SOFTWARE DEFECT ORIGINS AND REMOVAL METHODS

Capers Jones, Vice President and Chief Technology Officer

Namcook Analytics LLC www.Namcook.com

Draft 5.0 December 28, 2012

Abstract

The cost of finding and fixing bugs or defects is the largest single expense element in the history of software. Bug repairs start with requirements and continue through development. After release bug repairs and related customer support costs continue until the last user signs off. Over a 25 year life expectancy of a large software system in the 10,000 function point size range almost 50 cents out of every dollar will go to finding and fixing bugs.

Given the fact that bug repairs are the most expensive element in the history of software, it might be expected that these costs would be measured carefully and accurately. They are not. Most companies do not measure defect repair costs, and when they do they often use metrics that violate standard economic assumptions such as "lines of code" and "cost per defect" neither of which measure the value of software quality. Both of these measures distort quality economics. Lines of code penalize high-level languages. Cost per defect penalizes quality.

Poor measurement practices have led to the fact that a majority of companies do not know that achieving high levels of software quality will shorten schedules and lower costs at the same time. But testing alone is insufficient. A synergistic combination of defect prevention, pre-test defect removal, and formal testing using mathematical methods all need to be part of the quality technology stack.

SOFTWARE DEFECT ORIGINS AND REMOVAL METHODS

Introduction

The software industry spends about $0.50 out of every $1.00 expended for development and maintenance on finding and fixing bugs. Most forms of testing are below 35% in defect removal efficiency or remove only about one bug out of three. All tests together seldom top 85% in defect removal efficiency. About 7% of bug repairs include new bugs. About 6% of test cases have bugs of their own. These topics need to be measured, controlled, and improved. Security flaws are leading to major new costs for recovery after attacks. Better security is a major subset of software quality.

A synergistic combination of defect prevention, pre-test defect removal, and formal testing by certified personnel can top 99% in defect removal efficiency while simultaneously lowering costs and shortening schedules.

For companies that know how to achieve it, high quality software is faster and cheaper than low quality software. This article lists of all of the major factors that influence software quality as of year-end 2012.

These quality-related topics can all be measured and predicted using the author's Software Risk Master ™ tool. Additional data is available from the author's books, The Economics of Software Quality, Addison Wesley 2011 and Software Engineering Best Practices, McGraw Hill 2010.

Software Defect Origins

Software defects originate in multiple origins. The approximate U.S. total for defects in requirements, design, code, documents, and bad fixes is 5.00 per function point. Best in class projects are below 2.00 per function point. Projects in litigation for poor quality can top 7.00 defects per function point.

Defect potentials circa 2012 for the United States average:

- Requirements 1.00 defects per function point
- Design 1.25 defects per function point
- Code 1.75 defects per function point
- Documents 0.60 defects per function point
- Bad fixes 0.40 defects per function point
- Totals 5.00 defects per function point

The major defect origins include:

1. Functional requirements
2. Non-functional requirements
3. Architecture
4. Design
5. New source code
6. Uncertified reused code from external sources
7. Uncertified reused code from legacy applications
8. Uncertified reused designs, architecture, etc.
9. Uncertified reused test cases
10. Documents (user manuals, HELP text, etc.)
11. Bad fixes or secondary defects in defect repairs (7% is U.S. average)
12. Defects due to creeping requirements that bypass full quality controls
13. Bad test cases with defects in them (6% is U.S. average)
14. Data defects in data bases and web sites
15. Security flaws that are invisible until exploited

Far too much of the software literature concentrates on code defects and ignores the more numerous defects found in requirements and design. It is also interesting that many of the companies selling quality tools such as static analysis tools and test tools focus only on code defects.

Unless requirement and design defects are prevented or removed before coding starts, they will eventually find their way into the code where it may be difficult to remove them. It should not

be forgotten that the famous "Y2K" problem ended up in code, but originated as a corporate requirement to save storage space.

Some of the more annoying Windows 8 problems, such as the hidden and arcane method needed to shut down Windows 8, did not originate in the code, but rather in questionable upstream requirements and design decisions.

Why the second most common operating system command is hidden from view and requires three mouse clicks to execute it is a prime example of why applications need requirement inspections, design inspections, and usability studies as well as ordinary code testing.

Proven Methods for Preventing and Removing Software Defects

Defect Prevention

The set of defect prevention methods can lower defect potentials from U.S. averages of about 5.00 per function point down below 2.00 per function point. Certified reusable materials are the most effective known method of defect prevention. A number of Japanese quality methods are beginning to spread to other countries and are producing good results. Defect prevention methods include:

1. Joint Application Design (JAD)
2. Quality function deployment (QFD)
3. Certified reusable requirements, architecture, and design segments
4. Certified reusable code
5. Certified reusable test plans and test cases (regression tests)
6. Kanban for software (mainly in Japan)
7. Kaizen for software (mainly in Japan)
8. Poka-yoke for software (mainly in Japan)
9. Quality circles for software (mainly in Japan)
10. Six Sigma for Software
11. Achieving CMMI levels => 3 for critical projects
12. Using quality-strong methodologies such as RUP and TSP
13. Embedded users for small projects < 500 function points
14. Formal estimates of defect potentials and defect removal before starting projects
15. Formal estimates of cost of quality (COQ) and technical debt (TD) before starting
16. Quality targets such as > 97% defect removal efficiency (DRE) in all contracts
17. Function points for normalizing quality data
18. Analysis of user-group requests or customer suggestions for improvement

Analysis of software defect prevention requires measurement of similar projects that use and do not use specific approaches such as JAD or QFD. Of necessity studying defect prevention needs large numbers of projects and full measures of their methods and results.

Note that two common metrics for quality analysis, "lines of code" and "cost per defect" have serious flaws and violate standard economic assumptions. These two measures conceal, rather than reveal, the true economic value of high software quality.

Function point metrics are the best choice for quality economic studies. The new SNAP non-functional size metric has recently been released, but little quality data is available because that metric is too new.

The new metric concept of "technical debt" is discussed later in this article, but needs expansion and standardization.

Pre-Test Defect Removal

What happens before testing is even more important than testing itself. The most effective known methods of eliminating defects circa 2012 include requirements models, automated proofs, formal inspections of requirements, design, and code; and static analysis of code and text.

