Software Quality and Software Costs

CAPERS JONES
Capers Jones & Associates LLC

Large software projects above 10,000 function points are hazardous business undertakings. The cancellation rate for large software projects tops 50 percent. Of large applications that are completed, many are late and over budget. One of the main reasons for delay or termination is due to excessive volumes of serious defects that are not discovered until testing, and then extend testing cycles far longer than anticipated. Conversely, large software projects that are successful are always characterized by excellence in both defect prevention and defect removal, including pre-test defect removal. It can be concluded that achieving state-of-the-art levels of software quality is perhaps the most important single objective of software process improvements.

INTRODUCTION

This article explores the application of two metrics frameworks—software cost of quality (CoQ) and software defect containment—to model and manage the cost and delivered quality consequences of alternative strategies for software defect detection and correction during the software development lifecycle. Both leading and lagging indicators are considered.

The data in this article come from two main sources. One source consists of a sample of about 13,000 software applications examined between 1973 and 2011 as a result of benchmark and assessment studies. The other source is information noted during 12 lawsuits dealing with canceled projects or allegations of poor quality where the author and colleagues worked as expert witnesses. Nondisclosure agreements prevent identifying specific organizations. In total, data were derived from about 150 large corporations and 30 government groups at both state and federal levels.

For the data derived from litigation, it is significant that every case except one involved applications larger than 10,000 function points in size. Eleven of the cases were suits by clients against contractors. The 12th case was a suit by corporate shareholders against corporate management, asserting that poor software quality control was lowering the value of the company's stock.

Four common problems surfaced in the lawsuits: 1) poor quality control; 2) poor change control combined with creeping requirements in excess of 1 percent growth per calendar month; 3) inadequate status tracking by management that concealed problems until too late to recover; and 4) either inadequate estimation or the arbitrary rejection of accurate estimates by clients who then imposed unachievable schedules.

The author and his colleagues have examined the results of about 13,000 software projects between 1973 and 2010.

TABLE 1 Software project outcomes by size of project

Probability of Successful Outcomes					
	Early	On-Time	Delayed	Canceled	Sum
1 FP	14.68%	83.16%	1.92%	0.25%	100.00%
10 FP	11.08%	81.25%	5.67%	2.00%	100.00%
100 FP	6.06%	74.77%	11.83%	7.33%	100.00%
1,000 FP	1.24%	60.76%	17.67%	20.33%	100.00%
10,000 FP	0.14%	28.03%	23.83%	48.00%	100.00%
100,000 FP	0.00%	13.67%	21.33%	65.00%	100.00%
Average	5.53%	56.94%	13.71%	23.82%	100.00%

Table 1 shows the percentage of projects that are on time, early, late, or canceled for six size plateaus, each an order of magnitude apart.

Table 1 uses function point metrics for expressing software size. (For additional information on function point metrics, it is useful to visit the International Function Point Users Group (IFPUG) website at http://www.ifpug.org).

The reasons for using function points are that with more than 2,500 programming languages in existence, and multiple languages used concurrently in thousands of applications, lines of code (LOC) are not suitable for large-scale economic studies. Also, defects in requirements and design outnumber code defects and cannot be measured with LOC metrics. As can be seen from Table 1, small software projects are usually successful, but large systems are not. Why not?

One reason is that optimistic estimates are endemic. Also endemic are refusals by clients to accept accurate estimates, combined with the substitution of unachievable dates by client executives. (A separate study by the author noted that optimism in software estimates increased with application size.)

A second reason is that the measured rate of requirements growth is more than 1 percent per calendar month from the end of the requirements phase until the start of testing. This information comes from counting function points at requirements completion and then again at delivery. For large systems with schedules of 36 months or more, the total volume of creeping requirements can top 25 percent compared to the original requirements. (In one lawsuit there were 84 major requirements changes, which almost doubled the size of the application compared to the initial requirements.)

