Software Quality, Reliability, and Error Prediction
How long can you wait for
CMMI® Compliance?

Manage your software and systems engineering projects
in guaranteed compliance with the CMMI NOW!

processMax 2i and 3i include all the necessary policies, procedures, guidelines, criteria,
templates, and forms in role-based, step-by-step instructions, ready for use—everything you
need for compliance with the CMMI-DEV at Level 2 and Level 3 respectively. Integrated with
robust document management, workflow, and automated measurement and reporting,
processMax is the intranet web-based solution for effective and efficient management of your
software and systems engineering projects. processMax enables you to start operating in
compliance with the CMMI immediately! We guarantee that organizations using processMax
will not fail a formal appraisal led by an independent SEI-licensed appraiser.

pragma SYSTEMS CORPORATION
www.pragmasystems.com
877.838.PMAX
info@pragmasystems.com

GSA Schedule Contract No. GS-35F-0559S; processMax is a registered trademark of pragma SYSTEMS CORPORATION.
Although processMax makes use of portions of “CMMI” for Development, Version 1.2, CMU/SEI-2006-TR-008, copyright 2006 by
Carnegie Mellon University, neither the Software Engineering Institute nor Carnegie Mellon University have reviewed or endorsed this product.

Copyright © 2008 pragma SYSTEMS CORPORATION.
Software quality is sometimes overlooked as a key objective in developing software; cost, schedule, and scope considerations prevail, and quality is treated as a “nice to have”, if even that. However, quality is usually the missing piece of the software puzzle! Stories of poor quality software, or software-related problems, abound both in the DoD and industry. According to Schwalbe [Schwalbe, 2010], quality should be a “fourth constraint” of equal importance to cost, schedule, and scope (or performance). Indeed, the Project Management Body of Knowledge [PMBOK, 2008] addresses quality as one of the four “core knowledge areas” of project management. (Cost, schedule, and scope are the other three.) In a related document, the Software Engineering Body of Knowledge (SWEBOK) Guide, software quality is not only one of the ten key knowledge areas, but also is “considered in many of the (other) knowledge areas” [SWEBOK 2004].

As explained by Webb and Patton, there are two categories of costs for software quality: the costs of good quality (sometimes called conformance costs), and the costs associated with poor quality (sometimes called non-conformance costs) [Webb and Patton 2008]. Figure 1 illustrates the costs associated with software quality; a brief description of each cost is now presented:

- Prevention Costs: These are the costs to prevent software and documentation errors, such as early prototyping, requirements analysis, and staff training.
- Appraisal Costs: These are the costs of searching for errors during the development process, such as design reviews, inspections, and black and glass box testing.
- Internal Failure Costs: These are the costs of coping with errors discovered during development and testing, including bug fixes, regression testing, and costs of late product shipments.
- External Failure Costs: These are the costs of coping with errors discovered after a product is released, errors found by customers or users, including technical support calls and complaints, interim bug fix releases, and warranty and reliability costs. These costs are especially insidious because the costs transcend the cost of error correction. Figure 2 illustrates the potential costs of external failures.

![Figure 1: Cost of Quality](image1)

Figure 1: Cost of Quality [Webb and Patton 2008]

It is generally agreed that the costs of good quality will more than offset the costs of poor quality. For instance, in his book, “The ROI from Software Quality”, El Emam provides many examples to show the positive return-on-investment form emphasizing prevention and appraisal costs [El Emam 2005]. Webb and Patton state that, if preventive quality measures and rework are deferred until the testing phase, the cost of change is 40 to 100 times greater than if the defect was fixed when it was created. In other words, spending money for software quality activities early in the software development process is certainly cost-effective for the program.

There are several ways to measure software quality throughout the software life cycle. Probably the most well-known measure is software reliability, which can be defined as “the probability that an item will function without failure for a specified period of time under certain conditions” [PSM 2003]. Software reliability is often indicated by error or failure metrics such as defects per thousand lines of code or defects per function point. Other quality measures include maintainability (time to correct an error or defect), efficiency, technical performance, fault-tolerance, and compliance to standards [PSM 2003].

One way to address software quality is to predict the number of errors or defects that will occur in a software program, then taking steps to reduce this number. There are several tools available for estimating defects, including some of the well-known cost models. While the accuracy of these tools cannot always be ascertained, they do show the relative effects of practices such as formal requirements and design
inspections, process maturity, and use of modern development methods in reducing errors and improving quality.

This issue of the Software Tech News contains several articles that can help the reader to gain a better understanding of software quality and reliability, and how to develop higher quality software. The first article, by David Herron, discusses best practices in software quality and related areas. The article shows that best practices sometimes do not “work” because they are not applied correctly. The article then discusses four best practices that can be used to improve software quality and software development in general.

The next article, by Capers Jones, shows that large software projects have had a history of poor reliability, or high numbers of defects. Capers Jones then shows how a software manager can take steps to reduce defects in these programs. He also discusses the economic value of software quality improvement practices.

The third article, by Donald M. Beckett and Doug Putnam, addresses software quality estimation, primarily using defects (or errors) per thousands of lines of code as a measure of quality and reliability. They show how defect discovery follows a Rayleigh Curve, which can be useful in defect or error prediction. They also show the effects of schedule compression and team size on software quality.

The fourth article, by Arlene Minkiewicz, discusses the effects of a newly-popular software development method, agile software development, on software quality. The impact of agile development may either positively or negatively affect software quality based on the particular agile method used. She shows that some specific agile methods can have a positive impact on software quality.

The fifth article, by Ray Madachy, Barry Boehm, and Dan Houston, discusses recent enhancements to the Constructive Quality Model (COQUALMO). COQUALMO is a well-researched model that is useful in prediction the number of defects, or errors, per thousands of lines of code or function points. It is a publically-available model which can be found on the web site: http://csse.usc.edu.

The final article, by John Robb, discusses the effects of computer programming languages on software safety and reliability. He presents a history of languages used for safety-critical applications such as airborne weapon systems, then shows how the C and C++ languages are being used successfully for the new F-35 aircraft.

References


About the Authors

Dan Ferens is currently the DACS Director and serves as an instructor for a 12-part series in software affordability which has been taught mainly to Air Force Research Laboratory (AFRL) scientists, engineers, and managers in Rome, NY. Dan retired from AFRL in early 2007 after more than 35 years of service to the Air Force as a military and civilian employee. Dan has been involved in software estimating since he became a civilian in 1978, both as an AFRL analyst and program manager, and as a Professor at Air Force Institute of Technology where he taught classes in software quality and reliability, and other software engineering and management topics, for 13 years. He is currently an Adjunct Instructor at SUNY Institute of Technology in Utica, New York where he teaches a class in information technology project management. He is a life member of the International Society of Parametric Analysts (ISPA), where, in 1999, he received the prestigious Freiman award for lifetime achievements in parametric estimating. He is also a member of Toastmasters, International where he holds the rank of Distinguished Toastmaster. Mr. Ferens has a Masters degree in Electrical Engineering from Rensselaer Polytechnic Institute, and a Masters Degree in Business
Administration from the University of Northern Colorado. He and his wife, Marcie, currently reside in Fulton, New York.

Ellen Walker, is the Editor of the Software Tech News and Deputy Director of the DACS. She is currently developing a series of publications on software “best practices” as part of the DACS Gold Practice Initiative. She has spent the past 25 years as a software developer in various roles spanning the entire software life cycle including project management of multiple business process re-engineering efforts within the DoD community. She is also experienced with assessment initiatives such as the Capability Maturity Model for Software (CMM-SW) and the quality management practices of the New York State Quality Award program. Ellen has an MS in Management Science (State University of New York (SUNY) at Binghamton), and bachelor degrees in both Computer Science (SUNY – Utica/Rome) and Mathematics (LeMoyne College).

Author Contact Information
Email: Daniel Ferens: daniel.ferens@itt.com
Email: Ellen Walker: ellen.walker@itt.com

The DACS is a central distribution hub for software technology information sources. The DACS offers a wide-variety of Technical Services designed to support the development, testing, validation, and transitioning of Software Engineering technology.

Visit us at https://www.thedacs.com/
Satisfy Your Quality Obsession

Software quality and reliability are mission critical. The size, pervasiveness, and complexity of today’s software can push your delivery dates and budgets to the edge. Streamline communication, improve traceability, achieve compliance, and deliver quality products with Seapine Software’s scalable, feature-rich application lifecycle management solutions:

- **TestTrack Pro** — Development workflow and issue management
- **TestTrack TCM** — Test case planning and tracking
- **Surround SCM** — Software configuration management
- **QA Wizard Pro** — Automated functional and regression testing

Designed for the most demanding software development and quality assurance environments, Seapine’s flexible cross-platform solutions adapt to the way your team works, delivering maximum productivity and saving you significant time and money.

GSA Schedule 70 Contract GS-35F-0168U

Visit [www.seapine.com/gsa](http://www.seapine.com/gsa)
We often read articles about software development best practices. Sometimes these articles entertain us with first-person stories about how a particular development practice or a software development tool helped to save the day on an important product release or systems implementation. Other times, these best practices articles are informing the reader about some tool, technique or process that warrants a ‘best practices’ label because it can improve your organizations productivity and quality. But what is truly a software development best practice?

I don't know of any industry standard or certification that is used to qualify something as a best practice; nor are there any rules or guidelines that help us to classify something as a best practice. So how do we know if something is a best practice? What gives a practice or a process that special distinction of being the ‘best’?

Let's look at three fairly standard quality related practices and processes that are typically referred to as a best practice. The first one that comes to mind is the practice of conducting a formal review. This involves the reviewing of artifacts such as requirements documents and design specifications. The benefit of a formal review is to create a deliverable that is accurate and free of errors and omissions. I doubt there will be much debate among the readership that formal reviews are a quality best practice.

The second practice that is often mentioned in the best practices category is requirements definition. Namely, a rigorous, definable and repeatable process that enables analysts to effectively extract requirements from a customer or end user. There are numerous methods for defining requirements, and so this best practice isn’t labeling a specific process, like a well defined formal design review process, but it is addressing the practice of requirements definition.

The third and last best practice that we can include here with little debate is code inspections. This typically refers to the Fagan type of formal code inspection process that has a well defined process with defined roles and responsibilities. So here we have a best practice that is aligned with a specific process.

So how do we know that these are the ‘best’ among the many development processes and practices? Well, for one, we have been told over and over again that this is the case and often times these success stories are substantiated with qualifying and quantifying evidence. In other words, they have made a positive difference in the life of a software development project.

And here is our first big clue as to why something may be called or labeled a best practice. Because it works. Because it can be quantified and can be proven to be successful. Case in point – have you ever worked in a software development shop that has initiated a process improvement strategy to include reviews and inspections (an agreed upon best practice) only to see that program not well defined and therefore not properly executed. And then sooner or later the practice falls by the wayside for one reason or another? I am sure you have. It is an all too common occurrence. So was it not a best practice? So is it now not a best practice? Of course not; it simply was not executed effectively and therefore it did not provide the ‘best’ results for that particular organization.

The point here is that a best practice such as code inspection or design reviews or requirements definition is only as good as its execution. And the success of that execution is somewhat depending upon measuring the process and the results. Measures don't make the process work better but they will provide information along that way that monitor compliance to a process, that measure the output of the process and that evaluate the impact on the organization; thereby ensuring the effectiveness and long term use of the best practice. And also ensuring a return on the investment made in implementing the particular best practices strategy.

A brief look at some of the measures that are associated with the above named best practices include process compliance, defect density, effective removal rates and functional sizing.

