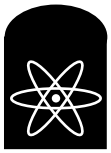
UCRL-ID-119153

Design Factors for Safety-Critical Software

J. Dennis Lawrence

G. Gary Preckshot

October 4, 1994



FESSP

Fission Energy and Systems Safety Program

Lawrence Livermore National Laboratory

This work was supported by the United States Nuclear Regulatory Commission under a Memorandum of Understanding with the United States Department of Energy, and performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract W-7405-Eng-48.

Design Factors for

Safety-Critical Software

Manuscript date: October 4, 1994

Prepared by
J. Dennis Lawrence
G. Gary Preckshot

Lawrence Livermore National Laboratory
7000 East Avenue
Livermore, CA 94550

Prepared for
U.S. Nuclear Regulatory Commission

ABSTRACT

This report is the fourth of a series of reports prepared for the Nuclear Regulatory Commission Office of Nuclear Reactor Regulation, and provides the summary and conclusion for this task.

It is widely believed in the software engineering community that almost anything can affect the ability of software to reliably perform its tasks, particularly when safety is at issue. While this statement is true, both in the abstract and in specific instances, it is not particularly helpful. It remains necessary for auditors and other reviewers to assure themselves and the public that safety-critical software has sufficiently low probability of failing in such a way as to cause death or injury to permit it to be used in safety-critical applications.

Achieving this assurance is best done by using a well-planned, methodical approach. A possible approach is to concentrate on those attributes of the software and the development process (design factors) that are most influential in achieving dependable software.

Seventy-four design factors are identified in this report, divided into nine categories. Seven categories relate to the development process, and one category relates to the products of that process. The remaining category contains negative factors whose presence should be regarded as cause for intense scrutiny of the development process.

Seven of the design factors should be considered mandatory for any organization responsible for developing safety-critical software. An additional nine factors are considered essential to safety, but not as important as the first seven. The remaining design factors can provide additional important indications of the quality of the development effort and the software resulting from that effort.

CONTENTS

ACKNOWLEDGMENT

The authors thank and acknowledge the efforts of Nuclear Regulatory Commission staff members, Mr. John Gallagher, Mr. Joe Joyce, and Mr. Michael E. Waterman, who reviewed this work and provided their insights and comments.

DESIGN FACTORS FOR SAFETY-CRITICAL SOFTWARE

1. INTRODUCTION

1.1. Purpose

The word "dependability" can be defined to be a measure of a system's "ability to commence and complete a mission without failure" (Lawrence 1993). This broad concept incorporates various characteristics of the software, including reliability, safety, availability, maintainability, and others.

The term "design factor" is used in this report to refer to any characteristic of the software, or of the process used to develop the software, which has the potential to affect its dependability.

It is widely believed in the software engineering community that almost anything can affect the ability of software to reliably perform its tasks, particularly when safety is at issue. While this statement is true, both in the abstract and in specific instances, it is not particularly helpful. It remains necessary for auditors and other reviewers to assure themselves and the public that safety-critical software has sufficiently low probability of failing in such a way as to cause death or injury to permit it to be used in safety-critical applications.

Achieving this assurance is best done by using a well-planned, methodical approach. A possible approach is to concentrate on those attributes of the software and the development process (design factors) that are most influential in achieving dependable software.

Seventy-four design factors are identified in this report, divided into nine categories. Seven categories relate to the development process, and one category relates to the products of that process. The remaining category contains negative factors whose presence should be regarded as cause for intense scrutiny of the development process.

1.2. Scope

This report is the fourth of a series of reports prepared for the Nuclear Regulatory Commission Office of Nuclear Reactor Regulation (NRC/NRR), and provides the summary and conclusion for this task. The reader is assumed to be familiar with the contents of the first three reports:

- J. Dennis Lawrence, *Workshop on Developing Safe Software: Final Report*, UCRL-ID-113438, Lawrence Livermore National Laboratory (November 1992).
- J. Dennis Lawrence, *Software Reliability and Safety in Nuclear Reactor Protection Systems*, NUREG/CR-6101, UCRL-ID-114839, Lawrence Livermore National Laboratory (November 1993).
- J. Dennis Lawrence and Warren L. Persons, *Survey of Industry Methods for Producing Highly Reliable Software*, UCRL-ID-117524, Lawrence Livermore National Laboratory (June 1994).

1.3. Sources of Information

The information presented in this report is a synthesis of information obtained from numerous sources. The following are the primary sources.

- A Workshop on Developing Safe Software was held July 22-23, 1992, in San Diego, California. The purpose was to have four internationally known software experts discuss among themselves software safety issues which are of interest to the NRC. The results of the workshop are documented in Lawrence 1992.
- Three companies with excellent reputations for developing high-quality software were visited during 1993 with the intent of discovering those attitudes and practices which each company deemed important to their success in constructing software of consistently high quality. An analysis of the results of these visits is documented in Lawrence 1994.
- Numerous standards, from IEEE and other organizations, exist which mandate or recommend development practices that are believed by the standards development bodies to improve software development. The following standards were used in preparing the report:

  Software for Computers in the Safety Systems of Nuclear Power Stations, IEC Publication 880 (1986).

