APPENDIX

A.0 Impact of Design and Coding Techniques on Software Reliability.

Software design characteristics and techniques are more dynamic for software than for hardware, however, just as with hardware, there is a definite correlation between design techniques and reliability. It makes sense to assume that the fewer the faults in a software system, the higher the reliability will be. Therefore, a developer should attempt to generate as few faults as possible (fault avoidance), should consider techniques to withstand faults (fault tolerance), and ensure that software is readily repairable when a fault is found (fault removal). To develop a system that is reliable, apart from performing any reliability analysis, a developer should consider those three elements. In the design process a general model can be used. In a sequential design process a series of steps are followed consisting of a specification phase followed by a realization phase. Realizations become less abstract as the system is decomposed into more detailed specifications.

Since faults are the underlying cause of failures in software, controlling the number of faults introduced in each development step and the number of faults that propagate undetected to the next development step is important in managing product reliability. Faults must be managed across all phases of the life cycle. Many development practices affect fault management. A few of the more important ones are as follows:

- *Practicing a development methodology*. Using a common approach in translating high-level design into code, and in documenting it, is particularly important for larger projects. It facilitates good communication between project team members, helping to reduce faults.
- *Constructing modular systems*. A modular system consists of well-defined, simple and independent parts interacting through well-defined interfaces. Small, simple modules are easier for designers and programmers to build and thus less prone to faults being introduced in the design process. Also, modular designs are more maintainable, further increasing the chance that detected faults are properly repaired.
- *Employing reuse*. Reuse of software components that have been well tested for an operational profile that is close to that expected for the new system reduces fault introduction. The alternative is to develop new components, which will have to go through a complete fault introduction and removal process inherent to most software development projects.
- *Unit and integration testing*. Testing plays a major role in preventing faults from propagating to a next development step. Unit tests verify modules' functions as specified in their low-level designs. Integration tests verify that modules interact as specified in the higher-level design (architecture).
- *Conducting inspections and reviews*. Inspections and reviews can be held on requirements, design documents, software code, user manuals, training materials, and test documents. Both reviews and inspections use a small team to compare the output of a development step with what was specified as input to that step.

- *Controlling change*. Many failures result from change in the intermediate items produced as part of the completed product. Such intermediate items include the code of software components, design and requirements specifications, test plans, and user documentation. To reduce the occurrence of such failures, *version control* is required. Also, an orderly procedure must be used to handle requested changes to items (*change control*). Version and change control together are referred to as *configuration management*.

Three Steps[1]

Fault Avoidance

The initial approach to software design is one where faults are less likely to be introduced into the software. The structural description of the software at any level of detail should always be supplemented with a precise specification that provides a complete description of the realization at that level. Configuring the design in this manner factors out the design detail so that clearness is maintained along with thoroughness and accuracy.

Reliability should be a design objective; unfortunately it is not a design characteristic that can be evaluated by a static analysis of the design. Software reliability depends on quality factors such as checkability and understandability, complexity and modifiability of the design. Two attributes which may be most important concerning reliability include *visibility* and *coupling* of the design.

design visibility. The visibility of a design is an appraisal of how well the attributes of a design specification are reflected in the fulfillment of that design. This means it should be feasible to take any part of the specification and distinguish where that part is achieved. The implementation may be expressed in a form that is readily understandable and structured to reflect the specification.

design coupling. The coupling of a design is an appraisal of the dependencies of the units of that design. Low coupling means that the design is made up of mutually independent modules with few shared data structures. These characteristics are important regarding reliability for the following reasons:

- The achievement of reliable software depends on continuous validation during software development. This validation process is simplified if each design unit can be considered alone, minimizing reference to other design units.
- Static design analysis can be a very effective means of detecting design faults. If the transition between design levels is clear, this analysis is simplified, increasing the probability of finding errors in that transition.
- The repair of faults in software subsystems is least likely to cause faults in other subsystems if the systems are independent. Further, the discovery of faults is simplified if the design is visible. However, even with independent subsystems some classes of faults such as timing faults may be introduced when one of the subsystems is modified.

[1] Software Reliability Handbook, Edited by Paul Rook, Centre for Software Reliability, Elsevier Applied Science, London, 1990.

Defect Removal

In addition to defect prevention, to produce high-quality, reliable software, it is important at all stages of development to look actively for defects or potential defects, and take corrective action for those detected. Defect detection can accurately be called *testing*. The fundamental strategy of testing, particularly in defect detection, is central to the process of producing quality software.

The development of software occurs over a number of phases. At the end of each phase, it is important minimize the defects passed on to later phases. If a problem such as a design error or misunderstood requirement is passed on, it will surface later. Correction may require major rework, tracing the problem back to its source and repeating development from there. Removal of a defect later in the life cycle is much more expensive. Therefore, defect detection must be implemented rigorously at each phase in order to capture defects as early as possible.

It must be recognized that the input to each phase may contain undetected defects that could cause the wrong output in that phase. When testing the result of a phase, it would be prudent to refer back to earlier phases. In testing there are two terms conventionally used, *verification* and *validation*. The distinction between the two can be described as:

> *Verification*: testing that the system is developed correctly
> *Validation*: testing that you have developed the correct system

The products of a phase are tested with respect to the requirements imposed for that phase. For this to occur it is important that the requirements are *testable*. This means that any requirement should include a means of verifying conformance with that requirement. The development of the product must recognize the defect detection and correction stages that follow, and development must be undertaken to facilitate these. This requires attention to a number of issues.

Requirements must be *traceable* to the part of the product that meets that requirement, and every part of the product must be *traceable* back to the requirement. Some form of *compliance* matrix could be produced to assist this. Alternatively, the information may be recorded in the configuration management system. Traceability assists in the verification of the product against its requirements. Traceability is assisted by *modularization*.

Testing that follows the development activity needs to be carefully planned. It is important that, as a minimum, all direct requirements are tested for. If appropriate, indirect requirements may also be considered. In addition to conformance to specific requirements, there may also be general requirements like usage of standards.

Initially, *test requirements* or *specifications* are established, to determine the content and detail of testing. Then *test plans* or *procedures* are produced. These describe how the tests will be carried out. Test planning and preparation should ideally be done by people independent of the developers of the product to be tested.

Records should be kept on the defects found and the underlying faults. The fault database allows defect *prevention* to be built back into the process. The development process which produced the faults should be investigated to determine the cause, and the process corrected. Tests, inspections and audits should be modified to ensure that they uncover any recurrence of a fault type immediately. During the execution of a phase, information should be gathered from all sources available. This information may include statistics about the defects found and the underlying faults. The information is used to make improvements to the process and eliminate more defects.

Software Fault Tolerance

All approaches for attempting to prevent faults combined will likely not be totally successful, which is why software reliability engineering is a legitimate discipline. Therefore, in effective engineering of reliable software, it may be prudent to augment fault prevention with approaches that attempt to minimize or control the effects of residual faults. Software fault-tolerance is the main approach. Applying fault-tolerant methods, however, adds a level of complexity which may cause more faults in the process as well as impact performance.

The dynamic behavior of a software system is designated by the series of *internal states* which the system experiences during its processing. Each internal state is comprised of a set of data values within the scope of the design: output values produced by the software (external states) and the values of any variables sustained directly by the design. Under normal processing conditions, the software will progress from one valid internal system state to the next by means of a *valid transition*. However, if a *fault* is encountered in the software during its processing, an *erroneous transition* may occur which transforms the system to an invalid internal state containing one or more defective values or *errors*. Once the system state is damaged, subsequent invalid states can be produced from valid transitions. Alternatively, if the subsequent processing repairs the damage (e.g., by overwriting an incorrect variable with a new correct value), then the system can revert to a valid internal state. If an error in an internal state maps on to the external state, then a failure of the system occurs. Consequently, all system failures can be attributed to errors in the internal state of the system. All errors, however, need not result in a failure[2].

[2]Software Reliability Handbook, Edited by Paul Rook, Centre for Software Reliability, Elsevier Applied Science, London, 1990.

The objective of software fault tolerance is to prevent software faults from causing system failure. In order to allow the system to operate successfully in the presence of a system design fault, the software must be constructed from a number of diverse system designs which have a low probability of exhibiting common-mode failure. The four major elements of fault-tolerance are:

1. *Error Confinement*. Software must be written in such a way that when an error occurs, it cannot contaminate portions of the software beyond the local domain where it occurred;

2. *Error Detection*. Software must be written such that it tests for and responds to errors when they arise;

3. *Error Recovery*. Software must be written such that after detecting an error, it takes sufficient steps to allow the software to continue to function successfully; and

4. *Design Diversity*. Software and its data must be created in such a way that there are fallback versions available.