Process compliance is the basic practice of creating a formal mechanism to monitor and report compliance to a particular process. It does not provide insight as to the effectiveness or efficiency of the process but it does provide management with a view into the behaviors of the software development teams.

Defect density is often used to quantify and evaluate the number of defects attributed to a particular piece of software,
systems application or software product. It is calculated by dividing the total number of defects found by the functionality delivered (measured in function points). The measure can be used to assess the overall quality of the software and also to predict the potential need for ongoing support.

The effective defect removal rate is used to measure the rate of defect removal throughout that lifecycle. The calculation involves calculating the number of defects removed at each phase of a lifecycle divided by the total number of defects discovered. This activity occurs at the various phases of a lifecycle. So for a waterfall lifecycle, you may have defect rates attributable to your requirements phase, your design phase, your coding phase, etc. This proves to be a very powerful quality measurement tool that provides insight as to the effectiveness of your quality practices. The chart above is an example of measuring defect removal effectiveness.

Another way to apply measurement and quality best practices is to conduct an internal assessment. This involves collecting quantitative data relating to productivity and quality indicators for a selection of projects and at the same time collecting qualitative data about the development practices used on those same projects.

<table>
<thead>
<tr>
<th>Rate = Ave. Defects/KSLOC</th>
<th>Peer Reviews</th>
<th>Testing</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Range</strong></td>
<td><strong>Reqs</strong></td>
<td><strong>Design</strong></td>
</tr>
<tr>
<td>Insertion Rate</td>
<td>2.5</td>
<td>7.2</td>
</tr>
<tr>
<td>Detection Rate</td>
<td>1.0</td>
<td>5.0</td>
</tr>
<tr>
<td>Leakage Rate</td>
<td>1.5</td>
<td>3.6</td>
</tr>
<tr>
<td>Removal Effectiveness</td>
<td>40%</td>
<td>58%</td>
</tr>
<tr>
<td><strong>Average</strong></td>
<td><strong>Best in Class</strong></td>
<td><strong>Best in Class</strong></td>
</tr>
</tbody>
</table>

those practice areas which are not performing at a best practices level.

I have touched on just a few of the quality processes that we have come to think of as best practices. In addition, methods and standards such as ISO, CMMI and ITIL may all be talked about as best practices, but they will only be a best practice in your shop if they are well defined, properly executed and, most importantly, measured for success.

**About the Author**


**Author Contact Information:**

Email: David Herron: Dherron@davidconsultinggroup.com

---

### Technical Reports from DACS

- **Software Project Management for Software Assurance: A DACS State of the Art Report**
  
  Released: 9/30/2007
  

- **Enhancing the Development Life Cycle to Produce Secure Software**
  
  Released: Version 2.0, October 2008
  
  [https://www.thedacs.com/techs/enhanced_life_cycles/](https://www.thedacs.com/techs/enhanced_life_cycles/)

  
  Released: August 2007
  

  
  Released: 9/30/2007
  

PDF versions of these reports can be downloaded free of charge by registered DACS visitors who are logged in.

Hard copies can be ordered (for fee) from the DACS store online at [https://store.thedacs.com/](https://store.thedacs.com/)
Large software projects are hazardous business undertakings. More than half of software projects larger than 10,000 function points (about 1,000,000 lines of code) are either cancelled or run late by more than a year. When examining troubled software projects, it always happens that the main reason for delay or termination is due to excessive volumes of serious defects. Conversely, large software projects that are successful are always characterized by excellence in both defect prevention and defect removal. It can be concluded that achieving state of the art levels of software quality control is the most important single objective of software process improvements.

As can easily be seen from Table 1 small software projects are usually successful, but large systems are not. Why not? The main 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 (> 6.0 per function point)
2. Inadequate defect removal activities (< 85% defect removal efficiency)

In order 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 volumes 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 2009 (Data Expressed in Terms of Defects per Function Point)

<table>
<thead>
<tr>
<th>Defect Origins</th>
<th>Defect Potentials</th>
<th>Removal Efficiency</th>
<th>Delivered Defects</th>
</tr>
</thead>
<tbody>
<tr>
<td>Requirements</td>
<td>1.00</td>
<td>77%</td>
<td>0.23</td>
</tr>
<tr>
<td>Design</td>
<td>1.25</td>
<td>85%</td>
<td>0.19</td>
</tr>
<tr>
<td>Coding</td>
<td>1.75</td>
<td>95%</td>
<td>0.09</td>
</tr>
<tr>
<td>Document</td>
<td>0.60</td>
<td>80%</td>
<td>0.12</td>
</tr>
<tr>
<td>Bad Fixes</td>
<td>0.40</td>
<td>70%</td>
<td>0.12</td>
</tr>
<tr>
<td>Total</td>
<td>5.00</td>
<td>85%</td>
<td>0.75</td>
</tr>
</tbody>
</table>

Table 2 illustrates two unfortunate aspects of average software projects:

1) Large volumes of defects likely to occur; 2) Defect removal efficiency is not very good.

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 Integration (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.

There are methods of quality improvement outside of the capability maturity model. These include the Rational Unified Process (RUP), Watts Humphrey’s Team Software Process (TSP), and several forms of Agile development such as extreme programming (XP) and Crystal development.

Reducing Defect Volumes

Since the number of defects found in requirements and designs outnumber 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” are 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)
- Use of Six-Sigma for Software and/or Lean Six-Sigma
- Use of formal requirements and design inspections
- Use of joint application design (JAD) to minimize downstream changes
- Use of formal prototypes to minimize downstream changes
- Formal review of all change requests
- Revised cost and schedule estimates for all changes > 10 function points

<table>
<thead>
<tr>
<th>CMM Level</th>
<th>Defect Potential per Function Point</th>
<th>Defect Removal Efficiency</th>
<th>Delivered Defects per Function Point</th>
</tr>
</thead>
<tbody>
<tr>
<td>SEI CMM 1</td>
<td>5.50</td>
<td>73.00%</td>
<td>1.49</td>
</tr>
<tr>
<td>SEI CMM 2</td>
<td>4.00</td>
<td>90.00%</td>
<td>0.40</td>
</tr>
<tr>
<td>SEI CMM 3</td>
<td>3.00</td>
<td>95.00%</td>
<td>0.15</td>
</tr>
<tr>
<td>SEI CMM 4</td>
<td>2.50</td>
<td>97.00%</td>
<td>0.08</td>
</tr>
<tr>
<td>SEI CMM 5</td>
<td>2.25</td>
<td>98.00%</td>
<td>0.05</td>
</tr>
</tbody>
</table>

Table 3: Software Quality and the SEI Capability Maturity Model Integration (CMMI)

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.

There are methods of quality improvement outside of the capability maturity model. These include the Rational Unified Process (RUP), Watts Humphrey’s Team Software Process (TSP), and several forms of Agile development such as extreme programming (XP) and Crystal development.

Reducing Defect Volumes

Since the number of defects found in requirements and designs outnumber 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” are 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)
- Use of Six-Sigma for Software and/or Lean Six-Sigma
- Use of formal requirements and design inspections
- Use of joint application design (JAD) to minimize downstream changes
- Use of formal prototypes to minimize downstream changes
- Formal review of all change requests
- Revised cost and schedule estimates for all changes > 10 function points

Table 3: Software Quality and the SEI Capability Maturity Model Integration (CMMI)

<table>
<thead>
<tr>
<th>CMM Level</th>
<th>Defect Potential per Function Point</th>
<th>Defect Removal Efficiency</th>
<th>Delivered Defects per Function Point</th>
</tr>
</thead>
<tbody>
<tr>
<td>SEI CMM 1</td>
<td>5.50</td>
<td>73.00%</td>
<td>1.49</td>
</tr>
<tr>
<td>SEI CMM 2</td>
<td>4.00</td>
<td>90.00%</td>
<td>0.40</td>
</tr>
<tr>
<td>SEI CMM 3</td>
<td>3.00</td>
<td>95.00%</td>
<td>0.15</td>
</tr>
<tr>
<td>SEI CMM 4</td>
<td>2.50</td>
<td>97.00%</td>
<td>0.08</td>
</tr>
<tr>
<td>SEI CMM 5</td>
<td>2.25</td>
<td>98.00%</td>
<td>0.05</td>
</tr>
</tbody>
</table>

Note that formal inspections are effective in two distinct ways. Obviously they are effective in terms of defect removal efficiency. However participants in formal inspections spontaneously avoid making the same kinds of mistakes that inspections find. As a result, formal inspections are among the most effective methods of defect prevention.

Automated static analysis is a fairly new form of 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 2009. Static analysis tools only work for perhaps 25 of the most common languages such as C, C++, Java, COBOL 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 about 2% per month in new and changing requirements. Leading projects where the requirements are carefully gathered and analyzed average only a fraction of 1% per month in unplanned changes. Joint application design (JAD), prototypes, and requirements inspections are all effective in reducing unplanned requirements creep.

It happens that creeping requirements tend to be buggier 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% efficient in finding
bugs or defects. However, formal design and code inspections are more than 65% efficient in finding bugs or defects and sometimes top 85%. Static analysis is also high in efficiency against many kinds of coding defects. Therefore all leading projects in leading companies utilize both formal inspections, static analysis, and formal testing. This combination is the only known way of achieving cumulative defect removal levels higher than 95%. 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:

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 utilize formal design and code inspections first, static analysis, and then a multi-stage testing sequence afterwards. 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 “lines of code” and “cost per defect.” Unfortunately both of these have serious economic flaws.

The “lines of code” 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% of all defects are not included in the measurements. A more subtle problem with lines of code 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. Refer to the author’s book Applied Software Measurement for more details of this problem.

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. Therefore “cost per defect” goes down as numbers of bugs go up. Refer to the author’s book Software Engineering Best Practices for more details of this problem.

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 less than about 3.0 defects per function point and > 95% in defect removal efficiency will cost about 20% less to develop than identical projects with poor quality. Their schedules will be shorter by about 15%. Annual maintenance costs will be less by about 40%. 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% lower than identical projects with poor quality.

One final value point is very important. For large applications > 5,000 function points in size, high quality levels will minimize the odds of failure. For poor quality, failure rates in excess of 30% can occur at 5,000 function points. For high quality projects, failure rates are usually less than 5% and cancellations are due to business reasons rather than excessive cost and schedule overruns. The economic value of excellent quality is directly proportional to application size. The larger the software application the more valuable quality becomes.

As of 2009 the overall cost drivers for software indicate why software has a bad reputation among CEO’s and corporate executives. Our two top cost drivers are finding and fixing bugs and cancelled projects! It is no wonder that software is poorly regarded by corporate executives.