  IEEE Standard Criteria for Digital Computers in Safety Systems of Nuclear Power Generating Stations, IEEE 7-4.3.2 (1993).

  IEEE Standard Glossary of Software Engineering Terminology, IEEE 610.12 (1990).

  IEEE Standard for Software Verification and Validation Plans, ANSI/IEEE 1012 (1986).

  IEEE Standard for Developing Life Cycle Processes, IEEE 1074 (1991).

IEEE Standard for Software Safety Plans, IEEE 1228 (1994).

- Research performed by LLNL for the NRC. This research is documented in the following references (in addition to those cited in Section 1.2): Preckshot 1993a, Preckshot 1993b, Persons 1994, and Scott 1994.
- A report, *Factors in Software Quality*, written for Rome Air Development Center in 1977 (McCall et al. 1977).
- The general literature on software engineering, as published in books, journal articles and technical reports.

Much of the information in this report also appears in Ploof and Preckshot 1993, Appendix A.

2. SUMMARY

No single design factor, and no combination of design factors, is sufficient to guarantee safety. Experience does show that there is a combination of design factors which, if properly used, can result in an adequate level of software dependability.

Some design factors are considered necessary to any project where safety must be assured, and others can be quite helpful. The most important design factors are discussed below. All 74 design factors are listed, with brief comments, in Section 3.

Seven design factors should be considered mandatory for any organization responsible for developing safety-critical software. These are listed below. The lack of any one of these factors should be considered as sufficient grounds to reject the software.

- **Personnel quality and experience.** Although this will be a controversial subject, personnel quality continues to be the single most important factor in the designing and coding of a software product. The controversy is not whether intellectual capability is a good thing, but rather who should determine it and how it should be determined. A distinction should be made between managerial ability and technical ability. Both are important. In particular, a lack of managerial ability can thwart considerable technical ability.
- **Use of configuration management.** Configuration management is crucial and is absolutely necessary to have confidence that the correct product is built, and that change occurs in an orderly way. Configuration errors are among the simplest and also the most prevalent made in the software industry. Adequate configuration management can be demonstrated by review of past and current company practices. Three configuration management functions of particular note should always be present: change control, interface documentation and control, and delivered product configuration control.
- **Clear, stable and validated software requirements.** The software development process should produce clear, stable, and validated software requirements. Company practices, plans, and example requirements from prior safety-related products provide evidence that clear, stable, and validated requirements are the norm. Positive findings include documented requirements analyses, requirements stability control using configuration management, and the use of prototyping or simulation to understand the implications of requirements more fully.
- **Independent verification and validation.**[1] Verification and validation (V&V) activities should independently confirm requirements and development attributes, as well as individual product quality. V&V leaves a significant trail of documentation, which can be inspected for prior safety-related projects. Significant positive findings include multi-level testing (e.g., unit, subsystem, and system), and products that are designed to facilitate V&V.
- **Use of a formal life cycle for product development.** The organization should have a very clear picture of, and a formal model for, the life cycle of its products. This should be clearly reflected as the temporal glue that binds all development and certification activities into an orderly sequence. The use of a life cycle model will be clear from review of development plans.
- **Traceability from system requirements and design, through software requirements, software design, code, and validation testing.** The software development process should produce clear, unambiguous, documented designs that are traceable item-by-item to requirements. There should be a systematic process for producing software products that are traceable item-by-item to designs. This is demonstrated by documented software development process models, which describe the systematic procedures and attributes of the software development process. The models may be validated by process measurement.
- **Use of hazards analysis and risk analysis to guide software development.** For ultra-reliable safety software, the requirements, development techniques, V&V rigor, and product factors should be guided by the results of hazards and risk analyses. The consistent use of reliability practices

[1] In this report, independent V&V includes independent testing.

can be seen in the documentation of analyses used for planning development, V&V, and designs for prior safety-related products.

An additional nine design factors are considered essential to safety. The lack of any one of the factors is a cause for concern, and should result in extensive assessment of the software. The lack of more than one should be considered as grounds for rejection of the software. These factors are listed next, also in no particular order.