Tables A-1 and A-2 are a summary of just some of the specific design and code techniques that are related to error confinement, error detection, error recovery and design diversity. The design and code techniques have been correlated to reliability and fault tolerance.[3]

TABLE A-1. Software Design Techniques

Design techniques
• Recovery designed for hardware failures
• Recovery designed for I/O failures
• Recovery designed for communication failures
• Design for alternate routing of messages
• Design for data integrity after an anomaly
• Design for replication of critical data
• Design for recovery from computational failures
• Design to ensure that all required data is available
• Design all error recovery to be consistent
• Design calling unit to resolve error conditions
• Design check on inputs for illegal combinations of data
• Design reporting mechanism for detected errors
• Design critical subscripts to be range tested before use
• Design inputs and outputs within required accuracy

[3]Science Applications International Corporation & Research Triangle Institute, Software Reliability Measurement and Testing Guidebook, Final Technical Report, Contract F30602-86-C-0269, Rome Air Development Center, Griffiss Air Force Base, New York, January 1992.

TABLE A-2. Software Coding Techniques

Coding techniques
• All data references documented
• Allocate all system functions to a CSCI
• Algorithms and paths described for all functions
• Calling sequences between units are standardized
• External I/O protocol formats standardized
• Each unit has a unique name
• Data and variable names are standardized
• Use of global variables is standardized
• All processes within a unit are complete and self contained
• All inputs and outputs to each unit are clearly defined
• All arguments in a parameter list are used
• Size of unit in SLOC is within standard
• McCabe's complexity of unit is within standard
• Data is passed through calling parameters
• Control returned to calling unit when execution is complete
• Temporary storage restricted to only one unit - not global
• Unit has single processing objective
• Unit is independent of source of input or destination of output
• Unit is independent of prior processing
• Unit has only one entrance and exit
• Flow of control in a unit is from top to bottom
• Loops have natural exits
• Compounded booleans avoided
• Unit is within standard on maximum depth of nesting
• Unconditional branches avoided
• Global data avoided
• Unit outputs range tested
• Unit inputs range tested
• Unit paths tested

A.1 The Link Between Software Reliability and Software Safety.

A.1.1 Use of Hypothesis Testing for Software Safety.

According to David L. Parnas in *Evaluation of Safety-Critical Software*[4], the vast literature on random testing is, for the most part, not relevant for safety evaluations. In lieu of reliability growth models, a very simple model can suffice. Hypothesis testing allows an evaluation of the probability that the system meets safety or high-reliability requirements (.99 or greater).

In many safety-critical applications it may not be necessary to know probability of failure; instead, it is important to confirm the failure probability is very likely to be below some upper bound. One method is to run random tests on the software, checking the results of each test. Since it is safety-critical software, if a test fails the software will be changed to correct the underlying fault. Random testing will resume. These tests continue until sufficient data is available to believe the probability of a failure is acceptably low. Because only a very small fraction of the conceivable tests can be run, there can never be 100% certainty that the failure probability is low enough. However, it is possible to calculate the probability that a product with unacceptable reliability would have passed the test (Type I Error).

A.1.2 Ultra High Reliability and Safety.

Assume the probability of a failure in a software test is 1/h. Assuming that N randomly selected tests (with replacement, from the operational profile) are performed, the probability there will be no failure encountered during the testing is $(1 - 1/h)^N = M$. In other words, if the failure probability is less than 1/h, and N tests have been run without failure, the probability that an unacceptable product would pass the tests is no higher than M. Testing must continue, without failure, until N is large enough to make M acceptably low. Then statements like, *the probability that a product with reliability worse than 0.999 would pass this test is less than one in a hundred*, could be made. Restated, *there is 99% confidence that the reliability is no worse than 0.999*. So, if a reliability requirement and a confidence level is established, the number of tests, N, which must pass consecutively to allow the software to pass the test, can be determined.

Using the equation above, if the design target was to have the probability of failure be less than 1 in 1000, performing between 4500 and 5000 tests (randomly chosen) without failure would mean that the probability of an unacceptable product passing the test is less than 1 in 100.

Because the probability of failure in practice is a direct function of the distribution of cases encountered in practice, the validity of this approach depends on the distribution of cases in the tests being typical of the distribution of cases encountered in practice (operational profile).

The same approach can be considered to obtain a measure of the *trustworthiness* of a program. Let the total number of cases from which tests are selected be C. Assume that it is unacceptable if F of those cases results in faulty behavior. By substituting F/C for 1/h gives $(1 - F/C)^N = M$. Now assume that N

[4] David L. Parnas "Evaluation of Safety-Critical Software", Communications of the ACM, Vol. 33, No. 6, June 1990

randomly selected tests have been run without finding an error. If, during testing, an error had been found, the problem would be corrected. The value of C can be estimated, and then it must be decided whether to use F=1 or some higher value. A higher number might be selected if it were unlikely that there would be only one faulty pair. In most computer programs, a programming error would result in many faulty pairs. After choosing F, M can be determined as above. (F,M) pairs provide a measure of trustworthiness. Systems considered trustworthy would have relatively low values of M and F.

Because C is almost always large and F relatively small, it is not practical to evaluate trust-worthiness by means of testing. Trustworthiness, in the sense defined here, must be obtained by means of formal, rigorous inspections.

Three classes of programs

The simplest class of programs to test comprises those that terminate after each run and retain no data between runs. These memory-less batch programs are given data, executed, and produce an output independent of earlier runs.

A second class consists of batch programs that retain data from one run to the next. The behavior of these programs on the *n*th run can depend on data supplied in any previous run.

A third class consists of programs that run continuously. These real-time programs consist of one or more processes. Some processes are run periodically, while others are randomly spawned in response to external events. Discrete runs cannot be identified, and the behavior at any time may depend on arbitrary events in the past.

- *Reliability estimates for memory-less batch programs*
  For a memory-less batch program a test consists of a single run using a randomly selected input set.

- *Reliability estimates for batch programs with memory*
  When a batch program has previous data dependencies, a test consists of a single run. However, a test case is comprised of both input data and an internal state. For reliability estimates, the distribution of internal states should approximate that expected in field operation. It may be more difficult to determine the appropriate distribution of internal states than to derive the input space. Determining the distribution of internal states requires an understanding of the software program.

- *Reliability estimates for real-time systems*
  In real-time systems, the concept of a batch run is not applicable. Because the real-time system is intended to simulate or replace an analogous system, the concept of an input sequence must be replaced by a multidimensional run. Each profile gives the input values as continuos functions of time. Each test involves a simulation in which the software samples the inputs for the run length.

The question of the duration of the run is critical in determining whether or not statistical testing is practical. In many computer systems there are states that can arise after long periods. Reliability

estimates derived from tests involving short runs will not be valid for systems that have been operating for longer periods. On the other hand lengthy runs render the testing time required impractical.

Statistical testing can be made more practical if the system design is such that a limit on the length of the runs can be established without invalidating the tests. To do this, the states must be partitioned. A small amount of memory is reserved for data that must be retained for arbitrary amounts of time. The remaining data are reinitialized periodically. The length of the period becomes the length of the test.

Testing to estimate reliability is only practical if a real-time system has limited long-term memory.

A.1.3 Picking Test Cases for Safety-Critical Real-Time Systems.
Particular attention must be paid to run selection if the system is required to act only in rare circumstances. Since the reliability is a function of the input distribution, the runs must be selected to provide accurate estimates under the conditions where critical performance matters. In other words, the population from which runs are drawn must include only runs in which the system must take action.

Determining the population of runs from which the tests are selected can be the most difficult part of the process. It is important to use knowledge of the physical situation to define a set of runs that can occur. However, there is always the danger that the model used to determine the runs overlooks the same situation overlooked by the programmer who introduced a serious fault. It is important that any model used to eliminate *impossible* runs be developed independently of the program. Most safety experts would feel more comfortable if, in addition to the tests using segments considered possible, some statistical tests were conducted with *crazy* runs.

A.1.4 Safety Analyses.
Risk is the possibility of something undesired occurring. *Safety* is relative freedom from risks. Quantitatively, risk is a composite of the probability and severity of loss. Risk is quantified to allow hazards to be ranked for prioritizing control and mitigation through expected loss.

