Converting System Failure Histories into Future Win Situations

Dolores R. Wallace and D. Richard Kuhn

Information Technology Laboratory

National Institute of Standards and Technology

Gaithersburg, MD 20899 USA

dwallace@nist.gov, *kuhn@nist.gov*

1. Introduction

Henry Petroski devotes an entire book to failures in engineering and lessons to be learned [1]. In his preface, he states "the concept of failure - mechanical and structural failure in the context of this discussion - is central to understanding engineering, for engineering design has as its first and foremost objective the obviation of failure." He further states "the lessons learned from … disasters can do more to advance engineering knowledge than all the successful machines and structures in the world."

We extend Petroski's views from mechanical and structural engineering into the domain of software system failures. Lessons learned can affirm proposed software engineering principles or help define new ones. Several industries, including telecommunications, space, business, and defense, were early drivers of computer technology. Within these, more and more systems are controlled by, or dependent on, software today than in the early years. Examination of software-based failures from many domains yields insight about possible common causes of failures and the means to prevent them in the next system or, at the very least, to detect them before the system is released. The purpose is to reduce costs by finding and detecting problems, that is, faults in any of the system artifacts, before systems are recalled from multiple users.

This study is on medical devices recalled by the manufacturers due to failures resulting from computer software faults. Findings may apply to other application domains. Like most industries, the health care industry depends on computer technology to perform many of its functions, ranging from financial management and patient information to patient diagnosis and treatment. The use of software in some kinds of medical devices has become widespread only in the last two decades or so. Their developers

had limited software experience and had to develop the expertise for avoiding preventable problems.[1] The Federal Food Drug & Cosmetic Act defines a medical device as:

> "an instrument, apparatus, implement, machine, contrivance, implant, in vitro, reagent, or other similar or related article, including a component part, or accessory which is:
>
> - recognized in the official Formulary, or the United States Pharmacopoeia, or any supplement to them,
> - intended for use in the diagnosis of disease or other conditions, or in the cure, mitigation, treatment, or prevention of disease, in man, or other animals, or
> - intended to affect the structure or any function of the body of man or other animals, and which does not achieve any of its primary intended purposes through chemical action within or on the body of man or other animals and which is not dependent upon being metabolized for the achievement of any of its primary intended purposes."

The failures cited in this study occurred in medical devices recalled by their manufacturers either in final testing, installation, or actual use from 1983 to 1997. No deaths or serious injuries resulted from these failures, nor was there sufficient information to estimate potential consequences had the systems remained in service. We examined the behavior exhibited by the failures, identified the software faults that may have caused them, provided some generic guidance, and assessed what could have been done to prevent or detect the classes of faults.

2. Characterization of the Data

While a medical device may be as simple as a tongue depressor, this paper is concerned with those devices containing software. The study includes devices in the categories of anesthesiology, cardiology, diagnostics, radiology, general hospital use, and surgery. Examples include insulin pumps, cardiac monitors, ultrasound imaging systems, chemistry analyzers, pacemakers, electrosurgical devices, and anesthesia gas machines. The failures have been observed as a response of the physical system and usually not as an obvious software fault.

---

[1] From the lecture by Lynn Elliott, "When Safe Patients Means Dependable Software," in the Lecture Series on High Integrity Systems, U.S. National Institute Standards and Technology, October 1995.

Data from the Food and Drug Administration (FDA) consist of four primary pieces of information: the recall number, the product name, a problem description, and a cause description. The code for the recall number yields the year of the recall and the general type of device. To protect the privacy of the manufacturers, we do not publish either the recall number or the product name. Our purpose is to use the problem and cause descriptions to understand the types of software faults underlying the failures and to abstract generic guidance about preventing and detecting the software faults before systems are released. Over time, manufacturers may have improved their software development processes and eliminated many factors contributing to these failures. Learning from recalls reinforces the need for software quality practices and provides specific guidance on how to prevent and detect faults.

For the Fiscal Years 1983-1991, there were 2,792 quality problems that resulted in recalls of medical devices, including devices that do not contain software. Of those, 165, or 6 %, were related to computer software. While the second group of data from 1992 - 1997 is not quite complete, the results are within the same ranges. We base our study on only the software recalls. Of the total number (383) of software recalls from 1983-1997, 30 % occurred in the years 1994 - 1996. One possibility for so high a percentage in three later years may be the rapid increase of software in medical devices. The amount of software in general consumer products is doubling every two to three years [2].

