

# CROSSTALK

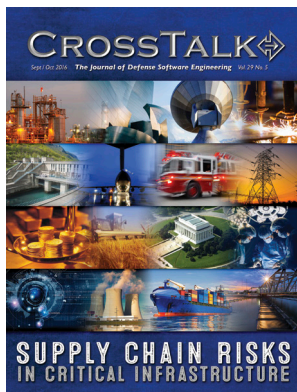
Sept / Oct 2016 *The Journal of Defense Software Engineering* Vol. 29 No. 5



## SUPPLY CHAIN RISKS IN CRITICAL INFRASTRUCTURE

## Departments

- 3 From the Sponsor
- 32 Upcoming Events
- 34 BackTalk



Cover Design by  
Kent Bingham

# Supply Chain Risks in Critical Infrastructure

**4 Minimizing the Risk of Litigation: Problems Noted in Breach of Contract Litigation**  
Much of the software literature deals with "best practices." This article concentrates on "worst practices," or the factors that most often lead to software failure and litigation.

By Capers Jones

**11 The Measurement of Software Maintenance and Sustainment: Positive Influences and Unintended Consequences**  
It is well understood that measurement influences behavior. However, when applied to complex human systems that are involved in software maintenance and sustainment, the inappropriate use of metrics can have negative unintended consequences.

By Rob Ashmore and Mike Standish

**16 Towards a Better Software Project Manager**  
This article examines the role software project managers play in the development of software systems, explains why this has been largely overlooked, and proposes that the practice of software project management has always been and continues to be critical to the success of software projects today and into the future.

by Lawrence Peters

**23 Legal Liability for Bad Software**  
This article focuses on lawsuits as a recourse for purchasers of defective COTS software — particularly safety-critical COTS software and software-controlled systems.

by Karen Goertzel

**29 Durability Challenges in Software Engineering**  
To be able to develop durable software cost-effectively, developers must investigate the connection between durability characteristics and software.

By Rajeev Kumar, Suhel Khan, and Raees Khan

# CROSSTALK

**NAVAIR** Jeff Schwalb  
**DHS** Peter Fonash  
**309 SMXG** Kelly Capener  
**76 SMXG** Mike Jennings

**Publisher** Justin T. Hill  
**Article Coordinator** Heather Giacalone  
**Managing Director** David Erickson  
**Technical Program Lead** Thayne M. Hill  
**Managing Editor** Mary Harper  
**Associate Editor** Colin Kelly  
**Senior Art Director** Kevin Kiernan  
**Art Director** Mary Harper

**Phone** 801-777-9828

**E-mail** Crosstalk.Articles@hill.af.mil

**Crosstalk Online** [www.crosstalkonline.org](http://www.crosstalkonline.org)

**CROSSTALK, The Journal of Defense Software Engineering** is co-sponsored by the U.S. Navy (USN); U.S. Air Force (USAF); and the U.S. Department of Homeland Security (DHS). USN co-sponsor: Naval Air Systems Command. USAF co-sponsors: Ogden-ALC 309 SMXG and Tinker-ALC 76 SMXG. DHS co-sponsor: Office of Cybersecurity and Communications in the National Protection and Programs Directorate.

**The USAF Software Technology Support Center (STSC)** is the publisher of **CROSSTALK** providing both editorial oversight and technical review of the journal. **CROSSTALK'S** mission is to encourage the engineering development of software to improve the reliability, sustainability, and responsiveness of our warfighting capability.

**Subscriptions:** Visit [www.crosstalkonline.org/subscribe](http://www.crosstalkonline.org/subscribe) to receive an e-mail notification when each new issue is published online or to subscribe to an RSS notification feed.

**Article Submissions:** We welcome articles of interest to the defense software community. Articles must be approved by the **CROSSTALK** editorial board prior to publication. Please follow the Author Guidelines, available at [www.crosstalkonline.org/submission-guidelines](http://www.crosstalkonline.org/submission-guidelines). **CROSSTALK** does not pay for submissions. Published articles remain the property of the authors and may be submitted to other publications. Security agency releases, clearances, and public affairs office approvals are the sole responsibility of the authors and their organizations.

**Reprints:** Permission to reprint or post articles must be requested from the author or the copyright holder and coordinated with **CROSSTALK**.

**Trademarks and Endorsements:** **CROSSTALK** is an authorized publication for members of the DoD. Contents of **CROSSTALK** are not necessarily the official views of, or endorsed by, the U.S. government, the DoD, the co-sponsors, or the STSC. All product names referenced in this issue are trademarks of their companies.

### CROSSTALK Online Services:

For questions or concerns about [crosstalkonline.org](http://crosstalkonline.org) web content or functionality contact the **CROSSTALK** webmaster at 801-417-3000 or [webmaster@luminpublishing.com](mailto:webmaster@luminpublishing.com).

**Back Issues Available:** Please phone or e-mail us to see if back issues are available free of charge.

**CROSSTALK** is published six times a year by the U.S. Air Force STSC in concert with Lumin Publishing [luminpublishing.com](http://luminpublishing.com). ISSN 2160-1577 (print); ISSN 2160-1593 (online)

# CROSSTALK would like to thank DHS for sponsoring this issue.

*This sponsor's note is from a discussion with **Joe Jarzombek** sharing his perspectives on advancements in software supply chain assurance relative to enterprise risk management and user safety and security.*

With the Internet of Things (IoT) enabling a cyber-reliant world of connected devices and products, many enterprises and informed buyers have begun to demand that those products be evaluated from a perspective of enterprise resilience and consumer protection, especially since much of IoT is enabled and controlled by third-party open source software. Safe and secure use of cyber assets places responsibilities on operators and users; yet realization of cyber assurance requires more focus on cyber 'hygiene' in the supply chain. It requires network-connectable devices to be developed and deployed with cyber-physical security and safety in mind. IoT devices need to be patchable to be responsive to a changing threat environment, and before being deployed, they should be independently evaluated not to possess exploitable weaknesses, known vulnerabilities, and malware.

The U.S. Digital Service and General Services Administration's 18F have been big proponents of open source. The Office of Management and Budget (OMB) is launching "Code.gov" to give agencies more tools and best practices to help implement the 8 Aug 2016 open source policy. Code.gov will eventually serve as an inventory of all of agencies' open and accessible code. Hopefully, the open source software (OSS) that will be made accessible to all agencies will be evaluated not to possess exploitable weaknesses, known vulnerabilities, and malware. To be of real utility, the OSS will need a bill of materials such that those reusing the OSS assets can operationally respond to changing threat environments in which software has become a favorite vector of attack.

National Institute of Standards and Technology (NIST) Special Pub 800-161 provides guidance and security controls for Supply Chain Risk Management (SCRM), and NIST SP 800-160 Systems Security Engineering, Appendix J 'Software Security and Assurance' provides controls, consistent with SP 800-53, that include architecture choices, design choices, added security functions, activities and processes, code assessments, design reviews, and various types of testing. NIST SPs are provided for voluntary adoption; yet more of those controls are now specified in government policies, directives, and contracts; thus making those controls mandatory for procured and deployed products, systems and services.

Cybersecurity professionals within DoD and the intelligence community already understand these controls are not optional, and that they need skills to accomplish or oversee the implementation of these controls. The Committee for National Security Systems (CNSS) Instruction 1253 and Director of Central Intelligence Directive (DCID) 6/3 have provided precursors for the

transition to Intelligence Community Directive (ICD) 503, "Intelligence Community Information Technology Systems Security Risk Management, Certification and Accreditation." With the transition from DCID 6/3 to the Risk Management Framework (RMF), cybersecurity practitioners need to understand available options in terms of relevant security control families, and common/hybrid/system-specific security controls; tailoring and the identification of control enhancements. Practitioners need to implement security controls in the System Development Life Cycle (SDLC) and through operational continuous monitoring.

The unfortunate reality has been that many cyber assets have been procured without requisite security controls and without a bill of materials to indicate the composition or content of third-party open source software that enables and controls assets vital in critical infrastructure operations. That makes it extremely difficult for cybersecurity professionals to secure those exploitable assets after procurement. As a minimum, cybersecurity professional need products that have been evaluated to determine they do not have malware, known vulnerabilities and software security weaknesses. This includes relevant test and certification activities that focus on mitigating exploitable weaknesses that could have otherwise been vectors of future zero-day exploits, if not mitigated prior to use. IoT trust can be enabled with verification and validation activities focused on quality, safety, and security of devices in the context of the environments in which they would be used.

Fortunately for consumers, many white-hat researchers and test labs are equipped with tools and methods that provide third-party analysis of IoT products relative to cyber-physical security and safety. Underwriters Laboratories (UL) launched its Cybersecurity Assurance Program in April 2016 (addressing the needs stated above),

and UL has already started putting its certification mark on network-connectable systems and IoT products, such as industrial control systems. These efforts provide better synergy between cyber assurance and cyber insurance since both provide a focus for mitigating residual risk.

Many tools are available to detect and aid in mitigating known vulnerabilities (CVEs), exploitable weaknesses (CWEs) and malware. The fact that new products are still being released with known vulnerabilities and weaknesses causes many to question the cyber hygiene of those in the supply chain. Why are users left to find those exploitable flaws and mitigate or patch those products when developers could/should have mitigated those risks prior to delivery or as part of a product release update? Why do suppliers not provide a bill of materials for third party OSS that would enable using enterprises to identify newly reported CVEs? The answers seem to revolve around the fact that suppliers have no liability associated with products tainted with malware, known vulnerabilities and weaknesses; so what are their motivations to address those flaws in the supply chain prior to putting users at risk?

'Enterprise/consumer demand' specified in terms of procurement language in contracts is making a difference. The DoD Software Assurance Community of Practice (CoP) has drafted sample contract language, and the "2016 Cyber Insurance Buying Guide," published by the Financial Services Sector Coordinating Council for Critical Infrastructure Protection and Homeland Security, provides supply chain cyber assurance procurement requirements. More sectors are developing exemplar procurement language; pushing demand signals for properly tested products with bill of materials to enable responses to changing threat environments.

It seems supply chain assurance can best be achieved with consumer demand via contracts and through adoption of independent evaluation and certification of IoT products because realization of risks attributable to known vulnerabilities and weaknesses are primarily on the use side; not the supply side. Cyber insurance is leveraging this relationship with cybersecurity assurance practices for IoT products.



**Joe Jarzombek** has been involved with CrossTalk for over two decades. He previously served Director for Software and Supply Chain Assurance in the US Department of Homeland Security. Having retired from both DHS and DoD, he is now serving as Global Manager for Software Supply Chain Management in the Synopsys Software Integrity Group. He continues to collaborate on addressing software assurance, supply chain risk management, and security automation initiatives to enable scalable information sharing among organizations and security researchers. He freely shares SwA/SCRM best practices and resources, such as procurement language. He can be reached via [joe.jarzombek \[at\] synopsys.com](mailto:joe.jarzombek@synopsys.com).

# Minimizing the Risk of Litigation

## Problems Noted in Breach of Contract Litigation

Capers Jones, Vice President and CTO, Namcook Analytics LLC

**Abstract** While working as an expert witness in a number of lawsuits where large software projects were canceled or did not operate correctly when deployed, the author has observed six major problems that occur repeatedly: 1 – Accurate estimates are not produced or are overruled. 2 – Accurate estimates are not supported by defensible benchmarks. 3 – Requirements changes are not handled effectively. 4 – Quality control is deficient. 5 – Progress tracking fails to alert higher management to the seriousness of issues. 6 – Contracts omit important topics such as change control and quality, or include hazardous terms.

### Introduction

Much of the software literature deals with “best practices.” This article concentrates on “worst practices,” or the factors that most often lead to software failure and litigation.

For the purposes of this article, software “failures” are defined as software projects that have any of these attributes:

- 1. Termination of the project due to cost or schedule overruns.
- 2. Schedule or cost overruns in excess of 50 percent of initial estimates.
- 3. Applications that, upon deployment, fail to operate safely.
- 4. Law suits brought by clients for contractual noncompliance.

Although there are many factors associated with schedule delays and project cancellations, the failures that end up in court always seem to have six major deficiencies:

- 1. Accurate estimates were either not prepared or were rejected.
- 2. Accurate estimates were not supported by objective benchmarks.
- 3. Change control was not handled effectively.
- 4. Quality control was inadequate.
- 5. Progress tracking did not reveal the true status of the project.
- 6. The contracts omitted key topics, such as quality and out-of-scope changes

Readers are urged to discuss outsource agreements with their attorneys. This paper is based on observations of actual cases, but the author is not an attorney, and the paper is not legal advice. It is advice about how software projects might be improved to lower the probability of litigation occurring.

To begin the discussion of defenses against software litigation, let us consider the normal outcomes of 15 types of U.S. software projects. Table 1 shows the percentage of projects that are likely to be on time, late, or canceled without being completed at all due to excessive cost, schedule overruns or poor quality:

As can be seen, schedule delays and canceled projects are distressingly common among all forms of software this year. This explains why software is viewed by most CEOs as the least competent and least professional form of engineering in the current business world.

Note that the data in Table 1 is from benchmark and assessment studies carried out by the author and his colleagues between 1984 and 2016. Unfortunately, recent data since 2010 is not much better than older data before 1990. This is due to several factors, including the following: 1) very poor measurement practices and distressingly bad metrics, which prevent improvements from being widely known, and 2) software that continues to use custom designs and manual coding, both of which are intrinsically expensive and error prone. Until the software industry adopts modern manufacturing concepts that utilize standard reusable components instead of custom-built artifacts, software can never be truly cost effective.

Let us consider each of these six topics in turn.

### Problem 1: Estimating Errors and Estimate Rejection

Although cost estimating is difficult, there are a number of commercial software parametric cost estimating tools that do a capable job. COCOMO III, CostXpert, ExcelerPlan, KnowledgePlan, True Price, SEER, SLIM and the author's Software Risk Master™ (SRM) are examples available in the United States.

Despite the proven accuracy of parametric estimation tools and their widespread availability, as of 2016, less than 20 percent of the author's clients used any formal estimating methods at all when we first carried out software process evaluation studies. It is alarming that 80 percent of U.S. software companies and projects in 2016 still lag in formal sizing and the use of parametric estimation tools.

| Application Types              | On-time | Late   | Canceled |
|--------------------------------|---------|--------|----------|
| 1 Scientific                   | 68%     | 20%    | 12%      |
| 2 Smart phones                 | 67%     | 19%    | 14%      |
| 3 Open source                  | 63%     | 36%    | 7%       |
| 4 U.S. outsource               | 60%     | 30%    | 10%      |
| 5 Cloud                        | 59%     | 29%    | 12%      |
| 6 Web applications             | 55%     | 30%    | 15%      |
| 7 Games and entertainment      | 54%     | 36%    | 10%      |
| 8 Offshore outsource           | 48%     | 37%    | 15%      |
| 9 Embedded software            | 47%     | 33%    | 20%      |
| 10 Systems and middleware      | 45%     | 45%    | 10%      |
| 11 Information technology (IT) | 45%     | 40%    | 15%      |
| 12 Commercial                  | 44%     | 41%    | 15%      |
| 13 Military and defense        | 40%     | 45%    | 15%      |
| 14 Legacy renovation           | 30%     | 55%    | 15%      |
| 15 Civilian government         | 27%     | 63%    | 10%      |
| Total Applications             | 50.13%  | 37.27% | 13%      |

Table 1: Outcomes of U.S. software projects circa 2016

However, even if accurate estimates can be produced using commercial parametric estimating tools, clients or executives may not accept them. In fact, about half of the cases the author observed during litigation did not produce accurate estimates at all and did not use parametric estimating tools. Manual estimates tend toward optimism, or predicting shorter schedules and lower costs than actually occur.

### Problem 2: Missing Defensible Objective Benchmarks

Somewhat surprisingly, the other half of the cases in litigation had accurate parametric estimates that had been rejected and replaced by arbitrary forced “estimates” based on business needs rather than team abilities. These pseudo-estimates were not produced using parametric estimation tools but were arbitrary schedule demands by clients or top executives based on perceived business needs.

The main reason that the original accurate parametric estimates were rejected and replaced was the absence of supporting historical benchmark data. Without accurate history, even accurate estimates may not be convincing. A lack of solid historical data makes project managers, executives and clients blind to the realities of software development.

Some foreign governments have improved contract accuracy by mandating function point metrics: the governments of Brazil, Japan, Malaysia, Mexico and Italy require function point size and cost information for all government software contracts. Eventually all governments will probably require function point metrics for contracts, but no doubt U.S. state governments and the U.S. federal government will be among the last to do this since they lag in so many other software disciplines. (The author has been an expert witness in more lawsuits involving state governments than any other industry. Government software problems are often national news, e.g., the delay of President Barack Obama’s Affordable Care Act.)

### Problem 3: Rapidly Changing Requirements

The average rate at which software requirements change has been measured to range from about 0.5 percent per calendar month to as high as 4 percent per calendar month. Thus, for a project with a 12-month schedule, more than 10 percent of the features in the final delivery will not have been defined during the requirements phase. For a 36-month project, almost a third of the features and functions may have come in as afterthoughts.

The current state of the art for dealing with changing requirements includes the following:

- Estimating the number and rate of development changes before starting.
- Using function point metrics to quantify changes.
- Using a joint client/development change control board or designated domain experts.
- Using model-based requirements methodologies.
- Calculating the FOG and Flesch readability indices of requirements.
- Involving full-time user representatives for Agile projects.
- Using joint application design (JAD) to minimize downstream changes.
- Using quality function deployment (QFD) for quality requirements.
- Training in requirements engineering for business analysts and designers.

- Using formal requirements inspections to minimize downstream changes.
- Using formal prototypes to minimize downstream changes.
- Planning usage of iterative development to accommodate changes.
- Formally reviewing all change requests.
- Revising cost and schedule estimates for all changes greater than 10 function points.
- Prioritizing change requests in terms of business impact.
- Formally assigning change requests to specific releases.
- Using automated change control tools with cross-reference capabilities.

In projects where litigation occurred, requirements changes were numerous but their effects were not properly planned for in cost, schedule and quality estimates. As a result, unplanned slippages and overruns occurred.

Requirements changes will always occur for large systems. It is not possible to freeze the requirements of any real-world application, and it is naive to think this can occur. Therefore, leading companies are ready and able to deal with changes and do not let them become impediments to progress. For projects developed under contract, the contract itself must include unambiguous language for dealing with changes.

### Problem 4: Poor Quality Control

It is dismaying to observe the fact that two of the most effective technologies in all of software are almost never used on projects that turn out to be disasters and end up in court. First, formal design and code inspections have a 50-year history of successful deployment on large and complex software systems. All “best in class” software producers utilize software inspections.

Second, the technology of static analysis has been available since 1984 and has proven to be effective in finding code bugs rapidly and early (although static analysis does not find requirements, architecture and design problems).

Effective software quality control is the most important single factor that separates successful projects from delays and disasters. This is because finding and fixing bugs is the most expensive cost element for large systems and takes more time than any other activity.

Both “defect potentials” and “defect removal efficiency” should be measured for every project. The “defect potentials” are the sum of all classes of defects; i.e., defects found in requirements, design, source code, and user documents and “bad fixes” or secondary defects. It would be desirable to include defects in test cases too, since there may be more defects in test libraries than in the applications being tested.

The phrase “defect removal efficiency” (DRE) refers to the percentage of defects found before delivery of the software to its actual clients or users. If the development team finds 900 defects before delivery and the users find 100 defects in a standard time period after release (normally 90 days), then the defect removal efficiency is 90 percent.

The author strongly recommends that defect removal efficiency levels (DRE) be included in all software outsource and development contracts, with 96 percent being a proposed minimum

acceptable level of defect removal efficiency. For medical devices and weapons systems, a higher rate of about 99 percent defect removal efficiency should be written into the contracts.