- **Commitment to quality.** A commitment to quality is the most important of the essential factors. The organization's reward structure should match the quality commitment claim and should have a relatively long history demonstrated by documentation.
- **Clearly defined and stated management policy and a well-managed and complete documentation activity.** From a regulatory viewpoint, a clearly defined and stated management policy and a well-managed and complete documentation activity are the only ways the NRC can obtain reliable visibility into other design factors. The organizational record should show a number of years of successful practice under stable policy.
- **Independence of configuration management and quality assurance.** Configuration management and product assurance should at least be independent of the managers and programmers responsible for developing the product. Independence can be demonstrated by review of a company's management structure and reward system.
- **Continuous process improvement.** Management should have a clear picture of the development process and should be prosecuting continuous improvement efforts. This should be demonstrated by a reasonably long (several years) documented history of this activity. Other checkpoints are development and use of documented internal or external (e.g., national) standards, and use of process models.
- **Measurement of the results of the software development process.** Management should be measuring the results of the software process and management's own performance. Without a measurement record, claims of process improvement cannot be substantiated. The database of measurement results is evidence of process and product measurement.
- **Sufficient available resources and training, appropriate to the difficulty of the development tasks.** The availability of resources and training assigned by management should be appropriate for the difficulty of the software tasks. Resource allocation is documented by management plans and histories of plan execution. Training is documented by personnel assignment records.
- **A history of on-time, on-budget, within-specification product deliveries.** Management's ability to assess development risk and history of on-time, on-budget, within-specification deliveries is a significant indicator of probable quality. Managerial performance is documented by plans and results for previous safety-related projects similar in scope and nature to current work or future work under the purview of the NRC.
- **Early problem detection and resolution.** The practice of early problem detection and resolution is a positive indicator of eventual product quality and can be demonstrated by documenting detected software errors systematically.
- **Defect tracking.** Defect tracking, root-cause determination, and correction of both the product and the process are positive indicators of process improvement, if the defect-tracking activity is done with due regard for statistical validity. Documentation of this activity provides a record of organizational performance.

The remaining design factors, included in the full list of design factors presented in Chapter 3, can provide additional important indications of the quality of the development effort and the software resulting from that effort. Particular attention should be paid to the negative factors (Section 3.7) and the product factors (Section 3.8).

3. DETAILED LIST OF DESIGN FACTORS

Design factors are organized under nine headings and are described in one-paragraph appraisals below. The rationale for each heading is described in a single paragraph following the heading. Each design factor description gives the justification for the design factor and notes restrictions where appropriate. The design factors listed are taken from the sources listed in Section 1.4 and are not the work of the authors. The arrangement, headings, and descriptions represent the opinions or work of the authors.

3.1. General Design Factors

General design factors apply to all members of an organization in all phases of software development.

3.1.1. All levels of the organization are committed to quality.

Since software quality, like a chain, is only as strong as the weakest link, all members of a software development organization must be committed to making quality happen. Management commitment is particularly important because management controls the resources allocated to quality assurance activities.

3.1.2. There is longevity in personnel, policy, and process.

The process of building a quality software development organization takes time, by some accounts two years for each incremental improvement in SEI maturity level.[2] Therefore, personnel, policies, and the development process must exist and be under improvement for a relatively long period of time. Data on product performance and software process must also be collected for significant periods of time to be statistically valid.

3.1.3. Configuration management is used extensively.

Configuration management was cited by all respondents as being crucial to any scheme of product or process control, irrespective of any other software method or process model. Without effective configuration management, it is impossible to determine what has been delivered, how it was produced, who made it, and whether it met requirements.

3.1.4. Testing, V&V, and software quality assurance (SQA) are independent.

Independence of testing, validation and verification, and quality assurance activities from development activities that are under schedule and budget pressure is necessary to prevent compromise of quality for expediency.

3.1.5. An appropriate life cycle model is used.

The discipline of using a life cycle model is more important than the actual details of the model selected. A life cycle model allows the various related activities of software development and quality assurance to be coordinated in a rational progression.

3.1.6. There is continuous process improvement.

No software development process is perfect, and a good process will degrade without continuous attention. Improvement is a general watchword regardless of actual process details.

3.1.7. Reviews, walkthroughs, and inspections are used.

Reviews, walkthroughs, and inspections are recommended at all stages of development and for all products, including V&V and quality assurance products. Code inspections and walkthroughs are credited by one respondent with finding 85% of errors prior even to testing.

3.1.8. Automation is used where appropriate.

Automation is suggested for all activities that are tedious, repetitive, error-prone, and sufficiently well-defined that automated software tools can be written to accomplish them. This allows human effort to be redirected to areas where the human intellect is superior to machine performance. Automation may also permit enforcement of standards and customs.

3.1.9. Vendors, products, and services are certified.

Products and services used in the development process should be certified to the level required to support the product(s) being developed.

3.1.10. Software is the company's primary business.

The company, or division responsible for software, should be in the software development business directly, not as a peripheral activity to the company's real business. This ensures that software concerns and software expertise are sufficiently high in the company's business plans that they receive adequate attention and resources.

3.1.11. The organization adapts to changing environments.

The computer industry, and particularly the software industry, has undergone rapid change during its entire existence. Software development organizations and their software development processes must continue to adapt to this changing environment, both because old methods used in new situations may be inappropriate, and because the tools and equipment available may force the change.

3.1.12. The organizational goal is defect-free software.

Even though achieving this goal may be impossible, no safety-critical software developer should aim to have defects. The defect-free goal and the resources devoted to it are evidence of commitment to quality.

