, fixations were matched up with line numbers. The result is a plot of which line of code was looked at over time.

Six different C snippets were used, each between 12 and 23 lines, each an entire function or small set of functions viewable without scrolling. Five subjects were subjected to each snippet yielding 27 trials (three of the 30 had to be discarded because the subject was distracted during the trial).

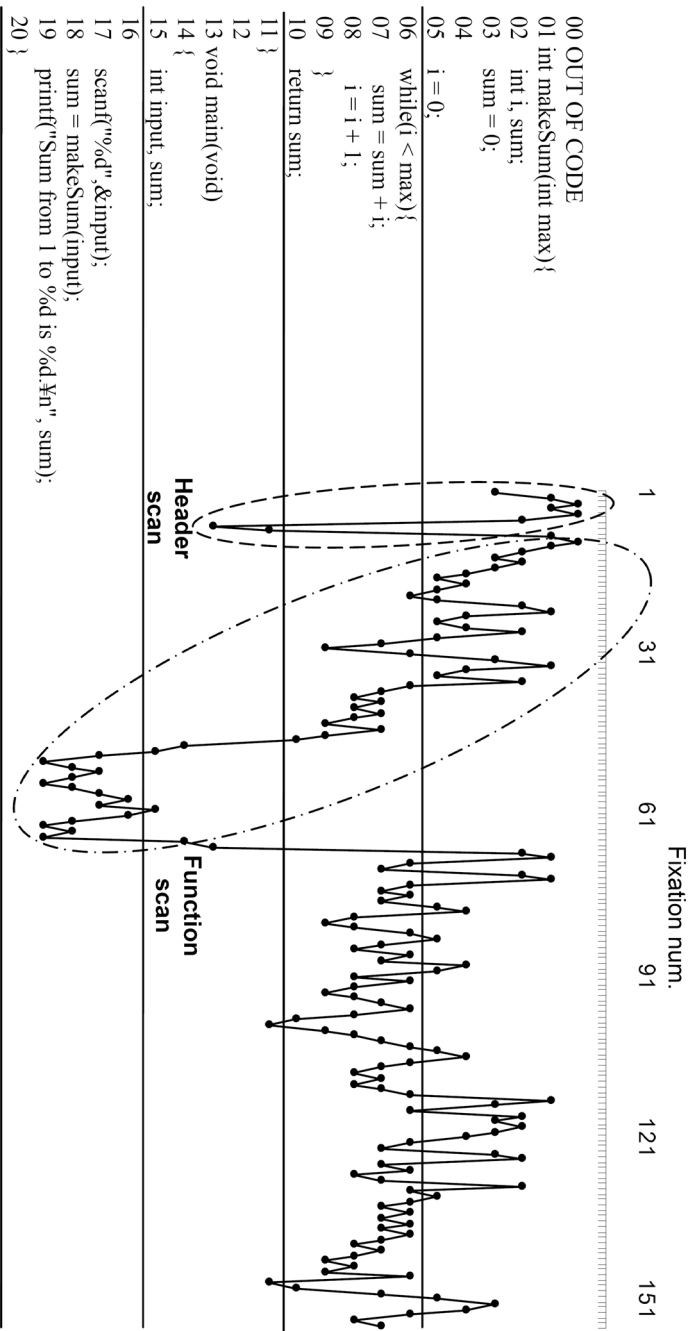
---

<sup>11</sup> Uwano, H., Nakamura, M., Monden, A., and Matsumoto, K. 2006. Analyzing individual performance of source code review using reviewers' eye movement. In *Proceedings of the 2006 Symposium on Eye Tracking Research & Applications* (San Diego, California, March 27 - 29, 2006). ETRA '06. ACM Press, New York, NY, 133-140 © 2006 ACM, Inc. Figures reprinted by permission.



a) Subject E reviewing Prime





b) Subject C reviewing Accumulate

The general pattern is the reviewer would read lines from top to bottom in a “bumpy slope.” That is, generally straight-through but with short, brief “back-tracks” along the way. They called this the “first scan.” Typically 80% of the lines are “touched” during this first scan. Then the reviewer concentrates on a particular portion of the code – 4 or 5 lines – presumably where the reviewer believed the problem was most likely to be.

Other patterns were observed depending on the nature of the code. For example, with code examples containing two functions instead of one, frequently there was a very fast “header scan” where the eyes rested on function declaration lines before the usual “first scan” began. It was also common to see a second scan similar to the first before the concentration begins.

