

# Software Design for Reliability

Mike Silverman, Ops A La Carte LLC  
George de La Fuente, Ops A La Carte LLC

Key Words: Defect, Fault, Failure, Software, Reliability, Availability, Maintainability,

## SUMMARY & CONCLUSIONS

Despite the increased importance of the role that reliability plays in commercial product development, most companies are still unable to produce reliable software. The practice of software reliability is rare with few techniques that are compatible with commercial schedules and staffing capabilities. Should organizations wait for the next generation of tools, programming languages and development processes to improve their software reliability? No, the answer has always been within their reach.

By optimizing best practices for defect removal, development organizations can produce high reliability software. However, most organizations are not aware of the enormous potential for defect prevention that can be achieved before the software is even tested. Upstream improvements in software design for reliability (DfR) will generally produce greater returns than further investments in the test phase. This approach offers the option of implementing software DfR without making large changes to their development processes.

## 1 INTRODUCTION

Engineering teams must balance cost, schedule, performance and reliability to achieve optimal customer satisfaction.



Figure 1 – The “Big 4” Parameters to Balance

Reliability is no longer a separate activity performed by a distinct group within the organization. Product reliability goals, concerns and activities are integrated into nearly every

function and process of an organization. The organization must factor reliability into every decision.

Design for Reliability is made up of four key steps:

- assess customer’s situation
- develop goals
- write program plan
- execute program plan

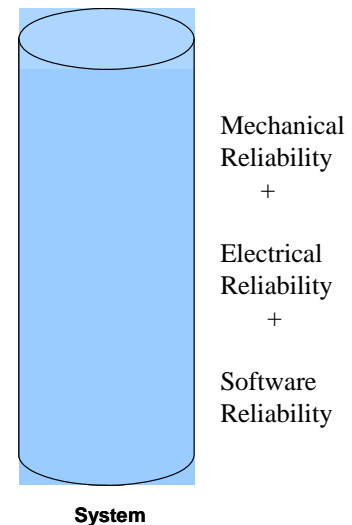
The focus is on developing reliable products and improving customer satisfaction. This is true for the electrical, mechanical, and software portion of the system.

### 1.1 Different Views of Reliability

Product development teams view reliability as the domain to address mechanical and electrical, and software issues.

Customers view reliability as a system-level issue, with minimal concern placed on the distinction into sub-domains.

Since the primary measure of reliability is made by the customer, engineering teams must maintain a balance of both views (system and sub-domain) in order to develop a reliable product.



## 2 SOFTWARE DESIGN FOR RELIABILITY

### 2.1 Software Quality vs. Software Reliability

Figure 2 – Different Views of Reliability

#### 1.2 Reliability vs. Cost

Intuitively, the emphasis in reliability to achieve a reduction in warranty and in-service costs results in some minimal increase in development and manufacturing costs. The use of the proper tools during the proper life cycle phase will help to minimize total Life Cycle Cost (LCC). To minimize total Life Cycle Costs (LCC), an organization must do two things: 1) Choose the best tools from all of the tools available and apply these tools at the proper phases of the product life cycle; and 2) Properly integrate these tools together to assure that the proper information is fed forwards and backwards at the proper times. This is Design for Reliability.