These methods have been measured to top 85% in defect removal efficiency individually. Methods such as inspections also raise testing defect removal efficiency by more than 5% for each major test stage. The major forms of pre-test defect removal include:

1. Desk checking by developers
2. Debugging tools (automated)
3. Pair programming (with caution)
4. Quality Assurance (QA) reviews of major documents and plans
5. Formal inspections of requirements, design, code, UML, and other deliverables
6. Formal inspections of requirements changes
7. Informal peer reviews of requirements, design, code
8. Editing and proof reading critical requirements and documents
9. Text static analysis of requirements, design
10. Code static analysis of new, reused, and repaired code
11. Running FOG and FLESCH readability tools on text documents
12. Requirements modeling (automated)
13. Automated correctness proofs
14. Refactoring
15. Independent verification and validation (IV&V)

Pre-test inspections have more than 40 years of empirical data available and rank as the top method of removing software defects, consistently topping 85% in defect removal efficiency (DRE).

Static analysis is a newer method that is also high in DRE, frequently toping 65%. Requirements modeling is another new and effective method that has proved itself on complex software such as that operating the Mars Rover. Requirements modeling and inspections can both top 85% in defect removal efficiency (DRE).

One of the more unusual off shoots of some of the Agile methods such as extreme programming (XP) is "pair programming." The pair programming approach is included in the set of pre-test defect removal activities.

With pair programming two individuals share an office and work station and take turns coding while the other observes.

This should have been an interesting experiment, but due to poor measurement practices it has started into actual use, with expensive results. Individual programmers who use static analysis and inspections have better quality at about half the cost and 75% of the schedule of a pair.

If two top guns are paired the results will be good, but the costs about 40% higher than either one working alone. Since there is a severe shortage of top-gun software engineers, it is not cost effective to have two of them working on the same project. It would be better for each of them to tackle a separate important project. Top-guns only comprise about 5% of the overall software engineering population.

If a top gun is paired with an average programmer, the results will be better than the average team member might product, but about 50% more expensive. The quality is no better than the more experienced pair member working alone. If pairs are considered a form of mentoring there is some value for improving the performance of the weaker team member.

If two average programmers are paired the results will still be average, and the costs will be about 80% higher than either one alone.

If a marginal or unqualified person is paired with anyone, the results will be suboptimal and the costs about 100% higher than the work of the better team member working alone. This is because the unqualified person is a drag on the performance of the pair.

Since there are not enough qualified top-gun programmers to handle all of the normal work in many companies, doubling them up adds costs but subtracts from the available work force.

There are also 115 occupation groups associated with software. If programmers are to be paired, why not pair architects, designers, testers, and project managers?

Military history is not software but it does provide hundreds of examples that shared commands often lead to military disaster. The Battle of Cannae is one such example, since the Roman commanders alternated command days.

Some pairs of authors can write good books, such as Douglas Preston and Lincoln Child. But their paired books are no better than the books each author wrote individually

Pairing should have been measured and studied prior to becoming an accepted methodology, but instead it was put into production with little or no empirical data. This phenomenon of rushing to use the latest fad without any proof that it works is far too common for software.

Most of the studies of pair programming do not include the use of inspections or static analysis. They merely take a pair of programmers and compare the results against one unaided programmer who does not use modern pre-test removal methods such as static analysis and peer reviews.

Two carpenters using hammers and hand saws can certainly build a shed faster than one carpenter using a hammer and a hand saw. But what about one carpenter using a nail gun and an electric circular saw? In this case the single carpenter might well win if the pair is only using hammers and hand saws.

By excluding other forms of pre-test defect removal such as inspections and static analysis the studies of pair programming are biased and incomplete.

Test Defect Removal

Testing has been the primary software defect removal method for more than 50 years. Unfortunately most forms of testing are only about 35% efficient or find only one bug out of three.

Defects in test cases themselves and duplicate test cases lower test defect removal efficiency. About 6% of test cases have bugs in the test cases themselves. In some large companies as many as 20% of regression test libraries are duplicates which add to testing costs but not to testing rigor.

Due to low defect removal efficiency at least eight forms of testing are needed to achieve reasonably efficient defect removal efficiency. Pre-test inspections and static analysis are synergistic with testing and raise testing efficiency.

Tests by certified test personnel using test cases designed with formal mathematical methods have the highest levels of test defect removal efficiency and can top 65%. The major forms of test-related factors for defect removal include:

1. Certified test personnel
2. Formal test plans published and reviewed prior to testing
3. Certified reusable test plans and test cases for regression testing
4. Mathematically based test case design such as using design of experiments
5. Test coverage tools for requirements, code, data, etc.
6. Automated test tools
7. Cyclomatic complexity tools for all new and changed code segments
8. Test library control tools
9. Capture-recapture testing
10. Defect tracking and routing tools
11. Inspections of major code changes prior to testing
12. Inspections of test libraries to remove bad and duplicate test cases
13. Special tests for special defects; performance, security, etc.
14. Full suite of test stages including:
    a. Subroutine test
    b. Unit test
    c. New function test
    d. Regression test
    e. Component test
    f. Performance test
    g. Usability test
    h. Security test
    i. System test
    j. Supply-chain test
    k. Cloud test
    l. Data migration test
    m. ERP link test
    n. External beta test
    o. Customer acceptance test
    p. Independent test (primarily military projects)

Testing by itself without any pre-test inspections or static analysis is not sufficient to achieve high quality levels. The poor estimation and measurement practices of the software industry have long slowed progress on achieving high quality in a cost-effective fashion.

However modern risk-based testing by certified test personnel with automated test tools who also use mathematically-derived test case designs and also tools for measuring test coverage and cyclomatic complexity can do a very good job and top 65% in defect removal efficiency for the test stages of new function test, component test, and system test.

Untrained amateur personnel such as developers themselves seldom top 35% for any form of testing. Also the "bad fix injection" rate or new bugs added while fixing older bugs tops 7% for repairs by ordinary development personnel.

Bad fixes are inversely proportional to cyclomatic complexity, and also inversely proportional to experience. Bad fixes by a top-gun software engineer working with software with low cyclomatic complexity can be only a fraction of 1%.

At the other end of the spectrum, bad fixes by a novice trying to fix a bug in an error-prone module with high cyclomatic complexity can top 25%.

In one lawsuit where the author was an expert witness the vendor tried four times over nine months to fix a bug in a financial package. Each of the first four fixes failed, and each added new bugs. Finally after nine months the fifth bug repair fixed the original bug and did not cause regressions. By then the client had restated prior year financials, and that was the reason for the lawsuit.

Another issue that is seldom discussed in the literature is that of bugs or errors in the test cases themselves. On average about 6% of test cases contain errors. Running defective test cases adds costs to testing but nothing to defect removal efficiency. In fact defective test cases lower DRE.

