Journal of Cost Analysis and Parametrics, 7:41–47, 2014
Copyright © ICEAA
ISSN: 1941-658X print / 2160-4746 online
DOI: 10.1080/1941658X.2014.890493

Software Industry Goals for the Years 2014 through 2018

CAPERS JONES

Namcook Analytics LLC, Hingham, Massachusetts

This article discusses 20 quantitative targets for software engineering projects that are technically feasible to be achieved within a five year window. Some leading companies have already achieved many of these targets, but average and lagging companies have achieved only a few, if any. Software needs firm achievable goals expressed in quantitative fashion. For example the first goal is to remove 99.5% of bugs or defects before delivery rather than today's average of below 90%. Inspections and static analysis prior to testing can achieve this goal for essentially every project.

Introduction

The following is a collection of 20 goals or targets for software engineering progress developed by Namcook Analytics LLC for the five years between 2014 and 2018. All of these goals are achievable in 2014 and, in fact, have already been achieved by a small selection of leading companies.

Unfortunately, less than 5% of U.S. and global companies have achieved any of these goals, and less than 1% have achieved most of them. None of the author's clients have achieved every goal.

The author suggests that every major software-producing company and government agency have their own set of five-year targets, using the current list as a starting point.

Twenty Software Engineering Targets from 2014 through 2018

1. **Raise defect removal efficiency (DRE) from <90.0% to >99.5%.** This is the most important goal for the industry. It cannot be achieved by testing alone but requires pre-test inspections and static analysis. DRE is measured by comparing all bugs found during development to those reported in the first 90 days by customers.
2. **Lower software defect potentials from >4.0 per function point to <2.0 per function point.** Defect potentials are the sum of bugs found in requirements, design, code, user documents, and bad fixes. Requirements and design bugs often outnumber code bugs. Achieving this goal requires effective defect prevention, such as joint application design (JAD), quality function deployment (QFD), certified reusable components, and others. It also requires a complete software quality measurement program. Achieving this goal also requires better training in common sources of defects found in requirements, design, and source code.

Address correspondence to Capers Jones, 95 Old Boston Neck Road, Narragansett, RI 02882. E-mail: Capers.Jones3@gmail.com

3. **Lower cost of quality (COQ) from >45.0% of development to <20.0% of development.** Finding and fixing bugs has been the most expensive task in software for more than 50 years. A synergistic combination of defect prevention and pre-test inspections and static analysis are needed to achieve this goal.
4. **Reduce average cyclomatic complexity from >25.0 to <10.0.** Achieving this goal requires careful analysis of software structures, and of course it also requires measuring cyclomatic complexity for all modules.
5. **Raise test coverage from <75.0% to >98.5% for risks, paths, and requirements.** Achieving this goal requires using mathematical design methods for test case creation, such as using design of experiments. It also requires measurement of test coverage.
6. **Eliminate error-prone modules in large systems.** Bugs are not randomly distributed. Achieving this goal requires careful measurements of code defects during development and after release with tools that can trace bugs to specific modules. Some companies, such as IBM, have been doing this for many years. Error-prone modules (EPM) are usually less than 5% of total modules but receive more than 50% of total bugs. Prevention is the best solution. Existing error-prone modules in legacy applications may require surgical removal and replacement.
7. **Eliminate security flaws in all software applications.** As cyber crime becomes more common, the need for better security is more urgent. Achieving this goal requires use of security inspections, security testing, and automated tools that seek out security flaws. For major systems containing valuable financial or confidential data, ethical hackers may also be needed.
8. **Reduce the odds of cyber attacks from >10.0% to <0.1%.** Achieving this goal requires a synergistic combination of better firewalls, continuous anti-virus checking with constant updates to viral signatures; and also increasing the immunity of software itself by means of changes to basic architecture and permission strategies.
9. **Reduce bad-fix injections from >7.0% to <1.0%.** Not many people know that about 7% of attempts to fix software bugs contain new bugs in the fixes themselves commonly called "bad fixes." When cyclomatic complexity tops 50, the bad-fix injection rate can soar to 25% or more. Reducing bad-fix injection requires measuring and controlling cyclomatic complexity, using static analysis for all bug fixes, testing all bug fixes, and inspections of all significant fixes prior to integration.
10. **Reduce requirements creep from >1.5% per calendar month to <0.25% per calendar month.** Requirements creep has been an endemic problem of the software industry for more than 50 years. While prototypes, agile embedded users, and joint application design (JAD) are useful, it is technically possible to also use automated requirements models to improve requirements completeness.
11. **Lower the risk of project failure or cancellation on large 10,000 function point projects from >35.0% to <5.0%.** Cancellation of large systems due to poor quality and cost overruns is an endemic problem of the software industry, and totally unnecessary. A synergistic combination of effective defect prevention and pre-test inspections and static analysis can come close to eliminating this far too common problem.
12. **Reduce the odds of schedule delays from >50.0% to <5.0%.** Since the main reasons for schedule delays are poor quality and excessive requirements creep, solving some of the earlier problems in this list will also solve the problem of schedule delays. Most projects seem on time until testing starts, when huge quantities of bugs begin to stretch out the test schedule to infinity. Defect prevention combined with pre-test static analysis can reduce or eliminate schedule delays.