3.1.13. Quality must be built in; testing cannot find all defects.

It is not possible to "test in quality." Quality must be designed into the product and that fact should be

[2] From the Carnegie Mellon University Software Engineering Institute (SEI). See Paulk 1993.

demonstrated by testing, V&V, and quality assurance activities. Quality, in this definition, means adherence to requirements.

3.2. Process Control Design Factors

The factors described below apply specifically to controlling or measuring the software development process.

3.2.1. Processes are defined.

The software development process should be defined in detail so that practitioners can judge whether or not they are accomplishing development according to the process model.

3.2.2. Process is stabilized by measurement and feedback.

Performance of actual development activities is measured and compared with the defined process model. If discrepancies exist, either development activities are redirected or the model is changed until a stable, well-understood development process is achieved.

3.2.3. The number of process variants is reduced by standardization.

An organization should settle on one or a few process models to guide its development activities, depending upon purpose and business. For instance, a spiral life cycle model with repeated prototypes might be used for products whose requirements are inexactly known but whose failure consequences are low. A waterfall life cycle model might be used for products whose requirements are well known, but whose performance requirements are strict and whose failure consequences are severe. Limiting the number of process variants makes sense because scarce resources can be applied more effectively to process improvement.

3.2.4. Processes are improved only after they are stabilized.

Hitting a moving target is always more difficult than hitting a stationary target. Development processes should be stabilized and measurements made so that the effects of changes can be determined and adjustments made in additional change efforts. Two developer organizations suggest that changes to process should be made one at a time, allowing time for stabilization and measurement before making additional changes. This is one contributor to the longevity factor (See Section 3.1.2).

3.2.5. Data collection and use of data is balanced.

The amount of data collected should be appropriate to the use of it. Collecting data that will not be used (including usage later in historical databases) wastes effort. Conversely, making decisions with inadequate data is just guessing. Data should be collected for historical databases as part of building a long-term process or product history, but not to the extent that it overwhelms short-term data usage activity.

3.3. Management Design Factors

These are factors that are primarily the responsibility of management to implement or enforce.

3.3.1. The reward structure matches the quality commitment.

If management gives lip service to quality, but rewards for other performance, other performance is what will be achieved.

3.3.2. Management uses process models.

While all persons involved in the software development effort benefit from understanding the process models in use, management's ability to control development processes by allocation of resources and effort is greatly enhanced by using process models.

3.3.3. There is constant process measurement and improvement effort.

Software development is a perishable process. Quality can only be maintained by constant measurement and improvement effort. Management's responsibility is to see that this effort is expended, even though it does not contribute immediately to a product.

3.3.4. Management makes predictions using models.

Control of process is only achieved by predicting what effect proposed actions will have, and modifying actions to have the desired effect. Management should use model predictions as a first cut at determining what the effect of management actions will be. The use of models predictions and subsequent measurement is essentially feedback control, with the model predictions providing a feed-forward component. In the language of control systems, the measurement lag would make an extremely sluggish system or an unstable one without the anticipation provided by predictions.

3.3.5. Management achieves predicted cost, schedule, and quality goals more often than not.

It is important that management have a track record of planning, allocating resources, and meeting schedules within cost and quality constraints because the first thing to go under schedule and cost pressures is usually quality.

3.3.6. Management controls risks by adopting appropriate strategies.

In the commercial software development world, risk is perceived as failing to deliver a minimally acceptable product on time and more or less within budget. Management uses such strategies as "descoping" (delivering less), delivering with bugs, or renegotiating schedules, depending upon contractual provisions (if any) and commercial conditions. Delivering buggy software is a strategy often used by commercial software vendors trying to hit a market window, although it is greatly disliked by customers.

3.3.7. Management abandons methods that do not work.

This may seem like an obvious thing to do, but abandoning a work practice is often a serious career risk for a manager because it is viewed as an admission of error. Mature management expects some percentage of methods attempted to perform relatively poorly, and plans to acquire data about method performance with a view to discontinuing those that do not work well (See item 3.7.9).

3.3.8. Management ensures planning, production, and control of documentation.

Accurate and complete documentation is necessary for product maintenance as well as data collection about organizational performance. Documentation is one of the first things to be neglected under stress, and serves as a sensitive indicator of management performance. Documentation also serves as a record of organizational longevity and history, validating claims of sufficient experience to be considered at one of the higher SEI maturity levels.

3.3.9. Management invites external review.

Nobody is objective about himself.

3.3.10. Improvement takes time — an average of two years.

Respondents were unanimous in noting that no quality software development organization can be put together overnight. Empirically, it appears that about two years are required for the average software organization to move up one rank in the SEI maturity scale.

3.4. Personnel Design Factors

These are factors that characterize the personnel involved in the software development process.

3.4.1. Programming skill is not enough; some personnel must be skilled in the problem domain.

When software is applied to problems whose solution has a significant non-software component, as would occur in reactor protection systems, aerospace control systems, or the like, programmers who have no knowledge in the application field are prone to make mistakes of ignorance. In the highly specialized space shuttle program, for example, there is close cooperation between engineers and scientists who are cognizant of astronautics and shuttle systems, and programmers.