Severity can be described as:
Catastrophic. Personnel: death. Facilities/equipment/vehicles: system loss, severe damage.
Critical. Personnel: severe injury/illness; required admission to a health-care facility.
Facility/equipment/vehicle: major system damage. Loss of primary mission capability.
Marginal. Personnel: minor injury/illness. System: Loss of non-primary mission capability.
Negligible. superficial injury/illness. Lost time less than one day. Less than minor damage.

Probability or likelihood can be described as:
Frequent - likely to occur repeatedly during the life cycle of the system.
Probable - likely to occur several times during the life cycle of the system.
Occasional - likely to occur sometime during the life cycle of the system.
Remote - not likely to occur in the life cycle of the system, but possible.
Improbable - probability of occurrence cannot be distinguished from zero.
Impossible - physically or logically impossible to occur.

The purpose of a software safety program is to *eliminate hazards or reduce their associated risk to an acceptable level*[5]. System safety engineering provides methods for identifying, tracking, evaluating and removing hazards associated with a system and ensures that safety is designed into the system in a timely, cost-effective manner, that the risk is minimized, and that the potential effects in the event of a mishap are minimized. Software can cause or contribute to a hazard by:

- Not performing a function required - failing to produce an output
- Performing a function not required - commission
- Performing a function out of sequence or at the wrong time
- Failure to recognize a hazardous condition requiring corrective action
- Inadequate response to a contingency
- Wrong decision as a solution to a problem that arises.
- Poor timing, resulting in a response that is too late or too soon for an adverse situation

Safety is a property of an executing program, just like reliability. In software reliability, every failure is taken into account. Reliability is concerned with the frequency of failure. Each failure also has a severity associated with its consequences. Reliability looks at frequency. Safety is only concerned with those failures that result in system hazards. While reliability concerns itself with whether the program is doing what is required, safety concerns itself with seeing to it that the software does not do bad things.

The following are safety analyses that apply to software:

Preliminary Hazard List	Hazard Analysis	Causes	Fault Tree Analysis
Backward Threading	Software Requirements Hazard Analysis	Identifying Safety-critical Design Elements	Design of Safety-critical CSUs

A.1.5 Software FMEAs and Fault Tree Analyses.

Table A-3 illustrates how two specific safety analyses can be applied to hardware/software systems.

[5] Voas, Jeffrey, Friedman, Michael, "Software Assessment: Reliability, Safety and Testability", John Wiley & Sons, NY, 1995.

TABLE A-3. How FMEA and Fault Tree Analyses Apply to Software

Safety Analyses	**How to apply**
Fault Tree Analysis	This can be applied in a similar manner as for hardware with just a few differences.* 1. Develop the top level failure events based on historical information on similar or previous products 2. Determine the severity and relative probability of each event. A threshold for severe and probable events is determined prior to this. 3. From the requirements phase forward, continue developing the fault tree and determine how and if the event can occur in the software system. Drive the requirements, design, code and test activities based on the results. 4. The tree is expanded or pruned based on the thresholds for severe and probability determined prior to analysis. 5. The software requirements, design, code and test cases should reflect the analysis results.
Failure Modes Effect Analysis	This can be applied in a similar manner as for hardware with just a few differences described below.* 1. The failure events for software typically are not predictable, i.e., there is normally no way to know for sure that a software fault is about to happen before it actually does. 2. There are typically only two repair activities that apply to software. A) restarting the software or B) Laboratory repair. Software units are not replaced with new software units on site. 3. As discussed earlier, individual failure rates are not assigned to CSU's as a CSU does not fail independently.

Key to table: * - One difference between hardware and software fault trees is that piece part failure rates are not assigned to individual CSU's. As discussed previously, an individual CSU does not have an independent failure rate as it is the operational profile that must be modeled.

A.2 Description of SEI CMM model[6].

The Software Engineering Institute has developed a Capability Maturity Model that is as a framework describing the key elements of an effective software process. It covers key practices for planning, engineering, management, development and maintenance that when followed may improve the ability for an organization to meet cost, schedule, and quality goals.

The five CMM levels are:

1. *Initial* - Ad hoc processes dependent on individuals. Cost, schedule and quality are unpredictable.

2. *Repeatable* - Policies for managing software are in place. Planning is based on prior experience. Planning and tracking can be repeated.

3. *Defined* - The process for developing the software is documented and integrated coherently. The process is stable and repeatable and the activities of the process are understood organization wide.

4. *Managed* - The process is measured and operates within measured limits. The organization is able to predict trends and products are of predictable quality.

5. *Optimizing* - The focus is on continuous process improvement. The organization can identify weaknesses in the process and is proactive in preventing faults. The organization is also innovative throughout.

As was shown in Section 7, the CMM level can be tied to the fault density prediction.

A.3 Matrix of Skill Sets for Software Reliability.

Software reliability has generally not yet become an established part most organizations. Typically there is a software organization and a reliability organization but software reliability departments are a rarity. The question then is who performs the software reliability tasks? Table A-4 is presented as a guide to the skill set needed for the tasks discussed in this notebook. The software reliability tasks cannot be performed by one engineering discipline working in a vacuum. For the tasks to be effective a team of individuals from several disciplines should collaborate.

[6] "Key Practices of the Capability Maturity Model", Version 1.1, Software Engineering Institute, SEI-93-TR-025, Pittsburg, PA, 1993.

TABLE A-4. Skills Required for Software Reliability Tasks

Task	**Skills Required**	**Cooperation and data required**
Reliability Allocation	1. Understanding of reliability engineering principles. 2. Understanding of software design characteristics. 3. Understanding of software fault characteristics and behavior.	Need to know overall reliability allocations and reliability block diagram.
Reliability Prediction	1. Understanding of reliability engineering principles. 2. Understanding of software design characteristics. 3. Understanding of software fault characteristics and behavior.	Need to have access to software engineers, software requirements, software designs, development plans and organization policies.
Reliability Growth Modeling	1. Understanding of software reliability models and which ones to use under which fault profile conditions. 2. Understanding of what data is required for model and how to effectively and efficiently collect that data. 3. Needs to be involved with software testing organization.	1. Must be able to collect failure data during testing. 2. Must have access to software failure reporting systems. 3. Must have automated tools to aid in this modeling.
Fault Tree Analysis and/or FMEA	1. Basic understanding of how to generally apply these. 2. Intimate understanding of the software requirements and design. 3. Must be performed with or by software engineers or someone very familiar with software design.	1. Must have access to any historical failure events. 2. Must have access to requirements, design and possibly code. 3. Must be able to influence the test plan development.

Table A-5 describes the typical engineering components and how they can be involved in the software reliability process:

TABLE A-5. Relationship of Engineering Disciplines

Engineering discipline	**Potential primary software reliability responsibility**
Software engineers	• Providing information needed for predictions. • Analyzing, prioritizing and scheduling failures for corrective action. • Making improvements/adjustments based on analyses. • Managing the project and the schedule with respect to reliability growth parameters and predictions.
Software Quality Engineers or Software Test Engineers	• Aiding the software engineers in collecting and/or organizing failure data. • Recording and tracking failure events. • Participating in the analysis of the failure events.
Reliability Engineers	• Determining prediction technique in conjunction with software engineering. • Determining reliability growth modeling technique based on exhibited fault profiles in conjunction with software engineering. • Interviewing software engineers to collect data required for predictions. • Using failure data generated during testing and development for prediction and growth models.

A.4 Differences Between Software and Hardware Reliability.

- MTTR does not exist for software. The reason is that software units are not replaced in the manner that hardware units are. When software fails, the configuration changes permanently in order to correct the fault. Also, software is generally not corrected in the field. The software may, however, be restored or restarted in the field, but this does not constitute a corrective action.

- Since MTTR does not exist for software, MTBF may not be very helpful. The term MTTF may be more appropriately used instead.

- Software failure rates can be allocated to the CSCI level because CSCIs are generally assumed to be independent and to fail independently. However, assigning failure rates to lower level CSUs is not valid since CSUs do not fail independently. Even though software units may be designed to be modular and cohesive from a data and processing point of view, they are still never independent. Software units experience failures in association with their operational profile and not due to environmental wear out.

- Whenever a corrective action or any other modification is made to a software system, that change results in a new configuration. Therefore there is a risk that functionality will be inadvertently changed after a corrective action or modification takes place.

- Error or fault seeding (the process of inserting software faults into the existing software version for the purposes of measuring test effectiveness and reliability) is not applicable to software for many reasons. The most important is that software faults are caused by 4 types of human error: 1) requirements 2) design 3) code and 4) corrective action or bad fix. Other reasons include:

  Error seeding assumes that all faults are coding related and that errors representing the types that would typically be committed during development are known and established.