13. **Reduce the odds of cost overruns from >40.0% to <3.0%.** Software cost overruns and software schedule delays have similar root causes; i.e., poor quality control combined with excessive requirements creep. Better defect prevention combined with pre-test defect removal can help to cure both of these endemic software problems.
14. **Reduce the odds of litigation on outsource contracts from >5.0% to <1.0%.** The author of this article has been an expert witness in 12 breach of contract cases. All of these cases seem to have similar root causes, which include poor quality control, poor change control, and very poor status tracking. A synergistic combination of early sizing and risk analysis prior to contract signing plus effective defect prevention and pre-test defect removal can lower the odds of software breach of contract litigation.
15. **Lower maintenance and warranty repair costs by >75.0% compared to 2014 values.** Starting in about 2000 the number of U.S. maintenance programmers began to exceed the number of development programmers. IBM discovered that effective defect prevention and pre-test defect removal reduced delivered defects to such low levels that maintenance costs were reduced by at least 45% and sometimes as much as 75%.
16. **Improve the volume of certified reusable materials from <15.0% to >75.0%.** Custom designs and manual coding are intrinsically error-prone and inefficient no matter what methodology is used. The best way of converting software engineering from a craft to a modern profession would be to construct applications from libraries of certified reusable material; i.e., reusable requirements, design, code, and test materials. Certification to near zero-defect levels is a precursor, so effective quality control is on the critical path to increasing the volumes of certified reusable materials.
17. **Improve average development productivity from <8.0 function points per month to >16.0 function points per month.** Productivity rates vary based on application size, complexity, team experience, methodologies, and several other factors. However, when all projects are viewed in aggregate, average productivity is below 8.0 function points per staff month. Doubling this rate needs a combination of better quality control and much higher volumes of certified reusable materials; probably 50% or more.
18. **Improve work hours per function point from >16.5 to <8.25.** Goal 17 and this goal are essentially the same but use different metrics. However, there is one important difference. Work hours per month will not be the same in every country. For example, a project in Sweden with 126 work hours per month will have the same number of work hours as a project in China with 184 work hours per month. But the Chinese project will need fewer calendar months than the Swedish project.
19. **Shorten average software development schedules by >35.0% compared to 2014 averages.** The most common complaint of software clients and corporate executives at the CIO and CFO level is that big software projects take too long. Surprisingly, it is not hard to make them shorter. A synergistic combination of better defect prevention, pre-test static analysis and inspections, and larger volumes of certified reusable materials can make significant reductions in schedule intervals.
20. **Raise maintenance assignment scopes from <1,500 function points to >5,000 function points.** The metric "maintenance assignment scope" refers to the number of function points that one maintenance programmer can keep up and running during a calendar year. The range is from <300 function points for buggy and complex software to >5,000 function points for modern software released with effective quality control. The current average is about 1,500 function

points. This is a key metric for predicting maintenance staffing for both individual projects and also for corporate portfolios. Achieving this goal requires effective defect prevention, effective pre-test defect removal, and effective testing using modern mathematically based test case design methods. It also requires low levels of cyclomatic complexity.

