FAILURE MODES IN MEDICAL DEVICE SOFTWARE: AN ANALYSIS OF 15 YEARS OF RECALL DATA

DOLORES R. WALLACE[1] and D. RICHARD KUHN
Information Technology Laboratory
National Institute of Standards and Technology
Gaithersburg, MD 20899 USA
dwallace@nist.gov, kuhn@nist.gov

Most complex systems today contain software, and systems failures activated by software faults can provide lessons for software development practices and software quality assurance. This paper presents an analysis of software-related failures of medical devices that caused no death or injury but led to recalls by the manufacturers. The analysis categorizes the failures by their symptoms and faults, and discusses methods of preventing and detecting faults in each category. The nature of the faults provides lessons about the value of generally accepted quality practices for prevention and detection methods applied prior to system release. It also provides some insight into the need for formal requirements specification and for improved testing of complex hardware-software systems.

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 take license in extending Petroski's views from mechanical and structural engineering into the domain of software system failures. Many software assurance techniques, including inspections, failure modes and effects analysis, flaw hypothesis penetration testing, and some specification-based test methods, benefit from knowledge of the types of faults that typically occur in a given class of software. Lessons learned from failure analysis can either affirm proposed software engineering principles or help define new ones.

Several industries, including telecommunications, space, finance, and defense, were early drivers of computer technology. Within these industries, more and more systems are controlled by, or dependent on, software today than in the early years. We find a great need to examine software-based failures from many domains to gain 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 before systems are recalled from multiple users. Loss of revenue from the customer and additional costs for fixing a faulty system after release can become exorbitant.

[1] This author performed the work while an employee of NIST and is now working for Unisys at the NASA Goddard Space Flight Center.

Systems in all industries can fail for many reasons, including acts of nature, hardware failures, human error, vandalism and software. While the distribution of failures due to specific causes may differ by industry, most experience failures attributable to these causes at some time or another. Our long-term objective is to study failure data from several industries individually and then in the aggregate, to identify the relationships to software problems. We have previously examined failures of the public switched telephone network. [2].

We focus our current study on medical devices that have been voluntarily recalled by the manufacturers due to computer software problems. Any findings may well 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 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[2] 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 problems cited in this study were found in medical devices recalled by their manufacturers either in final testing, installation, or actual use from 1983 to 1997. It is important to note that there were no deaths or serious injuries caused by these failures, nor was there sufficient information to guess at potential consequences had the systems remained in service.

Using the Food and Drug Administration (FDA) database of medical device failures, we have examined the symptoms that indicated there were problems, identified the software faults that may have caused the problems, provided some generic guidance, and assessed what could have been done to prevent or detect the classes of faults. Section 2 contains a characterization of the system failure data, while Section 3 provides an analysis of the software faults. Section 4 contains a synopsis of the lessons learned with Section 5 providing conclusions about this study and recommendations for additional work.

[2] 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.

2 Characterization of the Data

A medical device may be as simple as a tongue depressor, but this paper is concerned only with those containing software. The study includes only those devices in the categories of anesthesiology, cardiology, diagnostics, radiology, general hospital use, and surgery. Examples of these devices are insulin pumps, cardiac monitors, ultrasound imaging systems, chemistry analyzers, pacemakers, electrosurgical devices, and anesthesia gas machines. The following highly simplified description is provided only to enable understanding of the classes selected for observed symptoms of malfunctions. A device is a system providing a service, involving one or more components. Some components may contain computer software, executing functions that produce an output either to the next function within a component or to another component of the system (e.g., a display device). The system behaves according to the values or messages it receives from the functions' output. An alarm may sound and / or the device may cease operation. A dosage rate or volume may change. Equipment may move. Measurements of various specimens or human reactions may be taken, and data may be recorded and associated with a patient's name. The failures have been observed as a response of the physical system and usually not as an obvious software fault.

2.1 General features of the recall data

The FDA recall data consists of 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 understand the types of software problems 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. This study 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. The total number of software recalls from 1983-1997 is 383. The years 1994, 1995, 1996 have 11%, 10%, and 9% of the software recalls. One possibility for this higher percentage in 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 [3].