  Error seeding also does not consider the fact that during testing, corrective actions are being made thereby changing the configuration of the software.

  Error seeding is also not advised because seeded faults can mask real ones.

- Software maintainability and hardware maintainability are divergent disciplines. Software maintainability means that a given software unit can be modified with relative *ease by the software engineer who developed it or engineer responsible for corrective action*. Software maintainability does not reflect the effort required by an end user or operator to restore software functionality (the only maintenance action that can be made by an end user).

Many of the differences between hardware and software reliability, particularly from a reliability engineer's viewpoint, are discussed in *Evaluation of Safety-Critical Software*[7] by David L. Parnas, as summarized in the next sections.

A.4.1 Does Software Reliability Make Sense?

Developers, users and military organizations are often concerned about the reliability of systems that include software. Over the years, reliability engineers have developed detailed and elaborate methods of estimating the reliability of hardware systems based on an estimate of the reliability of their components. Software can be viewed as one of those components, and an estimate of the reliability of software is considered essential to estimating the reliability of the overall system.

Traditional Reliability engineers are often misled by their experience with hardware. Their background has them concerned with the reliability of devices that work correctly when new, but wear out and fail as they age. In other cases, they are concerned with mass-produced components where manufacturing techniques introduce defects that affect a small fraction of the devices. These hardware failure mechanisms result in the *bathtub* curve. Neither of these situations applies to software. Software does not wear out, and errors introduced when software is copied are minimal.

As a result of these differences, it is not uncommon to see reliability assessments for large systems based on an estimated software reliability of 1.0. Many reliability engineers are convinced that software errors are only due to poor design, so software failures are deterministic. The software is either correct (R=1.0) or incorrect (R=0). Assuming a reliability of 0 doesn't provide them a useful reliability estimate for the system containing the software. As a result, they assume correctness. Many consider it nonsense to talk about *reliability of software*. They say the two words together do not make sense.

Nonetheless, practical experience has shown over and over that software approximates stochastic properties. It is quite useful to associate reliability figures such as MTTF with an operating system or other software product. Some software experts believe the approximate random behavior is a result of ignorance. They think that all software failures would be predictable if the software were fully understood, but the inability of software engineers to understand their own creation justifies the treatment of software failures as random.

When a program fails, it is the result of an input state that had not occurred before. The reason that software appears to behave randomly, and that it is useful to talk about the MTTF of software, is that the input sequences for any particular case is unpredictable. Over the long run the input space will cover some predictable distribution of inputs, thus the software operation can be considered a stochastic process. Addressing the MTTF of software systems is done by predicting the probability of encountering an input state that will cause the system to fail.

[7] David L. Parnas, "Evaluation of Safety-Critical Software", Communications of the ACM, Vol. 33, No. 6, June 1990

Strictly speaking from a reliability engineer's point of view, software should not be considered as a component in systems at all. The software is simply the initial data in the computer and it is the initialized computer that is the component in question. However, in practice, the reliability of hardware is high and failures caused by software errors dominate those caused by hardware in most current systems.

A.4.2. What Should be Measured?

If the sequences of inputs that lead to failure could be accurately characterized, it would be simple to measure the distribution of input histories directly. Due to ignorance, however, the software itself must be used to measure the frequency with which failure- inducing sequences occur as inputs.

In safety-critical applications, particularly those for which a failure would be considered catastrophic, it may be prudent to take the position that design errors that would lead to failure are always unacceptable. In other technologies a system with a known design error would not be put in service. The complexity of software, and its consequent poor track record, means that seldom can there be confidence that software is free of serious design errors. Under those circumstances, it may be appropriate to evaluate the probability that serious errors have been missed by testing. This gives rise to a second probabilistic measure of software quality, *software trustworthiness*. Software trustworthiness is the probability that no serious design errors remain after the software passes a set of randomly chosen tests.

A.4.3 Software Failure Rate Cannot be Predicted From Failure Rates of Components.

The fundamental tenet of hardware system reliability studies is the computation of the reliability when given the reliability of the parts. It is tempting to try to do the same thing for software, but the temptation should be resisted. The modules of a program are not analogous to the components of a hardware system. The components of a hardware system operate independently and concurrently. The units of a computer program function sequentially and the effect of one execution depends on the state that results from earlier executions. A failure at one part of the code may lead to problems elsewhere in the code. When evaluating the reliability of a software product, the only sound approach is to treat the whole computer, hardware and software, as a black box.

A.4.4 The Finite State Machine Model of Programs.

Used for more than five decades, the finite state model recognizes that every digital computer has a finite number of states and there is a limit to the number of possible input and output signals at any point in time. Each machine is described by two functions: *next-state* and *output*. Both have a domain consisting of (state, input) pairs. The range of the next-state function is the set of states. The range of the output function is the set of symbols known as the output alphabet. These functions describe machine behavior that starts in an initial state and periodically selects new states and outputs in accordance with the functions.

In this model, it makes sense for the software to be viewed as part of the initial data. It determines the initial state of the programmed machine. Von Neumann introduced a machine architecture in which

the program and data could be combined. Code can be replaced with data or vice-versa. It does not make sense to deal with the program and data as if they were different.

The software can be viewed as a finite state machine described by two very large tables. This model of software enables a definition of the number of faults in the software; it is the number of entries in the table that specify faulty behavior. This fault count does not have a simple relation to the number of statements that must be corrected to remove the faults. It serves only to help determine the number of tests that need to be performed.

A.5 Software Testability.
According to Voas and Friedman[8]:

Software Testability analysis predicts the likelihood that if there are faults in the software, they will be revealed through testing. The analysis is used to optimize the testing process to determine how much testing is enough, determine where to concentrate resources, and determine the value of any particular testing approach.

The Software Engineering Life Cycle is simply a process where many development decisions are made at various points during the process. These decisions directly impact future decisions during the process and eventually affect the software product itself. At the early phases in the life cycle, the emphasis is on *achieving* quality in the end product; later in the life cycle, emphasis is shifted toward *assessing* and *assuring* how much quality has been achieved. Achievement and assessment will both, however, occur throughout the life cycle.

Testing plays a role in both achieving and assessing quality. In most cases a particular testing technique is intended for use in either achieving or assessing, but not both. An organization should not try to use a quality achievement technique as a quality assessment technique. Software testing is a validation and verification (V&V) technique that occurs late in the software life cycle. As mentioned in Section A.0, the distinction between the two can be described as:

> *Verification*: testing that the system is developed correctly
> *Validation*: testing that you have developed the correct system

Testability is the degree to which a system or component facilitates the establishment of test criteria and the performance of tests to determine whether those criteria have been met. Jeffrey Voas defines software testability of a program *P* to be a prediction of the probability that if a fault exists in *P*, the fault will be detected by whatever testing means are applied.

[8] Voas, Jeffrey, Friedman, Michael, "Software Assessment: Reliability, Safety and Testability", John Wiley & Sons, NY, 1995.

A.6 Computing Complexity.

McCabe's Complexity[9]
Step 1. Determine where the branches in logic are. These include:

- if-then
- if-then-else
- loops
- returns
- GOTO's (not good structure)
- case statements

Count the branches in logic and add 1. The minimum complexity any unit can have is 1. According to Thomas McCabe, the recommended maximum threshold is 11 for each unit.

Functional complexity
This metric is subjective. Ideally each unit should have a functional complexity of one, meaning the unit performs one indivisible function. The following can help in determining how many indivisible functions a unit performs:

A. Are all inputs related to each other?
B. Are all outputs related to each other?
C. Are all inputs and outputs related to each other?
D. Can the unit be concisely named?
E. Can the unit be understood just by knowing the inputs and outputs?

Function Points
The Software Productivity Research, Inc. method of estimating functions points is summarized as[10]:

A. Determine problem complexity - Are algorithms
    1. Simple
    2. Mostly simple
    3. Average complexity
    4. Some difficult
    5. Many difficult or complex

B. Determine code complexity - Are modules
    1. non-procedural
    2. well structured and/or reusable
    3. well structured and small

[9] Thomas McCabe, "Structured Software Testing", McCabe & Associates, Columbia, Md, Course Materials, 1985.
[10] Jones, Capers;"Applied Software Measurement", McGraw-Hill, NY, 1995.

