

# *1. Schedule versus Defects*

**George Huling**  
**Disciplined Software Consulting**  
**P.O. Box 1753**  
**Agoura Hills, CA 91376-1753**  
**g.huling@ieee.org**  
**(805) 381-4936**  
**<http://www.laacn.org/firms/dsc/>**

---

Few businesses are interested in product quality as an end in itself. Instead, their market determines the necessary level of quality, often called “good enough quality” described in [Bach 1997].<sup>1</sup> However, almost every business could benefit from a reduction in its product’s planned time to market, usually called the cycle time. But if you cannot meet your currently planned schedules, it may be difficult to reduce cycle time, and software projects commonly exceed their planned schedules.

The usual reason for schedule overruns is that good enough quality is not achieved until after the planned date. If the necessary level of quality could be achieved more efficiently, in less schedule time, then perhaps we could make faster progress in reducing software cycle time. At least to this extent, schedule and quality should not be incompatible.

Software Productivity Research, Inc. (SPR) collects data during software assessments and benchmark studies. The Chairman of SPR, Capers Jones, presented some results in [Jones 1998]. He compared “lagging”, “average”, and “leading” software projects in terms of schedule, defects, and productivity normalized for 1000 function point projects. His results are summarized in Table 1. The leading projects take 71% as long as the average and 50% as long as the lagging. The leading projects deliver only 20% as many defects as do the average projects and less than 9% as many as do the lagging projects. Although the most dramatic improvement is in quality, there is significant improvement in both productivity and schedule. The observation that software process improvement leads to shorter schedules, higher delivered qual-

---

1. “References” on page 1-9.

ity, and higher productivity simultaneously is apparently not widely appreciated. Jones<sup>1</sup> says:

**Table 1. Comparison of lagging, average, and leading projects.**

factor	lagging lower quartile	average middle half	leading upper quartile
schedule, months	22.4	15.8	11.2
defects injected per FP	7	5	3
removal effectiveness	75%	85%	95%
defects delivered per FP	1.75	0.75	0.15
productivity, FP per staff month	1–5	6–12	15–50
CMM level	low 1	almost 2	3 or higher

“Software projects in the upper quartile of our data base have shorter schedules, higher quality levels, and higher productivity rates simultaneously. This is not surprising, because the costs, effort, and time to find software defects is usually the largest cost driver and the most significant barrier to rapid development schedules.”

“The reduced levels of defect potentials [defects injected] stem from better methods of defect prevention, while the elevated rates of defect removal efficiency are *always* [my emphasis] due to the utilization of formal design reviews and code inspections. Testing alone is insufficient to achieve defect removal rates higher than about 90% so *all* [my emphasis] of the top-ranked quality organizations utilize inspections also.”

In addition, CMM levels 3 and higher require reviews or inspections, so all projects requiring CMM level 3 or higher must use reviews.

## *Reviews*

---

Formal or Fagan’s inspections [Wheeler et al. 1996], Cleanroom [Linger 1994], and the Personal Software Process (PSP)<sup>2</sup> [Humphrey 1995] are all techniques in which in process reviews are a key component. Unfortunately, these techniques are widely regarded as being oriented entirely toward quality as an end in itself, as being too difficult, and as having an adverse impact on schedule and cost. Reviews are thought to be suitable only for

---

1. [Jones 1998, p. 4]

2. Personal Software Process and PSP are service marks of Carnegie Mellon University.

applications demanding very high quality. For example, the Space Shuttle and commercial fly-by-wire aircraft have made extensive use of rigorous software reviews.

Some commercial companies, Hewlett-Packard (HP) for instance, have been quite successful in their use of reviews. When these successful companies have published data, it always shows that reviews remove defects with less effort and in less elapsed time on average than does testing. Furthermore, the time benefit, or leverage, is greater for reviews earlier in the development cycle. Design reviews have more leverage than code reviews, and requirements reviews have more leverage than design reviews.

If reviews can save time, why are they not more widely used?

---

### *Metrics*

All the successful users of reviews gather metrics. Without data, it is impossible to know whether the reviews are providing enough return on investment (ROI) to justify continuation, and what might be done to make them better.

#### **Not all defects are equal**

Table 2 gives Hewlett-Packard's estimate of the distribution of defect find

**Table 2. Find and fix time distribution.**

Defect %	Time to find and fix	Time %
25%	2 hours/defect	8%
50%	5 hours/defect	40%
20%	10 hours/defect	32%
4%	20 hours/defect	12%
1%	50 hours/defect	8%

and fix times in testing [Grady 1992, p. 75]. The easiest 25% and the hardest 1% of the defects each account for 8% of the total find and fix time.