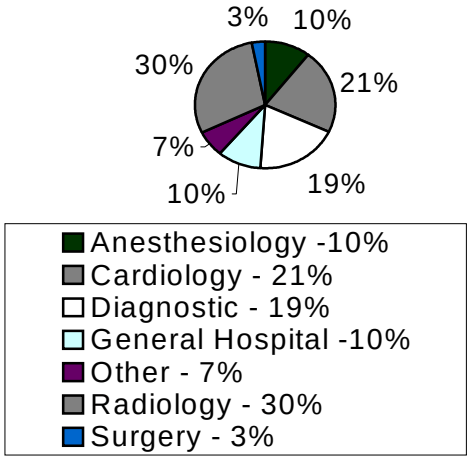

Figure 2.1 Failure distribution by device panel

The medical devices can be grouped into classification panels according to the primary function of the medical device. The medical devices fit into 7 panels: anesthesiology, cardiology, general hospital ,

diagnostic, radiology, general & plastic surgery, or other. Diagnostic includes chemistry, hematology, immunology, microbiology, pathology, and toxicology. The label "other" includes anything else such as obstetrics & gynecology or ophthalmology for which there were not enough recalls to be grouped into their own panels. The distribution of recalls by classification panel is shown in Figure 2.1. The pie wedges match the legend going clockwise, starting with anesthesiology, near the top, at 10 %. Some systems are more difficult to develop than earlier similar devices, such as in radiology where ultrasound and tomography are highly complex. The added complexity in algorithms and system interactions may have affected the failure rates for radiology.

3. Analyses of the Data

The information available provides some insight about the nature of the failures of the medical devices, but the vendor did not always state the actual software fault. We were limited in determining the fault by the problem and description data, without any mechanism for getting any further details. We derived the final fault class terminology from published taxonomies and reasoned how the various problems best fit, based on the problem and description as provided in the FDA database. From this limited information, we could discern the fault type for 342 failures; only these are discussed in the rest of this study. We derived the final fault class terminology from published taxonomies [3, 4] and reasoned how the various problems best fit. One lesson resulting from this study is that unless the fault and failure when collected have descriptions based on published classification systems, it is difficult to force the data into those classifications. While such classifications may fit during the actual fault discovery and resolution, they can be used only as guidelines for retrofit.

3.1 Fault distributions

In many cases there could have been 2 or 3 fault types contributing to a failure. For example, the observed behavior may indicate that two or more events had occurred at their boundary values simultaneously, resulting in an incorrect or unexpected response. Either the developers had not specified these events correctly in the requirements or the logic of the design or code failed to account for these simultaneous events. In the first case, the problem would be classified as a requirements problem (e.g., omission, ambiguity, conditions not considered) but the descriptions usually did not define the source of the problem. We classified most of these as logic problems at the point of failure, although we recognize

the value of better specification methods, specifically formal methods, in some of these situations and the prevalent belief that most faults are errors in the requirements specification.

In Table 3.1 the primary fault type is shown first, followed by one or more specific problems related to it, for example, "rate" following "algorithm" indicates a function performed at the wrong rate in an algorithm. We reduced the number of fault categories to the final list in Figure 3.1, placing the detailed fault type into its "best fit" class. For example, “incorrect change to counting” was placed under “calculation” because the error occurred in the counting algorithm and did not cause additional problems that would have fit under “change impact.” In Figure 3.1, the pie wedges match the legend of thirteen fault classes, going clockwise starting with calculation, near the top, at 24 %.