3.4.2. High intellectual ability of staff is crucial to success.

Most writers in the software development field note that the single greatest factor in ensuring quality software is staff quality. A distinction should be made between managerial ability and technical ability. Often, an individual may be adept in only one area.

3.4.3. Inaccurate interpersonal communications are an obstacle to producing high-reliability software.

The large organizations in the LLNL survey noted that as software teams get larger, efficient inter-team-member communications become more important, and sometimes become a bottleneck. The corollary of this point is that small teams are preferred.

3.4.4. Personnel in influential positions should be highly skilled in all aspects of the development of high-reliability software.

Developing high-reliability software is a special skill that is learned, not inherent. Persons with average schooling or experience cannot be expected to do this and should not be in positions where their inexperience can affect the development process.

3.5. Development Design Factors

These are factors specifically related to the software development process.

3.5.1. Configuration management, V&V, and SQA are coordinated with development activities.

This reflects the fact that in an orderly software development process, certain products subject to V&V and SQA are available at process milestones. V&V and SQA products are produced from development products and lose their effectiveness if not fed back into the development process in a timely way.

3.5.2. Requirements are stable.

One of the major markers of failed software development efforts is unclear and constantly changing requirements. Stable requirements are crucial to success.

3.5.3. A requirements analysis is performed.

Having stable requirements is merely the first step. Requirements must be analyzed to understand their implications. Analysis often reveals inconsistencies, unneeded but expensive specifications, or requirements that may be extremely difficult or impossible to fulfill. Analysis is also necessary when converting requirements provided by non-software specialists to requirements suitable for software.

3.5.4. A requirements validation is performed if possible.

Requirements validation is the process of returning to the original statement of requirements and examining the detailed, analyzed list in light of the original. In some cases, such as reachability analysis of communication protocols, requirements validation can be automated.

3.5.5. Much of the development effort concerns getting the requirements right.

This is true, at least, of experienced software developers who have discovered that it is easier in the long run to do something once right, than several times wrong. Long and detailed scrutiny of requirements is a marker of successful developers.

3.5.6. Prototyping or simulation is an important tool.

Prototypes or simulations are useful in three ways in software development (Preckshot 1993a). First, they are often used to demonstrate proposed designs to prospective users as an iterative method of refining requirements. Second, they may be used to test an approach to solving a problem in which there are uncertainties, including hardware performance uncertainties. Third, they may be used in the traditional sense of an engineering prototype, to demonstrate or validate performance of a scalable portion of the final system.

3.5.7. Critical components are identified early.

This point was emphasized by several respondents and reflects the view that management must identify where to apply the most valuable resources (personnel) early, so that they have time to solve the problems. This is a form of development risk management.

3.5.8. Development activities promote early detection of errors.

It is a widely held view in the software industry that it is less costly to fix errors early in the development process. It is true in general that bug fixes of late errors are often limited by earlier design decisions and the small remaining time (schedule) and funds (budget). Early error detection is another form of development risk management.

3.5.9. Defect tracking is done uniformly and consistently.

One of the indications of how well a software development organization is doing is the number of software errors being committed. Error or defect tracking is not very easy, however, and represents considerable effort to do in a statistically valid fashion. Invalid methods of error accounting affect both estimates of software reliability and corrective efforts applied to the software development process.

3.5.10. Root causes of defects are determined and corrective actions are taken.

Once errors are identified and tracked, the reasons they occurred should be determined and then two corrective actions should be taken: the product should be fixed and the development process should be modified, if appropriate, to reduce the probability of similar errors in the future. This factor is typical of developers that maintain close control of errors and error causes.

3.5.11. Testing is done in several levels, viz. unit, subsystem, system.

It has been found that testing at different levels is necessary because of two countervailing effects. As level becomes more complex (toward system) low-level errors may be exercised rarely, thus reducing the probability of finding them. On the other hand, complex interaction errors may only exist when the entire software system is assembled. Also, inasmuch as early error detection can only be done on software that is ready early, perforce unit and subsystem testing must be done because the system is not yet available. Consequently, testing at multiple levels of system assembly is done by high-reliability software suppliers.

3.5.12. V&V is planned early in the life cycle and results are peer-reviewed.

Validation and verification as an afterthought is a marker of an inexperienced or sloppy developer. V&V should be planned early so that sufficient resources can be allocated to accomplish it and so that there is sufficient time to do it correctly. Peer review ensures that V&V results are not reviewed exclusively by those who planned the tests, analyses, and inspections. This helps avoid “expectation blindness,” in which the planners may see only the results they expected to get and ignore signs of possible trouble.

3.5.13. The product is designed to validatable and verifiable.

Much as an electronic device can have test points built in for test and calibration, software can be designed

with a view to making V&V easier. This also means that design documentation and coding style should permit review and easy understanding by others not directly involved in development.