A third reason for failure is the unfortunate tendency for project managers to omit serious problems in their status reports to clients and executives. If problems are not recognized and dealt with early, it is often impossible to recover later.

The fourth and most serious reason is that the most expensive and time-consuming activity for large software development projects is the cost of finding and fixing bugs. Testing alone is not very efficient in finding many kinds of defects, and testing is very expensive.

If too many unanticipated bugs are present when testing begins, then test schedules will stretch out and delay the final delivery of the application. Test costs will rise until they degrade the project's return on investment to negative levels. Defect prevention, combined with pre-test defect removal such as inspections and static analysis, can reduce and eliminate delays and overruns due to poor quality.

Overall, what seems to be the chief reason for the failure of large software projects is poor quality. The phrase "poor quality" in this context has two meanings:

1) Excessive numbers of defects or bugs (fewer than 6.0 per function point).
2) Inadequate defect removal activities (more than 85 percent defect removal efficiency).

To know what volume of defects might be "excessive" and what level of defect removal is "inadequate," it is useful to know current U.S. averages. Table 2 shows defects originating in requirements, designs, code, user documents, and "bad fixes" or secondary defects. Table 2 shows the average number of defects found on software projects, and the percentage of defects removed prior to delivery to customers.

TABLE 2 Defect removal efficiency by origin of defects circa 2011 (data expressed in terms of defects per function point)

Defect Origins	Defect potentials	Removal efficiency	Delivered defects
Requirements	1.00	77%	0.23
Design	1.25	85%	0.19
Coding	1.75	95%	0.09
Document	0.60	80%	0.12
Bad fixes	0.40	70%	0.12
Total	*5.00*	*85%*	*0.75*

Table 2 illustrates three unfortunate aspects of average software projects: 1) large numbers of defects likely to occur; 2) defect removal efficiency is not very good and especially so for noncode defects; and 3) defects originate in multiple sources.

However, when examining the results of software projects developed by leading companies that have implemented successful process improvement programs, it can be seen that total defect volumes are lower than average, while defect removal efficiency levels are better than average.

Table 3 illustrates typical results for defect potentials and defect removal levels based on the capability maturity model (CMMI) developed by the Software Engineering Institute (SEI).

As can be seen, levels 3, 4, and 5 are significantly better than U.S. averages in terms of both overall volumes of defects and defect removal efficiency levels. One of the main benefits of achieving the higher CMMI levels is better quality control, which pays off in more predictable project outcomes. This raises interesting questions as to exactly what kinds of process improvements benefit defect volumes and defect removal efficiency.

Although statistically higher CMMI levels benefit quality, the author's study noted overlap among levels. The quality of the best CMMI level 1 groups was higher than the worst at level 3. Some projects by level 5 groups have had quality problems.

There are methods of quality improvement outside of the Capability Maturity Model. These include the Rational Unified Process, Watts Humphrey's Team Software Process, and several forms of agile development such as extreme programming and Crystal development.

REDUCING DEFECT VOLUMES

Since the number of defects found in requirements and designs outnumbers coding defects, leading companies and leading projects are very thorough in gathering requirements and in producing specifications. Of course, reducing coding defects and "bad fixes" is important too.

Some of the methods noted that reduce requirements and design defects include:

- A joint client/development change control board or designated domain experts.
- Use of quality function deployment (QFD).

TABLE 3 Software quality and the SEI Capability Maturity Model (CMMI) for projects of 5,000 function points in size

CMMI Level	Defect potential per function point	Defect removal efficiency	Delivered defects per function point
SEI CMMI 1	5.50	73.00%	1.49
SEI CMMI 2	4.00	90.00%	0.40
SEI CMMI 3	3.00	95.00%	0.15
SEI CMMI 4	2.50	97.00%	0.08
SEI CMMI 5	2.25	98.00%	0.05

- Quality assurance review of architecture and design documents.
- Use of Six Sigma for software and/or Lean Six Sigma.
- Use of the Japanese methods such as kaizen, kanban, and poka yoke.
- Use of formal requirements and design inspections.
- Use of joint application design (JAD) to minimize downstream changes.
- Use of the practices found in the higher levels of the CMMI.
- Use of formal prototypes to minimize downstream changes.
- Use of certified reusable components.
- Use of data mining to extract requirements and designs from legacy software.
- Formal reviews of all change requests.
- Revised cost and schedule estimates for all changes greater than 10 function points.
- Prioritization of change requests in terms of business impact.
- Formal assignment of change requests to specific releases.
- Use of automated change control tools with cross-reference capabilities.