Table 3.1 Partial list of detailed fault categories

Accuracy; rounding	Logic; initialization
Algorithm; logic	Memory; dead code
Algorithm; rate	Missing code
Assignment	Missing information in user manual
Calculation; factor	Not enough information
Change impact; QA	Reinitialization
COTS; memory lost; size	Requirement-wrong formula
Improper impact of change	Sequence of operations; QA
Incorrect change to counting	Transposition
Initialization; data passing	Typo
Input; data passing	Units, calculation
Interface; parameter value	Volume

Among the fault types, logic faults appear at 43 %; with further details, some of these faults might fit into other classes. This class includes possible errors such as incorrect logic in the requirement specification, unexpected behavior of two or more conditions occurring simultaneously, and improper limits. The group “data” includes units, assigned values, or problems with the actual input data. The group “other” includes problems in COTS, EPROM, hardware, resources (e.g., memory), typos, and mistakes in translating requirements into code. For quality assurance, either the processes were not sufficient, or a new version was not validated.

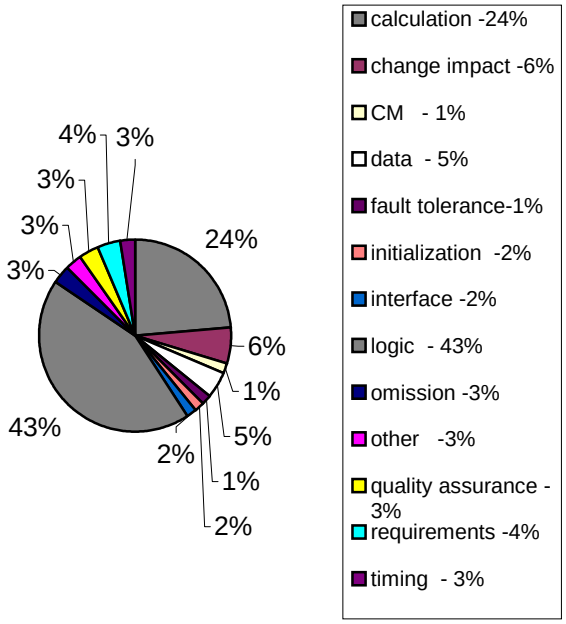

Figure 3.1 Fault class distribution

3.2 Prevention and detection of faults

These software recalls were distributed over 342 devices built by different vendors. What could have been done, individually, to prevent or detect each fault before the release of the device? We examined each fault in each of the thirteen classes and attempted to determine an answer to this question. By prevent, we mean some method applied by the development group before testing. By detect, we mean some method applied during testing or by quality assurance staff.

Obviously we cannot ascertain whether these methods were used or not. We have no evidence that more experienced companies used these more than inexperienced companies. Rather, we can indicate perhaps an affirmation that these are best practices consistent with today's focus on process [5]. First, by each class, for each fault, we considered various techniques/ methods for prevention and for detection. Next we reduced the results to a smaller, generic set for each fault class. Table 3.2 shows descriptions of typical problems, along with prevention or detection approaches. Only one problem per class is shown here; the complete tables are available at http://hissa.nist.gov/effProject/handbook/failure.

Certain methods appear frequently in the complete synopses as well as in the few examples provided in Table 3.2. Inspection may be either a prevention or a detection technique, where inspection as prevention is used in a broader sense than the original Fagan inspection [6]. Glass explains this broader view that is based on practitioners' presentations in workshops and conferences [7]. As prevention, inspection may include code reading and various static analyses. When inspection appears as a detection technique, it generally means the traditional Fagan-type inspection.

Table 3.2 Summary of fault types and guidance approaches

Fault class	**Prevention**	**Detection**
Calculation: Constants or table of constants incorrectly coded	Design, code reading for correct relationship between code and specified constant or table.	Code reading, inspection. Unit test.
Change impact: No verification against original design specification	Traceability analysis. Change impact analysis	Inspection of proposed changes. Regression test.
CM: Use of wrong master program for the software revision.	Use of CM tools.	Verification of appropriate master program. CM manager inspects the versions.
Data: System failed due to invalid input data	Assertions for invalid values, check of ranges that imply incorrect data. In design, set criteria for validation of input data. Code: implementation of input data validation.	Review for completeness of data specification, all data specifications are included in the user instructions. Focus inspection on data validation. Test against invalid data.
Fault tolerance: Excessive use of the program causes failure	Fault tolerance such as warnings to operators.	Stress/ volume test. Testing against boundary and abnormal conditions.
Initialization: For first execution, program fails to store initialization values for the succeeding run.	Document initial conditions for initial and consecutive runs. Design review.	Code review. Stress test (run the program multiple times).
Interface: Software does not properly interface with external device or other software component.	Trace requirements to design to code- all functions must interface to software module, output device, user or other system component. Examine spec for each interface.	Inspections, reviews. Integration test.
Logic: Incomplete or incorrect control logic	Apply traceability analysis from design to code. Walk through code against design.	Code Review. Inspection. Testing.
Omission: Vital system functions are missing.	Traceability, all interfaces, to user and test documentation. System test scenarios for requirements specification. Critical path analysis.	Inspections, reviews to examine traceability of functions. System Test.
Other: Incompatibility between two devices caused by typographic error	Code reading against algorithm specifications.	Walkthrough for on algorithms. Testing.
QA: Test plan was not implemented or executed appropriately.	Software project management oversight.	Project status review. QA process checks.
Requirements: Exceptional conditions were not specified in requirement specification.	Modeling. Formal methods. Traceability	Interface analysis. Requirement review. System test.
Timing: Two inter-react processes out of time synch with one another	Simulation. Design review. Code review.	Timing analysis. Integration test.