National Defect Removal Efficiency and Software Quality

Some countries such as Japan, India, and South Korea place a strong emphasis on quality in all manufactured products including software. Other countries, such a China and Russia, apparently have less interest and less understanding of quality economics and seem to lag in quality estimation and measurement. Among the quality-strong countries Japan, for example, had more than 93% DRE on the projects examined by the author.

From a comparatively small number of international studies the approximate national rankings for defect removal efficiency levels of the top 20 countries in terms of DRE are:

Countries

1. Japan
2. India
3. Denmark
4. South Korea
5. Switzerland
6. Israel
7. Canada
8. United Kingdom

9. Sweden
10. Norway
11. Netherlands
12. Hungary
13. Ireland
14. United States
15. Brazil
16. France
17. Australia
18. Austria
19. Belgium
20. Finland

All of the countries in the top 20 can produce excellent software and often do. Countries with significant amounts of systems and embedded software and defense software are more likely to have good quality control than countries producing mainly information technology packages.

Almost all countries in 2012 produce software in significant volumes. More than 150 countries produce millions of function points per year. The preliminary ranks shown here indicate that more studies are needed on international software quality initiatives.

Countries that might be poised to join the top-quality set in the future include Malaysia, Mexico, the Philippines, Singapore, Taiwan, Thailand, and Viet Nam. Russia and China should be top-ranked but are not, other than Hong Kong.

Quality measures and predictive quality estimation are necessary precursors to achieving top quality status. Defect prevention and pre-test defect removal must be added to testing to achieve top-rank status.

One might think that the wealthier countries of the Middle East such as Dubai, Saudi Arabia, Kuwait, and Jordan would be in the top 20 due to regional wealth and the number of major software-producing companies located there. Although the Middle East ranks near the top in requesting benchmark data, little or no data has been published from the region about software quality.

Industry Defect Removal Efficiency and Software Quality

In general the industries that produce complex physical devices such as airplanes, computers, medical devices, and telephone switching systems have the highest levels of defect removal efficiency, the best quality measures, and the best quality estimation capabilities.

This is a necessity because these complex devices won't operate unless quality approaches zero defects. Also, the manufacturers of such devices have major liabilities in case of failures including possible criminal charges. The top 20 industries in terms of defect removal efficiency are:

Industries

1. Government – intelligence agencies
2. Manufacturing – medical devices
3. Manufacturing – aircraft
4. Manufacturing – mainframe computers
5. Manufacturing – telecommunications switching systems
6. Telecommunications – operations
7. Manufacturing – defense weapons systems
8. Manufacturing – electronic devices and smart appliances
9. Government – military services
10. Entertainment – film and television production
11. Manufacturing – pharmaceuticals
12. Transportation - airlines
13. Manufacturing – tablets and personal computers
14. Software – commercial
15. Manufacturing – chemicals and process control
16. Banks – commercial
17. Banks – investment
18. Health care – medical records
19. Software – open source
20. Finance – credit unions

As of 2012 more than 300 industries produce software in significant volumes. Some of the lagging industries that are near the bottom in terms of software defect removal efficiency levels are those of state governments, municipal governments, wholesale chains, retail chains, public utilities, cable television billing and finance, and some but not all insurance companies in the areas of billing and accounting software.

For example in Rhode Island one of the author's insurance companies seems to need more than a week to post payments and often loses track of payments. Once the author's insurance company even sent back a payment check with a note that it had been "paid too early." Apparently the

company was unable to add early payments to accounts. (The author was leaving on an international trip and wanted to pay bills prior to departure.)

Another more recent issue involved the insurance company unilaterally changing all account numbers. Unfortunately they seemed not to develop a good method for mapping clients' old account numbers to the new account numbers.

A payment made just after the cut over but using the old account number required 18 days from when the check reached the insurance company until it was credited to the right account. Before that the company had sent out a cancellation notice for the policy. From discussions with other clients, apparently the company loses surprisingly many payments. This is a large company with a large IT staff and hundreds of clerical workers.

Companies that are considering outsourcing may be curious as to the placement of software outsource vendors. From the author's studies of various industries outsource vendors rank as number 25 out of 75 industries for ordinary information technology outsourcing. For embedded and systems software outsourcing the outsource vendors are approximately equal to industry averages for aircraft, medical device, and electronic software packages.

Another interesting question is how good are the defect removal methods practiced by quality companies themselves such as static analysis companies, automated test tool companies, independent testing companies, and defect tracking tool companies?

Interestingly, these companies publish no data about their own results and seem to avoid having outside consulting studies done that would identify their own defect removal efficiency levels. No doubt the static analysis companies use their own tools on their own software, but they do not publish accurate data on the measured effectiveness of these tools.

All of the test and static analysis companies should publish annual reports that show ranges of defect removal efficiency (DRE) results using their tools, but none are known to do this.

Software Development Methods

Some software development methods such as IBM's Rational Unified Process (RUP) and Watts Humphrey's Team Software Process (TSP) can be termed "quality strong" because they lower defect potentials and elevate defect removal efficiency levels.

Other methods such as Waterfall and Cowboy development can be termed "quality weak" because they raise defect potentials and have low levels of defect removal efficiency. The 30 methods shown here are ranked in approximate order of quality strength. The list is not absolute and some methods are better than others for specific sizes and types of projects. Development methods in rank order of defect prevention include:

1. Mashup (construction from certified reusable components)
2. Hybrid
3. IntegraNova
4. TSP/PSP
5. RUP
6. T-VEC
7. Extreme Programming (XP)
8. Agile/Scrum
9. Data state design (DSD)
10. Information Engineering (IE)
11. Object-Oriented (OO)
12. Rapid Application Development (RAD)
13. Evolutionary Development (EVO)
14. Jackson development
15. Structured Analysis and Design Technique (SADT)
16. Spiral development
17. Structured systems analysis and design method (SSADM)
18. Iterative development
19. Flow-based development
20. V-Model development
21. Prince2
22. Merise
23. Data state design method (DSDM)
24. Clean-room development
25. ISO/IEC
26. Waterfall
27. Pair programming
28. DoD 2167A
29. Proofs of correctness (manual)
30. Cowboy

Once again this list is not absolute and situations change. Since Agile development is so popular, it should be noted that Agile is fairly strong in quality but not the best in quality. Agile projects frequently achieve DRE in the low 90% range, which is better than average but not top-ranked.

Agile lags many leading methods in having very poor quality measurement practices. The poor measurement practices associated with Agile for both quality and productivity will eventually lead CIO's, CTO's, CFO's, and CEO's to ask if actual Agile results are as good as being claimed.