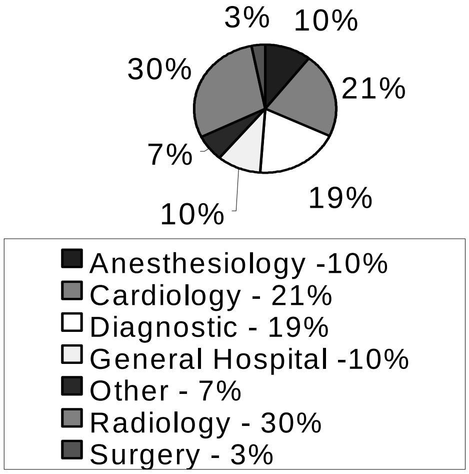

Fig. 1. Failure distribution by device panel

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 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.

2.2 Observed behavior signifying recall

The problem and cause descriptions contain information on which we base our analysis. They provide observations about the system or a feature as shown in the following examples:

- An alarm failed to sound.
- Dosages were too fast, too slow or were stopped inconsistent with the data on the display unit.
- Display unit values were inconsistent with other visual outputs of the device, for example, name of patient on screen not correct.
- The system simply stopped.
- The device performed in a manner completely unplanned, when several conditions

occurred simultaneously.

- Data were lost or corrupted.
- A calculation or other function was missing, or an instruction was omitted from the user manual.

For each recall, we reduced the problem description to a symptom of the failure (e.g., behavior–alarm did not sound; output – incorrect relationship with display). We next reduced the list to only the key attribute and one description, such as behavior alarm and ended with thirteen primary symptoms shown in Figure 2. The pie wedges match the legend going clockwise, starting with behavior, near the top, at 22%.

Definitions for the thirteen primary symptoms are the following:

- **Behavior:** the system performs an action due to some output of some function. The action is a physical action, e.g., movement of the gantry.
- **Data:** a consequence to the data, usually corruption or loss of input data.
- **Display:** the visual display on a screen –numbers, text, or images in various formats.
- **Function**: usually a single calculation or activity; a software function in one module.
- **General:** not enough information to assign to a category.
- **Input:** the initial input (typed, sampled, read off equipment, database, file or tape, etc.) on which some operation is performed.
- **Output**: result of some function; generally an output to be used by the next function.
- **Quality:** user observations stated that “quality requirement was not met".
- **Response:** something has happened that should not, e.g., power emitted above allowed amount; manifested in some hardware function.
- **Service:** an identifiable system service involving multiple functions such as pumping, ventilating, giving medication; generally involves more than one component (module, subsystem).
- **System:** the total system.
- **Timing**: timing of the instrument or a service of the device
- **User instruction**: manual, or other descriptions for the operator/ user.

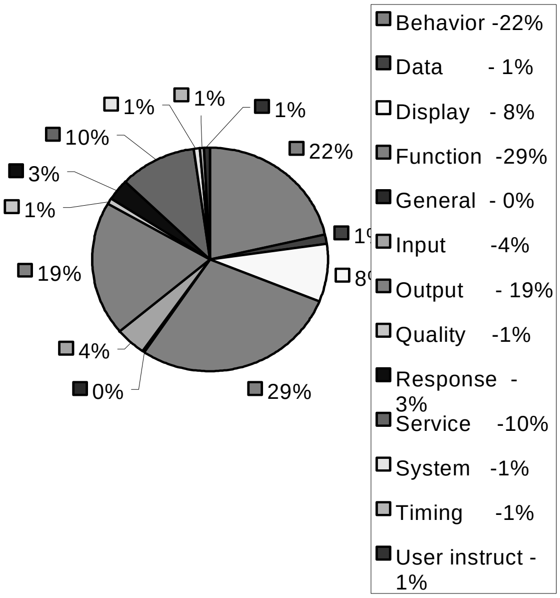

Fig. 2. Distribution of 383 failures by symptom

3 Analyses of the Data

While the observed symptoms provide some insight about the nature of the failures of the medical devices, the vendors' determination of the software fault is important information. In many cases, the vendors did not provide this information. We were limited in determining the fault by the problem and description data; there was no mechanism for getting any further details. We want to understand the nature and reason for occurrence of the software fault and to develop lessons regarding software quality practice. We selected the final fault class terminology from several published taxonomies and reasoned how the various problems best fit, based on the problem and description as provided in the FDA database. We had no access to the manufacturers or to any other data. From this limited information, we could discern the fault type for only 342 failures. Only these 342 failures are discussed in the rest of this study.