(The U.S. average in 2016 is only about 92 percent. Agile projects average about 92 percent; waterfall are often below 85 percent. TSP and RUP are among the quality strong methods that usually top 96 percent in defect removal efficiency.)

A rate of 96 percent is a significant improvement over current norms. For some mission-critical applications, a higher level such as 99.8 percent might be required. It is technically challenging to achieve such high levels of defect removal efficiency, and it can't be done by testing alone.

In order to top 98 percent in defect removal efficiency, formal inspections and pre-test static analysis plus at least eight forms of testing are needed (1 – unit test; 2 – function test; 3 – regression test; 4 – component test; 5 – performance test; 6 – usability test; 7 – system test; 8 – acceptance or beta test).

Table 2 shows combinations of quality control factors that can lead to high, average or poor defect removal efficiency (DRE).

Successful projects in the 10,000 function point range accumulate development totals of around 4.0 defects per function point and remove about 98 percent of them before delivery to customers. In other words, the number of delivered defects is about 0.2 defects per function point, or 800 total latent defects. Of these, about 10 percent – or 80 – would be fairly serious defects. The rest would be minor or cosmetic defects. Stabilization, or the number of calendar months required to achieve safe operation of the application, would be about 2.5 months.

By contrast, the unsuccessful projects of 10,000 function points that end up in court accumulate development totals of around 6.0 defects per function point and remove only about 85 percent of them before delivery. The number of delivered defects is about 0.9 defects per function point, or 9,000 total la-

tent defects. Of these, about 15 percent – or 1,350 – would be fairly serious defects. This large number of latent defects after delivery is very troubling for users. The large number of delivered defects is also a frequent cause of litigation. Stabilization, or the number of calendar months required to achieve safe operation of the application, might stretch out to 18 months or more.

Unsuccessful projects typically omit design and code inspections and static analysis and depend solely on testing. The omission of upfront inspections and static analysis causes four serious problems: 1) The large number of defects still present when testing begins slows the project to a standstill; 2) The “bad fix” injection rate for projects without inspections is alarmingly high; 3) The overall defect removal efficiency associated with testing only is not sufficient to achieve defect removal rates higher than about 85 percent; and 4) Applications that bypass both inspections and static analysis have a strong tendency to include error-prone modules.

Problem 5: Poor Software Milestone Tracking

Once a software project is underway, there are no fixed and reliable guidelines for judging its rate of progress. The civilian software industry has long utilized ad hoc milestones, such as completion of design or completion of coding. However, these milestones are notoriously unreliable.

Tracking software projects requires dealing with two separate issues: 1) Achieving specific and tangible milestones; and 2) Expending resources and funds within specific budgeted amounts.

Because software milestones and costs are affected by requirements changes and “scope creep,” it is important to measure the increase in size of requirements changes when they affect function point totals. However, there are also requirements changes that do not affect function point totals, which are termed “requirements churn.” Both creep and churn occur at random intervals. Churn is harder to measure than creep and is often measured via “backfiring,” or mathematical conversion between source code statements and function point metrics.

There are also “non-functional requirements,” often due to outside influences. These can change abruptly and many are not under control of software groups. For example, a change in federal or state laws may require changes to hundreds of applications, including some that are under development.

As of 2016, there are automated tools available that can assist project managers in recording the kinds of vital information needed for milestone reports. These tools can record schedules, resources, size changes, and issues or problems.

Examples of tracking tools include Automated Project Office (APO), Microsoft project management suite, OmniTracker, Capterra, and perhaps 50 others with various capabilities. However, in spite of the availability of these tools, less than 45 percent of the author's clients in our initial process evaluation studies used any of them.

For an industry now more than 65 years of age, it is somewhat surprising that there is no general or universal set of project milestones for indicating tangible progress. From the author's assessment and baseline studies, following are some representative milestones that have shown practical value.

Note that these milestones assume an explicit and formal review or inspection connected with the construction of every major soft-

| Defect Removal Efficiency (DRE)  | > 99 % | 95% | < 87% |
|----------------------------------|--------|-----|-------|
| 1 Formal requirement inspections | Yes    | No  | No    |
| 2 Formal design inspections      | Yes    | No  | No    |
| 3 Formal code inspections        | Yes    | No  | No    |
| 4 Formal security inspections    | Yes    | No  | No    |
| 5 Static analysis                | Yes    | Yes | No    |
| 6 Unit test                      | Yes    | Yes | Yes   |
| 7 Function test                  | Yes    | Yes | Yes   |
| 8 Regression test                | Yes    | Yes | Yes   |
| 9 Integration test               | Yes    | Yes | Yes   |
| 10 Usability test                | Yes    | Yes | No    |
| 11 Security test                 | Yes    | Yes | No    |
| 12 System test                   | Yes    | Yes | Yes   |
| 13 Acceptance test               |        | Yes | Yes   |

Table 2: Ranges of DRE for 1,000 function point applications

were deliverable. Formal reviews and inspections have the highest defect removal efficiency levels of any known kind of quality control activity and are characteristics of “best in class” organizations.

The most important aspect of Table 3 is that every milestone is based on completing a review, inspection or test. Just finishing up a document or writing code should not be considered a milestone unless the deliverables have been reviewed, inspected or tested.

In the litigation where the author worked as an expert witness, these criteria were not met. Milestones were very informal and consisted primarily of calendar dates without any validation of the materials themselves.

Also, the format and structure of the milestone reports were inadequate. At the top of every milestone report, problems and issues or “red flag” items should be highlighted and discussed first. These “red flag” topics are those that are likely to cause schedule delays, cost overruns or both.

During depositions and review of court documents, it was noted that software engineering personnel and many managers were aware of the problems that later triggered the delays, cost overruns, quality problems and litigation. At the lowest levels, these problems were often included in weekly status reports or discussed at team meetings. But in the higher-level milestone and tracking reports that reached clients and executives, the hazardous issues were either omitted or glossed over.

A suggested format for monthly progress tracking reports delivered to clients and higher management would include these sections:

|                                                                          |
|--------------------------------------------------------------------------|
| 1. Status of last month's “red flag” problems                            |
| 2. New “red flag” problems noted this month                              |
| 3. Change requests processed this month versus change requests predicted |
| 4. Change requests predicted for next month                              |
| 5. Size in function points for this month's change requests              |
| 6. Size in function points predicted for next month's change requests    |
| 7. Schedule impacts of this month's change requests                      |
| 8. Cost impacts of this month's change requests                          |
| 9. Quality impacts of this month's change requests                       |
| 10. Defects found this month versus defects predicted                    |
| 11. Defects predicted for next month                                     |
| 12. Costs expended this month versus costs predicted                     |
| 13. Costs predicted for next month                                       |
| 14. Deliverables completed this month versus deliverables predicted      |
| 15. Deliverables predicted for next month                                |

*Table 4: Suggested format for monthly status reports for software projects*

Although the suggested format somewhat resembles the items calculated using the earned value method, this format deals explicitly with the impact of change requests and also uses function point metrics for expressing costs and quality data.

An interesting question is the frequency with which milestone progress should be reported. The most common reporting frequency is monthly, although exception reports can be filed at any time that it is suspected that something has occurred that can cause perturbations. For example, serious illness of key project personnel or resignation of key personnel might affect project milestone completions, and this kind of situation cannot

|                                                                                |
|--------------------------------------------------------------------------------|
| 1. Application sizing completed using both function points and code statements |
| 2. Application risk predictions completed                                      |
| 3. Application size and risk predictions reviewed                              |
| 4. Requirements document completed                                             |
| 5. Requirements document inspection completed                                  |
| 6. Initial cost estimate completed                                             |
| 7. Initial cost estimate review completed                                      |
| 8. Development plan completed                                                  |
| 9. Development plan review completed                                           |
| 10. Cost tracking system initialized                                           |
| 11. Defect tracking system initialized                                         |
| 12. Prototype completed                                                        |
| 13. Prototype review completed                                                 |
| 14. Complexity analysis of base system (for enhancement projects)              |
| 15. Code restructuring of base system (for enhancement projects)               |
| 16. Functional specification completed                                         |
| 17. Functional specification review completed                                  |
| 18. Data specification completed                                               |
| 19. Data specification review completed                                        |
| 20. Logic specification completed                                              |
| 21. Logic specification review completed                                       |
| 22. Quality control plan completed                                             |
| 23. Quality control plan review completed                                      |
| 24. Change control plan completed                                              |
| 25. Change control plan review completed                                       |
| 26. Security plan completed                                                    |
| 27. Security plan review completed                                             |
| 28. User information plan completed                                            |
| 29. User information plan review completed                                     |
| 30. Code for specific modules completed                                        |
| 31. Code inspection for specific modules completed                             |
| 32. Code for specific modules unit tested                                      |
| 33. Test plan completed                                                        |
| 34. Test plan review completed                                                 |
| 35. Test cases for specific test stage completed                               |
| 36. Test case inspection for specific test stage completed                     |
| 37. Test stage completed                                                       |
| 38. Test stage review completed                                                |
| 39. Integration for specific build completed                                   |
| 40. Integration review for specific build completed                            |
| 41. User information completed                                                 |
| 42. User information review completed                                          |
| 43. Quality assurance sign off completed                                       |
| 44. Delivery to beta test clients completed                                    |
| 45. Delivery to clients completed                                              |

*Table 3: Representative tracking milestones for large software projects*

be anticipated. The same is true of natural phenomena such as hurricanes or earthquakes, which can shut down businesses.

The simultaneous deployment of software sizing tools, estimating tools, planning tools and methodology management tools can provide fairly unambiguous points in the development cycle that allow progress to be judged more or less effectively. For example, software sizing technology can now predict both the sizes of specifications and the volume of source code needed. Defect estimating tools can predict the number of bugs or errors that might be encountered and discovered. Although such milestones are not perfect, they are better than former approaches.

Project management is responsible for establishing milestones, monitoring their completion, and reporting truthfully whether the milestones were successfully completed or encountered problems. When serious problems are encountered, it is necessary to correct the problems before reporting that the milestones have been completed.

Failing or delayed projects usually lack serious milestone tracking. Activities are often reported as finished while work is still ongoing. Milestones on failing projects are usually dates on a calendar rather than completion and review of actual deliverables.

Delivering documents or code segments that are incomplete, contain errors and cannot support downstream development work is not the way milestones are used by industry leaders.

In more than a dozen legal cases involving projects that failed or were never able to operate successfully, project tracking was inadequate. Problems were either ignored or brushed aside rather than being addressed and solved.

Because milestone tracking occurs throughout software development, it is the last line of defense against project failures and delays. Milestones should be established formally and should be based on reviews, inspections and tests of deliverables. Milestones should not be the dates that deliverables were more or less finished. Milestones should reflect the dates that finished deliverables were validated by means of inspections, testing and quality assurance review.

### **Problem 6: Flawed Outsource Agreements That Omit Key Topics**

In several of the cases where the author has been an expert witness, the contracts themselves seemed flawed and omitted key topics that should have been included. Worse, some contracts included topics that probably should have been omitted. Here are some examples:

- In one case the contract required that the software delivered by the vendor should have "zero defects." Since the application approached 10,000 function points in size, zero-defect software is beyond the current state of the art. The software as delivered did not have very many defects and, in fact, was much better than average. But it was not zero-defect software, and hence the vendor was sued.

- A fixed-price contract had clauses for "out of scope" requirements changes. In this case, the client unilaterally added 82 major changes totaling about 3,000 new function points. But the contract did not define the phrase "out of scope," and the client asserted that the changes were merely elaborations to existing requirements and did not want to pay for them.

- In another fixed-price contract the vendor added about 5,000 function points of new features very late in development. Here the client was willing to pay for the added features. However, features added after design and during coding are more expensive to build than features during normal development. In this case the vendor was asking for additional payments to cover the approximate 15 percent increase in costs for the late features. Needless to say, there should be a sliding scale of costs that goes up for features added three, six, nine, 12 or more months after the initial requirements are defined and approved by the client. The fee structure might be something like an increase of 3 percent, 5 percent, 7 percent, 12 percent and 15 percent based on calendar month intervals.

—In several contracts where the plaintiff alleged poor quality on the part of the vendor, the contracts did not have any clauses that specified acceptable quality, such as defect removal efficiency (DRE) or maximum numbers of bugs found during an acceptance test. In the absence of any contractual definitions of "poor quality," such charges are difficult to prove.

The bottom line is that clients, vendors and their attorneys should be sure that all outsource contracts include clauses dealing with requirements changes, quality, and delivered defects, and also penalties for schedule delays caused by vendor actions.

Note that the author is not an attorney and this is not legal advice. But it is obvious that every software outsource contract should include clauses for quality and for requirements changes, especially late requirements changes. Attorneys should be involved in structuring the proper clauses in software outsource agreements.

### **Summary and Observations Based on Breach of Contract Litigation**

Successful software projects can result from nothing more than avoiding the more serious mistakes that lead to disaster. A set of basic steps can lower the odds of a failing project and litigation: 1) Use parametric estimation tools and avoid manual estimates; 2) Look at the actual benchmark results of similar projects; 3) Make planning and estimating formal activities; 4) Plan for and control creeping requirements; 5) Use formal inspections as milestones for tracking project progress; 6) Include pre-test static analysis and inspections in quality control; 7) Collect accurate measurement data during your current project to use with future projects; 8) Ensure with your attorneys that contracts have suitable clauses for requirements growth and quality levels of delivered materials. Omitting these two topics can lead to very expensive litigation later.

Overcoming the risks shown here is largely a matter of opposites, or doing the reverse of what the risk indicates. Thus, a well-formed software project will create accurate estimates derived from empirical data and supported by automated tools for handling the critical path issues. Such estimates will be based on the actual capabilities of the development team and will not be arbitrary creations derived without any rigor. The plans will specifically address the critical issues of change requests and quality control. In addition, monthly progress reports will also deal with these critical issues. Accurate progress reports are the last line of defense against failures.

## SUGGESTED READINGS

1. Abrain, A. (2015.) *Software Estimating Models*. Wiley-IEEE Computer Society.
2. Abrain, A. (2010.) *Software Metrics and Metrology*. Wiley-IEEE Computer Society.
3. Abrain, A. (2008.) *Software Maintenance Management: Evolution and Continuous Improvement*. Wiley-IEEE Computer Society.
4. Beck, K. (2002.) *Test-Driven Development*. Addison Wesley, Boston, Mass. ISBN 10: 0321146530, 240 pages.
5. Black, R. (2009.) *Managing the Testing Process: Practical Tools and Techniques for Managing Hardware and Software Testing*. Wiley. ISBN-10 0470404159, 672 pages.
6. Boehm, B. (1981.) *Software Engineering Economics*. Prentice Hall, Englewood Cliffs, N.J., 900 pages.
7. Brooks, F. (1974, rev. 1995.) *The Mythical Man-Month*. Addison-Wesley, Reading, Mass.
8. Bundschuh, M. & Dekkers, C. (2005.) *The IT Metrics Compendium*. Springer.
9. Charette, B. (1989.) *Software Engineering Risk Analysis and Management*. McGraw-Hill, New York, N.Y.
10. Charette, B. (1990.) *Application Strategies for Risk Management*. McGraw-Hill, New York, N.Y.
11. DeMarco, T. (1982.) *Controlling Software Projects*. Yourdon Press, N.Y. ISBN 0-917072-32-4, 284 pages.
12. Ebert, C., Dumke, R. & Bundschuh M. (2004.) *Best Practices in Software Measurement*. Springer.
13. Everett, G. D. & McLeod, R. (2007.) *Software Testing – Testing Across the Entire Software Development Life Cycle*. IEEE Press.
14. Ewusi-Mensah, K. (2004.) *Software Development Failures*. MIT Press, Cambridge, Mass. ISBN 0-26205072-2276.
15. Fernandini, P. L. (2002.) *A Requirements Pattern*. Addison Wesley, Boston, Mass. ISBN 0-201-73826-0.
16. Flowers, S. (1996.) *Software Failures: Management Failures; Amazing Stories and Cautionary Tales*. John Wiley & Sons.
17. Gack, G. (2010.) *Managing the Black Hole: The Executive's Guide to Software Project Risk*. Business Expert Publishing, Thomson, Georgia. ISBN10: 1-935602-01-9.
18. Galorath, D. & Evans, M. (2006.) *Software Sizing, Estimation, and Risk Management: When Performance is Measured Performance Improves*. Auerbach; Philadelphia, Penn.
19. Garmus, D. & Herron, D. (2001.) *Function Point Analysis – Measurement Practices for Successful Software Projects*. Addison Wesley Longman, Boston, Mass. ISBN 0-201-69944-3, 363 pages.
20. Garmus, D. & Herron, D. (1995.) *Measuring the Software Process: A Practical Guide to Functional Measurement*. Prentice Hall, Englewood Cliffs, N.J.
21. Garmus, D., Russac, J. & Edwards, R. (2010.) *Certified Function Point Counters Examination Guide*. CRC Press.
22. Glass, R.L. (1998.) *Software Runaways: Lessons Learned from Massive Software Project Failures*. Prentice Hall, Englewood Cliffs.
23. Gibbs, T. W. (1994, September.) *Trends in Computing: Software's Chronic Crisis*. Scientific American Magazine, 271(3), International edition, 72-81.
24. Gilb, T. & Graham, D. (1993.) *Software Inspection*. Addison Wesley, Harlow, U.K. ISBN 10: 0-201-63181-4.
25. Glass, R.L. (1998.) *Software Runaways: Lessons Learned from Massive Software Project Failures*. Prentice Hall, Englewood Cliffs.
26. Harris, M. D. S., Herron, D. & Iwanicki, S. (2008.) *The Business Value of IT*. CRC Press, Auerbach; Boca Raton, Fla. ISBN 978-14200-6474-2.
27. Hill, P., Jones, C. & Reifer, D. (2013 September.) *The Impact of Software Size on Productivity*. International Software Standards Benchmark Group (ISBSG), Melbourne, Australia.
28. International Function Point Users Group (IFPUG). (2002.) *IT Measurement – Practical Advice from the Experts*. Addison Wesley Longman, Boston, Mass. ISBN 0-201-74158-X, 759 pages.
29. Johnson, J. et al. (2000.) *The Chaos Report*. The Standish Group, West Yarmouth, Mass.
30. Jones, C. (2015.) *The Technical and Social History of Software Engineering*. Addison Wesley (contains summaries of important software industry lawsuits such as anti-trust and patent violations).
31. Jones, C. (2012.) *Studio 38 in Rhode Island – A Study of Software Risks*. Published in various Rhode Island newspapers such as "The Providence Journal," "South County Independent," "Narragansett Times," etc.
32. Jones, C. & Bonsignour, O. (2011.) *The Economics of Software Quality*. Addison Wesley, Boston, Mass. ISBN 10 0-13-258220-1, 587 pages.
33. Jones, C. (2010.) *Software Engineering Best Practices*. McGraw Hill, New York, N.Y. ISBN 978-0-07-162161-8, 660 pages.
34. Jones, C. (2008.) *Applied Software Measurement*. McGraw Hill, 3rd edition. ISBN 978-0-07-150244-3, 662 pages.
35. Jones, C. (1994.) *Assessment and Control of Software Risks*. Prentice Hall. ISBN 0-13-741406-4, 711 pages.
36. Jones, C. (1995, December.) *Patterns of Software System Failure and Success*. International Thomson Computer Press, Boston, Mass. 250 pages. ISBN 1-850-32804-8, 292 pages.
37. Jones, C. (1997.) *Software Quality – Analysis and Guidelines for Success*. International Thomson Computer Press, Boston, Mass. ISBN 1-85032-876-6, 492 pages.
38. Jones, C. (2007.) *Estimating Software Costs*. McGraw Hill, New York, N.Y. ISBN 13-978-0-07-148300-1.
39. Jones, C. (2000.) *Software Assessments, Benchmarks, and Best Practices*. Addison Wesley Longman, Boston, Mass. ISBN 0-201-48542-7, 657 pages.
40. Jones, C. (1998, December.) *Sizing Up Software*. Scientific American Magazine, Vol. 279, No. 6, 104–111.
41. Jones, C. (2007.) *Conflict and Litigation Between Software Clients and Developers; Software Productivity Research technical report*. Narragansett, R.I., 65 pages.
42. Kan, S. H. (2003.) *Metrics and Models in Software Quality Engineering*, 2nd edition. Addison Wesley Longman, Boston, Mass. ISBN 0-201-72915-6, 528 pages.
43. Pressman, R. (2005.) *Software Engineering – A Practitioner's Approach*. McGraw Hill, N.Y. 6th edition. ISBN 0-07-285318-2.
44. Radice, R. A. (2002.) *High Quality Low Cost Software Inspections*. Paradoxicon Publishing, Andover, Mass. ISBN 0-9645913-1-6, 479 pages.
45. Robertson, S. & Robertson, J. (2005.) *Requirements-Led Project Management*. Addison Wesley, Boston, Mass. ISBN 0-321-18062-3.
46. Wiegers, K. E. (2002.) *Peer Reviews in Software – A Practical Guide*. Addison Wesley Longman, Boston, Mass. ISBN 0-201-73485-0, 232 pages.
47. Yourdon, E. (1997.) *Death March – The Complete Software Developer's Guide to Surviving "Mission Impossible" Projects*. Prentice Hall PTR, Upper Saddle River, N.J. ISBN 0-13-748310-4, 218 pages.
48. Yourdon, E. (2005.) *Outsource: Competing in the Global Productivity Race*. Prentice Hall PTR, Upper Saddle River, N.J. ISBN 0-13-147571-1, 251 pages.