3.5.14. A design philosophy suitable for safety-critical software is used.

In general, this means avoiding the use of "risky" practices. Depending upon the challenges the developer must face (e.g., aerospace vehicle flight control is more difficult than reactor protection systems), the most dependable and least complicated way of solving the software problem should be selected.

3.5.15. There is extensive reuse of "middleware."

"Middleware" is defined by the respondent that proposed this factor as middle-level subroutines that are general enough to be used by several applications. Such routines are also known as "trusted" routines, and reuse implies that a software developer maintains a library of trusted subroutines that are well documented, extensively tested, and understood by the programming staff. The advantage of reuse is that scarce intellectual resources are freed for application to problems specific to the software job at hand. The disadvantage is that trusted routines may be misapplied because they "almost" fit the function needed. Estimates in the literature suggest that reuse can save from 20% to 50% of the effort involved in creating software modules from scratch, but claims of greater savings should be viewed with caution unless the new application is very similar to previous applications.

3.5.16. Software layers are identified and managed appropriately according to risk.

This factor recommends an hierarchical structure for software and makes a statement about risk that is ambiguous. From a software developer's viewpoint, the riskiest software layers are those upon which the whole product depends, so these must be done before any version of the product can be delivered. From a regulator's viewpoint, the riskiest software is that software that is essential for safety, followed by that software that is important for safety, followed by all other software. The respondent probably meant the first definition of risk.

3.5.17. An appropriate level of complexity is defined for the product, and practices are followed that control it.

The minimum level of complexity that the product must have is set by the functional complexity of the requirements the product must meet. Many software products have more than the minimum complexity because of implementation practices or because the designers choose an approach that is unnecessarily complex. Complexity control must occur over the entire life cycle of the product because unneeded complexity can creep in during requirements analysis, design, implementation, or maintenance.

3.5.18. Project teams are small (6–8 members).

This factor addresses the difficulty of communication in large project teams. It can be done, but it is difficult to maintain currency and direction in large project teams, and management's role in defining and maintaining critical interfaces and project team communications becomes more important as the team gets larger. Small teams avoid many of the pitfalls.

3.5.19. Software interfaces are documented and controlled.

Software interfaces (subroutine calling conventions, system call conventions, interrupt handling, network protocols, distributed system interactions) are always important and form a significant part of design documentation. With large project teams or multiple software contractors, they are even more important. Lack of interface documentation and control is almost always a sign of trouble.

3.5.20. Automated tools are used to enforce standards.

Automated tools, if easy to use, are a way of getting everyone on a team to do things the same way (the tool way), and thus provide a relatively low-conflict way of enforcing standards. From a regulator's viewpoint, the use of automated tools improves consistency and performance (the job is more likely to be done), both of which are positive factors.

3.6. Reliability and Safety Factors Design Factors

These are factors directly related to producing safety-critical software.

3.6.1. Hazards analyses must be part of the development process for safety-critical products.

Hazards analysis shows pathways a system can follow to get into hazardous conditions, and is recommended by several experts to ensure that software takes these pathways into account. Hazards can also be introduced by the selection of design approaches, certain hardware, software tools, or the use of software itself as a solution to a safety problem. The same expert recommends additional hazards analyses at points during the development life cycle to ensure that existing hazards continue to be covered and that new hazards are not introduced.

3.6.2. Diversity used to improve reliability is carried out at the system level.

Software diversity (e.g., N-version programming) has not yet been demonstrated to be adequate to counter common-mode failure due to programming error. For this reason, diversity at the system level (i.e., diverse, non-software methods) should be considered for improving total system reliability. Safety is a system issue, not solely a software issue. In safety systems containing software, software is only one of several components that must function correctly to perform the safety functions. Like diversity for reliability, non-software elements should be used to improve total system safety.

3.6.3. The developer understands that accidents are often caused by non-technological factors.

The safety system of which software is a part may be circumvented, turned off, or driven into failure by operator actions. Neither the system or the software should be expected to prevent these problems.

3.6.4. Ultra-high reliability is not claimed for software systems.

No known method of testing can be or has been applied for sufficient time or number of demands to demonstrate ultra-high reliability (10^{-7} to 10^{-9} failures per demand). No method of logical analysis has yet been accepted by safety system experts as sufficient to prove that ultra-high reliability has been achieved. Therefore, claims of ultra-high reliability should be viewed with caution. System designs that depend upon ultra-high reliability of components should receive the highest level of scrutiny.

3.6.5. Testing is not claimed to demonstrate reliability beyond 10^{-4} to 10^{-5} failures per demand.

This is based upon approximately two years of testing time on an unchanged product without error. Since software products, including compilers, linkers, and other software tools, often have a new version cycle of approximately two years, two years may be the practical maximum testing time available in the current commercial environment.

3.6.6. Complexity measures are understood to be of very limited utility in estimating software reliability or remaining software errors.

The most effective use of complexity metrics is as a guide for allocating resources during development (Preckshot 1993a). No complexity metrics have been validated against reliability measures or software errors. In the opinion of software safety experts, complexity metrics are “snake oil” (Lawrence 1992).