3.1 Fault distributions

In many cases there could have been 2 or 3 fault types contributing to a failure. Often study of the symptom revealed the generic nature of the fault. 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. Possibly, the developers had not specified in the requirements that these events could occur, or the logic of the design failed to account for these simultaneous events, or the code logic was incorrect. If the first situation had been true, then the problem would have been classified as a requirements problem (e.g., omission, ambiguity, conditions not considered). While recognizing the value of better specification methods, specifically formal methods in some of these situations, we classified most of these as logic problems at the point of failure. Without additional information we could not classify some of these problems as requirements. In Table 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 wrong rate in an algorithm.

Table 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
Calculation; fault tolerance	Not validated; QA
Change impact; QA	Reinitialization
COTS; memory lost; size	Requirement - wrong formula
Data passing; QA	Scaling
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

We reduced the number of fault categories to the final list in Figure 3.1, placing the detailed fault type into the class it best fit. 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 going clockwise, starting with calculation, near the top, at 24%.

Among the fault types, logic appears very high 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), configuration management, typos, mistakes in translating requirements into code, and quality assurance. For quality assurance, either the processes were not sufficient, or a new version was not validated.

For 1996-7, calculation faults occur 9 times in radiology compared with 13 faults in all the panels. For 1996, logic has 4 faults in cardiology and 3 in radiology out of 11. For 1997, logic has 5 in diagnostics, but only 1 in radiology. The other fault classes are smaller and vary over the years. For the other years, also, the higher percentages are generally for calculation and logic. The obvious questions are "Why are logic and calculation the prevalent types?" and "What can prevent or detect them before product release?”

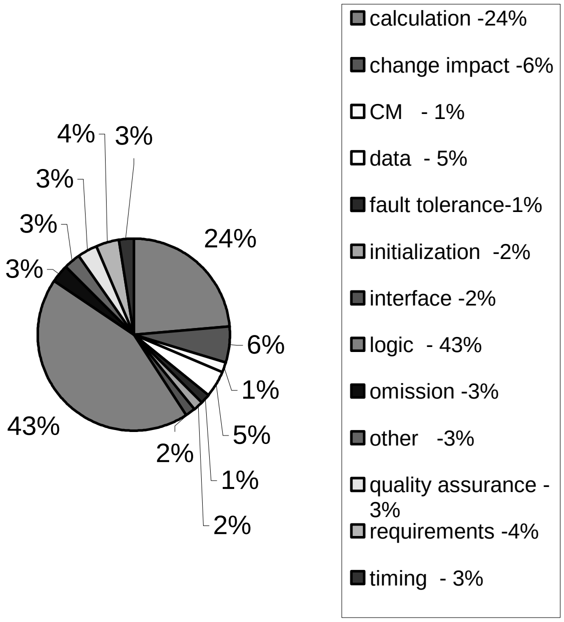

Fig. 3. Fault class distribution

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 and need to be utilized [4]. Thirteen fault classes contain 342 faults. First, by each class, for each fault, we considered various techniques/ methods for prevention, and then for detection. Next we reduced the results to a smaller, generic set for each fault class. While we provide descriptions of typical problems, only one problem per class is shown below with prevention or detection approaches. The complete tables are available at Error! Bookmark not defined..

Certain methods appear frequently in the complete synopses as well as in the few examples provided in this paper. We include inspection as both a prevention and detection technique, where inspection as prevention is used in a broader sense than the original Fagan inspection [5]. Glass explains this broader view which is based on practitioners' presentations in workshops and conferences [6]. In the prevention approach, then, inspection may include code reading and various static analyses. Sometimes we were specific, because the fault description warranted more specificity. When inspection appears as a detection technique, it generally means the traditional Fagan-type inspection.

While these faults occurred in medical devices, same faults occur in many other types of system. For example, truncation or rounding problems may occur in almost any software. The intent of providing details in the tables is to provide understanding of problems that may occur. Each system that may possibly have that type of error can benefit from the prevention and detection techniques.