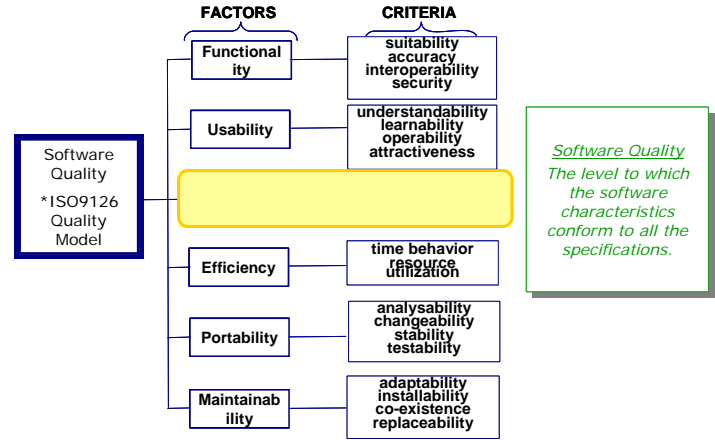


Figure 4 – Software Quality vs. Software Reliability

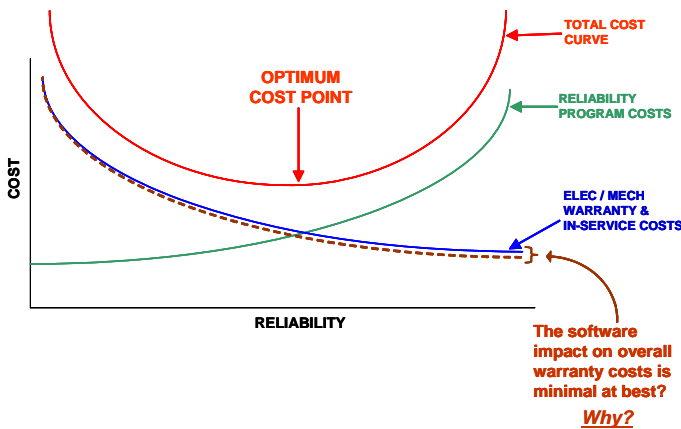


Figure 3 – Reliability vs. Cost

#### 1.3 Software Reliability and Cost

Software has no associated manufacturing costs and minimal replacement costs. Warranty costs and savings are almost entirely allocated to hardware. Software development and test teams are considered fixed product development costs.

If software reliability does not result in a perceived cost savings, why not focus solely on improving hardware reliability in order to save money? Sadly, some organizations take just this approach. But since they sell systems, not just hardware, sometimes customers or market forces cause companies to reconsider this approach. The primary root causes of embedded system failures are usually software, not hardware, by ratios as high as 10:1.

The benefits of improved software reliability are not in direct cost savings, but rather in increased availability of software staff for additional development due to reduced operational maintenance demands and improved customer satisfaction resulting from increased system reliability.

#### 2.2 Software Reliability Terminology

Software reliability is a measure of the software failures that are visible to a customer and prevent a system from delivering essential functionality.

##### 2.2.1 Defects

A flaw in software requirements, design or source code that produces unintended or incomplete run-time behavior. This includes Defects of Commission and Defects of Omission.

Defects of commission are one of the following: Incorrect requirements are specified, requirements are incorrectly translated into a design model; the design is incorrectly translated into source code; and the source code logic is flawed.

Defects of omission are one of the following: Not all requirements were used in creating a design model; the source code did not implement all the design; or the source code has missing or incomplete logic.

Defects are static and can be detected and removed without executing the source code. Defects that cannot trigger software failures are not tracked or measured for reliability purposes. These are quality defects that affect other aspects of software quality such as soft maintenance defects and defects in test cases or documentation.

##### 2.2.2 Faults

A fault is the result of triggering a software defect by executing the associated source code. Faults are NOT customer-visible. Example: memory leak or a packet corruption that requires retransmission by the higher layer stack.

A fault may be the transitional state that results in a failure. Trivially simple defects (e.g., display spelling errors) do not have intermediate fault states.

### 2.2.3 Failures

A failure is a customer (or operational system) observation or detection that is perceived as an unacceptable departure of operation from the designed software behavior. Failures are the visible, run-time symptoms of faults.

Failures MUST be observable by the customer or another operational system. Not all failures result in system outages. Note that for the remainder of this paper, the term “failure” will refer only to failure of essential functionality, unless otherwise stated.

### 2.3 Summary of Defects and Failures

There are 3 types of run-time defects:

1. Defects that are never executed (so they don't trigger faults)
2. Defects that are executed and trigger faults that do NOT result in failures
3. Defects that are executed and trigger faults that result in failures

Practical Software Reliability focuses solely on defects that have the potential to cause failures by detecting and removing defects that result in failures during development and implementing fault tolerance techniques to prevent faults from producing failures or mitigating the effects of the resulting failures.