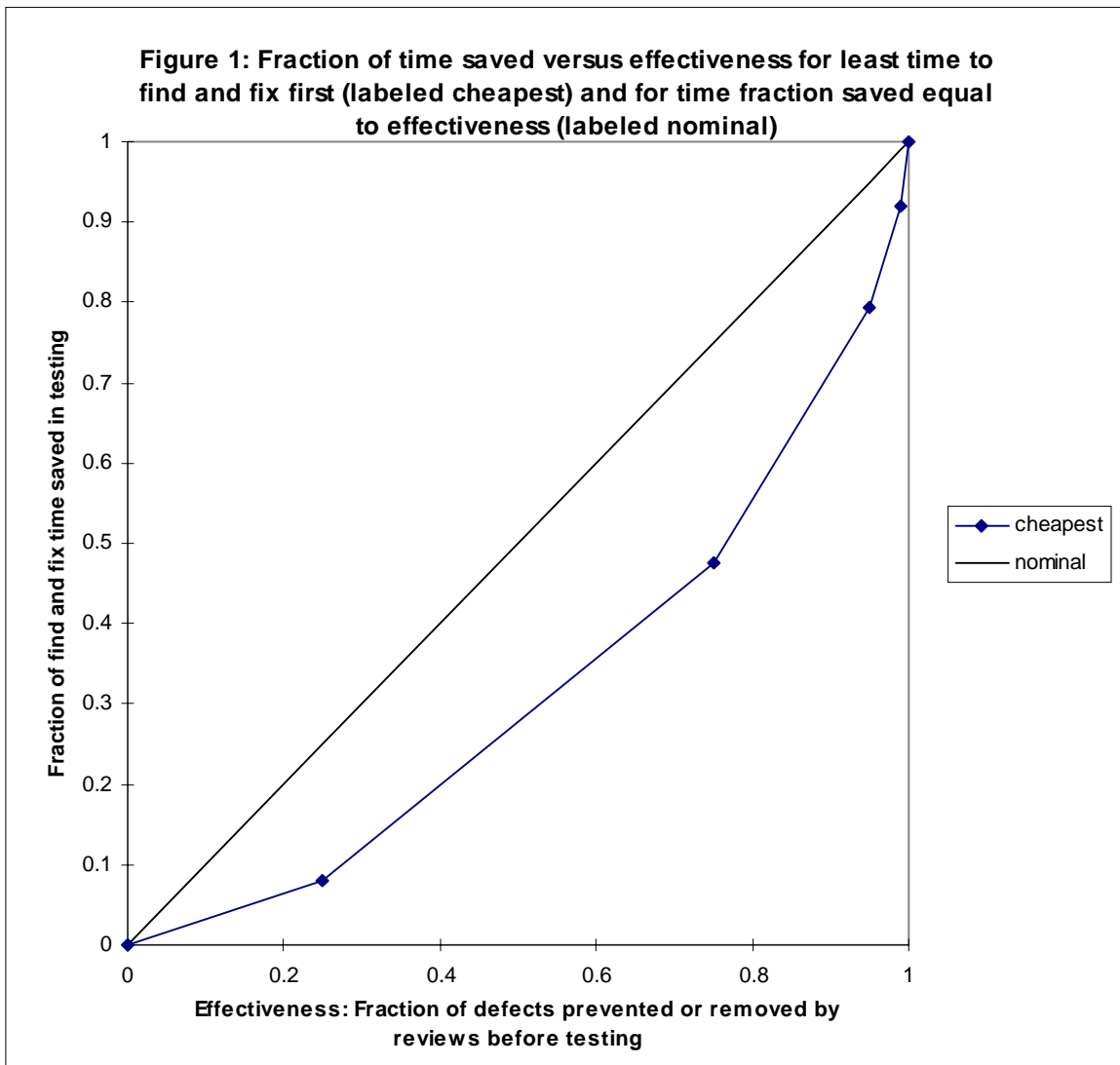
**Table 3. Worst case time savings.**

Effectiveness	Time saved
50%	28%
75%	48%
85%	64%

**Table 3. Worst case time savings.**

Effectiveness	Time saved
90%	72%
95%	80%

Table 3 and Figure 1 show what can happen if reviews find first only the defects having the least find and fix times in testing.



The line labeled “cheapest” was plotted assuming that first all the defects requiring 2 hours to find and fix were found, then all the defects requiring 5 hours to find and fix were found, and so on in the order of the rows in Table

2 from top to bottom. This represents the worst time savings reviews could produce given the distribution in Table 2.