4. fair structure, some complex
5. poor structure, complex and large

C. Determine data complexity - Are file structures and data relationships
1. Simple with few variables
2. Numerous but simple
3. Multiple files, fields and data relationships
4. Complex structure and interactions
5. Very complex structure and interactions

D. Add the complexity adjustments from A through C
E. Compute number of inputs and multiply by weight of 4
F. Compute number of outputs and multiply by weight of 5
G. Compute number of inquiries and multiply by weight of 4
H. Compute number of data files and multiply by weight of 10
I. Compute number of interfaces and multiply by weight of 7
J. Add total of E through I
K. Multiply complexity adjustment by J

Feature Points

The Software Productivity Research, Inc. method of estimating feature points[11]:

A. Determine number of algorithms and multiply by 3
B. Determine number of inputs and multiply by 4
C. Determine number of outputs and multiply by 5
D. Determine number of inquiries and multiply by 4
E. Determine number of data files and multiply by 7
F. Determine number of interfaces and multiply by 7
G. Add A through F
H. Multiply by complexity adjustment

[11] Jones, Capers;"Applied Software Measurement", McGraw-Hill, NY, 1995.

A.7 Software Metrics.

Establishing a software metrics program is paramount to having accurate software reliability predictions, allocations and estimations. It is also pertinent to making any tactical or strategic improvements, including cost and productivity as well as reliability improvements. Historical data based on an organization's development practices and industry (providing it is complete) is superior to using any other data for prediction or measurement purposes. TABLE A-6 is a summary of the data that should be collected. From this data the following cost measures can also be calculated:

- Cost per defect - Increases by a factor of 10 for each phase it goes undetected in.
- Cost of rework = *Percentage of rework x Cost of corrective action x Inherent cost of defect*.

  Percentage of rework can be measured as:
  - percentage of project time spent re-analyzing, re-designing, re-coding, re-testing what has been analyzed, designed, coded, or
  - percentage of project effort in man-years spent re-analyzing, re-designing, re-coding, re-testing what has been analyzed, designed, coded.

- Productivity - how much product, resources and calendar needed to complete project

According to Boehm, the cost of a defect increases by a factor of ten for each phase that it remains in the software product undetected. This is because more personnel and resources are required to address a defect towards the later phases of the life cycle. Several metrics for computing productivity exist. These include but are not limited to:

- The Software Life Cycle Model - Quantitative Software Management[12]
- COCOMO[13] - Barry Boehm
- Function point productivity[14] - Software Productivity Research, Inc.

Some types for a good metrics program:

- Measure the process or product but avoid measuring the individuals
- Use measures that are easily understood
- Use measures that are correlated to the item being measured
- Use measures that can be automated

Section 7.2.3 contains empirical data associated with many of these metrics shown above.

[12] Putnam, Larry, "Measures for Excellence", Yourdon Press, Englewood Cliffs, NJ., 1992.
[13] Boehmn, Barry, "Software Engineering Economics", Prentice Hall, Inc., Englewood Cliffs, NJ., 1981.
[14] Jones, Capers, "Applied Software Measurement", McGraw-Hill, NY, 1991.

TABLE A-6. Suggested Software Metrics

Size	• SLOC - Executable source lines of code not including blank lines and comments • Function points • McCabe's complexity
Failure and defect data	• Experienced fielded failures When & How (operational characteristics) • Defect removal efficiency (percent of corrective actions that are successful at removing fault) • Defect density • Total number detected and corrected at any time • Profiles such as severity, root cause, inputs that caused failure event • Number of corrective actions that require rework • Defects detected per phase including requirements, design and code reviews.
Development environment characteristics	• How was the software developed? Methods? Tools? Organization structure? Standards used? Techniques used?
Operational/execution/calendar time in testing or usage	• Needed to compute failure rate
How long and how many	• How many calendar months from start of project to delivery • How many man-months of effort from start of project to delivery - including everything.

A.8 Additional Information on the Keene-Cole Model[15].

Dr. Samuel Keene and G.F. "Gerry" Cole have developed a reliability growth model for fielded software that incorporates two factors into a exponential growth profile model. These two factors are the recurrence factor ρ and the usage factor μ.

If the reliability growth profile during testing follows a saturating profile with time that is an exponential curve and the number of faults F decreases proportionally to the number of faults in the system N then

$$F = Ne^{-kt} \qquad \textbf{(A.1)}$$

which is the anticipated reliability growth profile.

Traditional exponential reliability growth models such as the one above assume that once a fault is detected that it is immediately removed. The way to circumvent this limiting assumption is to predict a recurrence factor ρ that measures the average number of occurrences of a single fault. Keene and Cole found this number to be between 1 and 5.

The second factor considered is the impact of a fault occurrence on multiple copies of the software. The usage factor μ is a value between 1 and the number of copies of the software concurrently operating. Measuring μ is analogous to accumulating operating hours for multiple hardware systems.

Equation **(A.1)** is now modified for these two factors as

$$F = \rho Ne^{-k\mu t} \qquad \textbf{(A.2)}$$

Example:

1. Previous historical data shows that
   - N = 500
   - KSLOC = 250
   - $k\mu$ = .048
   - 56% will remain after first year

2. 500 x (1-.56) = 220 faults expected after first year.

3. The recurrence factor ρ is found to be 4. Historical data on this project shows that 25% of the faults are perceived by the user. So now ρ x .25 = 1.

4. MTTF = 8760 hours per year / 220 failures = 40 hours /failure average during first year of operation.

[15] Keene, Dr. Samuel, Cole, G.F. "Gerry", "Reliability Growth of Fielded Software", Reliability Review, Vol 14, March 1994.

A.9 Additional Information on the Musa Model.

One of the models discussed in Section 8.4-1 is the Basic Execution Time Model. It views the phenomenon of software failure as a Non-homogeneous Poisson Process (NHPP). The counting process $\{M(\tau), \tau \geq 0\}$ represents the cumulative number of software failures in the execution time interval $[0, \tau)$. The Greek letter tau, τ, is used for execution time to distinguish it from calendar time, t. Execution time is the failure-inducing stress for software. The model also has a "calendar time component" that relates τ to t. The mean value function is the expected cumulative number of failures in the interval:

$$\mu(\tau) \equiv E\{M(\tau)\} \quad \textbf{(A.3)}$$

The failure rate can be defined as the execution time derivative of the mean value function:

$$\lambda(\tau) \; equiv \; \frac{d\mu(\tau)}{d\tau} \quad \textbf{(A.4)}$$

When the program code is frozen and subjected to a stationary operational profile, the software is modeled as having a constant failure rate

$$\lambda(\tau) = \lambda, \; \tau \geq 0 \quad \textbf{(A.5)}$$

resulting in a (homogeneous) Poisson process. The probability that the software will execute for execution time τ' measured from the present is given by the reliability function

$$R(\tau') = \exp[-\lambda\tau'] \quad \text{(A.6)}$$

In the Basic Execution Time Model, ν_0 is the total number of failures that would have to occur to uncover all faults. These faults include ω_0 faults that were present at the start of system test--called *inherent* faults--plus any faults that might be inadvertently introduced into the program as the result of fault correction activity.

Not every failure results in exactly one fault being removed from the program code. Sometimes additional faults are discovered from code reading, when a failure reveals a whole class of closely related faults. And sometimes the fault that caused a failure is not found, or a new fault is introduced. In the model, the net number of faults removed per failure is called the *fault reduction factor*, denoted B. The fault reduction factor is related to the inherent faults and total failures by

$$B = \frac{\omega_0}{\nu_0} \quad \textbf{(A.7)}$$

The initial failure rate, the one at the start of system test, is denoted λ_0. The contribution of each fault to the overall program failure rate is called the *per-fault hazard rate*, denoted ϕ. The per-fault hazard rate is related to the initial failure rate and the inherent number of faults by

$$\phi = \frac{\lambda_0}{\omega_0} \qquad \textbf{(A.8)}$$

It is called a *hazard rate* (called also force of mortality [FOM]) because a fault is considered to have a finite lifetime: If testing is continued long enough, the fault will be discovered and corrected.

The failure rate is expected to improve as time goes on, as faults are removed from the code. Since B faults are removed per failure occurrence, the failure rate declines by $\beta = B\phi$ upon each failure. If m is the expected number of failures at time τ, then the overall program failure rate is

$$\lambda(\mu) = \beta(\nu_0 - \mu) \qquad \textbf{(A.9)}$$

or

$$\lambda(\tau) = \beta[\nu_0 - \mu(\tau)] \qquad \textbf{(A.10)}$$

Since

$$\lambda(\tau) = \frac{d\mu(\tau)}{d\tau} \qquad \textbf{(A.11)}$$

it must be the case that

$$\frac{\delta\mu(\tau)}{\delta\tau} + \phi\beta\mu(\tau) = \phi\nu_0\beta \qquad \textbf{(A.12)}$$

The solution to this differential equation provides the mean value function

$$\mu(\tau) = \nu_0(1 - \exp[-\beta\tau]) \qquad \textbf{(A.13)}$$

The parameters β and ν_0 can be determined by prediction or estimation. Prediction procedures depend on the software development phase in which the prediction is made (see Section 7).