### 2.4 Software Availability

System outages caused by software can be attributed to recoverable software failures, software upgrades, and unrecoverable software failures. Note that recoverable software failures are the most frequent software cause of system outages.

For outages due to recoverable software failures, availability is defined as:

$$A(T) = \frac{MTTF}{MTTF + MTTR}$$

where,

MTTF is Mean Time To [next] Failure

MTTR (Mean Time To [operational] Restoration) is still the duration of the outage, but without the notion of a “repair time.” Instead, it is the time until the same system is restored to an operational state via a system reboot or some level of software restart.

A(T) can be increased by either increasing MTTF (i.e., increasing reliability) using software reliability practices or reducing MTTR (i.e., reducing downtime) using software availability practices.

MTTR can be reduced by:

- Implementing hardware redundancy (sparingly) to mask most likely failures.
- Increasing the speed of failure detection (the key step).
- Software and system recovery speeds can be increased by implementing Fast Fail and software restart designs.

Modular design practices allow software restarts to occur at the smallest possible scope, e.g., thread or process vs. system or subsystem. Drastic reductions in MTTR are only possible when availability is part of the initial system/software design (like redundancy).

Customers generally perceive enhanced software availability as a software reliability improvement, even if the failure rate remains unchanged.

Availability Class	Availability	Timeframe vs. Mission Downtime (Unavailability Range)	
		Timeframe = 1 year	Timeframe = 3 months
(1) Unmanaged	90% (1 nine)	<b>36.5 days/year</b> <b>(52,560 mins/year)</b>	<b>9.13 days</b>
(2) Managed (good web servers)	99% (2 nines)	<b>3.65 days/year</b> <b>(5,256 mins/year)</b>	<b>21.9 hours</b>
(3) Well-managed	99.9% (3 nines)	<b>8.8 hours/year</b> <b>(525.6 mins/year)</b>	<b>2.19 hours</b>
(4) Fault Tolerant (better commercial systems)	99.99% (4 nines)	<b>52.6 mins/year</b>	<b>13.14 minutes</b>
(5) High-Availability (High-reliability products)	99.999% (5 nines)	<b>5.3 mins/year</b>	<b>1.31 minutes</b>
(6) Very-High-Availability	99.9999% (6 nines)	<b>31.5 secs/year</b> <b>(2.6 mins/5 years)</b>	<b>7.88 seconds</b>
(7) Ultra-Availability	99.99999% (7 nines) to 99.9999999% (9 nines)	<b>3.2 secs/year</b> <b>to</b> <b>31.5 millisecs/year</b> <b>(15.8 secs/5 years or less)</b>	<b>0.79 seconds</b>

Figure 5 – System Availability Timeframes

1) Software Failure Detection Mechanism	a) Slow Recovery Action	a) Fast Recovery Action
2) Implementation Example	b) Recovery Speed	b) Recovery Speed
3) Failure Detection Speed	c) MTTR	c) MTTR
	d) Mission failure rate	d) Mission failure rate
1) Very fast, automatic detection mechanism	a) Board reboot	a) Thread/Driver restart
2) Thread or Driver monitors	b) 90 seconds	b) (average) 50 mils
3) Up to 1 second	c) 91 seconds	c) 1.05 seconds
	d) 10 failures/mission	d) 914 failures/mission
1) Fast, automatic detection mechanism	a) Board reboot	a) Board restart
2) Watchdog timer reset	b) 90 seconds	b) 30 seconds
3) 5-10 seconds	c) 100 seconds	c) 40 seconds
	d) 9 failures/mission	d) 24 failures/mission
1) Slow, automatic det. Mech.	a) System reboot	a) System restart
2) Health message protocols w/retries	b) 4 minutes (average)	b) 100 seconds (average)
3) 30 seconds	c) 4 mins & 30 secs	c) 130 seconds
	d) 3 failure/mission	d) 7 failures/mission
1) Manual detection (No S/W mechanism)	a) System reboot	a) System restart
2) Reboot/restart via user intf	b) 4 minutes (average)	b) 100 seconds (average)
3) (avg detection time) 10 min.	c) 14 minutes	c) 11 mins & 40 secs
	d) 1 failure/mission	d) 1 failure/mission