The class *Calculation* 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). Examples for calculation appear in Table 2.

Table 2. Some examples for calculation

Generic Problem	**Prevention**	**Detection**
Constants or table of constants incorrectly coded.	Design, code reading to ensure correct relationship between the specified constant or table and the code.	Code reading, inspection. Unit test.
Improper handling of boundary conditions (e.g. circumstances close to limitation of the operating environment).	Assertions -- Fault tolerance.	Focused inspections, code reading or walkthrough. Unit test.
Improper handling of data structure (array, bitset, list, queue, set, stack, vector, etc.)	Low level design review. Code review.	Code reading or walkthrough, review, inspection. Unit test.
Precision problem (truncation or rounding error during I/O or calculation).	Low level design review. Code review.	Code review. Unit test.
Improper handling of abnormal conditions (e.g. wire disconnect from the device, electrical noise).	Assertion. Fault tolerance.	Unit test.
Graphical output meaningless.	Review requirements for relationship of computation output to next function.	Interface test.
Overflow.	Assertion. Fault tolerance.	Unit test.

While *change impact* is not necessarily considered a fault type, these cases indicate that failure to examine the impact of changes hides other problems. In all cases, another practice, performing a traceability analysis, is a prerequisite for performing change impact analysis. The analyses identify the region the proposed change will affect. Examples for change impact appear in Table 3.

Table 3. Some examples for change impact

Generic Problem	**Prevention**	**Detection**
Logic (incorrect conditions or incompatibility with sequential relationship)	Traceability analysis. Change impact analysis.	Inspection of logic relative all areas affected by change with focus on original assumptions (input values, selection criteria for a function)
No verification against original design specification	Traceability analysis. Change impact analysis	Inspection of proposed changes. Regression test
Loss of correct functions over several upgrades; Reversion to defect from at least two version back	Configuration management. Change impact analysis	Traceability analysis. Verification against original specifications. Interface test.

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. Others may have been prevented simply by using CM. The use of tools to manage the software versions would be helpful. In some cases, the problems stem not from improper software versions, but from selecting a software program that is not compatible 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. Some examples for for CM appear in Table 4.

Table 4 Example problems for CM

Generic fault	**Prevention**	**Detection**
Incorrect configuration for non-domestic systems	Use of CM tools.	Verify usage of CM tools for all changes. Verify configuration for non-domestic use.
Software incompatible with other components	Record assumptions about all components, in the CM data	Inspection of requirements for component interfaces. CM approval for configuring system components
Failure to upgrade accompanying system, to match software changes	Traceability analysis. CM tools.	Verification of changes; regression test.
Use of wrong master program when making software revision	Use of CM tools.	Verification of appropriate master program. CM manager releases the versions.

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. Of course, 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. Table 5 provides examples for data. :

Table 5. Some examples for data

Generic Problem	**Prevention**	**Detection**
System failed due to invalid input data	Assertions for invalid values, checks for ranges that imply incorrect data. Design: set criteria of input data validation. Code: implementation of input data validation.	Review for completeness of data specification, and that all data specifications are included in the user instructions. Inspection: focus on data validation. Test against invalid data.
Inconsistency of data retrieved from database and that expected by the program	Assertions on validity of data retrieved from database.	Testing focused on data retrieval .
Database corruption	Database administration.	Error handling routine in software.

The *fault tolerance* category relates to safety-critical systems that should include facilities to handle abnormal or anomalous conditions.   Fault tolerance examples appear in Table 6.

Table 6 Fault tolerance examples

Generic Problem	**Prevention**	**Detection**
Excessive use of the program causes failure	Fault tolerance such as warnings to operators.	Stress/ volume test. Testing against boundary and abnormal conditions.
Incorrect action due to external abnormal/ unexpected condition related to power supply or other components	Fault tolerance. Software cannot control abnormal condition external to it, but can provide a procedure in that event. Requirement needs to be written for FT.	Test against boundary, abnormal, and special conditions. Exception handling routine in software.
Incorrect action due to operator error	Fault tolerance in design through code to protect against human error.	Inspect, review for protection against operator error. Test against unacceptable data entry.