Point estimation of model parameters.

Estimation establishes values for the Basic Execution Time Model parameters β and ν_0 based on the history of software failure during system test. The method of maximum likelihood estimation chooses the values of β and ν_0 that maximize the likelihood of obtaining the failure times that were in fact observed.

Once system test begins, the cumulative execution times

$$\tau_1, \tau_2, \ldots, \tau_{m_e} \qquad \textbf{(A.14)}$$

at which failures occur are recorded. The quantity m_e is the cumulative number of failures. The time at which the parameters are estimated is denoted τ_e and may or may not coincide with the time τ_{m_e} of the last failure. This "time censoring" is taken into account in the estimation equations.

The maximum likelihood estimate of β is obtained as the solution to

$$\frac{m_e}{\beta} - \frac{m_e \tau_e}{\exp(\beta \tau_e) - 1} - \sum_{i=1}^{m_e} \tau_i = 0 \qquad \textbf{(A.15)}$$

and then ν_0 is given by

$$\nu_0 = \frac{m_e}{1 - \exp[-\beta \tau_e]} \qquad \textbf{(A.16)}$$

rounded to the nearest integer.

Steps.

A. Upon the occurrence of each software failure, the failure identification personnel record the cumulative execution time, in CPU seconds since the start of system testing. The execution time can be obtained from the operating system's accounting facility, or the program can be instrumented to provide this information. Collect these failure times, store in a table, and denote the ordered failure times using equation **(A.14)**

B. To assess the current failure rate and reliability of the software, follow these steps:

i. Record the current cumulative execution time, in CPU seconds since the start of system testing. Denote this time τ_e.

ii. Record the cumulative number of failures that have occurred since the start of system testing. Denote this count m_e.

iii. Using the knowns: m_e, τ_e, and **(A.15)**, solve for the unknown parameter β using equation **(A.7)**

The equation is best solved using a root-finding procedure on a computer or programmable calculator. The remaining parameter of the model, ν_0, is found by substituting β into equation **(A.16)** and rounding ν_0 to the nearest integer.

iv. With the point estimates obtained for β and ν_0, the failure rate of the software is given by equation **(5.2)** and the reliability function by using equation **(A.6)** where τ' is execution time measured from the present.

Example:
In this example, there are seven software failures, so $m_e = 7$. The current cumulative execution time is $t_e = 445$. The software failure times are presented in Table A-7.

TABLE A-7. Example Failure Times

FAILURE NUMBER i	CPU SECONDS t_i
1	5
2	35
3	144
4	229
5	342
6	353
7	441

To estimate the parameter β, the following equation is solved:

$$\frac{m_e}{\beta} - \frac{m_e \tau_e}{\exp[\beta \tau_e] - 1} - \sum_{i=1}^{m_e} \tau_i = 0$$

The sum term is

$$\sum_{i=1}^{7} \tau_i = (5 + 35 + 144 + 229 + 342 + 353 + 441) = 1549$$

yielding

$$\frac{7}{\beta} - \frac{(7)(445)}{\exp[\beta(445)] - 1} - 1549 = 0$$

The solution β **=7.3x10^{-5}** is obtained. Several methods for solving equations can be found in any text on numerical analysis. The solution here was obtained using the method of "bisection," which involves repeatedly halving the interval containing the root of the relevant function. The value for ν_0 is found as

$$\nu_0 = \frac{7}{1 - \exp[-(7.35x\,10^{-5})(445)]} \approx 217.54$$

which, rounded to the nearest integer, gives ν_0 **= 218**.

The failure rate of the software is obtained as

$$\lambda(\tau_e) = \beta \nu_0 \exp[-\beta \tau_e]$$

$$= (7.3x\,10^{-5})(218)\exp[-(7.3x\,10^{-5})(218)] \approx 0.016$$

The reliability function is obtained as

$$R(\tau') = \exp[-\lambda\tau'] = \exp[-0.016\tau']$$

Confidence intervals.

The degree of uncertainty present in the point estimates of β and ν_0 can be expressed through the use of confidence intervals. To determine confidence intervals, compute the "Fisher information"

$$I(\beta) = m_e\{ 1/\beta^2 - (\tau^2_e \exp(\beta\tau_e))/([\exp(\beta\tau_e) - 1)^2) \quad \textbf{(A.17)}$$

Then a 100(1-α)% confidence interval for β is

$$\beta \pm \frac{\kappa_{1-\alpha/2}}{\sqrt{I(\beta)}} \quad \textbf{(A.18)}$$

where $\kappa_{1-\alpha/2}$ is the corresponding normal deviate. To obtain a 100(1-α)% confidence interval for ν_0, the high and low confidence limits for β are substituted into the equation for ν_0.

Steps.

A. Compute the "Fisher information" from equation **(A.17)**

B. Choose a confidence level, α. Find the corresponding normal deviate $\kappa_{1-\alpha/2}$. Table A-8 shows the normal deviate values for selected values of α. More-extensive tables can be found in many statistics textbooks.

TABLE A-8. Normal Deviates

α	$\kappa_{1-\alpha/2}$
0.001	3.29
0.002	3.09
0.01	2.58
0.02	2.33
0.05	1.96
0.10	1.64
0.20	1.28

C. To find the lower limit of a 100(1-α)% confidence interval for β, substitute the point estimate for β into the formula

$$\beta_{low} = \beta - \frac{\kappa_{1-\alpha/2}}{\sqrt{I(\beta)}}$$

To find the upper limit, use

$$\beta_{high} = \beta + \frac{\kappa_{1-\alpha/2}}{\sqrt{I(\beta)}}$$

To find the same confidence interval for ν_0, substitute β_{low} and then β_{high} into

$$\nu_0 = \frac{m_e}{1 - \exp[\beta \tau_e]} \qquad \textbf{(A.19)}$$

Example:
Suppose that there were m_e = 19 software failures during the interval through time τ_e = 150.0 and that β was estimated to be .04. Then, the "Fisher information" is computed from **(A.17).**

For a 95% confidence interval, α = 0.05 and $\kappa_{1-\alpha/2}$ = 1.96. Thus

$$\beta_{low} = \beta - \frac{\kappa_{1-\alpha/2}}{\sqrt{I(\beta)}}$$
$$= 0.04 - 1.96 / \sqrt{10810.06} \approx 0.02$$

$$\beta_{high} = \beta + \frac{\kappa_{1-\alpha/2}}{\sqrt{I(\beta)}}$$
$$= 0.04 + 1.96 / \sqrt{10810.06} \approx 0.06$$

Grouped data.
Sometimes it is more convenient to work with the number of failures that occurred over execution time intervals rather than with failure times. Suppose the failure data is grouped into z intervals, with interval i ending at cumulative execution time x_i. The duration of interval i is then x_i-x_{i-1}, with x_0 = 0. Let the number of failures in interval i be denoted y_i' and the cumulative number of failures through interval i be denoted y_i. The total test time is x_z, and the cumulative number of failures for the test is y_z.

The maximum likelihood estimate for β is given by solving the following equation for β:

$$\sum_{\ell=1}^{z} \frac{y_{\ell'}(x_\ell \exp[-\beta x_\ell] - x_{\ell-1} \exp[-\beta x_{\ell-1}])}{\exp[-\beta x_{\ell-1}] - \exp[-\beta x_\ell]} - \frac{y_z x_z}{\exp[\beta x_z] - 1} = 0 \qquad \textbf{(A.20)}$$

The maximum likelihood estimator for ν_0 is then given by

$$\nu_0 = \frac{y_z}{1 - \exp[-\beta x_z]} \qquad \textbf{(A.21)}$$

rounded to the nearest integer. In this case

$$I(\beta) = \frac{y_z x_z^2 \exp[\beta x_z]}{(\exp[\beta x_z] - 1)^2} - \sum_{\ell=1}^{z} \frac{y_{\ell'}(x_\ell - x_{\ell-1})^2 \exp[-\beta(x_\ell + x_{\ell-1})]}{(\exp[-\beta x_{\ell-1}] - \exp[-\beta x_\ell])^2} \qquad \text{(A.22)}$$

This can be used in the same way as for ungrouped failures to construct confidence intervals for β and ν_0.