The class c*alculation* includes many types of algorithmic problems. Attention to algorithms and computations includes such details as verifying units, operators, intervals, limits, ranges, transformations from mathematical expressions into their implementation, and others. Sometimes even verifying that the original algorithm requirement is the correct version may require significant effort. Understanding how the specific computer will handle registers and floating point values is mandatory. Verifying all the issues for a calculation may require expertise outside computer science or software engineering. Often someone must verify that the algorithm is adequate for its intended use, e.g., increments used in the algorithm will be useful in the displayed output (neither too large nor too small to be meaningful).

While *change impact* is not necessarily considered a fault type, these cases indicate that failure to examine the impact of changes keeps other problem hidden. Another practice, performing a traceability analysis, is a prerequisite for performing change impact analysis. The analyses identify the region the proposed change will affect.

For *configuration management (CM)*, that is, keeping all artifacts correctly associated with the appropriate version of the system, several problems may have been due to the incorrect exercise of CM procedures. Simply using CM may have prevented others. In some cases, the problems stem not from improper software versions, but from selecting a software program incompatible with the hardware. This is also a problem of requirement specification; once hardware and software configurations are selected, the assumptions about each component need to be recorded as part of the CM history.

Problems in software programs can arise from input *data*. Data requirements for a program must be specified, entered in a data dictionary, and validated before the operation using the data is executed. The specification includes information such as units, acceptable range of values, the expected quantity or frequency with which values will change. The specification is published in the data dictionary of the database and in user instructions, emphasizing values that could cause program stoppage if they are out of range. The program itself may address some potential problems by containing assertions for input values or input omission, with actions to take when data are incorrect or missing. When a program is fielded, data in a database should be protected against database corruption. The software should facilitate an error-handling package to detect database corruption.

The *fault tolerance* category relates to safety-critical systems that should include facilities to handle abnormal or anomalous conditions.

Initialization is essential for enabling programs either to begin or to perform more than one cycle of a function. Default values for variables are a necessity, and likewise, re-initialization of a variable must be established. Explicitly documenting initial conditions in requirements through the code is essential. Code reviews and code reading need to focus not on whether initialization is specified, but specified according to good programming practices.

In a system, *interfaces* allow software to send and receive data to and from physical components of the system, other software modules and users. Clearly, the requirement specification must be accurate, complete, and consistent. A traceability scheme provides a basis for ensuring that all interfaces are addressed and included correctly. A well-developed test plan for integration testing must be executed to verify the interfaces between devices or software components.

Logic problems are the most numerous of the failures. But, while some failures of the devices did result from poor logic, the poor logic might have resulted from incorrect, incomplete, or inconsistent requirements or designs. For example, many problems resulted from interactions among different functions or at boundary conditions of a function. These were classified as logic problems, because the problem descriptions were provided at the point of failure in the code. But, we recognize that the source of the problems could have been requirements, design, or code. Two examples include 1) "When power is lost and then restored, system defaults to off status, which causes false information to operator and possible hazard to the operator " and 2) "When a second cartridge is in the other slot and detects an artifact condition, the monitor is prevented from alarming below set levels."