Some methods of defect prevention are only partially successful. For example, the clean-room development method is effective when requirements are stable, but is not effective when requirements growth tops 1 percent per calendar month.

The agile approach of embedding a user in a development team to provide requirements works when there are only a small number of total users, but is ineffective for applications with thousands or millions of users. No one person understands all of the requirements for such applications. Focus groups and market surveys are needed to capture potential requirements for high-usage applications such as operating systems, telephone switching systems, automotive control systems, and the like.

Note that formal inspections are effective in three distinct ways. Obviously, inspections are effective in terms of defect removal efficiency. However, participants in formal inspections spontaneously avoid making the same kinds of mistakes that inspections find. This phenomenon was initially noted by IBM in the 1970s when inspections were first being studied, and has been validated in many companies. As a result, formal inspections are among the most effective methods of defect prevention. A final benefit is that formal inspections provide better source documents for test case design, so testing efficiency is raised by 5 percent or more as a byproduct of using inspections.

Without use of inspections, U.S. quality averages approximately 5.0 bugs per function point and about 85 percent in terms of defect removal efficiency. When formal inspections are added to the cycle, defect potentials gradually drop below 3.0 per function point, while defect removal efficiency levels routinely top 95 percent and may hit 99 percent. This combination yields shorter development schedules and lower development costs than testing alone.

Automated static analysis is a fairly new form of pre-test defect removal that also has benefits in terms of both defect prevention and defect removal. The caveat with static analysis is that there are more than 2,500 programming languages in use circa 2011. Static analysis tools only work for perhaps 25 of the most common languages, such as C, C+, C#, Java, COBOL, SQL, and a small number of others.

One interesting aspect of controlling requirements is a reduction in unplanned changes or "requirements creep." Ordinary U.S. projects average between 1 percent and 2 percent per month in new and changing requirements.

Leading projects where the requirements are carefully gathered and analyzed average only a fraction of 1 percent per month in unplanned changes. JAD, prototypes, QFD, and requirements inspections are all effective in reducing unplanned requirements creep.

It happens that creeping requirements tend to contain more bugs than original requirements. Testing defect removal efficiency is also lower against creeping requirements. Therefore, both static analysis and formal inspections are key process tools to minimize the damages that often occur from poor quality control of creeping requirements.

RAISING DEFECT REMOVAL EFFICIENCY LEVELS

Most forms of testing are less than 35 percent efficient in finding bugs or defects. However, formal design and code inspections are more than 65 percent efficient in finding bugs or defects and sometimes top 85 percent.

Static analysis is also high in efficiency against many kinds of coding defects. Therefore, all leading projects in leading companies use formal inspections, static analysis, and formal testing. This combination is the only known way of achieving cumulative defect removal levels higher than 95 percent. Table 4 illustrates the measured ranges of defect removal efficiency levels for a variety of reviews, inspections, static analysis, and several kinds of test stages. Note that Table 4 is an excerpt from a forthcoming book, *The Economics of Software Quality* (Jones and Subrmanyam 2011).

The low defect removal efficiency levels of most forms of testing explain why the best projects do not rely upon testing alone. The best projects use formal requirements, architecture, and design inspections first, and then static analysis of code, code inspections for key features, and a multistage testing sequence afterward. This combination of inspections followed by static analysis and testing leads to the shortest overall development schedules, and lowers the probabilities of project failures.