Table 4: Software Defect Removal Efficiency Ranges

<table>
<thead>
<tr>
<th>Defect Removal Activity</th>
<th>Ranges of Defect Removal Efficiency</th>
</tr>
</thead>
<tbody>
<tr>
<td>Formal requirement inspections</td>
<td>50% to 90%</td>
</tr>
<tr>
<td>Formal design inspections</td>
<td>45% to 85%</td>
</tr>
<tr>
<td>Formal code inspections</td>
<td>45% to 85%</td>
</tr>
<tr>
<td>Static analysis (automated)</td>
<td>55% to 90%</td>
</tr>
<tr>
<td>Unit test (manual)</td>
<td>15% to 50%</td>
</tr>
<tr>
<td>Unit test (automated)</td>
<td>20% to 60%</td>
</tr>
<tr>
<td>New function test</td>
<td>20% to 35%</td>
</tr>
<tr>
<td>Regression test</td>
<td>15% to 30%</td>
</tr>
<tr>
<td>Integration test</td>
<td>25% to 40%</td>
</tr>
<tr>
<td>Performance test</td>
<td>20% to 40%</td>
</tr>
<tr>
<td>System test</td>
<td>25% to 55%</td>
</tr>
<tr>
<td>Acceptance test (1 client)</td>
<td>25% to 35%</td>
</tr>
<tr>
<td>Low-volume Beta test (&lt; 10 clients)</td>
<td>25% to 40%</td>
</tr>
<tr>
<td>Overall cumulative ranges</td>
<td>70% to 99%</td>
</tr>
</tbody>
</table>

Table 5: The Top 15 U.S. Software Cost Drivers in Rank Order Circa 2009

1. The cost of finding and fixing bugs
2. The cost of cancelled 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 cancelled 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 5 is a professional embarrassment. No true engineering discipline should have defect repairs and cancelled 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 2009.

Table 6 shows a hypothetical rearrangement of cost drivers that should be a goal for software engineers over the next 10 years. Our top cost driver should be innovation and designing new feature: not bug repairs. Table 6 illustrates how costs should be apportioned circa 2019:

Table 6: The Top 15 U.S. Software Cost Drivers in Rank Order Circa 2019
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 cancelled projects
13. The cost of finding and fixing bugs
14. The cost of recovery from security flaws and attacks
15. The cost of cancelled projects

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 cancelled projects should be at the bottom of the list of cost drivers and not at the top as they are in 2009.

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 large numbers of projects that were either failures or quite successful, 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, cancelled projects and disasters are the norm.

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

References and Readings


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


About the Author

**Capers Jones** is currently the President and CEO 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. Capers Jones founded SPR in 1984. Before founding SPR Capers was Assistant Director of Programming Technology for the ITT Corporation at the Programming Technology Center in Stratford, Connecticut. He was also a manager and researcher at IBM in California.

Capers Jones is a well-known author and international public speaker. Some of his books have been translated into six languages. All of his books are translated into Japanese and his newest books are available in Chinese editions as well. He has been the keynote speaker at the annual Japanese Symposium on Software Testing in Tokyo, the International Function Point Users Group (IFPUG), the World Congress of Quality, and the opening of the Singapore chapter of the Project Management Institute. Mr. Jones also speaks at internal corporate events for companies such as IBM, Satyam, Hewlett Packard, many others.

Capers Jones’ research studies include quality estimating, quality measurement, software cost and schedule estimation, software metrics, and risk analysis. He has consulted at more than 150 large corporations and also at a number of government organizations such as NASA, the U.S. Air Force, U.S. Navy, Internal Revenue Service, and the U.S. Courts. He has also worked with several State governments.

Author Contact Information

Email: **Caper Jones**; CJonesiii@cs.com
Accessible
Classes available 24/7/52
From:
• Home
• Office
• Travel

Accreditation
Certification
• CEUs granted
• IACET accreditation supported

Affordable
• Pay one annual subscription fee ($475)
• Take as many classes as you want

Comprehensive
• 450+ classes
• 12,000 topics
• Programming & Web Development
• Course Catalog

Latest Technologies
• Java XML
• Oracle
• Server technologies
• .NET Framework

Flexible
• Fit course work into your schedule
• Self Paced
• Refresh Knowledge
• Study during lunch or after work
Predicting Software Quality

DEFECT DISCOVERY FOLLOWS A RAYLEIGH CURVE, WHICH CAN BE USEFUL IN DEFECT OR ERROR PREDICTION. SCHEDULE COMPRESSION AND TEAM SIZE CAN ALSO HAVE AN EFFECT ON SOFTWARE QUALITY.

by Donald M. Beckett and Douglas T. Putnam

Quality. It’s such a positive word, one in the software world that engenders thoughts of superior design, ease of use, speed, stability, security, and accuracy. Some of these attributes can be measured; others are subjective. Taken as a whole, they define quality to us. Ironically, in software development the quality measures we employ are largely negative and focus on problems encountered testing or using the software. In this sense, the fewer defects encountered, the higher the quality. Measures such as Defects/KLOC and Mean Time to Failure (Defect) illustrate this concept. As useful as these are (and the bulk of this article focuses on them), superior results in them represent an absence of problems, which while good, only captures a portion of what we understand as quality.

What is acceptable quality?

During a recent presentation I asked the audience to raise their hands if they had ever encountered a bug in a popular suite of desktop software from a large West Coast company. Everyone raised his or her hand. In this case acceptable quality did not preclude plenty of residual defects. Perhaps more important to them, the software package was feature rich, widely used, and competitively priced: all of which supported the vendor’s business objectives of profitability and market dominance. The problems everyone encountered with the package were below the threshold of pain that would cause them to abandon it. The quality of the product was acceptable.

In contrast, look at the software that controls pacemakers and other medical devices, the U.S. missile defense system, or monitors a nuclear power plant. What is acceptable quality for desktop software would be unacceptable for them. Failure could cause death, war, or a nuclear catastrophe. There is an important point here: acceptable quality is a relative measure that is tightly coupled to the objectives it is designed to fulfill. Desktop software fulfills business needs for word processors, spreadsheets, and presentation media. It sells at a price that promotes mass distribution and maximum profit for the vendor. It has a short life cycle with “upgrades” released every few years (planned obsolescence) that virtually guarantee that it is never thoroughly tested and debugged. Medical device software deals with life and death – and potential lawsuits if it fails.

Quality, a team and process measure

The good news and the bad news is that organizations are creatures of habit that develop software in a similar fashion from one project to another. Although these patterns can be modified over time when consciously addressed, they are remarkably persistent and can be used to predict future performance for cost, time to market, and quality. If an organization consistently produces superior software it has the processes and culture in place that will help it to do so on the next project. Unfortunately, the reverse is also true. Knowledge of an organization’s past performance helps determine a defect rate that can be used to predict the number of defects on the next project. Figure 1 shows defect trend lines from a large (8000 software project) database. The horizontal axis represents the size (how much software) of the project. The vertical axis shows the number of defects. The trend lines represent the average and plus and minus one standard deviation. The projects used to create the trend lines are from hundreds of business projects from many distinct organizations completed.
in the last five years. Notice, too, that the scale on the graph is log-log: the relationship between how much software is developed and the defects created is exponential not linear.

What is striking and intimidating is the amount of variability. A project that creates 50,000 lines of code could create 40 defects at -1, 168 if average, and 685 (seventeen times as many) at +1. Clearly, comparing one's organization to an industry standard is not a first choice. Unfortunately, for those organizations who do not maintain software project history, it may be the only one.

The defect trend lines in Figure 2 are from one organization. While the number of defects increases with project size, as expected, the range is much narrower. For a 50,000 lines of code project the -1 is 9 defects, the average 17, and +1 is 33. Clearly, knowledge of an organization's own patterns is a big help in estimating the number of defects that a future project will create.

Three aids for predicting software quality and improving it

- In a software project, defect detection follows a predictable pattern, one that is best described by a Rayleigh Curve
- A project's schedule has a profound impact on the quality of the software that is delivered
- The size of the project team directly influences the number of defects a project creates

The Rayleigh Curve

The software development process is a continuous one in which functionality is designed then expressed in languages which we refer to as source code. Defects are introduced as the source code is created. In this context it is appropriate to model the defect creation, discovery and elimination process as a function of time.

From the time that we start to define, design, write, integrate and test source code we have the capability of introducing defects into the product. In fact, at the beginning of a project there is nothing but a high level abstraction of what must be accomplished by this system. If we were to draw a graph of the defect discovery process at this point in time we would show no product and, of course, there would be no defects. As time progresses we complete some design and begin to generate code. As the design and code are completed, problems that were introduced earlier are discovered and fixed. At some point in time the product reaches a maximum defect discovery rate. As work progresses the volume of remaining defects is reduced and the discovery rate gradually falls off.

The QSM defect estimation approach uses the Rayleigh function to forecast the discovery rate of defects as a function of time throughout the software development process. The Rayleigh function is a specific instance of one of the models in the Weibull family of reliability models. QSM believes there is a solid theoretical basis for its use as a software reliability modeling tool. The Rayleigh function was discovered by the English physicist Lord Rayleigh in his work related to scattering of acoustic and electro-magnetic waves. In statistics it has been found that in processes with a large number of random sources of Gaussian noise, none of which are dominant, the Rayleigh function represents well the vector sum of all those Gaussian sources. We have empirically found that the Rayleigh model seems to represent well iterative design processes in which significant feedback is inherently part of the solution process. Further, we have found in our research that a Rayleigh reliability model closely approximates the actual profile of defect data collected from software development efforts. In the QSM reliability modeling approach the Rayleigh equation is used to predict the number of defects discovered over time. The QSM application of the Rayleigh model has been formulated to cover the time period from Preliminary Design Review (PDR - High Level Design is Complete) until 99.9% of all the defects have been discovered. A sample Rayleigh defect estimate is shown in Figure 3.

For defect detection and remediation this has important implications. If, based on our history, we know how many
defects a project is likely to create and we know that defect
discovery follows a Rayleigh curve, we will be able to see if
we are ahead of or behind schedule in defect detection and
correction and also how many defects remain to be uncovered.
So, whether the software is being developed to the reliability
specifications of a word processor or that of the space shuttle
we will be able to determine when it meets the applicable
reliability standard and is ready to go live.

Project Schedule and Quality

In the 1970’s Frederick Brooks in “The Mythical
Manmonth” identified schedule pressure as the cause of more
software project woes that perhaps all other factors combined.
Its impact on software quality is pronounced and negative.
Parametric estimation tools can be effectively employed to
model the impact of schedule compression. Figure 4 is a model
of a 100,000 line of code command and control project. It was
modeled so that schedule, effort, and defects were all average
when plotted against their corresponding trend lines. The
trend lines, themselves, were created from recently completed
command and control projects. What stands out is that the
more aggressively the project is compressed, the rate of defect
creation virtually explodes.

<table>
<thead>
<tr>
<th>Schedule/Quality Trade-off</th>
</tr>
</thead>
<tbody>
<tr>
<td>Default</td>
</tr>
<tr>
<td>Duration Mths</td>
</tr>
<tr>
<td>Defect Count</td>
</tr>
<tr>
<td>% Change</td>
</tr>
</tbody>
</table>

Figure 4

It should surprise no one that projects with aggressive
schedules create more defects. When design and development
are done under pressure more errors occur. Likewise, the
delivered product will contain more latent defects since the
testing cycle will be short changed. But, it is very useful to be
able to model and predict the impact of choosing a particular
schedule before committing to it. And that can be done.

Project Team Size

A project’s team size directly affects the number of defects
that will be created and the quality of the project at release.
But, how do you determine the right team size? Different
sized projects require different teams. Some projects require
specialized skills that others do not. The way we have chosen
is to create trend lines for staff vs. size from our database
that look just like those in Figures 1 and 2 only with staff
on the Y axis. Where the Average trend line intersects with
a particular project size is the Y coordinate (staff). Using the
same command and control project we used for schedule
we created estimate models for the average staff, which was
16, and for a team size of 32 (around one standard deviation
above average). The results are summarized in Figure 5.