## Websites

Information Technology Metrics and Productivity Institute (ITMPI): [www.ITMPI.org](http://www.ITMPI.org)  
 International Software Benchmarking Standards Group (ISBSG): [www.ISBSG.org](http://www.ISBSG.org)  
 International Function Point Users Group (IFPUG): [www.IFPUG.org](http://www.IFPUG.org)  
 Namcook Analytics LLC: [www.Namcook.com](http://www.Namcook.com)  
 Namcook Analytics Blog: <http://NamcookAnalytics.com>  
 Reifer Consulting: [www.Reifer.com](http://www.Reifer.com)  
 Software Engineering Institute (SEI): [www.SEI.cmu.edu](http://www.SEI.cmu.edu)  
 Software Productivity Research (SPR): [www.SPR.com](http://www.SPR.com)

## SUGGESTED WEBSITES

<http://www.IASAHome.org>

This is the website for the nonprofit International Association of Software Architects (IASA). Software architecture is the backbone of all large applications. Good architecture can lead to applications with useful life expectancies of 20 years or more. Questionable architecture can lead to applications with useful life expectancies of fewer than 10 years, coupled with increasing complex maintenance tasks and high defect levels. The IASA is working hard to improve both the concepts of architecture and the training of software architects via a modern and extensive curriculum.

<http://www.IIBA.org>

This is the website for the nonprofit International Institute of Business Analysis. This institute deals with the important link between business knowledge and software that supports business operations. Among the topics of concern are the Business Analysis Body of Knowledge (BABOK), training of business analysts, and certification to achieve professional skills.

<http://www.IFPUG.org>

This is the website for the nonprofit International Function Point Users Group. IFPUG is the largest software metrics association in the world and the oldest association of function point users. This website contains information about IFPUG function points themselves and also includes citations to the literature dealing with function points. IFPUG also offers training in function point analysis and administers. IFPUG also administers a certification program for analysts who wish to become function point counters.

<http://www.ITMPI.org>

This is the website for the Information Technology Metrics and Productivity Institute. ITMPI is a wholly owned subsidiary of Computer Aid Inc. The ITMPI website is a useful portal into a broad range of measurement, management and software engineering information. The ITMPI website also provides useful links to many other websites that contain topics of interest on software issues.

<http://www.ISBSG.org>

This is the website for the nonprofit International Software Benchmark Standards Group. ISBSG, located in Australia, collects benchmark data on software projects throughout the world. The data is self-reported by companies using a standard questionnaire. About 4,000 projects comprise the ISBSG collection as of 2007, and the collection has been growing at a rate of about 500 projects per year. Most of the data is expressed in terms of IFPUG function point metrics, but some of the data is also expressed in terms of COSMIC function points, NESMA function points, Mark II function points, and several other function point variants. Fortunately, the data in variant metrics is identified. It would be statistically invalid to include attempts to average IFPUG and COSMIC data, or to mix up any of the function point variations.

<http://www.iso.org>

This is the website for the International Organization for Standardization (ISO). The ISO is a nonprofit organization that sponsors and publishes a variety of international standards. As of 2007 the ISO published about a thousand standards per year, and the total published to date is approximately 17,000. Many of the published standards affect software. These include the ISO 9000-9004 quality standards and the ISO standards for functional size measurement.

<http://www.namcook.com>

This website contains a variety of quantitative reports on software quality and risk factors. It also contains a patented high-speed sizing tool that can size applications of any size in 90 seconds or fewer. It also contains a catalog of software benchmark providers that currently lists 20 organizations that provide quantitative data about software schedules, costs, quality and risks.

<http://www.PMI.org>

This is the website for the Project Management Institute (PMI). PMI is the largest association of managers in the world. PMI performs research and collects data on topics of interest to managers in every discipline: software, engineering, construction, and so forth. This data is assembled into the well-known Project Management Body of Knowledge, or PMBOK.

<http://www.sei.cmu.edu>

This is the website for the Software Engineering Institute (SEI). The SEI is a federally sponsored nonprofit organization located on the campus of Carnegie Mellon University in Pittsburgh, Penn. The SEI carries out a number of research programs dealing with software maturity and capability levels, with quality, risks, measurement and metrics, and other topics of interest to the software community.

<http://www.stsc.hill.af.mil/CrossTalk>

This is the website of both the Air Force Software Technology Support Center (STSC) and also the CrossTalk journal, which is published by the STSC. The STSC gathers data and performs research into a wide variety of software engineering and software management issues. The CrossTalk journal is one of few technical journals that publish full-length technical articles of 4,000 words or more. Although the Air Force is the sponsor of STSC and CrossTalk, many topics are also relevant to the civilian community. Issues such as quality control, estimating, maintenance, measurement and metrics have universal relevance.

## 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, Conn. He was also a manager and researcher at IBM in California.

Capers 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.

**[www.Namcook.com](http://www.Namcook.com)**

**<http://namcookanalytics.com>**

**[Capers.Jones3@gmail.com](mailto:Capers.Jones3@gmail.com)**

# The Measurement of Software Maintenance and Sustainment

## Positive Influences and Unintended Consequences

**Rob Ashmore, U.K. Defence Science and Technology Laboratory**  
**Mike Standish, U.K. Defence Science and Technology Laboratory**

**Abstract.** Software metrics can provide valuable information to decision makers and can assist with the management of the software supply chain. However, a poorly chosen set of metrics may have negative unintended consequences, resulting in software that is more expensive to maintain. Among other things, a crucial component in the intelligent and effective use of software metrics is a sound, system-level understanding of the underlying software sustainment process. This article illustrates one mechanism that can be used to develop this understanding and highlights the benefits that it can deliver.

### Introduction

In this paper we are interested in what motivates the choice of a particular set of software metrics, especially when these metrics are to be used as part of a Performance Based Logistics (PBL) [1] contract. These types of contract are being implemented for large and complex software systems such as the F-35 Lightning II [2]. In particular, we are concerned with how the chosen metrics will influence the behavior of the software maintenance and sustainment system. Of particular concern is protecting against negative unintended consequences that may occur if metrics are chosen poorly.

Initially, we consider how measurement can affect the behavior of human systems. We then develop some simple, and incomplete, models that illustrate how poorly chosen metrics could drive the software maintenance and sustainment system toward undesirable behaviors. We close by summarizing how to protect against this risk.

### Measurement and Behavior

The relationship between measurement and system behavior has been the subject of a number of studies of complex human systems (e.g., implementation of Government policies [3], the English National Health Service [4], and U.S. Veterans Health Administration facilities [5]). In addition, the unintended consequences of publishing performance data for U.K. public sector organizations have been studied; this topic is discussed in the following paragraphs, which are heavily based on Smith [6].

In his abstract, Smith states, “the performance indicator philosophy is based on inadequate models of production and control.” To put that quote into the context of this article, Performance Indicators (PIs) can be considered analogous to software metrics,

and the phrase “models of production and control” relates to a system-level model of the activity that is being undertaken, which in our case is software maintenance and sustainment.

Smith goes on to note, “the findings of the paper are therefore likely to be relevant to any situation in which performance data — whether directed at political, agency or managerial control — play a significant part in guiding the activities of the organization.” This situation is precisely the one we are interested in, where performance-based measures (i.e., software metrics) are used to influence the behavior of the software maintenance and sustainment organization so that defined outcomes are achieved for the warfighter.

### Potential Unintended Consequences

Smith highlights a number of negative unintended consequences of using performance data to influence system behavior. These are all based on observations of U.K. public sector organizations and can be grouped into eight distinct types:

- **Tunnel vision**, when management focuses on quantified aspects of performance rather than overall quality.
- **Suboptimization**, where narrow, local objectives are prioritized over the wider objectives of the organization as a whole.
- **Myopia**, which involves the pursuit of short-term targets at the expense of legitimate long-term objectives or outcomes.
- **Measure fixation**, where managers focus on the metric, rather than the objective for which the metric was developed.
- **Misrepresentation**, where the reported metrics do not match the behavior on the ground.
- **Misinterpretation**, where those to whom the metrics are reported make incorrect or inappropriate decisions.
- **Gaming**, where behavior is deliberately altered so as to exploit loopholes in the measurement system.
- **Ossification**, where an overly rigid measurement system prevents innovation.

### Mitigation Strategies

Smith advances a number of strategies that may be used to mitigate the risk of such unintended consequences. Two prime examples are the following:

- Involving staff at all levels when setting metrics; this readily protects against suboptimization.
- Retaining flexibility in the chosen metrics and not relying on them exclusively for control purposes; this readily protects against ossification.

While these strategies provide some level of mitigation, a more holistic approach involves gaining a system-level understanding of what is being measured. This is a key consideration, which is highlighted in the abstract of Smith's paper. Although the development of a completely accurate model is impossible, relatively simple models can still provide a helpful level of understanding. More importantly, they allow the effects of measurement to be monitored so that, if necessary, the adopted measures can be altered.

One way of representing system-level models is by using Causal Loop Diagrams (CLDs). To illustrate these diagrams, we begin with a simple example relating to technical debt and its potential effect on software release schedules.

### Technical Debt and Preventive Changes

The term “technical debt” was coined by Ward Cunningham in a talk at the 1992 Object-Oriented Programming, Systems, Languages and Applications (OOPSLA) conference [7]. It refers to code that is known to be ‘not quite right’ but a decision has been made to postpone making it right. Often, but not always, the ‘not quite right’ piece relates to the way new code integrates with an existing software architecture.

Incurring technical debt allows, for example, a new release to be produced sooner than otherwise would be the case. This usually comes at a longer-term cost, as indicated by the debt metaphor. In particular, as this level of debt grows, it becomes more difficult to make changes, slowing down future releases. Ultimately, an unchecked growth in technical debt is likely to shorten the lifespan of the software, hastening the need for its replacement.

Paying back the debt involves a software change, specifically one that aims to ease future maintenance. This is one of the four types of change discussed by Williams and Carver [8]:

- **Perfective** changes result from new or changed requirements.
- **Corrective** changes occur in response to defects.
- **Adaptive** changes occur when moving to a new environment, or to a new platform, or to accommodate new standards.
- **Preventative** changes ease future maintenance by restructuring or reengineering the system.

Figure 1 illustrates some of these concepts. Initially, we focus on the items shown in black text; i.e., we focus on the reinforcing loop, which is named “impending bankruptcy.” This may be interpreted as follows:

- An increase in software schedule pressure leads to an increase in technical debt. (The “**S**” on the arrow means that an increase in the quantity at the arrow’s tail leads to an increase in the item at its head; that is, they move in the Same direction.)
- The increase in technical debt leads to a reduction in the ease with which future changes can be met. (The “**O**” on this arrow means the quantities it joins move in Opposite directions.)

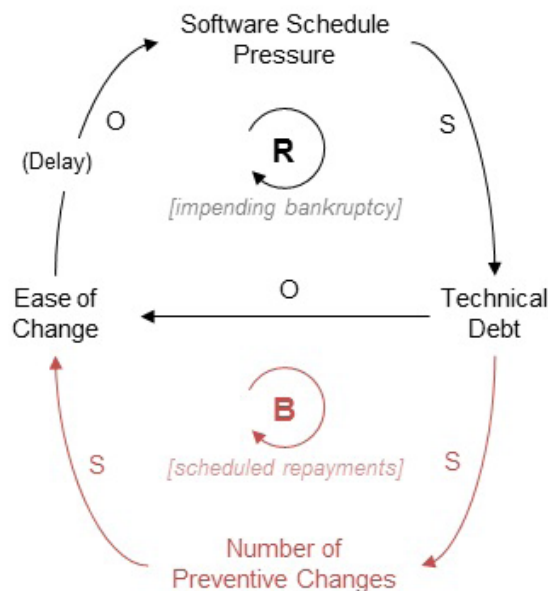


Figure 1. Initial causal loop diagram for technical debt

- The reduction in the ease with which future changes can be met leads, in turn, to schedule pressure for future releases (hence the delay).

Overall, working around the loop, an increase in schedule pressure leads to a further increase in schedule pressure. Hence, this is a reinforcing loop and is annotated with an “**R**” (in simple terms, a loop that has an even number of “**O**” arrows will be Reinforcing and a loop with an odd number of “**O**” arrows will be Balancing). The system-level effect is that increasing technical debt to achieve one release will, if left uncorrected, make it more and more difficult to complete future releases. Ultimately, this will reduce the software’s lifespan or, extending the debt metaphor, lead to technical bankruptcy.

It is appropriate to note that the preceding description (and the associated CLD) is just one selection from a range of potential situations. For example, a small amount of technical debt residing in a stable part of the software (i.e., a part where few, if any, changes are made) might be maintained indefinitely with little adverse effect. More generally, any CLD is just one way of representing a system. Different representations, and hence different system-level behaviors, are often plausible. This is one reason why flexibility should be retained in the chosen metrics. It also highlights the importance of monitoring system behavior and comparing it with expectations.

With that caveat in mind, consider the red text in Figure 1; i.e., where we focus on the loop named “scheduled repayments.” This may be interpreted in the following fashion:

- An increase in schedule pressure leads to an increase in technical debt; this prompts an increase in the number of preventive changes, which makes future changes easier, thus re-balancing schedule pressure across future releases.

In this case, an initial increase in schedule pressure works through the loop to result in a reduction in schedule pressure later. This is a balancing loop (as denoted by the “**B**”).

Applying Smith’s observations on unintended consequences to this simple example suggests that using software metrics based solely on the time taken to complete the current release risks the system being driven to behave as in the “impending bankruptcy” loop; that is, the longer-term future of the software will be jeopardized by tightly focusing on short-term issues. In contrast, including metrics that encourage the implementation of perfective changes is, according to this model, more likely to drive the system into a balanced behavior, which should yield through-life benefits.

Of course, both of these examples are simplistic and fail to capture the full complexities of system behavior. Nevertheless, they still provide useful insights on the potential unintended consequences of adopting a particularly narrow set of software metrics.

### A Simple Model of Software Maintenance and Sustainment

Figure 2 provides a larger, but still incomplete, model of software maintenance and sustainment; a separate model of the same activity was discussed by Ferguson, et. al., in “Modeling Software Sustainment” [9]. Like the model discussed in the previous sub-section, the one presented here is just one representation of a

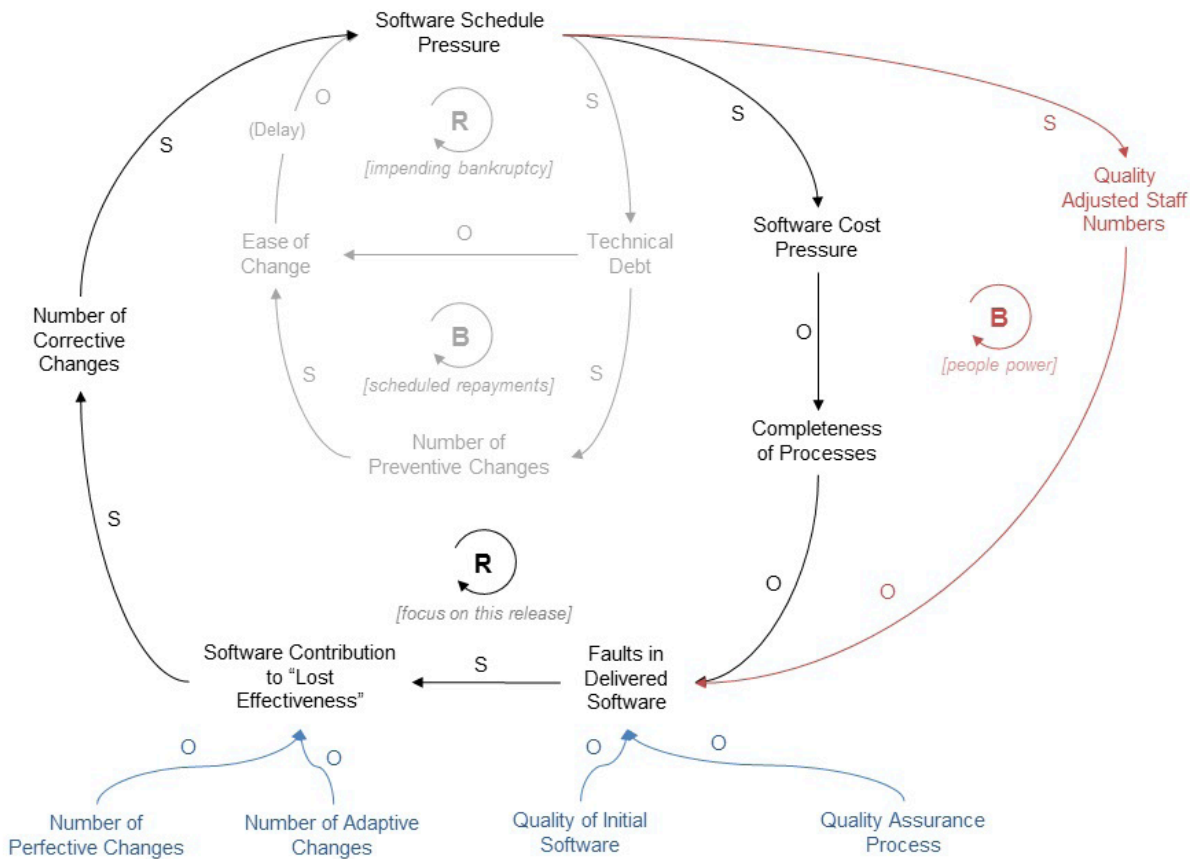


Figure 2. A simple model of software maintenance and sustainment

potential behavior. Nevertheless, it captures some important features, which can be used to inform the choice of metrics.