Figure 6 – Software Availability

### 2.5 Reliable Software Characteristics Summary

Reliable Software has the following 3 characteristics:

1. Operates within the reliability specification that satisfies customer expectations. This is measured in terms of

failure rate and availability level. The goal is rarely “defect free” or “ultra-high reliability”

2. “Gracefully” handles erroneous inputs from users, other systems, and transient hardware faults, and attempts to prevent state or output data corruption from “erroneous” inputs.

3. Quickly detects, reports and recovers from software and transient hardware faults. Software provides system behavior as continuously monitoring, self-diagnosing” and “self-healing”. It prevents as many run-time faults as possible from becoming system-level failures.

2.6 Software Design for Reliability (DfR)Options

There are four basic options for software DfR:

1. Use formal methods.
2. Follow an approach consistent with Hardware Reliability Programs.
3. Improve reliability through software process control.
4. Improve reliability by focusing on traditional software development “best practices.”

In the next sections, we will review each of these options and show why “best practices” is really the best method to use.

2.6.1 Formal Methods

Formal methods utilize mathematical modeling of a system’s requirements and/or design in order to prove correctness. This is primarily used for safety-critical systems that require very high degrees of confidence in expected system performance, quality audit information, and targets of low or zero failure rates.

Formal methods are not applicable to every software project. They cannot be used for all aspects of system design (e.g., user interface design). Also, they do not scale to handle large and complex system development.

2.6.2 Follow an Approach Consistent with Hardware Reliability Programs

Software and hardware development practices are still fundamentally different. Architecture and design level modeling tools are not prevalent. They provide limited simulation verification, especially if a real-time operating system is required. Two-way code generation is a common feature. Software engineers find it difficult to complete a design with coding. All software faults are from design, not manufacturing or wear-out.

Software is not built as an assembly of preexisting components. Off-the-shelf software components do not provide reliability characteristics. Most “reused” software components are modified and not re-certified before reuse. Developing designs with very small defect-densities is the Achilles' heal.

There are no true mechanisms available for general accelerated software testing. Increasing processor speeds is usually implemented to find timing failures. These cannot be used for all aspects of system design (e.g., user intf. design).

Extending software designs after product deployment is commonplace. Software updates are the preferred avenue for product extensions and customizations. Software updates provide fast development turnaround and have little or no manufacturing or distribution costs.

Hardware failure analysis techniques (FMEAs and FTAs) are rarely used with software. Software engineers find it difficult to adapt these techniques below the system level.

2.6.3 Software Process Control Methodologies

Software process control assumes a correlation between development process maturity and latent defect density in the final software [1].

CMM Level	Inherent Defects/KSLOC	Delivered Defects/KSLOC
5	7.8	0.39
4	15.6	1.09
3	31.2	2.1
2	62.4	3.4
1	124.8	5.8

Figure 7 – Software Process Control Methodologies

If the current process does not produce the desired software reliability results, audits are performed and stricter controls are implemented in the subsequent iterations. Unfortunately, the nature of how process controls are implemented generally leads to slow increases in reliability improvement.

2.6.4 Defect Removal Potential of “Best Practices”

The best option for Software Design for Reliability is to optimize the returns from software development “best practices.” Figure 8 shows the difference in defect removal efficiency between inspections and testing. Most commercial companies do not measure defect removal in pre-testing phases. This leads to inspections that provide very few benefits. Unstructured inspections result in weak results. Software engineers simply do not know how to effectively apply their efforts as reviewers in order to find defects that will lead to run-time failures

Defect Removal Technique	Efficiency Range
Design inspections	45% to 60%
Code inspections	45% to 60%
Unit testing	15% to 45%
Regression test	15% to 30%
Integration test	25% to 40%
Performance test	20% to 40%
System testing	25% to 55%
Acceptance test (1 customer)	25% to 35%

Figure 8 – Defect Removal Potential of “Best Practices” [2]

Inspection results are increased by incorporating prevalent defect checklists based on historical data and assigning reviewer perspectives to focus on vulnerable sections of designs and code. By performing analysis techniques, such as failure analysis, static code analysis, and maintenance pre-reviews for coding standards compliance and complexity assessments, code inspections become smaller in scope and uncover more defects. Once inspection results are optimized, the combined defect removal results with format testing and software quality assurance processes have the potential to remove up to 99% of all inherent defects (Figure 9).