Adding staff to a project is a time-honored method used
to try to bring projects in on time. Its impact on schedule is
minimal. The cost in quality and dollars is substantial.
Conclusion

So, what is a business leader planning a project supposed to do with all of this? The first action item is to begin capturing project metrics for completed projects. These are an organizational self-portrait: one that will quickly be lost as projects compete and teams are reassigned. This does not need to be a large scale undertaking; but it does need to be a conscious one. For starters, collect

- How long the project lasted (duration)
- How much effort was expended (from time tracking)
- Cost
- Defects discovered during testing
- A measure of the software size (lines of code from the configuration management system, function points, modules created, etc.

Although the focus here has been on quality, this information is directly applicable when estimating the cost and schedule of projects. In the quality arena, it will define an organizational profile that will help determine when a product is ready for release.

The second point is that there are optimal schedules and team sizes for projects, which if followed will directly improve the quality of the software.

Third, software projects can and should be modeled to determine the best schedule and staffing solutions and to predict the impact if alternatives are necessary. Modeling is quick, inexpensive, and provides information before the fact rather than after.

About the Authors

Donald M. Beckett has been active in software as a developer, manager, trainer, researcher, analyst, and consultant for over 25 years. Since 1995 the focus of his work has been software measurement and estimating, first with EDS and since 2004 with Quantitative Software Management. For many years he has worked with parametric models and tools to estimate and create forecasts to completion for software projects. Don lives in Poulsbo, WA.

Douglas T. (Doug) Putnam is a Managing Partner for Quantitative Software Management (QSM) Inc. He has 28 years of experience in the software measurement field and is considered a pioneer in the development of this industry. Mr. Putnam has been instrumental in directing the development of the SLIM Suite of software estimation and measurement tools. The SLIM tools are considered a best in class application tool set. Mr. Putnam is a sought after international author, speaker and consultant. His responsibilities include managing and delivery of QSM software measurement services, defining requirements for the SLIM Product Suite and overseeing the research activities derived from the QSM project history repository.

Author Contact Information:
Email: Don Beckett: don_beckett@qsm.com
Email: Doug Putnam: doug_putnam@qsm.com
Applying Agile Practices to Improve Software Quality

The Impact of Agile Development May Either Positively or Negatively Affect Software Quality Based on the Particular Agile Method Used. Some Specific Agile Methods Can Have a Positive Impact on Software Quality.

by Arlene Minkiewicz

Increased frustration with failed software projects coupled with the need to keep up with rapidly changing business needs is driving software development organizations to revisit the way they go about building software. Agile development has emerged as one possible solution to the woes of the software industry. Agile enthusiasts claim significant increases in the quality of their software while detractors cite instances where rapid development and loose structure lead to decreases in quality. This happens because not all ‘agile’ is created equally. Some agile practices are more likely, when implemented correctly, to impact quality than others. This article discusses specific agile practices which have been proven to have a positive impact on quality and offers practical advice about how best to implement them to maximize their impact on quality.

Agile Development

As documented in the agile manifesto [1]...

“We are uncovering better ways of developing software by doing it and helping others do it. Through this work we have come to value:

• Individuals and interactions over processes and tools
• Working software over comprehensive documentation
• Customer collaboration over contract negotiation
• Responding to change over following a plan

That is, while there is value in the items on the right, we value the items on the left more.”

Agile development was introduced in the mid ‘90’s as organizations were struggling with fast-paced software development projects brought on by changing business cultures. Its premise is that productivity and quality are derived from the techniques and disciplines used and our interpersonal relationships.[2] Agile practices focus on simplicity, customer focus, shared responsibility, close collaboration, and frequent and direct communication. More traditional methods focus on well documented and reviewed work products at each phase of the software development lifecycle. They require that the entire release be understood and designed before coding starts. The agile philosophy assumes that things will change before the release is complete, thus some of the upfront design and planning effort is wasted. Agile requires the development team to break a software project into small pieces of functionality, developing enough code to deliver only this small piece
of functionality, and then frequently building incremental releases to be reviewed by customers. Agile practices allow for continuous integration and continuous customer reviews and feedback.

Agile development is based on several values (different forms of agile name them differently): communication, simplicity, feedback, courage and respect. While different organizations implement different agile practices, these core values are a constant theme. Communication can be accomplished through various mechanisms including daily stand-up meetings, iteration planning and retrospective meetings, and customer walkthroughs. Co-location of the development team is also a catalyst for improved communication. While this does tend to create a culture of interruption, it leads to an environment where no one is afraid to ask for help, advice, suggestions or ideas.

Basic agile concepts include:

• Test Driven Development – unit tests are written prior to any code, developer writes just enough code to pass the test
• Simple Design – do the simplest thing that can possibly work without speculating about future features
• Pair Programming – all production code is written in pairs as an on-going collaboration
• Refactoring – improving existing code without changing functionality based on lessons learned through implementations
• Continuous Integration – integrations occur hourly or daily and automated tests are applied
• Collective ownership – all developers are responsible for the integrity of the code
• User Stories – stories describe a piece of system capability to be implemented, a placeholder for an on-going conversation about requirements
• Short iterations – Frequent releases (not necessarily external) that deliver business value.

Software Quality

Wikipedia defines software quality as a measure of how well software is designed (quality of design) and how well the software conforms to the design (quality of conformance) [3]. This definition implies two aspects of software quality:

• Building the right thing
• Building it right

Building it right can be measured by a count of defects per size unit (Source Lines of Code, Function Points, Use Cases, etc.), number of passed tests, defect rate, etc. Determining whether the right thing is being built is somewhat problematic as it really is a measure of how delighted the customer is with the software that is delivered.

The literature contains many examples of studies and experiments (both academic and in industry) that indicate agile improves both software quality and customer satisfaction. The Data & Analysis Center for Software (DACS) published a report in 2007 [4] cites many instances where agile practices have improved quality both with reduced defects and improved customer satisfaction.

Agile Practices Most Likely to Impact Software Quality

While it’s generally the combination of agile practices that lead to quality improvements, there are several practices, when implemented correctly, that specifically target quality improvements. These are described in more detail in the following paragraphs.

Pair Programming

Pair programming is often cited as a reason for increased software quality. With pair programming two programmers are paired together to complete a single programming task. They have one computer between them and they collaborate to determine the best design and best implementation for that design. Pair programming applies the “two heads are better than one” paradigm while enforcing on-going continuous review of the code.

While the results are mixed on whether pair program increases or decreases productivity, its impact on quality has been documented in many studies. In a study reported on in [5], comparing programs developed by single programmers to those developed by pairs, the number of post development tests that passed increased by 15%. The report also presented experiential data that pair programming increased the quality of designs as well. [6] cites two examples supporting the quality benefits of pair programming – an experiment conducted at the University of Utah finding an average of approximately 14% increase in passed tests on programs using pair program and experiential findings on the Chrysler Company's C3 project in 1997.

It is important that pair programming be implemented effectively. Pair programming is not just having one
programmer looking over the shoulder of another as they type. At any given time, one programmer assumes the role of driver, sitting at the keyboard and writing code or tests. The job of the other team member is to navigate. This role requires that the programmer be thinking through the problem solution looking for better, cleaner ways to solve it while at the same time keeping an eye on the work product of the driver, scanning for mistakes in logic and coding. It is important that roles shift during the execution of the task. It is also important that within a development team, many different pairing combinations occur. If the same two programmers are always pairing with each other, the pairing may get stale and other benefits of pair programming (such as on the job training and team awareness of the code) will be lost. The more eyes on the code the more likely it is to be clean and error free.

There are situations where pair programming might not be effective. Distributed teams will find it difficult to be successful with pair programming, although [2] suggests that techniques and tools to help distributed teams realize some of the benefits of pair programming. It could also be problematic in environments where telecommuting and flexible schedules are part of the culture. There are also types of project which may not be well suited to successful pair programming—where technologies and/or software content is very diverse within the organization requiring highly specialized programmers.

Test Driven Development (TDD)

Test driven development requires that no code is written for a feature until the tests for that feature have been written and shown to fail. Before a developer can begin to code for a particular feature they need to understand the requirements, intent, and exceptions of the feature well enough to write tests for it. The process for each new feature begins with a study of the user story in order to assess requirements and write an automated test for the feature. This test is then executed to ensure that it fails (since the feature has yet to be written). Once the test has been proven to fail, the developer writes what they believe to be the minimal amount of code to make the test pass. Once enough code has been written to make the test pass, the developer then incrementally refactors the solution to make it simpler and cleaner. Because each increment includes rerunning of the test, the developer can refactor with confidence that the feature will not negatively impacted.

Tests conducted at Microsoft in two different environments showed defect rate (defects/KLOC) decreases at factors of 2.5x and 4.2x between projects determined to be similar in size and scope, one with TDD and one without.[7] Both tests indicated increased development time as well but not nearly as significant as the increase in quality. [8] contains two tables which summarize the results of 18 studies – 9 from academia and 9 from industry. These studies ranged from controlled experiments, to quasi-controlled experiments to case studies and the length of studies ranged from 1.75 hours to 1.5 years. Different measures of quality were employed by different studies (functional tests passed, defect rates) Of the studies, 10 found mild to significant improvement in quality, 7 were inconclusive or showed no difference while only one of the studies indicated a decrease in quality. Lisa Crispin in [9] correctly points out that there is more to quality than defect counts or passed functional tests. Her teams employ customer test driven development in addition to developer test driven development. Testers work with customers to develop high level functional tests for features. The developers use these tests to fully understand the user’s requirements. The result is happy (often delighted) customers – contributing to the ‘doing the right thing’ aspect of quality.

Clearly there are factors that will make TDD more or less successful as a quality initiative, but it seems to deliver promise in many circumstances. A (non-scientific) perusal of the studies that have been accomplished to date seems to indicate that smaller projects and studies see less quality increases than larger projects. This could be an indicator that the investment may not be warranted strictly as a quality initiative, although TDD reaps other benefits such as documentation of design details (through the test artifacts) and smoother maintenance over time as automated tests constantly monitor the impacts of changes to the code base. Other items important to the success of TDD include concrete understanding of the expectations of TDD, effective training and an organizational culture friendly toward TDD.

Continuous Integration

Continuous Integration is an agile practice that requires that changes to the code base be continuously integrated into an operational system. Generally this process is automated and integrated with an automated test suite in order to give real time feedback when a developer makes a change that causes bad behavior in unexpected places in the system. During development, individual developers will ‘check out’ a copy of the code, make changes to fix a bug or add a feature, tests this change and then ‘check in’ the changed code. During this time, other developers have made changes to other parts of the code base. A common problem arises when the ‘check out’ time is too long because while each individual developer is able...
The DACS Gold Practice Initiative:

- Promotes effective selection/use of software acquisition & development practices
- Defines essential activities/benefits of each practice
- Considers the environment in which each practice is used
- Addresses the timeliness of practice benefits
- Recognizes interrelationships between practices that influence success or failure
- Contains quantitative and qualitative information
- A continually evolving resource for the DoD, Government, Industry and Academia
- Free to use/free to join

Learn More About the DACS Gold Practice Initiative:
http://www.goldpractices.com

Current Gold Practices:

- Acquisition Process Improvement
- Architecture-First Approach
- Assess Reuse Risks and Costs
- Binary Quality Gates at the Inch-Pebble Level
- Capture Artifacts in Rigorous, Model-Based Notation
- Commercial Specifications and Standards/Open Systems
- Defect Tracking Against Quality Targets
- Develop and Maintain a Life Cycle Business Case
- Ensure Interoperability
- Formal Inspections
- Formal Risk Management
- Goal-Question-Metric Approach
- Integrated product and Process Development
- Metrics-Based Scheduling
- Model-Based Testing
- Plan for Technology Insertion
- Requirements Management
- Requirements Trade-Off/Negotiations
- Statistical Process Control
- Track Earned Value
to pass the suite of tests with their instance of the code, the combination of their code with the changes of other developers causes tests to fail.