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. Examples for initialization appear in Table 7

Table 7. Examples for initialization

Generic Problem	Prevention	Detection
Lack of initialization of the runtime environment while the program initially executes or restarts	Use assertions for initialization. For C or C++, see http://hissa.ncsl.nist.gov/effProject/handbook/c++/variables.htm	Inspections; Code review. Test against initial conditions.
When the program executes first time, it fails to store necessary initialization values for the	Document initial conditions for both initial run and consecutive run. Design review.	Code review. Stress test (run the program multiple times).

succeeding run.		

In a system, *interfaces* allow software to send and receive data (that is, interface) to physical components of the system, as well to other software modules and to 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. Table 8 provides examples for interface.

Table 8. Examples for interface

Generic Problem	**Prevention**	**Detection**
Software does not properly interface with external device or other software component.	Trace requirements through design through code to ensure all software functions have interfaces to either another software module or to an output device or other system component or user. Examine the specification for each interface.	Inspections, reviews Integration test.

Logic problems appear to be significant. While some failures of the devices did result from bad logic, the "bad" logic might have resulted from incorrect, incomplete, or inconsistent requirements or designs. Frequently, interactions among different functions might not have been considered at all or might have been neglected at boundary conditions of a function. Sometimes the logic might have been incorrect in the design. All of these were classified as logic problems, but it should be understood that the source of the problem could have been requirements, design, or code. Two examples include 1) "When power 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." Table 9 provides examples for logic.

Table 9 Examples for logic

Incomplete or incorrect control logic	Design review. Walk through the software implementation against design.	Code Review. Inspection. Testing.
Configuration scheme for component interaction allows incorrect behavior.	Modeling. Simulation. Formal methods.	Code review. Interface analysis. Integration test. System test.
Improper handling of boundary conditions. (e.g. limits of value range)	Design review. Verify logic for all conditions, esp. at boundaries. Fault tolerance. Code review.	Code review. Inspection; Test against boundary and abnormal conditions.
Improperly handle abnormal or exceptional (e.g. power lost, multiple inoperative conditions occurred, I/O interrupt, I/O error)	Design review. Assertion. Fault tolerance. Review error recovery routines. Code review.	Code review. Inspection; Test against abnormal and exceptional conditions.
Improper data validation. (e.g. input or output data out of range)	Design review. Walk through the software implementation against design. Verify logic for data out of limits.	Code review. Inspection; Test against I/O boundary conditions.

Programming error (e.g. error in pointer, addressing, looping, indexing, subscript, memory management)	Low level design review. Code review.	Code review. Unit test.

The class *omission* indicates a required system function that is missing from the final implementation. Documentation provided is missing or not sufficient to install or operate the product. Two examples for omission are shown in Table 10.

Table 10. Examples of omission faults

Vital system functions are missing	Trace requirements through design through code, focus on all interfaces. Trace into user and test documentation. Use critical path analysis to ensure completion. Prepare system test scenarios at requirements specification and examine them for relationship to trace through code.	Inspections, reviews examining traceability of functions. System Test.
Lack of documentation, or improper documentation.	Proper release procedure. Traceability.	Verify completeness by examining trace. Inspection.

Other faults too low in frequency to be classified separately include problems such as performance issues, I/O problems, typographical errors. Other types of faults appear in Table 11.

Table 11. Other types of faults

Generic Problem	**Prevention**	**Detection**
Out of compliance with the performance standard.	Simulation. Design review. Code review.	Performance test.
Calculations associated with the "%" activity curve have been printed incorrectly. Formatting subroutine for screen display.	Code review: review special I/O routine. Understand the hardware/software requirements of the display system.	Unit testing with focus on verifying output against internal calculations.
A typographic error in software algorithm causes incompatibility between two devices.	Code reading against algorithm specifications.	Walkthrough focused on algorithms. Testing.

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 fault type, it is a process problem whose use might have prevented some of the failures. For this category, prevention techniques refer to discovering problems with QA. The responsibility for quality belongs to everyone on the project. QA examples appear in Table 12.

Table 12. Examples for QA

Generic Problem	**Prevention**	**Detection**
Test plan was not implemented or	Software project management	Project status review.

executed appropriately.	oversight.	QA process checks.
Regression test was not performed on modified software.	Software project management oversight. Change impact analysis.	Project status review. QA process checks.
No validation before initial release.	Specified procedures regarding testing before product release. Software project management oversight.	Project status review. QA process checks.
No validation on software changes.	Software project management.	Project status review. QA process checks.