MEASURING THE ECONOMIC VALUE OF SOFTWARE QUALITY

For more than 50 years, the economic value of software quality has been poorly understood due to inadequate metrics and measurement practices. The two most common software metrics in the early days of software were LOC and "cost per defect." Unfortunately, both of these have serious economic flaws.

The LOC metric cannot be used to measure either requirements or design defects, which collectively outnumber coding defects. It is not possible to understand the real economic value of quality if more than 50 percent of all defects are not included in the measurements.

TABLE 4 Pre-test and test defect removal efficiency ranges

Pre-test Defect Removal	Minimum	Average	Maximum
Formal design inspections	65.00%	87.00%	97.00%
Formal code inspections	60.00%	85.00%	96.00%
Static analysis	65.00%	85.00%	95.00%
Formal requirement inspections	50.00%	78.00%	90.00%
Pair programming	40.00%	55.00%	65.00%
Informal peer reviews	35.00%	50.00%	60.00%
Desk checking	25.00%	45.00%	55.00%
Average	*48.57%*	*69.29%*	*79.71%*
Test Defect Removal	**Minimum**	**Average**	**Maximum**
Experiment-based testing	60.00%	75.00%	85.00%
Risk-based testing	55.00%	70.00%	80.00%
Security testing	50.00%	65.00%	80.00%
Subroutine testing	27.00%	45.00%	60.00%
System testing	27.00%	42.00%	55.00%
External beta testing	30.00%	40.00%	50.00%
Performance testing	30.00%	40.00%	45.00%
Supply-chain testing	20.00%	40.00%	47.00%
Cloud testing	25.00%	40.00%	55.00%
Function testing	33.00%	40.00%	55.00%
Unit testing (automated)	20.00%	40.00%	50.00%
Unit testing (manual)	15.00%	38.00%	50.00%
Regression testing	35.00%	35.00%	45.00%
Independent verification	20.00%	35.00%	47.00%
Clean-room testing	20.00%	35.00%	50.00%
Acceptance testing	15.00%	35.00%	40.00%
Independent testing	15.00%	35.00%	42.00%
Average	*29.24%*	*44.12%*	*55.06%*

TABLE 5 The top 15 U.S. software cost drivers in rank order circa 2011

1. The cost of finding and fixing bugs.
2. The cost of canceled projects.
3. The cost of producing paper documents and English words.
4. The cost of recovery from security flaws and attacks.
5. The cost of requirements changes during development.
6. The cost of programming or coding.
7. The cost of customer support.
8. The cost of meetings and communication.
9. The cost of project management.
10. The cost of application renovation.
11. The cost of innovation and new kinds of features.
12. The cost of litigation for canceled projects.
13. The cost of training and learning software applications.
14. The cost of avoiding security flaws.
15. The cost of acquiring reusable components.

TABLE 6 Potential top 15 U.S. software cost drivers in rank order circa 2025

1. The cost of innovation and new kinds of features.
2. The cost of acquiring reusable components.
3. The cost of requirements changes during development.
4. The cost of programming or coding.
5. The cost of training and learning software applications.
6. The cost of avoiding security flaws.
7. The cost of producing paper documents and English words.
8. The cost of customer support.
9. The cost of meetings and communication.
10. The cost of project management.
11. The cost of application renovation.
12. The cost of litigation for canceled projects.
13. The cost of finding and fixing bugs.
14. The cost of recovery from security flaws and attacks.
15. The cost of canceled projects.

A more subtle problem with LOC is that this metric penalizes high-level languages such as Java and Ruby and makes older low-level languages such as C and assembly language look better than they really are.

For example, if an application is coded in 1,000 lines of assembly code that contains 10 defects, the result is obviously 10 defects per KLOC. If the same application is coded in 250 lines of Java that contains only three defects, the results are now up to 12 defects per KLOC even though total defects are reduced by 70 percent. The LOC metric ignores the substantial decrease in absolute defect volumes.