Continuous integration requires that an application is rebuilt and tested each time a change is made to the code base. Some shops automate this process through their version control system while others have automatic builds and tests that run every couple of hours with developers checking in code often. Solving any failures in these tests becomes a top priority for the team. When there are failures found during integration it is easy to isolate the problem code and correct the problem. The longer the period between integrations the more likely the team is thrown into integration hell – unable to determine which of many code changes is responsible for failed tests.

While there doesn’t appear to be much quantitative evidence specifically relating continuous integration to increases in quality, it has been observed that projects using continuous integration tend to have dramatically less bugs in production and in process[10]. Steve McConnell in [11] indicates that one of the benefits of frequent builds is reduced risk of low quality. Clearly this is influenced significantly by the number and quality of the tests and the amount of automation applied to the testing for each build. Quality impacts also depend significantly on how seriously the team acts when tests do fail. If failed tests are not treated as a top priority the value of continuous integration is seriously depleted.

Short Iterations

Most agile shops ‘release’ software with added business value every couple of weeks. These releases may never see the light of day outside of the development group and customer stakeholders but the team works toward each release as though it were intended to be delivered to a customer. The team basically transcends a whole ‘waterfall’ inside the iterations with planning, requirements analysis, test development, coding and integration and test. Although the time and effort associated with creating production ready releases frequently introduces overhead that more traditional projects don’t experience, there are several good reasons why creating software in short iterations makes sense. If a release is built every few weeks with incremental implementations of features, the customer has the opportunity to see how the development team is interpreting their requirements and has the chance to redirect efforts when they deviate from their expectations. If a software project gets into trouble and falls behind the customer has the opportunity to reprioritize requirements so the best set of functionality gets to market in a timely fashion. Frequently releases also give the development team and the customer lots of time to perform functional tests on features as they emerge.

As with continuous integration, there is little quantitative evidence that specifically speaks to short integrations increasing quality. Certainly early and frequent visibility of implemented features to the customer is highly likely to increase the probability that the software implemented ‘does the right thing’ contributing to delighted customers. Additionally, frequent small releases give the test team time to focus on features as they emerge, increasing the chance that defects are detected and addressed as they emerge.

Conclusion

Agile development in all of its many implementations has been proven in many, instances to decrease defects and increase customer satisfaction. There is also research to support the opposite conclusion. This disparity stems from the fact that different organizations adopt different forms of agile and often adopt different sets of agile practices within the form of agile they choose. Some agile practices, or combination of agile practices, if implemented correctly, are more likely to result in quality improvements than others. Practices such as pair programming, test driven development, continuous integration and short iterations have all be cited as components of a good quality improvement initiatives.

In addition to practices targeted to instilling quality in the software, agile brings another quality enhancer to the table. All forms of agile encourage frequent and meaningful communication both within the development team and with the customer. The best way to ensure that the software that is delivered meets the customer needs and does the right thing is to have on-going conversations with the stakeholders and between the developers.

References


quality (retrieved January 2010)


About The Author

Arlene Minkiewicz is the Chief Scientist at PRICE Systems L.L.C. In this role she leads the Cost Research activity for the entire suite of cost estimating products that PRICE develops and maintains. Ms. Minkiewicz has over 24 years of experience with PRICE, designing and implementing cost models. Her recent accomplishments include the development of new cost estimating models for software and Information Technology projects. She has published articles on software measurement and estimation in Crosstalk, Software Development and British Software Review. She has received Best Paper awards on several of her recent research projects from two professional estimating societies (ISPA, SCEA), and was named Parametrician of the Year for ISPA in 2002.

Author Contact Information

Email: Arlene Minkiewicz: Arlene.minkiewicz@PRICESystems.com

ACCESS THE DACS ROI DASHBOARD!

http://www.thedacs.com/databases/roi/
Modeling Software Defect Dynamics

RECENT ENHANCEMENTS TO THE CONSTRUCTIVE QUALITY MODEL (COQUALMO) HELP IN ASSESSING DEFECT DYNAMICS TO BETTER UNDERSTAND THE TRADEOFFS OF DIFFERENT PROCESSES AND TECHNOLOGIES FOR REDUCING DEFECTS.

by Raymond Madachy, Barry Boehm and Dan Houston

Software defects are not created equal and exhibit various dynamic behaviors that complicate project decisions. Interrelated factors that affect defect dynamics include: project environment and practices that impact the overall defect generation and detection rates, phases and the timing of activities, the types of defects, and mission circumstances.

The practices to find and remove defects have varying efficiencies with respect to the types of defects and the lifecycle timing. Difference classes of defects have mission-dependent risk profiles (e.g. real-time or not), some methods are better suited than others for finding certain types of defects, and there are overlaps between the methods with respect to the classes of defects.

Parametric modeling and simulation can help reason about strategies for reducing defects by quantifying the impact of different processes and technologies. This paper presents our ongoing work to extend and refine the COnstructive QUALity MOdel (COQUALMO) for assessing defect dynamics to better understand the tradeoffs. Using parametric cost and defect removal inputs, static and dynamic versions of the model help one determine the impacts of quality strategies on defect profiles, cost and risk. We describe the evolution of the models into increasingly detailed forms.

Software quality processes can be assessed with the models that predict defects introduced and removed. The models are calibrated with empirical data on defect distributions, introduction and removal rates; and supplemented with Delphi results for detailed defect detection efficiencies.

The basic version of COQUALMO [1] models medium grain defect introduction and detection rates. It uses COCOMO II [2] cost estimation inputs with defect removal parameters to predict the numbers of generated, detected and remaining defects for requirements, design and code. It models the impacts of defect reduction practices for the primary activities of automated analysis, peer reviews, and execution testing and tools on these defect categories. It is a static model with time-invariant factors and presents the final cumulative quantities of defects remaining.

However, the top-level decomposition by phase injection was not fine enough when considering the relative dynamics of different types of defects. Subsequently we refined the taxonomy for NASA to classify them using the Orthogonal Defect Classification (ODC) [3].

Though the static models demonstrated the rough tradeoffs between practices, we reformulated the ODC extension into a dynamic version that provides insight into time trends and is suitable for continuous usage on a project. It is more realistic because it simulates the changing of factors throughout a project (e.g. improved peer reviews), their interrelationships and time-dependencies. It uses system dynamics for simulation modeling.

Most recently, Dynamic COQUALMO is an improved system dynamics model [4] that further refines the defect practice impacts in terms of their varying efficiencies. It is also calibrated to project practices and empirical defect rates at The Aerospace Corporation, and serves for retrospective studies of completed projects for process improvement.

The original COQUALMO model was developed with industrial data supported by affiliates of the University of Southern California Center for Systems and Software Engineering (USC-CSSE). Further empirical data was used from manned and unmanned flight projects to tailor and calibrate the models for NASA. At The Aerospace Corporation we used empirical data for local calibration. This paper presents the latest developments in the ongoing empirical research.

COQUALMO Background

Cost, schedule and quality are highly correlated factors in software development. They essentially form three sides of a triangle, because beyond a certain point it is difficult to increase the quality without increasing either the cost or schedule, or both. Similarly, development schedule cannot be drastically compressed without hampering the quality of the software
product and/or increasing the cost of development. Software estimation models can (and should) play an important role in facilitating the balance of cost/schedule and quality.

Recognizing this important association, COQUALMO was created as an extension of the CONstructive COst MOdel (COCOMO) [2], [5] for predicting the number of residual defects in a software product. The model enables ‘what-if’ analyses that demonstrate the impact of various defect removal techniques. It provides insight into the effects of personnel, project, product and platform characteristics on software quality, and can be used to assess the payoffs of quality investments. It enables better understanding of interactions amongst quality strategies and can help determine probable ship time.

A black box representation of COQUALMO’s submodels, inputs and outputs is shown in Figure 1. Additions to COCOMO II are shown in blue. Its input domain includes the COCOMO cost drivers and three defect removal profile levels. Defect introduction and removal is illustrated as a pipe and tank flow model in Figure 2. The defect removal profiles and their rating scales are shown in Table 1. More details on the removal methods for these ratings are in [2]. From these inputs, the tool produces an estimate of the number of requirement, design and code defects that are introduced and removed as well as the number of residual defects remaining in each defect type.

The COQUALMO model contains two submodels: 1) the defect introduction model and 2) the defect removal model. The defect introduction model uses a subset of COCOMO cost drivers and three internal baseline defect rates (requirements, design, code and test baselines) to produce a prediction of defects that will be introduced in each defect category during software development. The defect removal model uses the three defect removal profile levels, along with

![Figure 1: COQUALMO Extension to COCOMO](image1)

![Figure 2: Defect Introduction and Removal Model](image2)

<table>
<thead>
<tr>
<th>Rating</th>
<th>Automated Analysis</th>
<th>Peer Reviews</th>
<th>Execution Testing and Tools</th>
</tr>
</thead>
<tbody>
<tr>
<td>Very Low</td>
<td>Simple compiler syntax checking</td>
<td>No peer reviews</td>
<td>No testing</td>
</tr>
<tr>
<td>Low</td>
<td>Basic compiler capabilities for static program analysis, syntax, type-checking</td>
<td>Ad-hoc informal walkthroughs, minimal preparation, no follow-up</td>
<td>Ad-hoc testing and debugging, basic text-based debugger</td>
</tr>
<tr>
<td>Nominal</td>
<td>Some compiler extensions for static program analysis and type checking, basic requirements and design, consistency, traceability checking</td>
<td>Well-defined sequence of preparation, review, minimal follow-up, informal review roles and procedures</td>
<td>Basic unit test, integration test, system test process, basic test data management, problem tracking support, test criteria based on checklists</td>
</tr>
<tr>
<td>High</td>
<td>Intermediate level module and inter-module code syntax and semantic analysis, simple requirements and design view consistency</td>
<td>Formal review roles and procedures applied to all products using basic checklists, follow-up</td>
<td>Well-defined test sequence tailored to organization (acceptance/alpha/beta/flight/etc) test, basic test coverage tools, test support system, basic test process management</td>
</tr>
<tr>
<td>Very High</td>
<td>More elaborate requirements and design view consistency, basic distributed processing and temporal analysis, model checking, symbolic execution</td>
<td>Formal review roles and procedures applied to all product artifacts and changes, basic review checklists, root cause analysis, use of historical data on inspection rate, preparation rate, fault density</td>
<td>More advanced test tools, test data preparation, basic test code support, distributed monitoring and analysis, assertion checking, metrics-based test process management</td>
</tr>
<tr>
<td>Extra High</td>
<td>Formulated specification and verification, advanced distributed processing and temporal analysis, model checking, symbolic execution</td>
<td>Formal review roles and procedures for fixes, change control, extended review checklists, root cause analysis, continuous review process improvement, user and customer involvement, statistical process control</td>
<td>Highly advanced test tools for test analysis, distributed monitoring and analysis, assertion checking, integration of automated analysis and test tools, model-based test process management</td>
</tr>
</tbody>
</table>

Table 1: COQUALMO Defect Removal Ratings
the prediction produced by the defect introduction model, to produce an estimate of the number of defects that will be removed from each category.