Until Agile projects publish productivity data using function point metrics and quality data using function points and defect removal efficiency (DRE) the effectiveness of Agile remains ambiguous and uncertain.

Studies by the author found Agile to be superior in both quality and productivity to waterfall development, but not as good for quality as either RUP or TSP.

Also, a Google search using phrases such as "Agile failures" and "Agile successes" turns up about as many discussions of failure as success. A new occupation of "Agile coach" has emerged to help reduce the instances of getting off track when implementing Agile.

Overall Quality Control

Successful quality control stems from a synergistic combination of defect prevention, pre-test defect removal, and test stages. The best projects in the industry circa 2012 combined defect potentials in the range of 2.0 defects per function point with cumulative defect removal efficiency levels that top 99%. The U.S. average circa 2012 is about 5.0 bugs per function point and only about 85% defect removal efficiency. The major forms of overall quality control include:

1. Formal software quality assurance (SQA) teams for critical projects
2. Measuring defect detection efficiency (DDE)
3. Measuring defect removal efficiency (DRE)
4. Targets for topping 97% in DRE for all projects
5. Targets for topping 99% in DRE for critical projects
6. Inclusion of DRE criteria in all outsource contracts (> 97% is suggested)
7. Formal measurement of cost of quality (COQ)
8. Measures of "technical debt" but augmented to fill major gaps
9. Measures of total cost of ownership (TCO) for critical projects
10. Monthly quality reports to executives for on-going and released software
11. Production of an annual corporate software status and quality report
12. Achieving > CMMI level 3

IBM started to measure defect origins, defect potentials, and defect removal efficiency (DRE) levels in the early 1970's. These measures were among the reasons for IBM's market success in both hardware and software. High quality products are usually cheaper to produce, are much cheaper to maintain, and bring high levels of customer loyalty.

The original IBM DRE studies used six months after release for calculating DRE, but due to updates that occur before six months that interval was difficult to use and control. The switch from six month to 90-day DRE intervals occurred in 1984.

Defect removal efficiency is measured by accumulating data on all bugs found prior to release and also on bugs reported by clients in the first 90 days of use. If developers found 90 bugs and users reported 10 bugs then DRE is clearly 90%.

The International Software Benchmark Standards Group (ISBSG) uses only a 30 day interval after release for measuring DRE. The author measures both 30-day and 90-day intervals.

Unfortunately the 90-day defect counts average about four to five times larger than the 30-day defect counts, due to installation and learning curves of software which delay normal usage until late in the first month.

A typical 30-day ISBSG count of DRE might show 90 bugs found internally and 2 bugs found in 30 days, for a DRE of 97.82%.

A full 90-day count of DRE would still show 90 bugs found internally but 10 bugs found in three months for a lower DRE of only 90.00%.

Although a fixed time interval is needed to calculate DRE, that does not mean that all bugs are found in only 90 days. In fact the 90-day DRE window usually finds less than 50% of the bugs reported by clients in one calendar year.

Bug reports correlate strongly with numbers of production users of software applications. Unless a software package is something like Windows 8 with more than 1,000,000 users on the first day, it usually takes at least a month to install complex applications, train users, and get them started on production use.

If there are less than 10 users the first month, there will be very few bug reports. Therefore in addition to measuring DRE, it is also significant to record the numbers of users for the first three months of the application's production runs.

If we assume an ordinary information technology application the following table shows the probable numbers of reported bugs after one, two, and three months for 10, 100, and 1000 users:

Defects by Users for Three Months

Month	10 Users	100 Users	1000 Users
1	1	3	6
2	3	9	18
3	6	12	24

As it happens the central column of 100 users for three months is a relatively common pattern.

Note that for purposes of measuring defect removal efficiency a single month of usage tends to yield artificially high levels of DRE due to a normal lack of early users.

Companies such as IBM with continuous quality data are able to find out many interesting and useful facts about defects that escape and are delivered to clients. For example for financial software there will be extra bug reports at the end of standard fiscal years, due to exercising annual routines.

Also of interest is the fact that about 15% of bug reports are "invalid" and not true bugs at all. Some are user errors, some are hardware errors, and some are bugs against other software packages that were mistakenly reported to the wrong place. It is very common to confuse bugs in operating systems with bugs in applications.

As an example of an invalid defect report, the author's company once received a bug report against a competitive product, sent to us by mistake. Even though this was not a bug against our software, we routed it to the correct company and sent a note back to the originator as a courtesy.

It took about an hour to handle a bug against a competitive software package. Needless to say invalid defects such as this do not count as technical debt or cost of quality (COQ). However they do count as overhead costs.

An interesting new metaphor called "technical debt" was created by Ward Cunningham and is now widely deployed, although it is not deployed the same way by most companies. Several software quality companies such as OptiMyth in Spain, CAST Software, and SmartBear feature technical debt discussions on their web sites.

The concept of technical debt is intuitively appealing. Shortcuts made during development that lead to complex code structures or to delivered defects will have to be fixed at some point in the future. When the time comes to fix these problems downstream, the costs will be higher and the schedules longer than if they had been avoided in the first place.

The essential concept of technical debt is that questionable design and code decisions have increasing repair costs over time. As a metaphor or interesting concept technical debt has much to recommend it.

But the software industry is far from sophisticated in understanding finance and economic topics. In fact for more than 50 years the software industry has tried to measure quality costs with "lines of code" and "cost per defect" which are so inaccurate as to be viewed as professional malpractice for quality economics.

Also, many companies only measure about 37% of software project effort and 38% of software defects. Omitting unpaid overtime, managers, and specialists are common gaps. Omitting bugs found in requirements, design, and by unit testing are common quality omissions.

Until the software industry adopts standard charts of accounts and begins to use generally acceptable accounting principles (GAAP) measures of technical debt will vary widely from company to company and not be comparable.

Technical debt runs head on into the general ineptness of the software world in understanding and measuring the older cost of quality (COQ) in a fashion that matches standard economic assumptions. Cost per defect penalizes quality. Lines of code penalize modern high-level languages and of course make requirements and design defects invisible. Defect repair costs per function point provide the best economic indicator. However the new SNAP metric for non-functional requirements needs to be incorporated.

The main issues with technical debt as widely deployed by the author's clients are that it does not include or measure some of the largest quality costs in all of software history.

About 35% of large software systems are cancelled and never delivered at all. The most common reason for cancellation is poor quality. But since the cancelled projects don't get delivered, there are no downstream costs and hence no technical debt either. The costs of cancelled projects are much too large to ignore and just leave out of technical debt.