The class *omission* indicates a required system function is missing from the final implementation. Documentation is missing or not sufficient to install or operate the product.

Other faults too low in frequency to be classified separately include problems such as performance issues, I/O problems, typographical errors.

The role of *quality assurance* (QA) is to ensure that quality practices are defined in company standards and that they are used. Procedures are necessary for validation after modifications. The problems described in the recall data often cite that process checks were not made on the testing process and that testing was not performed after modifications. The problem descriptions do not reveal whether procedures for testing or other quality practices had been defined. Change impact analysis is a key task to ensure appropriate tests after modifications. While QA is not a direct fault type, it is a process problem. QA might have prevented some faults that resulted in failures. For this category, prevention techniques refer to discovering problems with QA. The responsibility for quality belongs to everyone on the project.

Some faults, such as omission, logic, and calculation, may have their genesis in the *requirements* specification but those included in the requirements category were clearly stated as requirements issues. This category demonstrates the need to develop, verify and validate a requirement specification, in some cases uses formal methods. The document specifying the product requirements is critical to the completeness and correctness of the software of the final product. The review of the requirements may require experts with different types of expertise to ensure that the requirements call for the right functions, appropriate algorithms, correct interfaces, function interaction, and other aspects.

Timing is vital to the execution of real-time applications. Processes that are interacting in real time must be completely synchronized.

3.3 Testing Required for Detection of Errors

How good is software testing in the medical device industry? One possible answer is to look at what conditions are required to trigger the faults that remained after release. Is the fault manifested in a single condition, or two or more conditions? Some of the failure reports (109 out of the complete set of 342) contained sufficient detail to determine the amount of testing would be required to detect the fault. For example, one problem report indicated that "if device is used with old electrodes, an error message will display, instead of an equipment alert." In this case, testing the device with old electrodes would have detected the problem. Another indicated that "upper limit CO2 alarm can be manually set above upper

limit without alarm sounding." Again, a single test input that exceeded the upper limit would have detected the fault.

Other problems were not so easily manifested. One report noted that "if a bolus delivery is made while pumps are operating in the body weight mode, the middle LCD fails to display a continual update." In this case, detection would have required a test with the particular pair of conditions that caused the failure: bolus delivery while in body weight mode. One vendor's description of a failure manifested on a particular pair of conditions was "the ventilator could fail when the altitude adjustment feature was set on 0 meters and the total flow volume was set at a delivery rate of less than 2.2 liters per minute."[2]

Only three of 109 failure reports indicated that more than two conditions were required to cause the failure. The most complex of these involved four conditions and was presented as "the error can occur when demand dose has been given, 31 days have elapsed, pump time hasn't been changed, and battery is charged." The remaining 233 failure reports did not contain sufficient detail to make a judgement on the number of test conditions required to demonstrate a fault; many described the cause as simply "software error." It is significant however, that of the 109 reports that are detailed, 98 % showed that the problem could have been detected by testing the device with all pairs of parameter settings.

Medical devices vary among treatment areas, but in general have a relatively small number of input variables, each with either a small discrete set of possible settings, or a finite range of values. For example, consider a device that has 20 inputs, each with 10 settings, for a total of 10^{20} combinations of settings. The few hundred test cases that can be built under most development budgets will of course cover less than a tiny fraction of a percent of the possible combinations. The number of *pairs* of settings is in fact very small, and since each test case must have a value for each of the ten variables, more than one pair can be included in a single test case. Algorithms based on orthogonal latin squares are available that can generate test data for all pairs (or higher order combinations) at a reasonable cost. One method makes it possible to cover all pairs of values for this example using only 180 test cases [8]. This amount of test effort should be practical for most devices in the categories reviewed in this paper.

[2] The policy of the National Institute of Standards and Technology is to use metric units of measurement in all its technical papers. In this document however, works of authors outside NIST are cited which describe measurement values in certain non-metric units, and it is not appropriate to provide converted values.

.