Steps.
A. Divide the execution time since the start of system testing into p intervals. Denote the cumulative execution time at the end of the i-th interval as x_i.

B. Count the number of software failures that occurred in each interval i. Denote the count for interval i as y_i'. Denote the cumulative number of failures through interval i as y_i.

C. Using the knowns--z;

$$\begin{aligned} &y_1,\ y_2, \ldots,\ y_z; \\ &y_{1'},\ y_{2'}, \ldots,\ y_{z'}; \\ &x_1,\ x_2, \ldots,\ x_z \end{aligned} \qquad \textbf{(A.23)}$$

Solve for the unknown parameter β using equation **(A.20).** To find the value of the parameter ν_0, substitute the estimate found for β into equations **(A.21)** and **(A.22).**

Example:
In this example, there are four intervals, so z = 4. The total test time is $x_p = 55$. The total number of failures is $y_z = 12$. The failure data appears in Table A-9.

TABLE A-9. Example of Grouped Failures

Interval Number l	Ending Time x_l	Number of Failures y_l'	Cumulative Number of Failures y_l
1	15	4	4
2	25	3	7
3	35	3	10
4	55	2	12

The solution to the equation is β **= 0.022**. The parameter ν_0 is found to be

$$\nu_0 = \frac{y_z}{1 - \exp[-\beta x_z]} = \frac{12}{1 - \exp[-0.022(55)]} \approx 17.1$$

which, rounded to the nearest integer, is 17.

Calendar time modeling. The relationship between cumulative execution time t and cumulative calendar time t is determined from the "calendar time component" of the Basic Execution Time Model. The calendar time component takes into account the constraints involved in applying test and repair resources to the software development project. The rate of testing is constrained by failure identification personnel (test team), failure resolution personnel (debuggers), and available computer time. Due to long lead times for training and computer procurement, the model assumes that the quantities of those resources are constant throughout the system test period.

The subscript r is an index that indicates the particular resource involved:

I = failure identification personnel
F = failure resolution personnel
C = computer time

Let θ_r be the amount of resource r required per unit of execution time, and let μ_r be the amount of resource r required per failure experienced. Note that $\theta_F=0$ since failure resolution personnel only address failures. The expected resource requirement χ_r is

$$\chi_r = \theta_r \tau + \mu_r \mu(\tau) \qquad \textbf{(A.24)}$$

where τ is cumulative execution time. Then the change in resource usage per unit of execution time is

$$\frac{\partial \chi_r}{\partial \tau} = \theta_r + \mu_r \lambda \qquad \textbf{(A.25)}$$

If P_r is the available quantity of resource r that is available and ρ_r is its utilization, then $P_r\rho_r$ represents the effective amount of resource r that is available. Then

$$\frac{d t_r}{d \tau} = \frac{1}{P_r \rho_r} \frac{\partial \chi_r}{\partial \tau} \qquad \textbf{(A.26)}$$

Note that $\rho_I=1$, because failure identification personnel can be fully utilized. At any point in execution time, one resource will be limiting, the one that yields the maximum derivative of calendar time with respect to execution time:

$$\frac{dt}{d\tau} = \overline{\max_{r}} \left(\frac{d t_r}{d \tau} \right) \qquad \textbf{(A.27)}$$

The testing phase can be divided into segments. During each segment, exactly one of the resources C, F, or I will be limiting. Each segment will exhibit its own calendar-to-execution time ratio t/τ. To find the potential transition points (in terms of failure rate values) between resource-limited segments, compute

$$\lambda_{rs} = \frac{P_s \rho_s \theta_r - P_r \rho_r \theta_s}{P_r \rho_r \mu_s - P_s \rho_s \mu_r}, \quad r \neq s \qquad \textbf{(A.28)}$$

for each pair of resources. To find the limiting resource within each segment, calculate

(A.29)

for an arbitrary choice of λ within each segment.

If the boundaries of a resource-limited segment are λ_1 and λ_2, the expected number of failures during the segment is

$$\Delta\mu = \nu_0 \frac{\lambda_1 - \lambda_2}{\lambda_0} \qquad \textbf{(A.30)}$$

while the execution time interval is

$$\Delta\tau = \frac{\nu_0}{\lambda_0} \ln \frac{\lambda_1}{\lambda_0} \qquad \textbf{(A.31)}$$

The calendar time increment during the segment is

$$\Delta t_r = \frac{1}{P_r \rho_r \beta} \left[\theta_r \ln \frac{\lambda_1}{\lambda_2} + \mu_r (\lambda_1 - \lambda_2) \right] \qquad \textbf{(A.32)}$$

Confidence limits are obtained by substituting high and low endpoints of the confidence interval for β.

Typically, three resource-limited segments occur:

> $[0, \lambda_{FI}]$: Failures occur frequently. The limiting resource is failure resolution personnel. Testing has to be stopped to allow the debugging team to catch up.
>
> $[\lambda_{FI}, \lambda_{IC}]$: Intervals between failures lengthen. The test team becomes the bottleneck, as the team can only make test runs and evaluate the results so fast.

$[\lambda_{IC}, \ldots]$: Interfailure times become very long. Only the computing capacity limits how fast testing can be accomplished.

The total increment of calendar time over the three segments is given by

$$\Delta t = \Delta t_F + \Delta t_I + \Delta t_C \qquad \textbf{(A.33)}$$

Steps.

A. For each resource--computer time (C), failure resolution personnel (F), and failure identification personnel (I)--determine resource quantity:

P_I: available identification personnel (man-hours)

P_F: available failure resolution personnel (man-hours)

P_C: available CPU time (CPU hours)

B. For each resource r, determine the utilization fraction ρ_r.

C. Determine the amount of each resource expended per failure:

I_C: computer time (CPU hours per failure)

I_F: failure resolution personnel (man-hours per failure)

I_I: failure identification personnel (man-hours per failure)

D. Determine the amount of each resource expended per CPU hour:

θ_C: computer resource expenditure (=1)

θ_F: failure resolution personnel (=0)

θ_I: failure identification personnel

E. Compute the potential transition points (in terms of failure rate) between resources by applying the following formula with the combinations r=I, s=F; r=I, s=C; and r=F, s=C using equation **(A.28).**

Disregard any λ_{rs} that is negative. Put the potential transition points in descending order. Determine which resource is limited in the interval between each pair of successive transition points by choosing an arbitrary λ in each interval and determining the resource r for which the following expression is maximized:

$$\frac{dt_r}{d\tau} = \frac{1}{P_r \rho_r}(\theta_r + \mu_r \lambda) \qquad \textbf{(A.34)}$$

F. To determine the incremental calendar time, in hours, between two execution times, τ_1 and τ_2, that lie within the same resource-limited period, calculate using equation **(A.33).** r is the limiting resource in that interval. For two points that lie in different intervals, sum the incremental calendar time incurred in each intervening interval.

Example:
Suppose that the calendar time component parameters are

$$\theta_C = 1.5;\ \ \theta_I = 3.0;\ \ \theta_F = 1.0;\ \ \mu_C = 2.0;\ \ \mu_F = 6.5;\ \ \mu_I = 2.0;$$

$$\rho_C = 1.0;\ \ \rho_I = 1.0;\ \ \rho_F = 1.0;$$

$$P_F = 200;\ \ P_I = 300;\ \ P_C = 250$$

Then, from

$$\lambda_{rs} = \frac{P_s \rho_s \theta_r - P_r \rho_r \theta_s}{P_r \rho_r \mu_r - P_s \rho_s \mu_r}, \quad r \neq s$$

the potential transition points are found to be

$$\lambda_{IC} = \frac{(2540)(1.0)(3.0) - (300)(1.0)(1.5)}{(300)(1.0)(2.0) - (250)(1.0)(2.0)} = 3.0$$

$$\lambda_{IF} = \frac{(200)(1.0)(3.0) - (300)(1.0)(1.0)}{(300)(1.0)(6.5) - (200)(1.0)(2.0)} \approx 0.19$$

$$\lambda_{FC} = \frac{(250)(1.0)(1.0) - (200)(1.0)(1.5)}{(200)(1.0)(2.0) - (250)(1.0)(6.5)} \approx 0.04$$

The intervals are thus (3,0.19), (0.19,0.04), and (0.04,0). The next step is to find out which resource is limiting in each of these intervals. For resource I,

$$\frac{dt_I}{d\tau} = \frac{1}{P_I \rho_I}(\theta_I + \mu_I \lambda) = 0.003(3.0 + 2.0\lambda)$$

For resource C,

$$\frac{d\,t_C}{d\tau} = \frac{1}{P_C \rho_C}(\theta_C + \mu_C \lambda) = 0.004(1.5 + 2.0\lambda)$$

And for resource F it gives

$$\frac{d\,t_F}{d\tau} = \frac{1}{P_F \rho_F}(\theta_F + \mu_F \lambda) = 0.005(1.0 + 6.5\lambda)$$

Table A-10 summarizes the results of the calculations.