The second issue involves software that does get delivered and indeed accumulates technical debt in the form of changes that need to be repaired. But some software applications have such bad quality that clients sue the developers for damages. The costs of litigation and the costs of any damages that the court orders software vendors to pay should be part of technical debt.

What about the consequential damages that poor software quality brings to the clients who have been harmed by the upstream errors and omissions? Currently technical debt as used by most companies is limited to internal costs borne by the development organization.

For example suppose a bug in a financial application caused by rushing through development costs the software vendor $100,000 to fix a year after release, and it could have been avoided for only $10,000. The expensive repair is certainly technical debt that might have been avoided.

Now suppose this same bug damaged 10 companies and caused each of them to lose $3,000,000 due to having to restate prior year financial statements. What about the $30,000,000 in consequential damages to users of the software? These damages are currently not considered to be part of technical debt.

If the court orders the vendor to pay for the damages and the vendor is charged $30,000,000 that probably should be part of technical debt. Litigation costs and damages are not currently included in the calculations most companies use for technical debt.

For financial debt there is a standard set of principles and practices called the "Generally Accepted Accounting Principles" or GAAP. The software industry in general, and technical debt in particular, need a similar set of "Software Generally Accepted Accounting Principles" or SGAAP that would allow software projects and software costs to be compared in a uniform fashion.

As this article is being written the United States GAAP rules are being phased out in favor of a newer set of international financial rules called International Financial Reporting Standards (IFRS). Here too software needs a set of "Software International Financial Reporting Standards" or SIFRS to ensure accurate software accounting across all countries.

Software engineers interested in technical debt are urged to read the GAAP and IFRS accounting standards and familiarize themselves with normal cost accounting as a precursor to applying technical debt.

The major GAAP principles are relevant to software measures and also to technical debt:

1. Principle of regularity
2. Principle of consistency
3. Principle of sincerity
4. Principle of permanence of methods
5. Principle of non-compensation or not replacing a debt with an asset
6. Principle of prudence
7. Principle of continuity
8. Principle of periodicity
9. Principle of full disclosure
10. Principle of utmost good faith

The major software metric associations such as the International Function Point User Group (IFPUG) and the Common Software Metric International Consortium (COSMIC) should both be participating in establishing common financial principles for measuring software costs, including cost of quality and technical debt. However neither group has done much outside of basic sizing of applications. Financial reporting is still ambiguous for the software industry as a whole.

Interestingly the countries of Brazil and South Korea, which require function point metrics for government software contracts, appear to be somewhat ahead of the United States and Europe in melding financial accounting standards with software projects. Even in Brazil and South Korea costs of quality and technical debt remain ambiguous.

Many companies are trying to use technical debt because it is an intriguing metaphor that is appealing to CFO's and CEO's. However without some form of SIFRS or standardized accounting principles every company in every country is likely to use technical debt with random rules that would not allow cross-country, cross-company, or cross-project comparisons.

Harmful Practices to be Avoided

Some of the observations of harmful practices stem from lawsuits where the author has worked as an expert witness. Discovery documents and depositions reveal quality flaws that are not ordinarily visible or accessible to standard measurements.

In every case where poor quality was alleged by the plaintiff and proven in court, there was evidence that defect prevention was lax, pre-test defect removal such as inspections and static analysis were bypassed, and testing was either perfunctory or truncated to meet arbitrary schedule targets.

These poor practices were unfortunate because a synergistic combination of defect prevention, pre-test defect removal, and formal testing leads to short schedules, low costs, and high quality at the same time.

The most severe forms of schedule slips are due to starting testing with excessive numbers of latent defects, which stretch out testing intervals by several hundred percent compared to original plans. Harmful and dangerous practices to be avoided are:

1. Bypassing pre-test inspections
2. Bypassing static analysis
3. Testing by untrained, uncertified amateurs
4. Truncating testing for arbitrary reasons of schedule
5. The "good enough" quality fallacy
6. Using "lines of code" for data normalization (professional malpractice)
7. Using "cost per defect" for data normalization (professional malpractice)
8. Failure to measure bugs at all
9. Failure to measure bugs before release
10. Failure to measure defect removal efficiency (DRE)
11. Error-prone modules (EPM) with high defect densities
12. High cyclomatic complexity of critical modules
13. Low test coverage of critical modules
14. Bad-fix injections or new bugs in bug repairs themselves
15. Outsource contracts that do not include quality criteria and DRE
16. Duplicate test cases that add costs but not test thoroughness
17. Defective test cases with bugs of their own

It is an unfortunate fact that poor measurement practices, failure to use effective quality predictions before starting key projects, and bypassing defect prevention and pre-test defect removal methods have been endemic problems of the software industry for more than 40 years.

Poor software quality is like the medical condition of whooping cough. That condition can be prevented via vaccination and in today's world treated effectively.

Poor software quality can be eliminated by the "vaccination" of early estimation and effective defect prevention. Pre-test defect removal such as inspections and static analysis are effective therapies. Poor software quality is a completely treatable and curable condition.

It is technically possible to lower defect potential from around 5.00 per function point to below 2.00 per function point. It is also technically possible to raise defect removal efficiency (DRE) from today's average of about 85% to at least 99%. These changes would also shorten schedules and reduce costs.

Illustrating Software Defect Potentials and Defect Removal Efficiency (DRE)

Figure one shows overall software industry results in terms of two dimensions. The vertical dimension shows defect potentials or the probable total number of bugs that will occur in requirements, design, code, documents, and bad fixes:

Note that large systems have much higher defect potentials than small applications. It is also harder to remove defects from large systems.