4. Lessons Learned

The information about the software faults that caused these system failures provides valuable lessons and affirmation of quality practices. These concern development procedures, assurance practices during development & maintenance activities, and testing or assurance strategies. For this domain, methods to prevent and detect faults should focus on logic and calculation errors. For logic, methods should address improved handling of various conditions, assumptions, and interactions among functions. Attention must be given to the details of calculations, such as verifying that the correct algorithm has been specified in the first place or that the programmed operators and increments are correct. The lessons addressed below are based on problems observed in this study; they stood out as prevalent problems for this set of data and are related to the faults indicated in the fault tables in Section 3. The practices suggested in this paper will likely vary in other domains. Studies of other domains may provide a variation of the lessons learned here along with a roadmap for selecting the best quality strategy within a company or domain from more general guidance on quality practices. Other guidance discussing general good practices on software development and assurance includes the Capability Maturity Model, and NIST documents on life cycle development and assurance, and verification and validation [5], [9], [10].

Development & Maintenance

While software development processes are already well defined by such models as the CMM, this study indicates particular practices that would help prevent the faults that led to these specific failures. For example, training in the characteristics of the computer on which the device will reside might have prevented some of the computation errors concerning registers. Training in the application domain concerning how the outputs of functions interact and will be used by the operator might have prevented wrong interval size that produced unusable charts. Attention to details such as checking and verifying one's work as related to the specifications for that work might have prevented several problems. A member of the software team with experience in the application domain might have caught several problems. Many logic faults stemmed from misunderstanding of how various functions interact, that is, under certain conditions, and in some cases, that they would interact at all. A traceability map can

identify inconsistencies or incompleteness. The following list highlights some of the practices recommended for development and maintenance tasks:

- Complete specification of requirements, emphasizing conditions and interactions of functions. Formal methods may be considered for highly complex systems.
- Traceability of the development artifacts: requirements to design to code to user documentation and to all test documentation, especially location of source of faults. The analysis should be conducted forward and backward.
- Traceability and configuration management of all changes to the product.
- Software configuration management.
- Change impact analysis.
- Expertise in the application domain by at least one person involved with quality practices such as requirements analysis, inspections, testing.
- Daily attention to details of the current process, the mapping to results of the previous process, and personal reviews of one's work.
- Training.

Assurance Practices

The quality of software is the responsibility of everyone involved in its development. Practices listed above for development and maintenance are a few enabling factors in establishing an environment in which this responsibility is recognized. Other tasks fall into the category of quality assurance, but may be performed by the persons engaged in development of the software artifacts or by those separated organizationally under some quality assurance name. Every artifact of development processes needs to be scrutinized. The list of techniques supporting this scrutiny is long and is published elsewhere. Instead we focus on the few techniques whose value is indicated by the faults causing the failures of these devices. The inspection technique, as per Glass[7], can be perceived as a variety of techniques that examine artifacts, ranging from requirements to design to code to test cases. Such techniques may include code reading, formal inspection meetings, review by programmer using various analytic techniques, and focused inspections. Porter and Votta describe scenario-based inspections in which participants looked for certain classes of errors [11]. Having some knowledge of the prevalent classes of

errors of the product they are examining may help the participants in conducting these types of inspections. The following list summarizes these suggestions:

- Focused review, inspection of the artifact against the types of faults characteristic of the domain, and the vendor's history.
- Traceability analysis, especially focused on completeness.
- Mental execution of potentially troublesome locations (e.g., an algorithm, a loop, an interface).
- Code reading.
- Recording of fault information from the assurance activities and better usage of this information.
- Recording, during development and quality assurance activities, of behavior indicating faults.
- Checklists, questions, methods designed to force faults to manifest themselves.
- Formal or informal proof of algorithm correctness.
- Use of simulation in complex situations where several interactions may occur, especially involving several components of the system.

Testing

Testing is part of the general quality practices, with unit, integration, and system testing all conducted. The failures in this study indicated specific test strategies might have been useful in detecting problems before the systems were delivered. In particular, nearly all could be detected by testing all pairs of input values, a test criterion that should be practical for most of the devices. Many failures were recognized by behavior of the system, for example, a part moved unexpectedly, or medication was provided at an incorrect rate. Most of these resulted from logic faults, so test cases in complex systems should attempt to force behavior manifesting faults. In some cases, the systems were updated versions, so previous test histories may also have been helpful. The list summarizes these points:

- Test cases aimed at manifesting prevalent behavior observed by device operators.
- Test cases that cover all pairs of input values, and all three or four-way combinations where practical.
- Stress testing.
- Change impact analysis and regression testing.