TABLE A-10. Execution Time Derivatives

Interval	Arbitrary λ	$dt_I / d\tau$	$dt_C / d\tau$	$dt_F / d\tau$
(3.0,0.19)	2.0	0.021	0.022	0.07
(0.19,0.04)	0.1	0.0096	0.0068	0.00825
(0.04,0.0)	0.02	0.00912	0.00616	0.00565

In the first row, the maximum value for $dt_r/d\tau$ is from resource F; for the second and third rows it is from resource I. Therefore, in the interval (3.0,0.19), the limiting resource is failure resolution personnel and, during the interval (0.19,0.04) and the interval (.04,0.0) it is failure identification personnel.

Suppose now that the Basic Execution Time Model parameters are $\beta = 0.001$, and $\nu_0 = 200$. Then the time-dependent failure rate of the software is To find the calendar time increment from $\tau=69$ to t=184, compute the failure rate at those points: $\lambda(69)$»0.101 and $\lambda(184)$»1.006. The failure rate interval (1.006,0.19) is in a failure resolution personnel limited period, and the failure rate interval (0.19,0.101) is in a failure identification personnel limited period. The total calendar time over the failure rate interval (1.006,0.101) is the sum of the increments over the two intervals (1.006,0.19) and (0.19,0.101).

For the first interval, the calendar time increment is given by

$$\Delta t_F = \frac{1}{(200)(1.0)(0.003)}$$

$$x \left[(1.0) \left(\ln \frac{1.006}{0.19} \right) + (6.5)(1.006 - 0.19) \right]$$

$$\approx 11.62$$

For the second interval, the calendar time increment is

$$\Delta t_F = \frac{1}{(1.0)(1.0)(0.003)} \left[(0) \ln \left(\frac{0.19}{0.101} \right) + (6.5)(0.19 - 0.101) \right] \approx 192.83$$

Thus, the total calendar time increment is

$$\Delta t = \Delta t_F + \Delta t_I = 11.62 + 192.83 = 204.45$$

hours.

The recalibration technique.

The parameters β and ν_0 are estimated on the basis of the first (i-1) failures and used to evaluate

$$\lambda(\tau) = \beta \nu_0 \exp[-\beta\tau] \qquad \textbf{(A.35)}$$

The estimated cumulative distribution function (Cdf) is then

$$\hat{F}_i(\tau') = 1 - \exp[-\lambda(\tau_{m_e})\tau'] \qquad \textbf{(A.36)}$$

where τ_{me} is the cumulative execution time to the end of the growth test, $\lambda(\tau_{me})$ is the failure intensity at that time, and τ' is execution time measured from the present. When the i-th failure--and thus the interfailure time τ'_i between the (i-1)st and i-the failure--is later observed, the probability integral transform

$$u_i = \hat{F}_i(\tau_{i'}) \qquad \textbf{(A.37)}$$

is recorded. Each failure results in another u_i. The probability integral transform implies that the u_i's should look like a random sample from a U(0,1) distribution, if the sequence of one-step-ahead predictions was good. The accuracy of the model with respect to the particular program can be gauged by drawing a u-plot. In a u-plot the sample Cdf of the u_i's is visually compared with the Cdf of the uniform distribution over (0,1). Let m be the number of u_i's. To create a u-plot the m u_i's are put in ascending order

$$u_{(1)} \le u_{(2)} \le \ldots \le u_{(m)} \qquad \textbf{(A.38)}$$

and then the points

$$(u_{(1)}, 1/(m+1)),\ (u_{(2)}, 2/(m+1)),\ \ldots, (u_{(m)}, m/(m+1)) \qquad \textbf{(A.39)}$$

are plotted. The line of unit slope (uniform Cdf) is also plotted on the same graph, for comparison.

Furthermore, the u-plot can be employed to recalibrate the software reliability model. The recalibrated model corrects systematic bias or noisiness that the model is experiencing when being used on a particular program. The recalibration takes place by applying a function $G^*(\times)$ to the estimated Cdf. The function $G^*(\times)$ is expressed as

$$G_i^*[\hat{F}_i(\tau')] = \frac{\hat{F}_i(\tau') + j[u_{(j+1)} + u_{(j)}] - u_{(j)}}{(m+1)[u_{(j+1)} + u_{(j)}]}, \quad u_{(j)} \le \hat{F}_i(\tau') \le u_{(j+1)} \qquad \textbf{(A.40)}$$

where m is the number of u_i's and, for convenience, $u_{(0)} \equiv 0$ and $u_{(m+1)} \equiv 1$.

To perform the recalibration the user applies the transformation

$$\hat{F}_i^*(\tau') = G_i^*[\hat{F}_i(\tau')] \qquad \textbf{(A.41)}$$

Steps.

A. Upon the i-th software failure, use the Basic Execution Time Model to estimate the failure rate based on the software failures

$$\tau_1,\ \tau_2,\ \ldots,\ \tau_i \qquad \textbf{(A.42)}$$

The estimated cumulative distribution function is found using formula **(A.36).**

When failure number (i+1) occurs, record

$$u_i = \hat{F}_i(\tau_{i+1} - \tau_i) \qquad \textbf{(A.43)}$$

B. To form a u-plot, put the sequence $\{u_i\}$ into ascending order. Denote the ordered u values using formula **(A.37)**

C. Plot the points using formulas **(A.38).** If the points mostly lie above the line, the model is producing optimistic estimates of the failure rate. If the points mostly lie below the line, the model is producing pessimistic estimates. If the points are spread out both above and below the line, the problem is noisiness. The bias or noisiness can be corrected by recalibration.

D. Correct the value cumulative distribution function by substituting it into formula **(A.40)** to obtain a recalibrated value.

Example:
The estimated failure rate after the m software failures is $\lambda = 0.27$. The estimated cumulative distribution function is thus

$$\hat{F}_7(\tau') = 1 - \exp[-0.27\tau']$$

Suppose further that the ordered u sequence is

$$u_{(1)} = 0.03, u_{(2)} = 0.06, u_{(3)} = 0.12,$$
$$u_{(4)} = 0.59, u_{(5)} = 0.8, u_{(6)} = 0.86,$$
$$u_{(7)} = 0.92$$

It is desired to recalibrate

$$\hat{F}_7(6.35) = 0.82$$

Since

$$u_{(5)} \leq 0.82 \leq u_{(6)}$$

the recalibrated value is

$$\frac{0.82 + 5(0.86 + 0.8) - 0.8}{7(0.8 + 0.86)} \approx 0.72$$

Since

$$\hat{R}_i(\tau') \text{ equiv } 1 - \hat{F}_i(\tau')$$

the reliability is estimated to be 1-0.72=0.28.

A.10 Bibliography.

MIL-STD-721 Definition of Terms for Reliability and Maintainability, 12 June 1981

MIL-STD-756 Reliability Modeling and Prediction, 18 November 1981

MIL-STD-781 Reliability Testing for Engineering Development, Qualification and Production, 17 October 1987

MIL-STD-1521 Technical Reviews and Audits for Systems, Equipments, and Computer Software, 4 June 1985

MIL-STD-2155 Failure Reporting Analysis and Corrective Action System, 24 July 1985

MIL-HDBK-189 Reliability Growth Management, 13 February 1981

MIL-HDBK-217 Reliability Prediction of Electronic Equipment, 2 December 1991

MIL-HDBK-781 Reliability Test Methods, Plans, and Environments for Engineering Development, Qualification, and Production, 14 July 1987.

DOD-STD-2167 Defense System Software Development, 29 February 1988

MIL-STD-973 Configuration Management, 24 November 1993

R-013-1992 "Recommended Practice Software Reliability" ANSI/AIAA R-013-1992

NSWC TR-82-171 "A Survey of Software Reliability Modelling and Estimation" by Dr. William H. Farr, Naval Surface Weapons Center NSWC TR 82-171, Dahlgren, VA.,1983.