The following points highlight some of the key nodes in the model:

- The central portion of the model (shown in gray) reproduces the relationship discussed previously (shown within Figure 1).
- On the bottom left of the diagram is a node that captures “Software Contribution to ‘Lost Effectiveness.’” Note that this node also captures effectiveness that is “lost” due to an inability to meet new requirements.
- The lost effectiveness highlighted in the previous node is likely to affect the “Number of Corrective Changes” that are required.
- The “Software Cost Pressure” node is self-explanatory. It is included in the model as it addresses a metric that, by definition, has to be included in any PBL contract (i.e., cost).
- The “Completeness of Processes” node is included to represent the balance between maintaining a planned release date (which can involve reducing the amount of time available for testing) and, for want of a better phrase, the level of “thoroughness” involved in the process.
- The “Faults in Delivered Software” node implicitly captures both the number and the importance of any faults that are present (i.e., it covers issues like whether a workaround is available, the number of missions that would exercise the faulty software, and so on).
- The “Quality Adjusted Staff Numbers” node (shown in red on the right of the diagram) combines both the number of staff that are available and their quality. This node covers

all software-related staff, including, for example, managers, developers, reviewers and testers.

- On the bottom of the diagram are four external inputs, which are shown in blue text:
  - The first two external inputs, “Number of Perfective Changes” and “Number of Adaptive Changes” capture the impact of new requirements, whether these arise, for example, from a desire to conduct new missions (perfective) or a need to accommodate new standards (adaptive).
  - The next external input relates to the “Quality of the Initial Software,” i.e., the quality of the software when the system achieves Initial Operating Capability (IOC). Unlike new requirements, which may arise multiple times during the aircraft’s life, this external input is a one-off.
  - The final external input relates to the “Quality Assurance Process.” This is included as the Quality Assurance (QA) organization is independent of the software development one and, as such, QA should be able to, for example, prevent inappropriate short cuts (i.e., ones that violate agreed procedures) being taken.

### Reinforcing Loop – “Focus on This Release”

Initially, we consider the reinforcing loop named “focus on this release.” This interpretation of system behavior can be described as follows:

- An increase in “lost effectiveness” due to shortfalls in software functionality increases both the demand for corrective changes

and the pressure for a new release to be produced quickly. This is assumed to lead to an increase in cost pressure, a link that is somewhat debatable; sometimes doing things more quickly also means doing them more cheaply.

- The combination of increased schedule pressure and increased cost pressure reduces the completeness of the processes (e.g., the number of resources devoted to testing). In turn, this increases the number of faults in the software, which further increases “lost effectiveness.”

As with all the causal loops discussed in this article, this description presents a very simple view of what is, in reality, a complex system; it also contains general statements for which specific counter-examples could be provided. For example, the “lost effectiveness” may not be felt immediately, as the new release may be expected to fix most (if not all) of the faults that were impacting effectiveness. Likewise, the relationship between schedule pressure and testing completeness is not entirely governed by cost pressure. Nevertheless, this discussion again highlights the potential risk of focusing too tightly on cost and schedule metrics for each upcoming release in isolation.

### Balancing Loop – “People Power”

Finally, we consider the balancing loop named “people power.” Much of the description of this loop is similar to the previous discussion. The new features may be summarized as follows:

- An increase in schedule pressure is used to justify an increase in staff numbers and/or quality. This acts to reduce the number of faults in the delivered software, thus providing a degree of balance to the system.

Yet again, this description is somewhat simplified and idealistic. It could be argued, for example, that schedule pressure will not lead to an increase in staff numbers and/or staff quality (e.g., because suitable staff cannot be recruited, or because there is a perception that there is no spare time for training). Likewise, there is some evidence that simply adding more staff to a late-running software project may make it later (this is sometimes referred to as Brooks’ Law and is described in [10]). Conversely, bringing experienced staff with detailed knowledge of the project’s history back onto the team could be beneficial.

Despite these limitations, this loop does illustrate the benefit that can be obtained by combining an appropriate set of metrics with an understanding of the desired system-level behavior. In particular, measuring the planned staff attributes at the beginning of a release cycle and comparing this with the likely schedule pressure – which may be informed by data from previous releases – should help drive the overall system into a more balanced state than otherwise would be the case. Achieving and maintaining such a state is an important aspect of the through-life cost-effectiveness of defense software; it should also help make software release schedules more predictable.

### Conclusions

It is well understood that measurement influences behavior. However, when applied to complex human systems that are involved in software maintenance and sustainment, the inappropriate use of metrics can have negative unintended consequences.

There are several strategies that can be used to mitigate the risk of unintended consequences. However, the most comprehensive mitigation strategy involves gaining a system-level understanding of the process that is being measured and using that understanding to identify likely responses to different measurement choices. The system-level understanding should also be used to monitor the measurement-induced effects so that, if necessary, corrective action can be taken.

To illustrate the concept, a simple CLD representation of the software maintenance and sustainment process has been developed. This indicates that, for example, focusing solely on metrics associated with the current release could have negative impacts on through-life cost-effectiveness. Although this observation (and others discussed in this article) may be helpful, the key conclusion is that any proposed set of software maintenance and sustainment metrics should be accompanied by the following:

- A system-level description of the process that is being measured.
- A description of how the metrics are intended to influence the system toward the desired behavior, including how they might interact to generate unintended consequences.
- An explanation of how the risk of unintended consequences will be mitigated. This should include a description of how the effects induced by the metrics will be monitored and how the selection of metrics will be altered if necessary.

### Acknowledgements

Lieutenant Commander Steven “Dutch” Holland, R.N., provided valuable assistance as the concepts discussed in this article were developed.

### Disclaimer and Copyright

This article is an overview of UK MOD-sponsored research and is released for informational purposes only. The contents of this article should not be interpreted as representing the views of the UK MOD, nor should it be assumed that they reflect any current or future UK MOD policy. The information contained in this article cannot supersede any statutory or contractual requirements or liabilities and is offered without prejudice or commitment.

(c) Crown copyright 2016, Dstl. This material is licensed under the terms of the Open Government Licence except where otherwise stated. To view this licence, visit <http://www.nationalarchives.gov.uk/doc/open-government-licence/version/3> or write to the Information Policy Team, The National Archives, Kew, London TW9 4DU, or email: [psi@nationalarchives.gsi.gov.uk](mailto:psi@nationalarchives.gsi.gov.uk). [DSTL/DOC86875].

### ADDITIONAL READING

- Meadows, D.H. *Thinking in Systems: A Primer*. Chelsea Green Publishing, ISBN 1-603-58055-7, 2008.
- Sterman, J. *Business Dynamics: Systems Thinking and Modeling for a Complex World*. McGraw-Hill Higher Education, ISBN 0-071-17989-5, 2000.
- Coyle, R. *System Dynamics Modelling: A Practical Approach*. Chapman and Hall/CRC, ISBN 0-412-61710-2, 1996.

## REFERENCES

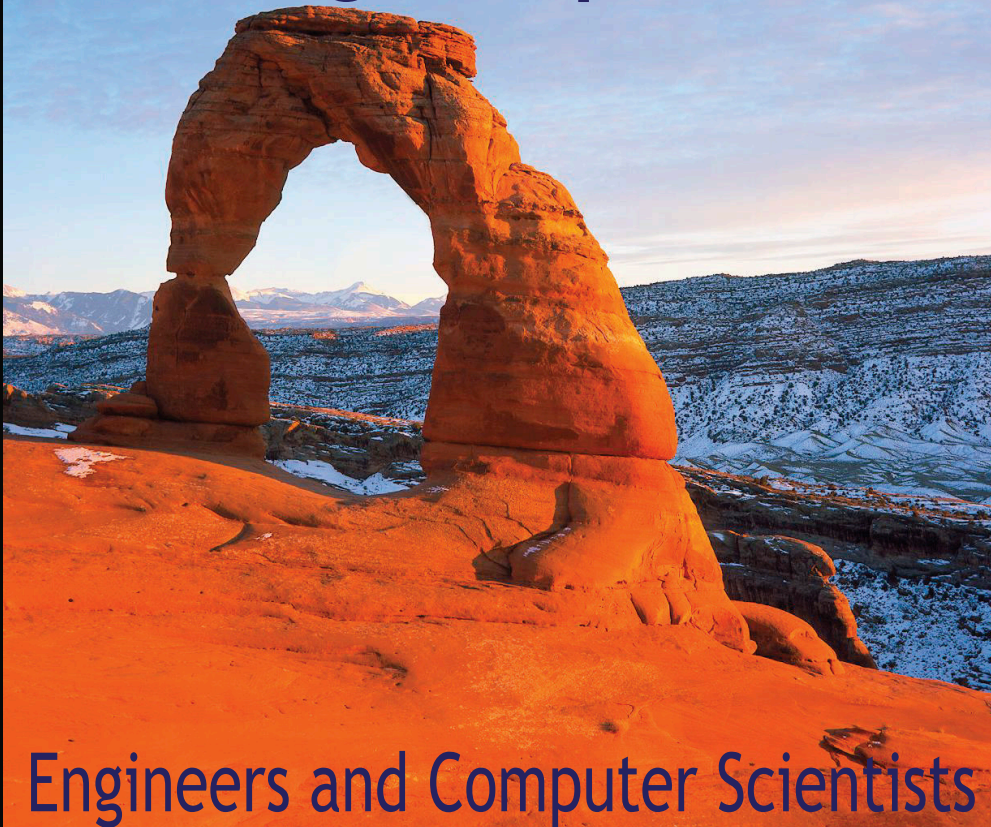
1. Defense Acquisition University. Performance Based Logistics Community of Practice. Retrieved from <https://acc.dau.mil/pbl>. (Accessed 2015, February 10.)
2. Huff, L. & Novak, G. (2007). Performance-Based Software Sustainment for the F-35 Lightning II. *CrossTalk – The Journal of Defense Software Engineering*. 20 (12) 9-14.
3. Johnsen, Å. (2005). What Does 25 Years of Experience Tell Us About the State of Performance Measurement in Public Policy and Management? *Public Money & Management*. 25 (1) 9-17.
4. Mannion, R. & Braithwaite, J. (2012). Unintended consequences of performance measurement in healthcare: 20 salutary lessons from the English National Health Service. *Internal Medicine Journal*. 42 (5) 569-574.
5. Powell, A., et al. (2012). Unintended Consequences of Implementing a National Performance Measurement System into Local Practice. *Journal of General Internal Medicine*. 27 (4) 405-412.
6. Smith, P. (1995). On the Unintended Consequences of Publishing Performance Data in the Public Sector. *International Journal of Public Administration*. 18 (2/3) 277-310.
7. Cunningham, W. (1992). The WyCash Portfolio Management System. *Proc. OOPSLA (Object-Oriented Programming, Systems, Languages & Applications)*, ACM. Retrieved from <http://c2.com/doc/oopsla92.html>.
8. Williams, B. J. & Carver, J. C. (2010). Characterizing Software Architecture Changes: A Systematic Review. *Information and Software Technology* 52, 31-51.
9. Ferguson, R., Phillips, M. & Sheard, S. (2014, January/February). Modeling Software Sustainment. *CrossTalk*, 19-22.
10. Brooks, F. P. (1995). *The Mythical Man Month*. Addison-Wesley, ISBN 0-201-83595-9.

## ABOUT THE AUTHORS

**Rob Ashmore** is a principal software specialist at the U.K. Defence Science and Technology Laboratory (Dstl), a trading fund of the U.K. Ministry of Defence. He has over 20 years' experience in defense software, covering all aspects of the software life cycle. He holds both bachelor's and master's degrees from the University of Cambridge and is a Chartered Scientist (CSci) and a Fellow of the Institute of Mathematics and its Applications (FIMA).

**Mike Standish** is a senior engineer in systems at the U.K. Defence Science and Technology Laboratory (Dstl). He has gained experience of all aspects of software and systems life cycles through over 10 years within the defence sector. He holds a Bachelor of Science in software engineering and a Master of Science in Strategic Information Systems. He is currently undertaking an Engineering Doctorate in Systems. He is a Chartered Engineer (CEng) gained via the British Computer Society (BCS).

# Hiring Expertise



# Engineers and Computer Scientists

**T**he Software Maintenance Group at Hill Air Force Base is recruiting **civilians** (U.S. Citizenship Required). Benefits include paid vacation, health care plans, matching retirement fund, tuition assistance, paid time for fitness activities, and workforce stability with 150 positions added each year over the last 5 years.

**Become part of the best and brightest!**

Hill Air Force Base is located close to the Wasatch and Uinta mountains with skiing, hiking, biking, boating, golfing, and many other recreational activities just a few minutes away.



**Send resumes to:**

309SMXG.Recruiting@us.af.mil  
or call (801) 777-9828

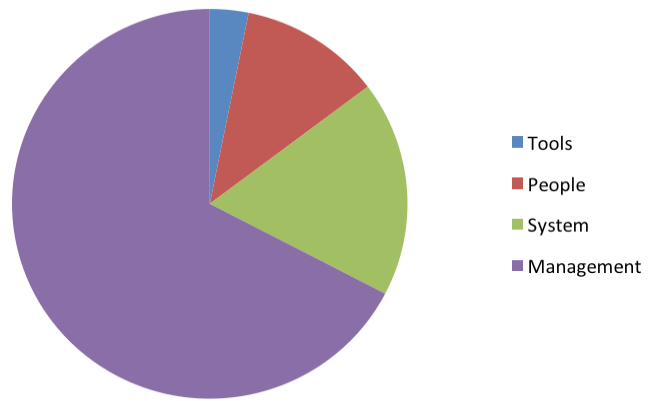


www.facebook.com/  
309SoftwareMaintenanceGroup



**Abstract.** The software engineering profession has experienced several decades of almost unnoticed successes and famous failures. Over that time, its accomplishments have far exceeded what the founders of this profession could have foreseen. One aspect of software engineering that has gone almost unnoticed is the necessity for competent software project managers. Recent studies have shown that the software project manager plays a much more significant role in the success of software projects than previously thought. This article examines the role software project managers play in the development of software systems, explains why this has been largely overlooked, and proposes that the practice of software project management has always been and continues to be critical to the success of software projects today and into the future.

All significant engineering efforts require someone to manage them. Even from their simple beginnings, software development efforts have needed someone responsible for ensuring the work was completed satisfactorily. The common term for the person responsible for such efforts is “project manager.”



Early in the evolution of the software engineering profession, projects were, as compared with today's efforts, relatively simple, standalone efforts to automate some aspect of a business or government activity. But as time went on, these single-purpose systems became integrated and evolved into complex interactive systems operating in real time, simultaneously serving the needs of thousands of users online. That complexity did not occur overnight, but it occurred rapidly enough to overwhelm software project managers who were largely untrained in project management. Many had created their own approaches to managing software projects, but even simple projects often failed to meet budget and schedule constraints.

## Competent Software Project Managers

## The Impact of the Software Project Manager

Two decades ago, Weinberg showed that the impact of the software project manager on a software project ex-

ceeded that of all other groups combined. He did this by combining the various cost driver groups in COCOMO [7]. See Figure 1 (based on Jensen's "Don't Forget About Good Management"). [6]

That work seems to have gone unnoticed for many years. More recently, a study by IBM [8] found that 54 percent of project failures were attributable to poor project management, while three percent were due to technical challenges. In spite of these and other qualitative and quantitative results highlighting the importance of the software project manager, today's software engineering curricula, conferences, publications and teaching position announcements rarely mention software project management as a subject of interest. When undergraduate and graduate software engineering curricula do include software project management as a course, it is frequently listed as optional.

In spite of the many cost overruns, delayed deliveries and mediocre results of some software systems, others have been successfully delivered by teams of software engineers led by largely untrained software project managers. [9] The problem today is that the increased complexity and importance of these software systems means that conducting software development in a "business as usual" manner — i.e., with uncertain delivery dates, cost overruns, poor quality, and poor maintainability — is likely to lead to a self-limiting future in which new systems and system upgrades are avoided or curtailed altogether. If the software engineering profession is to have a successful future, it needs more well-trained, competent software project managers. Perhaps more importantly, software engineers need to be made aware of the importance of the software project manager and the value software project managers bring to software projects.

### Improving the Status Quo

In software engineering, we are trying very hard to change the status quo. However, the areas we are focusing on are not those that can provide the highest return on investment. For example, programming methods like Agile and Extreme have improved many aspects of software engineering. But as Weinberg observed [7] and, more recently, the work by Gulla demonstrated, [8] these are at best secondary effects. For example, from 1960 to 1990, the focus was on programmer productivity. It increased at a linear rate of just one source line per programmer month per year. [6] We have no reason to believe that the linear increase over that period has changed significantly since then. While that modest increase is positive, it occurred during a period when dozens of computer-aided software engineering tools, as well as analysis, design and programming methods, were published and adopted by the software engineering community. [2] It should also be noted that producing large amounts of source code has not always been the problem. Obviously, the biggest improvement in software engineering project performance, and the highest leverage area, is software project management. But how can this be accomplished?

### A Better Software Project Manager and Software Project Management

Effective software project managers are not born; they are made through education, experience, mentoring and other related means. Our profession's lack of recognition of the importance of software project management is reflected in many ways. For example, the Software Engineering Body of Knowledge (SWEBOK) [10] treats software project management in the same way it treats other, often esoteric, topics in software engineering. If we really recognized the impact the software project manager can have on a project, SWEBOK would devote much more space to it or, perhaps due to its importance and the significant differences between software project management and software engineering, software project management should have its own standalone body of knowledge (e.g., SWPMBOK). [17] Recently, a software extension to the Guide to the Project Management Body of Knowledge, (PMBOK®) [12] was published in an attempt to remedy this situation. [13] But it is an extension, not a standalone document that focuses squarely on software project management. This extension requires the user to continually reference PMBOK to obtain the needed information — assuming that the information is present, which it may not be because information is generalized and covers a broad range of industries. A more specific discussion of what software project managers need to know in order to be successful has been developed and published. [14] Based on this new reference, the extension does not completely address the knowledge, methods, techniques, and data software project managers need to understand in order to be successful.

Software project management has little to do with programming and a great deal to do with what are commonly referred to as "soft" skills (i.e., communication, staffing, motivating, coping with complexity, risk management, and personnel issues) that software project managers need in order to be successful. The real question today is what resources — e.g., referential documents, university training, and professional development courses — exist to assist current and future software project managers in acquiring the knowledge and skills they need to be more consistently successful? Granted, not all software engineering professionals may want to become software project managers, but many will. When they do, they will need to know what software project management is about.

The most important fact they need to accept is that software project management does not involve the same domain of technology as programming. It requires a different mindset. The software project manager is not developing the software, but is instead working with the development team to create a plan by which the team will develop the software. The project manager re-plans as needed, monitors progress, and reports the project's status to key stakeholders while working to remove obstacles that prevent the software engineering team from working at their highest performance level. It mostly involves the personnel and business end of the software engineering profession. Failure to accept this fact has probably contributed significantly to the number of software projects that have failed.

As one noted software professional observed, “Management, not technology, determines success.” [15] Also, the education side of this problem will help software engineers to better understand the importance of software project management and the value that software project managers bring to software projects. But software project managers do not need to be among the best software engineers in order to be effective. [16] This is because the best software engineers often become frustrated software project managers who have little patience with those who are less proficient than they are and may not mentor others to reach their full potential. This often results in a frustrated software project manager who reverts to being a software engineer, leaving the project without a manager. The dangers involved in putting high performers into management were documented long ago, but the message has still not had widespread acceptance. [17] [18]

Who Should be Educated?