- SCM release of versions with evidence of change impact analysis, regression testing; validation of changes.
- Integration testing focused on interface values under varying conditions.
- System testing under various environmental circumstances, with some conditions, input data incorrect or different from expected environmental conditions.
- Recording of test results, with special recording of all failures and their resolution, by failure and behavior of the system, and by fault type of the software.

5. Conclusions

This study yielded information affirming use of quality practices and identifying approaches for using fault and failure information to improve development and assurance practices. The nature of several faults indicates that known practices may not be used at all or may be misused. An important conclusion is that the use of many generally accepted quality practices, rather than use of a "silver bullet" is significant toward reduction of system failures. Questions remain for further research:

- If the practices were not used, what can be done to make them more readily usable?
- If the practices were used, why did they fail to prevent or detect the fault?
- What methods not yet generally accepted may help to prevent some faults and subsequent failures?

The analysis in this study demonstrates that different application domains may have different prevalent fault. Suggestions for improvement of assurance practices include:

- gathering failure and fault data,
- understanding the types of faults that are prevalent for a specific domain, and
- developing prevention and detection approaches specific to these.

The subject of this study, failures of medical devices, is dealing with a relatively young industry, often new to adding microprocessors to devices. As experience with software development and complexity of the software grow, the prevalent fault classes may change. In domains with a long history of software, the classes may also differ. In newer applications such as Electronic Commerce, which rely on newer technologies, operating systems, and languages, we would anticipate perhaps new fault classes for the domains as well as for the underlying software technologies. Data collection and

analysis can help to identify the most prevalent faults and the areas where better methods are needed to prevent and detect them before system delivery.

This paper has shown that valuable lessons can be learned from system failures involving software. Some lessons may apply specifically to the application domain of study while some apply universally. It is important to continue this research on failures using modern technologies in various domains. The authors may be contacted by anyone willing to supply data.

6. References

[1] Petroski, Henry, *To Engineer Is Human,* Vintage Books of Random House, Inc., New York, 1992.

[2] Gibbs, W., "Software's Chronic Crisis," *Sci. Am. (Int.Ed.) 271*, 3 (sept.1994), 72-81.

[3] Beizer, Boris, *Software Testing Techniques,* International Thomson Computer Press, 1990.

[4] IEEE Std 1044-1993, IEEE Standard for Classification for Software Anomalies, *IEEE Standards Software Engineering Volume Four Resource and Technique Standards,* The Institute of Electrical and Electronics Engineers, Inc., 1999, isbn 0-7381-1562-2.

[5] Paulk, et al., "Capability Maturity Model, Version 1.1," *IEEE Software*, July 1993, pp. 18-27.

[6] Fagan, M.E., "Design and Code Inspections to Reduce Errors in Program Development," *IBM Systems Journal*, Volume 15, number 3, 1976, pp. 219-248.

[7] Glass, Robert L., "Inspections -- Some Surprising Findings," *Communications of the ACM*, April 1999-Volume 42, Number 4, pp. 17- 19.

[8] Cohen, D.M., S.R. Dalal, M.L. Friedman, and G.C. Patton, "The AETG System: an Approach to Testing Based on Combinatorial Design." IEEE Trans. S. E. vol. 23, no. 7, 437-443, July,1997.

[9] Wallace, Dolores R. and Laura M. Ippolito, "A Framework for the Development and Assurance of High Integrity Software," NIST SP 500-223, December, 1994, National Institute of Standards and Technology, Gaithersburg, MD 20899. http://hissa.nist.gov/publications/sp223/

[10] Wallace, Dolores R., Laura Ippolito, and Barbara Cuthill, “Reference Information for the Software Verification and Validation Process,” NIST SP 500-234, National Institute of Standards and Technology, Gaithersburg, MD 20899, April 1996. http://hissa.nist.gov/VV234/

[11] Porter, A., et al., "An Experiment to Assess the Cost-Benefit of Code Inspections in Large Scale Software Developments," *Proceedings of the Ninth Annual Software Engineering Workshop*, National Aeronautics and Space Administration Goddard Space Flight Center, Greenbelt, MD 20771, December 1994.