**ODC COQUALMO Models and Tools**

There are different implementations of ODC COQUALMO. Initially we created our static model in a spreadsheet and then transitioned to a web-based version as part of the integrated COCOMO Suite tool at http://cse.usc.edu/tools/COQUALMO.php or https://diana.nps.edu/MSAcq/tools/COQUALMO.php. The inputs to the static model are shown in Figure 3 while Figure 4 shows an example of ODC defect outputs. A dynamic simulation version models the defect generation and detection rates over time for continuous project usage, and provides continuous outputs as shown in the next section.
ODC Extension

At USC-CSSE we were evaluating and updating software cost and quality models for critical NASA flight projects [3]. A major focus of the work was to assess and optimize quality processes to minimize operational flight risks. We extended the COQUALMO model for software defect types classified with ODC. ODC COQUALMO decomposes the top-level defect types into more granular ODC categories.

The ODC taxonomy provides well-defined criteria for the defect types and has been successfully applied on NASA projects. The ODC defects are then mapped to operational flight risks, allowing “what-if” experimentation to determine the impact of techniques on specific risks and overall flight risk. The tool was calibrated to ODC defect distribution patterns per JPL studies on unmanned missions. A Delphi survey was completed to quantify ODC defect detection efficiencies, gauging the effect of different defect removal techniques against the ODC categories.

The approach is value-based [6] because defect removal techniques have different detection efficiencies for different types of defects, their effectiveness may vary over the lifecycle duration, different defect types have different flight risk impacts, and there are scarce resources to optimize. Additionally the methods may have overlapping capabilities for detecting defects, and it is difficult to know how to best apply them. Thus the tools help determine the best combination of techniques, their optimal order and timing.

ODC COQUALMO can be joined with different risk minimization methods to optimize strategies. These include machine learning techniques, strategic optimization and the use of fault trees to quantify risk reductions from quality strategies. We have demonstrated the integration with automated risk minimization methods to design higher value quality processes, in shorter time and with fewer resources, to meet stringent quality goals on projects [7].

ODC COQUALMO decomposes defects from the basic COQUALMO model using ODC [8]. The top-level quantities for requirements, design and code defects are decomposed into the ODC categories per defect distributions input to the model. With more granular defect definitions, ODC COQUALMO enables tradeoffs of different detection efficiencies for the removal practices per type of defect. Table 2 lists the ODC defect categories used in the model, and against which data is collected.

This more detailed approach takes into account the differences between the methods with specific defect pairings. Peer reviews, for instance, are good at finding completeness defects in requirements but not efficient at finding timing errors for a real-time system. Those are best found with automated analysis or execution and testing tools.

<table>
<thead>
<tr>
<th>Requirements</th>
<th>Design/Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>- Correctness</td>
<td>- Interface</td>
</tr>
<tr>
<td>- Completeness</td>
<td>- Timing</td>
</tr>
<tr>
<td>- Consistency</td>
<td>- Class/Object/Function</td>
</tr>
<tr>
<td>- Ambiguity/Testability</td>
<td>- Method/Logic/Algorithm</td>
</tr>
<tr>
<td></td>
<td>- Data Values/Initialization</td>
</tr>
<tr>
<td></td>
<td>- Checking</td>
</tr>
</tbody>
</table>

Table 2: ODC Defect Categories

The model also provides a distribution of defects in terms of their relative frequencies. The tools described in the next section have defect distribution options that allows a user to input actuals-based or expert judgment distributions, while an option for the Lutz-Mikulski distribution is based on empirical data at JPL [9].

The sources of empirical data used for analysis and calibration of the ODC COCOQUALMO model were described in [10]. The quality model calculating defects for requirements, design and code retains the same calibration as the initial COQUALMO model. The distribution of ODC defects from [9] was used to populate the initial model with an empirically-based distribution from the unmanned flight domain at JPL. The Lutz-Mikulski distribution uses the two-project average for their ODC categories coincident across the taxonomy used in this research for design and code defects. Their combined category of “Function/Algorithm” is split evenly across our two corresponding categories.

A comprehensive Delphi survey [11], [12] was used to capture more detailed efficiencies of the techniques against the ODC defect categories. The experts had on average more than 20 years of related experience in space applications. The ODC Delphi survey used a modified Wideband Delphi process and went through two rigorous iterations [12]. The results are summarized separately for automated analysis, execution testing and tools, and peer reviews in [12].

The values represent the percentages of defects found by a given technique at each rating (sometimes termed “effectiveness”). The different relative efficiencies of the defect removal methods can be visualized, in terms of the general patterns between the methods and against the defect types within each method. For example, Figure 5 shows the results for Peer Reviews, where it is evident that Timing defects are more difficult to find than the other types.
ODC COQUALMO and Risk Minimization

Different methods for risk analysis and reduction have been performed in conjunction with ODC COQUALMO, which can produce optimal results in less time and allow for insights not available by humans alone. In [13] machine learning techniques were applied on the COQUALMO parameter tradespace to simulate development options and measure their effects on defects and costs, in order to best improve project outcomes. Another technique to reduce risks with the model is a strategic method of optimization. It generates optimal risk reduction strategies for defect removal for a given budget, and also computes the best order of activities [14].

An integration of ODC COQUALMO was also been prototyped with the DDP risk management tool [15], [16], which uses fault trees to represent the overall system’s dependencies on software functionality. These experiments to optimize quality processes are described in more detail in [7].

Dynamic Simulation Model

This section summarizes a continuous simulation model version using system dynamics [17] to evaluate the time-dependent effectiveness of different defect detection techniques against ODC defect categories. As a continuous model, it can be used for interactive training that demonstrates the effects of changes midstream or for tracking product quality through continuous updating with project actuals [17].

The model uses standard COCOMO factors for defect generation rates and the defect removal techniques for automated analysis, peer reviews and execution testing and tools. The model can be used for process improvement planning, or control and operational management during a project.

COQUALMO is traditionally a static model, which is a form not amenable to continuous updating because the parameters are constant over time. Its outputs are final cumulative quantities, no time trends are available, and there is no provision to handle the overlapping capabilities of defect detection techniques. The defect detection methods and the defect removal techniques are modeled in aggregate, so it is not possible to deduce how many are captured by which technique (except in the degenerate case where two of the three methods are zeroed out).

In this system dynamics extension to ODC COQUALMO, defect and generation rates are explicitly modeled over time with feedback relationships. It can provide continual updates of risk estimates based on project and code metrics. This model includes the effects of all defect detection efficiencies for the defect reduction techniques against each ODC defect type per Figure 4.

The defect dynamics are based on a Rayleigh curve defect model of generation and detection. The buildup parameters for each type of defect are calibrated for the estimated project schedule time, which may vary based on changing conditions during the project.

The defect detection efficiencies are modeled for each pairing of defect removal technique and ODC defect type. These are represented in graph functions for defect detection efficiency against the different ODC defect types.

Scenarios are demonstrated with dynamic responses to changing defect removal settings on different defect types. The variable impact to the different defect types can be visualized...
in the curves. An example for requirements consistency defects are in Figure 6, showing the perturbation from the defect removal changes. Another is the graph in Figure 7 from the simulation model showing representative dynamics for code timing defects, including the impact of changing the defect removal practices in the midst of the project at 18 months. At that time the setting for execution testing and tools goes high, and the timing defect detection curve responds to find more defects at a faster rate.

**Dynamic COQUALMO**

Dynamic COQUALMO was independently calibrated to software development projects at The Aerospace Corporation for estimating counts of residual defects [4]. The simulation also provides a basis for considering the effects of re-planning, project revision, and process improvements. In this section we show how two project defect profiles were replicated and a project retrospective was performed on one of them.

**Description**

In Dynamic COQUALMO the duration of each phase is specified. It then provides a decomposition of defect estimates
across phases so as to achieve a project defectivity profile. A Rayleigh curve generator was used for the defect patterns [4] that is more generalized and calibrated than the ODC COQUALMO dynamic version previously described.

To translate COQUALMO into a simulation model, it was necessary to decompose some of its components. First, peer reviews were separated into requirements, design, and code reviews by decomposing and allocating the peer review DRFs, under the constraint that they aggregate to those specified in COQUALMO. Also, COQUALMO assumes the same quality of practice for each type of quality activity, for example, all peer reviews at performed at a nominal level. Realistically, quality activities are performed to varying degrees within a project. The model accommodates this variation by weighting the quality levels of each activity and taking a weighted average.

Testing is also decomposed to distinguish software development testing from system testing, in which reliability is usually measured. This decomposition was also accomplished by weighting the two sets of testing defect removal factors under the constraint that they aggregate to COQUALMO values.

Due to the many differences in testing processes across software types and organization, the two testing phases are simply called Testing 1 and Testing 2, thereby allowing a user to define the differences between them and weight their effectiveness accordingly.

Dynamic COQUALMO has six defect flows, an inflow and an outflow for each artifact type: requirements, design, and code. Each of these flows has a single source, but multiple outflows, one for each quality-inducing activity as shown for requirements defects in Figure 8.

Model inputs include the following:
- Estimated job size in KSLOC.
- Settings for COCOMO II factors, including effort multipliers and scale factors.
- Estimated phase durations and degrees of phase concurrency (sum to the project duration).
- Usage profile of quality levels for each defect removal activity.
- Relative effectiveness estimates:
  - Effectiveness of requirements, design, and code reviews in finding requirements defects.
  - Effectiveness of design and code reviews in finding design defects.
  - Effectiveness of the two test phases in finding defects (requires definition of the differences between the two phases).

Results

It was found the default COQUALMO values for nominal defects introduced were high. Values between .5 Defects/KSLOC (Project C requirements) and 6.1 Defects/KSLOC (Project C code) were used to produce the modeled curves such as in Figure 9, from [18] loosely constrained by limited knowledge of the ratios of the actual defect sources.

For the retrospective studies we looked at the factors affecting defect introduction and used them to perform a cost benefit analysis. The cost of improving the factors was estimated and Dynamic COQUALMO provided a measure of the benefits in terms of reduced numbers of defects. These were plotted and those with the highest benefit against cost were selected for action in the next project.

A combination of four cost factors was selected as the most cost effective and beneficial set of improvements [4]. This set, plus Requirements Review, was modeled as the Improved Project A. Figure 9 displays the modeled defect profile for the actual project as well as for the two alternate cases. The Dynamic COQUALMO modeling suggested two conclusions:

1. Revising the project earlier would not have made a large difference in the final product quality, but would have saved finding and fixing about 600 defects.

2. Starting the project with the selected set of improvements, would have both reduced the effort of finding and fixing defects and improved the final product quality. The number of defects introduced for the Improved Project would have been about...
1000 less than in the actual project and the residual in the final product would have been less than half that of the actual final product.

Conclusions and Future Work

Software estimation models can and should play an important role in facilitating the right balance of activities to meet quality goals. By predicting software defect introduction and removal rates, COQUALMO variants are useful for identifying appropriate defect reduction strategies.

The extension for ODC defect types provides more granular insight into defect profiles and their impacts to specific risks. We have shown that the ODC COQUALMO model can be used in different ways to reason about and optimize quality processes.

The use of value-neutral software engineering methods often causes software projects to expend significant amounts of scarce resources on activities with negative returns on investment. The use of models and risk minimization techniques can be used to optimize the scarce resources. Results of experiments combining ODC COQUALMO with various methods show they can produce optimal results in less time and allow for insights not available by humans alone.

We will continue to integrate the ODC COQUALMO model with complementary techniques to support risk management practices, and to compare their performances.

Software development projects seem to have characteristic defect discovery profiles. Dynamic COQUALMO can replicate a discovery profile and, by inference, produce a realistic defect profiles for use in managing quality effort in future projects.