3.6.7. If reliability better than 10^{-3} failures per demand is required, adequate resources are made available.

Estimates of the development cost of space shuttle flight software are that it cost five to ten times what comparable ground support software cost to develop. Additional quality control measures are expensive, and justified where consequences of failure are great.

3.7. Negative Factors Design Factors

These are factors whose presence should be cause for caution or more thorough scrutiny.

3.7.1. There is high turnover.

The most obvious implication of high turnover is that building a team of high-quality people with a team memory is impossible. Less obvious is the fact that high turnover is a comment by programmers and managers who leave on the competence of management that is left behind. It should not be ignored.

3.7.2. Projects are schedule-driven, rather than quality-driven.

The first victims of a missed deadline are usually quality assurance and documentation. The next victim is the testing program. A “deliver at all costs” mentality is cause for caution.

3.7.3. Organizational process history is short or lacking.

Most respondents were explicit about the length of time it takes to build a quality software operation. SEI generally regards maturity level changes as requiring significant time (at least upwards). ISO requires several years to achieve certification.

3.7.4. Management cannot enforce stable requirements.

Stable and complete requirements are necessary for quality software products, but the role of management in ensuring this cannot be emphasized enough. Not only must management demand that requirements be locked down, but management itself must not be the source of requirements thrashing. Requirements instability and weak management control are indicators of potential failures.

3.7.5. Management’s estimates of product reliability greatly exceed what is actually measurable or provable.

Unrealistic claims of product reliability may be an indication that management does not understand the limits of the current state of the art in software development. Such claims should be investigated and

management should be given an opportunity to prove its claims.

3.7.6. Management has a record of failing to meet predicted cost, schedule, and quality goals for products.

This is typically an indication of management by chaos or paradoxically, schedule-driven rather than quality-driven development. Schedule- and budget-driven development schemes often fail to meet delivery schedules because of product non-performance problems. Something is delivered, but it is not the contracted-for product. A record of failing to meet cost, schedule, and quality goals should be taken seriously as an indicator of deeper troubles.

3.7.7. The organization fails to track errors and causes.

An organization's record of errors, causes, and corrective actions is its win–loss track record. No record, or a haphazard record, should be taken in default as meaning a bad record.

3.7.8. The development effort is underfunded.

Several of the developers interviewed suggested that most large government software contracts are underfunded by at least a factor of two, with the expectation that more funds can be obtained later by litigation, contract expansion, or cost overrun procedures. Whatever the reason, underfunding results in staff transients and failures to carry out "non-essential" activities such as quality assurance, documentation, and V&V. While it may be difficult for an outside reviewer to estimate what a correct funding profile should be, this negative factor is a very real one.

3.7.9. The organization exhibits "kill the messenger" syndrome.

Several of the developers had administrative procedures by which bearers of bad tidings could unburden themselves without jeopardizing their careers. They noted that organizations without these mechanisms were often the last to know about internal problems.

3.8. Product Design Factors

Product factors characterize the software product itself. The factors listed below represent product characteristics that are considered to be low-risk implementation methods for ultra-reliable software. The presence of these factors is considered a positive indication of lowered complexity or easier error detection.

Product factors are not usually found in standards or process models, because standards setters and process model makers consider that such factors restrict the generality of the standard or process model. Nonetheless, where safety-critical software is concerned, these design factors are useful as product quality indicators. The British Ministry of Defence (MoD) has taken this approach, for instance, in the first draft of its proposed standards for safety-critical software (MoD 1991a, 1991b).

3.8.1. No interrupts.

The use of interrupts, beyond a simple clock interrupt, is considered a higher-risk implementation method because of the extra care required to ensure correct synchronization between interrupt code and interrupted code, and to ensure that interrupted code is correctly resumed.

3.8.2. No multi-tasking.

Multi-tasking requires context switching and task management in addition to the complications attendant upon using interrupts.

3.8.3. Simple loop.

A single-loop program structure is the simplest program organization capable of continuous operation that is possible.

3.8.4. Deterministic, predictable timing.

Evidence that software product timing is a predictable function of load, and that load is limited by design is a positive factor. See Preckshot 1993b for additional information.

3.8.5. No pointers.

The use of explicit pointers (addresses) of data has been taken by some as a risky practice. The potential exists for errors in programmer-directed address arithmetic which would not exist if named variables were used and the addresses were computed automatically by compiler.

3.8.6. Strong data typing.

Data typing permits compilers to detect data misuse errors (e.g., using an integer as if it were a floating point number). This class of error represents a significant proportion of all errors made, and strong data typing with good compilers almost eliminates it.

4. CONCLUSION

The primary conclusion from the work listed in Section 1.2 is that there is no known method for absolutely guaranteeing that a software system is adequate for a

safety-critical application. In this respect, software is no different from any other method of achieving safety.