If we look at companies engaged in software engineering projects from a system-level point of view, the delivery side is composed of software engineers, software project managers and senior managers. This last group, senior managers, are the ones who decide who becomes a software project manager and reviews their performance. A few decades ago, a very successful chief executive officer of major international

corporations wrote an advisory against putting high performers into management. [16] [18] He pointed out that putting the best at performing certain tasks into management was not advisable. This is because doing so reduces the overall performance of the group they were in, and this new manager is likely to be frustrated by the non-technical problems that he or she would be required to solve every day — the most challenging of these being personnel-related problems. [9] Compounding this phenomenon are the various perquisites attached to being in management that can attract people into management for the wrong reasons (e.g., better pay, a better office, status within the company, a preferred parking space). But this sage advice has been largely ignored by the corporate world. The result is, in part, what we have today. Therefore, three groups must be educated:

—Software engineers. In most aspects, we are doing a good job of training software engineers. They are being taught software design, quality-oriented programming practices, testing methods and strategies, how to address security issues, and nearly every other aspect of software engineering. They are being exposed to new concepts and nuances, many of which are being published at a phenomenal rate. However, we are failing to accurately communicate the role of the software project manager to software engineers. The role of software project manager is not technical

in nature but is oriented toward organization, planning, scheduling, controlling, staffing and motivating. [19] [20] [21] Many software engineers still believe that the software project manager should be the most technically astute member of the team. Their training does not accurately portray what software project managers do or how this role differs significantly from developing code. The sports analogy of manager versus player is an appropriate one, [22] but it is not often conveyed in software engineering courses. Training in this area will at least improve the software engineers’ understanding of what they are getting into should they pursue a position in software project management.

- Software project managers. Most software project managers are not trained in appropriate management skills. [9] [21] This leaves them with little or no basis for actions and decision-making as managers of software engineering teams. This has resulted in the creation of a large number of “antipatterns.” An antipattern is a management action taken to solve a problem that actually makes things worse. [23] These are difficult to remove from the manager’s lexicon of problem solutions, since many were created by the manager or suggested by a colleague who also created antipatterns. These actions

| Senior Manager Rank | Project Manager Rank | Problem / Issue                    |
|---------------------|----------------------|------------------------------------|
| 1                   | 10                   | Insufficient front end planning    |
| 2                   | 3                    | Unrealistic project plan           |
| 3                   | 8                    | Project scope underestimated       |
| 4                   | 1                    | Customer/management changes        |
| 5                   | 14                   | Insufficient contingency planning  |
| 6                   | 13                   | Inability to track progress        |
| 7                   | 5                    | Inability to detect problems early |
| 8                   | 9                    | Insufficient number of checkpoints |
| 9                   | 4                    | Staffing problems                  |
| 10                  | 2                    | Technical complexity               |
| 11                  | 6                    | Priority shifts                    |
| 12                  | 11                   | Personnel not committed to plan    |
| 13                  | 12                   | Uncooperative support groups       |
| 14                  | 7                    | Sinking team spirit                |
| 15                  | 15                   | Unqualified project personnel.     |

Table 1: Differing perceptions of causes of project failure [27]

can effectively sabotage a software project. [21] One antipattern that demonstrates a clear lack of understanding of personnel issues is a common belief relating to software quality. More than one published paper has tacitly assumed that producing quality results is more costly than producing poor or mediocre results. [24] What this concept ignores is the fact that everyone wants to be associated with quality results and is motivated to achieve them, [25] which results in higher productivity. As one author put it, "Quality is free." [24] [26] There are many other misconceptions related to software development, but this one clearly demonstrates both a misunderstanding of what drives software engineers to excel and the assumption of a negative impact without data to support it. Since software will never be perfect, the software project manager working with the rest of the team plays a critical role in determining whether the system is good enough to ship. In addition to antipatterns, there are many sources of communication issues between the software project manager and senior management – the people who put this person into the position of software project manager. These issues have been documented and are presented in Table 1. [27]

- Human resources professionals. In most firms, the path to becoming a software project manager is neither clear nor well documented. [9] This can result in software engineers who wish to become software project managers being frustrated by the happenstance nature of such a transition, while those who may not have sought to become software project managers have this transition thrust upon them. Others may be attracted by the perquisites but not really motivated to pursue this change in their career path. [14] [21] These circumstances have resulted in what we have today – software project managers of varying quality, inconsistency in the success of software projects, general lack of knowledge of the role of a software project manager, and a lack of recognition of their value to a project. [8] [9]

### The Consequences of Our Inattention

In these situations, the software project manager often reverts to being a software engineer, effectively leaving the project without a manager. This leads to predictable project failure. A brief examination of the few software project manager courses reveals their content is mostly focused on software development, not on management issues such as personnel management, [9] negotiation, risk management, effective cost and schedule estimating methods, communication, planning methods, and so forth. As an example of personnel issues, consider this: software project managers and non-managers have very different value systems that lead to significant communication issues. Table 1 [27] summarizes these issues. At the very top

| Factor                         | Manager's Importance Rank | Non-Manager's Importance Rank |
|--------------------------------|---------------------------|-------------------------------|
| Salary                         | 1                         | 5                             |
| Job Security                   | 2                         | 4                             |
| Promotion/Growth Opportunities | 3                         | 7                             |
| Working Conditions             | 4                         | 9                             |
| Interesting/Challenging Work   | 5                         | 6                             |
| Personal Loyalty to Workers    | 6                         | 8                             |
| Tactful Discipline             | 7                         | 10                            |
| Appreciation for Work Done     | 8                         | 1                             |
| Help with Personal Problems    | 9                         | 3                             |
| Being in on things             | 10                        | 2                             |

Table 2: Value system differences between managers and non-managers [33]

of the list is what may be the most serious misconception on the part of software project managers. In fact, people do not work for money. Their motivation goes much deeper than that. Research has determined that people work for self-fulfillment, self-realization, and other factors. [28] [29] [30] Money is not a motivator in knowledge work. Money is a motivator in repetitive, assembly line work like one would find in a factory. The point is that value systems differences between software project managers and software engineers create a climate within which communication is inhibited. This predictably lowers productivity. Another significant example is "appreciation for work done." Software project managers are often puzzled by the resignation of a top performing software engineer who receives an extraordinary salary increase. If the recipient did not view it as a "thank you" for their efforts, the money would mean very little. Thus, we have a "reward paradox" wherein the most expensive reward is the least effective and the least expensive reward – a simple "thank you" – is the most effective. [31]

- Almost any improvement in what we teach software project managers will have a positive impact on projects.

- Senior managers. This group has been mostly ignored by our educational efforts. Due to their position in their respective companies, they tend to rely more on their own beliefs and conclusions than facts and data. [32] Stated another way, when at the lower levels of the power structure in an organization, one bases their decisions almost solely on facts and data. As one advances higher in the power structure, reliance on facts and data diminishes and decisions are based almost solely on intuition. The group at the top, senior managers, is responsible for making the most accomplished software engineers into software project managers, often with disastrous results. [9]

Compounding this situation is a natural communication gap between senior managers and project managers. (Table 2) [33] This may disappear as more software project managers get

into senior management positions, but for now, it needs to be contended with. What this group needs to accept is an overhaul of the way in which people become software managers and what perquisites (if any) are appropriate for such positions.

### A Natural Communications Gap

A software project manager's actions are based on his or her value systems. These value systems have been documented by different authors over several decades. [33] (See Table 2.) Note how different the values are between the two groups. The values of each group have changed little over the years and vary slightly from one researcher to the next. One area that highlights misconceptions on the part of many software project managers is salary. As a reward for work well done, software project managers will award an outstanding software engineer with either a salary increase or a bonus because salary is the manager's most important value. Unfortunately, the non-manager's most important value is appreciation for work done. This could be achieved via a simple "thank you" conveyed in private by the software project manager. [31]

Another challenge for software project managers is managing the differences in perceptions of what caused a project to fail. In Table 1 we see several obvious sources of problems for the software project manager. Remember, it is the senior management team that will determine the software project manager's future with the company.

One example of the communication issues in this domain is the perceived reason for project failure. The software project manager lists the most serious issue causing project failure as changes in the customer and/or the customer's management team. Changes of this sort are inevitable but are still listed as causes of failure. Unfortunately, this is consistent with a prevalent attitude regarding project failure, which is that much of what causes a project to fail is ambiguous. This usually allows the manager of a failed project to shirk responsibility for the failure. [34] This has the effect of preventing the software project manager from learning from failure. This phenomenon may explain why software engineering seems to continue to experience serious project cost and schedule overruns for so many years — we do not learn from our failures.

Another important aspect of the contents of Table 2 relates to the nature of these factors. Regardless of how they are ranked, none are, strictly speaking, technical. They break down into the following categories:

- Planning and scheduling. Contrary to the opinion of some, [35] planning and scheduling are not the same activity. They are closely related, but are not the same. Planning focuses on developing a list of tasks and subtasks to be performed and who will perform them. Scheduling deals with the order in which the tasks and subtasks must be performed, the length of time each will take and the start and end date of each. To most software project managers and

software engineers, planning and scheduling are uncomfortable tasks. A quote by Sir John Harvey Jones captures this discomfort: "People don't like to plan — planning is unnatural — it is far more fun to just do. And the nice thing about just doing is that failure comes as a complete surprise, whereas if you have planned, the failure is preceded by a long period of despair and worry."

- Controlling and tracking progress. A concern most software project managers have is runaway development. Unlike other engineering professions (e.g., civil engineering), software engineers can begin developing with a plan, schedule, design or even a set of requirements. This has caused many software project managers to develop and enforce a software development process that includes project reviews, milestones and GO/NO-GO decision points. "Earned value management" [36] and "earned schedule" [37] are often used in this regard.

- Personnel management and motivating. Most software project managers agree that personnel-related tasks are the most difficult to deal with. [9] There are many reasons for this, including the fact that work holds such an important place in the human psyche [28] [29] [30]. In addition, high technology workers like software engineers



## CIVILIAN TALENT IS MISSION-CRITICAL. LET'S GET TO WORK.



Work for Naval Air Systems Command (NAVAIR) and you'll support our Sailors and Marines by delivering the technologies they need to complete their mission and return home safely. NAVAIR procures, develops, tests and supports Naval aircraft, weapons, and related systems. It's a brain trust comprised of scientists, engineers and business professionals working on the cutting edge of technology.

You don't have to join the military to protect our nation. Become a vital part of NAVAIR, and you'll have a career with endless opportunities. As a civilian employee you'll enjoy more freedom than you thought possible.

Discover more about NAVAIR. Go to [www.navair.navy.mil](http://www.navair.navy.mil).

Equal Opportunity Employer | U.S. Citizenship Required



CHOICE IS YOURS.

are motivated to work on projects that advance the state of the art and provide them with talking points with colleagues that indicate they are exceptionally skilled. [9] This creates a particularly challenging situation for software project managers, since it is unlikely that all software projects they will manage — as well as the roles within those projects — will advance the state of the art. Rotating teams among projects can provide a means of ensuring that the software engineers do not continually work on projects they consider uninteresting and unchallenging.

- **Personnel and team composition.** Selecting the “right” team members is not nearly as easy and straightforward as it sounds. Creating a team of the best software engineers does not often lead to successful project completion due to a phenomenon known as “The Apollo Syndrome.” [38] It refers to documented evidence that the most highly skilled technical people rarely work together as a mutually supportive team. Even interviewing can be difficult due to company policies, various labor laws and, sometimes, the belief systems of potential candidates. This often leads software project managers to select people who think the same way the manager does. This results in a team that shares the same mistaken beliefs about software project execution as their project manager, which results in project failure. [19]

- **Communications with team, client, senior managers.** A common complaint among clients of software engineering projects is a lack of communication. [36] [21] Keeping the client up-to-date regarding project status, issues currently being addressed, accomplishments, and so forth help establish a positive relationship between the client and the development team. This results in an environment of collegial cooperation. Unfortunately, this is currently more often the exception than the rule. For example, while a project plan is practically a “pro forma” element of any project, many do not include a communications plan. [38] [39]

- **Risk management.** Risk is frequently viewed as something that just occurs unpredictably, requiring the development team to respond extemporaneously. This is a tactical and not very effective view. Certainly, unforeseeable events may happen that threaten the project and must be dealt with. However, there are several methods that can be employed to pre-emptively reduce the severity of risks early in the project. [21] These include:

- Identifying factors that could jeopardize the project.
- Assigning a potential monetary cost in case it occurs (or “fires”).
- Attaching a probability of occurrence to it.
- Setting aside a contingency fund to address the risk if it fires. [21]
- Mitigating the risk by eliminating what could

cause it to “fire.”

- Establishing a confidence factor regarding the project's ability to withstand or overcome the risk if it fires. [21]
- Complexity management. Accurately assessing how complex the project is likely to be and working to address challenges early on is another strategic approach to help ensure project success. This process involves assessing complexity via an inventory of complexity factors and challenging each factor to see if it can be mitigated or eliminated altogether. [21] [36] [40] [41]

This more or less holistic approach to the software project management “problem” is a dramatic shift away from the philosophy that technology will “win the day” and toward a philosophy that taps into the incredible potential of the people who are ultimately the foundation of the software engineering profession.

### Closing comments

The software engineering profession has tried to develop and use technology as a solution to its project problems for nearly 50 years, but the problems have persisted. Although it may seem a bit risky, it is time for us to try a new approach: focusing on how software projects are managed and, hence, how software engineers are managed. We have little to lose and much to gain.



## Homeland Security

The Department of Homeland Security, Office of Cybersecurity and Communications (CS&C) is responsible for enhancing the security, resiliency, and reliability of the Nation's cyber and communications infrastructure and actively engages the public and private sectors as well as international partners to prepare for, prevent, and respond to catastrophic incidents that could degrade or overwhelm these strategic assets. CS&C seeks dynamic individuals to fill critical positions in:

- Cyber Incident Response
- Cyber Risk and Strategic Analysis
- Networks and Systems Engineering
- Computer & Electronic Engineering
- Digital Forensics
- Telecommunications Assurance
- Program Management and Analysis
- Vulnerability Detection and Assessment

To learn more about the DHS, Office of Cybersecurity and Communications, go to [www.dhs.gov/cybercareers](http://www.dhs.gov/cybercareers). To apply for a vacant position please go to [www.usajobs.gov](http://www.usajobs.gov) or visit us at [www.DHS.gov](http://www.DHS.gov).

## ABOUT THE AUTHOR



**Dr. Lawrence (Larry) Peters** has more than 40 years experience in software engineering as a software engineer, instructor, project manager, and consultant. He has worked in the defense, aerospace, telecommunications, and other fields. His area of specialization is software project management. He has published several papers and books focusing on the education, evaluation and importance of the software project manager. He teaches software project management at the M.S. level in Spain via Skype and on-site short courses on software project management.

ljpeters42@gmail.com

## REFERENCES

1. Naur, P. and Randell, B. (Editors). (1969, January.) Software Engineering: Report on a Conference Sponsored by the NATO Science Committee. Garmisch, Germany, 7 to 11 October, 1968, Brussels, Scientific Affairs Division, NATO.
2. Rico, D.F. Short History of Software Methods. Downloaded off the web, August 2010. Referenced with author's permission.
3. Peters, L. (1983.) Software Design: Methods and Techniques. Prentice-Hall, Englewood Cliffs, New Jersey.
4. Schlumberger, M. (1991.) Software Engineering Management. Position paper in Proceedings of the 13th International Conference on Software Engineering. 152-153.
5. Boehm, B. (1981.) Software Engineering Economics. Prentice-Hall, Englewood Cliffs, N.J. 486-487.
6. Jensen, R. (2000, August.) Don't Forget About Good Management. CrossTalk, 30.
7. Weinberg, G. M. (1994.) Quality Software Management, Vol. 3: Congruent Action, New York, N.Y. Dorset House Publishing. 15-16.
8. Gulla, J. (2012, February.) Seven Reasons IT Projects Fail. IBM Systems Magazine.
9. Katz, R. (2013.) Motivating Technical Professionals Today. IEEE Engineering Management Review, 2013, 41, (1), 28-37.
10. Bourque, P. & Fairley, R.E. (2014.) Guide to the Software Engineering Body of Knowledge. Version 3.0, IEEE Computer Society. Retrieved from www.swebok.org.
11. Moreno, A., Sanchez-Segura, M-I, Medina-Dominguez, F., Peters, L. & Araujo, J. (2016, May.) Enriching traditional Software Engineering curricula with Software Project Management Knowledge. International Conference on Software Engineering Education and Training (CSEET), Austin, Texas.
12. Project Management Institute, PMBOK®. (2013.) A Guide to the Project Management Body of Knowledge. Fifth Edition. Project Management Institute, Newtown Square, Pennsylvania.
13. Project Management Institute. (2013.) Software Extension to the PMBOK Guide Fifth Edition. Project Management Institute, Newtown Square, Pennsylvania.
14. Peters, L. & Moreno, A. (2015, May.) Educating Software Engineering Managers Revisited – What Software Project Managers Need to Know Today. International Conference on Software Engineering Education and Training (CSEET). Florence, Italy.
15. Cusumano, M. (2004.) The Business of Software: What Every Manager, Programmer, and Entrepreneur Must Know to Thrive and Survive in Good Times and Bad. Free Press, New York, N.Y.
16. Townsend, R. (1970.) Up the Organization: How to Stop the Corporation from Stifling People and Strangling Profits, (commemorative edition). Knopf, New York, N.Y.
17. Chen, J. & Lin, L. (2004.) Modeling Team Member Characteristics for the Formation of a Multifunctional Team in Concurrent Engineering. IEEE Transactions on Engineering Management, 15 (2), 111-124.
18. Townsend, R. (1984.) Further Up the Organization: How to Stop the Corporation from Stifling People and Strangling Profits, (commemorative edition). Knopf, New York, N.Y.
19. Peters, L. (2008, May.) Getting Results from Software Development Teams. Microsoft Press Best Practices Series. Redmond, Washington.
20. Kerzner, H. (2013.) Project Management: A Systems Approach to Planning, Scheduling, and Controlling (11th ed.). New York, NY: John Wiley & Sons.
21. Peters, L. (2015, May.) Managing Software Projects On the Edge of Chaos – From Antipatterns to Success. Software Consultants International Ltd., Auburn, Washington. Kindle eBook.
22. Tarim, T. (2012, September.) Making a Transition from Technical Professional to .... IEEE Engineering Management Review, Vol. 10, No. 3, 3-4.
23. Silva, P., Moreno, A. & Peters, L. Software Project Management: Learning from Our Mistakes. IEEE Software, Vol. 32, Issue 3, 40-43.
24. Crosby, P. (1980.) Quality is Free. Signet, New York, N.Y.
25. Petre, M. & Damian, D. (2014, November 14, 16–21.) Methodology and Culture: Drivers of Mediocrity in Software Engineering? Foundations of Software Engineering Conference. Hong Kong, China, 829-832.
26. Crosby, P. (1995.) Quality is still free. McGraw-Hill, New York, N.Y.
27. Thamhain, H. Team Leadership Effectiveness in Technology-Based Project Environments. IEEE Engineering Management Review, 36(1), 165-180.
28. Herzberg, F. (1966.) Work and the Nature of Man. The World Publishing Company, Cleveland, Ohio.
29. Maslow, A.H. (1971.) The Farther Reaches of Human Nature. Viking Press, New York, N.Y.
30. McClelland, D.C. (1961.) The Achieving Society. Van Nostrand-Rheinhold, Princeton, N.J.
31. Grant, A. & Gino, F. A Little Thanks Goes a Long Way: Explaining Why Gratitude Expressions Motivate Prosocial Behavior. Journal of Personality and Social Psychology 2010, Vol. 98, No. 6, 946-955.
32. McAfee, A. (2010, January 7.) The Future of Decision Making: Less Intuition, More Evidence. Harvard Business Review.
33. National Study. (1993.) National Study of the Changing Workforce. Published by the Families and Work Institute, N.Y., N.Y.
34. Myers, C., Staats, B. & Gino, F. (2014, April 18.) 'My Bad!' How Internal Attribution and Ambiguity of Responsibility Affect Learning from Failure. Harvard Business School Working Paper, 14-104.
35. McConnell, S. (2000, July/August.) The Software Manager's Toolkit. IEEE Software.
36. Fleming, Q. & Koppelman, J. (2010.) Earned Value Project Management – Fourth Edition. Project Management Institute. Newtown Square, Pennsylvania.
37. Lipke, W. (2012, January.) Earned Schedule. Retrieved from lulu.com.
38. Belbin, R. (1996.) Management Teams – Why They Succeed or Fail. Butterworth Heinemann, London.
39. Dow, W. & Taylor, B. (2008.) Project Communications Bible. Wiley, New York, N.Y.
40. Maylor, H.R., Turner, N. W., & Murray-Webster, R. (2013, July/August.) How Hard Can It Be? Research-Technology-Management, 45-51.
41. Chipulu, M., Neoh, J., Ojako, U. & Williams, T. (2013.) A Multidimensional Analysis of Project Manager Competences. IEEE Transactions on Engineering Management, Vol. 60, No. 7, 506-517.