Organizations should consider local calibration, because the usage profiles of default COQUALMO may require adjustment.

Using Dynamic COQUALMO in a retrospective cost-benefit analysis for product quality provides credible results.

One of the next steps for Dynamic COQUALMO will be to use the model prospectively to assess project options as part of either initial planning or planning a revision.

We are analyzing more empirical data for further calibrations and improvements. We are continuing to gather data from case studies at the Naval Postgraduate School, The Aerospace Corporations, and CSSE industrial affiliates. There are also affiliates in commercial domains undertaking ODC defect analysis for which specialized calibrations are being done.

With more comprehensive data the quality model will be further improved, tailored for government projects and other organizations, and integrated with complementary methods for value-based decision making on quality strategies.

Acknowledgments

This work was partially supported by NASA AMES Cooperative Agreement No. NNA06CB29A for the project Software Risk Advisory Tools. The authors wish to thank the CSEE affiliates who contributed data, the ODC Delphi participants, Dr. Mike Lowry and John Powell at NASA. At the Aerospace Corporation this work was conducted with Douglas Buettner and Myron Hecht.

References


About the Authors

Raymond Madachy is an Associate Professor in the Systems Engineering Department at the Naval Postgraduate School. His research interests include systems and software cost/quality estimation and measurement; process simulation; risk management; integrating systems engineering and software engineering disciplines; and integrating empirical-based research with process simulation.

Barry Boehm is the TRW Professor of Software Engineering and Director Emeritus of the USC Center for Systems and Software Engineering; and Director of Research at the Systems Engineering Research Center for the DoD. He is an AIAA Fellow, an ACM Fellow, an IEEE Fellow, and a member of the National Academy of Engineering.

Dan Houston is a Senior Engineering Specialist with The Aerospace Corporation where he researches and develops software process simulation models for providing quantitative decision support to system program offices. Prior to joining Aerospace, he spent 16 years at Honeywell as a software developer, Six Sigma Black Belt, and software metrician. He is a Senior Member of both the IEEE and the American Society for Quality. His publications address software development economics and process improvement, software process simulation, and statistical methods in software engineering.

Authors Contact Information

Email: Raymond Madachy: rjmadach@nps.edu
Email: Barry Boehm: boehm@usc.edu
Email: Dan Houston: daniel.x.houston@aero.org
Two Great Reliability Solutions from the RIAC & DACS.

**System Reliability Toolkit**

The RIAC/DACS System Reliability Toolkit provides technical guidance in all aspects of system reliability, allowing the user to understand and implement techniques to ensure that system and product designs exhibit satisfactory hardware, software and human reliability, and to minimize the inherent risks associated with deficiencies in system reliability.

To purchase, please contact:
The Reliability Information Analysis Center
6000 Flanagan Road
Suite 3
Utica, NY 13502-1348
1-877-363-RIAC
http://theRIAC.org

---

**The DACS Software Reliability Sourcebook**

This first edition of the DACS Software Reliability Sourcebook provides a concise resource for information about the practical application of software reliability technology and techniques. The Sourcebook is divided into nine major sections, plus seven supporting appendices.

To purchase, please contact:
The Data & Analysis Center for Software
775 Daedalian Drive
Rome, NY 13441-4909
1-800-214-7921
http://thedacs.com
Hey- C and C++ Can Be Used In Safety Critical Applications Too!

COMPUTER PROGRAMMING LANGUAGES CAN HAVE AN IMPACT ON SOFTWARE SAFETY AND RELIABILITY. HERE, THE C AND C++ LANGUAGES ARE BEING USED SUCCESSFULLY FOR THE NEW F-35 AIRCRAFT FOR SAFETY-CRITICAL APPLICATIONS.

by John H. Robb

Recently, a few industry articles have inferred or directly implied that C and C++ is not a good choice for safety critical applications. Software engineers grow fond of the language that they have spent many hours getting to know, tooling, and finally melding into an elegant way of expressing the application that they have designed. In some way then these articles are like someone trash talking a close relative and the record has to be set straight.

Lockheed Martin Aeronautics in Fort Worth has been using digital computers in safety critical applications for almost 30 years now. Being that most people would mark the start of the programming era around the late 1940s - these applications then have been used safely for half of man’s history in programming computers. Naturally then, there has been an evolution of technologies, tools, and languages used during this development. During this evolution, a pattern has been used by leading safety critical software development foundries to ensure that the chosen programming language supports the deployment of highly reliable software.

We in the software engineering industry owe it to each other to keep software development out of the dark realm where certain phrases evoke fear or hushed tones especially if there not is much substance behind them. As in the classic “Wizard of Oz” movie – we not only need to see the man behind the curtain but what is motivating him to speak through the towers of flame and smoke. Is the wizard speaking from a viewpoint of someone who lives in a castle or one who actively works in the busy village marketplace on a daily basis? This article shows that many languages have been successfully utilized in safety critical applications and the basic elements required to make a language work in this domain are persistence, determination, and good software engineering discipline. In fact, this article is a long overdue rebuttal to several articles that appeared in the August 2006 edition of Crosstalk magazine.

Okay, so case in point. In the August 2006 edition of Crosstalk magazine, several articles casted doubt, either directly or indirectly on the use of C and/or C++ in safety critical applications. … Ada’s focus on safety is in strong contrast with certain other languages, where the attitude might be expressed as give programmers very sharp tools and then get out of their way... [1] Another article said while this comes as no great shock to programmers that C and C++ are not considered safety-critical languages… [2] A third article said high-integrity software can, in principle, be written in any computer language, but the effort will be simplified by choosing an appropriate language – one that is designed for reliability and safety with expressiveness to capture a broad range of applications including real-time systems. Both the original Ada language and the Ada 95 revision meet these requirements, and Ada 2005 has continued in this vein… [3]

Now that the trumpets announcing the Ada 2005 emergence have died down, what is a view from developers and practitioners who utilize a wide variety of languages to accomplish delivery of safety critical systems? This is a market where developers can ill afford to take large risks as the main emphasis is on developing a working and reliable product in a very demanding safety critical environment. The utilization of the latest technologies then is not an endeavor unto itself but a supporting function to enable the delivery of highly reliable safety critical systems that play key roles in supporting strategic interests.

To understand our more recent activities in the use of languages in safety critical systems, we need to understand that Lockheed Martin Aeronautics in Fort Worth has been using digital computers in safety critical applications for almost 30 years now and evolving this approach through the use of many programming languages. We are then not a new comer to the market, but we do not hold a patent on this market either – we are simply one of many in the industry to develop safety critical systems. Our long history in this shows that we are interested in developing reliable systems where utilization of leading edge languages is, at best, a secondary consideration. In order to understand our approach with the use of C++ in safety critical applications, we need to see how we have steadily, over
time developed ultra-reliable products and infused different computing technologies in them.

During the late 1970s through the early 1980s assembly language was used to program the digital flight control computers for the Advanced Technology Fighter Integration (AFTI) F-16 – a triple redundant fly-by-wire system using the BDX-930 16-bit fixed point arithmetic processor based on the AMD-2901A bit slice microprocessor. This aircraft was successfully retired to the Wright-Patterson AFB museum, Ohio on Feb 11, 2001 after completing 756 flights for ten different flight test programs during a 19-year period.

As a note, the F-16 was the world’s first aircraft to be aerodynamically unstable by design “relaxed static stability.” Relaxed static stability gives an aircraft unparalleled maneuverability but also means that the computer/software must operate correctly one hundred percent of the time as the aircraft is on the verge of departure (loss of controlled flight). The system architecture used in the AFTI/F-16 and modeled in many fighter aircraft thereafter is that common software is running in each of the redundant hardware branches of the flight control computer; thus, any common mode software bug could cause the aircraft to depart. Proper software operation is essential – here it really is just like Dad used to say – “there is no second chance.”

During the mid-1980s the Block 40 F-16 was implemented using a quadruple redundant digital computer using the Mil-Std 1750A processor architecture. Technology advancements made beyond the AFTI/F-16 were the utilization of floating point for control law computations and the use of JOVIAL/J73B. The use of floating point arithmetic greatly improved the productivity of the software engineering team and the science of fixed-point scaling was forever lost on the world of programming (with many software engineers celebrating that demise by the way)! The use of a Higher Order Language (HOL) meant that time would need to be spent analyzing language structures and the code generated with the eventual triumphant emergence of the programming standards document. In truth, the programming standards document existed for each of the languages used starting with even good old assembly language. As the complexity and richness of the language increased the size of the document also increased. JOVIAL/J73 was a simple HOL, thus there were few more limitations in it than the BDX-930 assembly language. The restrictions against certain language features were manually checked during code inspection.

As a side note for the history buffs, even with the deployment of the F-16 J73 flight control computer there were those even in the aerospace industry that were absolutely convinced that this could never be accomplished due to the uncertainties in dealing with compiler generated code for use in safety critical applications. There were others who were convinced that the use of floating point in control equations would cause severe instabilities. Despite all of that early hysteria, the F-16 J73 flight control computer has enjoyed an extraordinary record of reliability and success in the field. The F-111 digital flight control system, the F-16 Block 40 production system and other F-16 derivative programs contained J73 flight control computers targeted to a Mil-Std 1750A architecture. With these systems literally hundreds of thousands of flight hours have been flown around the world and in combat.

From the mid to late 1980s, work was being performed on the YF-22 flight control computers using a new language - Ada (83). The computers were using the Mil-Std-1750A architecture but much work was done to understand language structures and the code generated to develop the Ada/83 programming standards. The restrictions against the Ada programming language were enforced through a combination of compiler switches/pragmas and manual checks performed during code inspections. The Ada language because of its inherent robustness required only a few dozen checks for even the largest and most complicated of systems. The production F-22 utilizes several safety critical systems that are programmed in the Ada/83 language targeted to a Mil-Std-1750A instruction set architecture. Several hundred F-22’s have been manufactured and deployed representing several tens of thousands of hours use for these systems.

As promised, now to the latest developments. With the F-35 Joint Strike Fighter both C and C++ have been used in the safety critical systems developed by the team of Lockheed Martin Aeronautics, Northrop Grumman Aerospace, and BAE Systems. This is also true for the F-35 supplier team. Ada was seen as the technically superior and more robust language, but concern over the ability to successfully staff the software engineers required to develop the massive amounts of safety critical software caused the F-35 team to carefully look and finally to choose C and C++ for the implementation of safety critical software. Primary factors in this choice were training availability, tool support, and processor support. Another key factor was type casting, not as a language feature, but as a hiring feature. Many of the university students simply refused to work Ada as it was not seen as a marketable experience base. When all factors were considered, C++ and C emerged as the languages of choice (these depending on the processor chosen). This evaluation was not only made by each of the
F-35 Prime team members, but across the entire F-35 supplier base where safety critical products were also implemented. In almost every case, the same decision to use C or C++ was made except when a large amount of reuse was possible from previous (F-22) developments. By our latest counts, we have developed approximately 1.59 million lines of code for safety critical applications for the F-35 as shown in the figure. Another important point here is that the safety critical code developed for the F-35 was an international effort with teams from the United Kingdom developing safety critical code in C and C++ for various F-35 systems as well.

![Safety Critical Language Use On F-35](image)

Although many articles have derided the use of C and especially C++ in safety critical applications the good news is that it has now been successfully utilized in these domains. Ironically enough, some of the software engineers that were working the original AFTI/F-16 flight control system helped to ensure safe deployment of the feature rich C and C++ languages almost 30 years later.