Design Inspections / Reviews	n	n	n	n	Y	n	Y	Y
Code Inspections / Reviews	n	n	n	Y	n	n	Y	Y
Formal SQA Processes	n	Y	n	n	n	Y	n	Y
Formal Testing	n	n	Y	n	n	Y	n	Y
Median Defect Efficiency	40 %	45 %	53 %	57 %	60 %	65 %	85 %	99 %

Figure 9 – Impact of Integrating Multiple “Best Practices”

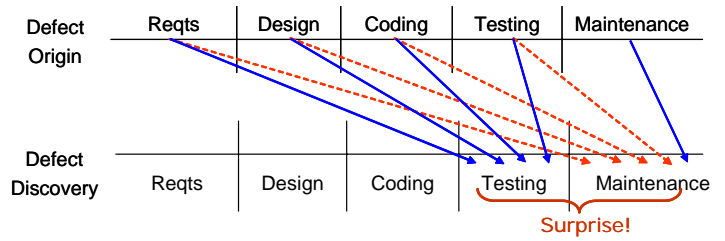
By redirecting their efforts upstream, most development organizations will see greater improvements in software reliability with investments in design and code inspections than further investments in testing (Figure 10).

Application Type	“Best in Class” Defect Removal Efficiency
System Software	94%
Embedded Software	95%
Commercial Software	90%
Military Software	96%
IT Software	73%
Web Software	72%
Outsourced Software	92%

Figure 10 – “Best in Class” Results by Application

Software DfR practices increase confidence even before the software is executed. The view of defect detection changes from relying solely on the test phases and customer usage to one of phase containment across all development phases (Figure 11). By measuring phase containment of defects, measurements can be collected to show the separation between defect insertion and discovery phases.

Typical Behavior



Goal of Phase Containment

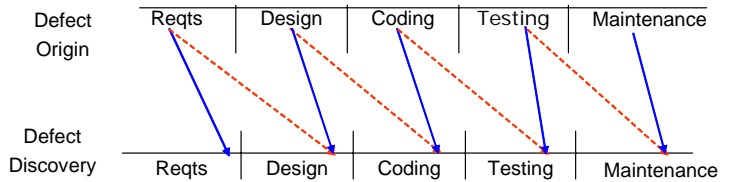


Figure 11 – Phase Containment of Defects and Failures

Reliability improvement goals can then be planned in phases (Figure 12): first reduce the customer impact by detecting most of the defects in-house (over 90%); second find most of the defects before the system test and integration phases (over 80%).