# Legal Liability for Bad Software

**Karen Mercedes Goertzel**

**Abstract.** This article focuses on lawsuits as a recourse for purchasers of defective COTS software — particularly safety-critical COTS software and software-controlled systems, such as software used in commercial aircraft, motor vehicles, unmanned aerial vehicles, medical devices, physical security systems, automated teller machines, commercial robots and industrial control systems, a wide variety of COTS diagnostic and sensor systems, and the whole growing panoply of cyber-physical devices and systems that collectively comprise the “Internet of Things.” [1]

## Background: Why Sue Software Companies?

Most commercial software contains serious flaws and defects, some of which are exploitable as vulnerabilities. This has always been the case, because it has proven impossible to fully test software, and while flaws/defects can be substantially reduced by adopting software assurance processes, [2] software products are never entirely free of flaws, either overlooked during development or intentionally left uncorrected at time of shipment. Over the last four decades, advances there have been exponential advances in software technology associated with the ubiquity of computers (first mainframes, then personal computers, and now mobile devices) and computer networks, the digital automation of once-mechanical controls for physical systems and processes, growing concerns over Information Warfare, the rise of hacking, malware, and computer crime, the open source movement, cloud computing, and on the horizon, the Internet of Things and The Singularity. With each advance, software has increased in size, complexity, ubiquity, exposure, and criticality due to by now almost complete human dependence on software's correct, reliable operation.

|         |                                                                       |                                                                                                                                                                                         |
|---------|-----------------------------------------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| 1974    | London Ambulance Service                                              | Poorly designed computer-assisted dispatch software delayed multiple ambulance dispatches, resulting in up to 30 deaths.                                                                |
| 1985-87 | Therac-25 radiotherapy machine                                        | Concurrent programming errors caused the machine to give patients massive radiation overdoses; in two cases, these were fatal.                                                          |
| 1991    | MIM-104 Patriot surface-to-air missile                                | A software miscalculation error prevented the missile from intercepting an Iraqi missile on its way to strike a military compound in Saudi Arabia; 28 American soldiers were killed.    |
| 1992    | F-22 Raptor flight control system                                     | A software error caused the \$150,000,000 fighter jet to crash.                                                                                                                         |
| 1994    | Royal Air Force Chinook engine control system                         | A software failure was found to be the likely cause of the helicopter's crash, which killed 29 people.                                                                                  |
| 1996    | European Space Agency (ESA) Ariane 501                                | A buffer overflow caused a hardware exception, leading to the rocket's crash upon launch.                                                                                               |
| 1999    | National Space and Aeronautic Administration (NASA) Mars Polar Lander | The software's misinterpretation of sensor data led to the craft's crash into Mars's surface.                                                                                           |
| 2003    | General Electric power grid monitoring system                         | A race condition in the system's software left a local power outage undetected, which escalated into a 48-hour blackout that extended across eight U.S. states and a Canadian province. |
| 2005    | ESA CryoSat-1 rocket propulsion unit                                  | The absence of a critical software command led to the satellite's crash upon launch.                                                                                                    |
| 2009    | Toyota (multiple models) electronic throttle control systems          | Software errors caused multiple incidents of sudden acceleration, resulting in crashes that killed 89 people.                                                                           |
| 2009    | Toyota Prius anti-lock braking system                                 | Software errors led to braking failures, resulting in at least three crashes and multiple injuries.                                                                                     |

Table 1. Some noteworthy software disasters [2]

But software has fallen short of its ability to meet expectations. And some of its failures have resulted in catastrophes of a magnitude that for other sectors and industries—from meat packing to automobiles—have raised outrage so great that the government was compelled to establish national regulations (and in a few cases, international) with whole agencies to enforce them. [3]

Inexplicably, despite these costly and fatal software-related disasters, and despite the recent spate of software malfunction- and defect-related automobile recalls at Toyota, Jeep-Chrysler, Honda and Volvo, and despite the software-related fraud committed by Volkswagen, nothing has inspired a general or sustained outcry or driven the government to undertake regulation of the software industry, as it has when other industries have been involved in safety or security catastrophes.

Despite decades of improvement of software safety and quality, most commercial software is still shipped with serious flaws and defects, some of which are exploitable as vulnerabilities. This has, in fact, been going on so long, it has simply become expected and accepted by most software customers. This is in large part because the software industry continues to audaciously insist that errors, defects and vulnerabilities in software are not only unavoidable, but represent a perfectly acceptable standard for software. [3] Moreover, they argue that the alternative of adhering to strict software assurance standards would not only be so costly as to make them uncompetitive and drive them out of business (or at least reduce their profit margins), it would severely inhibit innovation and delay the release of new features, which would harm the consumer. Apparently, it is with the consumer's best interests in mind that the software industry persists in producing poor quality, vulnerable products.

The truism that it is impossible to test software fully is widely accepted. This has been interpreted by many in the industry as free licence to adopt ultra-rapid agile methods and DevOps practices that enable developers to produce new software releases at an amazing rate but allow little time to fix coding errors or patch critical vulnerabilities that are found during testing and allow even less time to rethink designs that contain larger defects or requirements that are deficient. [4] Issues that cannot be mitigated by simple bug fixes, and which can only be discovered through thorough reviews and testing — or revealed when the software fails or is successfully hacked after deployment — are unlikely to ever be addressed. [5]

These problems have been reduced somewhat in larger software firms, such as Microsoft [6], that have the budgets to implement software assurance programs that help their developers reduce the overall number of flaws, vulnerabilities and even design defects in their software. But most software vendors — despite the publication of numerous “quality” and “secure” software methodologies over the past two decades — still do little to improve the quality or security of their processes. As a result, they ship products that contain easily avoidable flaws that their developers actually discovered during testing but chose not to correct because doing so would have delayed the release by a few days or weeks. (Even Microsoft, under pressure to adopt rapid agile development practices, has reduced the robustness of their security development lifecycle for products they develop under their agile regime.) The almost universal philosophy in the commercial software industry is “ship now, patch later.”

|                  |                                                                                                                 |
|------------------|-----------------------------------------------------------------------------------------------------------------|
| <i>Liability</i> | The state of being bound or obliged in law or justice to do, pay, or make good something; legal responsibility; |
| <i>Privity</i>   | The existence of a direct relationship between the parties of a contract;                                       |
| <i>Theory</i>    | (as in Theory of Case) Facts upon which a lawsuit will be founded, and which form the basis of a right to sue;  |
| <i>Tort</i>      | A legal injury or wrong committed upon a person or property independent of contract.                            |

Table 2. A few definitions [7]

Outside of specialist firms that focus on developing real-time embedded software for safety-critical or cryptographic systems, the software industry has never shown interest in self-regulation and has loudly resisted external regulation of its products' quality, safety or security. Governments have also been unwilling to regulate the software industry out of fear of stifling industry innovation and the significant economic growth the industry brings in countries where it is active. Even the European Commission, which is far more active in regulating standards for European Union industries than most other government bodies, has been reluctant to regulate the European software industry.

In the absence of regulation, any customer who falls victim to the catastrophic results of software faults and failures has only one recourse: the courts.

### Software and the Courts

The ambiguities surrounding software liability have been widely discussed in academic law literature (see Bibliography). These questions include whether software is a good or a service under the law, how to deal with a product as intangible and highly technical as software, and how binding End User License Agreements (EULAs) are under contract law, particularly in cases alleging gross negligence by the vendor or significant injury to the customer. [8]

The laws governing tort lawsuits have not yet been adapted to a world in which software is so universally present and prevalent. The literature on software liability (as typified by the footnotes throughout this article) reveals that the majority of software liability legal precedents were set in the 1980s. Little has changed since then in how contract and tort liability law are interpreted in reference to software. Lawyers who litigate such suits in the 2010s must spend a great deal of time and intellectual energy interpreting laws designed to govern commerce in physical products or professional services such as health care or architecture to figure out whether and how they apply to software.

### Suing for Breach of Warranty Under Contract Law

In many cases, recovery of damages resulting from software failures have relied on contract law, tort law or both. When software defects result in only economic losses (with a few extreme exceptions discussed in the section below on economic loss and tort liability), plaintiffs seeking restitution rely on the contract theory of "breach of warranty." Such suits are possible when software vendors include in their licence agreements express warranties about their product's performance, functional capabilities or attributes such as security and quality, or when warranties are implied by legal requirements imposed on all products,

such as merchantability and fitness for a particular purpose. A software defect often represents a deviation from the software's express or implied warranties.

To successfully sue under breach of warranty, a plaintiff must satisfy the requirements of privity, which means the software contract (license) must have been made directly between the buyer and the software vendor. Privity does not exist for software that is licensed to a reseller, other equipment manufacturer (OEM) or integrator, then resold to an end user. [9]

In addition to needing privity, an admissible breach of warranty suit must have a software licence that:

1. Includes express warranties about the product or its performance, or includes implied warranties.
2. Excludes enforceable limitation of liability clauses that preclude the purchaser from recovering more than the original purchase price of the software licence.

Despite a growing body of breach of warranty case law for software products, it remains unclear whether the Uniform Commercial Code (UCC) — which defines the widely accepted interpretation of civil contract law in the USA — even applies to software licenses because they do not transfer ownership of the actual software product from vendor to purchaser, but only transfer the right to use the vendor-owned product. If this is the case, software contracts are not product sales contracts but contracts for a service. In this case, it is a loan service (loan of use of the software). By extension, the software is not a product but a part of service delivery. This distinction between good and service is pertinent to determining not only how breach of warranty liability can be applied under contract theory, but also whether the theory of strict products liability can be applied to software as a product.

COTS software and software embedded in COTS products are frequently accepted as goods because they are mass-produced and transferred to unknown buyers who must rely upon the representations of the software vendor — two key prerequisites of being considered a product under the UCC. It doesn't matter whether the software is "bundled" with services such as the vendor's standard technical support package or user training; the software itself remains a product. By contrast, when software is developed under contract for a specific customer(s), or bundled into a much larger set of third-party services like system integration services, the ability to label the delivered software as a good or product rather than a service is cast into doubt.

### Civil Liability for Fraud

It may be possible to sue a software vendor for fraud if the plaintiff can prove that he or she accepted the defective software product only as a result of the vendor's willful misrepresentation of that product's performance. Courts have favourably ruled against software vendors for fraudulent misrepresentation when those vendors have misrepresented facts that are known exclusively to the vendor, such as undisclosed vulnerabilities, defects, malicious logic or failure to test the software. [10]

### Tort Products Liability

Because warranty law limits damages to recovery of product cost, plaintiffs often turn to tort theories of product

liability because contractual limitations do not normally apply to them and they have no requirements for privity or foreseeability. Tort liability arises when:

1. A defect in a product results in a failure of that product to operate safely as reasonably expected. For software, it doesn't matter whether the failure was accidental or caused by a Denial of Service attack exploiting the defect; and
2. Personal injury, death or property damage results.

According to the American Law Institute's *Restatement (Third) of Intentional Torts: Products Liability*, "A product is defective in design when the foreseeable risks of harm posed by the product could have been reduced or avoided by the adoption of a reasonable alternative design ... and the omission of the alternative design renders the product not reasonably safe." [11] Three tort theories of product liability are potentially applicable to COTS software: negligence, malpractice and strict products liability.

Under all three tort theories, the plaintiff can recover damages associated with the following:

- loss of valuable data. Data can be valuable due to security classification or regulated privacy;
- destruction of raw materials;
- destruction or loss of property other than the product itself.

Under a strictly limited set of extreme conditions, plaintiffs may also recover damages due to destruction of the product itself.

Another area of tort liability, intentional liability, is almost exclusively limited to acts of battery (as in "assault and battery"). However, some have questioned whether intentional liability might be applicable in cases when a product contains a defect, backdoor or malicious logic introduced through malicious developer or supply chain sabotage. Stay tuned for the first lawsuit to explore this possibility.

## Tort Theory of Negligence

General tort theories of negligence apply to developers, integrators, testers, maintainers, technical support personnel, trainers and anyone else that is under contract to deliver the services of manufacturing, maintaining or supporting the software product or another service in which delivery of the software product plays a part. Tort negligence permits permit recovery if a software defect can be proven to result from the service provider's failure to apply due care when designing or implementing ("manufacturing") the software.

Under negligence, the service provider cannot be held responsible for every software product or service defect that causes loss to the customer or a third party; responsibility is limited to only harmful defects that could have been detected and corrected through "reasonable" software practices.

A buyer of COTS software products or software-controlled devices or systems will find it very difficult to prove vendor negligence because the buyer lacks visibility into the "black box" software and its development process. The complex and mysterious nature of software, which is not well understood by anyone but its developers, forces buyers to trust the vendors' representation of that software. As a result, anyone suing for negligence must be prepared to incur significant expenses associated with hiring technology-savvy lawyers, researchers and expert witnesses. [12]

## Tort Theory of Malpractice

Tort theories of malpractice are a more rigorous, specialized form of tort negligence. They apply only to defendants in recognized, licensed professions, such as medicine, architecture and engineering. To sue under the tort theory of malpractice, the plaintiff must prove that the software's developer belongs to a recognized (ideally government-licensed) profession like software engineering and has failed to comply with the standards of that profession while engineering the defective software product.

A recognized system of professional licensure for software engineers has not yet been established nationwide, despite limited attempts at software engineering professionalization. For this reason, establishing tort liability of a software contractor — or a COTS software vendor — under tort theory of malpractice remains impractical. To date, no successful software malpractice suit has been undertaken in the U.S., though a few jurisdictions may have paved the way for successful litigation of computer malpractice claims. At least one court, the U.S. Court of Appeals for the 8th Circuit, agreed to hear an explicit computer malpractice case, *Diversified Graphics, Ltd. v. Groves*, 868 F.2d 293, in 1989. [13] By contrast, in *Columbus McKinnon Corp. v. China Semiconductor Co., Ltd.*, 867 F. Supp. 1173, 1182-83 (1994), the Western District Court of New York refused to recognize software programmers as professionals and rejected the malpractice suit. [14]

The ability to sue software engineers for malpractice is one of several arguments put forth in favor of establishing a broadly recognized professional licensing scheme for software engineers, and such licenses are already being issued in some U.S. states and Canadian provinces [15]. Moreover, a growing number of independent software developers and software engineering firms now invest in software malpractice insurance in anticipation of an increase in liability and malpractice lawsuits and the eventual professionalization of software engineering. This is particularly common among those who serve industries in which the profession of engineering is well understood, such as the industries that produce avionic and space systems, medical devices or industrial control systems for nuclear plants. [16]

## Tort Theory of Strict Products Liability

In most states, recovery is possible under strict liability tort theories regardless of any proven responsibility on the vendor's part. Neither negligence nor malpractice needs to be proven; strict liability can apply even if the vendor exercised reasonable care to avoid a defect and followed professional conduct standards. Proof that the defect was present and caused the plaintiff injury or property damage or loss is the only consideration.

Strict products liability for software is based on several premises:

1. The software is defective.
2. The software is a product, not a service. This tends to limit strict liability claims to COTS software and software embedded in COTS products.
3. The plaintiff used the software in the intended manner and did not introduce the defect through that usage. If the product requires user modification to operate, the vendor cannot be held liable for any injuries arising from defects introduced by the user's modifications.

According to *Restatement (Third) of Torts*, for a product to be subject to strict liability, it must be unreasonably dangerous to the ultimate consumer. Defects (including flaws/errors) in software and software-controlled products may be deemed unreasonably dangerous when:

1. They were present in the software code at time of product sale.
2. The design is defective, so that while it performs exactly as specified, the software is essentially designed.
3. The customer was not adequately warned by the vendor of the presence of known hazards, or of the limitations and parameters under which the software must operate.
4. The software was defectively designed, which means that while it performs exactly as the vendor intended, the intended performance is not reasonably safe.

The *Third Restatement* also suggests that if a manufacturer's knowledge of expected harms from defects can be proven, this will help support a claim of strict liability: "Because manufacturers invest in quality control at consciously chosen levels, their knowledge that a predictable number of flawed products will enter the marketplace entails an element of deliberation about the amount of injury that will result from their activity." [17] In other words, the sheer fact that manufacturers have quality control processes are admissions that they know their raw products are highly likely to be defective, and thus require careful scrutiny and correction before being released to the public.

Most courts accept that tort liability generally does not permit recovery of damages arising from the loss or destruction of the defective product itself. However, an exception can be made when defective software within a larger product is unreasonably dangerous and leads to the failure of the product, resulting in a calamitous event. This interpretation allows for recovery of product-destruction-related costs in most of the incidents in Table 1.

In such cases, the plaintiff doesn't need to prove that the defect and the associated product loss were clearly foreseeable due to the inherent nature of the product and its function, though one could clearly predict that defective flight control software could cause an airplane to crash or that failed controller software in an autonomous automobile could cause the vehicle to crash. In any case, such property losses would be recoverable under strict liability together with the personal injury damages recoverable under indisputable strict tort liability and the damage or destruction of whatever the airplane or automobile crashed into. [18]

Because strict liability and other tort lawsuits against software vendors are so expensive for plaintiffs to pursue, they have only proven practical when a class action lawsuit can be initiated.

### The Federal Trade Commission as Watchdog and Plaintiff

The Federal Trade Commission (FTC) has expanded its purview from suing high technology companies for unfair competition to suing for unfair or deceptive trade practices such as "software misrepresentation." Most FTC cases are settled out of court with results that are binding on the defendant. In February 2013, the FTC settled one of the

first liability suits involving inadequate software security against HTC America, Inc., manufacturer of Android-based smartphones. The FTC's complaint alleged "HTC's failure to employ reasonable security in the customization of its mobile devices," and stated that "had HTC implemented an adequate security program, it likely would have prevented, or at least timely resolved, many of the serious security vulnerabilities it introduced through the process of customizing its mobile devices." The FTC found that HTC had failed to follow commonly accepted secure programming practices and Android's documented secure customization practices, and that HTC also failed to adequately test the security of their products, to remove debug code before shipping, and to establish a vulnerability reporting mechanism.

In the settlement, HTC agreed to a number of remediation measures specified by FTC, including the establishment of a full-scale software security assurance program that would be independently validated every two years. At the time, the HTC case was heralded by many IT lawyers as one of the "most significant decisions in cybersecurity law." [19] In February 2016, the FTC reached a similar settlement with almost identical terms with router manufacturer ASUSTeK Computer, Inc., arising from multiple ASUS' AiCloud and AiDisk software and firmware. As with HTC, the FTC confronted ASUS for systemic deficiencies in its software development and system engineering processes. With these two settlements, the FTC has shown not only its willingness but also its competence in confronting and altering the bad practices of producers of vulnerable software.