Closer inspection of the eye movements reveals interesting insights into how anyone reads source code for comprehension. For example, the eyes rest on initial variable declarations consistently throughout the session. The mind needs a constant refresher on what these variables mean. Even when there are only two or three local variables, all integers, nothing special or difficult to retain, the pattern is the same. The researchers called this eye movement “retrace declaration.” What this means for code structure is that local variable declarations should also be visible on the same screen as code that uses them. Otherwise the reader will have to scroll back and forth. A common way to enforce this is to limit the number of lines of code in a function. We’ve all heard the arguments for limiting function length for general readability; here’s evidence that cements that notion.

As another example, loops have a tremendous fixation rate, far more even than other control structures such as conditionals. Perhaps this is partly a function of the review task – loops are a common location for errors and just the fact that a section of code repeats makes it more difficult to analyze. The lesson for coding

standards is that loop conditionals should be as simple as possible. For example, avoid setting variables inside the conditional and try to simplify complex compound Boolean expressions.

But back to reviews. The researchers had initially set out to answer the question: Is there anything about eye movements that can be correlated with review effectiveness or efficiency? The answer turned out to be yes.

There is a negative correlation between the amount of time it takes for the “first scan” and defect detection speed. That is, the more time the reviewer spends during that “first scan” period, the *faster* the reviewer will be at finding the defect. This seems contradictory – the reviewer spends more time scanning the code, yet he finds the defect faster than someone who doesn’t bother.

The key is that it’s the first, preliminary scan that the reviewer must spend more time on. When a reviewer doesn’t take enough time to go through the code carefully, he doesn’t have a good idea of where the trouble spots are. His first guess might be off – he might have completely missed a section that would have set off warning bells. The reviewer that takes more time with the initial scan can identify all the trouble spot candidates and then address each one with a high probability of having selected the right area of code to study.

This result has a few ramifications. First, slow down! As we talked about in the conclusion section, the longer you take in review, the more defects you’ll find. Haste makes waste.

Second, a preliminary scan is a useful technique in code review. This experiment demonstrates that a reasonably careful “first scan” actually increases defect detection rates.

**Laitenberger 1999: Factors affecting number of defects<sup>12</sup>**

Under what circumstances would we expect to find more or fewer defects during a review? Does the size of the code under inspection matter? How about the amount of time reviewers spend looking at it? What about the source code language or the type of program?

All of these things are “a factor,” but can we state something stronger than that? Perhaps some things matter more than others.

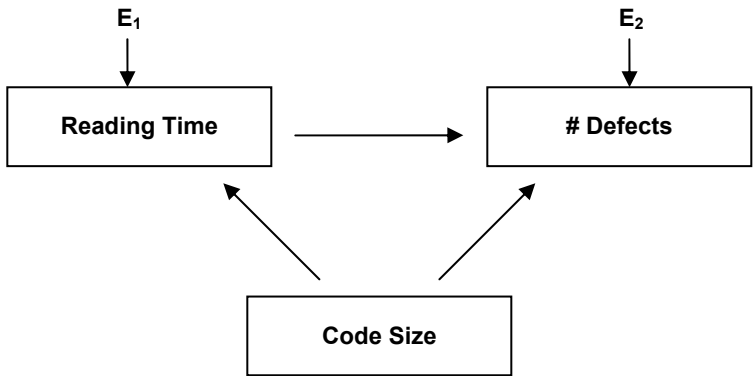
In 1999, three researchers performed an analysis of 300 reviews from Lucent’s Product Realization Center for Optical Networking (PRC-ON) in Nuremberg, Germany. This group spends 15% of their total development time in reviews, but were they using their time wisely? Were they detecting as many defects per hour as possible? If not, what specifically should they do to maximize the defect detection rate?

These were the questions Laitenberger set out to answer. A survey of other studies suggested the two most likely factors in the number of defects found during a review: (a) time spent in preparation, and (b) the size of the code under inspection. So they came up with a causal model – that is, a theoretical model of how they expected these two factors might influence the number of defects. This model is shown in Figure 6.

But drawing a diagram doesn’t prove anything! How can we test whether this model is accurate and how can we measure just how important each of those causal links are?

---

<sup>12</sup> Evaluating a Causal Model of Review Factors in an Industrial Setting. Oliver Laitenberger, Marek Leszak, Dieter Stoll, and Khaled El-Emam, National Research Council Canada.