So reviews could find 50% of the defects but only save 28% of the time. To see what this might mean, consider 50 KLOC system having 25 defects per KLOC to be removed in testing without inspections.<sup>1</sup> That's a total of 12,500 defects. Assuming an average of 10 hours per defect in test and an average of 40 hours per KLOC in inspections, some possible outcomes are shown in Table 4. At the beginning of an inspection program, finding 80% of the

**Table 4. Test versus Inspection**

Times are in hours (years)	No inspection	Inspections remove 80% of defects & save 80% of rework	Inspections remove 50% of defects & save 28% of rework	Inspections remove 75% of defects & save 48% of rework
test (average of 10 hr/defect)	12500 (6)	2500	9000	6500
inspection (average of 40hr/KLOC)	0	2000	2000	2000
saving (not counting overhead)	0	8000 (4)	1500 (3/4)	4000 (2)

defects and saving 80% of the test time seems quite unrealistic. Finding 50% of the defects and saving 28% of the test time is more reasonable, perhaps slightly pessimistic.

If something like this is happening and management is not aware of it, management can easily conclude that reviews are not worth the effort. From management's viewpoint, the savings calculation must account for all the overhead costs of training, metrics gathering and analysis, management effort, and so on *in addition to* the direct costs of the inspections.

To make reviews pay off, management needs to ensure that the defect detection techniques are finding a significant fraction of the defects that take more time in testing.

**Not all defects cost time**

Any defect that does not affect behavior cannot be detected by testing and therefore has no find and fix time in testing. Review issues concerning standards, style, or code comments will not be found in testing, and management must be able to distinguish between defects that do not affect behavior and defects that do, even if all defects are corrected. In particular, standards defects should be corrected, otherwise few people will bother observing the standard, and the purpose of the standard will not be achieved.

1. Adapted from overhead charts 9 and 10 of lecture 8 in the instructor's material for [Humphrey 1995].

For example, review teams that find half of the defects affecting behavior should reduce find and fix time by at least 28%. However, if teams find the same number of total defects per unit work product but only half these defects affect behavior, then those teams might reduce find and fix time by as little as 8%. Such a small reduction will certainly result in a negative return on investment.

Defects that can be found by compilers, pretty printers, or other static analysis tools such as lint should all be corrected before reviews. Static analysis is a far more cost effective way of removing these defects, and removing these defects before review enables the reviewers to focus on the defects requiring their attention.

Where do the difficult defects come from?

More recent data from HP [Grady 1997, p.65, 266-267] gives some idea of how defects and rework costs are distributed between the lifecycle phases.

Table 5. Lifecycle Phase Ratios

Phase	% Defects	% Rework	% Rework Range	Difficulty vs. Implementation
requirements	21	49	49 - 55	5.75
design	34	34	29 - 34	2.5
implementation	42	17	14 - 17	1

The data in Table 5 include maintenance which ranged from 20% to 80% of the lifecycle cost depending on the application, and half of maintenance is *knowledge recovery*. Defects and rework found in maintenance are analyzed and assigned to the correct phase. The difficulty ratio is based on data in which maintenance is 55% of the lifecycle, work is 41%, rework is 31%, and knowledge recovery is 28%. Here we have a situation in which  $31\% + 28\% = 59\%$  of the lifecycle cost is redoing things that have been done or known before!

This data indicate that requirements and design defects generate far more than their share of rework and are likely to be the source of the difficult defects. A successful inspection program must be able to remove a high percentage of requirements and design defects.

Reuse

Based on data from [Grady and Caswell 1987, pp. 111,112], Grady [Grady 1997, p. 67] estimates that increasing reuse from a fortuitous 10% to a systematic 50% could reduce total software costs by 20%, assuming the lifecycle profile above. This is certainly a desirable outcome, but as Meyer points out [Meyer 1999] reusable components must be of high quality themselves and have high quality documentation as well. Reusable components must have at least these attributes:

1. Careful specification: precisely defined circumstances and precisely defined results
2. Correctness in all specified situations

3. Robustness: never crash or produce wrong results
4. Easy to determine suitability or applicability—to make identification easy
5. Easy to learn: excellent user documentation—both easy for a novice to learn and complete enough to meet the needs of expert users

Components lacking any of the first three attributes present users with the risk of having unexpected defects and incompatibilities. A component that might crash or produce wrong results without warning presents a serious and often unacceptable risk. If a component lacks the fourth attribute, it will be difficult to determine whether it fits your application. And a component lacking the fifth attribute will be too difficult to learn to use correctly. Many COTS software products lack these attributes to some degree and therefore represent potentially serious risks when incorporated into systems without extensive investigation of the COTS products.