Note that the function point metrics used in this article refer to function points as defined by the International Function Point User's Group (IFPUG). Other function points, such as Common Software Measurement International Consortium (COSMIC), Finnish Software Measurement Association (FISMA), Netherlands Software Measurement Association (NESMA), unadjusted, etc., can also be used but would have different quantitative results.

The technology stack available in 2014 is already good enough to achieve each of these 20 targets, although few companies have done so. Some of the technologies associated with achieving these 20 targets include but are not limited to the following.

Technologies Useful in Achieving Software Engineering Goals

- Use early risk analysis, sizing, and both quality and schedule/cost estimation before starting major projects, such as Namcook's Software Risk Master (SRM).
- Use effective defect prevention, such as Joint Application Design (JAD) and Quality Function Deployment (QFD).
- Use pre-test inspections of major deliverables, such as requirements, architecture, design, code, etc.
- Use both text static analysis and source code static analysis for all software.
- Use the Systems Administration, Audit, Network, and Security (SANS) Institute list of common programming bugs and avoid them all.
- Use the Gunning FOG index and FLESCH-KINKAID readability score tools on requirements, design, etc.
- Use mathematical test case design, such as design of experiments.
- Use certified test and quality assurance personnel.
- Use function point metrics for benchmarks and normalization of data.
- Use effective methodologies, such as agile and Extreme Programming (XP) for small projects; Rational Unified Process (RUP) and Team Software Process (TSP) for large systems.
- Use automated test coverage tools.
- Use automated cyclomatic complexity tools.
- Use parametric estimation tools that can predict quality, schedules, and costs. Manual estimates tend to be excessively optimistic.
- Use accurate measurement tools and methods with at least 3.0% precision.
- Consider applying automated requirements models, which seem to be effective in minimizing requirements issues.
- Consider applying the new SEMAT method (Software Engineering Methods and Theory), which holds promise for improved design and code quality. SEMAT comes with a learning curve so reading the published book is necessary prior to use.

It is past time to change software engineering from a craft to a true engineering profession. It is also past time to switch from partial and inaccurate analysis of software results to results with high accuracy for both predictions before projects start and measurements after projects are completed.

The 20 goals shown above were positive targets that companies and government groups should strive to achieve. But "software engineering" also has a number of harmful practices that should be avoided and eliminated. Some of these are bad enough to be viewed as professional malpractice. Following are six hazardous software methods, some of which have been in continuous use for more than 50 years without their harm being fully understood.

Six Hazardous Software Engineering Methods to be Avoided

1. **Stop trying to measure quality economics with "*cost per defect*."** This metric always achieves the lowest value for the buggiest software, so it penalizes actual quality. The metric also understates the true economic value of software by several hundred percent. This metric violates standard economic assumptions and can be viewed as professional malpractice for measuring quality economics. The best economic measure for cost of quality is "defect removal costs per function point." Cost per defect ignores the fixed costs for writing and running test cases. It is a well-known law of manufacturing economics that if a process has a high proportion of fixed costs the cost per unit will go up. The urban legend that it costs 100 times as much to fix a bug after release than before is not valid; the costs are almost flat if measured properly.
2. **Stop trying to measure software productivity with "*lines of code*" (LOC) metrics.** This metric penalizes high level languages. This metric also makes non-coding work, such as requirements and design, invisible. This metric can be viewed as professional malpractice for economic analysis involving multiple programming languages. The best metrics for software productivity are work hours per function point and function points per staff month. Both of these can be used at activity levels and also for entire projects. These metrics can also be used for non-code work, such as requirements and design. LOC metrics have limited use for coding itself, but are hazardous for larger economic studies of full projects. LOC metrics ignore the costs of requirements, design, and documentation, which are often larger than the costs of the code itself.
3. **Stop measuring "design, code, and unit test" or DCUT.** Measure full projects, including management, requirements, design, coding, integration, documentations, all forms of testing, etc. DCUT measures encompass less than 30% of the total costs of software development projects. It is professionally embarrassing to measure only part of software development projects.
4. **Be cautious of "*technical debt*."** This is a useful metaphor but not a complete metric for understanding quality economics. Technical debt omits the high costs of canceled projects and it excludes both consequential damages to clients and also litigation costs and possible damage awards to plaintiffs. Technical debt only includes about 17% of the true costs of poor quality. Cost of quality (COQ) is a better metric for quality economics.
5. **Avoid "*pair programming*".** Pair programming is expensive and less effective for quality than a combination of inspections and static analysis. Do read the literature on pair programming, and especially the reports by programmers who quit jobs specifically to avoid pair programming. The literature in favor of pair programming also illustrates the general weakness of software engineering research, in that it does not compare pair programming to methods with proven quality results, such as inspections and static analysis. It only compares pairs to single programmers without any discussion of tools, methods, inspections, etc.
6. **Stop depending only on testing** without using effective methods of defect prevention and effective methods of pre-test defect removal, such as inspections and static analysis. Testing by itself without pre-test removal is expensive and seldom