*Figure 6: Causal model for the two largest factors that determine the number of defects found during review.*

*Arrows indicate a causal link. The “E” values represent external factors not accounted for by the model.*

As you might expect, there’s a statistical system that can do exactly this. It’s called Path Analysis. Given the model above and raw data from the individual reviews, we can determine how each of those arrows really matter<sup>13</sup>.

---

<sup>13</sup> Each review contains three pieces of data: code size, reading time, and number of defects. Then each of those variables is compared pair-wise; the beta coefficient from a logarithmic least-squares analysis is used as the measure of the pair-wise correlation strength. Correlations must be significant at the 0.01 level to be accepted.

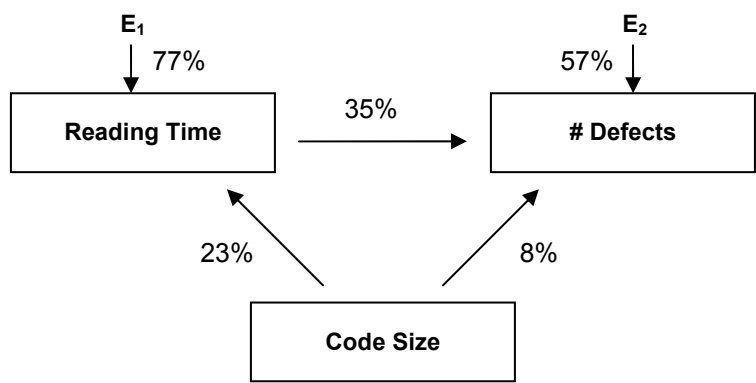


Figure 7: Results of Laitenberger’s path analysis for code reviews. Numbers represent percent of total influence.

Results are similar for design and specification reviews.

The results are shown in Figure 7. There are two important things to notice.

First, reading time is twice as influential as code size<sup>14</sup>. This is unexpected – with more code under the magnifying glass you would expect to find more defects. But in fact it’s the amount of

<sup>14</sup> It may appear that reading time influence is four times larger than code size (35÷8), but note that code size also influences reading time, thereby indirectly influencing number of defects through that path. Thus the total code size influence is  $0.08 + 0.23 \times 0.35 = 16\%$ .

time you spend looking at the code – no matter how big or small the code is – that determines how many defects you find.

Second, the “external factors” are far more important than any of the others. Code size alone predicts only 8% of the variation in number of defects found; reading time predicts 35%, but most of the variation comes from elsewhere. Reviews of new code are different than maintenance code. Complex algorithms differ from complex object relationships which differ from simple procedural code. Languages differ. The experience level of the author and reviewer matters. These external effects collectively constitute the most important factor in how many defects you can expect to find during a review.

This has several implications. First, if you want to find more defects, spend more time reviewing. You could also reduce the amount of code under inspection, but that’s less important. Even though various external factors are collectively more influential, the individual reviewer often cannot control those factors; reading time is something that can be controlled directly.

Second, you cannot average all your metrics and say “We should find X defects per 1000 lines of code.” Each project, language, module, even class could be different. It might be worth breaking your metrics down at these levels in an attempt to find some real patterns.

Ultimately the most influential variable that you can actually control in your review process is inspection time. That’s the knob you turn when you want to find more defects.

## **Conclusions**

Each study in this survey has a different story to tell and uncovers unique insights in the process, but what general information can be gleaned from the literature?

Review for at most one hour at a time.

Although not every study addressed this issue, a common result is that reviewers' effectiveness at finding defects drops off precipitously after one hour. This is true no matter how much material is being reviewed at once. Some studies specifically test for this, seeing how many defects are discovered after 15, 30, 45, 60, 75, 90, and 120 minutes devoted to the task. In all cases there is roughly linear increase in number of defects found up to one hour, then a significant leveling-off after that. This result has been echoed in other studies not covered by this survey.

Study	Review Minutes
Dunsmore, 2000	60
Blakely, 1991	90
Cohen, 2006	90

*Figure 8: Cut-off point where further review produced no (significant) benefit.*

There are at least two explanations for this effect. First, each reviewer is capable of finding only a certain set of defects. No matter how long you stare at the code, there are some problems that you just don't know how to find. This is especially true if the review is being driven by checklist – the reviewer often focuses on the checklist and anything significantly outside that list is not in scope.