Figure 1: U.S. Ranges of Software Defect Potentials and Defect Removal Efficiency (DRE)

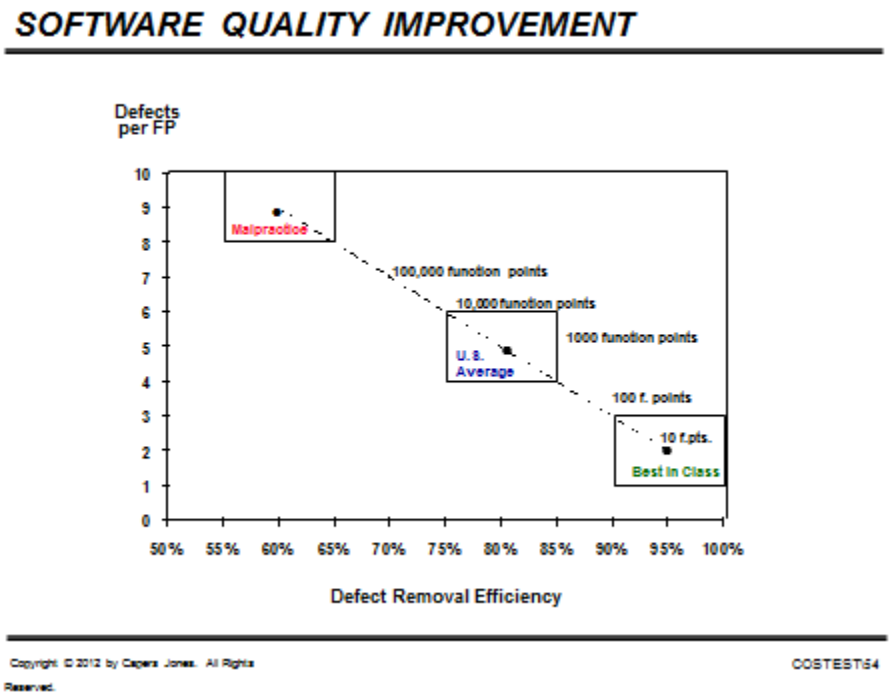
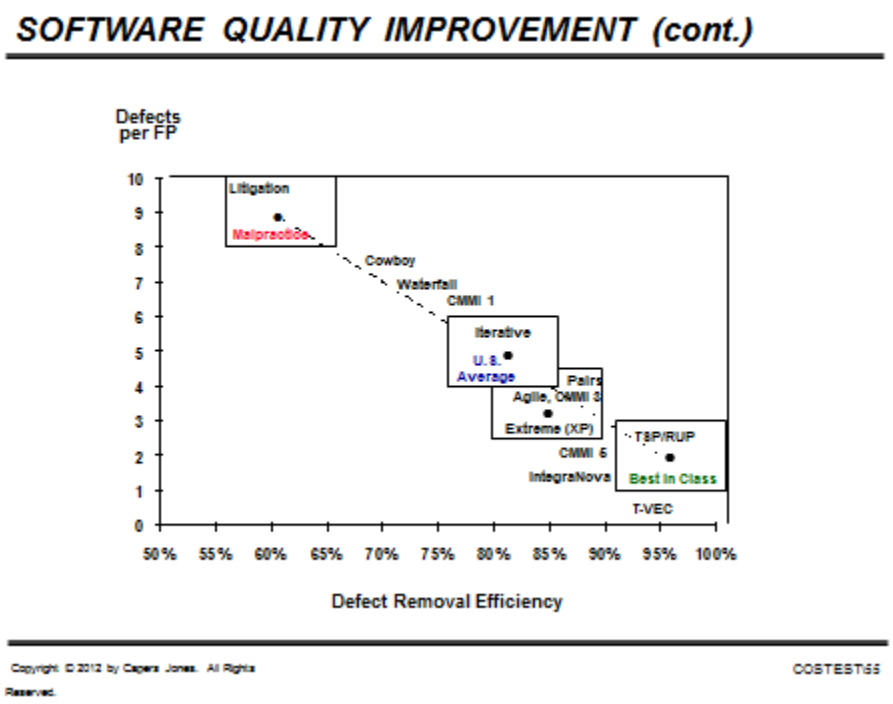


Figure 2 shows the relationship between software methodologies and software defect potentials and defect removal:

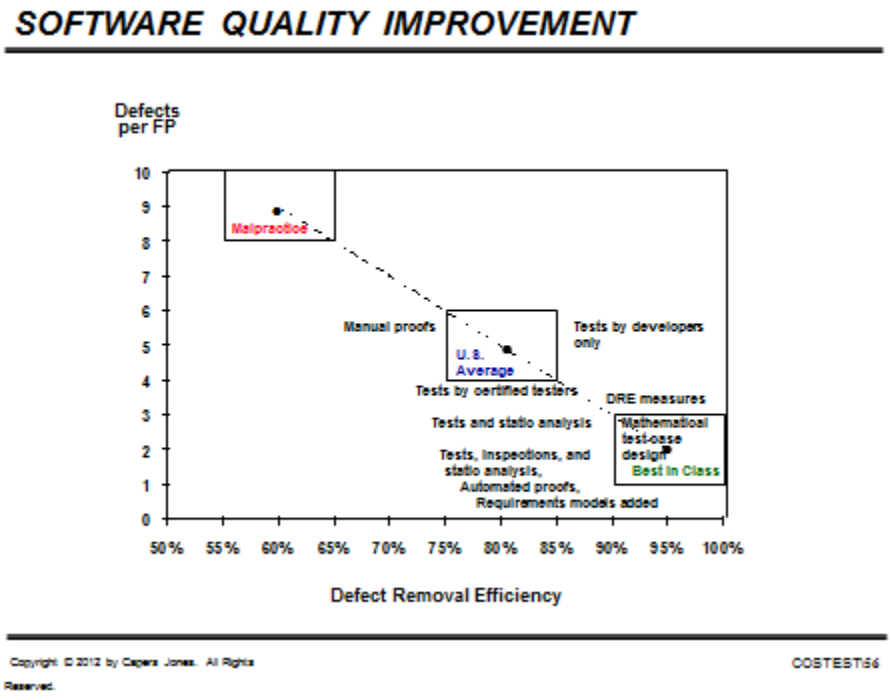
Figure 2: Methodologies and Software Defect Potentials and Removal



Note that "quality strong" methods have fewer defects to remove and remove a much higher percentage of software defects than the "quality weak' methods.

Figure 3 shows the relationship between various forms of defect removal and effective quality results:

Figure 3: Software Defect Removal Methods and Removal Efficiency



Note that testing by untrained amateur developers is neither efficient in finding bugs nor cost effective. A synergistic combination of defect prevention, pre-test defect removal, and formal testing gives the best quality results at the lowest costs and the shortest schedules.

These three illustrations show basic facts about software defects and defect removal efficiency:

- Defect volumes increase with application size
- Quality-strong methods can reduce defect potentials
- Synergistic combinations of defect prevention, pre-test removal and testing are needed.

Achieving a high level of software quality is the end product of an entire chain of methods that start with 1) defect measurements, 2) defect estimation before starting projects, 3) careful defect prevention, 4) pre-test inspections and static analysis, and ends with 5) formal testing using mathematically designed test cases.

All five links of this chain are needed to ensure optimal software quality. Omitting any of the links will lead to poor quality, higher costs, and longer schedules than including all five links.