If both versions are five function points in size, then the assembly version has 2.0 coding defects per function point, while the Java version has only 0.6 coding defects per function point.

The "cost per defect" metric actually penalizes quality and tends to achieve the lowest result for the buggiest applications. This phenomenon is due to fixed costs associated with defect removal, such as the cost of writing test cases and the cost of executing test cases. Even in situations where the application has zero defects, there will still be costs for writing and executing test cases.

The most effective method for measuring the economic value of quality is to analyze the total cost of ownership (TCO) for software applications. It will be discovered that applications with fewer than about 3.0 defects per function point and greater than 95 percent

in defect removal efficiency will cost about 20 percent less to develop than identical projects with poor quality.

The high-quality project schedules will be shorter by about 15 percent. Annual maintenance costs will be lower by about 40 percent. The cumulative TCO of high-quality applications from the start of the first release through five years of maintenance and enhancement will be about 30 percent lower than identical projects with poor quality.

One final value point is very important. For large applications, high quality levels will minimize the chances of failure. This is because high-quality applications tend to have quicker testing schedules and hence quicker overall schedules. The economic value of excellent quality is directly proportional to application size. The larger the software application, the more valuable quality becomes.

As of 2011, the overall cost drivers for software indicate why software has a bad reputation among CEOs and corporate executives. The two top cost drivers are finding and fixing bugs and canceled projects (see Table 5). It is no wonder software is poorly regarded by corporate executives.

No true engineering discipline should have defect repairs and canceled projects as the two top cost drivers. For software engineering to become a true engineering discipline, quality control will have to be much better than it is in 2011.

Table 6 shows a hypothetical rearrangement of cost drivers that should be a goal for software engineers over perhaps the next 15 years. The top cost driver should be innovation and designing new features, not bug repairs. Table 6 illustrates how costs should be apportioned circa 2025 assuming serious attempts at better quality control.

Additional Readings

Ewusi-Mensah, Kweku. 2003. *Software development failures.* Cambridge, MA: MIT Press.

Galorath, Dan. 2006. *Software sizing, estimating, and risk management: When performance is measured performance improves.* Philadelphia: Auerbach Publishing.

Garmus, David, and David Herron. 2001. *Function point analysis—Measurement practices for successful software projects.* Boston: Addison Wesley Longman.

Gilb, Tom, and Dorothy Graham. 1993. *Software inspections.* Reading, MA: Addison.

Glass, R. L. 1998. *Software runaways: Lessons learned from massive software project failures.* Englewood Cliffs, NJ: Prentice Hall.

Johnson, James et al. 2000. *The chaos report.* West Yarmouth, MA: The Standish Group.

Jones, Capers. 2010. *Software engineering best practices.* New York: McGraw Hill.

Jones, Capers. 2008. *Applied software measurement,* Third Edition. New York: McGraw Hill.

Jones, Capers. 2007. *Estimating software costs.* New York: McGraw Hill.

Jones, Capers. 2000. *Software assessments, benchmarks, and best practices.* Boston: Addison Wesley Longman.

Jones, Capers. 1998. Sizing up software. *Scientific American Magazine* 279, no. 6, (December).

Jones, Capers. 1997. *Software quality—Analysis and guidelines for success.* Boston: International Thomson Computer Press.

Jones, Capers. 1994. *Assessment and control of software risks.* Englewood Cliffs, NJ: Prentice Hall.

Jones, Capers. 1995. *Patterns of software system failure and success.* Boston: International Thomson Computer Press.

Kan, Stephen H. 2003. *Metrics and models in software quality engineering,* Second Edition. Boston: Addison Wesley Longman.

Pressman, Roger. 2005. *Software engineering—A practitioner's approach.* New York: McGraw Hill.

Radice, Ronald, A. 2002. *High quality low cost software inspections.* Andover, MA: Paradoxicon Publishing.

Wiegers, Karl E. 2002. *Peer reviews in software—A practical guide.* Boston: Addison Wesley Longman.