tops 85% in defect removal efficiency levels. A synergistic combination of defect prevention, pre-test removal, such as static analysis and inspections, can raise DRE to >99% while lowering costs and shortening schedules at the same time.

The software engineering field has been very different from older and more mature forms of engineering. One of the main differences between software engineering and true engineering fields is that software engineering has very poor measurement practices and far too much subjective information instead of solid empirical data.

This short article suggests a set of 20 quantified targets that if achieved would make significant advances in both software quality and software productivity. But the essential message is that poor software quality is a critical factor that needs to get better in order to improve software productivity, schedules, costs, and economics.

References

Beck, K. (2002). *Test-Driven Development*. Boston, MA: Addison Wesley (240 pp.). ISBN 10: 0321146530.

Black, R. (2009). *Managing the Testing Process: Practical Tools and Techniques for Managing Hardware and Software Testing*. Hoboken, NJ: Wiley (672 pp.). ISBN 10: 0470404159.

Chess, B., & West, J. (2007). *Secure Programming with Static Analysis*. Boston, MA: Addison Wesley (624 pp.). ISBN 13: 978-0321424778.

Cohen, L. (1995). *Quality Function Deployment—How to Make QFD Work for You*. Upper Saddle River, NJ: Prentice Hall (368 pp.). ISBN 10: 0201633302.

Crosby, P. B. (1979). *Quality is Free*. New York, NY: New American Library, Mentor Books (270 pp.).

Everett, G. D., & McLeod, R. (2007). *Software Testing*. Hoboken, NJ: John Wiley & Sons (261 pp.). ISBN 978-0-471-79371-7.

Gack, G. (2010). *Managing the Black Hole: The Executives Guide to Software Project Risk*. Thomson, GA: Business Expert Publishing. ISBN 10: 1-935602-01-9.

Gack, G. (2009). *Applying Six Sigma to Software Implementation Projects*. Retrieved from http://software.isixsigma.com/library/content/c040915b.asp

Gilb, T., & Graham, D. (1993). *Software Inspections*. Reading, MA: Addison Wesley. ISBN 10: 0201631814.

Hallowell, D. L. (2006). *Six Sigma Software Metrics, Part 1*. Retrieved from http://software.isixsigma.com/library/content/03910a.asp

IFPUG. (2012). *The IFPUG Guide to IT and Software Measurement*. Auerbach Publishers.

International Organization for Standards. (1987). ISO 9000/ISO 14000. http://www.iso.org/iso/en/iso9000-14000/index.html

Jacobsen, I., Ng, P.-W., McMahon, P., Spence, I., & Lidman, S. (2013). *The Essence of Software Engineering: Applying the SEMAT Kernel*. Reading, MA: Addison Wesley.

Jones, C. (1978). Measuring Programming Quality and Productivity. *IBM Systems Journal*, 17(1),39–63.

Jones, C. (1981). *Programming Productivity—Issues for the Eighties* (First edition: 1981; Second edition: 1986). Los Alamitos, CA: IEEE Computer Society Press (489 pp.). ISBN 0-8186–0681-9.

Jones, C. (1988). *A Ten-Year Retrospective of the ITT Programming Technology Center*. Burlington, MA: Software Productivity Research.

Jones, C. (1993a). *Critical Problems in Software Measurement*. Information Systems Management Group (195 pp.). ISBN 1-56909-000-9.

Jones, C. (1993b). *Software Productivity and Quality Today—The Worldwide Perspective*. Los Alamitos, CA: Information Systems Management Group (200 pp.). ISBN 1-156909-001-7.