### Whither Regulation?

In the U.S., there has been a long-running debate about whether the government should take a more active regulatory role to require vendors to produce secure computer software. One of their biggest concerns is that regulating the software industry could stifle innovation or drive software companies to other countries with less restrictive laws, as has happened in other U.S. industries. Based on historical precedent, there is also a case to be made that, absent strict regulation of commerce, exemplary punitive damage awards by courts and the fear of lawsuits they engender serve the same purpose as government regulation— they deter manufacturer misconduct and incentivize quality improvements. [20]

Proponents of regulation argue that continued reliance on the civil courts to improve the behaviors and products of the software industry unfairly favors the industry over injured consumers and is therefore unjust. Clearly agreeing with this sentiment, in 2009, the European Commission designated consumer protection for software buyers as a priority area for possible EU legislative action. [21]

In the current corporations-write-the-laws climate that exists in the U.S. (and, to a lesser extent, the U.K.), it is common to cynically assume that any regulations that did make it into law would be more likely to protect the software industry than the consumer — the exact opposite result proponents of regulation are seeking.

## REFERENCES

1. This article does not discuss issues of intellectual property law with regards to software, nor does it explore the tantalizing question of criminal liability when a software-controlled autonomous machine such as a self-driving automobile or a robot/embodyed artificial intelligence operates in a way that breaks the law. See Miller, C. C., (2014, May 13). When Driverless Cars Break the Law. The New York Times. Retrieved from <http://www.nytimes.com/2014/05/14/upshot/when-driverless-cars-break-the-law.html> See also: Paul, (2013, October 11). When Autonomous Vehicles Crash, Is the Software Liable? The Security Ledger. Retrieved from <http://securityledger.com/2013/10/when-autonomous-vehicles-crash-is-the-software-liable/>
2. List of Software Bugs. Wikipedia.org. Retrieved from [http://en.wikipedia.org/wiki/List\\_of\\_software\\_bugs](http://en.wikipedia.org/wiki/List_of_software_bugs); London Ambulance Service. Wikipedia.org. Retrieved from [http://en.wikipedia.org/wiki/London\\_Ambulance\\_Service#Computerisation](http://en.wikipedia.org/wiki/London_Ambulance_Service#Computerisation)
3. In his testimony to the House of Lords in 2006, Microsoft's Jerry Fishenden stated that the company was "making our platform as secure as we possibly can within the complex nature of software" [italics added]. See: House of Lords (U.K.) Science and Technology Committee. (2007, July 24). 5th Report of Session 2006–07, Personal Internet Security, Volume I: Report, Chapter 4: "Appliances and applications." HL Paper 165–I. Retrieved from <http://www.publications.parliament.uk/pa/ld200607/ldselect/ldstech/165/165i.pdf>
4. For examples, see: U.S. Department of Agriculture Food Safety and Inspection Service, "FSIS History" Retrieved from <http://www.fsis.usda.gov/wps/portal/informational/aboutfsis/history>; (2011, March 25). After the Triangle Fire: State and National Workplace Safety Reforms. [Blog post]. Political Correction blog. Retrieved from <http://politicalcorrection.org/factcheck/201103250003>; Constitutional Rights Foundation. (Fall 2008). Upton Sinclair's The Jungle: Muckraking the Meat-Packing Industry. . Bill of Rights in Action, Vol. 24 (No. 1). Retrieved from <http://www.crf-usa.org/bill-of-rights-in-action/bria-24-1-b-upton-sinclair-the-jungle-muckraking-the-meat-packing-industry.html>; U.S. Coast Guard. International Ice Patrol. Retrieved from [http://www.uscg.mil/history/articles/iip\\_history.asp](http://www.uscg.mil/history/articles/iip_history.asp); The British Dams Society. About Dams–Safety: The Dolgarrog Disaster, 1925. Retrieved from [http://britishdams.org/about\\_dams/safety.htm](http://britishdams.org/about_dams/safety.htm); Jensen, C. (2015, November 26). 50 Years Ago, 'Unsafe at Any Speed' Shook the Auto World. The New York Times. Retrieved from <http://www.nytimes.com/2015/11/27/automobiles/50-years-ago-unsafe-at-any-speed-shook-the-auto-world.html>; Piper Alpha. Wikipedia.org. Retrieved from [http://en.wikipedia.org/wiki/Piper\\_Alpha](http://en.wikipedia.org/wiki/Piper_Alpha)
5. With the adoption of agile development processes, software firms have come to rely increasingly on public beta and even alpha testing of their products. The jury is out, however, on whether crowdsourcing of testing has been spurred by a desire to improve the quality of software or merely to shift the vendor's cost of testing to the consumer.
6. Lohr, S. (2003, October 6). Product liability lawsuits are new threat to Microsoft. The New York Times. Retrieved from <http://www.nytimes.com/2003/10/06/business/product-liability-lawsuits-are-new-threat-to-microsoft.html>
7. Adapted from Black's Law Dictionary Free Online Legal Dictionary, 2nd Edition. Retrieved from <http://thelawdictionary.org>
8. The last of these questions has been clearly addressed in Kingsway Hall Hotel Ltd. v. Red Sky IT (Hounslow) Ltd. (2010). The Court ruled that Red Sky IT, a software vendor, could be held liable for damages beyond the value of the software it sold. This overturned contract law precedent, which had previously limited such damages to the purchase price of the software, The ruling also held that the exculpatory terms of Red Sky's software license agreement were unreasonable, and thus invalid. See: England and Wales High Court (Technology and Construction Court) Decisions, Neutral Citation Number: [2010] EWHC 965 (TCC), Case No: HT-08-111. Retrieved from <http://www.bailii.org/ew/cases/EWHC/TCC/2010/965.html>. Note that EULAs are no longer fully binding in Australia, either thanks to a 2011 revision of Australian Consumer Law that eliminated the right of developers and importers of business software to limit their own liability for consequential loss arising from faults in their software. See: Westmoreland, R. (2011, November 8). Unlimited Liability for Software Companies. HWL Ebsworth blog. Retrieved from <http://www.hwlebsworth.com.au/latest-news-a-publications/publications/intellectual-property-and-trade-marks/item/275-unlimited-liability-for-software-companies.html>. For French law, see: Rambaud, S. (2004, June 7). French Supreme Court strikes out limitation of liability provision from an IT contract where the software publisher has breached a material obligation. Bird & Bird blog. Retrieved from <http://www.twobirds.com/en/news/articles/2007/frenchsupremecourt-strikes-out-limitation-liability-provision>
9. Increasingly, when no privity of contract exists between the plaintiff and defendant, and the plaintiff was financially injured through use of defective software licensed through a third party, courts are allowing tort actions for economic loss. Otherwise, tort actions exclude economic loss except in the extreme conditions described later in this article. See Abdullah, F., Jusoff, K., Mohamed, H., & Setia, R. (2009, November). Strict versus Negligence Software Product Liability. Computer and Information Science, Vol. 2 (No. 4). Retrieved from <http://pdfs.semanticscholar.org/502a/604e4ff0e3b3028ff1ee9d82f63235055134.pdf>
10. Kaner, C. (1997). Software Liability. Self-published whitepaper. Retrieved from <http://kaner.com/pdfs/theories.pdf>
11. Simons, K. W. (2006). A Restatement (Third) of Intentional Torts? Arizona Law Review, Vol. 48. Retrieved from <http://www.bu.edu/lawlibrary/facultypublications/PDFs/Simons/RestatementThird.pdf>. Restatement (Third) of Torts: Products Liability (1998) is one of a series of tort law-clarifying restatements published by the American Law Institute, a group of America's leading legal scholars. The U.S. legal profession considers the Restatements authoritative in their interpretations of tort theories, including strict product liability. The Restatement scholars' omission of the reasonable expectations of consumers as an independent governing standard, or test, for whether a product is defective is significant; this "consumer expectations test," which was present in the Second Restatement, was missing from the more recent Third Restatement.
12. By contrast, customers who contract service providers to develop custom software – either from scratch, or through significant modification of existing software – are much closer to, and have more visibility into, the software and the processes used to produce it. For this and other reasons, strict products liability rarely applies to custom-developed software. Tort negligence and, in some cases, tort malpractice may potentially apply, however. See McCullagh, D. (2003, August 26). A Legal Fix for Software Flaws. CNET News.com. Retrieved from [http://news.com.com/2100-1002\\_3-5067873.html](http://news.com.com/2100-1002_3-5067873.html)
13. Buchanan, I. & Rooney. (2001, June 1). PC, Computer Malpractice Actions: Are They the Wave of the Future?. Retrieved from <http://www.bipc.com/Computer-Malpractice-Actions-Are-They-the-Wave-of-the-Future-06-01-2001/>; Kaner, C. (1996). Computer Malpractice. Software QA Quarterly, Vol. 3 No. 4. Retrieved from <http://kaner.com/pdfs/Malprac.pdf>
14. Kornecki, & Cunningham, (2003). Software Safety–Ethics, Professionalism, and Legal Issues. Proceedings of the 21st International System Safety Conference. Retrieved from [http://pages.erau.edu/~kornecka/papers/21SSC\\_ethics.pdf](http://pages.erau.edu/~kornecka/papers/21SSC_ethics.pdf); Also Op. cit. Buchanan, Ingersoll & Rooney.
15. Kowalenko, K. (2012, February 3). Licensing Software Engineers Is in the Works. The Institute. Retrieved from <http://theinstitute.ieee.org/career-and-education/career-guidance/licensing-software-engineers-is-in-the-works>. Also, Peters, D. K. (2009, October 30). Licensure for Software Engineers. Class notes for Engineering 7893: Software Engineering (Memorial University of Newfoundland). Retrieved from <http://www.engr.mun.ca/~dpeters/7893/Notes/presentations/SElicensing.pptx>
16. O'Brien, H. M. (2015, February 2). The Internet of Things: The Inevitable Collision with Product Liability. Product Liability Advocate. Retrieved from <http://www.productliabilityadvocate.com/2015/02/the-internet-of-things-the-inevitable-collision-with-product-liability/>
17. Op. cit. Simons.
18. Brennenman, S. R. (1986). Computer Malfunctions–What Damages May Be Recovered in a Tort Product Liability Action. Santa Clara High Technology Law Journal, Vol. 2 Issue 2. Retrieved from <http://digitalcommons.law.scu.edu/cgi/viewcontent.cgi?article=1025&context=chtj>
19. Rubens J. T. & Morse, E. A. (2013, November). Survey of the Law of Cyberspace: Introduction. The Business Lawyer, Vol. 69. Retrieved from [http://apps.americanbar.org/dch/thedl.cfm?filename=/CL320000/sitesofinterest\\_files/Cyberspace\\_Law\\_Survey\\_2012.pdf](http://apps.americanbar.org/dch/thedl.cfm?filename=/CL320000/sitesofinterest_files/Cyberspace_Law_Survey_2012.pdf)
20. Vandall, F. J. (1981). Undermining Torts' Policies: Products Liability Legislation. The American University Law Review, Vol. 30. Retrieved from <https://www.wcl.american.edu/journal/lawrev/30/vandall.pdf>
21. Op. cit. Simons.

## ADDITIONAL READING

Please contact the author to request a comprehensive list.

1. Bad Software Website. [http://badsoftware.com/?page\\_id=62](http://badsoftware.com/?page_id=62).
2. Armour, J. & Humphrey, W. S. (1993, August). Software Product Liability. Technical Report CMU/SEI-93-TR-13 ESC-TR-93-190. <http://www.sei.cmu.edu/reports/93tr013.pdf>.
3. Beard, T. R., Ford, G. S., Koutsy, T. M. & Spiwak, L. J. (2010, July 7). Tort Liability for Software Developers: A Law and Economics Perspective. The Journal of Computer and Information Law, Vol. 27. <http://www.phoenix-center.org/JCILTortLiability.pdf>.
4. Chong, J. Five part series in The New Republic.—(2013, October 3). Part 1, Bad Code: Should Software Makers Pay?. <http://newrepublic.com/article/114973/bad-code-should-software-makers-pay-part-1>—(2013, October 11). Part 2, Why Is Our Cybersecurity so Insecure?. <http://newrepublic.com/article/115145/us-cybersecurity-why-software-so-insecure>—(2013, October 22). Part 3, What You Don't Know About Internet Security Will Definitely Hurt You. <http://newrepublic.com/article/115281/what-you-dont-know-about-internet-security-will-definitely-hurt-you>—(2013, October 30). Part 4, We Need Strict Laws if We Want More Secure Software. <http://newrepublic.com/article/115402/sad-state-software-liability-law-bad-code-part-4>—(2013, October 31). Part 5, The Security Burden Shouldn't Rest Solely on the Software User. <http://newrepublic.com/article/115421/security-burden-shouldnt-rest-solely-software-user5>.
5. Răman, J. (2006, May 26). Regulating Software Development. Juris Doctorate Dissertation, University of Lapland Faculty of Law. <http://archive.nyu.edu/handle/2451/149936>.
6. Scott, M. D. (2008). Tort Liability for Vendors of Insecure Software: Has the Time Finally Come? Maryland Law Review, Vol. 67 Issue 2. <http://digitalcommons.law.umaryland.edu/mlr/vol67/iss2/5>
7. Spruell, J. & Kamal, M. (2002). Defective Software and the Issues of Malpractice, Negligence, Fraud, and Misrepresentation. Proceedings of the International Association of Computer Investigative Specialists (IACIS) 2002 Conference (Fort Lauderdale, Florida). <http://iacis.org/iis/2002/SpruellKamal.pdf>.
8. Turner, C. S. (1999). Software as Product: the Technical Challenges to Social Notions of Responsibility. Ph.D. Dissertation, University of California, Irvine. <http://users.csc.calpoly.edu/~csturner/fulltechreport.pdf>.
9. Vihul, L. (2014). The Liability of Software Manufacturers for Defective Products. The Tallinn Papers, Vol. 1 No. 2. [http://ccdcoe.org/publications/TP\\_Vol1No2\\_Vihul.pdf](http://ccdcoe.org/publications/TP_Vol1No2_Vihul.pdf)

## ABOUT THE AUTHOR



**Karen Mercedes Goertzel** is an internationally recognized cyber security, information assurance and software assurance expert with more than years of experience in research and analysis, technology strategy, solution specification and architecture, process definition and improvement, policy and guidance development, and technical communication. Her areas of subject matter expertise include system, software, and hardware assurance; application security (including security for web, mobile, Internet of Things, and cloud applications); information and communications technology; supply chain risk management; insider threat to information systems; and information protection, assured information sharing, and data loss prevention. Her passion is research, and she is the author or co-author of numerous published peer-reviewed articles and conference papers and of several book-length research studies.

## WE ARE HIRING

### ELECTRICAL ENGINEERS AND COMPUTER SCIENTISTS

As the largest engineering organization on Tinker Air Force Base, the 76<sup>th</sup> Software Maintenance Group provides software, hardware, and engineering support solutions on a variety of Air Force platforms and weapon systems. Join our growing team of engineers and scientists!

### BENEFITS INCLUDE:

- Job security
- Potential for career growth
- Paid leave including federal holidays
- Competitive health care plans
- Matching retirement fund (401K)
- Life insurance plans
- Tuition assistance
- Paid time for fitness activities

*Tinker AFB is only 15 minutes away from downtown OKC, home of the OKC Thunder, and a wide array of dining, shopping, historical, and cultural attractions.*



**Send resumes to:**  
76SMXG.Tinker.Careers@us.af.mil  
*US citizenship required*



Oklahoma City SkyDance Bridge, Photo © Will Hider

# Durability Challenges in Software Engineering

**Rajeev Kumar**  
**Suhel Ahmad Khan**  
**Raees Ahmad Khan**

**Abstract:** Practices show that software quality is not as high as it could be. Development organizations spend a relatively large amount of money and effort on fixing quality issues during late-stage development of software. One of the software qualities that has received significant attention in recent years is durable serviceability. Software with poor durability is likely to fail in a highly competitive market; therefore, software development organizations are paying more attention to ensuring the durability of their software. To be able to develop durable software cost-effectively, developers must investigate the connection between durability characteristics and software. In software engineering, durability is determined mainly by four characteristics; trustworthiness, human trust, dependability, and usability. To address the relationships among these characteristics, software designers analyze the durability requirements that may need to be implemented to fulfill these specific requirements of software serviceability. The main objective of this article is to gain an in-depth understanding of the relationship between durability characteristics and software.

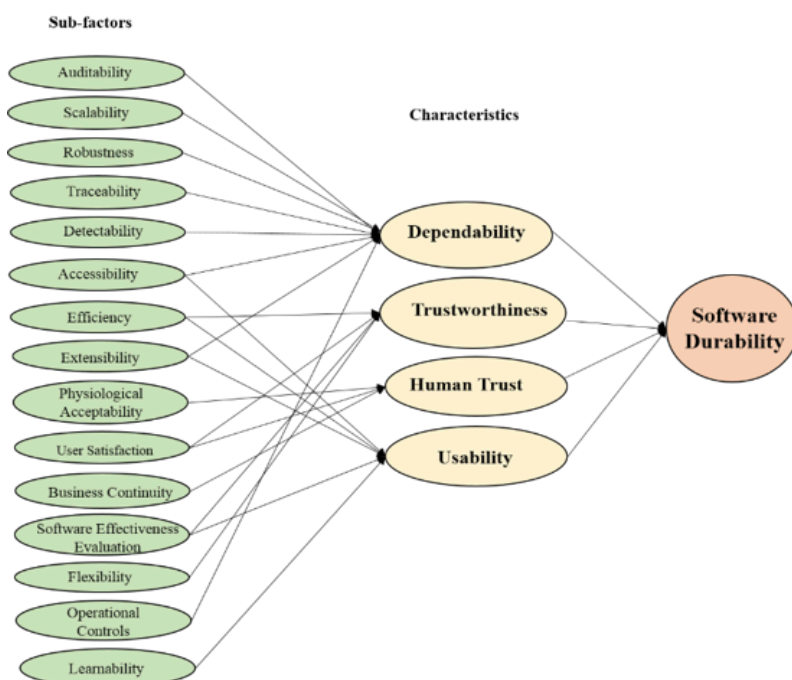


Figure 1: A structure of software durability factors [5]

## 1. Introduction

The software development life cycle contains many phases, including requirements engineering, design, coding, testing and debugging, and maintenance. Maintenance is regarded as the last stage of development [1]. But what if this phase continues for years? This is where the issue of serviceability comes in. Serviceability of software can be defined as the condition in which software is still useful or maintainable. Serviceability of software should be durable to achieve maintainability. Durability, in terms of software, is the time period during which software gives services.

There has been lot of work done in the field of software maintenance with regard to durability. In his article "When good software goes bad: the surprising durability of an ephemeral technology," Nathan Ensenger discusses problems in maintenance [2]. He also stated that there is a need to focus more on problems related to maintenance achievement. He stated that software durability is related to software serviceability, and it has been pointed out that achieving durability may enhance software serviceability. Service-oriented durable design of software is the aim of this study. The remainder of the paper is organized as follows: in section two, basic concepts of durability are defined. In section three, emergence of software durability is defined. In section four, successful strategies for developers are given. The paper's conclusions are contained in section five.

## 2. Basic Concepts of Durability

The evolving flexible environment of the early 21st century creates new challenges for all, including software developers [3]. Many programmers collaborate on each software project, with each programmer working on an individual software function or architectural component [4]. These components are often developed separately within a fixed time frame. When the time comes to bring it all together, a project manager integrates the different components as a required unit to achieve the desired result. This process makes software development a complex activity.