Yourdon, Ed. 1997. *Death march—The complete software developer's guide to surviving "Mission Impossible" projects.* Upper Saddle River, NJ: Prentice Hall PTR.

Yourdon, Ed. 2005. *Outsource: Competing in the global productivity race.* Upper Saddle River, NJ: Prentice Hall PTR.

If software quality is improved, it should be possible to spend a much higher percentage of available funds on innovation, new features, and certified reusable materials. Today's top cost drivers of defect repairs and canceled projects should be at the bottom of the list of cost drivers and not at the top, as they are in 2011.

SUMMARY AND CONCLUSIONS

The phrase "software process improvement" is somewhat ambiguous. The phrase by itself does not indicate what needs to be improved. However, from analysis of a large number of projects that were either failures on one hand, or quite successful on the other, it is obvious that quality control is the top-ranked issue that needs to be improved. With state-of-the-art quality control, successful projects become the norm. With inadequate defect prevention and defect removal, canceled projects and disasters are the norm.

An occupation where failures and disasters are the top cost drivers is not a true engineering discipline. To become a true engineering discipline, software engineering needs better quality control, better quality measures, and better economic analysis than current norms.

REFERENCES

International Function Point Users Group (IFPUG). 2002. IT measurement—Practical advice from the experts. Boston: Addison Wesley Longman.

Jones, Capers, and Jitendra Subrmanyam. 2011. The economics of software quality. Englewood Cliffs, NJ: Prentice Hall.

BIOGRAPHY

Capers Jones is currently the president of Capers Jones & Associates LLC. He is also the founder and former chairman of Software Productivity Research LLC (SPR). He holds the title of chief scientist emeritus at SPR. Jones founded SPR in 1984.

Before founding SPR, Jones was assistant director of programming technology for the ITT Corporation at the Programming Technology Center in Stratford, CT. He was also a manager and software researcher at IBM where he designed IBM's first software cost estimating tool in 1973.

Jones is a well-known author and international public speaker. Some of his books have been translated into five languages. His two most recent books are *Software Engineering Best Practices* (McGraw Hill, 2010) and *The Economics of Software Quality* (Addison Wesley, summer of 2011). He can be reached at cjoneslll@cs.com.

Last Call for Papers!
The Fifth World Congress for Software Quality (5WCSQ)

The date is set! **The Fifth World Congress for Software Quality (5WCSQ)** will be held October 31–November 4, 2011, in Shanghai, China. Shanghai won the bid to host the World Expo in 2010 and is aiming to construct itself into the center of international economy, finance, trade, and shipping. The conference will be held at the New Expo Centre, Shanghai, China. The deadline to submit your papers is fast approaching. Patricia McQuaid will once again be the chairperson for North, South, and Central America. Contact her at pmcquaid@calpoly.edu with any questions, and for the final date for submissions. Please put "5WCSQ" in the message heading.

The event is sponsored by the following organizations:

- ASQ Software Division
- European Organization for Quality (EOQ)
- Union of Japanese Scientists and Engineers (JUSE)
- For China: Shanghai Economic and Informatization Commission (SHEITC), Shanghai Software Industry Association (SSIA), and the China Software Industry Association (CSIA)

The Agenda

October 31	Tutorial Session/Welcome Session
November 1	Opening Ceremony/Plenary Session/ Keynote Speech/Presentations
November 2	Presentations/Panel Discussion/Special Speech/Banquet
November 3	Presentations/Special Speech/Closing Session
November 4	Industrial Visit (Shanghai Pudong Software Park)

History

The World Congress for Software Quality is an international conference sponsored by the ASQ Software Division, JUSE, and the EOQ Software Group.

1995: 1WCSQ—ASQ Software Division, San Francisco, CA

2000: 2WCSQ—JUSE, Yokohama, Japan

2005: 3WCSQ—EOQ Software Group, Munich, Germany

2008: 4WCSQ—ASQ Software Division, Washington, DC, area