Jones, C. (1994). *Assessment and Control of Software Risks*. Englewood Cliffs, NJ: Prentice Hall (711 pp.). ISBN 0-13-741406-4.

Jones, C. (1995a). *New Directions in Software Management*. Los Alamitos, CA: Information Systems Management Group (150 pp.). ISBN 1-56909-009-2.

Jones, C. (1995b). *Patterns of Software System Failure and Success*. Boston, MA: International Thomson Computer Press (292 pp.). ISBN 1-850-32804-8.

Jones, C. (1997). *Software Quality—Analysis and Guidelines for Success*. Boston, MA: International Thomson Computer Press (492 pp.). ISBN 1-85032-876-6.

Jones, C. (2004). Software Project Management Practices: Failure versus Success. *Crosstalk*, October, 5–9.

Jones, C. (2005). Software Estimating Methods for Large Projects. *Crosstalk*, April, 8–12.

Jones, C. (2007). *Estimating Software Costs* (2nd Edition). New York, NY: McGraw Hill (700 pp.).

Jones, C. (2008). *Applied Software Measurement* (3rd Edition). New York, NY: McGraw Hill (662 pp.). ISBN 978=0-07-150244-3.

Jones, C. (2010). *Software Engineering Best Practices*. New York, NY: McGraw Hill (660 pp.). ISBN 978-0-07-162161-8.

Jones, C. (2014a). *A Short History of Lines of Code Metrics*. Narragansett, RI: Namcook Analytics LLC.

Jones, C. (2014b). *A Short History of the Cost per Defect Metric*. Narragansett, RI: Namcook Analytics LLC.

Jones, C. (2014c). *The Technical and Social History of Software Engineering*. Boston, MA: Addison Wesley Longman.

Jones, C. (2014d). *The Economics of Object-Oriented Software*. Narragansett, RI: Namcook Analytics.

Jones, C., & Bonsignour, O. (2011). *The Economics of Software Quality*. Boston, MA: Addison Wesley (587 pp.). ISBN 978-0-13-258220-9.

Kan, S. H. (2003). *Metrics and Models in Software Quality Engineering* (2nd Edition). Boston, MA: Addison Wesley Longman (528 pp.). ISBN 0-201-72915-6.

Land, S. K., Smith, D. B., & Walz, J. Z. (2008). *Practical Support for Lean Six Sigma Software Process Definition: Using IEEE Software Engineering Standards*. Wiley Blackwell (312 pp.). ISBN 10: 0470170808.

Mosley, D. J. (1993). *The Handbook of MIS Application Software Testing*. Englewood Cliffs, NJ: Yourdon Press, Prentice Hall (354 pp.). ISBN 0-13-907007-9.

Myers, G. (1979). *The Art of Software Testing*. New York, NY: John Wiley & Sons (177 pp.). ISBN 0-471-04328-1.

Nandyal, R. (2007). *Making Sense of Software Quality Assurance*. New Delhi, India: Tata McGraw Hill Publishing (350 pp.). ISBN 0-07-063378-9.

Radice, R. A. (2002). *High Quality Low Cost Software Inspections*. Andover, MA: Paradoxicon Publishing (479 pp.). ISBN 0-9645913-1-6.

Royce, W. E. (1998). *Software Project Management: A Unified Framework*. Reading, MA: Addison Wesley Longman. ISBN 0-201-30958-0.

Wiegers, K. E. (2002). *Peer Reviews in Software—A Practical Guide*. Boston, MA: Addison Wesley Longman (232 pp.). ISBN 0-201-73485-0.

About the Author

Capers Jones is Vice President and Chief Technology Officer of Namcook Analytics LLC. Namcook Analytics has offices in Hingham, MA and Narragansett, RI.

Capers Jones is the author of 15 books on software topics and one history book. His most recent software book is *The Technical and Social History of Software Engineering*, Addison Wesley, 2014. He has also published more than 100 journal articles.

He is co-founder of Namcook Analytics LLC. Prior to founding Namcook he was president of Capers Jones & Associates LLC. He was also founder and chairman of Software Productivity Research LLC. Prior to founding his own companies Capers Jones was assistant director of programming at the ITT corporation and also a manager and research specialist at IBM.