Quantifying a Best-Case Scenario for Defect Removal Efficiency

To illustrate the principles of optimal defect prevention, pre-test removal, and test defect removal table 1 shows sample outputs from Software Risk Master ™ for a best-case scenario. This scenario assumes 1,000 function points, a top-gun team, CMMI level 5, hybrid methodology, the Objective-C programming language, and a monthly burdened compensation rate of $10,000:

Table 1: Best-Case Scenario	**OUTPUT DATA**
Requirements defect potential	**134**
Design defect potential	**561**
Code defect potential	**887**
Document defect potential	**135**
Total Defect Potential	**1,717**
Per function point	**1.72**
Per KLOC	**32.20**

Defect Prevention	**Efficiency**	**Remainder**	**Bad Fixes**	**Costs**
JAD	27%	**1,262**	**5**	$28,052
QFD	30%	**888**	**4**	$39,633
Prototype	20%	**713**	**2**	$17,045
Models	68%	**229**	**5**	$42,684
Subtotal	**86%**	**234**	**15**	$127,415

Pre-Test Removal	**Efficiency**	**Remainder**	**Bad Fixes**	**Costs**
Desk check	27%	**171**	**2**	$13,225
Static analysis	55%	**78**	**1**	$7,823
Inspections	93%	**5**	**0**	$73,791
Subtotal	**98%**	**6**	**3**	$94,839

Test Removal	Efficiency	Remainder	Bad Fixes	Costs
Unit	32%	4	0	$22,390
Function	35%	2	0	$39,835
Regression	14%	2	0	$51,578
Component	32%	1	0	$57,704
Performance	14%	1	0	$33,366
System	36%	1	0	$63,747
Acceptance	17%	1	0	$15,225
Subtotal	**87%**	1	0	$283,845

				Costs
PRE-RELEASE COSTS		1,734	3	**$506,099**
POST-RELEASE REPAIRS	**(TECHNICAL DEBT)**	1	0	**$658**
MAINTENANCE OVERHEAD				**$46,545**
COST OF QUALITY	**(COQ)**			**$553,302**

Defects delivered	1
High severity	0
Security flaws	0
High severity %	11.58%
Delivered Per FP	0.001
High severity per FP	0.000
Security flaws per FP	0.000
Delivered Per KLOC	0.014
High severity per KLOC	0.002
Security flaws per KLOC	0.001
Cumulative Removal Efficiency	99.96%

This scenario utilizes a sophisticated combination of defect prevention and pre-test defect removal as well as formal testing by certified test personnel. Note that cumulative DRE is 99.96%, which is about as good as ever achieved.

Quantifying a Worst-Case Scenario for Defect Removal Efficiency

To illustrate the principles of inadequate defect prevention, pre-test removal, and test defect removal table 2 shows sample outputs from Software Risk Master ™ for a worst-case scenario. This scenario assumes 1,000 function points, a novice team, CMMI level 1, waterfall methodology, the Java programming language, and a monthly burdened compensation rate of $10,000:

Table 2: Worst-Case Scenario		**OUTPUT DATA**		
Requirements defect potential		**327**		
Design defect potential		**584**		
Code defect potential		**1,109**		
Document defect potential		**140**		
Total Defect Potential		**2,160**		
Per function point		**2.16**		
Per KLOC		**40.50**		
Defect Prevention	**Efficiency**	**Remainder**	**Bad Fixes**	**Costs**
JAD - not used	0%	**2,158**	**0**	$0
QFD - not used	0%	**2,156**	**0**	$0
Prototype	20%	**1,725**	**22**	$17,045
Models - not used	0%	**1,744**	**0**	$0
Subtotal	**19%**	**1,743**	**21**	$17,045
Pre-Test Removal	**Efficiency**	**Remainder**	**Bad Fixes**	**Costs**
Desk check	23%	**1,342**	**67**	$19,734
Static analysis - not used	0%	**1,408**	**0**	$0
Inspections - not used	0%	**1,407**	**0**	$0
Subtotal	**19%**	**1,407**	**67**	$19,734

Test Removal	Efficiency	Remainder	Bad Fixes	Costs
Unit	28%	1,013	51	$59,521
Function	31%	734	73	$134,519
Regression	10%	726	36	$53,044
Component	28%	549	55	$91,147
Performance	6%	568	28	$30,009
System	32%	405	41	$102,394
Acceptance	13%	388	19	$25,058
Subtotal	72%	388	304	$495,691

				Costs
PRE-RELEASE COSTS		2,144	371	$532,470
POST-RELEASE REPAIRS	(TECHNICAL DEBT)	388	19	$506,248
MAINTENANCE OVERHEAD				$298,667
COST OF QUALITY	(COQ)			$1,337,385

Defects delivered	407
High severity	103
Security flaws	46
High severity %	25.25%
Delivered Per FP	0.407
High severity per FP	0.103
Security flaws per FP	0.046
Delivered Per KLOC	7.639
High severity per KLOC	1.929
Security flaws per KLOC	0.868
Cumulative Removal Efficiency	81.14%

Note that with the worst-case scenario defect prevention was sparse and pre-test defect removal omitted the two powerful methods of inspections and static analysis. Cumulative DRE was only 81.14% which is below average and below minimum acceptable quality levels. If this had been an outsource project then litigation would probably have occurred.

It is interesting that the best-case and worst-case scenarios both used exactly the same testing stages. With the best-case scenario test DRE was 87% while the test DRE for the worst-case scenario was only 72%. The bottom line is that testing needs the support of good defect prevention and pre-test defect removal.

Note the major differences in costs of quality (COQ) between the best-case scenarios. The best-case COQ was only $553,302 while the COQ for the worst-case scenario was more than double, or $1,337,385.

The technical debt differences are even more striking. The best-case scenario had only 1 delivered defect and a technical debt of only $658. For the worst-case scenario there were 388 delivered defects and repair costs of $506,248.

For software, not only is quality free but it leads to lower costs and shorter schedules at the same time.

Summary and Conclusions on Software Quality

The software industry spends more money on finding and fixing bugs than for any other known cost driver. This should not be the case. A synergistic combination of defect prevention, pre-test defect removal, and formal testing can lower software defect removal costs by more than 50% compared to 2012 averages. These same synergistic combinations can raise defect removal efficiency (DRE) from the current average of about 85% to more than 99%.

Any company or government group that averages below 95% in cumulative defect removal efficiency (DRE) is not adequate in software quality methods and needs immediate improvements.

Any company or government group that does not measure DRE and does not know how efficient they are in finding software bugs prior to release is in urgent need of remedial quality improvements.