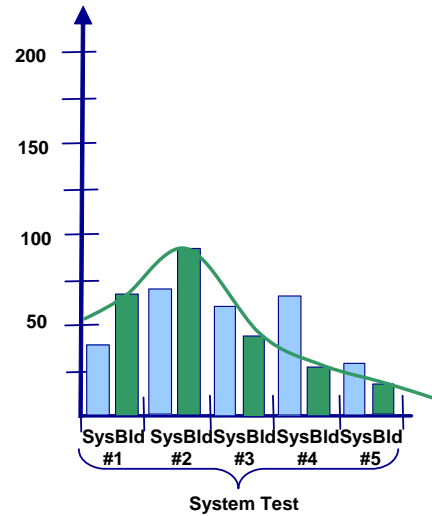


Figure 12 – DfR Phase Containment Behavior

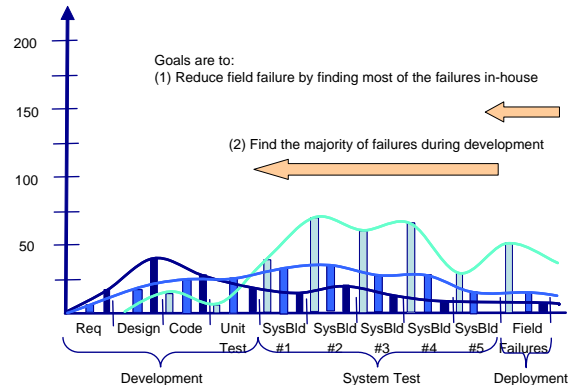


Figure 13 – DfR Phase Containment Behavior

## 2.9 Code Phase DfR Practices

Code reviews should be carried out in stages to remove the most defects. Properly focused design reviews coupled with techniques to detect simple coding defects will result in shorter code reviews. Code reviews should focus on implementation issues, and not design issues. Language defects can be detected with static and dynamic analysis tools. Maintenance defects are caught with coding standards pre-reviews. Requiring authors to review their own code significantly reduces simple code defects and possible areas of code complexity.

The inspection portion of a review tries to identify missing exception handling points.

Software failure analysis will focus on the robustness of the exception handling behavior. Software failure analysis should be performed as a separate code inspection once the code has undergone initial testing.

## 2.10 Test Phase DfR Practices

Unit testing can be effectively driven using code coverage techniques. It allows software engineers to define and execute unit testing adequacy requirements in a manner that is meaningful and easily measured. Coverage requirements can vary based on the critical nature of a module

System-level testing should measure reliability and validate as many customer operational profiles as possible. It requires that most of the failure detection be performed prior to system testing. System integration becomes the general functional validation phase.

## 2.11 Reliability Measurements and Metrics

Measurements – data collected for tracking or to calculate meta-data (metrics). Example: defect counts by phase, defect insertion phase, defect detection phase.

Metrics – information derived from measurements (meta-data). Example: failure rate, defect removal efficiency, defect density

Reliability measurements and metrics accomplish several goals:

- Provide estimates of software reliability prior to customer deployment.
- Track reliability growth throughout the life cycle of a release.
- Identify defect clusters based on code sections with frequent fixes.
- Determine where to focus improvements based on analysis of failure data.

Tools for software configuration management and defect tracking should be updated to facilitate the automatic tracking of this information. They should allow for data entry in all phases, including development. Also, they should distinguish code base updates for critical defect repair vs. any other changes, (e.g., enhancements, minor defect repairs, coding standards updates, etc.).

## 2.7 Requirements Phase DfR Practices

The requirements for software reliability are to: identify important software functionality, including essential, critical, and non-essential; explicitly define acceptable software failure rates; and specify any behavior that impacts software availability. We must define acceptable durations for software upgrades, reboots and restarts and we must define any operating cycles that apply to the system and the software in order to define opportunity for software restarts or rejuvenation. Example: maintenance or diagnostic periods, off-line periods, and shutdown periods.

## 2.8 Design Phase DfR Practices

The software designs should evolve using a multi-tiered approach. It includes the following three elements:

1. System Architecture: Identify all essential system-level functionality that requires software and identify the role of software in detecting and handling hardware failure modes by performing system-level failure mode analysis.

2. High-level Design (HLD): Identify modules based on their functional importance and vulnerability to failures. Essential functionality is most frequently executed. Critical functionality is infrequently executed but implements key system operations (e.g., boot/restart, shutdown, backup, etc.). Vulnerability points are points that might flag defect clusters (e.g., synchronization points, hardware and module interfaces, initialization and restart code, etc.). Identify the visibility and access major data objects outside of each module.

3. Low-level Design (LLD): Define availability behavior of the modules (e.g., restarts, retries, reboots, redundancy, etc.). Identify vulnerable sections of functionality in detail. Functionality is targeted for fault tolerance techniques. Focus on simple implementations and recovery actions.