The second explanation is that after an hour the human mind becomes saturated. The reviewer cannot process more possibilities because his brain refuses to concentrate. “I’ve been looking at this for too long. I’m sick of it” is a common complaint during extended reviews. This second point could be tested by having reviewers return to the code the next day to see if the same person can find more defects after a “reset.” No study did this, however, and perhaps it doesn’t matter because taking that much time is impractical.

To detect more defects, slow down code readings.

The more time spent in review, the more defects are detected. This might sound obvious; what’s not obvious is that this is by far the dominant factor in the number of defects detected.

This is also true whether or not your private code readings are followed up by an inspection meeting.

Inspection meetings need not be in person.

For the sensitive reader accustomed to institutional formal inspections descended from the legacy of Fagan, Gilb, and Wiegers, this statement is heresy. Traditionally the in-person moderator-directed inspection meeting is considered the lynchpin of a successful process. The synergy<sup>15</sup> arising from a properly-conducted meeting produces results that could never be obtained by any reviewer individually, even with training.

In the past fifteen years this effect has been questioned in many studies and from many perspectives. Several conclusions on this point are clear from all the studies in this survey and many others.

First, the vast majority of defects are found in the pre-meeting private “reading” phase. By the time the meeting starts, all

---

<sup>15</sup> Fagan’s evocative “phantom inspector.”

questionable aspects of the code are already identified. This makes sense; it would be difficult to determine whether a complex algorithm was implemented correctly by discussion rather than by concentrated effort.

Second, the primary result of a meeting is in sifting through and possibly removing false-positives – that is, “defects” found during private code readings which turn out to not actually be defects. Even then, false-positives are often the result of poorly documented or organized code; if a reader is confused, perhaps the code should be changed anyway to avoid future confusion, even if this just means introducing better code comments.

The result is that short meetings with just a few participants (or even just the author and a single reviewer) appear to provide the benefits of the inspection meeting (identifying false-positives and knowledge transfer) while keeping inspection rates high (not wasting time). And these “meetings” are just as effective over e-mail or other electronic communication medium.

### Defects per line of code are unreliable.

It’s the forecaster’s dream. How many defects are lurking in these 5000 lines of code? No problem, just take our standard number of defects per kLOC during review and multiply. 12 defects/kLOC? Expect to find 60 defects. If you’ve only found 30 so far, keep looking.

Unfortunately, this is a pipe dream. Studies agree to disagree: this ratio is all over the map. It is possible that more careful study broken out by file type, project, team, or type of application might reveal better numbers<sup>16</sup>. But for now, give up the quest for the “industry standard” density of defects.

---

<sup>16</sup> Our own in-depth analysis of 2500 reviews revealed two significant factors: time spent in review (more time increased defect density) and

Study	Defects/kLOC
Kelly 2003	0.27
Laitenberger 1999	7.00
Blakely 1991	105.28
Cohen 2006	10-120

*Figure 9: Defects per 1000 lines of code as reported by various studies. The pattern is... there is no pattern.*

### Omissions are the hardest defects to find.

It's easy to call out problems in code you're staring at; it's much more difficult to realize that something isn't there at all.

Although most studies mentioned this, none measured this problem specifically. The informal consensus is that a checklist is the single best way to combat the problem; the checklist reminds the reviewer to take the time to look for something that might be missing.

For example, in our own experience the utility of a checklist item like "make sure all errors are handled" is of limited usefulness – this is an obvious thing to check for in all code. But we forgot to kick the build number before a QA session started about 30% of the time. After installing a release checklist we haven't forgotten since.

### Studies are too small.

An unfortunate common element to these studies is that they are almost all too small to have statistical significance. It's rare to find more than 100 reviews or 100 defects in any of them. Most are along the lines of "In our 21 reviews, we found that..."

---

author preparation comments (reduced defect density). See that essay in this collection for details.

Twenty-one reviews are not statistically significant, no matter what data you collect!

This doesn't completely invalidate the studies; it just means that we need to consider all the studies in aggregate, not any one by itself. Also, in each the authors make observations which are interesting and relevant regardless of the metrics they collected.

Study	# Participants	Review Hours	# Defects
Uwano 2006	5	2.5	6
Blakely 1991		7.6	21
Conradi 2003	10	17.3	64
Kelly 2003	7	48.0	147
Dunsmore 2000	64	58.0	7
Laitenberger 1999			3045

*Figure 10: Various statistics from each study.  
Blank cells where data not available.*

*The small numbers show that almost none of the studies are large enough by themselves to be statistically significant.*