When applications crash due to a code quality issues, the common question is, “How could those experts have missed that?” The problem is, most people imagine software development as a room full of developers, keyboards clacking away with green, Matrix-esque code filling up the screen as they try and perfect the newest ground-breaking feature. However, in reality most of the work developers actually do is maintenance work fixing the bugs found in the production code to ensure a higher level of code quality.

Not only does this severely reduce the amount of business value IT can bring to the table, it also exponentially increases the cost in developing and maintaining quality applications. And even though the IT industry has seen this rise in cost happening for years, they’ve done little to stem the rising tide. The time has come to draw a line in the sand.

Capers Jones, VP and CTO of Namcook Analytics, recently released a collection of 20 goals software engineers should be aiming to reach by 2018 and we thought this was a great starting point to get software engineering focused on fixing the problems that lie before them, and not just spinning their gears.

However, having ambitious goals is only part of the challenge. In our experience, the organizations aren’t equipped to meet these goals because:

- Functional testing isn’t enough
- Code analyzers are myopic
- Productivity measurement is manual and laborious

Responsible IT managers need to change the way they think about software development and arm their teams with better tools and processes if they want to come close to achieving any of these goals. This starts with gaining better visibility into their software risk, performance measurement, portfolio analysis, and quality improvement -- and it needs to be instantaneous, not quarterly. The problems are happening now, in development, and management is wasting precious time and money waiting until testing to try and put it all together to work out all the kinks.

Once management has a transparent view into the code quality of their application portfolio, then they can shift their focus to achieving the software engineering goals outlined by Jones. They’re great goals to aspire to, but let’s make sure we’re not putting the cart before the horse.

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.

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 paper 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 will 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.

**Pete Pizzutillo** is Vice President at CAST and has spent the last 15 years working in the software industry. He passionately believes Software Intelligence is the cornerstone to successful digital transformation, and he actively helps customers realize the benefits of CAST's software analytics to ensure their IT systems are secure, resilient and efficient to support the next wave of modern business.