When companies that do not measure DRE are studied by the author during on-site benchmarks, they are almost always below 85% in DRE and usually lack adequate software quality methodologies. Inadequate defect prevention and inadequate pre-test defect removal are strongly correlated with failure to measure defect removal efficiency.

Phil Crosby, the vice president of quality for ITT, became famous for the aphorism “quality is free.” For software quality is not only free but leads to shorter development schedules, lower development costs, and greatly reduced costs for maintenance and total costs of ownership (TCO).

References and Readings on Software Quality

Beck, Kent; Test-Driven Development; Addison Wesley, Boston, MA; 2002; ISBN 10: 0321146530; 240 pages.

Black, Rex; Managing the Testing Process: Practical Tools and Techniques for Managing Hardware and Software Testing; Wiley; 2009; ISBN-10 0470404159; 672 pages.

Chelf, Ben and Jetley, Raoul; "*Diagnosing Medical Device Software Defects Using Static Analysis*"; Coverity Technical Report, San Francisco, CA; 2008.

Chess, Brian and West, Jacob; Secure Programming with Static Analysis; Addison Wesley, Boston, MA; 20007; ISBN 13: 978-0321424778; 624 pages.

Cohen, Lou; Quality Function Deployment – How to Make QFD Work for You; Prentice Hall, Upper Saddle River, NJ; 1995; ISBN 10: 0201633302; 368 pages.

Crosby, Philip B.; Quality is Free; New American Library, Mentor Books, New York, NY; 1979; 270 pages.

Everett, Gerald D. And McLeod, Raymond; Software Testing; John Wiley & Sons, Hoboken, NJ; 2007; ISBN 978-0-471-79371-7; 261 pages.

Gack, Gary; Managing the Black Hole: The Executives Guide to Software Project Risk; Business Expert Publishing, Thomson, GA; 2010; ISBN10: 1-935602-01-9.

Gack, Gary; *Applying Six Sigma to Software Implementation Projects*; http://software.isixsigma.com/library/content/c040915b.asp.

Gilb, Tom and Graham, Dorothy; Software Inspections; Addison Wesley, Reading, MA; 1993; ISBN 10: 0201631814.

Hallowell, David L.; *Six Sigma Software Metrics, Part 1.*; http://software.isixsigma.com/library/content/03910a.asp.

International Organization for Standards; ISO 9000 / ISO 14000; http://www.iso.org/iso/en/iso9000-14000/index.html.

Jones, Capers and Bonsignour, Olivier; The Economics of Software Quality; Addison Wesley, Boston, MA; 2011; ISBN 978-0-13-258220-9; 587 pages.

Jones, Capers; Software Engineering Best Practices; McGraw Hill, New York; 2010; ISBN 978-0-07-162161-8;660 pages.

Jones, Capers; "Measuring Programming Quality and Productivity; IBM Systems Journal; Vol. 17, No. 1; 1978; pp. 39-63.

Jones, Capers; Programming Productivity - Issues for the Eighties; IEEE Computer Society Press, Los Alamitos, CA; First edition 1981; Second edition 1986; ISBN 0-8186—0681-9; IEEE Computer Society Catalog 681; 489 pages.

Jones, Capers; "A Ten-Year Retrospective of the ITT Programming Technology Center"; Software Productivity Research, Burlington, MA; 1988.

Jones, Capers; Applied Software Measurement; McGraw Hill, 3rd edition 2008; ISBN 978=0-07-150244-3; 662 pages.

Jones, Capers; Critical Problems in Software Measurement; Information Systems Management Group, 1993; ISBN 1-56909-000-9; 195 pages.

Jones, Capers; Software Productivity and Quality Today -- The Worldwide Perspective; Information Systems Management Group, 1993; ISBN -156909-001-7; 200 pages.

Jones, Capers; Assessment and Control of Software Risks; Prentice Hall, 1994; ISBN 0-13-741406-4; 711 pages.

Jones, Capers; New Directions in Software Management; Information Systems Management Group; ISBN 1-56909-009-2; 150 pages.

Jones, Capers; Patterns of Software System Failure and Success; International Thomson Computer Press, Boston, MA; December 1995; 250 pages; ISBN 1-850-32804-8; 292 pages.

Jones, Capers; Software Quality – Analysis and Guidelines for Success; International Thomson Computer Press, Boston, MA; ISBN 1-85032-876-6; 1997; 492 pages.

Jones, Capers; Estimating Software Costs; 2nd edition; McGraw Hill, New York; 2007; 700 pages..

Jones, Capers; "The Economics of Object-Oriented Software"; SPR Technical Report; Software Productivity Research, Burlington, MA; April 1997; 22 pages.

Jones, Capers; "Becoming Best in Class"; SPR Technical Report; Software Productivity Research, Burlington, MA; January 1998; 40 pages.

Jones, Capers; "Software Project Management Practices: Failure Versus Success"; Crosstalk, October 2004.

Jones, Capers; "Software Estimating Methods for Large Projects"; Crosstalk, April 2005.

Kan, Stephen H.; Metrics and Models in Software Quality Engineering, 2nd edition; Addison Wesley Longman, Boston, MA; ISBN 0-201-72915-6; 2003; 528 pages.

Land, Susan K; Smith, Douglas B; Walz, John Z; Practical Support for Lean Six Sigma Software Process Definition: Using IEEE Software Engineering Standards; WileyBlackwell; 2008; ISBN 10: 0470170808; 312 pages.

Mosley, Daniel J.; The Handbook of MIS Application Software Testing; Yourdon Press, Prentice Hall; Englewood Cliffs, NJ; 1993; ISBN 0-13-907007-9; 354 pages.

Myers, Glenford; The Art of Software Testing; John Wiley & Sons, New York; 1979; ISBN 0-471-04328-1; 177 pages.

Nandyal; Raghav; Making Sense of Software Quality Assurance; Tata McGraw Hill Publishing, New Delhi, India; 2007; ISBN 0-07-063378-9; 350 pages.

Radice, Ronald A.; High Qualitiy Low Cost Software Inspections; Paradoxicon Publishingl Andover, MA; ISBN 0-9645913-1-6; 2002; 479 pages.

Royce, Walker E.; Software Project Management: A Unified Framework; Addison Wesley Longman, Reading, MA; 1998; ISBN 0-201-30958-0.

Wiegers, Karl E.; Peer Reviews in Software – A Practical Guide; Addison Wesley Longman, Boston, MA; ISBN 0-201-73485-0; 2002; 232 pages.