For software engineers, the highest ROI for defect and failure detection and removal is low level design (LLD). LLD defines sufficient module logic and flow control details to allow analysis based on common failure categories and vulnerable portions of the design. Failure handling behavior can be examined in sufficient detail.

LLD bridges the gap between traditional design specs and source code. Most design defects that were previously caught during code reviews will now be caught in the LLD review. We are more likely to correctly fix design defects since the defect is caught in the design phase. Most design defects found after the design phase are not properly fixed because the scheduling costs are too high. Design defects require returning to the design phase to correct and review the design and then correcting, re-reviewing and unit testing the code!

LLD can also be reviewed for testability. The goal of defining and validating all system test cases as part of an LLD review is achievable.

### 3 CONCLUSION

Software developers can produce commercial software with high reliability by optimizing their best practices for defect removal.

Most software organizations are not aware that this type of software DfR is a viable option, which is why much of our initial contact is made via the hardware organizations.

Final Point: Companies do not require more tools, more testing or more processes to develop highly reliable software. The answer lies within the reach of their development team's best practices!

### REFERENCES

1. Jones, C., "Conflict and Litigation Between Software Clients and Developers", March 4, 1996
2. Jones, C., "Software Quality in 2002: A Survey of the State of the Art", July 23, 2002

### BIOGRAPHIES

Mike Silverman  
Ops A La Carte  
20151 Guava Court  
Saratoga, CA 95070 USA

e-mail: [mikes@opsalacarte.com](mailto:mikes@opsalacarte.com)

George de la Fuente  
Ops A La Carte  
20151 Guava Court  
Saratoga, CA 95070 USA

e-mail: [georged@opsalacarte.com](mailto:georged@opsalacarte.com)

Mike is founder and Managing Partner at Ops A La Carte, a Professional Business Operations Company that offers a broad array of expert services in support of new product development and production initiatives. The primary set of services currently being offered is in the area of reliability. Through Ops A La Carte, Mike has had extensive experience as a consultant to high-tech companies, and has consulted for over 200 companies including Cisco, Ciena, Apple, Siemens, Intuitive Surgical, Abbott Labs, and Applied Materials. He has consulted in a variety of different industries including

telecommunications, networking, medical, semiconductor, semiconductor equipment, consumer electronics, and defense electronics. Mike has 20 years of reliability, quality, and compliance experience, the majority in start-up companies. He is also an expert in accelerated reliability techniques, including HALT and HASS. He set up and ran an accelerated reliability test lab for 5 years, testing over 300 products for 100 companies in 40 different industries. Mike has authored and published 8 papers on reliability techniques and has presented these around the world including China, Germany, and Canada. He has also developed and currently teaches over 20 courses on reliability techniques. Mike has a BS degree in Electrical and Computer Engineering from the University of Colorado at Boulder, and is both a Certified Reliability Engineer and a course instructor through the American Society for Quality (ASQ), IEEE, Effective Training Associates, and Hobbs Engineering. Mike is a member of ASQ, IEEE, SME, ASME, PATCA, and IEEE Consulting Society and currently the IEEE Reliability Society Santa Clara Valley Chapter Chair and IEEE Consulting Society Santa Clara Valley Chapter Vice Chair.

George de la Fuente is a senior software reliability consultant and course instructor with OALC. George has over 20 years of product development and management experience with embedded systems. His professional software background spans the following industries: telecommunications, networking, gaming, and satellite operations. He has worked on many different types of products, including telecomm switches, telephony gateways, voicemail systems, large-scale routers and switches, gaming systems and satellite simulators. George has experience in commercial, start-up, and defense contracting companies. George has expertise in the following areas: full life-cycle development, rapid prototyping, zero-defect development, sustaining engineering, coding standards, systems testing, release management, software configuration management, project leadership, organizational management and program management. George educational background includes an M.S. degree in Computer Science from Santa Clara University and a B.S. degree in Mechanical Engineering from Yale University. George developed the core Software Reliability program, including training modules and services covering software reliability testing, software fault tolerance, software failure analysis, system availability design, and best development practices.