As a result, both developers and assessors find themselves attempting to reduce the probability of errors in software leading to accidents to acceptable levels in an environment in which relevant quantitative measures are lacking or difficult to apply. The short history of software engineering of safety-critical software, compared to other forms of engineering, increases unease.

A variety of evidence will be required for an assessor to accept software whose correct operation is critical to safety. This evidence involves an examination of the history and culture of the development organization, the actual process used to develop the application under review, and characteristics of the programs and documents which result from that process.

The Carnegie Mellon University Software Engineering Institute (CMU/SEI) has defined a Capability Maturity Model (CMM) for assessing software development organizations. The model defines five levels of maturity, each with specific characteristics (See Table 1). The mandatory and essential design factors listed in Section 2 above can be matched against the characteristics of the different maturity levels. The result of this is that any organization that wishes to develop software for a safety-critical nuclear application should be assessed at the equivalent of level 3.

Considerable assistance in achieving safe software can be provided by using company, industry, national or international software engineering standards. If a well designed set of company standards exists, they are preferred since the personnel in the company should be familiar with them, and understand how to use them. The set of IEEE Software Engineering Standards provides an excellent source of standards, and should continue to improve over the next few decades (Lawrence 1993).

There is a great deal of emphasis in the literature on the negative consequences of software failures in safety-critical applications, and this is appropriate. However, there are also several balancing, positive factors which deserve equal emphasis. In particular, software does not wear out, potentially can be used to identify and compensate for hardware failures, and potentially can provide much greater control to operators during unexpected events. These factors should be carefully evaluated on a case-by-case basis to determine the suitability of software in each plant application.

The challenge faced by software developers is to use software safely to increase the reliability of the application, while the challenge for assessors is to ensure that this is done. The research presented in this series of reports suggests that convincing evidence can be obtained in practice that reliable safety-critical software is being or has been developed. However, neither the development of such software nor the effort required to certify it for safety-critical usage is easy.

Table 1. Key Process Areas by Maturity Level

Level Number	Level Name	Level Characteristics
1	Initial	(none)
2	Repeatable	Software configuration management Software quality assurance Software subcontract management Software project tracking and oversight Software project planning Requirements management
3	Defined	Peer reviews Intergroup coordination Software product engineering Integrated software management Training program Organization process definition Organization process focus
4	Managed	Software quality management Quantitative process management
5	Optimizing	Process change management Technology change management Defect prevention

Paulk 1993

Section 4. Conclusion

REFERENCES

Lawrence 1992. J. Dennis Lawrence, *Workshop on Developing Safe Software: Final Report*, UCRL-ID-113438, Lawrence Livermore National Laboratory (November 1992).

Lawrence 1993. J. Dennis Lawrence, *Software Reliability and Safety in Nuclear Reactor Protection Systems*, NUREG/CR-6101, UCRL-ID-114839, Lawrence Livermore National Laboratory (November 1993).

Lawrence 1994. J. Dennis Lawrence and Warren L. Persons, *Survey of Industry Methods for Producing Highly Reliable Software*, UCRL-ID-117524, Lawrence Livermore National Laboratory (June 1994).

McCall et al. 1977. Jim A. McCall, Paul K. Richards, and Gene F. Walters, *Factors in Software Quality: Concept and Definitions of Software Quality*, RADC-TR-77-0369, Rome Air Development Center (November 1977).

MoD 1991a. British Ministry of Defence, *The Procurement of Safety Critical Software in Defence Equipment Part 1: Guidance,*Interim Defence Standard 00-55, 5 April 1991a.

MoD 1991b. British Ministry of Defence, *The Procurement of Safety Critical Software in Defence Equipment Part 2: Requirements*, Interim Defence Standard 00-56, 5 April 1991b.

Paulk et al. 1993. Mark C. Paulk, Bill Curtis, Mary Beth Chrissis, and Charles V. Weber, *The Capability Maturity Model for Software*, Version 1.1, Software Engineering Institute, Carnegie Mellon University, CMU/SEI-93-TR-24 (February 1993).

Ploof and Preckshot 1993. Frank Ploof and Gary Preckshot, "Subject: Assessment of Vendors (Task 16)," Letter to John Gallagher, NRC, LLNL CS&R 93-08-18, (August 20, 1993).

Preckshot 1993a. G. Gary Preckshot, *Real-Time Systems Complexity and Scalability*, UCRL-ID-114566 (1993).

Preckshot 1993b. G. Gary Preckshot, *Reviewing Real-Time Performance of Nuclear Reactor Safety Systems*, NUREG/CR-6083, UCRL-ID-114565, Lawrence Livermore National Laboratory (August 1993).

Persons 1994. Warren L. Persons and J. Dennis Lawrence, *Assessing Safety-Critical Software in Nuclear Power Plants*, Lawrence Livermore National Laboratory (in prep).

Scott 1994. John A. Scott and J. Dennis Lawrence, *Testing Existing Software for Safety-Related Applications*, Lawrence Livermore National Laboratory (in prep).

Technical Information Department • Lawrence Livermore National Laboratory
University of California • Livermore, California 94551