Concerns with dynamic binding, polymorphism, and inheritance mechanisms [4] have been dealt with in the coding standards which are publicly available at [http://www.jsf.mil/downloads/documents/JSF_AV_C++_Coding_Standards__Rev_C.doc](http://www.jsf.mil/downloads/documents/JSF_AV_C++_Coding_Standards__Rev_C.doc). How was this achieved? MISRA-C was used as the basis for the C applications and a coding standards was developed with the assistance of Bjarne Stroustrup, original author of the C++ language. For both C and C++ Static Code Analysis (SCA) tools are used to ensure that restricted features are not utilized. Arguments about the lack of reliability in either C or C++ are addressed by programming standards restrictions and SCA checks. In truth, this approach is probably more consistent and robust than the manual checks used for previous development efforts including Ada. The typical software engineer develops the software, runs the SCA tool, makes fixes, and then submits the code for inspection – thus there is little difference between developing Ada code where the compiler makes automatic checks and developing C/C++ code where the SCA tool checks for compliance immediately after a successful compilation. The only real difference is the amount of up-front time spent investigating language features and developing the necessary safeguards. The amount of time to do this for C++ for example was more than Ada but not by a substantial amount since we had language expertise available.

Our secondary line of defense is in unit testing where full structural coverage is required either at the object code level (for the highest reliability software) or at the source level (safety involved software). Unit testing especially at the object code level ensures that even compiler generated branches that aren’t apparent in the source code are covered during test. Object code coverage has led to the identification of two errors in compiler generated code, where work-arounds have been documented and put in practice to avoid these from being generated by the compiler. Compiler generated errors are something that we have been dealing with in the safety critical arena ever since we made the transition to HOL so this does not reflect on either C or C++ and in fact the occurrence of only two speaks well of this environment.

Unit testing to the object code level has been a regular practice for safety critical software not only in the military environment but also the commercial where MC/DC testing achieves a similar level of coverage for Do-178 products. Between these two lines of defense (SCA checks and unit testing) and many levels of system test that follow, we could see early in the development cycle that C++ was not going to wreck havoc on the development of the safety critical/safety involved code.

Measures associated with defect densities, phase containment, and productivity for both the C and C++ code all point to quality consistent with legacy efforts if not better for these safety critical applications. Does this make sense? Yes, remember that most of the language issues are dealt with by developing a set of programming standards that are meant to address safety issues. The power of this approach comes not in the development of the programming standards but in the enforcement of it through the SCA tools.
used. Here the paradigm shifts toward the software engineer using the SCA tools something like a pair programmer, at least for the syntax (language). After a few runs through the SCA tools, the software programmer becomes familiar with what constructs are allowed. He or she has code snippets that have passed SCA checkout/code inspections and can be used as a palette of preferred approaches. The learning curve is shortened and productivity is impacted in a positive way because programming errors are detected in-phase before they can become faults or failures. Given this approach and the fact that we now use more automated tools than before it is any wonder that our productivity and error rates are at least as low as previous efforts and the languages used?

The reliability of these safety critical systems has been phenomenal with no failures attributed to language issues. Remember that most of these systems are utilized on relaxed static stability aircraft with common software running in redundant computers, meaning that software errors can be catastrophic and visible. Okay, so what is the secret to successfully deploying ultra-reliable software? Software engineering discipline, but that is hardly a secret. The keys here are to study the programming language and to develop a list of restrictions, to provide a method of enforcement (preferably automated) and to perform code coverage during unit testing. One major vote of confidence in this approach is that because of processor obsolescence many of the F-22 safety critical systems are embarking on a path to recode them in C from the Ada 83 language because of many of the same issues we dealt with on the F-35 program.

**Conclusions**

We were motivated to address both C and C++ on the F-35 to address primarily staffing concerns associated with the relatively low demand for Ada programmers and the lack of formal Ada training in both the corporate and academic environments. Many flagship universities that were once offering training classes in Ada have long ago ceased to do so. This is a disappointment to all of us because Ada was and is clearly the superior technical language. This article though is not about the virtues of one language over another - we are not programmers, we are software engineers and ours is the tradeoff between factors such as cost, schedule, technology, and reliability of our delivered products to provide the best overall system. We work in an industry that is ever evolving and changing – we earn our title as software engineers because we accept that we must navigate a river that requires our constant attention as its speed and direction ever shifts.

**References**


**About the Author:**

John H. Robb is the Senior Manager of the F-35 Joint Strike Fighter Air Vehicle Software team at Lockheed Martin Aeronautics Fort Worth. His team is responsible for the definition, coordination, and support of processes and tools used by over 50 different software development teams across the F-35 Joint Strike Fighter (JSF) program. His team is also responsible for leading the Software Quantitative Management activities for the program. The total size of air vehicle software for the F-35 will grow to approximately 19 million Source Lines of Code (SLOC) by 2011.

Mr. Robb has a BS in Electrical Engineering, a BS in Computer Science, both from the University of Texas at Austin, an MS in Computer Science from the University of Texas at Arlington and is working toward a Doctor of Engineering in Software Engineering at the Southern Methodist University. He was a regular lecturer in Software Engineering classes at the University of Texas at Austin, made presentations at SSTC, and is a Certified Project Management Professional (PMI, Inc), a Certified Software Project Manager (QAI), a Certified Software Test Engineer (QAI) and a Certified Software Quality Engineer (ASQ).

**Author Contact Information:**

Email Address: John. H. Robb: john.h.robb@lmco.com
The STN is a theme-based quarterly journal. In the past DACS has typically solicited specific authors to participate in developing each theme, but we recognize that it is not possible for us to know about all the experts, programs, and work being done and we may be missing some important contributions. In 2009 DACS adopted a policy of accepting articles submitted by the software professional community for consideration.

DACS will review articles and assist candidate authors in creating the final draft if the article is selected for publication. Note that DACS does not pay for articles published. Note also that submittal of an article constitutes a transfer of ownership from the author to DACS with DACS holding the copyright.

Although the STN is theme-based, we do not limit the content of the issue strictly to that theme. If you submit an article that DACS deems to be worthy of sharing with the community, DACS will find a way to get it published. However, we cannot guarantee publication within a fixed time frame in that situation. Consult the theme selection page and the Author Guidelines located on the STN web site (see https://www.softwaretechnews.com/) for further details.

To submit material (or ask questions) contact news-editor@thedacs.com

Recent and upcoming themes include:

- Earned Value
- Software Testing
- Project Management
- Model Driven Development
- Software Quality and Reliability
- Cyber Security
The first 50 people to send in a completed survey will receive a FREE DoD/IT Acronym CD from the DACS.

This valuable CD-ROM contains over 9,000 Department of Defense and Information Technology acronyms. There are hundreds of acronym lists available but none are as well done as this CD AND specifically targeted towards DoD and Information Technology. This unique-shaped CD-ROM plays in your computer's regular, hub-mounted, CD drive. You'll use this great resource over and over again. It's FREE, just for filling out our brief survey on the next page!

Data & Analysis Center for Software (DACS)

http://thedacs.com
1. Which volume of the Software Tech News did you receive?  

STN Vol 13, No. 1

2. When did you receive the newsletter? (month/year)

3. How satisfied were you with the CONTENT of the newsletter? (Article Quality)
   - Very Satisfied
   - Satisfied
   - Neither Satisfied nor Dissatisfied
   - Dissatisfied
   - Very Dissatisfied

4. How satisfied were you with the APPEARANCE of the newsletter?
   - Very Satisfied
   - Satisfied
   - Neither Satisfied nor Dissatisfied
   - Dissatisfied
   - Very Dissatisfied

5. How satisfied were you with the OVERALL QUALITY of the newsletter?
   - Very Satisfied
   - Satisfied
   - Neither Satisfied nor Dissatisfied
   - Dissatisfied
   - Very Dissatisfied

6. How satisfied were you with the ACCURACY of the address on the newsletter?
   - Very Satisfied
   - Satisfied
   - Neither Satisfied nor Dissatisfied
   - Dissatisfied
   - Very Dissatisfied

7. Approximately how much of the newsletter do you read?
   - The entire issue
   - Most of the content
   - About half the content
   - Briefly Skimmed
   - Didn’t Read

8. Would you read this newsletter in an E-mail newsletter format?
   - Definitely
   - Probably
   - Not Sure
   - Probably Not
   - Definitely Not

9. How did you request the product or service?
   - Phone Call
   - E-mail
   - DACS Website
   - Subscription Form
   - Other ______________________

10. Would you recommend the Software Tech News to a colleague?
    - Definitely
    - Probably
    - Not Sure
    - Probably Not
    - Definitely Not

11. What topics would you like to see this newsletter devoted to?  
    _____________________________________________________________________
    _____________________________________________________________________
    Comments (optional)  _____________________________________________________________________
    _____________________________________________________________________

REGISTER FOR THE FIRST TIME | UPDATE CURRENT SUBSCRIPTION

Name:  ___________________________________________  Position/Title:  __________________________
Organization:  ______________________________________  Office Symbol:  __________________________
Address:  __________________________________________  City:  _________________________________
State:  ___________________  Zip:  _______________  Country:  ______________________________________
Telephone:  _____  -  ______ -  _______  Fax:  _____  -  ______  -  _______  Email:  _______________________
Functional Role:  __________________________________________________________________________
Organization Type:  □ Air Force □ Army □ Navy □ Other DoD  __________________________
                   □ Commercial □ Non-Profit □ Non-US □ US Government □ FFR&D □ Other  ____________

*Note: You must give us your address to receive the CD.
The Software Tech News is now accepting advertisements for future newsletters. In addition to being seen by the thousands of people who subscribe to a paper copy, an electronic version is available at the DACS website, exposing your product, organization, or service to hundreds of thousands of additional eyes every month.

For rates and layout information contact: news-editor@thedacs.com

Distribution Statement:
Unclassified and Unlimited

DACS
P.O. Box 1400
Rome, NY 13442-1400

Phone: 800-214-7921
Fax: 315-838-7130
E-mail: dacs@thedacs.com
URL: http://www.thedacs.com/
## STN Vol. 13, No. 1 April 2010 Software Quality, Reliability, and Error Prediction

### IN THIS ISSUE

<table>
<thead>
<tr>
<th>Topic</th>
<th>Authors</th>
<th>Pages</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Tech Views</strong></td>
<td>by Dan Ferens, Co-Editor and Ellen Walker, Editor, STN</td>
<td>3-5</td>
</tr>
<tr>
<td><strong>Identifying Your Organizations True Best Practices</strong></td>
<td>by David Herron</td>
<td>7-9</td>
</tr>
<tr>
<td><strong>Software Quality and Software Economics</strong></td>
<td>by Capers Jones</td>
<td>10-14</td>
</tr>
<tr>
<td><strong>Predicting Software Quality</strong></td>
<td>by Donald M. Beckett and Douglas T. Putnam</td>
<td>16-19</td>
</tr>
<tr>
<td><strong>Applying Agile Practices to Improve Software Quality</strong></td>
<td>by Arlene Minkiewicz</td>
<td>20-25</td>
</tr>
<tr>
<td><strong>Modeling Software Defect Dynamics</strong></td>
<td>by Raymond Madachy, Barry Boehm, and Dan Houston</td>
<td>26-34</td>
</tr>
<tr>
<td><strong>Hey-C and C++ Can Be Used In Safety Critical Applications Too!</strong></td>
<td>by John H. Robb</td>
<td>36-39</td>
</tr>
</tbody>
</table>