Some faults, such as omission, logic, and calculation, may have their genesis in the *requirements* specification. 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. Examples for requirements appear in Table 13.

Table 13. Examples of requirements faults

Exceptional conditions were not specified in the requirement specification.	Modeling. Analysis. Traceability	Interface analysis. Requirement review. System test.
Functions missing in the requirement specification.	Modeling. Analysis. Traceability.	Interface analysis. Requirement review. System test.
Requirement specification was incorrect for its usage with other components.	Modeling. Interface Analysis. Traceability.	Requirement review. Interface analysis. Design review. System test.
Test hooks or monitors were not specified	Requirement review. Design review.	Integration, system test.

Timing, or synchronization, is vital to the execution of real-time applications. Examples for timing appear in Table 14.

Table 14. Examples for timing

Generic Problem	**Prevention**	**Detection**
Two inter-react processes are out of time synch with one another	Simulation. Design review. Code review.	Timing analysis. Integration test.
Real time clock was not accurate.	High quality real time operating system. Fault tolerance	Timing analysis. System test.
Scheduled event did not occur due to timer failure.	High quality real time operating system. Fault tolerance	Timing analysis. System test.

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. 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 that were observed in this study, that is, 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. Therefore 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 [4], [7], [8].

4.1 Development & Maintenance

While software development processes are already well defined by such models as the CMM, this study indicates particular practices which 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 which produced unusable charts. Attention to details, that is, 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 may 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, used regularly, can identify inconsistencies or incompleteness. The following list highlights some of the practices recommended for development and maintenance tasks:

- Complete specification of requirements, with emphasis on conditions and interactions of functions. Formal methods may be considered for highly complex systems.
- Traceability of the development artifacts: requirements to design (high, low levels) 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 as result of any assurance activities
- 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.

4.2 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 again, 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 [6], 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 [9]. To focus on a class of errors, the inspectors need to have some idea of the prevalent classes of errors of the product they are examining. 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 the symptoms that indicated there are faults
- Checklists, questions, methods designed to force those symptoms 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.

4.3 Testing

How thorough was the testing applied to the devices that were recalled? One way to study this question is to look at what conditions are required to trigger the faults that remained after release. That is, is the fault manifested in a single condition, or two or more conditions? Some of the failures (109 out of the complete set of 342) contained sufficient detail to determine what level of testing would be required to detect the fault.

For example, one problem report said 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 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."[3]

Only three of 109 failures 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 failures did not contain sufficient detail to make a judgment 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 generally have a relatively small number of input variables, each with either a small discrete set of possible settings, or a finite range of values. Nevertheless, testing all possible combinations of settings may not be practical. 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. But 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 such method makes it possible to cover all pairs of values for this example using only 180 test cases[9]. This level of test effort should be practical for most devices in the categories reviewed in this report.

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. 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 drive these symptoms to appear. In some cases, the systems were updated versions, so previous test histories may also have been helpful. The list summarizes these points:

[3] 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.

- Test cases aimed at manifesting prevalent symptoms observed by device operators
- Stress testing
- Change impact analysis and regression testing
- SCM release of versions only 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 symptom 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 classes and different characteristic failure symptoms. 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[4]. 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.

[4] A medical device manufacturer adding software to a device for the first time called one author during preparation of this paper.

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. Acknowledgments

The authors are grateful to the Food and Drug Administration (FDA) for making this data available to us. Our analyses and conclusions do not reflect any analyses or conclusions by the FDA. We appreciate the reviews and suggestions of Dr. Larry Reeker and the efforts of Mark Zimmerman and Michael Koo for their technical support.

7. References

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

2. D. R. Kuhn, "Sources of Failure in the Public Switched Telephone Network," *Computer* Vol. 30, No. 4 (April, 1997).

3. W. Gibbs, “Software’s Chronic Crisis,” *Sci. Am. (Int.Ed.) 271,* 3 (sept.1994), 72-81.

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

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

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

7. D. R. Wallace, and L. 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/

8. D.R. Wallace, L. Ippolito, and B. 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/

9. A. Porter, 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.