The complexity of software development leads to many problems, with design vulnerabilities being one of the most significant [6]. Some of the fundamental principles of design and its related systematic tools include availability, reliability, security and usability. Service of a software product is durable if it works efficiently and effectively to the user's satisfaction and for the expected duration. Many factors of software quality affect the serviceability of software, among them these few: trustworthiness, human trust, dependability, and usability. These factors affect durability directly, while factors like auditability, scalability, robustness, traceability, detectability, accessibility, efficiency, extensibility, physiological acceptability, user satisfaction, business continuity, learnability, effectiveness, flexibility, and operational controls affect durability indirectly [7] [8].

The meanings of these factors are different in a software scenario. Specifically, trustworthiness is assurance that software will perform as expected; human trust is a willingness to rely on the software with confidence; dependability refers to the ability to deliver service that can justifiably be trusted; and usability refers to how well software can be used by particular users to reach quantified results with effectiveness and satisfaction. The factors that affect durability directly and indirectly have positive and negative impacts on software service design as shown in Figure 1. A problem often

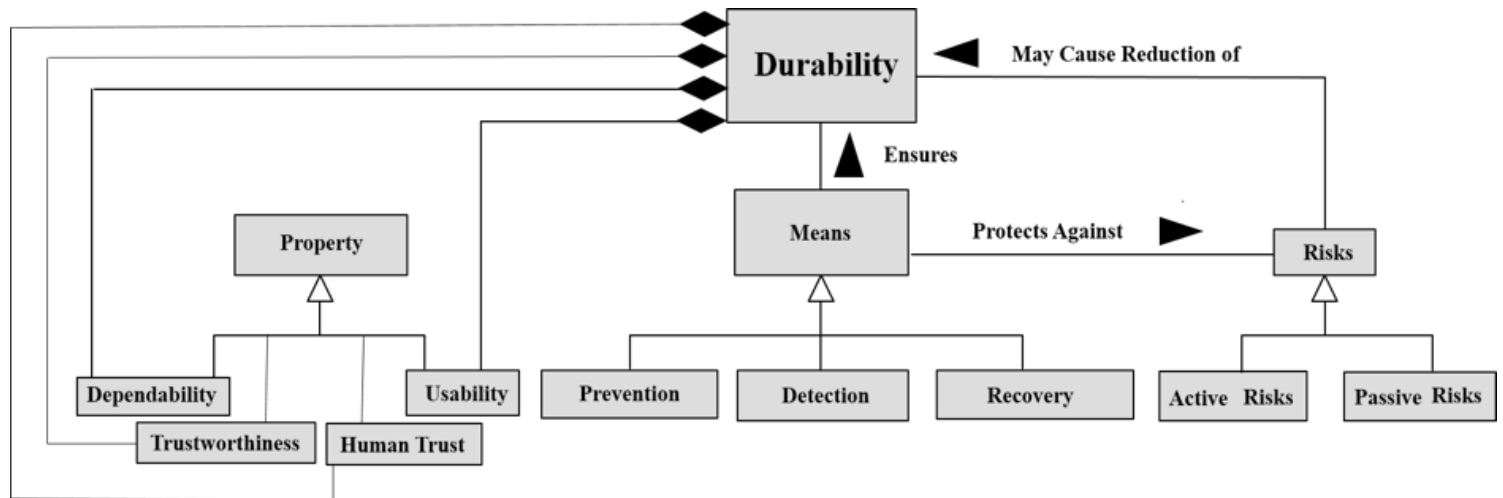


Figure 2: General activities for durability of software

comes while assuring durability by modifying the architecture of software. Modification is very expensive and time-consuming.

### 3. Emergence of Software Durability

Usually software is delivered without considerable security. This invites vulnerabilities. To mitigate these vulnerabilities, patching is done, which further results in more vulnerabilities in the future. It is normally expected that the design will remain serviceable for the entire life of the software and that services and qualities may come and go [9][10]. This leaves software designers and users to consider the relationship of the durability to the rest of the software architecture. Software durability is a term used to describe the usefulness and service life of a software product, which involves designing and construction with optimal maintenance [11] [12].

The term may also be used to describe the whole software development life cycle by comparing the service life of the design and its functional undesirability. A review of international research indicates that, except for operational components of software, all elements require different levels of service maintenance, repair, and replacement during the life cycle of the software development [13] [14]. The extent and strength of these services demands vary considerably, depending on how appropriately the durability of software and systems are synchronized and how accessible they are for regular maintenance, repair and replacement.

The durability of software may be expressed as a function of service quality and service life during the development cycle. There are three important service quality thresholds associated with durability: first, the quantified quality, recognized by the software developer or defined by minimum codes; second, the minimum acceptable quality, indicating the need for replacement; and third, failure. As shown in Figure 2, risk should be minimized to achieve durability of software. Two types of risk – active and passive – affect software during the development stage. Durability of software increases if risks are properly managed by means of detection, prevention and recovery.

### 4. Successful Strategies for Software Developers

Quality is a significant feature of software to be addressed, and durability is an important factor in evaluating software quality. Development of software design is not a one-time, built-in process; it is based on reuse of existing specifications in the market. The key point of this research work is the analysis of software's service-life relating to quality. This research is focused on increasing service life with secure and durable serviceability of software. Following are the steps which form a process that is effective in achieving durability when performed iteratively, incrementally and in parallel with the other activities, tasks and primary objectives:

- Establish durability as a powerful factor in software quality.
- Identify threat models of durability for degradation mechanisms.
- Develop a durability program plan that includes trustworthiness, human trust, usability and dependability.
- Identify and investigate their potential sources.
- Estimate the risks associated with durability for these respected assets.
- Arrange the risks according to the severity of the negative impacts.
- Identify and investigate the durable necessities of serviceability an arrangement as a benchmark for quality.
- Identify new attributes to provide a secure service-life of software for a specific duration.
- Identify durability subfactors, and determine their impact on overall software.
- Analyze software risks relating to durability.
- Make an objective to lessen the complexity of software design by establishing durability, which optimizes maintainability.
- Enhance the quality of software by improving service-oriented design.
- Calculate durability parameters using available or developed calculation models.
- If possible, update the ordinary architectural design tools for durability.

## 5. Conclusion

It is evident that generating a fully secure system is not possible; therefore, the creation of perfect and secure software cannot be considered the objective of evaluating software durability. Thus, the objective is to decrease the maintenance issue for longtime serviceable software. It could be concluded from the above discussion that the achievement of durable software is going to be a new challenge in the software industry. It is also as important as achieving any other attributes of quality, i.e., dependability, usability and supportability. One significant result has also been observed – that achieving durability early in development will raise the level of quality in the software. In this article, a general structure for assuring durable software is designed, and its processes are defined briefly.

## Acknowledgment

This work is sponsored by UGC-MRP, New Delhi, India, under F. No. 43-391/2014 (SR).

## REFERENCES

1. Kelly, C. & Erickson, S. (2015). *The Durability of Software*. Germany: Meson Press, 1-13.
2. Ensmenger, N. (2014). *When Good Software Goes Bad: The Surprising Durability of an Ephemeral Technology*. MICE (Mistakes, Ignorance, Contingency, and Error) Conference. Munich, 1-16.
3. Firesmith, D. G. (2003). *Common Concepts Underlying Safety, Security and Survivability Engineering*, Technical Note CMU/SEI-2003-TN033. Software Engineering Institute. Pittsburgh, Pennsylvania. 1-75.
4. Becker, S., Boskovic, M. & Dhama A. (2006, September). *Trustworthy Software Systems: A Discussion of Basic Concepts and Terminology*. Graduate School Trustsoft, Carl-von-Ossietszky University of Oldenburg, Germany.
5. Continuous Integration Strategies. Retrieved from <http://developerblog.redhat.com/2013/11/08/ci-strategies-1of3/>. Last visit 2016, March 3.
6. Becker, S., Boskovic, M. & Dhama A. (2006, September). *Trustworthy Software Systems: A Discussion of Basic Concepts and Terminology*. Graduate School Trustsoft, Carl-von-Ossietszky University of Oldenburg, Germany.
7. R. A. Khan, "From Threat to Security Indexing: A Causal Chain", *Computer Fraud & Security*, pp. 9-12, Vol 2009, Issue 5, May 2009.
8. Sahu K., Rajshree, "Software Security: A Risk Taxonomy", *International Journal of Computer Science & Engineering Technology*, pp- 36-41, 2015.
9. Continuous Integration Strategies. Retrieved from <http://developerblog.redhat.com/2013/11/08/ci-strategies-1of3/>. Last visit 2016, March 3.
10. *Software for Dependable Systems: Sufficient Evidence*. Retrieved from <http://www.nap.edu/read/11923/chapter/4>. Last visit 2016, March 2.
11. Kumar, R., Khan, S. A. & Khan, R. A. (2015). *Revisiting Software Security: Durability Perspective*, *International Journal of Hybrid Information Technology (SERSC)*. 8(2), 311-22.
12. Kluwer, W. (2015, May 21). *Starting your Software Security Assurance Program*. ITARC. Stockholm, Sweden.
13. Kumar, R., Khan, S. A. & Khan, R. A. (2015, October). *Durable Security in Software Development: Needs and Importance*. *CSI Communications*, 34-36.
14. Rogers R. L. (2004). *Principles of Survivability and Information Assurance*. Carnegie Mellon University. Pittsburgh, Pennsylvania.

## ABOUT THE AUTHORS



**Mr. Rajeev Kumar** is pursuing a Ph.D. in information technology from Babasaheb Bhimrao Ambedkar University (A Central University), Vidya Vihar, Raibareli Road, Lucknow. He has completed his master's degree in information technology from the same university. Kumar is a member of many national and international bodies, including ACM-CSTA, IBM-TechTarget, IAENG and BVICAM. His research interests are in the areas of software security, software durability and software risk. He is currently working in the area of software security durability.

**Babasaheb Bhimrao Ambedkar Central University  
Department of Information Technology  
Lucknow-226025, UP, India  
rs0414@gmail.com**



**Dr. Suhel Ahmad Khan** has earned his doctoral degrees from Babasaheb Bhimrao Ambedkar University, (A Central University), Vidya Vihar, Raibareli Road, Lucknow. He is currently working as an assistant professor in the Department of Computer Application, Integral University, Lucknow, UP, India.

Dr. S. A. Khan is a young, energetic researcher and has completed a full-time major project funded by University Grants Commission, New Delhi, India. He has more than five years of teaching and research experience. He is currently working in the area of software security and security testing. He has also published and presented papers in refereed journals and conferences. He is a member of IACIT, UACEE and Internet Society.

**Integral University  
Department of Computer Application  
Lucknow-226026, UP, India  
ahmadsuhel28@gmail.com**



**Professor Raees Ahmad Khan** has earned his doctoral degrees from Jamia Millia Islamia, New Delhi, India. He is currently working as a professor and Head in the Department of Information Technology, Babasaheb Bhimrao Ambedkar University, (A Central University), Vidya Vihar, Raibareli Road, Lucknow, India. Professor R. A. Khan has more than 13 years of teaching and research experience. His areas of interest are software security, software quality and software testing. He has published a number of national and international books, research papers, reviews and chapters on software security, software quality and software testing.

**Babasaheb Bhimrao Ambedkar Central University  
Department of Information Technology  
Lucknow-226025, UP, India  
khanraees@yahoo.com**



## CALL FOR ARTICLES

If your experience or research has produced information that could be useful to others, **CROSSTALK** can get the word out. We are specifically looking for articles on software-related topics to supplement upcoming theme issues. Below is the submittal schedule for the areas of emphasis we are looking for:

### **Process Improvement Best Practices**

*March/April 2017 Issue*

Submission Deadline: Oct 10, 2016

### **Operations & Maintenance**

*May/June 2017 Issue*

Submission Deadline: Dec 10, 2016

### **Model Based Testing**

*July/August 2017*

Submission Deadline: Feb 10, 2017

Please follow the Author Guidelines for **CROSSTALK**, available on the Internet at <[www.crosstalkonline.org/submission-guidelines](http://www.crosstalkonline.org/submission-guidelines)>. We accept article submissions on software-related topics at any time, along with Letters to the Editor and BackTalk. To see a list of themes for upcoming issues or to learn more about the types of articles we're looking for visit <[www.crosstalkonline.org/theme-calendar](http://www.crosstalkonline.org/theme-calendar)>.



# Upcoming Events

Visit <http://www.crosstalkonline.org/events> for an up-to-date list of events.

## **SLE 2016- ACM SIGPLAN**

31 October – 1 November 2016  
Amsterdam Netherlands  
<http://www.sleconf.org/2016/>

## **HCOMP 2016**

30 October – 3 November 2016  
Austin, TX  
<http://www.humancomputation.com/2016/>

## **34th IEEE International Conference on Computer Design**

3-5 October 2016  
Phoenix, AZ  
<http://www.iccd-conf.com/Home.html>

## **SC 2016- ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis**

13-18 November 2016  
Austin, TX  
<http://www.computer.org/portal/web/conferences/calendar>

## **FSE- International Symposium on the Foundations of Software Engineering**

13-19 November 2016  
Seattle, WA  
<http://www.cs.ucdavis.edu/fse2016/>

## **24th IEEE International Conference on Network Protocols (ICNP 2016)**

8-11 November 2016  
Singapore  
<http://icnp2016.comp.nus.edu.sg/index.html>

## **Better Software East Conference**

13-18 November 2016  
Orlando, FL  
<https://bsceast.techwell.com/>

## **The Ninth International Conferences on Advanced Service Computing**

19-23 February 2017  
Athens, Greece  
<http://www.iaria.org/conferences2017/SERVICE-COMPUTATION17.html>

# It's the simple things!

Once upon a time there lived a software developer named Joe. Joe loved developing software – it gave him great joy to weave a tapestry of code that would solve problems for users. Completing a project brought Joe great happiness, and he developed software all his adult life. And Joe was content.

One day Joe suffered a fatal exception, and he passed on. And Joe was rewarded for his life and went to The Hereafter.

In The Hereafter, Joe was allowed to develop software – it made his soul so happy. He happily wrote code, having a new 27" iMac, 2 TB solid-state drive, and true 100 GB Ethernet connection. And Joe was full of joy.

Eternity is a long time, and one day Joe grew bored. He asked "Is this all there is?" Joe was given a job as team lead, and he supervised his team to provide truly exceptional software. Unfortunately, his team was too small. Requesting new team members took lots of time, and the delays multiplied. The new team members needed new equipment – which required additional paperwork and more delays. And Joe was vexed.

Joe realized that small projects, which he had once happily solved, were just the tip of the iceberg. Even in his just reward – there were requirements issues. His team, while producing code, had delays and test equipment shortages and unexpected re-writes based on incomplete requirements. And Joe was troubled.

"Oh my gosh" said Joe. "I have seen this on an episode of Twilight Zone<sup>1</sup>. I passed onto my great reward, and I thought I was in Heaven. I realize now that I was a sinful developer. I ignored good exception handling. I truncated floating points without rounding. I promoted and demoted types without checking for proper representation. I failed to deallocate pointers and release them back to the operating system. I compared double-

precision numbers to zero without adding a tolerance. I coded in poor languages without strong typing. I failed to embrace Ada." With a gasp, Joe realized that he had not made it to Heaven, but to "The Other Place". And Joe was very sad.

Joe's sadness inevitably gave way to depression and his performance suffered. He was summoned to the office of the CTO of The Hereafter, and asked why his work performance in The Hereafter was suffering. Choking back tears, Joe explained that he realized his sins, and was prepared to spend the rest of eternity suffering in "The Other Place". "Oh my gosh" said the CTO. "You are in Heaven. I'm St. Peter, and you lived an exemplary life. This is just the way things are up here. We're doing the best we can. Updates to Salvation 3.5 are eons behind, HeavenOS9.3 is late, we can't find enough COBOL programmers<sup>2</sup>, and SOMEBODY promised a bunch of new features that were never meant to be in this iteration. If you think things are bad here – you should go to the "other place" and see how bad they have it!"

We all have problems creating software. Teams don't work right. Equipment is in short supply. It takes a long time to find qualified personnel. Test equipment is hard to procure, and doesn't respond exactly like the target software. Hard problem, right? Not really. You see, most supply chain problems are really simple problems that have mushroomed so that complex solutions are required to solve them.

At SFASU, I teach a course in both Project Management (for the IT majors) and Software Engineering (for the CS and CIS majors). I use a variety of books in the Software Engineering course, from recent textbooks and papers all the way back to Frederick Brook's classic, *The Mythical Man Month*. In the Project Management course, I use a variety of new papers, but basically teach from the PMBOK (Program Managers' Body of Knowledge). And in both classes, I stress Risk Management.

**David A. Cook, Professor**  
**Stephen F. Austin State University**  
[cookda@sfasu.edu](mailto:cookda@sfasu.edu)

**P.S.** Don't like the ending? Apologies – I'll elaborate. One of the web sites I frequent for class is [softwarethinktank.com](http://www.softwarethinktank.com). The article I had both classes read last year was <http://www.softwarethinktank.com/articles/%EF%BB%BF-top-5-reasons-supply-chains-fail/>. Well worth reading – it highlights the top 5 reasons... wait – you can read the web site name, right?

The top five reasons (from this web site) are:

- Swift, dramatic order cutbacks
- Long lead times causing delayed delivery
- Overemphasis on cost cutting
- Inaccurate demand forecasts
- Lack of risk management strategies

The first four are pretty much business decisions, and while they affect software – they are usually beyond the control of lower-level managers and developers. However, the fifth reason? As the article states – risks are inevitable. But if properly managed, they are fixable and – while they can't always be prevented – the risks can be mitigated and proactively managed. In fact, they must be. And, if not managed? Congratulations. You just became the weakest link in the supply chain.

As the above article also states - avoiding these common supply chain failures can save you time, money and resources. As the wise scientist Isaac Newton observed, "For every action there is an equal and opposite reaction." – your supply chain follows this same principle. Good (and simple) risk management allows you to act carefully and react appropriately.

**P.P.S.** Regarding the web site [softwarethink.com](http://www.softwarethink.com) – how can you not love and respect a website that has a link to the following?

### Project Delivery

A **Project Manager** is the one who thinks 9 women can deliver a baby in 1 month.

An **Onsite Coordinator** is the one who thinks 1 woman can deliver 9 babies in 1 month.

A **Developer** is the one who thinks it will take 18 months to deliver 1 baby.

A **Marketing Manager** is the one who thinks he can deliver a baby even if no man and woman are available.

A **Client** is the one who doesn't know why he wants a baby.

A **Tester** is the one who always tells his wife that this is not the right baby.

[http://www.dnserp.com/it\\_humor.htm](http://www.dnserp.com/it_humor.htm)

## NOTES

1. "A Nice Place To Visit", Season 1, Episode 28, 15 April 1960. And yes, I remember the original broadcast.
2. I am not implying that COBOL programmers won't make it to heaven. To be honest, I would think Fortran developers would be more at risk.



# Homeland Security

The Department of Homeland Security, Office of Cybersecurity and Communications (CS&C) is responsible for enhancing the security, resiliency, and reliability of the Nation's cyber and communications infrastructure and actively engages the public and private sectors as well as international partners to prepare for, prevent, and respond to catastrophic incidents that could degrade or overwhelm these strategic assets. CS&C seeks dynamic individuals to fill critical positions in:

- Cyber Incident Response
- Cyber Risk and Strategic Analysis
- Networks and Systems Engineering
- Computer & Electronic Engineering
- Digital Forensics
- Telecommunications Assurance
- Program Management and Analysis
- Vulnerability Detection and Assessment

To learn more about the DHS, Office of Cybersecurity and Communications, go to [www.dhs.gov/cybercareers](http://www.dhs.gov/cybercareers). To apply for a vacant position please go to [www.usajobs.gov](http://www.usajobs.gov) or visit us at [www.DHS.gov](http://www.DHS.gov).



NAV  AIR



CROSSTALK thanks the above organizations for providing their support.