**Use Pareto analysis**

Hewlett-Packard appears to use the classic 80-20 rule—80% of the defects are contained in 20% of the code. The trick is to predict the right 20%, which they do by risk analysis. HP inspects all requirements because requirements reviews have the greatest leverage and these reviews provide a sound basis for risk analysis. They inspect about half the design—the half judged to have the most risk based on the requirements review risk analysis. They inspect about 20% of the code based on risk analysis done during the design phase and static analysis of the code. Table 6 summarizes this approach [Nikolaropoulos 1997]. The idea is to maximize the net return—the saved

**Table 6. Inspection effort by phase.**

Project phase	% inspected
requirements	100%
design	~50%
code	~20%

find and fix time minus the time spent in reviews. [Grady 1992, pp 190-194]

**Metrics are necessary**

To be useful, all the techniques mentioned in this section need to be based on data gathered by a metrics “program”.

1. Learning your defect find and fix time distribution.
2. Measuring how your reviews affect your find and fix time distribution.
3. Measuring the review time lost finding defects that do not affect behavior or that could be found by static analysis.
4. Using Pareto analysis to maximize the benefits.

The cost associated with a metrics program is part of the investment needed to gain the benefits of reviews. Without metrics the results are virtually certain to be disappointing.

### *Better Defect Detection Methods*

---

In a study of space shuttle software inspections, Letovsky [Letovsky et al. 1987] found that 34% of the time was spent doing design reconstruction<sup>1</sup> and 37% of the time was spent in what they called mental simulation, playing computer. Spending one-third of the time in design reconstruction means the reviewers had difficulty understanding *how* the specification was implemented or *why* it was implemented in that way. If the design and its connection to the specification were made explicit, a significant part of the design reconstruction time might be saved.

Mental simulation is very much like testing except humans are able to simulate many test cases in parallel provided the cases differ in data values but not in path through the program. Like testing, mental simulation may overlook cases that reveal defects because a defect may only be revealed by a particular *sequential combination* of data values. These sequential combinations may be much more difficult to see even for experienced reviewers.

We would like techniques that

1. make the design explicit,
2. have a clear connection to the specification, and
3. allow an approach to verification that does not depend on finding all the revealing data combinations.

Rifkin and Deimel [Rifkin and Deimel 1994] present a commercial example in which inspection training was given to three groups of programmers, and one of the groups received training in program comprehension techniques in addition. The group having the training in program comprehension techniques reduced their defect escape rate by 70% while the other groups reduced their escape rate by between 45% and 50%. The comprehension training appeared to be particularly effective in reducing post release defects—the comprehension trained group reduced these by 90% while the other groups reduced them by a negligible amount. The company in this example was very sensitive to post release defects.

---

1. Called *knowledge recovery* in [Grady 1997] where the number is 50%.

---

*References*

---

- Bach 1997** James Bach. Good Enough Quality: Beyond the Buzzword. *IEEE Computer* 30(8): 96-98.
- Grady and Caswell 1987** Robert B. Grady and Deborah L. Caswell. *Software Metrics: Establishing a Company-Wide Program*. Englewood Cliffs, NJ 07632: Prentice-Hall, Inc.
- Grady 1992** Robert B. Grady. *Practical Software Metrics for Project Management and Process Improvement*. Englewood Cliffs, NJ 07632: Prentice Hall P T R.
- Grady 1997** Robert B. Grady. *Successful Software Process Improvement*. Upper Saddle River, NJ 07458: Prentice Hall PTR.
- Humphrey 1995** Watts S. Humphrey. *A Discipline for Software Engineering*. Reading, MA: Addison-Wesley.
- Jones 1998** Capers Jones. *Analyzing the Tools of Software Engineering*. Published as an electronic mail attachment.
- Letovsky et al. 1987** Stanley Letovsky, Jeannine Pinto, Robin Lampert, and Elliot Soloway. A Cognitive Analysis of Code Inspections. Reprinted in [Wheeler et al. 1996, pp. 211-227].
- Linger 1994** Richard C. Linger. Cleanroom Process Model. *IEEE Software* March: 50-58.
- Meyer 1999** Bertrand Meyer. Rules for Component Builders. *Software Development* May 1999:26-30.
- Nikolaropoulos 1997** Evangelos Nikolaropoulos. Another Approach to Testing: Inspection. *Hewlett-Packard Journal* June, Article 12a.  
<http://www.hp.com/hpj/journal.html>
- Rifkin and Deimel 1994** Stan Rifkin and Lionel Deimel. Applying Program Comprehension Techniques to Improve Software Inspections. *19th Annual NASA Software Engineering Laboratory Workshop, Greenbelt, Maryland, Nov. 30-Dec. 1.*
- Wheeler et al. 1996** D. A. Wheeler. B. Brykczynski, and R. N. Meeson, Jr. eds. *Software Inspections: An Industry Best Practice*. Los Alamitos, CA: IEEE Computer Society Press.